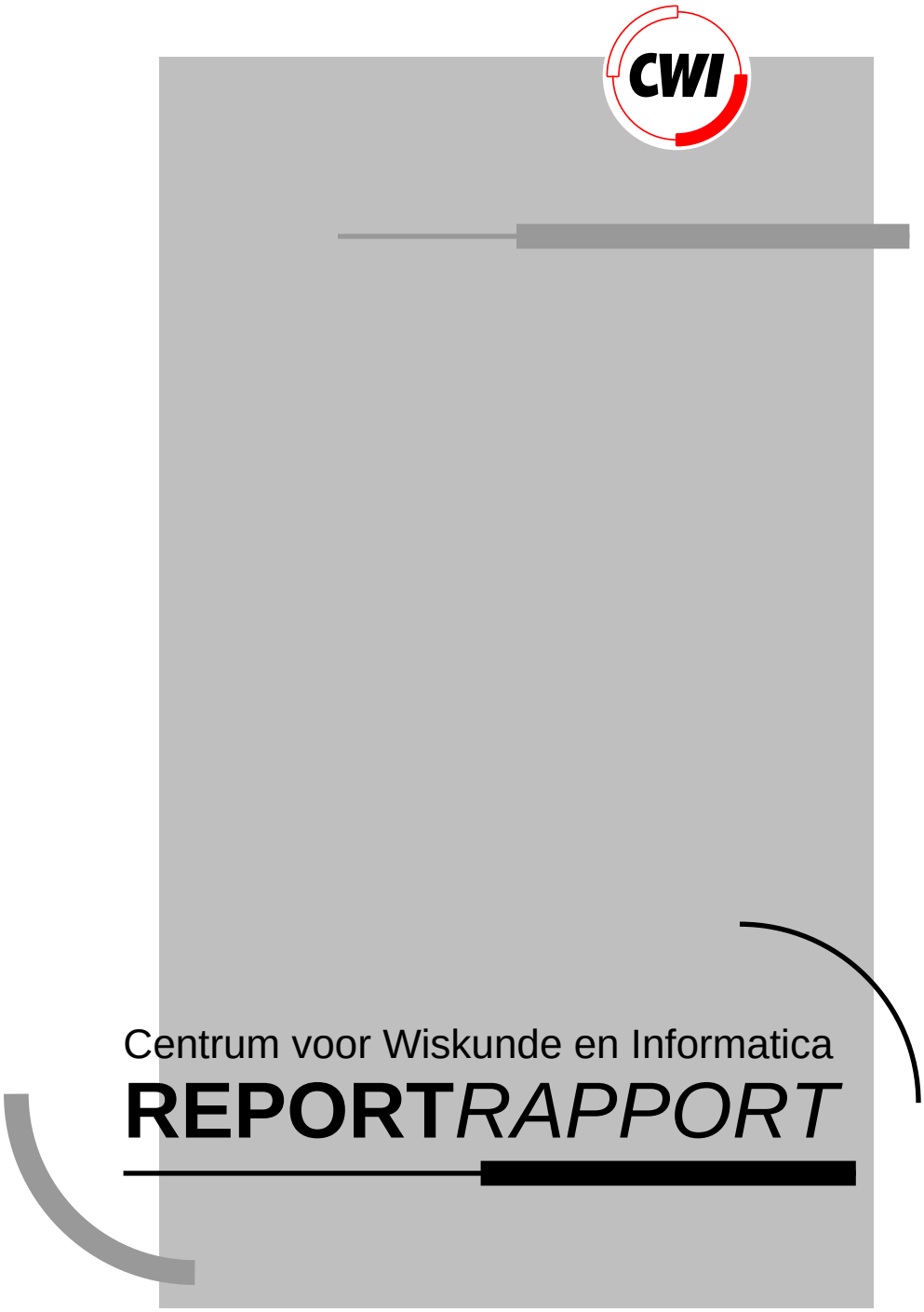

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

The Goblin database programming language

M.L. Kersten, C.A. van den Berg, A.P.J.M. Siebes, C.J.E. Thieme, M.H. van der Voort

Computer Science/Department of Algorithmics and Architecture

CS-R9407 1994

The Goblin Database Programming Language

M.L. Kersten, C. van den Berg, A.P.J.M. Siebes,
C. Thieme, M.H. van der Voort

CWI
P.O. Box 94079, 1090 GB, Amsterdam, The Netherlands
{mk,carel,arno,ct,leonie}@cwi.nl

Abstract

Goblin is a database programming language for application development and ad-hoc querying an object-oriented database. Its salient features include: a strong and extensible type system, automatic object classification, a core of imperative programming language concepts, declarative database querying, a trigger mechanism, and transaction management.

The language design is strongly influenced by several envisioned application domains, such as office automation, robotic applications, cartographic applications, and astronomy. These application domains are surveyed and their requirements on a database programming language identified.

CR Subject Classification:
H.2.1 [**Information Systems**]: Database Management- *Logical Design*,
H.2.3 [**Information Systems**]: Database Management- *Languages*,
H.2.4 [**Information Systems**]: Database Management- *Systems*,
H.2.8 [**Information Systems**]: Database Management- *Database Applications*,
H.3 [**Information Systems**]: Information Storage and Retrieval.
Keywords and phrases: database programming languages, triggers, object-oriented database systems, extensible database systems.
Note: This work was partly supported by NFI project NFI-74 and SION project 612-317-025.

Contents

1 Introduction

The relational database approach invented over two decades ago has found wide acceptance in the traditional, business kind of information systems, because it improves the flexibility of database management and it greatly improves the productivity through its 4GL tools. In particular, the optimization of database access and automatic enforcement of database integrity has proved valuable to attain these gains. The necessary information for an adequate database management job is extracted with SQL, a standardized [SQL] database query language, which provides an easy to learn and uniform interface to database administrators, application programmers, and end-users. As a result, SQL has become a standard language for manipulating data in both centralized/distributed and homogeneous/heterogeneous environments.

Despite the success of the relational model in the commercial arena, many limitations of the relational model have been pointed out when applied to non-traditional information systems, such as Office Automation, Geographical Information Systems, CAD/CAM, etc.. The current awareness is that an enhanced datamodel and a more flexible architecture is needed to cope with the problems posed.

1.1 The post-relational era

One of the first serious attempts to enhance the expressiveness of the relational model is the Non-First Normal-Form (NF^2) model proposed by Schek and Pistor [Schek 82]. They observed that in many applications there is a need for set-valued and list-valued attributes. Consequently, they dropped the (dogmatic) requirement for a relational scheme that each attribute is atomic, i.e. non-decomposable. The result is again a datamodel for hierarchical structured objects with a declarative query language.
This approach has been generalized in [Pistor 86] into Extended NF^2, where all restrictions placed on the type constructors have been dropped. Moreover, an SQL-like query language is defined to recap the learning phase of the last decade. Other research activities in this area are described in [Härder 87] [Batory 86]. The formal semantics of NF^2 models are a central research issue in database theory, such as in [Schek 86, Roth 88].

The main limitation of the NF^2 approach is its focus on the structuring aspects, while largely ignoring the behavioral aspects of composite objects. These behavioral aspects form the focus of object-oriented databases with the salient features of abstract data types, object identity, type inheritance, and persistency independence [Atkinson89, Bancilhon88]. Abstract datatypes enable the encapsulation of data and their methods, while completely hiding their implementation details. Object identity associates a unique identifier with each object stored. This identifier can thus be used to model object sharing and thus to build graph-like objects. Inheritance enables objects of different types to share part of their behavior, i.e. their methods. This encourages re-use of design and code. Finally, persistency supports the survival of an object in the face of system malfunctioning.

An alternative track is to focus on the inclusion of abstract datatypes within the relational framework. The hypothesis here is that the collection of built-in types is often insufficient for advanced applications. Thus, one should open up the DBMS architecture to permit the inclusion of user-defined types, written in a convenient implementation language, such as C/C++. Exemplar research projects in this area are the extensible database architectures of [Gardarin 88, Batory 84, Carey 86].

Object-oriented database systems are strongly influenced by object-oriented programming concepts [Goldberg 83, America87, Stroustrup]. Therefore, many analogies can be drawn between programming concepts and database concepts as surveyed in [Dearle89][Matthes 89]. Moreover, some researchers have taken the route to enhance an existing object-oriented programming environment to arrive at an object-oriented DBMS [Maier 84, Agrawal89]. The prime issues to be addressed are support for persistency, concurrency control, recovery, and declarative query formulations.

Although many recent DBPL papers [Dearle89, Agrawal89, LeCluse, Matthes 89, Atkinson81] aim at these potential markets, few have taken the requirements imposed by them as a driving force

for language design. To partially alleviate this situation, we believe that the application domain characteristics sketched in this paper have a significant impact on the syntax and semantics of a general purpose DBPL, such as:

- *union types and type discrimination*, to deal effectively with heterogeneous data structures,
- *classification*, to (automatically) organize objects by structure, behavior, and constraints,
- *triggers*, to model semi-autonomous processes or active objects,
- *Abstract Data Types*, to extend the type system and the compiler/optimizer.

We have taken the route to develop a database programming language, called Goblin, as an intermediate step towards a domain specific DBMS, i.e. with a domain specific data model, query language, and user interface. Its design has produced a balanced solution between common practice (and expertise) to write applications in a mixture of C (++) and SQL-like language concepts and the need for a new full-fledged high-level programming language to obtain a seamless interface to the database management system. The rationale is that a database programming language should not force users to convert existing software. Instead, it should attract users by providing a more convenient programming platform with an open interface to reuse existing software. A design track also followed in the O_2 project [Deux 90].

In the design process of Goblin, we have used the application characteristics described below as a yardstick to identify the essential language features, to exercise language expressiveness, and to select convenient notation.

1.2 The Role of Application Domains

As mentioned above, the main thrust of research vested in post-relational systems aims at supporting novel application domains, such as office automation, geographic,robotics, CAD/CAM, and astronomy. Although many papers re-iterate these potential market for the new-generation database systems, few have identified the predominate requirements imposed by all of them together on the design of a DBPL syntax and semantics.

To partially alleviate this situation, we believe that the characteristics sketched below significantly influence both the database programming language features and the data model, query language, and architecture of its DBMS kernel. These application characteristics have been used as a yardstick in the design of Goblin to test language expressiveness and to select convenient syntax shortcuts.

1.2.1 Office Automation

Database management in office automation concerns the capturing, archiving, and dissemination of documents for decision making purposes [Gibbs 85], [Tsichritzis 85]. These document databases are large collections of semi-structured objects, that range from simple bit maps, ASCII strings, and letters, to highly regular data entry forms and EDI messages. Furthermore, collections of documents are structured in two (independent) ways: a hypertext structure and folders. The hypertext structure is caused by the (unpredictable) way office documents refer to one another, while folders emerge from (unpredictable) aggregation of heterogeneous document structures.

From a DBPL point of view, this application domain requires a rich type system to support document structuring, a flexible reference mechanism to manage hypertext structures, and objects that can hold values of different types. We will elaborate a little on these characteristics.

The first issue deals with the role of typing schemes in such office applications. In principle, a typing scheme can be used to describe the classification hierarchy of office documents [Barbic 85]. For, each document can be seen as a hierarchical aggregation of components, which can be described with tuple types. However, since document structures differ widely, this leads to large type descriptions.

For example, a common business letter may contain a 'your ref' and 'our ref' section, the secretary's name, the topic, a telephone extension, a logo, a bank account footnote, etc.. These properties differ per letter, per business market, and per country. Catching these differences with a single hierarchical tuple type that includes all properties yields an unwieldy structure, because many of the attributes will be set to NIL and inappropriate attributes can not easily be distinguished from missing values.

Furthermore, the structure of a document is not completely described [Rabitti 86, Meghini 87]. An office worker tends to classify a document by the major structuring criteria, leaving the detailed classification for the future, as need arises. This means that at any level in the document type hierarchy we have to permit for a non-interpreted portion. This calls for a union type constructor where one alternative permits typing a document portion as a reference to another representation, such as its bit-map or ASCII string.

The second issue stems from the liberal use of references inside office documents. They appear both in specific document areas, e.g. an 'our ref' section, as well as in their content part, e.g. 'in answer to your letter dd. 12-4'. Furthermore, the format ranges from document identifiers, 'contract nr 97-33-77' to hints 'our meeting last week'. This broad spectrum of formats and their embedding in documents requires both union types and an extensible coercion mechanism. For example, a contract number can be coerced to a document identifier regardless its format.

The third issue stems from office practice to physically group documents into folders to simplify reference, querying, and message handling. The content of these folders is based on a simple predicate, such as a client name, and it contains different document styles, i.e. memos, EDI messages, letters, and forms. Thus, an office folder assembles items from anywhere in the database, which severely complicates type checking and storage optimization. At the language level this leads to classes over union types and a materialized view concept, which allow a programmer to (declaratively) describe object grouping.

Although querying a document base may start with a folder, subsequent selection is often based on presenting a set of <property, value-constraints> pairs. For example, find all the documents in the folder named Competition that refer to the product Milk Soap. Such queries stress the functionality of a query processor, because it should first locate all objects in the folder that have a product attribute and, thereafter, select those that satisfy the restriction.

The requirements posed by this area on a DBPL can be summarized as follows:

- It should support heterogeneous data structures,
- it should support a cross document reference mechanism,
- it should support (declarative) aggregation of objects, and
- it should support querying the database both by physical aggregation and by predicates over individual document properties.

1.2.2 Spatial Databases

Spatial database applications, such as geography and mechanical CAD, illustrate another dimension in advanced database applications. They deal with the representation of n-dimensional data with explicit knowledge about objects, their extent, and their position in space. The basic data structures and algorithms required in this area do not neatly match the relational model. Instead, it requires handling of multiple spatial representations and data models, access methods and optimizations [Guenther 90].

The primitive types dealt with include points, polygons, edges, etc.. The variety in semantics and the availability of many graphic packages hinder inclusion of such types within a DBPL implementation. Instead, one should include an Abstract Data Type facility, which permits the functions and behavior of these types to be described in an abstract and manageable way. Their actual implementation can be borrowed from the environment.
Such ADTs extend the vocabulary of the query languages. The behavior of geometric types,

described by equations, aids query optimization. Provided the compiler includes a rule driven optimizer as researched in [Güting 89]

The last issues stems form the experience that no single spatial representation suites all purposes in a given application. Instead, objects are often represented in different ways, for example a constructive solid geometry representation (CSG) versus a boundary representation (BR). Such applications would benefit from a language mechanism, such as triggers and constraints, that permits concise description of propagation of updates on either representation.

Therefore, the additional requirements posed upon a DBPL are:

- it should support ADTs for interfacing and query optimization,
- it should support management of replicated data structures with different representations.

1.2.3 Computer Integrated Manufacturing

In Computer Integrated Manufacturing (CIM) a distinction is made between the design of a production environment, i.e. the off-line environment, and the completed, running system, i.e. the on-line environment [Camarinha 87, Freedman 89]. The off-line environment is a highly interactive process, where a designer is focussed on finding an optimal solution for product assemblages. This involves choosing robots and sensor systems, defining the cell layout, and producing a task description for the robot systems. Once an initial design is made, the designer experiments to find defects and to improve the design.

The online environment can be characterized as a semi-autonomous system, where robot tasks have been fixed up front and the interaction with the image recognition system is well-defined and stable. That is, the focus in the on-line case shifts towards scheduling processing tasks under hard deadlines.

At the language level it implies, amongst others, support for the definitions of semi-autonomous tasks, support for hypothetical queries and a flexible simulation environment, and a (distributed) processing platform. We illustrate these issues briefly.

Due to the nature of CIM systems, their designs are extensively tested before being taken into production. This involves both the physical design of the production cell and the programming of the robot tasks. The former requires facilities for geometric modelling, as described above. The robot tasks can be seen as threads of primitive actions, such as picking a component and moving it into place, which are triggered by scene analysis, such as the recognition of a new frame arrival. This calls for active database support, such as provided by *triggers* (active objects, actors, etc.). They allow a robot task designer to express the partial order among the tasks in terms of observed database states and events.

Once the robot tasks have been prototyped, the production cell will be simulated against a mock up state. In essence, this amounts to producing derivable database states that reflects the reachable cell states. The simulation analysis involves statistical aggregation of measurement data, such as the average production time of a piece, and the recognition of critical intermediate states, such as two robot arms in deadly embrace awaiting the other to leave a 3-D subspace. The latter area also includes hypothetical queries or 'what if' queries, such as testing the robot recovery procedures by presenting a dangerous situation.

The on-line CIM situation, i.e. the production cell is characterized as a distributed processing environment with autonomous sensory nodes, robot control, and component transport. A database system managing this environment is faced with hard deadlines, such as handling an event within 30 ms.. Yet, the fixed set of actions provide ample opportunities for extensive compiler-based optimizations.

Therefore, the development aspects of the CIM design phase imposes the following requirements on a DBPL:

- it should support a trigger concept to model semi-autonomous actions,
- it should support an incremental programming style,

- it should support flexible transaction semantics and synchronization primitives for specifying cooperative behavior.

1.2.4 VLSI Design

VLSI design has been an early focus of advanced database applications. It deals with a precisely defined and structured domain of chip components and their layout on the wavers. Furthermore, the algorithms that exercise a chip design are highly computation intensive. The data structures managed are spatial in nature. However, as pointed out in [Härder 87], the view on the geometry differs with each query problem being addressed.

For example, a chip design consists of several information layers; the wiring structure, the cell structure, and the functional structure. The wiring level can be viewed as being composed of faces built out of edges and edges built out of points. However, this hierarchical view may not be adequate for an analysis of electrical interference, which would require the chip to structured as junctions (points) connecting wires (edges).

In both situations we have a single geometric structure, but the ways in which this structure is being navigated differs. In terms of database concepts it means that alternative (navigational) views can be defined and at the implementation level that references between structures can always be reversed. The additional requirements posed on a DBPL become:

- it supports redefinition of navigational paths;
- it supports bi-directional access over object references;
- it should support efficient manipulation of large graph-like data structures.

1.2.5 Astronomy

Astronomy illustrates an extreme case of scientific databases. It deals with capturing, archiving, and analysis of information gathered from extra-terrestrial objects and space missions. Its databases are often huge (100-6000 Gigabyte) [Green 90] and they contain a variety of data items, ranging from radio- and telescope images, textual descriptions of observations, to highly condensed structural information, such as space catalogues.

From a DBPL viewpoint, its requirements extend those above by support for information retrieval, approximate query facilities, and statistical summaries.

A significant difference with previous domains is that the databases in astronomy have a tendency to accumulate information. Raw data obtained from observations is calibrated, summarized, and interpreted in scientific reports. Integration of all this information within the database is necessary to support the researcher. For example, having access to the raw observatory data about an object is as important as the related scientific reports. Thus, a DBPL for this environment should be prepared to accumulate a large type catalogue and be an open language to access remote resources.

Another complicating aspect is the querying mode encountered in scientific databases. A researcher often cannot identify the data of interest with a single request. Instead, he will browse the database by posing approximate or classification queries until the portion of interest is located; the working mode is *browsing* and *data dredging* within the context of a general classification framework. The isolated portion is often used for a lengthy period, which calls for a materialized view, and statistical summaries are constantly being added.

To summarize, the additional requirements posed on the design of a general purpose DBPL are as follows:

- it should support an evolutionary type and classification scheme,
- it should support information retrieval over multi-media types,
- it should support approximate querying and statistical grouping,
- it should support huge (read only) databases.

1.3 Design objectives

This snap shot of requirements posed by advanced application domains suggests that no DBMS is likely to evolve in the next few years to solve all problems at once. In fact, the wish list can be easily extended to arrive at a DBMS research program for the next decade [SIGMOD 90]. Instead, we have taken the route to develop a database programming language (DBPL) as an intermediate step towards a domain specific DBMS, i.e. with a domain specific data model, query language, and user interface. In its design we aim for a balanced solution between common practice (and expertise) to write applications in a mixture of C and SQL and the need for a full-fledged high-level programming language to obtain a seamless interface to the database management system. To summarize, the predominant language design issues produced by this course of action are:

- *Extensible Type System*
  Goblin supports an Abstract Data Type facility, that specifies the signatures and the equations of external types. The rationale is that a DBPL should provide a well-defined interface with an existing, possibly unsafe programming environment.

- *Type Checking*
  Goblin is a strongly typed language. Most type checking is done at compile time. However, it can not ensure correctness of all expressions that involve types over finite domains, such as array bounds, and union types. In those cases where information at compile time is not available it resorts to run time checks.

- *Classification System*
  Goblin supports values and objects with a clear separation of their role. A class provides a (semi) automatic classification scheme for objects based on their attribute value. Variables hold unclassified data values or references to classified objects. The rationale is that a flexible classification scheme leads to a concise and modular description of the database content and its behavior.

- *Triggers*
  Goblin supports triggers to model semi-autonomous actions. The rationale is that triggers are a convenient concept for capturing guarded processes in control applications.

- *C Compliance*
  The Goblin syntax and language facilities have a strong C-flavor, such that programmers may find it easy to switch. Dangerous concepts for distribution and parallel processing, such as address arithmetic and address manipulation, have been removed.

- *Database querying*
  Goblin supports declarative database queries using a SQL-like filter. The rationale is to re-cap learning SQL and to enable optimization decisions to be taken easily. A separate SQL-compatible interface is foreseen.

- *Parallelism*
  Goblin stimulates a programming style that supports coarse-grain parallel processing for bulk data types. The rationale is that effectiveness of database technology comes from optimizing independent actions over collections. It is this order independence that permits parallel processing without explicit action on the programmer's part and which has been studied in the database research community extensively.

The remainder of this document illustrates the Goblin syntax and its informal semantics. It provide a reference point after preliminary Goblin publications [Kersten 90, Kersten 91] and to provide a basis for in-depth studies into several language design aspects and to write real applications rapidly. We believe that desk-top analysis and subsequent formalization of language concepts should go hand-in-hand with practical experience to avoid imprecise and useless language features. For the remainder we emphasize the practical aspects and we assume a general background on

the semantics of common constructs in high-level programming languages, such as typing, parameter passing, and statements. Studies on related novel query processing architectures[vdBerg 91], adaptive storage techniques, trigger system formalization[vdVoort 91, Siebes 91], and database design theory are underway.

Section 2 and 3 give a synopsis of the Goblin type system, variables, and values. Section 4 provides an overview of the function abstractions being offered. Section 5 and 6 reflect on our notion of classes, and in Section 7 on its inheritance scheme. Section 8 describes the data manipulation facilities, including the database query language concepts. The trigger model is introduced in Section 9. Section 10 describes the core transaction management facilities and mechanisms to refine them according to the domain requirements. The extensibility aspects, such as abstract data types and external libraries, are introduced in Section 11. We conclude with a project status and future work plan.

2 Data Types

The core of Goblin consists of a type system common among database programming languages, such as PS-Algol [Atkinson81], Napier88 [Dearle89], and DBPL [Matthes 89]. These languages have their roots in the class of imperative programming languages and we assume a background in their predominant language concepts.

The rationale for choosing an imperative programming language is one of historical stubbornness and preference to provide an evolutionary route for the majority of programmers. This choice complicates the mathematical foundation of the language semantics at a later stage, but should not in itself form an a priori barrier in explorative language research.

In this section, we focus on those Goblin features that simplify the description and the analysis of a large type schema, such as needed for Office Information Systems. This application area requires a concise description of collections of heterogeneous values, which leads to tuple/ union type expressions with powerful operators to inspect the type of an instance at run time.

2.1 Basic Type Constructions

The basis of the type system is formed by atomic data types: BOOL, SHORT, INT, FLOAT, DOUBLE, CHAR, and STR. They are built-in abstract data types where the user is unaware of (or cannot rely on) implementation details. This collection can be extended using an ADT definition (See Section 11.2). All atomic types contain a null element, denoted by NIL. The least and most restrictive types are denoted by ANY and VOID, which are used to write polymorphic routines and to indicate type less results, respectively.

In addition, Goblin offers the conventional type constructors, such as: tuple, union, array, list, bag, and set. The constructors array, list, bag, and set are referred to as *collections*, i.e. a structuring mechanism for bulk data types with built-in operators to access their elements (See Section 4.4). The bounds of an array denote the number of accessible elements with indexing starting at zero. The initial value of all its elements is NIL.

Enumerated types can be used to introduce compile time constants. They are defined by an identifier list where their placement index determines the constant integer value. This default value can be overruled with an assignment of a constant expression.

A type definition is recognized by the keyword TYPE preceding the type name and its construction. Using a type name later on in the program is interpreted by copying the construction into place. A few examples illustrate these constructors:

```
TYPE    Name =LIST(CHAR);
TYPE    Vector =FLOAT[5];
TYPE    Text =SET(STR);
TYPE    Date =TUPLE(INT yr,mo,dy);
TYPE    SocialSec =CHAR[8];
```

```
TYPE    Palet = [red,yellow,blue];
```

2.2 Tuple Type Expressions

Tuple types describe aggregate values, whose components are identified with attributes. Attributes are typed and their names should be unique within the scope determined by the type definition. Furthermore, tuple types can be created from existing tuple types by applying a type modifier as follows.

Let T_1 and T_2 be two tuple types then the tuple-type expression $T_1 + T_2$ defines a specialized tuple whose attributes comprise the set union of the attributes of T_1 and T_2. If two attributes have the same name but different underlying types then the result is a single attribute based on the union of their types.

Alternatively, a tuple type can be constructed by projection over an existing tuple type, such as TYPE $T_2 = T_1.(A_1, ..., A_k)$ which introduces the type T_2 by copying a portion of T_1. For large attribute list it pays off to specify the attributes that are not needed from the original. This is indicated by a subtraction operation, which constructs a tuple type using set difference over the attribute sets. That is, TYPE $T_3 = T_1 - T_2$ is equivalent to $T_3 = T_1(attr(T_1) \backslash attr(T_2))$. A single attribute can be removed using its name for T_2.

For example, consider a letter that takes several shapes: job mail, junk mail, love mail, and private mail. The common part `Letter` describes its sender and its content. It is refined in the next statement by addition of a date component. Then, the `JobMail` adds both a receiver and an archive attribute; for junk mail the archive might be subsequently ignored; love letters are presumable addressed to a single person in a given city.

```
TYPE    Letter = TUPLE(Person sender; STR content);
TYPE    Letter += TUPLE(Date d);
TYPE    JobMail = Letter + TUPLE(Person receiver; INT archive);
TYPE    JunkMail = JobMail − archive;
TYPE    LoveMail = Letter + TUPLE(Person receiver; STR city) ;
TYPE    Year = Date(yr);
```

2.3 Union Type Expressions

In our introduction, we pointed out the need for a union type constructor to capture the structure variations encountered in office applications. In Goblin union types are formed by expressions over existing types, i.e. if T_1 and T_2 are two type constructions then $T_1 \mid T_2$ defines a union type whose domain comprises the disjoint set union over its constituents.[1]

The labels to distinguish the union cases in expressions are the type names (if any) used in their construction. Inline definition of constituent types lead to an unlabeled union, which requires a type discriminator expression to differentiate the cases (See Section 2.4).

For example, below we define the notion of a `Memo` using the shorthand notation for tuple types. All mail occurrences are captured by the union type `Mail`, whose operand names can be used as a type discriminator. Finally, the type `Cards` illustrates the definition of an unlabeled union type.

```
TYPE    Memo =(STR department; STR date) ;
TYPE    Mail =Memo | Letter | JobMail | JunkMail | LoveMail;
TYPE    Cards =(STR postCard,city) | (STR valentineCard);
```

2.4 Optional Attributes

Although union and tuple type expressions ease the description of a type covering multiple simple domains, it does not yet lead to a concise description of the document categories encountered in office applications. For, documents are characterized by optional properties, many of which may

[1] The precedence of the operators align with their counterpart in arithmetic expressions.

be irrelevant for a specific instance. To simplify their type definition we have included the notion of *optional* attributes and a *type discriminator* expression to handle variant selection effectively. A small example illustrates our approach.

Consider a person with optionally a telephone (`phone`), a home address (`home`), a social security number (`ss`), and children (`kids`). Here optional means that persons exist for which a property might be inappropriate, not just unknown (NIL). The domain of all persons can then be described concisely in Goblin by the following tuple type:

```
TYPE Person =TUPLE( STR name; INT phone?; STR home?; SocialSec ss?; SET(Person) kids?);
```

which is equivalent to the more verbose unlabeled union type that enumerates all possible combinations:

```
TYPE Person =TUPLE( STR name)
      | TUPLE ( STR name; INT phone)
      | TUPLE ( STR name; INT phone; STR home)
      | TUPLE ( STR name; INT phone; SocialSec ss)
      | TUPLE ( STR name; INT phone; SET(Person) kids)
      | TUPLE ( STR name; STR home)
      | TUPLE ( STR name; STR home; SocialSec ss)
      /* etc */;
```

The union type requires a type analysis predicate to identify the individual cases. For a union type built out of previously defined types their type name can be used to differentiate the cases using the comparison operator ISA. This operator takes a value as the left-hand-side operand and a type expression as the second operand. It holds if indeed the value is of the type indicated. For example, a variable `V` defined over the type INT | FLOAT | BOOL can be analysed with the predicates `V` ISA FLOAT or `V` ISA (INT | FLOAT).

Since the expanded form of tuples with optional attributes is an unlabeled union type, we do not have a single identifier to identify the cases. Instead, we discriminate them using a predicate parameterized with attributes.

For example, a tramp can be described in terms of a person `P` without `home` and `phone` as shown in a predicate below. The terms of a discriminator expression are not limited to the top level tuple structure, any path expression (Section 4.3) is allowed. Thus, the last statement assures that the variable `xyz` has a component called `map` and a composite `address` with `housenr`.

```
P WITH name  &&  ~ P WITH (phone,home);
xyz WITH map  && xyz WITH address.street.housenr
```

2.5 Subtype Relationships

The tuple and union types are organized in a subtype hierarchy [Cardelli 84] based on their attributes (name and type) and union labels, respectively. That is, a tuple type T_1 is a considered a subtype of T_2, denoted by $T_1 \prec T_2$, if the attributes of T_2 form a subset of those of T_1. For union types, the precedence relation between $U_1 \prec U_2$ holds when $domain(U_1) \subset domain(U_2)$. The subtype hierarchy extends to the collection constructions. That is, let k denote a collection constructor then $k(T_1) \prec k(T_2)$ if $T_1 \prec T_2$.

A few examples suffice to illustrate subtype relationship for the types defined so far:

`JobMail` $\prec$ `Letter`

SET `(Memo)` $\prec$ SET `(Mail)`

INT | FLOAT $\prec$ INT | FLOAT | STR

(INT `i,j`) $\prec$ (INT `i,j?`)

3 Scopes, Variables, and Values

3.1 Lexical Scopes

In Goblin a name denotes a TYPE, variable, CLASS, function, TRIGGER or ADT. A name is introduced by a definition or declaration and it can only be used in a region of program text called its scope. We distinguish two kinds of scopes: local and global.

A name introduced in a statement block is local to that block; it can only be used after the declaration block and in blocks enclosed. Attribute names are treated as if they are declared in the corresponding tuple constructor. The formal function arguments are also treated as variables declared within the outer most enclosing block of the function body.

A name is called global if it is associated with a definition or a declaration at the outermost program level. In principle, all global names can be accessed by other Goblin programs through compile time directives.

A name may be hidden by an explicit re-declaration of that same name in an enclosing block. A hidden enumerated value can still be used through type casting. For example, consider the definition of `palet` before and a code fragment that hides the enumerated value `red`. Then the outer definition can be obtained with the expression `palet(red)`.

3.2 Variable Declarations

Goblin uses a conventional two-level name space for variables within a program. The inner level consists of local variables to hold temporary results within function, trigger, tuple construction, and statement blocks. The outer level name space consists of global variables. They persist throughout the life-time of the corresponding Goblin database.

The declaration syntax is based on the C++ convention, which consists of the type construction followed by a list of variable names. Moreover, variable declarations can appear anywhere in a statement block. Their initial binding is the value NIL, which can be overruled with an assignment statement. Furthermore, a variable can be turned into a write-once-read-many times storage structure by adding the keyword CONST to its definition. Assignment to a CONST variable can either take place at compile time or in a user controlled initialization phase at run time.

```
Person tramp;
SET(Person) group;
CONST FLOAT pi= 3.147;
```

3.3 Value Denotations

The denotation for atomic types and collections follows well-know conventions. The atomic, ADT, collection, and tuple types come with a predefined constructor function. It takes a fixed number of arguments, whose order and their type correspond with a sequential interpretation of the corresponding type definition. The value constructor function derived from a union type takes the shape determined by its constituents, provided the union case can be determined uniquely.

For example, in the union types declared before `Mail( Memo('Toys','31/1/92'))` denotes a non-ambiguous letter value, while `Mail('Secret')` causes a compile error, because it ambiguously denotes a `postCard` or a `Memo`. A person value with an unknown properties can be created by calling `Person('John Dough', ?, 'Graveyard', ?, ?)`.

Values for Abstract Data Types are also identified by their constructor function with (literal) arguments of the proper type. It is up to their underlying implementation to create a new instance. For example, given the ADT point and polygon (See Section 11.2) with constructor `point(int x,y)` and `polygon(point p1,p2)`. Then, the term `point(1,1)` creates a point instance and `polygon(point(1,1),point(0,0))` a literal polygon.

A collection value is denoted by a square bracketed list of expressions. The collection values are coerced to the proper type based on the context in which they appear. The user can indicate

its type by prepending the list with SET, BAG, LIST, or ARRAY. The empty collection is indicated by an empty list `[]`.

For example, SET `[2,3,5,7,11]` denotes a literal set of (some) primes. An initialized array could be ARRAY `[ 'Amsterdam', NIL, 'London', 'Paris']`.

An alternative for calling the tuple constructor function is to use a tuple construction block, which is indicated by the type name followed by a statement block. Within the statement block the attributes are treated as local variables, whose value is saved upon leaving the block in the tuple value under construction. The block can be left prematurely using a BREAK statement. An optional attribute comes into existence only when an assignment within the block takes place.
For example, the person John Dough below can also be introduced with a tuple construction block.

```
Person{name='John Dough'; home='Graveyard';}
```

4 Functions

In Goblin we distinguish two base classes of functions: *computational* and *stored*. Computational functions provide an abstraction mechanism over statement sequences. This class is further divided into *imperative* and *targeted* functions. Imperative functions correspond with the notion of functions (with side effects) as they occur in imperative programming languages. The targeted functions provide control over successive states of the value that is the focus of the function call. They are used both to describe derivable properties of the focus and to encapsulate algorithmic actions whose effects can be (partly) undone.

The stored functions encompass the representation of partial functions in terms of data structures. They are divided into the class of *projection* functions and *access* functions The projection functions are defined over tuple types only and they extract a portion using an attribute list. The access functions generalize array indexing to permit predicate-based access to any sub collection.

The parameters in a function header are considered local variables bound to the actual parameters upon activation. The parameter passing scheme follows the convention of C, where atomic and tuple types are passed by value and collection types by reference.

4.1 Imperative Functions

In the design of a programming language one faces the choice to take either a pure functional approach, i.e. it does not produce side effects, or the routine approach, i.e. side effects are allowed. We have chosen the latter, because in practice side effects from within the function body can model useful behavior, such as input/output and database updates.

Definition and application of the imperative functions follows the conventional approach from imperative languages. Each function names a mapping between a (possibly empty) tuple type, its domain, and another type, called its range. The function body is a statement block to describe the transition in algorithmic terms. For example, the function below introduces a skeleton of an imperative function to construct polar coordinates from a point.

```
TYPE Point = (FLOAT y,x) ;
TYPE Polar = (FLOAT r,phi) ;
Polar polar1(Point p) { RETURN Polar(sqrt(p.x* p.x + p.y* p.y),atan(p.y/p.x));}
```

4.2 Targeted Functions

A *targeted* function is a computational function for which a focus is specified. The targeted function generalize the notion of a method encountered in OOPLs by providing useful semantics for compound values as well. They allow partial side-effect-free functions to be defined and they provide a stepping stone for transaction management by provision of recoverability (See Section 10.1). A targeted function comes in two flavors: as a retrieval method and an update method.

A retrieval method obeys the general format $R\ T.function()$ with T called the *target* and R the *result* type. The function is called by replacing T by a value or lvalue.[2] The target indicates a portion of the program store that should be kept invariant. The compiler flag attempts to assign to T and its attributes. Moreover, side-effects upon T are catched at run time. Then T is restored to its value upon entering the function body and it causes termination with an ERROR value. It does not involve restoring effects on program parts other than the target.

For example, the code fragment below introduces a retrieval method to convert a point into polar form. We say that the function is applied to a target.

```
Polar Point.polar2() { RETURN Polar(sqrt(x* x+y* y),atan(y/x));}
Point pvar; pvar.polar2();
```

An update method obeys the format $T.function()$, i.e. we omit a result type. The update method constructs a new version of the target. A target can be accessed in the function body by SELF and OLD, which denote the current state and the value upon entering the body, respectively. For retrieval methods they are evidently aliases. Moreover, the user can ABORT the function, which immediately terminates the body and restores the target.

A targeted function introduces an embedded lexical scope where the attributes of the target may shield global variables with the same name (See Section 3.1).

Each targeted function is automatically overloaded with a variant that can be applied to a collection. The order of function application over collections is explicitly left undefined, because this hinders potential parallel execution.[3] To illustrate, let `s` denote a variable of type SET `(Point)`. Then `s.polar2()` is a shorthand to apply the function to all elements in the collection. The result is a set of `Point`.

4.3 Projection Functions

Projection is a generic function defined over tuple types only. Its general format is a tuple value `V` followed by a projection list, like `V.(a,...,z)` where a,...,z denote a subset of its attributes. The result is a tuple type derived from the projection list. A tuple with one component is coerced to its value automatically in the context of an expression.

If `V` denotes a collection K(`T`) then `V.(a,...,z)` is a collection of type K(`T.(a,...,z)`) . For example, consider a variable SET`(Person) g` and a LIST`(Person) l`. Then the expression `g.(name)` evaluates to the distinct person names and `l.(name)` denotes all names in the order encountered in the list.

A special case of the projection function is the path expression, which produces the lvalue of a single component of a composite value. Its format is a dot-separated identifier list, such as `V.a.b.c` where `a,b,c` are attributes of a three level composite value `V`. A path is interpreted as a repeated application of a single projection. The collection constructors encountered are retained. Thus, the expression `person.kids.name` becomes a family of sets with children names.

The path identifier can be replaced by an indefinite repetition term `(x.p!)`, which constructs an array with all paths of the form `x.p.p.`.... A definite repetition can be obtained by placing an index expression, e.g. `x.p![3] = x.p.p.p`.

4.4 Access Functions

In many situations we have to deal with functions over a finite domain, i.e. a partial mapping from type $T_1 \rightarrow T_2$. The difference with the computational functions is that these associations are explicitly stored. For example, arrays are a convenient syntactic aid to manipulate a partial function from the index type to the array element type.

[2] A *lvalue* is an expression referring to a storage container. The word was originally coined to mean 'something that can be on the left-hand side of an assignment'. In an object-oriented setting it also denotes an object identifier, an value of type OID.

[3] A function re-definition invalidates the system-defined variant.

In Goblin array access has been generalized to include selections over any collection. Associated with each collection variable there exists a generic access function parameterized by a boolean expression to identify a subcollection, a set of lvalues, for subsequent processing.

For example, consider variables `l,s,b` to denote a list, set,and bag of letters, respectively. Then, the first expression below evaluates to a list of lvalues to denote the letters sent by Smith. The others illustrate a subset and bag. The reserved name SELF provides access to the collection element under consideration. It can be omitted in most situation, because the selector forms a nested lexical scope initialized with the attributes of the array element type.

```
l[sender.name=='Smith'];
s[SELF.sender.name=='Smith'];
b[SELF WITH sender.phone];
```

Array denotations are access functions with a traditional syntax where an index expression produces the lvalue of a single array element. Actually, in Goblin the arrays can be used as any collection variable, where the built-in attribute `index` provides access to the index value and `value` to the value of the array element. Thus, the expression `a[5]` is a shorthand for `a[index==5].(value)`.

Incidently, note that access functions provide a convenient shorthand for selection from a collection of tuples. Together with the projection functions it captures two of the predominate database query primitives. For example, consider a variable `friends` of type BAG`(Person)` then the expression `friends[name=='Peter']` denotes a bag of all friends called Peter and the friends with up to three children are obtained with `friends[kids.count` IN `1..3]`.

5 The Role of Classes

The notion of object and class has been subject to much debate in the database arena [Atkinson89, Cattell 91]. This calls for a definition of our terminology used in the sequel of this report. Below we analyse first the properties of the class concept. In the next chapter the Goblin classification scheme is described in more detail.

5.1 Hiding and Encapsulation

Key concepts in object-oriented languages and OODBMS are information hiding and encapsulation. In Goblin they are used in the sense described below.

In Goblin 'hiding' is understood as 'not being aware of how things are implemented'. The abstract data types illustrate a language concept that supports hiding of information. From a user point of view, one is not really interested how they are implemented, as long as their functional, storage, and performance behavior is known. These aspects do not imply detailed implementation knowledge, but the necessary knowledge to consider alternative solutions after having observed the behavior of the algorithm.

For example, after using a SET`(Person)` for some time, one may conclude that due to circumstantial knowledge a re-implementation of this set pays off in storage and performance. For, the set size might be fixed up-front and membership might be the predominant operation. Then, it makes sense to implement this set with an array and to use a hash function to speed-up membership tests.

An alternative use of hiding occurs at the level of name spaces where names of outer lexical scopes do not automatically propagate to inner scopes such as the separation between an interface and implementation section. Goblin considers this an authorization issue at the level of file access, because local hiding means little when the user has already acquired access to a source file. Moreover, extreme large source files are better broken into smaller components for effective management any way.

In Goblin 'encapsulation' is understood as 'not allowed to modify information at all places in the program'. The motivation for encapsulation is to isolate modifications on instances of a

type into one place, such that maintenance and debugging the program becomes easier. One may have detailed knowledge on implementation issues. Encapsulation is provided through classes and methods (See Section 6.2) and hiding through an Abstract Data Type facility (See Section 11.2).

5.2 Authorization and Class Evolution

Encapsulation and hiding may easily be confused with authorization, which embodies the notion of disallowing users free access to resources, i.e. access control at the source level. Evidently, an ADT can be (partly) used for access control, because the user is given control over the implementation through a fixed interface. The actual implementation may involve hidden functions and a hidden complex state. Actually, its implementation may only be available in the form of a compiled library. Likewise, encapsulation can provide a form of authorization when the class implementation is separated at the source level from its interface description.

In Goblin authorization is considered an orthogonal issue to hiding and encapsulation to be handled primarily by the environment. That is, the compiler and operating environment gives discretionary access to ADTs and (class) definitions.

In our opinion, a related misconception is the effect of successive refinements of a class implementation, i.e. class evolution. A common argument for hiding the object representation and the algorithms for its methods is that it simplifies re-implementation with other data structures and algorithms. Since strict hiding overly constraints the implementation of subclasses, this restriction is sometimes removed explicitly for 'friends' [Stroustrup, Hailpern 90]. Another argument is that it simplifies reasoning about a class behavior and evolution, because one has to consider its interface definition only. This presupposes an interface description including pre- and post- conditions and a correctness proof of the (changing) implementation against this interface. Thus, the interface plays the role of a decorated program invariant or partially ('formal') specification.

The real argument to support ease of class evolution is that an interface should be as small as possible. The more becomes known about its semantics, the harder it gets to maintain these semantics during method re-implementations or class enhancements. Authorization at the source level partly helps in limiting visibility of these aspects.

5.3 Persistency and Sharing

Object persistency is an integral part of object-oriented systems with the following meaning in Goblin: any item (value, class, type, function, trigger) is persistent if it survives a session interrupt, such that upon restart it is in a committed state. Moreover, persistency is a transitive property that applies to object components and their definition.

Persistency presupposes a name space that extends beyond a program execution that binds names with persistent values or definitions. As such, the persistent names with their bindings form the primary means to share information between individual program invocations.

In Goblin all global definitions and variables are considered persistent and shared among the user having access to the database server. Thus, the name space is divided into the persistent names and pure local names. Transaction management regulates proper behavior of concurrent access against persistent items (See Section 10.1).

5.4 Object Classification

The primary role of classes in Goblin is driven by the need for (semi-automatic) classification of composite values against a given scheme. Classification in real-life situations takes into account several factors, where information can be classified by its properties, its value, its position in time, its behavior, etc. In many situations a single object falls into several categories.

The class notion in Goblin aims at providing precisely this mechanism. Each class describes a category of objects that bear the same set of properties and that exhibits the same behavior. Any tuple value can be put under the classification scheme by assignment of an oid. For example,

objects with a name and a telephone can be used to classify information, as can all printable objects together form a class.

After these reflections on the role of classes in Goblin, a more detailed description of its syntax and semantics can be given.

6 Objects and Classes

In this section, we introduce the Goblin language features for the description of object classes. A noticeable difference with other object-oriented languages, such as O_2, is that attributes can be inspected anywhere in the program. However, attribute modification remains confined to object creation time and through methods associated with a class definition. This approach permits querying classes and still provides encapsulation for update operations. Shielding attribute values against unintended disclosure should be implemented by encryption.

Each class describes a minimal set of properties, i.e. attributes and methods, that the objects in its extent obey. In addition to direct creation of a class member, an object becomes a member when it has acquired sufficient properties to satisfy the class definition. The latter is supported by direct manipulation of attributes and methods at run time.

The methods' body describes (update) actions to the object state, i.e. changes to the tuple that represent the object state. The targeted function introduced in Section 4.2 conveniently supports this algorithmic behavior. However, a targeted function can be applied to any instance of its target type to obtain a new version. For a classification scheme we would like to be more restrictive. Namely, a method should be constraint to objects from a specific class. This is achieved by overloading the function target type by a class name.

6.1 Objects

An object is an association between a value of type OID, called the object identifier, an attribute list, and methods. The object identifier can be freely moved around as a reference. It can be stored in a variable of type OID and it can be kept in tuple to describe associations. An object comes into being using the constructor NEW(OID) and ceases to exist after applying the method DESTROY.

The attribute set can be manipulated at run time using the *extension* and *reduction* operators.[4] The *extension* left-hand operand is an object identifier and the right-hand operand is a tuple valued expression. Its effect is extension of the attribute list with those defined for the tuple. The attributes are initialized with the values taken from the tuple. Alternatively, the right-hand-side is a method to be added to the list. In both cases the result of this operation is the object identifier and an error is generated upon attempts to redefine properties.

The *reduction* left-hand operand is an object identifier and the right-hand operand a tuple (type) expression. Its effect is the removal of the attributes that the object has in common with the tuple (type). Likewise, methods can be selectively discarded from the object. Alternatively, an attribute identifier or class method are given directly.

For example, below we construct two object B and P. The object B describes a book, from which we subsequently remove the publisher. P is extended with the attributes of a `Person` tuple value. It is successively refined with an attribute to describe his car model. Finally, we add a specific class method.

```
TYPE Carmodel = (STR carmodel);
TYPE Book = (STR title,author,publisher);
{ OID P=NEW(OID),B=NEW(OID);
B :+ Book('Database Machines','DeWitt','Springer');
B :- publisher;
```

[4]The attribute operators are overloaded for collections of OIDs in the left-hand-side. They can be redefined to capture application specific properties.

```
    P :+ Person('John Dough',?,'Graveyard',?);
    P :+ Carmodel('Ford');
    P :+ person.rename;}
```

Object encapsulation in Goblin is based on a strict separation of object inspection from object modification. That is, the state of an object can be inspected at any time by considering its identifier as a expression whose type is derived from its attributes. For example, the expression `P.carmodel` extracts the value of an object attribute. Access to a non-existing attribute raises an error on which to react with a trigger.

6.2 Class Definitions

A class in Goblin is an automatic aggregation of objects with identical properties, i.e. a class description enumerates the required attributes and methods explicitly. The behavior of the class member is described by targeted functions. Those already defined on the representation type remain valid for object inspection. Modification should be explicitly coded into a method, i.e. a targeted function constraint by a class name.

For example, the declaration CLASS `person` TYPE `Person` collects all objects whose state is recognized as encompassing a value of a type of `Person`. The class has no explicit methods, because we have not yet introduced a targeted function for members in its extent nor its underlying representation type. That is, the class is merely a container of similar structures.

The role of a class name depends on the context in which it appears. It can be used as a collection variable in all situations where such a variable is permitted. It can be used as a type name in a declaration to denote a reference to an element of its extent or as a term in a type discriminator. Moreover, it can be used as an object constructor (Section 6.3).

For example, below two classes called `folder` and `person` are introduced. The visible state is described by the type `Mail` and a tuple type description, respectively. The targeted functions specify update method; to be redefined with a body later on. The method `age` illustrates a parameterless derivable attribute. A person's name can be denoted by SELF`.who.name` or with the equivalent shorthand `who.name` within the body of method `rename`.

```
CLASS  folder TYPE Mail;
       folder.archive (File f);
       folder.forward (Person toP);

CLASS  person TYPE (Person who; Date dob);
       person.rename(STR n);
       person.move(STR newhome);
INT    person.age() {
       IF(Today.mo > dob.mo  || (Today.mo==dob.mo  && Today.dy>dob.dy))
                RETURN Today.yr- dob.yr + 1;
       RETURN Today.yr- dob.yr;
}
```

Common practice is to permit method overloading in sub-classes. In Goblin a method $C_1.m(P_1)$ can be overloaded to $C_2.m(P_2)$ if both $P_2 \prec P_1$ and $C_2 \prec C_1$. In that case, the method $m(P_2)$ replaces the definition of $m(P_1)$ for objects in the class C_2.

The method applicable is stored together with the object, such that the operation `o.m(p1,..,pn)` can be executed with a simple table lookup. Applying an inappropriate method raises a run time error.

For example, consider the subclass `employees` of the class `person` by introducing a working address and consider variables `p` and `e` to denote a person and an employee, respectively.

```
CLASS employees ISA person + (STR work);
       employees.move(STR nh,nw) {person p=SELF; p.move(nh); work=nw;}
```

Then the expression `p.move('Parklane')` execute the code specified for the class `person`. The person's home and working address are changed with `e.move('Parklane','Castle')` where the method in class `person` is called first.

6.3 Class Factory

From the class description a constructor function is derived that creates the object, initializes the attributes, and links the minimally required methods. A NIL initialized class instance is obtained with NEW(`Classname`). Further initialization and consistency checking is modelled with an update method or a trigger (See Section 9). Moreover, the functions NEW and DESTROY can be redefined by the user to incorporate application specific actions.

For example, the class `person` can be populated with Smith using a class instance constructor or from a tuple value followed by automatic object classification. The latter is illustrated by the second statement using the lower level primitives. The tuple construction also illustrates leaving out a trail of optional values.

person{name='John Smith'; phone=123142};
NEW(oid) :+ Person('John Smith',?,123142);

7 A Classification Scheme

A classification scheme gains in effectiveness when it simplifies descriptions of a taxonomy. This is supported in Goblin with an inheritance scheme based on the attributes, methods and a constraint. The scheme can be constructed both by class refinement and class generalization. These issues are described separately below.

7.1 Subclass Lattice

The classes are organized into an inheritance lattice based on their attributes and methods. Let C be a class with state representation type T_1. Then the class introduced by CLASS SC ISA $C + T_2$ is called a subclass or specialization of C. The members of SC have at least all attributes and methods of both C and T_2. Moreover, the extent of SC is a subset of C. We indicate this class relationship as $SC \prec C$.

For example, we can model the well-known diamond-shaped subclass scheme around the class of `person`, `students`, `teachers`, and teaching assistants `ta` where each subclass introduces a new class specific property, as follows.

CLASS student ISA person + (STR major);
CLASS teacher ISA person + (INT salary);
CLASS ta ISA student + teacher + (INT coursenr);

These constructions tell that each student and teacher belongs to the class of persons. The teaching assistant is an object with the combined property set of a person and a teacher extended with a course assignment. Additional methods are required to update the instances.

Subclasses can be used like any other class. An object c in class C can be included into the subclass SC through property extension, like `c :+` T_2. Conversely, the object c can be removed from the class SC and its subclasses with property reduction, while it remains a member of C. For example, `c:- major` removes the attribute `major` and, consequently, the student is not a `ta` and `student` anymore. However, the attribute `coursenr` is not automatically destroyed, because their may exist another (or future) subclass in the class lattice interested in this property. Integrity enforcement over relationships -and objects in general- is handled by the trigger mechanism discussed in Section 9.

7.2 Embedded Classes

The (sub) classes introduced so far are populated through property extension/ reduction. A refinement is to populate a class through a database query (See Section 8.2), called an *embedded class*, which resembles a database view with (possibly) new attributes. At this point two examples suffice to illustrate this language mechanism.

Reconsider the class `person` defined above. Then we can define two embedded classes `adults` and `children` based on the person's age. The class `adults` and `person` have the same structure and behavior; they only differ in their extent. The embedded class may be complemented with methods.

Maintenance of `person`, `adults`, and `children` is intimately coupled. Classification of an object as an `adults` implies classification as a `person`. Moreover, classification of a `persons` activates classification into either `children` or `adults`. Furthermore, objects migrate automatically from `children` to `adults` upon becoming of age 21. Methods defined for `adults` are not applicable to `children`.

CLASS adults ISA person WHERE age>=21 ;
CLASS children ISA person WHERE age <21 ;

The second example illustrates classification by behavior. It uses the classification mechanism by considering methods as optional object properties. Then, a type discriminator expression can be used to infer the behavioral class.

For example, assume that a method `print` exists. All objects for which this method is defined can be collected in a class of printable objects as follows:

CLASS printable WHERE SELF WITH print;

Property reduction can be used to construct an embedded class. That is, the class C can be generalized to CLASS GC ISA $C - I$ where I is an attribute or method name. The extent of GC becomes a superset of C, because there may already exist isolated objects satisfying the property constraints over GC (See Section 7.1) but without property I.

For example, Goblin permits a user to start with the definition of the class `employees` followed by the definition of a generalized class:

CLASS person ISA employees-sal;

After this declaration the class is interpreted like any class definition. Note that the employee methods remain applicable to employees only. They may have to be dropped too or to be redefined to obtain the appropriate behavior.

7.3 Object relationships

The class hierarchy can be seen as a unary association between classes through their OIDs [Albano et.al]. A relationship between objects can be explicit or derived. An explicit object relationship is described by a tuple value over several OIDs where the attributes describe the role of its constituents in the relationship. Consequently, the relationships themselves obtain an OID. An implicit relationship is obtained by a query expression over the database.

For example, below the relationship between student and teacher is an explicit relationship. The implicit object relationship `couples` locate persons living at the same address.

CLASS tutor TYPE (teacher t; student s);
CLASS couples ISA (person m,f) WHERE m<>f && m.address==f.address;

8 Data Manipulation

Most of the imperative programming features offered by Goblin are fairly standard. It includes expressions, assignment - , conditional - , repetition - , for -, switch - statement, and function calls. Their peculiarities are briefly described in Section 8.1 relying mostly on the reader's general understanding of these programming constructs.

An important design objective of Goblin is to provide database query facilities. The database selector and iterator introduced in Section 8.2 introduces the core language facility for this purpose and mimics an SQL SELECT-FROM-WHERE construct.

8.1 Operators and Expressions

The conventional set operators, union(+), difference(−), intersect(*), and set comparison (<, >, <=, >=, ==, ! =) are available for their manipulation. The membership operator is denoted by the IN, whose interpretation depends on the equality operator defined on its element type. For classes the set operators work on the OIDs rather than the object values.

The predefined operators on multi-sets (bags) are union (+), subtraction (−), equality (==) and inequality(!=), and membership (IN). For lists we have included concatenation (+). List and array slicing is provided by the access functions (See Section 4.4).

The built-in aggregate operators for integer collections are: `count, sum, avg, min`, and `max`.

We use the C-inspired mechanism to shortcut boolean expression over collections as well. That is, the predicate S is true if it denotes a non-empty collection. Two predicates are defined to distinguish object sharing from value equality. The binary operator *meet* (><) over path expressions holds when both paths end at the same object. The operator *diverge* (<>) holds otherwise.

8.2 Database Queries

The central language concept for querying a database, called a *selector*, is derived from the SQL SELECT-FROM-WHERE construct cast into the Goblin syntactic framework. The purpose of a database selector is to determine all variable bindings that satisfy a given predicate. Its format is $[BindingList; Condition; Target]$, where $BindingList$ introduces variables bound to elements in the collections, the (optional) $Condition$ denotes a predicate over a given variable binding, and the (optional) $Target$ the resulting value. The binding list is interpreted left to right, which permits references to components of previously bound variables, although the actual binding order to instances remains explicitly left undefined. A single variable binding satisfying the predicate is obtained by prepending the keyword SOME. The selector can be used as a *query* expression to produce a collection, which can be used in any expression where such a collection is permitted.

To illustrate, the query expression below collects all letters sent by children to their parent into a bag of tuples. The target constructs a tuple value from each qualifying binding. Duplicates can be removed by filtering the result to a set.

```
rv = [ l IN folder, p IN person,c IN p.kids;
            l.sender==c.name  && l.receiver==p.name;
            Rv(l.sender, p.receiver, c.name) ]
```

The target list may be omitted, which leads to producing a bag of tuples, called a *marking*, whose type is derived from the binding list. The aforementioned example is phrased as follows:

```
BAG(folder l; person p,c) rv;
rv = [l IN folder, p IN person, c IN p.kids; l.sender==c.name  && l.receiver==p.name];
```

The selector and access functions permits the GROUP-BY and HAVING constructors of SQL to expressed concisely as well. For example, grouping persons by age and counting these groups - producing a bag of lists with two elements each - is obtained by the expression:

```
{ result= [a IN person.age; true ; a,[p IN person;p.age=a].count]; }
```

8.3 Collection Iteration

In most situations, query expressions can be conveniently used to described actions over bulk data. The elements of a collection can be processed on a sequential basis with the collection iterator of the form SELECT(*Bindinglist*;*Condition*) where the condition is optional.

The statement block of the iterator is (conceptually) executed in parallel for each variable binding. This parallelism can be exploited to its fullest only when the statement block does not incur side-effects, such as modifying global variables. However, side-effects are not forbidden and the system will serialize access using the transaction semantics.

The former query expression can be phrased equivalently using a database iterator as follows:

```
BAG(Rv) rv;
SELECT ( l IN folder, p IN person,c IN p.kids;
            l.sender==c.name  && l.receiver==p.name)
            rv += Rv(l.sender, p.receiver, c.name);
```

9 Active Database Support

Several of the application domains require an active database system, which allows, for example, to model flexible integrity enforcement. In Goblin active database support is based on the notion of triggers, which is a function executed when a condition over the database state holds or when a state transitions occurs.

The design rationale for the trigger functionality is based on the observation that they are mostly used to deal with exceptional situations, such as program debugging, integrity enforcement, and view materialization. To avoid the overhead incurred by calling the trigger runtime system too often and to permit a database to go through a temporarily inconsistent state before reaching a valid one, the only state transitions considered for trigger activation are those implied by targeted functions. Namely, upon ENTRY, EXIT (the default), and ERROR status.

The formulation of a design theory for concurrent triggers that lead to predictable behavior is a research topic within the CWI Database Research Group [Siebes 91]. The facilities described here merely introduce the nucleus for a trigger system to experiment with applications and their implementation schemes.

9.1 Event Triggers

A trigger is an (semi) autonomous function call against the database upon encountering an activation event, which determines the state of affairs to be controlled/ inspected during the call. A trigger system often identifies a coarse-grain activation structure to avoid dealing with an overwhelming number of events at run time. The activation events considered in Goblin are state transitions produced by a targeted function because it provides a lexical unit whose scope can be easily controlled by the user.

The general format of a trigger definition is TRIGGER `T.F(Param)`{...} which denotes an enclosing target function. One trigger per event type can be defined per targeted function to avoid ambiguity on what to activate.

Since a trigger controls state transformations it often needs access to the parameters of the enclosing function. Instead of forcing the user to keep track of them in data structures - and to garbage collect them afterwards- Goblin provides access to the actual parameters of `T.F` through a (renaming of) `Param`.

The semantics of a trigger statement block is identical to that of any targeted function. Thus, within the statement block one can access the latest (non-committed) version of the target, denoted by SELF, and the (committed) version upon entering the enclosing `T.F`, denoted by OLD. They may be of a different type when the targeted function calculates a derived value.

For example, the trigger below is activated upon return of `rename`. Within its body SELF refers to the target value and n to a parameter of the enclosing call.

```
TRIGGER  Person.rename(STR n)
{       IF(SELF.name == '' ) printf('Found a nameless person');}
```

9.2 Trigger Inheritance

The consequence of a strong coupling of triggers with targeted functions is that each time a user introduces a targeted function, he may have to introduce/ change trigger definitions. This inconvenience is removed by focusing on the targeted values instead. That is, we omit the enclosing function `F` and interpret `T` as a trigger for all targeted functions on `T` loosing access to the parameters of the enclosing call. Moreover, this trigger does not hide more specific triggers.

For example, a class can be complemented by trigger enforced integrity rules.

```
TRIGGER employee
{ IF(sal<1400)
        printf('Invalid salary for %s', name);
}
TRIGGER person
{ IF(age<0)
        printf('Invalid date of birth of %s', name);
}
```

A decision problem is raised in this example by the inheritance scheme introduced in Section 7.1, because what triggers are activated when a method `M` defined for `person` is applied to an `employee`? And are triggers inherited in the same way as methods?

Since a trigger is -in many respects- tightly coupled to the invocation of a targeted function, it is appropriate to make it subject to the same inheritance scheme. This way, when methods are inherited by a subclass, so will be their triggers. However, the multiple trigger definition above also leads to an ordering problem. If we had not defined the second trigger things would clear, because then the first should be called under all circumstances. Now that we have triggers at both levels of the class hierarchy the trigger on employees is either a refinement or a redefinition. We have opted for the latter, because it aligns with method inheritance.

9.3 Conditional Triggers

Integrity enforcement often involves checking value ranges and object relationships, which can be directly encoded in the trigger body. However, trigger systems factor out these predicates to highlight the circumstances under which their body is executed. In Goblin this factorization scheme is supported by allowing a selector to indicate the triggering condition on interest.

For example, the class `employees` above is watched for underpaid persons. Factoring out the condition is shown below where the query variable is known.

```
TRIGGER employee WHERE sal<1400
{       printf('What is the new salary for %s',name);
        sal.get();
}
```

Another example, consider a variable `clock` and an agenda with scheduled committee meetings. The triggers below notify the user shortly before the next one and removes the meeting from the agenda. A trigger is activated upon changes to either the clock or the agenda.

```
TRIGGER agenda WHERE SELF.btime-5 > clock
{ printf('Next meeting starts @ %d',btime); SELF:-agenda; }

TRIGGER clock WHERE SELF-OLD >5
{       SELECT(a IN agenda; a.btime-5 > SELF)
        { printf('Next meeting starts @ %d',a.btime); a:-agenda; }
}
```

Conditional triggers easily lead to situations where a target value becomes potentially subject to several trigger blocks. A design theory for this situation is currently underway. The baseline policy for conflict resolution is to give precedence to the user-defined event trigger. Thereafter, the statement blocks of the query-based triggers are subsequently executed in random order.

10 Transaction Management

The transaction management facilities of Goblin are strongly related to targeted functions as well. To understand the transaction behavior, rudimentary knowledge on the user-program interaction is needed. First, the database designer writes a Goblin program containing the schema, i.e. classes and functions. This is compiled into a server process and subsequently installed on the processor pool. The application program, yet another Goblin program, can access the global objects and apply the functions defined on the database server.

Transaction management primitives ensure non-interference of application programs running on the same database server. The approach taken is similar to O_2 and Ithasca[Ithasca], which both assume transaction processing to take place on the *workstation*, rather than the database server. *Workstation* should be read as a generic term, because a single Goblin application may actually be running on a number of workstations in parallel. Upon transaction commit the modifications in the *workstation* are moved to the database server (i.e. class manager).

The rationale for workstation-based transaction processing is that the (CPU + memory) resources on the server may be limited. Furthermore, applications tend to exhibit time- and space-locality, i.e. the object used by a transaction is likely to be used again (for display) and structurally related objects (components) have a high probability of access. Furthermore, by placing the new copy on the workstations simplifies undo semantics and it improves reliability.

Many policies for transaction processing exist [Barghouti 91] and the choice often depends on the application domain. In the first release of Goblin we aim for traditional nested-transaction semantics. Multi-version and script-based schemes are subject for future research.

10.1 Nested Transactions

Goblin supports short transactions, i.e. atomic, consistent, isolated, and dependable operations on individual complex objects. Atomicity ensures an indivisible sequence of operations. If the transaction aborts, or otherwise fails to complete, then the object subject to modification is restored. This includes restoring the object attributes that have been modified by the transaction. Side-effects that may have occurred are never undone.

A method is the syntactic unit to denote a transaction. The rationale is that it avoids separate bracket structures to indicate the start and termination of a (sub) transaction sequence. It also simplifies analysis of transaction interference, because the syntax aids in localizing the scope of the transaction and the locks to be acquired by the system.

Condition and modification should be combined in methods to guarantee consistent decisions, because, in general, reading an object is only guaranteed to reflect some consistent object state, not necessarily the latest committed state. Thus, any decision test before a transaction is activated should be part of the transaction code as well. A successful transaction will temporarily provide an up-to-date view on the object's state. Its duration depends on the volatility of the object. The rationale for this approach is that it permits caching read-only object states at the workstation. Only upon transaction initiation should this cache be synchronized with the database server.

The method target describes the object with its (recursively accessible) components subject to transaction semantics. That is, upon transaction ABORT the system will restore the target value only. Side-effects that are not accessible from the target value upon entering are ignored. Thus, a method applied to several objects requires construction of a 'transaction object' first.

Transactions form hierarchies where a commit at the outer level depends on commits of all inner levels. Reaching a RETURN statement is interpreted as a successful execution with an implicit sub-transaction commit. Erroneous situations can be flagged with the statement ABORT, which

(partially) undoes the effects of the method body. Furthermore, a method call returns a boolean value TRUE upon success and FALSE otherwise. Ignoring the return value leads to automatic propagation of the ABORT, while an explicit test on success or failure leaves error propagation to the user.

For example, consider a banking environment with the Debet/ Credit application. A simplified class description of accounts is given by `acc`. The basic action is to change the balance of an account with a certain amount. A transaction abort occurs upon attempts to withdraw money from an insufficient balance. Moreover, we assume a limit on the balance as well.

```
TYPE Account =(INT nr; STR name; INT balance);
CLASS acc TYPE Account;

acc.change(INT amount)
{ balance += amount;
        IF(balance < 0) ABORT;
}
```

A money transfer requires two such sub-transactions on separate accounts. A global Goblin function encapsulating both would not work, because either or both `change` operations may fail. Instead we construct a transfer method whose target contains an entry in a journal `hist`. It aborts when no money can be obtained from the debitor or when the creditor fails to accept the money. In both cases the state of `fromAcc` and `toAcc` are restored only. The global variable `nroftrans` is always incremented. Success is also recorded in the journal.

```
TYPE History =(acc fromAcc, toAcc; INT amount; BOOL outcome );
CLASS journal TYPE History;
INT nroftrans;

journal.transfer()
{
        nroftrans++;
        if(  fromAcc.change(−amount) == ERROR) ABORT;
        if(  toAcc.change(amount) == ERROR) ABORT;
        outcome= TRUE;
}
```

In general one would pass the account number rather than the object identifiers. This leads to an additional transaction layer that constructs a journal object first, because we should ensure that the account objects exist before the transfer is issued. Outside the transaction boundaries one only knows that they have existed at some point in time.

```
journal.trans2(INT f,t,amount)
{
        acc fromAcc= acc[nr==f];
        acc toAcc= acc[nr==t];
        journal(fromAcc,toAcc,amount,FALSE).transfer();
}
```

11 Extensibility

Many years of research in programming language design have not produced a single accepted framework for writing software. No such framework is likely to emerge in the near future either. Therefore, new programming languages provide some level of extensibility. In Goblin, we assume a fixed syntax, but permit refinement of the semantics for experimentation by enhancing the semantic analyser/ optimizer.

Extensibility is illustrated through refinement of built-in operators (See Section 11.1), the Abstract Data Type facility, which forms the interface with (separately compiled) external libraries (See Section 11.2), and macro expressions (See Section 11.3).

11.1 Overloaded Operators

The built-in definitions of the Goblin operators have been chosen with care. However, there are situations where their definition does not exactly comply with the user's wishes. The new definition can be given to the compiler in the form of *term* variables and *rewrite* rules.

A term introduces a unique name for a Goblin expression and it plays the role of a pattern definition in the rewrite rules. The term can be parameterized by an expression to limit pattern matching.

For example, below the term `I` and `S` represents an integer and string expression, respectively. The term `Y` matches a multiplication of a string by an integer. The terms `A` and `B` match any Goblin expression.

```
TERM INT I;
TERM STR S;
TERM Y = I*S;
TERM A,B;
```

A rule introduces a rewrite action for the compiler. It starts with a Goblin expression to be looked after followed by an ordered list of reductions. Each reduct is optionally parameterized by a rewrite condition. Term rewriting is based on a left-right bottom-up reduction strategy.

For example, the built-in operator ++ can be enhanced to represent string concatenation by calling a C-function. The second rule implements repeated concatenation by a constant factor. Its first reduct fails when `I` does not represent a value known at compile time, which causes the error to be raised.

```
RULE    OPERATOR+(A,B)
=>      strcat(A,B) WHEN A ISA STR  && B ISA STR ;

RULE    A*S
=>      S+(B*S) WHEN A>0  && B=A-1
=>      ERROR 'Integer constant required';
```

Redefinition may easily cause confusion for the (other) user(s) and for the compiler to distinguish different cases. Therefore, it is best to limit operator overloading to those cases where the compiler does not yet provide a definition.

11.2 Abstract Data Types

The Goblin compiler supports linking packages with pre-compiled routines, such as provided by its operating environment. In particular, the compiler considers all unresolved function names as to be supplied at link-edit time.

To support type checking the Goblin program and to highlight the dependencies upon the external libraries, the user can specify the relevant aspects with a (partial) Abstract Data Type specification. A specification consists of four parts: a header, several applicable functions, term variables and behavior rules.

An ADT definition header names the ADT and the type describes the Goblin data structure to keep a representation of (or reference to) the external object. The properties of the representation can be directly accessed for inspection. However, they can only be modified by applying a targeted function. Furthermore, a constructor function is implied that takes a composite value to create the external representation of the ADT instance.

The header is complemented with a series of targeted functions to manipulate the external objects. An empty body is interpreted as a specification only. It assumes that the body is

available in a library to be linked during compilation of the Goblin program. Targeted functions with a body replace/ enhance the external library with program specific code. Their body has modify permission on the representation structure unless explicitly forbidden by the ADT manual.

An abstract data type can be implemented by a separate Goblin or C program. Conformance of their implementation with the ADT specification is left to the discretion of the user (or librarian).

For example, consider a geographic application domain that requires the types `point` and `polygons`. Their partial behavior is specified below. The user can access the x- and y- coordinates directly, but should use a method to change them, because the actual implementation might be based on a polar notation rather than coordinates. In general, an ADT representation should obey as much as possible the semantics of Goblin. Thus, two points are considered equal when their underlying representations are equal. Otherwise, the equality operator should be overloaded to deal with the refinements.

```
ADT    point TYPE(int x,y);
       point.put_x (int x);
       point.put_y (int y);
       point.copy (point src);
bool   OPERATOR==(point p1, p2);

ADT    polygon TYPE LIST(point);
       polygon.put_poly(LIST(point) y);
```

The next part of an ADT specification comprises term declarations and rewrite rules that describe their behavior. They can be used to obtain a more detailed description on their functionality and to optimize code.

For example, we might introduce two rewrite rules for the type `point` above. The first tells the user that point equality is semantically equivalent to comparing their original constructor values. The second illustrates point addition. The equality axioms illustrate that the compiler can assume that equality of point identity implies value equality and it can optimize code for expressions such as `==(point(1,1),Z)` without going through the mechanism to create a point representation first.

```
TERM point X,Y;
RULE      OPERATOR==(X,Y)
=>        X.x==Y.x  && X.y==Y.y;
RULE      OPERATOR+(X,Y)
=>        point(X.x+Y.x,X.y+Y.y);
```

11.3 Macros

Expression macros, i.e. most of the C **# define** statements encountered in programs, are conveniently captured by the term rewriter primitives. In particular, the Goblin rules directly support in-line functions and the declaration of compile time constants.

For example, the in-line function `max` below is defined for several atomic types. Moreover, a specific compiler error is generated if the types do not match. A preprocessor approach may blur the location where such a type error occurred. The compile time constant `pi` is equivalent to an initialized constant variable, i.e. CONST FLOAT `pi=3.1415927`.

```
TERM X,Y;
TYPE intflt =INT | FLOAT;
RULE      max(X,Y)
=>        (X>Y?X:Y) WHEN X ISA intflt  && Y ISA intflt
=>        strcmp(X,Y) WHEN X ISA STR  && Y ISA STR
=>        ERROR 'max requires str,int, or float argument';
RULE pi   => 3.147;
```

12 Summary and Future Work

In this report, we have described the Goblin Database Programming Language, designed as a vehicle for research into database support for advanced applications. The prime novelties of Goblin to DBPLs reported elsewhere are its rich typing scheme, its focus on the automatic classification of objects, and inclusion of a conditional term rewriter features to drive the compilation process.

The query facility bears similarities with the select-from-where concept encountered in relational systems and it provides a hook for parallel execution. The result is a flexible language that supports a wide range of queries types not as easily formulated in a traditional SQL systems.

In retrospect to the requirements posed on a DBPL, we have not yet addressed the issues arising from astronomy, such as schema evolution and approximate querying, and CIM, such as a flexible transaction mechanism. Moreover, we are investigating the extensions to accommodate trigger priority schemes[Siebes 91].

The work reported has been conducted primarily in 1992 and 1993 as a means to direct and stimulate related research issues. A prototype Goblin compiler has been development for two distributed platforms: Amoeba on Intel386 and Silicon Graphics MIPS machine. The Goblin compiler consists of several strongly related tools. The syntax checker *gql* performs a straightforward LR(1) parse. Its output is an ASCII representation of the parse tree, including default values and re-ordering of information to ease subsequent processing.

The parse trees are feed into the semantic analyser *gqc*, which is the conditional term rewriter mentioned before. It is designed for incremental compilation, which means that redefinition of a global name is permitted; it simply warns the user about the new definition. Undefined function names are considered external C-function to be provided at link-edit time.

Two PhD thesis have been written in 1993 addressing the architectural support [vdBerg 1994] and design theory for trigger systems[Voort 1994]. A implementation of the database kernel is operational and runs on a shared store multiprocessor.

Further implementation efforts are driven by the need to experiment with language and run time aspects. A hard constraint is the manpower required to fully implement and promote a language of this complexity. We therefore do not foresee a complete implementation in the near future.

Acknowledgements

The authors wish to thank S. Plomp, F. Kwakkel, F. v. Dijk, and guests of the CWI Database Research Group for constructive comments on early versions of this document.

References

[Albano et.al] A. Albano, G. Ghelli, R. Orsini, 'A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language', *Proc. 17-th VLDB* 1991, pp. 565-576

[Agrawal89] R. Agrawal and N.H. Gehani, 'Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++', *Proceedings 2-nd Int. Workshop on Database Programming Languages*, June 1989, pp. 25-40.

[Amble81] T. Amble et. al., Draft report on the programming language ASTRAL, *ASTRA-notat nr. 31*, Div. of Computing Science, Univ. of Trondheim, May 1981.

[America87] P. America, 'POOL-T — A parallel object-oriented language', In: Akinori Yonezawa, Mario Tokoro (eds.) *Object-Oriented Concurrent Programming*, MIT Press, 1987, pp. 199–220.

[Atkinson89] M. Atkinson, 'Questioning Persistent Types', *Proceedings 2-nd Int. Workshop on Database Programming Languages*, June 1989, pp. 2-24.

[Atkinson81] M.P. Atkinson et.al. 'PS-Algol: An Algol with a Persistent Heap', *ACM SIGPLAN Notices*, 17(7):24-31, 1981

[Bancilhon88] F. Bancilhon, Object-Oriented Database Systems, *Int. Symp. on Principles of Database Systems*, Texas, March 1988.

[Barghouti 91] Barghouti, "Survery on Transaction Models..."

[Batory 84] D.S. Batory, A.P. Buchmann, 'Molecular Objects, Abstract Data Types and Data Models: a Framework', *Proceedings 10th Int. Conf. on Very Large Databases*, Singapore 194.

[Batory 86] D.S. Batory 'GENESIS: A Project to Develop an Extensible Database Management System', *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, pp. 206-207.

[Barbic 85] F. Barbic and F. Rabitti, 'The Type Concept In Document Retrieval', *Proc. of the 11th Int. Conf. on Very Large Database*, Stockholm, pp.34-48, 1985.

[vdBerg 91] C. vd Berg and M.L. Kersten, 'An analysis of a dynamic query optimization scheme for different data distributions', *Trends in Query Processing*, 1991, Morgan Kaufmann.

[vdBerg 1994] C.A. van den Berg, 'On Dynamic Query Processing in a Parallel Object-Oriented Database System', Twente University, 1994.

[Carey 86] M.J. Carey et. al. 'The Architecture of the EXODUS Extensible DBMS', *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, USA, page 52-65.

[Camarinha 87] L. Camarinha and A. Steiger, 'An Information System Architecture fro Robot Cell Programming', *ESPRIT Project No. 623*, 1987,

[Cardelli 84] L. Cardelli, 'Semantics of datatypes', Vol. 173 of Lecture Notes in Computer Science, Chapter A Semantics of Multiple Inheritance,1984.

[Cattell 91] R.G.G. Cattell, 'What are next generation database systems', *Communications of the ACM*, Vol 34. No. 10. Oct 1991, pp. 31-33.

[Const 86] P. Constantopoulos et. al., 'Office Document Retrieval in MULTOS', in : *ESPRIT '86: Results and Achievements*

[Dearle89] A. Dearle, R. Connor, F. Brown, R. Morrison, 'Napier88- A Database Programming Language', *Proc. 2-nd Int. Workshop on Database Programming Languages*, June 1989, pp. 179-195.

[Deux 90] O. Deux, et. al. 'The Story of $O-2$', *IEEE Trans. on Knowledge and Data Engineering*, Vol 2. No.1, March 1990, pp. 91-108.

[Freedman 89] P. Freedman and C. Michaud and G. Carayannis and A. Malloway, 'A Database Design for the Runtime Environment of a Robotic Workcell', *Robotics & Computer-Integrated Manufacturing*, 1989, Vol. 5, pp. 21-31.

[Gardarin 88] G. Gardarin, J.M. Thevenin, P. Pucheral, 'GEODE, an object main-memory oriented extensible manager', *Esprit Technical Week*, 1988.

[Gibbs 85] S.J. Gibbs, 'Conceptual Modelling and Office Information Systems', in *Office Automation*, D. Tsichritzis (ed.), Springer-Verlag, 1985, ISBN, 3-540-15129-X, pp.193-226

[Goldberg 83] A. Goldberg, D. Robson, *Smalltalk-80, The language and its implementation*, Addison-Wesley 1983.

[Green 90] J.L. Green, 'The New Space and Earth Science Information Systems at NASA's Archive', *Government Information Quaterly*, 1990, Vol 7, No 2, pp. 141-147, ISSN: 0740-624X.

[Guenther 90] O. Guenther and A. Buchmann, 'Research Issues in Spatial Databases', *ACM SIGMOD Records*, 19(4), Dec 1990, pp 61-68.

[Güting 89] R. Güting, 'Gral: an extensible relational database system for geometric applications', *Proc. 15-th VLDB*, Amsterdam, 1989, pp 33-44.

[Hailpern 90] B. Hailpern and H. Ossher, 'Extending Objects to Support Multiple Interfaces and Access Control', *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, Nov 1990, pp.1247-1257.

[Härder 87] T. Härder, Meyer-Wegener K., Mitschang B.
¿ Sikeler A., 'PRIMA - a DBMS Prototype Supporting Engineering Applications', *Proc. 13th Int. Conf. on Very Large Databases*, Brighton 1987.

[Ithasca] Itasca Systems, Inc. *ITASCA User Manual Release 1.0* 1989

[Kersten 88] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, E.J.A. van Kuyk, R.L.W. van de Weg, 'A Distributed, Main-Memory Database Machine; Research Issues and a Preliminary Architecture', in:*Database Machines and Knowledge Base Machines* (ed. M. Kitsuregawa), 1988, pp. 353-369.

[Kersten 90] M.L. Kersten, 'Large Scale Handling of Complex Objects', *AI International AI Symposium 90*,Nov 1990, Nagoya, Japan, pp. 151-158

[Kersten 91] M.L. Kersten, 'Goblin: A DBPL designed for Advanced Database Applications', *2nd Int. Conf. on Database and Expert Systems Applications, DEXA'91* August 1991, Berlin, Germany.

[Kersten 92] M.L. Kersten, "A Modular Conditional Term Rewriter for Compiling DBPL Programs", Submitted for publication.

[LeCluse] R. Lécluse,C and Richard,P, 'The O_2 Database Programming Languages', *Proceedings 15th VLDB*, Amsterdam, Aug. 1989.

[Lorie 84] R. Lorie and Meier, 'Using a relational DBMS for geographical databases', in *Geoprocessing* no 2, 1984, pp. 243-257.

[Maier 84] G. Copeland and D. Maier 'Making Smalltalk a Database system', *ACM SIGMOD-Records*, Vol 8, 1984.

[Matthes 89] F. Matthes and J.W. Schmidt, 'The Type System of DBPL', *Proc. 2-nd Int. Workshop on Database Programming Languages*, June 1989, pp. 219-225.

[Meghini 87] C. Meghini, F. Rabitti, E. Bertino, 'Formal Semantics for MULTOS Document Model and Query Language', *ESPRIT Project 28*, DM-[IEI]-87-01, 1987.

[Rabitti 86] F. Rabitti, 'MULTOS Document Model', *ESPRIT Project 28*, DM-[IEI]-86-02, 1986.

[Riet 81] R.P. van der Riet, A.I. Wasserman, M.L. Kersten and W. de Jonge. 'High-level Programming Features for Improving the Efficiency of a Relational Database System', *ACM Trans. on Database Systems*, Vol 6, No. 3., Sep 1981, 464-487.

[Roth 88] M. Roth, H.F. Korth, A. Silberschatz, 'Extended Algebra and Calculus for Nested Relational Databases', *ACM Trans. on Database Systems* Vol. 13, no. 4, pp. 389-417.

[Schaller 87] J. Schaller, 'The Geographic Information System ARC/INFO', *EuroCarto VI*, Brno, Czechoslovakia, 1987.

[SIGMOD 90] A. Silberschatz, M. Stonebraker, and J.D. Ullman, 'Database Systems: Achievements and Opportunities', in *NSF Invitational Workshop on the Future of Database Systems Research*, SIGMOD Record Vol 19, no 4, Dec 1990.

[SQL] ISO ANSI, *Database Language SQL*, ??? International Standard.

[SQL2 89] J. Melton, 'Database Language SQL2 and SQL3', *ISO-ANSI working draft*, ISO DBL CAN-2b, May 1989.

[Pistor 86] P. Pistor & Anderson, F. 'Designing a generalized NF2 model with an SQL-type language interface', *Proc. 12th Int. Conf. on Very Large Databases*, Kyoto 1986.

[Schek 82] H.-J. Schek & Pistor P., 'Data structures for an Integrated Data Base Management and Information Retrieval System', *Proc. 8th Int. Conf. on Very Large Databases*, Mexico 1982.

[Schek 86] H.-J. Schek, M. Scholl, 'The Relational Model with Relation-Valued Attributes', *Information Systems*, Vol 2. No. 2, 1986, pp. 137-147.

[Siebes 91] A.Siebes, M.H. vd Voort, and M.L. Kersten, 'A framework for the formalisation of active objects' *Proceedings CSN'91*, nov 1991

[Stonebraker 86] M. Stonebraker, 'Object Management in POSTGRES Using Procedures', *Proc. Int. Workshop on Object-Oriented Database Systems*, Sep. 1986, pp. 66-72.

[Stroustrup] B. Stroustrup, 'The C++ Programming Language', AddisonWesley.

[Rabitti 95] F. Rabitti, 'A Model for Multimedia Documents', in *Office Automation*, D. Tsichritzis (ed.), Springer-Verlag, 1985, ISBN, 3-540-15129-X, pp.227-250

[Tsichritzis 85] D. Tsichritzis (ed.), 'Office Automation', Springer-Verlag, 1985, ISBN, 3-540-15129-X.

[vdVoort 91] M.H. vd Voort and M.L. Kersten, 'The Facets of Database Triggers', CWI Report CS-9122, April 1991.

[Voort 1994] M.H. van der Voort, 'A design theory for database triggers', PhD Thesis, University of Amsterdam, 1994.

[Wasserman 81] A.I. Wasserman et.al., 'Revised Report on the programming language PLAIN', *SIGPLAN Notices (ACM)*, Vol. 16, no. 5 (May 1981), 59-80.

A The Goblin Syntax

This appendix contains the syntax for Goblin 1.0 programs. The constructs `_list` and `_blk` indicate a comma and semicolon separated repetition, respectively. Optional constructs are indicated by square brackets and (possible) repetition is indicated by curly brackets. To highlight the language semantics, classified program identifiers end with `_id`, a typed value ends with `_val`.

```
Goblin_stmt:= compile_stmt | prg_stmt

prg_stmt    ::= body
              | type_def ';' | var_def ';' | enum_def ';'
              | class_def | func_def | trigger_def

compile_stmt= db_stmt';' | adt_def ';'
              | term_def ';' | rule_def ';'

type_def    ::= TYPE id type_op type_expr

type_op     ::= '=' | '+=' | '| =' | '-='

tuple_expr  ::= type_term
              | type_expr tuple_op tuple_term
              | '(' type_expr ')'

tuple_op    ::= '+' | '-' | '| '

type_term   ::= type_factor | tuple_def
              | tuple_id '.' '(' id_list ')'

type_factor ::= type_spec
              | type_factor index | type_factor discr

type_spec   ::= atomic | type_id | collection
              | TUPLE tuple_def

tuple_def   ::= '(' [attribute_blk] ')'

attribute   ::= tuple_expr aname_list

aname       ::= id {index} ['?']

collection  ::= kind tuple_def
              | kind '(' type_expr ')'

kind        ::= ARRAY | BAG | LIST | SET

atomic      ::= SHORT | INT | LONG |
              | FLOAT | DOUBLE | CHAR | STR
              | BOOL | OID | VOID | ANY

enum_def    ::= TYPE id '=' '[' const_list ']'

const       ::= id [ '=' expr]

var_def     ::= [CONST] type_spec var_list

var         ::= id {index} ['=' expr]

class_def   ::= CLASS id [cls_body] [body]

cls_body    ::= TYPE tuple_def [WHERE expr]
              | ISA type_expr [WHERE expr]

body        ::= '{' stmt_blk '}' | ';'

func_def    ::= [type_spec] header body
              | [type_spec] type_spec '.' header body

header      ::= id tuple_def | operator tuple_def

operator ::= OPERATOR '(' symbol ')' | event

trigger_def ::= [event] TRIGGER [cls_body] body

event       ::= ENTRY | EXIT | ERROR
```

Compiler Statements

```
db_stmt     ::= DATABASE id

adt_def     ::= ADT id [TYPE type_expr]

term        ::= TERM [type_spec] var_list

rule_def    ::= RULE expr reduction
              | RULE operator tuple_def reduction

reduction   ::= {'=>' reduct [WHEN expr]}

reduct      ::= expr | REDEX
              | ERROR string_val
              | '{' prg_stmt '}'
```

Expressions

```
expr        ::= expr infix expr
              | lvalue asgn_op expr
              | expr '?' expr [':' expr]
              | prefix primary | primary postfix
              | discr_expr | with_expr
              | migration
              | primary
```

```
discr_expr  ::= expr ISA type_spec

with_expr   ::= expr WITH with_item
              | expr WITH '(' with_item_list ')'

with_item   ::= id {'.' id }

prefix      ::= postfix | '+' | '−' | '~ '

postfix     ::= ++' | '−−'

infix       ::= IN | '><' | '<>' | '..'
              | '<' | '>' | '&&' | '||'
              | '+' | '−' | '*' | '/' | '%'
              | '==' | '! =' | '<=' | '>='

asgn_op     ::= '=' | '%=' | '*=' | '+=' | '−='
              | '/=' | ':+' | ':−'

primary     ::= value | lvalue | query
              | '(' expr ')' | type_cast
              | primary '.' lvalue
              | primary '.' '(' id_list ')'

value       ::= USER | NIL | ERROR | literal
              |TRUE | FALSE

pname       ::= id | SELF | OLD | '?'

lvalue      ::= path [index] | path '(' expr_list ')'

index       ::= '[' expr ']'

path        ::= pname [ '!' [index]]

type_cast   ::= atomic '(' expr ')'
              | [kind] '[' expr_list ']'
              | id '{' stmt_blk '}'

query       ::= '[' binding_list ';' [expr] [';' [expr]] ']'

binding     ::= [SOME] id IN expr
```

Statements

```
generator   ::= SELECT '(' binding_list [';' expr] ')' stmt
              | FOR '(' [expr] ';' [expr] ';' [expr] ')' stmt
              | WHILE '('[expr]')' stmt
              | DO stmt WHILE '('[expr]')' ';'

switch_stmt::= SWITCH'('expr')'
              '{' [switch_item_blk] '}'

switch_item::= switch ':' stmt_blk

switch      ::= CASE expr | DEFAULT
              | CASE discriminator

block       ::= ';' | '{' stmt_blk '}'

stmt        ::= var_def ';' | block | expr ';'
              | IF '('expr')' stmt [ELSE stmt]
              | generator | BREAK ';' | CONTINUE ';'
              | switch_stmt | RETURN [expr] ';'
              | ABORT [expr] ';'
```

B O_2 Program Sample

The paper 'The O_2 System' of O. Deux in CACM Oct'91 describes and illustrates a well-known (commercial) object-oriented DBMS product family. The sample program fragments are converted to Goblin code so as to highlight commonalities and differences.[5] We describe the datamodel and query facilities only.

B.1 Data Model

Both O_2 and Goblin distinguish values from objects and classes from types. Thus, the type definition of a Monument is translated directly with minor syntactic changes.

```
TYPE Adr =TUPLE(City city; STR street; INT number);
TYPE SItem =(Date date; INT visitors);
TYPE Stat =LIST(Sitem);

TYPE Monument =TUPLE( str name;
                      Adr address;
                      FLOAT admission;
                      Stat statistics);
```

The O_2 choice to put the type description after an attribute name in their database schemas improves readability, but conflicts with their C-language syntax orientation of O_2C. To achieve the same reading clarity, Goblin users are stimulated to assign a name to any complex type description first.

An O_2 program has a body, which forms the entry point. In Goblin, all routines can be used as an entry point by selection from the graphical user interface. For example, the routine *makeTower* illustrates a constructor function for monument value. Each tuple type in Goblin has a constructor function with the same name (*Adr*). Alternatively, the tuple value can be constructed through a statement block (*Sitem*) where the tuple attributes are the local variables.

```
monument makeTower(){
        Monument m= NEW(OID);
        m.name = 'Eiffel Tower';
        m.address = Adr(Paris, 'Champs de March',1);
        m.admission = 42.50;
        m.statistics = [ Sitem(toDay,9710) ];
        RETURN m;
}
```

A more concise description is obtained with the value constructor as follows.

```
Monument m;
m= monument('Eiffel Tower', Adr(Paris, 'Champs de March',1),
                      42.50, [ Sitem(toDay,9710) ] );
```

The classes *City* and *Hotel* below illustrate the separation in Goblin of the object state representation from the methods. There is no lexical rule that forces methods to be defined together with the object representation. This is considered a matter of style. Moreover, the user is free to provide the method body directly, or later on by method redefinition.

The method *reserveRoom* merely changes the number of free rooms. Failure to do so causes a transaction (method) abort. Likewise, *checkOut* only increases the number of free rooms.

In O_2 one can also declare a property as being externally read only. In Goblin attributes are read by default except within the body of its update methods. Moreover, a value can be declared as write-once to capture constant values. This feature is not available in the O_2 schema language.

[5] We assume you have access to the CACM issue for comparison.

```
CLASS City TYPE (
        STR name;
        bitmap map;
        SET(Hotel) hotels);

City.buildhotel(Hotel h);

CLASS Hotel TYPE(
        STR name;
        INT stars;
        INT freerooms);

Hotel.reserveRoom() {
        IF(freerooms>0) freerooms−−;
        ELSE ABORT;
}
Hotel.checkOut() { freerooms++;}
```

The O_2 example uses a retrieval method to obtain the vacancies in the city based on the number of stars. In Goblin this calls for the targeted function below, which uses a collection constructor to locate the hotels of interest. The method *sum* is a predefined (parameterless) aggregate function in the Goblin runtime library.

```
INT City.vacancies(INT s){ RETURN  [h IN hotels; h.stars==s; h.freerooms].sum;}
```

The example is further extended with *Restaurant* and *HotelRestaurant* classes to demonstrate handling name collision features in O_2. The latter also illustrates multiple inheritance, i.e. hotels that also have a restaurant. The attribute *star* is ambiguous, which is resolved in Goblin by introducing an association between the classes involved. Thus, *hr.h.star* denotes the stars of the hotel *hr*.

```
TYPE Mitem =(FLOAT rate; STR contents);
TYPE Menus =LIST(Mitem);

CLASS Restaurant TYPE (
        STR sign;
        INT star;
        Menus menu);

CLASS HotelRestaurant TYPE (Restaurant r; Hotel h);
```

The *TouristCity* class illustrates attribute replacement. This feature should be used with care, because it may easily confuse the user. The method *buildhotel* is overloaded to apply the method *neweq* instead.

```
CLASS TouristCity TYPE City - hotel + (
        SET(HotelRestaurant) hotels;
        SET(Monument) whattosee);

TouristCity.buildhotel(HotelRestaurant e)
{ City c=e;
        IF( e.r.star>1) c.buildhotel(e);
}
```

The root objects in O_2 correspond with the global variables of a Goblin program. They are accessible from the user interface. The routine *main* causes the database to be initialized.

```
TouristCity Paris;
set(City) FrenchCities;

VOID main(){
        Monument m;
        City c;

        m= makeTower();
        c= City{    name='Rocquencourt';
                    map= readscanner;
                    hotels= [];
                    };
        Paris.whattosee(m);
        FrenchCities += [c,Paris];
}
```

B.2 Query Language

The O_2Query uses an SQL-like query language extended to deal with complex values and objects. In Goblin we have sticked more closely to the C-like syntax conventions. However, a direct mapping from SQL-like syntax to C-like syntax is possible. For example, the query: "At which restaurants in Paris can one eat for less than 100FF? What choices do they offer?" can be phrases as follows:

```
SELECT(     r IN Restaurant, menu IN r.menus;
            r.address.city.name== 'Paris'  &&
            100 IN menu.rate) printf("%s ",r.name)
```

This query illustrates the selection iterator that performs the *printf* for each qualifying variable binding. It also shows projection over a collection (`menu.rate`) to obtain a set of values.

Alternatively a collection can be constructed for further manipulation as follows:

```
[r IN Restaurant, menu IN r.menus;
r.address.city.name== 'Paris'  && 100 IN menu.rate;
r.name, menu(rate,food) ]
```