Final Report
to
German-Israeli Foundation for Scientific Research and Development

"Bulk Data Classification: Formal Foundation and Enabling Technology"

Contract No.: I-183-060.6
January 1, 1993 to December 31, 1995

Professor Catriel Beeri
Institute of Computer Science
The Hebrew University
Jerusalem, Israel

Professor Joachim W. Schmidt
Computer Science Department
University of Hamburg
Hamburg, Germany

April 1996

Contents

1 Abstract

We will begin this report with a detailed description of the research activities and results of the three project years. Section 2 covers the work of the The Hebrew University group on the formal foundation of bulk data classification. We have developed a *categorical approach* to a general and formal definition of what a bulk type is (section 2.1). This abstract approach to bulk types is complemented by more concrete approaches of query and general programming languages. Therefore, a comparison of existing query languages for bulk types is presented in section 2.2.

Section 2.3 describes an approach on how bulk types can be organized in a hierarchy, which is intimately related to the availability of processing abstractions and query languages. Section 2.4 shows how this approach can be generalized to capture constrained bulk types. Section 2.5 introduces *derived operations* into the hierarchy for optimization purposes and *direct access operations* that can be used to model not only the conceptual database but also its physical organization.

The topic of section 2.6 is a formal theory that combines both recovery and concurrency into a single framework which can be used to understand the kinds of object interfaces that can reasonably and efficiently support concurrency and recovery. Finally, a unifying framework for query and programming languages is described in section 2.7.

Section 3 covers the work done at the University of Hamburg, i.e. the provision of enabling implementation technology building on recent advances in programming languages and system architectures such as polymorphism, higher-order functions, and subtyping. With a good understanding of bulk types and their properties, we developed query and general programming mechanisms that can be used by end-users for general database application developments, for a large variety of bulk types.

A major part of this work is the development of a persistent program representation (section 3.1) that can be utilized for dynamic program optimization at run-time (section 3.2). In section 3.3 we will sketch how this approach can be generalized in order to allow an integrated optimization of both programs and database queries. Another major part is the design and implementation of a purely object-oriented version of the Tycoon Language TL called TooL (Tycoon Object-Oriented Language, section 3.4), and the development of an object-oriented bulk type library (section 3.5) as a case study in type-safe bulk type implementation with a strong emphasis towards reusability and incremental extensibility.

An evaluation of the research achievements and results follows in section 4. These are compared with the original research proposal and its objectives. We conclude with a description of project cooperation including a list of the jointly produced research papers and scientific publications.

2 Research Activities and Results: Formal Foundation

2.1 A Categorical Approach to Bulk Types

A primary goal of the project was to formulate general notions of bulk types and their processing abstractions. Starting with the first part, we developed an approach to a general and formal definition of what a bulk type is. It captures the intuition that a bulk type is a *container* type whose instances can store an arbitrary number of instances of some other type. Formally, a bulk type is a *type constructor* β (for example, β could be *set*), and instances of $\beta(\sigma)$ are collections of σ elements, with access operations as defined by β. Also, each instance of $\beta(\sigma)$ is *generated* from σ elements by some constructors of $\beta(\sigma)$. For example, every finite set is either the empty set, or is constructed from singleton sets, using union; thus *emptyset, single, union* is a set of generators for finite sets. Such a set of constructors forms a *presentation* of $\beta(\sigma)$. Finally, these instances need to obey some constraints, particularly that they do not 'forget' elements inserted into them (this is explained in more detail below).

We have also proposed appropriate mappings between bulk types, called *bulk morphisms*, and have shown that bulk types with these mappings form a category, with the bulk type *set* as a terminal object. These mappings model type conversions: when there exists a morphism from type α to type β, one can convert an $\alpha(\sigma)$-expression to a $\beta(\sigma)$-expression.

A family of bulk types related by conversion morphisms arises when one considers a free type, i.e. a type given by a presentation without equations, and bulk types obtained from it by adding equations, including those that use domain-specific functions or predicates. The most frequently used, and best understood, such family is that induced by the three operations presentation *empty, single, combine* (e.g., *emptyset, single, union* for sets). The free type in this case is *binary trees* (with data at the leaves); adding axioms of associativity, commutativity, and idempotence one obtains *lists, bags, sets*. Lists ordered by domain-specific total order predicate p are also

included in the family. One can convert expressions in this family as long as one goes down the hierarchy, from trees towards sets, or from a type that does not use p to one that does, such as from sets to ordered lists. The results, in the categorical setting, have appeared in [BT93]. Similar results, assuming associativity of *combine*, for the language of comprehensions (see below) have appeared in [FeMa95].

At this stage (the initial stage of the project) we concentrated on an abstract approach to bulk types, but we also invested some effort to understand the computational properties. In particular, each given presentation for a family of bulk types supports definitions by *structural recursion*, denoted *SR*. Although *SR* can express all the algebraic operations used in relational and more advanced database languages, as well as other useful operations such as aggregates, it is not a good programming primitive, since in the general case a function defined by *SR* may not be well-defined. This is a disadvantage if one is looking for processing abstractions for nonexpert users. We investigated the relationship between the equations added to a free type to define a bulk type, and the properties of the function arguments that suffice to guarantee uniqueness of a function defined by *SR*. In later stages of the research we concentrated on weaker primitives and approaches to query programming on bulk types, that do not suffer from this disadvantage.

2.2 Comparing Query Languages for Bulk Types

An abstract approach to bulk types, as outlined above in Section 2.1, has to be complemented by more concrete approaches to query and general programming mechanisms. We studied candidate languages, for the complex object model, in [Bee93]. This model is essentially a database model based on the *set* bulk type; it allows arbitrary nesting and interleaving of tuple and set constructors. The languages we studied are the predicate-calculus based and the pure algebra for complex objects developed by Beeri and Abiteboul in [AB95], the λ-calculus based algebra of Breazu-Tannen, Buneman and Wong [BBW92], and the *comprehensions* language, well known (for lists) in functional programming, and also studied in [BBW92].

One conclusion of our study is that both algebras and comprehensions are better than the predicate calculus language — the latter does not naturally support useful notations, for expression nesting for example. Further, algebras are good candidates for internal representations of queries, providing a firm basis for query optimization by rewrite rules, whereas comprehensions

are a good basis for user-level languages. In the comparison of the two algebras just mentioned, essentially no difference was found for many queries. The difference between the two versions is felt when there is a name-bind problem, when the same name from two different places in a nested object needs to be used. For queries that contain such phenomena, the λ-calculus based language provides a good solution. The algebra of [AB95], as it stands, suffers from some minor problems in this respect, but these can be amended, and then it also can serve as an internal representation for query optimization. An example is the work reported in [ChZd96] on optimizations in a language based on the same principles. The two languages are closely related to the classical λ-calculus and *combinatory logic*. These have the same expressive power, expressions in one can be translated to the other. One use variables to address the binding problem, while the other manages this task without any variables. This study in particular justified the significance of a good approach to understanding and specifying the intended bindings for names used in a query, see section 2.7.

Another interesting observation in that study was that the class of *conjunctive queries* known from the theory of relational databases can be generalized to the bulk type context. As in the relational context, it can be expressed in each of the three language paradigms. Of these, the ones that shed light on the structure of queries in the class are again the algebraic and the comprehension-based forms. In the algebraic forms, the standard relation product is replaced by a generalized form of *product*, that allows one to navigate, for example, attribute chains in object databases. In comprehensions, it is obtained as the expressions that do not use aggregates on bulk expressions (e.g., set emptiness test) in the comprehension body. The significance of this class is that many optimizations apply only to conjunctive queries, and in classical optimization techniques often a query is broken in to conjunctive subqueries that form units of optimization. Although there is a much richer universe of optimizations for bulk types, a similar approach can provide a useful heuristic.

2.3 A Hierarchy of Bulk Types

In addition to the categorical approach to bulk types, at least two approaches have been extensively discussed in the literature in the last two years. One is the monad-based approach initiated by Trinder and Wadler [WattTri], and pursued by Breazu-Tannen, Buneman and Wong [BBW92]. The other is the monoid homomorphism approach advocated by Fegaras and Maier

[FeMa95]. Both concentrate on the *empty, single, combine* presentation of bulk types mentioned above. Numerous other results about languages for lists, bags and sets have recently appeared in the literature. The following picture of this family of common bulk types is described in [Bee94].

Let us denote *empty* by $[\,]$, *single*, applied to an element a, by $[a]$, and *combine* by $\oplus$. There are many types that have these constructors. To qualify as a bulk type, one also needs some *iterator*, and as remarked above, *SR* is too general. Instead, a bulk type must support the simple iterator called *ext*, defined by structural recursion:

$$ext(f)(c) = \begin{cases} [\,] & \text{if } c = [\,] \\ f(a) & \text{if } c = [a] \\ ext(f)(c_1) \oplus ext(f)(c_2) & \text{if } c = c_1 \oplus c_2 \end{cases} \quad (1)$$

For example, on sets (where $[\,]$ is $\emptyset$, $[a]$ is $\{a\}$, and $\oplus$ is *union*), if $f : \sigma \to set(\tau)$, then $ext(f) : set(\sigma) \to set(\tau)$, takes a set S to $\cup\{f(s) \mid s \in S\}$. As mentioned above, in general, definitions by structural recursion may not be well-defined, and indeed $ext(f)$, for various functions f, may be ill-defined for a type constructor whose instance 'forget' the past (e.g., when a collection only retains the last element inserted into it). However, in contrast to the general case of a definition by *SR* where well-definedness has to be checked on a per-expression basis, there are cases where it needs be checked on a per-type basis, and *ext* is well-defined for *set, bag, list, binary tree, sorted-list* (and a few more less well-known types). The claim that the existence of a well-defined *ext* is what makes a type a bulk type is amply justified by observing that from it, the three constructors, and very weak assumptions about the base types, one obtains the expressive power of all query algebras discussed in the literature [BBW92].

Another important property is that given *ext*, the query language of *comprehensions* is definable. This language is a clean form of SQL, it supports nesting of expressions, and is extensible to support recursion and aggregates in a clean fashion. The semantics of comprehensions for a bulk type with *ext* is defined by:[1]

[1] In the literature, different kinds of brackets are used to distinguish between expressions of different types, e.g. $\{,\}$ for sets. Since comprehensions can be written for many types, including mixed type expressions, a more uniform notation is desirable, but is not considered here.

$$
\begin{array}{lcl}
[e \mid] & = & [e] \\
[e \mid p, e_2 \ldots] & = & \text{if } p \text{ then } [e \mid e_2 \ldots] \text{ else } [\,] \\
[e \mid x \leftarrow R, e_2 \ldots] & = & ext([e \mid e_2 \ldots])(R)
\end{array} \tag{2}
$$

In this family, *binary tree* is freely generated, and does not satisfy even the associativity requirement, required both in the monad and the monoid homomorphism approach. It is the root of the family. Other types are obtained by adding equations. All types in this hierarchy of bulk types, support two languages that include the following: Both include the three constructors, (1st-order) *lambda* abstraction, and function application. One contains additionally *ext*; the other contains *comprehensions* as the additional iteration primitive. A comprehension can express more complex iterations than *ext*, in particular it allows one to express simple queries without much use of other constructs. One can mix different bulk types in both *ext*-based and comprehension-based expressions, the rule of thumb for conversions being that types with less constraints can be converted to types with more. A third language, essentially a pure algebra related to combinatory logic, also exists for these types that satisfy additional assumptions, essentially the monad axioms.

An important feature of this hierarchy is that most types in it possess additional operations that can be performed efficiently; some of these can be expressed in the languages above, but some cannot be expressed by the core languages. The type *set*, the terminal object in the category seems to be the only one for which such additional operations of interest do not exist, as it contains no order or any structure. Some of these additional operations are relevant to physical organization and fast access; this is discussed briefly below in section 2.5. Another issue is that of bulk types with constraints, for which *ext* may be ill-defined for some expressions. This is discussed next.

2.4 Constrained Bulk Types

Above, we proposed that a bulk type is a container type for which $ext(f)$ is always well-defined, independently of f. However, there exist container types that we would like to incorporate into the bulk type framework, where this requirement is too stringent — $ext(f)$ is not always well-defined. Constrained types are a primary example. We call a type *constrained* if it is obtained from a bulk type by adding some constraints. These can be additional equations, or restrictions on the element type, or on the contents of instances. Consider, for example, the type *finite map*, where an instance is a

$1:m$ association between a finite set of keys and a set of values. A combination of two finite maps is well-defined only if it does not result in associating the same key to two values. Since $\oplus$ is only partially defined, the right side of the third equation in the definition of *ext* may be undefined. For example, assume given the map $M = \{(1, \text{John}), (2, \text{Jack}), (3, \text{Helen})\}$, and let f be the function $\lambda(x, y).(1, y)$. Then $ext(f)(M) = \{(1, \text{John}), (1, \text{Jack}), (1, \text{Helen})\}$, clearly not a finite map, only a set of pairs.

Despite that, *finite map* has been considered a bulk type in several works [ART90]. One possible reason is that being obtained from adding a constraint to *set*, the operation $ext(f)$ is always guaranteed to be well-defined, as a set operation; the issue is only whether it is a finite map operation as well. This is in contrast to situations where $ext(f)$ is not well-defined in any sense, and evaluating it in different orders on the same argument may produce different results.

Arrays are an important type in many engineering and scientific applications, and the need for providing appropriate database support for them has been keenly felt [MaVa93]. We have investigated arrays [BeCh96], and have shown that they can be understood as a constrained bulk type, actually as two different such types.

One obvious approach to arrays is as a further specialization of finite maps. More precisely, we start from sets, a bona fide bulk type, then successively restrict to sets of pairs, to finite maps as such sets with a key constraint, then to arrays as maps with cubes (products of intervals) over the integers as the key domain. The advantage of this approach is that it allows one to write operations that use both the index values and the element values in the arrays. However, whereas so far we dealt with bulk types with a single *combine* that is always well-defined, this is not the case for arrays — *combine* decomposes into separate combine operations along the array dimensions. Even more important, although *ext* can be used to express most array operations that involve array iterations, in many cases expressions involving *ext* are not well-defined array operations. We have shown how many well-known array operations can be expressed in terms of *ext*, discussed conditions that guarantee the well-definedness of *ext* expressions of various forms, and the complexity of checking or proving these conditions.

Another viewpoint on arrays is as a multi-dimensional generalization of sequences. For the operations that do not utilize the index values, and do not change the arrays structure, this is a viable viewpoint, and for such operations arrays can be viewed as generalizing both lists and vectors. The

issue of *ext* not being well-defined does not arise for this viewpoint.

Given the significance of types such as finite maps and arrays, it is certainly desirable to regard them as bulk types, taking into consideration the additional complications involved. Just what are these considerations? This is discussed next.

2.5 Derived and Direct Access Operations

Although the bulk type languages based on *ext* can express all common algebraic iterations on bulk type instances, the expressions are often, in a sense, too low level. For example, cross product of sets or lists, and relational join are not primitives, but rather derived operations. But each *ext*-based expression for one of these expresses a specific implementation, for example, a nested loop for a join. For optimization purposes, it is convenient to introduce additional, derived operations, into the algebraic language. This allows for a translation from a user-level language (e.g., comprehension-based) into a richer language that offers more opportunities for the selection of optimization strategies that may depend both on the type system used in an application and on the specific statistics of the collections in a database. A well-known example is the relational algebra where rather than using *ext* one uses special cases such as *project, select, join*. The same idea applies to bulk types in general. Type checking can usually infer that a given function is a predicate, thus isolating selections. Products of two or more instances are another derived operation that is clearly useful to have as a primitive. Introduction of other primitives may depend on the specific system being developed.

Arrays, as an example of a constrained type, provide another motivation. General *ext*-based or comprehension-based operations on arrays may be ill-defined. If programmers are provided with a reasonably rich repertoire of well-defined built-in operations, then in most cases they can use these operations, which are known to be well-defined. These possibly have also built-in translations into optimized expressions in the internal language. The (potentially dangerous) use of general comprehensions is restricted to special cases.

There is another aspect of this issue that deserves attention. One may want to use the bulk type framework to model not only the conceptual database, but also its physical organization. For example, one may be tempted to consider *B-tree* to be a bulk type. But this does not work.

Consider again arrays. Using the *ext*-based language, one can express selection, in particular selection by a given index value. That is, the access operation $M[i]$ can be expressed. However, the implied semantics of the iterator *ext* is that of iterating over all the collection. Nothing in the theory we have developed so far indicates that this operation can be performed in $O(1)$ time. Thus, the operation should be provided explicitly, and the language processor should know that it has a special fast implementation. Succinctly, this can be stated as follows: Direct access operations are used to implement iterators, but not vice versa. The idea of a bulk type does not model fast access hence does not allow one to model physical organization.

2.6 Concurrency and Recovery

Classical transaction management theory and practice was developed in the context of the relational model. With the advent of rich data models, there is a need to rework the theory and the implementation technology for such models. This means in particular that one has to deal with arbitrary operations defined in object interfaces, not only with *read, write*. For concurrency control, this has come to be known as *semantic concurrency control*, and is faily well understood, although it has not yet found its way into textbooks. The situation for recovery is very different. A clean, simple theory for transaction aborts and recovery from failures even for the simple read-write model has been lacking and has only partially been addressed in some recent papers.

We have developed a theory for recovery that is both simple and general. It combines recovery and concurrency into one framework, generalizes the graph-based characterization of correct concurrency (i.e., conflict serializability) to include recoverability and strictness, so that the correctness of combined concurrency and abort protocols can be shown. The theory deals with several types of inverse operations that are used to roll back operations of aborted transactions. In particular, note that database objects need to support also concurrency and recovery, thus their interfaces need to offer appropriate inverse operations for all regular operations. The theory can be used to understand the kinds of object interfaces that can reasonably and efficiently support concurrency and recovery, and to explain potential problems and difficulties.

2.7 A Unified Framework for Query and Programming Languages

As noted in the proposal, a major problem of current database technology is the lack of a well-designed, commercially available database programming language (DBPL). Such a language should allow a user to formulate queries as in a query language, and also to develop complete applications, programming as in a general-purpose programming language, including updates via assignments.

The development of a DBPL that addresses the needs of modern data intensive applications was a major goal of the project. Issues to be aware of and addressed in developing such a language include: The different paradigms of query and programming languages; the binding problems introduced in advanced models that allow nested structures and a rich collection of query constructs, since a name may be used both in different places in a structure, with different meanings, and in several places in a query; the declarative nature of queries, often implying parallel processing semantics, as compared to the procedural, sequential nature of general programming languages.

In the first year, we developed a framework for describing the operational semantics of languages for advanced data models that can be used for both query and general programming constructs. The major points of this work are the following:

1. We present a very simple data model and argue that for the purpose of describing the binding disciplines of data manipulation languages this model is sufficient, since other more sophisticated data models can be mapped to it in a way that preserves the essential nesting structure. The simplicity of the model allows a simple yet precise description of the framework; its universality means that the framework is applicable to more sophisticated and realistic data models.

2. The meaning of a program component depends very much on the bindings to the names of data and program entities appearing in it. While binding is a non-issue in the relational model, due to its simplicity, it becomes a major issue in advanced data models that allow nesting of structures and of query expressions. It is also a major issue in programming languages where names may appear in different scopes. Our framework addresses this issue by using a stack-based abstract machine for describing the semantics of language features. We show that a stack-based discipline captures the data-induced binding dis-

cipline typically found in query languages. It is well known that it captures the binding disciplines of general-purpose programming languages. Thus, we have a single framework for both queries and updates, procedure calls, and so on.

3. The declarative, set-oriented, nature of queries is captured by endowing the machine with both sequential and parallel processing capabilities. Although most current implementations of query languages are sequential, the semantics of set-oriented updates may depend critically on whether parallel or sequential processing is assumed. Since our machine has both parallel and sequential capabilities, the semantics of updates can be specified in it precisely.

With respect to the first point, we note that our model is what is currently known as a *light weight object model*. Current interest in this concept follows its use as a universal representation for the translation of queries and data from heterogeneous sources, with a variety of models. This work, [SBMS95], has enhanced our understanding of the relationships between query and general constructs in a data intensive programming environment.

3 Research Activities and Results: Enabling Technology

3.1 Exploiting Persistent Program Representations

The traditional focus of database language research has been on high-level languages for data access and manipulation (query languages, trigger definition languages, 4th generation languages, script languages) or on implementation-oriented languages which capture the characteristic operations of a specific target system at hand (relational algebra, object algebra, structural recursion, ...).

A closer look at tools working on such code representations like query and program optimizers reveals a large number of common tasks which at present are addressed with often incompatible technologies:

- Binding analysis: Which entity (table, index, view, function, method, variable, etc.) is denoted by an identifier? Are there multiple references to the same entity?

- Identifier substitution by a bound value or expression: View expansion, procedure or method inlining, constant folding, substitution of host programming language parameters, etc.
- Free variable analysis: Does a variable appear in a query predicate? Does a procedure depend on global variables? Does a query contain programming language variables? Are there independent subexpressions? Which base relations appear inside an integrity constraint?

We made the radical decision to replace special-purpose representations for queries, programs and scripts with a single, expressive *intermediate* language, the Tycoon Machine Language TML [GaMa94]. TML is used for local compile-time as well as global run-time optimizations in the Tycoon system. We thereby avoid incompatibilities and redundancies arising from the repeated implementation of the above functionality.

TML provides the basis for integrating the analysis and rewriting of queries and programs and, therefore, to overcome the limitations resulting from the traditional separation between program and query optimization mentioned above. The development of TML was influenced heavily by continuation passing style (CPS) representations found in modern optimizing compilers for functional, imperative and object-oriented languages [Appe92, KKRHPA86, Kels89, Gawe92].

TML inherits the advantages of CPS representations which support a wide range of algorithmically-complete languages, multiple front-ends and back-ends and cross-language optimization. To address the specific needs of database environments, TML also supports optimizations based on run-time bindings to arbitrary complex values in the persistent store and mechanisms to work with persistent TML terms attached to executable code.

We have organized the TML optimizer into two separate passes, namely a *reduction* pass and the *expansion* pass. During the reduction pass, a number of generic rewrite rules are applied to the TML tree until no more rules are applicable. Termination is guaranteed because each of the rewrite rules reduces the size of the TML tree if it is applied.

The subsequent expansion pass tries to substitute bound λ-abstractions (procedures or continuations) at the positions where they are applied. Effectively, this CPS transformation performs *procedure inlining* in terms of traditional compiler optimization or view expansion in database terminology. The decision whether a given use of a bound abstraction is to be substituted is based on a heuristic cost model similar to the one described by [Appe92].

When one or more abstractions are substituted during the expansion pass, there usually is the opportunity to perform more reductions on the TML tree (this is indeed the main reason why inlining is performed in programming languages at all), so each expansion pass is followed by a reduction pass. Likewise, the reduction pass may reveal new opportunities to perform expansions, so the two passes are applied repeatedly until no more changes are made to the TML tree. To guarantee the termination of this process even in obscure cases, a penalty is accumulated at each round of the reduction/expansion phases. The optimization process stops when this penalty reaches a certain limit.

3.2 Optimization across Abstraction Barriers

Today's applications are constructed incrementally with heavy re-use of modular software components defined in shared program libraries or application frameworks. At the same time, many binding decisions are delayed until run-time. Abstraction barriers (module interfaces, class interfaces, schema layers) severely restrict the binding information available to local static program optimizers which become less effective with increasing modularization.

Effective optimization of highly modular languages and of database languages therefore requires the analysis and rewriting of CPS terms which have been declared and compiled in separate scopes (logical schema, physical schema, query modules, embedded query, application program), at different times and most often by different users.

Given a uniform representation of programs and queries, the "trick" to eliminate abstraction barriers is (1) to wait until link or execution time, when all the bindings between the contributing parts of a persistent application are established (database schemata, application modules, program libraries, program parameters, etc.), and (2) to keep sufficiently abstract code and binding information until that point in time. Based on this approach it is rather straightforward to collect (via transitive reachability) all declarations which contribute to a given TML term (for example an embedded query) into a single scope (represented again as a TML term) and to invoke the TML optimizer to generate a globally optimized TML term.

Since the compiler (and, therefore, the optimizer) is an integral part of the Tycoon persistent programming environment, it is not difficult to call the Tycoon compiler at run-time.

For each exported source code function `f` in a compilation unit, the compiler back end augments the generated code for `f` with a reference to a com-

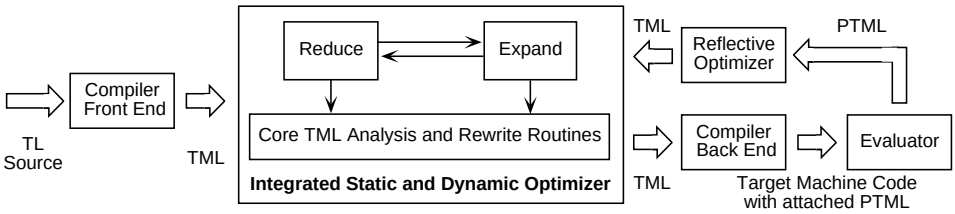

Figure 1: Interaction between compilation, optimization and evaluation in the Tycoon system

pact persistent representation of the TML tree (Persistent TML, PTML) for `f`. At run-time, it is possible to map PTML back into TML, re-invoke the optimizer and code-generator, link the newly-generated code into the running program, and execute it (Fig. 1).

The mapping from PTML back to TML also returns the set of R-value bindings established at run-time. These bindings correspond to free variables (module names, database names, table names, function names, constant names, etc.) in the source text and they naturally give rise to context-dependent, inter-procedure and inter-module optimizations (*optimization across abstraction barriers*).

To speed up repeated optimizations of (shared) functions, the optimizer attaches several derived attributes (costs, savings, ...) to the generated code which also become part of the persistent system state.

3.3 Towards Integrated Program and Query Optimization

There is a strong interest in improving the interface between query languages and programming languages. For example, the ODMG standard document explicitly states that "object database management systems provide an architecture which is significantly different than other DBMSs – they are a revolutionary rather than an evolutionary development. Rather than providing only a high-level language such as SQL for data manipulation, an ODBMS transparently integrates database capability with the application programming language" [Cate94].

Following this rationale, the syntax of many modern query languages allows programming language variables, function and method calls to appear in the **select** and **where** clauses of SQL statements. Furthermore, the

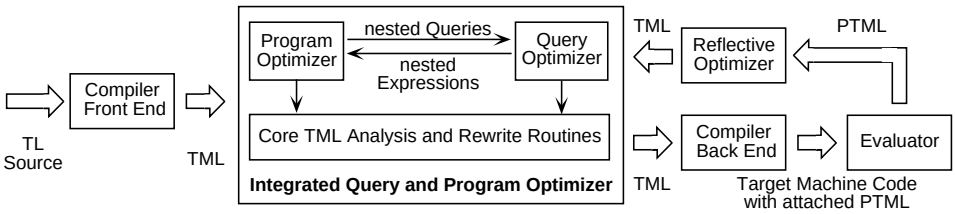

Figure 2: Embedded Query Optimization

body of element-at-a-time iterators (**for each** statements) and of database triggers may refer to programming language statements. User-defined data types lead to further interaction between query expressions and programming language expressions.

Since traditional query optimizers do not have access to an abstract representation of these program fragments, they have to work under worst-case assumptions (dependencies between subexpressions, side-effects) or they have to rely on programmer-supplied information (commutativity, idempotence, side-effects) which is difficult to keep consistent in large, long-lived systems. In particular, current query representations do not cover any form of control flow (conditionals, case analysis, loops, exceptions) which is "inherited" through embedded programming language expressions.

Given an integrated database language where user-defined code and query expressions are fully integrated [MS91a], query and program optimization have to interact closely (see Fig. 2): Whenever the program optimizer encounters an embedded query construct like a set-at-a-time (bulk) query or update, an element-at-a-time iterator, or a view definition, it invokes the query optimizer on the respective TML subtree with a TML environment which describes the global bindings (free variables) for that expression. Similarly, the query optimizer invokes the program optimizer to analyze and optimize nested programming language expressions which appear in query constructs (target list, selection predicate, iterator body). Again, binding information for free variables (e.g., range variables in queries or loop control variables in **for each** iterators) is passed along with the respective TML subtree. Recursive declarations of functions, values, or queries are represented uniformly through applications of the fixpoint combinator Y and do *not* lead to repeated traversals of TML terms.

In general, since the optimization of query expressions depends on run-time bindings (for example, knowledge about index structures), we have to delay query optimizations until run-time as described in the previous section. The translation of a declarative query construct embedded in the source language into a TML term is rather straightforward and resembles the usual approach of mapping a relational query 1:1 into a tree of algebraic operators [Ullm89].

For example, the SQL statement

```
select Target(x) from Rel x where Pred(x)
```

can be represented by the following TML term which uses the primitive procedures project and select as defined by the relational algebra:

```
(select λ(x ce cc) (Pred x ce cc)
  Rel
  ce
  cont(tempRel)
    (project λ(x ce cc) (Target x ce cc)
      tempRel
      ce
      cc))
```

The scope of the SQL correlation variable x is captured in TML by having two λ-abstractions with the bound variable x in addition to the two continuation variables ce and cc. The data dependency between the selection and projection is made explicit by introducing a named variable tempRel in the continuation for the selection which is then used as an argument to the projection. Since the variable ce which describes the current exception handler is simply passed through, exceptions which are raised during selection or projection are propagated to the enclosing block.

As can be seen in the simple example above, TML focuses on data and control dependencies, but leaves much freedom in the choice of the particular primitive procedures to be used for the representation of declarative queries. Instead of relational algebra operators, more general operators can be utilized. For example, the higher-order-functions proposed for the optimization of generalized queries over multiple bulk types in [Trind91, BBW92, Fega94].

For a given set of primitive procedures, algebraic and implementation-oriented query optimization rules can be expressed quite naturally in CPS [GaMa96a].

3.4 TooL: A Language for Improved Library Design

Several shortcomings of module-based languages have been recognized during several years of experience with the Tycoon language TL [MS92]: modularization and encapsulation within abstract data types is well suited for layered system architectures such as the DBPL run-time system, but of limited expressiveness concerning bulk type abstractions and implementations [LoRo96].

TL, as a statically-typed modular polymorphic programming language with subtyping, provides application programmers with rich re-usable generic class libraries organized into subtype hierarchies. However, the type system of TL obstructs programmers who intend to maximize code sharing between library classes through implementation inheritance following the successful library design principles of Smalltalk and Eiffel. As discussed in the literature [Bruc94, Bruc95, AbCa95], a more liberal notion of *type matching* is needed, for example, to support the type-safe inheritance of *binary methods* [BCC*95].

Based on our extensive experience using TL for large-scale programming (for example, building and maintaining systems with several hundred modules) we designed a purely object-oriented version of TL called TooL (Tycoon Object-Oriented Language). Our motivation behind the design of TooL was to verify the following hypotheses: (1) A *purely* object-oriented language (where objects and classes combine aggregation, encapsulation, recursion, parameterization and inheritance) leads to program libraries which are more uniform and easier to understand since programmers do not have the freedom to choose between combinations of modules, records, tuples, functions, recursive declarations, etc. (2) Type matching increases code reuse within complex libraries.

To verify these hypotheses it was not only necessary to design and to implement TooL, but also to *utilize* it for non-trivial library examples. In a nutshell, this experimental validation was carried out by augmenting the Tycoon type system by a notion of type matching, omitting existential type quantification and all higher-order type concepts like kinds, but otherwise adhering closely to the proven type and language concepts of Tycoon. We then used the functionality of the mature and highly-structured Eiffel collection library [Meyer90] as a yardstick for the construction of a type-safe TooL bulk type library.

The key aspects of the TooL language design can be summarized as follows:

Purely object-oriented: TooL supports the classical object model where objects are viewed as abstract data types encapsulating both state and behavior. Similar to Smalltalk [GoRo83] and Self [UnSm87], TooL is a *purely* object-oriented language in the sense that *every* language entity is viewed as an object and *all* kinds of computations are expressed uniformly as (typed) patterns of passing messages [Hewi77]. Even low-level operations such as integer arithmetic, variable access, and array indexing are uniformly expressed by sending messages to objects.

It should be noted that modern compiler technology eliminates most of the run-time performance overhead traditionally associated with the purely object-oriented approach [Hoel94, Gawe92]. TooL exploits dynamic optimization across abstraction barriers as described in the previous section to "compile away" many message sends.

Higher-order functions as objects: Contrary to other statically-typed object-oriented languages [Gogu90], TooL provides statically-scoped higher-order functions which are viewed as first-class objects that understand messages. Thereby, control structures like loops and conditionals do not have to be built into the language, but can be defined as add-ons using objects and dynamic binding. To improve code reusability, even instance and pool variables (which unify the concepts of global and class variables in Smalltalk) are accessed by sending messages [JoFo88]. This unification at the value level leads to a significant complexity reduction at the type level where it is only necessary to define type and scoping rules for class signatures, message sends and inheritance clauses.

Strong and static typing: No operation will ever be invoked on an object which does not support it, i.e. errors like "message not understood" cannot occur at run-time. Type rules are defined in a "natural-deduction" style based on the abstract TooL syntax similar to [MTH90] and [MS92].

Structural type checking: Several conventional object models couple the implementation of an object with its type by identifying types with class names (e.g. C++, ObjectPascal, Eiffel). In these models, an object of a class named A can only be used in a context where an object of class A or one of its statically declared superclasses is expected. This implies that type compatibility is based on a single inheritance lat-

tice which is difficult to be maintained in a persistent and distributed scenario.

Therefore, TooL has adopted a more expressive notion of type compatibility based on *structural subtyping*, called *conformance*. Intuitively, an object type A is a subtype of another object type B when it supports at least the operations supported by B. That is, TooL views types as (unordered) sets of method signatures, abstracting from class or type names during the structural subtype test. The additional flexibility of structural subtyping is especially useful if A and B have been defined independently, without reference to each other. Such situations occur in the integration of pre-existing external services, in the communication between sites in distributed systems [BNOW93], and on access to persistent data.

Modular type checking: During type checking of a given class only the interfaces of imported classes and of superclasses have to be accessed. In particular, it should be possible to type-check new subclasses without having to re-check method implementation code in superclasses again. Modular type checking speeds up the type-checking process significantly, thus supporting rapid prototyping within an incremental programming environment. It also has the advantage that class libraries – developed independently by different vendors – can be delivered in binary form without a representation of their source code with the option of type-safe subclassing at the customer side.

Modular type-checking requires the *contravariant* method specialization rule for soundness, which means that the types of method arguments are only permitted to be generalized when object types are specialized. The contravariant rule has been criticized as being counter-intuitive [Mey89]. Accordingly, Eiffel has adopted a *covariant* method specialization rule which is in conflict with substitutability. Therefore, Eiffel requires some form of global data flow analysis at link-time to ensure type correctness. Such an analysis generally requires a representation of the source code of all classes and methods which constitute the whole program to be available to the type-checker at link-time which is not acceptable in our setting. TooL provides a partial solution to the covariance/contravariance problem without giving up modular type-checking by adopting the notion of *type matching* which allows the covariant specialization of method arguments in the important

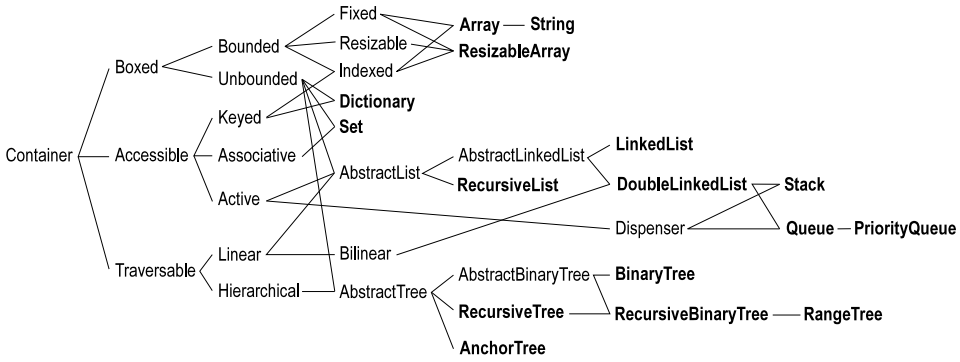

Figure 3: Overview of the TooL bulk type library

special case where the argument type is equal to the receiver type.

TooL minimizes built-in language functionality in favor of flexible system add-ons, both at the level of values and at the level of types.

3.5 The TooL Bulk Type Library: An Overview

On top of TooL we designed and implemented an object-oriented bulk and stream (iterator) library [LoRo96] exploiting heavily the inheritance of method specifications (including pre- and postconditions) as well as method implementations.

Figure 3 shows the part of the new TooL class library relevant for bulk data manipulation. A bold font indicates concrete (instantiable) classes, whereas a non-bold font indicates abstract classes.

This class hierarchy is designed to maximize code sharing between library classes through implementation inheritance following the successful library design principles of Smalltalk and Eiffel [Meyer90]. For example, the abstract classes Boxed, Accessible, and Traversable provide code fragments which capture certain aspects of containers and which can be inherited, combined and refined in subclasses to implement concrete classes such as priority queues.

A key feature of TooL is the ability to write *generic classes* by type parameterization. Type parameterization is especially important for the type-safe definition of generic container classes and polymorphic iteration

abstractions. For example, virtually all subclasses of Container (see figure 3) have one or several formal type parameters.

In the following example, the element type of sets is parameterized, but constrained to be a subtype of Object in order to allow some basic messages to elements (e.g., comparisons for object identity and printing):

```
class Set(E <: Object)
  add(e :E) :Void
  includes(e :E) :Bool
  inject(F <: Object, unit :F, f :Fun(:F, :E):F) :F
  map(F <: Object, f :Fun(:E):F) :Set(F)
  printOn(aStream :WriteStream(Char))
```

This class interface shows that type parameterization is also available in individual method and function signatures, for instance in the higher-order inject method which iterates over all elements of type E within the set, accumulating the values computed by a binary user-specified function f on arguments of type F and E, given an initial value unit of type F.

In TooL, parameterized classes are not simple templates which can only be type-checked after instantiation as in C++ or in Trellis. Type parameters are *bounded* by a type, permitting local, modular type checking within the scope of the quantifier. For example, the element type of the set returned by the polymorphic map method depends on the argument type of the function f passed as a run-time argument to the map function.

As indicated by the example above, TooL incorporates the full power of bounded parametric polymorphism as found in $F_{<:}$ [CMMS91].

TooL provides type argument synthesis in message sends and function applications, thus intSet.inject(0, plus) is equivalent to intSet.inject(:Int, 0, plus). This is particularly useful in the typing of control structures modeled with message passing. We use a simple but incomplete inference algorithm similar to the one described in [Card93] which works well in practice.

First-class functions do not introduce additional complexity at the type level since they are treated as objects supporting an apply method that captures the function signature. This also scales to polymorphic and higher-order functions.

4 Evaluation of the Research Achievements in Relation to the Original Research Proposal and its Objectives

The objectives of the research, as described in the original proposal, were:

- to develop the foundational theory of the data structuring facilities that are relevant to databases and to data-intensive applications, dealing with both type structures and processing abstractions.
- to develop the enabling technology for the implementation of systems for the development of data-intensive applications.

We feel that both objectives have been fully satisfied, although some work remains to be done to exploit in their entirety the ideas and technologies developed during these three years.

4.1 Formal Foundation

As explained in detail in section 2, the more theoretical direction of the research was meant to provide the underlying theory for bulk types both in terms of structure and processing abstractions. Having examined various proposals for the notion of bulk type, we now have a satisfactory theory that explains what a bulk type is, relates bulk types in several ways, and organizes bulk types into a hierarchy. We have shown that in addition to being a container type, there is just a little more that a type needs to satisfy to be regarded as a bulk type — that it supports a simple iterator. Even this requirement can be somewhat relaxed, as shown in the study of constrained types. The hierarchy is intimately related to the availability of processing abstractions and query languages. All bulk types, by definition, support two simple query languages. As additional properties are added, it becomes possible to add a third, variable-free language, that offers a different approach to query optimization. Additional properties can also be used directly in rewrite-based optimizations.

Our research also shed light on how additional types that may not fully satisfy the stringent requirement of full bulk types can be added to the framework. A crucial notion here is that of derived operations which allows an implementor to offer a rich set of operations suitable for an application and its specific types, together with translations into the general framework of bulk types, thus taking advantage of the general facilities for application

structuring, query optimization and transaction management available in the underlying system.

We also gained a much better understanding of the requirements posed by transaction management on advanced data-intensive applications. This should prove valuable in the development of advanced transaction facilities that offer the flexibility of a variety of data structuring mechanisms, long lived and nested transactions, with the security of concurrency control and recovery.

4.2 Enabling Technology

In the development of the technological infrastructure we made progress both on the language design level and the language implementation level.

On the language implementation level we built reusable TML analysis and rewrite tools to carry out the core tasks in symbolic code manipulation like binding analysis, identifier substitution, and free-variable analysis. The current version of the Tycoon system fully implements dynamic reflective optimization across abstraction barriers based on CPS representations. In particular the static and dynamic optimizers share the same code for TML analysis and rewriting. As described in more detail in [Kira94], performing local program optimizations on standard benchmarks for imperative programs (the Stanford Suite) do not yield a significant speedup in the Tycoon database programming language. The reason for this is the fact that even operations on integers and arrays are factored out into dynamically bound libraries and therefore not amenable to local optimization. However, a move to dynamic (link-time or run-time) optimization more than doubles the execution speed of the standard benchmarks as well as of most larger Tycoon programs we have experimented with (including the compiler itself, consisting of 98 modules containing more than 29,000 lines of high-level Tycoon code).

More work is required to evaluate the effectiveness of query optimization exploiting the availability of a uniform program and query representation at run-time. We are also very interested in exploiting TML for other tasks in data-intensive applications, such as code shipping in distributed systems [MMS95b], synchronization of persistent threads [MaSc94], access control and security issues [RMS95].

On the language design level, we integrated type matching and subtyping orthogonally and cleanly into a fully-fledged, practical programming language. We sought to assess the impact of the increased type system ex-

pressiveness gained by the introduction of matching on the practical value of the TooL language.

First, subtyping and type matching both interact well with other TooL language concepts such as type quantification and Self type constraints. Moreover, students with some background in strongly-typed higher-order programming languages grasp these concepts rather fast. However, problems arise as soon as programmers wish to combine the advantages of both partial orders on types. For example, library designers typically prefer match-bounded quantification to maximize code reuse. This may conflict with the goal of library clients which like to exploit the subsumption property of subtyping to absorb later (unforeseen) system extensions. A similar argument holds for type parameters of classes.

A comparison of the TooL class library with the Tycoon bulk type library supports both our hypotheses stated at the beginning of section 3.4: TooL as a purely object-oriented language with type matching leads to more uniform program libraries with an increased code reuse. However, we also observed some practical difficulties encountered by programmers designing large libraries involving both matching and subtyping.

5 Project Cooperation

The project was begun with a three month visit by Prof. Beeri to Hamburg Univeristy prior to commencement of the project. During this visit, we were able to discuss the various goals of the project, various approaches, modes of cooperation, and start some technical work. In particular, during this visit, initial ideas were formulated regarding the possible elements of a bulk type interface for Tycoon, and a small implementation was carried out. We also had comprehensive discussions regarding the fusion of query and programming approaches, which later led to the development of a unified approach to the semantics of both. Additional meetings during 1993 in two workshops, and in another visit to Hamburg, helped to examine the work done on the initial ideas developed in the first visit.

In 1994 a prolonged meeting took place in Hamburg during which we examined the various ideas and approaches to bulk types. This resulted in an approach to classifications of bulk types, and the proposal to generate a catalog. Additional ideas regarding possible extensions to bulk types with object structures were examined.

In 1995, a meeting was held to summarize the approach to a bulk type in-

terface design which allows the flexibility required for modern data-intensive applications.

The differing expertises brought into the project by the two groups proved to be very fruitful and a key factor in the success of the project. The abstract approach of the Hebrew University team helped influence the overall interface design at Hamburg. The extensive implementational experience of the Hamburg team, and the powerful and flexible programming environment they implemented, generated questions that tied together the theoretical and practical aspects of the research

6 Project Publications and Visits

6.1 Visits

- Three-month visit by Prof. Beeri to Hamburg Univeristy prior to commencement of the project for close identification of project goals. August 2, 1992 to October 28, 1992.
- Bulk Type Workshop, Bellcore, New Jersey. Participants: Beeri, Schmidt, Matthes (UHH). February 2 to 6, 1993
- Leibniz Workshop, Jerusalem. Participants: Beeri (Organizer), Schmidt. March 20 to 26, 1993
- Prof. Beeri visit to Hamburg. Project discussions with Hamburg group: P. Muennix, I. Wetzel, D. Juhasz. April 30 to May 5, 1993. A main goal of this visit was to commence work on a bulk type interface design.
- Database Programming Languages Workshop (No. 4), New York. Participants: Beeri, Schmidt, Matthes, P. Ta-Shma (Ross) (HUJ), August 27 to 31, 1993.
- Visit to Hamburg, Aug. 15–18, 1994. Participants: Prof. Beeri, P. Ta-Shma, Prof. Schmidt, Dr. Matthes. Purpose: Annual examination of research results; preliminary discussion of topics for third year, particularly the bulk type catalog.
- Visit to Hebrew University, June 2nd. Participants: Prof. Schmidt, Dr. Matthes, Prof. Beeri, P. Ta-Shma. Purpose: To examine and

decide on principal ideas regarding bulk types and their interfaces. particularly regarding the role of derived operations.

6.2 Publications and Technical Reports

- C. Beeri. Query languages for models with object-oriented features. In *Nato ASI summer school on object-oriented database systems.* Springer-Verlag, August 1993, to appear Springer-Verlag NATO ASI Series, 1994.

- C. Beeri., P. Ta-Shma (Ross) Bulk data types, a theoretical approach. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhatten, New York*, Workshops in Computing. Springer-Verlag, February 1994.

- C. Beeri. Bulk types and query language design. In *Proc. 10th Abstract Data Types Workshop (ADT94)*, LNCS. Springer-Verlag, 1994.

- C. Beeri and D. Chan. Bounded arrays: A bulk type perspective. Submitted for publication.

- C. Beeri. Concurrency and recovery for object bases. manuscript in preparation, March 1996.

- L. Cardelli, F. Matthes, and M. Abadi. Extensible Grammars for Language Specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhatten, New York*, Workshops in Computing. Springer-Verlag, February 1994.

- A. Gawecki and F. Matthes. The Tycoon Machine Language TML - an optimizable persistent program representation. FIDE Technical Report FIDE/94/100, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.

- A. Gawecki and F. Matthes. Exploiting Persistent Intermediate Code Representations in Open Database Environments. In *Proceedings of the 5th Conference on Extending Database Technology, EDBT'96*, Avignon, France, March 1996. Springer-Verlag (to appear).

- A. Gawecki and F. Matthes. Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, Linz, Austria, July 1996. Springer-Verlag (to appear).

- F. Matthes, R. Müller, and J.W. Schmidt. Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, final Version March 1993.

- F. Matthes and J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, revised March 1993.

- F. Matthes and J.W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

- J.W. Schmidt and F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, pages 2–16, Vienna, Austria, April 1993.

- J.W. Schmidt and F. Matthes. The DBPL Project: Advances in Modular Database Programming. Information Systems 19(2), pages 121–140, 1994.

- K. Subieta, C. Beeri, F. Matthes, and J.W. Schmidt. A Stack-Based Approach to Query Languages. In *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing. Springer-Verlag, 1995. Also appeared as FIDE Technical Report FIDE/95/134, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.

6.2 References

[AB95] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. *VLDB Journal*, 1996.

[AbCa95] Abadi, M. and Cardelli, L. On Subtyping and Matching. In *Proceedings ECOOP'95*. Springer-Verlag, 1995.

[AGO89] A. Albano, G. Ghelli, and R. Orsini. Types for Databases: The Galileo Experience. In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[ART90] M.P. Atkinson and P. Richard and P.W. Trinder. Bulk Types for Large Scale Programming. *Proc. 1st Int'l East/West Database Workshop on Next Generation Information System Technology*, Kiev, USSR, Oct. 9 –12, 1990, lncs 504, 1991, 229–250.

[Appe92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[BBN] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *DBPL*, 1991.

[BBW92] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In Biskup and Hull, editors, *ICDT*, Berlin, Germany, October 1992. Springer-Verlag. LNCS 646.

[Bee93] C. Beeri. Query languages for models with object-oriented features. In *Nato ASI summer school on object-oriented database systems*. Springer-Verlag, August 1993, to appear Springer-Verlag NATO ASI Series, 1994.

[Bee94] C. Beeri. Bulk types and query language design. In *Proc. 10th Abstract Data TypesWorkshop (ADT94)*, LNCS. Springer-Verlag, 1994.

[BeCh96] C. Beeri and D. Chan. Bounded arrays: A bulk type perspective. Submitted for publication.

[Bee96] C. Beeri. Concurrency and recovery for object bases. manuscript in preparation, March 1996.

[BCC*95] Bruce, K.B., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G.T., and Pierce, B. On binary methods. Technical report, DEC SRC Research Report, 1995.

[BNOW93] Birell, A., Nelson, G., Owicki, S., and Wobber, E. Network objects. In *14th ACM Symposium on Operating System Principles*, pages 217–230, June 1993.

[Bruc94] Bruce, Kim B. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994.

[Bruc95] Bruce, K.B., Schuett, A., and Gent, R. van. PolyTOIL: a type-safe polymorphic object-oriented language. In *Proceedings ECOOP'95*. Springer-Verlag, 1995.

[BT93] C. Beeri., P. Ta-Shma (Ross) Bulk data types, a theoretical approach. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhatten, New York*, Workshops in Computing. Springer-Verlag, February 1994.

[Bro89] A.L Brown. Persistent Object Stores. PPRR 71-89, Universities of Glasgow and St Andrews, March 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Technical report, Digital Equipment Corporation, Systems Research Center, Palo-Alto, California, 1990. (shipped as part of the Quest V.12 system distribution).

[Card93] Cardelli, L. An implementation of $F_{<:}$. Technical Report 97, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1993.

[Cate94] Catell, R.G.G., editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, 1994.

[ChZd96] M. Cherniak and S.B. Zdonik. Rule languages and internal algebras for rule-based optimizers. to appear, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.

[CMA94] L. Cardelli, F. Matthes, and M. Abadi. Extensible Grammars for Language Specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhatten, New York*, Workshops in Computing. Springer-Verlag, February 1994.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An Extension of System F with Subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software, TACS'91*, Lecture Notes in Computer Science, pages 750–770. Springer-Verlag, 1991.

[FeMa95] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–58. ACM Press, 1995.

[Fega94] Fegaras, L. Efficient optimization of iterative queries. In Beeri, C., Ohori, A., and Shasha, D.E., editors, *Database Programming Languages, New York City, 1993*, Workshops in Computing, pages 200–225, 1994.

[Gawe92] Gawecki, A. An optimizing compiler for Smalltalk. Bericht FBI-HH-B-152/92, Fachbereich Informatik, Universität Hamburg, Germany, September 1992. In German.

[GaMa94] Gawecki, A. and Matthes, F. The Tycoon Machine Language TML - an optimizable persistent program representation. FIDE Technical Report FIDE/94/100, Fachbereich Informatik, Universität Hamburg, Germany, July 1994.

[GaMa96a] A. Gawecki and F. Matthes. Exploiting Persistent Intermediate Code Representations in Open Database Environments. In *Proceedings of the 5th Conference on Extending Database Technology, EDBT'96*, Avignon, France, March 1996. Springer-Verlag. (to appear).

[GaMa96b] A. Gawecki and F. Matthes. Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In *Proceedings of the 10th European Conference on Object-Oriented*

Programming, ECOOP'96, Linz, Austria, July 1996. Springer-Verlag. (to appear).

[Gogu90] Goguen, J.A. Higher-order functions considered unnecessary for higher-order programming. In Turner, D., editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Wesley Publishing Company, 1990.

[GoRo83] Goldberg, Adele and Robson, David. *Smalltalk 80: the Language and its Implementation*. Addison-Wesley, May 1983.

[Hewi77] Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[Hoel94] Hölzle, U. *Adaptive Optimization for Self: Reconciling high performance with Exploratory Programming*. PhD thesis, Stanford University, August 1994.

[JoFo88] Johnson, Ralph E. and Foote, Brian. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[Kels89] Kelsey, R.A. Compilation by program transformation. Technical report, Yale University, Department of Computer Science, May 1989.

[Kira94] Kiradjiev, P. Dynamic optimization in CPS-oriented intermediate languages. Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, December 1994.

[KKRHPA86] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. ORBIT: an optimizing compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986.

[LR89] C. Lécluse and P. Richard. The O_2 Database Programming Language. Rapport Technique 26-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex, France, January 1989.

[LoRo96] Lotter, Björn and Römer, Thorsten. Entwurf und Realisierung von Softwarebibliotheken: Massendaten im Tycoon-System. Master's thesis, Fachbereich Informatik, Universität Hamburg, Germany, March 1996.

[MaVa93] David Maier and Bennet Vance. A call to order. In *pods93*, pages 1–16.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

[Mey89] Meyer, B. Static typing for Eiffel. (Technical report distributed with Eiffel Release 2), July 1989.

[Meyer90] Meyer, B. Lessons from the design of the eiffel libraries. *Communications of the ACM*, 33(9):69–88, September 1990.

[MM93] F. Matthes and S. Müßig. The Tycoon Language TL: An Introduction. DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, Germany, December 1993.

[MMS92] F. Matthes, R. Müller, and J.W. Schmidt. Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, final Version March 1993.

[MMS95b] Mathiske, B., Matthes, F., and Schmidt, J.W. Scaling database languages to higher-order distributed programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy.* Springer-Verlag, September 1995. (Also appeared as TR FIDE/95/137).

[Mos89] J.E.B. Moss. The Mneme Persistent Object Store. COINS Technical Report 89-107, University of Massachusetts at Amherst, October 1989.

[MaSc94] Matthes, F. and Schmidt, J.W. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

[MS91a] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

[MS91b] F. Matthes and J.W. Schmidt. Towards Database Application Systems: Types, Kinds and Other Open Invitations. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991. (Also appeared as TR FIDE/91/14).

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, revised March 1993.

[MS93a] F. Matthes and J.W. Schmidt. DBPL: The System and its Environment. In M. Jarke, editor, *Database Application Engineering with DAIDA*, volume 1 of *Research Reports ESPRIT*, pages 319–348. Springer-Verlag, 1993.

[MS93b] F. Matthes and J.W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

[MTH90] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[NMM92] C. Niederée, S. Müßig, and F. Matthes. P-Quest User Manual. DBIS Tycoon Report 102-92, Fachbereich Informatik, Universität Hamburg, Germany, February 1992. (In German.).

[Ric89] J.E. Richardson. E: A Persistent Systems Implementation Language. Technical Report 868, Computer Sciences Department, University of Wisconsin-Madison, August 1989.

[RMS95] Rudloff, A., Matthes, F., and Schmidt, J.W. Security as an add-on quality in persistent object systems. In *Second International East/West Database Workshop*, Workshops in Computing. Springer-Verlag, 1995.

[SBMS95] K. Subieta, C. Beeri, F. Matthes, and J.W. Schmidt. A Stack-Based Approach to Query Languages. In *Second International East/West Database Workshop*, Workshops in Computing. Springer-Verlag, 1995. (Also appeared as TR FIDE/95/134).

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.

[SM90] J.W. Schmidt and F. Matthes. Language Technology for Post-Relational Data Systems. In A. Blaser, editor, *Database Systems of the 90s*, volume 466 of *Lecture Notes in Computer Science*, pages 81–114, November 1990.

[SM91] J.W. Schmidt and F. Matthes. Naming Schemes and Name Space Management in the DBPL Persistent Storage System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems, Martha's Vineyard, Massachusetts.* Morgan Kaufmann Publishers, January 1991.

[SM93] J.W. Schmidt and F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering, Interoperability in Multidatabase Systems*, pages 2–16, Vienna, Austria, April 1993.

[SM94] J.W. Schmidt and F. Matthes. The DBPL Project: Advances in Modular Database Programming. (to appear in Journal 'Information Systems'), 1994.

[SMV93] J.W. Schmidt, F. Matthes, and P. Valduriez. Building Persistent Application Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability. In *Proceedings of Euro-Arch'93 Congress*, pages 270–287. Springer-Verlag, October 1993.

[Trind91] Trinder, P. Comprehensions, a query notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece.* Morgan Kaufmann Publishers, September 1991.

[Ullm89] Ullman, J.D. *Database and Knowledge-Base Systems, vol. 2.* Computer Science Press, 1989.

[UnSm87] Ungar, D. and Smith, R.B. Self: The power of simplicity. In *Proceedings of the Object-Oriented Programming Systems, Lan-*

guages and Applications Conference, Orlando, Florida, pages 227–242, 1987.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. Rapport Technique 27-89, GIP Altair, Domaine de Voluceau Rocquencourt 78153 Le Chesnay Cedex, France, February 1989.

[Wir85] N. Wirth. Report on the Programming Language Modula-2. In *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.

[WattTri] D.A. Watt and P. Trinder. Towards a theory of bulk types. Fide Technical Report 91/26, Department of Computing Science, University of Glasgow, July 1991.