Appeared in: ***Proceedings of the 5th Conference on Extending Database Technology, EDBT'96*, Avignon, France, March 1996. Springer Verlag.**

Exploiting Persistent Intermediate Code Representations in Open Database Environments[*]

Andreas Gawecki Florian Matthes

Universität Hamburg, Vogt-Kölln-Straße 30
D-22527 Hamburg, Germany
{gawecki,matthes}@informatik.uni-hamburg.de

Abstract. Modern database environments have to execute, store, analyze, optimize and generate code at various levels of abstraction (queries, views, triggers, query execution plans, methods, 4GL programs, etc.). We present TML, an abstract persistent intermediate code representation developed in the Tycoon[2] project to fully integrate static and dynamic code analysis and rewriting. TML is a continuation passing style (CPS) language which excels in its explicit, high-level representation of control and data dependencies. We formally define TML and its core rewrite rules which unify many well-known optimizing transformations. We also present Tycoon's innovative reflective system architecture which supports modular compile-time as well as global runtime optimizations. Moreover, we describe how this architecture enables optimizations across abstraction barriers in large modular persistent applications including embedded declarative queries.

1 Introduction and Motivation

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:

[*] This research is supported by ESPRIT Basic Research, Project FIDE, #6309 and by a grant from the German Israeli Foundation for Research and Development (*bulk data classification*, I-183 060).

[2] Tycoon: Typed Communicating Objects in Open Environments.

▷ 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?

After a decade of experience with building integrated database programming environments [Schmidt 1977; Mall *et al.* 1984; Schmidt and Matthes 1994] and successive versions of optimizers for their query languages [Jarke and Schmidt 1982; Jarke *et al.* 1982; Jarke and Koch 1984; Böttcher *et al.* 1986; Eder *et al.* 1991], we made the radical decision to replace special-purpose representations for queries, programs and scripts with a single, expressive *intermediate* language. We thereby avoid incompatibilities and redundancies arising from the repeated implementation of the above functionality.

In this paper, we present the Tycoon Machine Language (TML), the common abstract persistent intermediate code representation used for local compile-time as well as global runtime optimizations in the Tycoon system developed at the University of Hamburg. 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 [Appel 1992; Kranz *et al.* 1986; Kelsey 1989; Teodosiu 1991; Gawecki 1992].

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 runtime bindings to arbitrary complex values in the persistent store and mechanisms to work with persistent TML terms attached to executable code.

The paper is organized as follows: In section 2 we present the abstract syntax and semantics of the CPS-based intermediate representation TML. The core rewrite rules on TML terms are described in section 3. Examples are then given of innovative code optimization architectures which exploit the availability of persistent TML terms at runtime (section 4.1. A full description of TML and its rewrite rules can be found in [Gawecki and Matthes 1994; Kiradjiev 1994]. The paper ends with a perspective on future research work and on other innovative application domains for uniform persistent code and query representations in the spirit of TML.

2 The Tycoon Intermediate Representation TML

This section presents the CPS-based intermediate representation TML (Tycoon Machine Language) which is used for integrated program and query compilation, transformation and analysis, both at compile time and at runtime.

2.1 Advantages of CPS Intermediate Representations

Continuation Passing Style (CPS) is a powerful yet simple program representation technique. Using continuations, various control structures commonly found in programming languages such as conditionals, (non-block-structured) loops and exception handling can be expressed quite naturally. While this is also true for many other intermediate representations such as triples or quadrupels, the main advantage of CPS lies in the great reduction of the number of program constructs which have to be handled by the compile-time and runtime optimizer while preserving much of the structural information of the source language input.

CPS representations are well-suited for machine analysis by making the flow of control and of data explicit through the uniform use of one language construct: the procedure call. Since CPS does not have implicit procedure returns, this language construct can be viewed as a generalized **goto** with parameter passing [Steele 1978].

CPS has simple and clean semantics based on the λ-calculus. TML is effectively a call-by-value λ-calculus with store semantics. A number of predefined *primitive procedures* (section 2.3) operate on an implicit, hidden store.

CPS supports higher-order languages, i.e. languages where functions may take other functions as arguments. For example, the selection operation $\sigma_p(R)$ in relational algebra takes two arguments, a range relation R and a selection predicate p where p can be understood as a boolean function on the element type of R.

Like terms in the λ-calculus, CPS terms provide an integrated representation of code fragments and their associated data bindings. To meet the specific requirements of persistent languages and database languages, TML terms may contain simple literal values and object identifiers which denote arbitarily complex objects (tables, indices, ADT values) in the persistent Tycoon object store.

By representing programs in CPS, many well-known program and algebraic query optimization techniques become special cases of a few simple and general λ-calculus transformations. These transformations can be applied freely even in the presence of nonterminating computations and/or side-effecting calls to primitive procedures. This is due to the syntactical restrictions on CPS terms (see Sec. 2.2) which require actual parameters to function calls to be constants, variables or abstractions.

Six different node types are sufficient to represent the data structures for a TML tree. This simplicity facilitates the construction of compact language processors like compiler front ends, back ends and optimizers.

2.2 The Minimalistic Abstract TML Syntax

The complete abstract syntax of TML is defined in figure 1. The set of literal constants (*Lit*) includes simple values such as integers, characters and boolean

$lit \in Lit$	*Literal constants (including object identifiers)*	
$t \in Temp$	*Temporary variables*	
$c \in Cont$	*Continuation variables*	
$prim \in Prim$	*Primitive procedures (procedure constants)*	
v	$::= t \mid c$	*Variables*
val	$::= lit \mid v \mid abs$	*Values*
abs	$::= \lambda(v_1..v_n)\, app$	*Abstractions, $n \geq 0$*
app	$::= (val_0\, val_1..val_n)$	*Applications, $n \geq 0$*
	$\mid\ (prim\, val_1..val_n)$	

Fig. 1. Abstract Syntax for the Tycoon Machine Language TML

values, as well as references (*object identifiers, OIDs)* to complex objects in the persistent object store. These *values* can be bound to language *variables* (identifiers) by means of an *application.* In the following example, an integer literal, a character constant and an object identifier are bound to variables *i*, *ch* and *oid*, respectively. These variables might be used as values within the body *app* of the λ-abstraction (which in turn must be an application):

```
(λ(i ch oid) app
  13
  'a'
  <oid 0x005b4780>)
```

Abstractions are also values in TML, i.e. TML is a higher-order language. This means that abstractions may be bound to variables and that these variables may be used in the functional position of an application (val_0 on the right hand side of the production for *app* in figure 1). In TML, the body of an abstraction must be an application, and applications are surrounded by parentheses. Therefore, the scope of the abstraction is unambiguous. In the following example, an abstraction with a single formal parameter *t* is bound to the variable *fn*, and *fn* is used immediately within an application of the abstraction, whereby *t* is bound to an integer value:

```
(λ(fn) (fn 13)
  λ(t) app)
```

Although the semantics of TML is based on the general λ-calculus, well-formed TML programs must satisfy a number of additional constraints[3]:

1. A value used in the functional position of an application must, at runtime, evaluate to an abstraction. Furthermore, this abstraction must expect the same number of value and continuation arguments as the given application,

[3] It is important to note that these constraints are never violated by any of the TML rewrite rules introduced in section 3.

and it must expect them in the same order[4]. This property is statically enforced by the compiler front end which performs the necessary type checking on the input to the TML code generator, rejecting any program which contains an application which might violate this rule.

2. Similarly, an application of a primitive procedure must obey the calling conventions of the primitive. Again, the compiler front end (which generates calls to primitive procedures) has to enforce this constraint on any input program.
3. Continuations may not *escape* (by binding them to value identifiers), therefore, continuations are not first-class values in TML. It is not possible to store continuations in data structures where they might be retrieved and applied subsequently. This restriction allows TML procedure calls to be compiled into efficient (stack based) procedure calls and returns on stock hardware, i.e. the main motivation behind this restriction lies in the target code generation techniques we use.

   Several CPS-based compilers support continuations as first class values [Appel 1992; Kranz *et al.* 1986; Kelsey 1989; Teodosiu 1991]. However, these compilers have to translate source language constructs which capture the current continuation, for example, the *call/cc* of SCHEME [Steele 1986], or the built-in polymorphic function *callcc* of ML.
4. Identifiers (value and continuation variables) may not be bound more than once (unique binding rule), i.e. an identifier may occur only once in at most one formal parameter list. For example, the following two TML code fragments are not allowed:

   $\lambda(x\ x)app$
   $\lambda(x)(\lambda(x)app\ val)$

   This means that the TML code generator has to use fresh identifiers for the parameter list of every new λ-abstraction. The TML optimizer (section 3) and the target code generator rely heavily on this property.
5. Abstractions which are used as values (that is, not as continuations and not in functional position of applications) may take an arbitrary number of value parameters, but they must take exactly two continuation parameters: one for the *normal* continuation (which receives the computed value) and one for the *exception* continuation (which is invoked if a runtime exception occurs).

Abstractions which are used as values correspond to first-class, user level procedures. In order to make the printed TML representation used in this paper more readable, these procedures (**proc** abstractions) are differentiated from continuations (**cont** abstractions) even though both have the same internal representation and the same semantics (λ-abstractions). The differentiation is based on a purely syntactic property of abstractions: a continuation does not take any

[4] We currently investigate techniques for compiling and type-checking variable-length argument lists. These techniques would merely weaken the given well-formedness rule.

other continuation as a parameter. Thus, the parameter lists of continuation abstractions do not contain any continuation variables. The following two syntactic equivalences reflect these considerations ($n \geq 0$):

$$\lambda(t_1..t_n)\,app \equiv \mathbf{cont}(t_1..t_n)\,app$$
$$\lambda(t_1..t_n\;c_e\,c_c)\,app \equiv \mathbf{proc}(t_1..t_n\;c_e\,c_c)\,app$$

2.3 Adaptability through Primitive Procedures

In TML, most of the "real work" needed to implement source language semantics (e.g. integer arithmetic, query evaluation) is factored out into primitive procedures which are not considered part of the intermediate language itself.

A typical set of primitive procedures used for the compilation of a fully-fledged imperative, algorithmically-complete polymorphic programming language (TL [Matthes and Schmidt 1992]) is listed in figure 2. By definition, each primitive calls exactly one of its continuation arguments tail-recursively, passing the result of its computation, if any. For example, some arithmetic primitives take two continuations: the *normal* continuation which receives the calculated result, and an *exception* continuation which is invoked if the primitive fails due to overflow or division by zero.

Although the set of predefined primitive procedures is typically chosen to be rather small, the set does not need to be minimal due to efficiency tradeoffs. Moreover, it is possible to add new primitive procedures in order to meet the specific needs of more specialized source languages (e.g., supporting multiple bulk data types or scientific or statistical databases). The easiest way to support such complex instructions in TML is to define new primitives which are mapped directly to corresponding abstract machine instructions during target code generation.

New primitive procedures can be defined at back end compile-time by providing the following information to the generic TML rewriting tools:

1. A function to generate target machine code for a given call. This function is used by the code generator to map TML primitive procedure calls into sequences of target machine instructions or calls to the underlying runtime system.
2. A meta-evaluation function to perform optimizations on TML nodes representing calls to this primitive procedure. This function is used by the optimizer to perform constant folding and dead code elimination. To give an example, the primitive procedure '+' has an associated function which is able to reduce the TML application node

   (+ 1 2 c_e c_c)

   into an application of the continuation which represents *normal* (i.e. non-exceptional) execution with the result:

   (c_c 3)

3. A function to estimate the runtime cost of a given call (represented by a TML node) to the primitive procedure, measured in the number of instructions necessary to implement the primitive on an idealized abstract machine. This function is used by the optimizer to estimate the possible savings resulting from the inlining of a TML procedure containing calls to the primitive.
4. A collection of attributes useful for the optimizer, for example commutativity, side effect classes [Gifford and Lucassen 1986], and flags to enable or disable certain optimization rules. There is a default value for any of these attributes, representing the worst possible case (i.e no further information available) for the optimizer.

Note that exception handling is expressed in TML by passing continuations: Every function accepts an additional argument, the exception continuation, which is normally passed through to other functions called. To install a new exception handler, however, a new continuation function which handles exceptions in the callee's body is passed. The 'old' handler is stored automatically within the lexical environment.

This approach makes control flow explicit even in the presence of exceptions, with the advantage that exception handling can be optimized immediately without special optimization rules. This becomes important when the optimizer is inlining functions which perform extensive exception handling, which is quite common in high-level value-oriented languages.

The primitive procedure Y is a multiple-value-return CPS version of the lambda-calculus fixed point operator. The abstraction given to the Y-primitive takes n abstraction arguments $v_1..v_n$ and a continuation abstraction c_0, and returns $n+1$ abstractions. As usual in CPS, this multiple-value-return is expressed by calling the continuation c with the desired return values.

The Y-primitive computes the least fixed point of its abstraction argument. This fixed point is a vector of mutually recursive procedures and/or continuations. In other words, the effect of the Y-primitive is that the $n+1$ abstractions ***cont**()app* and $abs_1..abs_n$ are bound to the variables c_0 and $v_1..v_n$, respectively, and that these bindings are visible within the abstractions themselves. Moreover, the continuation ***cont**()app* which is bound to c_0 is invoked tail-recursively (by Y) after all the recursive bindings have been established.

To give a simple example, a loop which iterates from 1 up to 10, written in the Tycoon Language as ***for** i = 1 **upto** 10 **do** f(i) **end*** is expressed in TML as follows:

```
(Y proc(c0 for c)
   (c
     cont()              ; continuation, bound to c0
       (for 1)           ; loop entry
     cont(i)             ; loop head, bound to 'for'
       (> i 10 cc cont() ; loop exit
       (f i ce cont(t1)  ; loop body
       (+ i 1 ce cont(t2)
       (for t2))))))     ; recursion
```

$(p\ val_1\ val_2\ c_e\ c_c)$	integer arithmetic, $p \in \{+, -, *, /, \%\}$
$(p\ val_1\ val_2\ c_1\ c_2)$	integer comparison, $p \in \{<, >, <=, >=\}$
$(p\ val_1\ val_2\ c)$	bit operations on integers, $p \in \{<<, >>, \&, \vert, \hat{}, \tilde{}\}$
(char2int $val\ c$)	convert a byte to an integer value
(int2char $val\ c$)	convert an integer to a byte value
(array $val_1 \ldots val_n\ c$)	create a mutable array holding n object references
(vector $val_1 \ldots val_n\ c$)	create an immutable array
(new $val_1\ val_2\ c$)	create a mutable array holding val_1 object references, initialized with val_2
($new $val_1\ val_2\ c$)	create a mutable byte array holding val_1 simple byte values, initialized with val_2
([] $val_1\ val_2\ c$)	array access: indirect indexed load
([]:= $val_1\ val_2\ val_3\ c$)	array update: indirect indexed store
($[] $val_1\ val_2\ val_3\ c$)	byte array access
($[]:= $val_1\ val_2\ val_3\ c$)	byte array update
(== val	case analysis based on object identity with...
$val_1 \ldots val_n$	values and...
$c_1 \ldots c_n\ [c_{n+1}]$)	branches (optional else branch)
(Y $\lambda(c_0\ val_1 \ldots val_n\ c)\ app$)	the Y combinator
(size $val\ c$)	array or byte array size (in slots)
(move $val_1 \ldots val_5\ c$)	move array contents
($move $val_1 \ldots val_5\ c$)	move bytearray contents
(ccall $val_{fmt}\ val_{cfn}\ c_1\ c_2$)	C language function call
(pushHandler $c_1\ c_2$)	Install continuation c_1 as a new exception handler, continue with c_2
(popHandler c)	Remove the topmost exception handler, continue with c
(raise val)	Raise exception

Fig. 2. TML primitives for the compilation of an imperative programming language

As usual, *cc* and *ce* represent the current normal and the current exception continuation, respectively. The introduction of the Y-primitive obviates the need to extend the intermediate language with a special recursive binding operator similar to the letrec special form of SCHEME. A similar primitive is used in the ORBIT compiler [Kranz *et al.* 1986].

The set of TML primitives is described in more detail with additional examples in [Gawecki and Matthes 1994; Kiradjiev 1994].

3 Analysis and Rewriting of TML Intermediate Representations

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 [Appel 1992].

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.

In the following, we give a formal definition of the core TML rewrite rules. We present these rules using the notation

$$\begin{array}{c}(\mathit{precondition}):\\ A \xrightarrow{\mathit{rule\ name}} B\end{array}$$

indicating that the TML expression A may be rewritten to the TML expression B if *precondition* evaluates to true. By convention, an empty precondition evaluates to true.

A key feature of CPS-based representations is the fact that control and data dependencies are captured uniformly by the concept of bound variables (variable occurrences inside the scope of a binder). In the precondition, we denote the number of occurrences of a variable v in an TML expression E with $|E|_v$. This function is defined inductively on the abstract syntax of TML as follows:

$$\begin{aligned}
|v|_v &= 1\\
|\mathit{lit}|_v &= 0\\
|\mathit{prim}|_v &= 0\\
|v_1|_{v_2} &= 0 \quad (v_1 \neq v_2)\\
|\lambda(v_1..v_n)\ \mathit{app}|_v &= |\mathit{app}|_v\\
|(\mathit{val}_0\ \mathit{val}_1..\mathit{val}_n)|_v &= \textstyle\sum_{i=0}^{n} |\mathit{val}_i|_v
\end{aligned}$$

Similarly, on the right side of a TML rewrite rule, we use the notation $E[\mathit{val}/v]$ which denotes the expression E where every occurrence of the variable v is replaced by the value val. Name clashes cannot occur during substitution because each variable is bound only once in a TML tree (unique binding rule). This property is achieved by the α-*conversion* performed during TML code generation, and is never violated by any of the TML rewrite rules, except in one case: if, in an application of the substitution rule, the value substituted is an abstraction, the formal parameters of this abstraction occur temporarily at two different places within the TML tree. However, this does not do any harm because the first occurrence of the abstraction will be removed immediately (by an application of the rewrite rule *remove*) since the substituted variable is not referenced any more.

Variable substitution is defined inductively on the abstract syntax of TML as follows:

$$\begin{aligned}
v[val/v] &= val \\
v'[val/v] &= v' \quad (v \neq v') \\
lit[val/v] &= lit \\
prim[val/v] &= prim \\
\{\lambda(v_1..v_n)\ app\}[val/v] &= \lambda(v_1..v_n)\ \{app[val/v]\} \\
(val_0\ val_1..val_n)[val/v] &= (val_0[val/v]\ val_1[val/v]..val_n[val/v])
\end{aligned}$$

Values bound to λ-variables may be substituted freely within the TML tree since, due to CPS, they are not allowed to contain nested primitive or function calls which may cause side effects in the store.

The complete set of the TML rewrite rules which is currently implemented as a part of the reduction pass is given below. The expansion pass uses a variant of the *subst* rewrite rule in order to perform procedure inlining. Although each individual rule is fairly simple, the combination of these rules is surprisingly powerful. Many of the well-known standard program optimizations like constant and copy propagation, dead code elimination, procedure inlining or loop unrolling are just special cases of these general λ-calculus transformations.

The *subst* rewrite rule replaces each occurrence of a bound variable v_i with the corresponding value val_i. Note that the precondition of this rule states that, if the value val_i is an abstraction, the variable v_i must be referenced exactly once. This precondition prevents the TML code from growing arbitrary large:

$$\begin{array}{lcl}
(val_i \notin Abs \vee |app|_{v_i} = 1) : & & \\
(\lambda(v_1..v_i..v_n)\, app & \xrightarrow{subst} & (\lambda(v_1..v_i..v_n)\, app[val_i/v_i] \\
\quad val_1..val_i..val_n) & & \quad val_1..val_i..val_n)
\end{array}$$

The *remove* rewrite rule strikes out a bound variable v_i which is not referenced any more. The corresponding value val_i is also removed. Note that this is possible because, due to syntactical restrictions (cf. figure 1), val_i cannot be an application, and, therefore, cannot contain any calls to side-effecting primitive procedures:

$$\begin{array}{lcl}
(|app|_{v_i} = 0) : & & \\
(\lambda(v_1..v_i..v_n)\, app & \xrightarrow{remove} & (\lambda(v_1..v_{i-1}\ v_{i+1}..v_n)\, app \\
\quad val_1..val_i..val_n) & & \quad val_1..val_{i-1}\ val_{i+1}..val_n)
\end{array}$$

The *reduce* rewrite rule simply removes applications of λ-abstractions which do not bind any variables:

$$(\lambda()\, app\ ) \xrightarrow{reduce} app$$

The *η-reduce* rewrite rule removes unnecessary abstractions:

$$\begin{array}{l} (\forall_{i=1\ldots n}\ |val|_{v_i} = 0) : \\ \lambda(v_1..v_n)(val\ \ v_1..v_n) \xrightarrow{\eta\text{-}reduce} val \end{array}$$

The *fold* rewrite rule uses an evaluation function *eval* which knows details of the semantics of primitive procedures:

$$(prim\ \ val_1..val_n) \xrightarrow{fold} eval(prim, val_1, .., val_n)$$

Given an application of a certain primitive, it may be able to *meta-evaluate* the call, yielding a somewhat simpler TML tree than the original call. For example, if the evaluation function detects that a given call to a primitive will always compute the same value and invoke always the same continuation, it reduces the primitive call to an application of the continuation to the result. Typically, this is possible if some of the arguments are literal constants:

$$(+\ \ 1\ \ 2\ \ c_1\ \ c_2) \xrightarrow{fold\ +} (c_2\ \ 3)$$

To give another example, a call to the object identity primitive will fold if the value to be compared is identical to one of the case tags:

$$(==\ \ 2\ \ 1\ \ 2\ \ 3\ \ c_1\ \ c_2\ \ c_3) \xrightarrow{fold\ ==} (c_2)$$

If the *eval* function cannot perform any useful meta-evaluation, it simply returns the original call to the primitive.

The *case-subst* rewrite rule substitutes variables in case statements with the tag value of the corresponding branch:

$$\begin{array}{ll} (==\ v & \xrightarrow{case\text{-}subst} (==\ v \\ \quad val_1..val_n & \quad val_1..val_n \\ \quad val_1^c..val_n^c) & \quad val_1^c[val_1/v]..val_n^c[val_n/v]) \end{array}$$

$$\begin{array}{ll} (==\ v & \xrightarrow{case\text{-}subst} (==\ v \\ \quad val_1..val_n & \quad val_1..val_n \\ \quad val_1^c..val_n^c\ val_{n+1}^c) & \quad val_1^c[val_1/v]..val_n^c[val_n/v]\ val_{n+1}^c) \end{array}$$

Finally, there are two rewrite rules which operate on calls to the primitive procedure Y. The *Y-remove* rewrite rule strikes out any recursive procedure which is not referenced from within the bodies of the other (mutually) recursive procedures, whereas the *Y-reduce* rewrite rule removes empty Y applications:

$$
\begin{array}{lcl}
(|app|_{v_i} = 0 \wedge \forall_{i \neq j}\ |val_j|_{v_i} = 0): & & \\
(Y\ \ \lambda(c_0\ v_1..v_i..v_n\ c) & \xrightarrow{\textit{Y-remove}} & (Y\ \ \lambda(c_0\ v_1..v_{i-1}\ v_{i+1}..v_n\ c) \\
\quad (c\ \ \mathbf{cont}()\, app & & \quad (c\ \ \mathbf{cont}()\, app \\
\qquad abs_1 & & \qquad abs_1 \\
\qquad .. & & \qquad .. \\
\qquad abs_i & & \qquad abs_{i-1} \\
\qquad .. & & \qquad abs_{i+1} \\
\qquad abs_n)) & & \qquad .. \\
 & & \qquad abs_n))
\end{array}
$$

$$
\begin{array}{l}
(|app|_{c_0} = 0): \\
(Y\ \ \lambda(c_0\ c)(c\ \ \mathbf{cont}()\, app)) \xrightarrow{\textit{Y-reduce}} app
\end{array}
$$

4 Exploiting Persistent TML Representations

The TML intermediate representation and the TML rewrite rules described so far can be utilized fairly straightforward to build a static optimizer for a given source language and target architecture. Additionally, TML supports innovative code optimization scenarios which we describe in the following subsections.

4.1 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 runtime. 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.

In the following, we explain by an example how the Tycoon system achieves optimizations across abstraction barriers. Given a uniform representation of programs and queries, the "trick" to eliminate these 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.

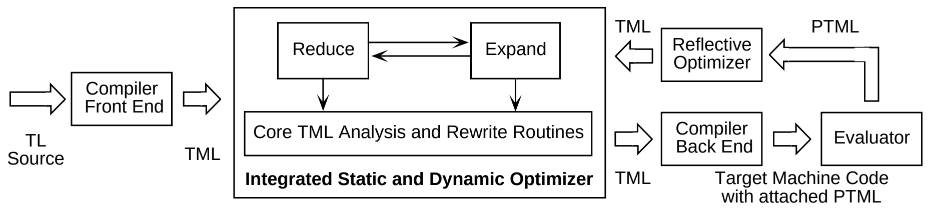

Fig. 3. Interaction between compilation, optimization and evaluation in the Tycoon system

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 runtime.

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 compact persistent representation of the TML tree (Persistent TML, PTML) for *f*. At runtime, 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. 3).

The mapping from PTML back to TML also returns the set of R-value bindings ([identifier, OID] pairs) established at runtime. 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.

For example, the following Tycoon module *complex* exports a (hidden) abstract data type *complex.T* and encapsulated accessor functions *complex.x()*, *complex.y()*, . . . on values of that type:

```
module complex export
  Let T = Tuple x,y :Real end
  let new(x,y :Real):T = tuple x y end
  let x(complex :T) :Real = complex.x
  let y(complex :T) :Real = complex.y
  ...
end
```

Here is a function *abs* which uses the functions exported from the module:

```
let abs(c :complex.T) :Real =
  sqrt(complex.x(c) * complex.x(c) + complex.y(c) * complex.y(c))
```

In the static context of this function, the implementation of the module (the binding to the module value) is not available. Only after module linkage (Tycoon has first class modules), the dynamic context of *abs* contains bindings to the exported function.

The programmer can obtain a (dynamically created) function *optimizedAbs* which is equivalent to the original function *abs* but which executes faster than the original by explicitly invoking the optimizer on *abs*:

```
let optimizedAbs = reflect.optimize(abs)
```

In our current implementation, the reflective dynamic optimizer inlines the bodies of *complex.x* and *complex.y*, i.e., *optimizedAbs* is equivalent to:

```
let optimizedAbs(c :complex.T) :Real = sqrt(c.x*c.x + c.y*c.y)
```

Finally, the optimized function which takes advantage of the particular encapsulated implementation of complex numbers can be applied:

```
optimizedAbs(complex.new(3 4))
```

The main extension which is necessary to be able to carry out this kind of dynamic optimization is to re-establish, in TML, the R-value bindings of global variables as they are stored in the *closure record* of the runtime representation of a given procedure. For the example above, this means that the values of the variables *complex*, '+', '*' and *sqrt* which are global to the function *abs* are fetched from the closure record of *abs* and are bound to the corresponding identifiers *before* the (original) body of *abs* is processed by the optimizer[5]:

```
proc(c_10 c_11)
 (λ(complex_6 *_7 +_8 sqrt_9)
   ([] complex_6 2 cont(t_12)          ; begin of the original body of abs
   (t_12 c_10 cont(t_13)
   ([] complex_6 2 cont(t_14)
   (t_14 c_10 cont(t_15)
   (*_7 t_13 t_15 cont(t_16)
   ([] complex_6 3 cont(t_17)
   (t_17 c_10 cont(t_18)
   (+_8 t_16 t_18 cont(t_19)
   ([] complex_6 3 cont(t_20)
   (t_20 c_10 cont(t_21)
   (*_7 t_19 t_21 cont(t_22)
   (sqrt_9 t_22 cont(t_23)
   (c_11 t_23)))))))))))))            ; end of the original body of abs
  <oid 0x005b4780>            ; value of module complex
  <oid 0x001b4044>            ; value of function '*'
  <oid 0x001b4024>            ; value of function '+'
  <oid 0x00993d28>)           ; value of function sqrt
```

[5] This is a TML listing similar to the output of our TML pretty-printer. Note that, during α-conversion, each identifier name is appended with a unique number in order to distinguish it from any other identifier.

Given these value bindings, the optimizer is able to perform substitution, constant folding and procedure inlining in the usual way, yielding a result which is equivalent to the above Tycoon code for *optimizedAbs*.

4.2 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" [Catell 1994].

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 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 [Matthes and Schmidt 1991], query and program optimization have to interact closely (see Fig. 4): 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 runtime bindings (for example, knowledge about index structures), we have to delay query optimizations until runtime as described in the previous section. The translation of a declarative query construct embedded in the source language into a TML

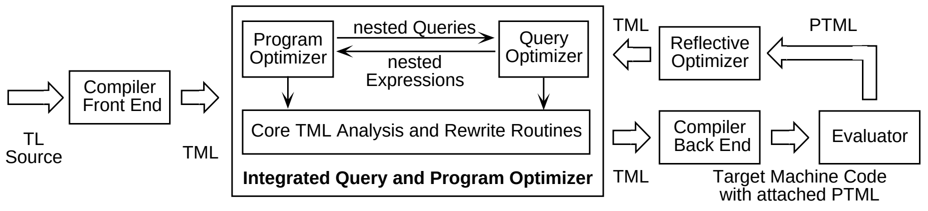

Fig. 4. Embedded Query Optimization

term is rather straightforward and resembles the usual approach of mapping a relational query 1:1 into a tree of algebraic operators [Ullman 1989].

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, CPS 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 [Trinder 1991; Breazu-Tannen *et al.* 1991; Fegaras 1994].

For a given set of primitive procedures, algebraic and implementation-oriented query optimization rules can be expressed quite naturally in CPS, for example, the simple equivalence $\sigma_p(\sigma_q(R)) \equiv \sigma_{p\wedge q}(R)$ can be written as:

```
                      merge-select
(select               ----------->  (select
  λ(r1 ce1 cc1)                       λ(r1 ce1 cc1)
    (q r1 ce1 cc1)                      (p r1 ce1
  R                                         cont(t1)
  ce                                          (q r1 ce1
  cont(tempRel)                                   cont(t2)
    (select                                         (and t1 t2 ce1 cc1)))
      λ(r2 ce2 cc2)                   R
        (p r2 ce2 cc2)                ce
      tempRel                         cc)
      ce
      cc))
```

In particular, scoping restrictions which limit the applicability of certain rewrite rules are also directly expressible using the notation introduced in section 3. For example, if the variable x does not appear in the predicate p of the quantified expression $\exists x \in R : p$, this predicate is equivalent to $p \wedge (R \neq \emptyset)$. This rule is written as follows, using CPS notation and the predefined procedures *and*, *exists* and *empty*:

```
(|p|_x = 0) :
(exists          trivial-exists
  λ(x ce' cc')p  ------------->  (empty R
  R                                ce
  ce                               cont(t1)
  cc)                                (λ(ce' cc')p
                                       ce
                                       cont(t2)
                                         (and t1 t2 ce cc)))
```

Note that the resulting TML tree will be further reduced and optimized using any other applicable rewrite rule.

5 Related Work

The issue of uniform intermediate code representations in database environments arose in the integration of program and query optimization.

Freytag [Freytag and Goodman 1989] investigated the problem of translating relational queries into iterative programs which are quite effectively simplified using a set of transformation rules. Queries are rewritten into nested applications of stream operators which are similar to our polymorphic higher-order iterator functions. The transformation process is based on purely algebra-based relational query specifications which may neither contain embedded (user-defined) function calls nor side effects.

Lieuwen and DeWitt [Lieuwen and DeWitt 1991] have applied loop transformations on queries written in the database programming language O++

[Agrawal and Gehani 1989] which provides constructs to iterate through a set in an unspecified order. Similar constructs can be found, for example, in Pascal/R [Schmidt 1977] and DBPL [Schmidt and Matthes 1994]. Lieuwen and DeWitt focus on the reordering of joins which are expressed via nested set iterations. Iterations may contain embedded function calls and output statements which constrain reorderings. However, they do not interact with the program optimizer. They have developed their own query tree representation which is quite different from the AST used by the compiler.

Breazu-Tannen et al. [Breazu-Tannen *et al.* 1991] propose a programming paradigm based on structural recursion on sets which comes close to both the semantic simplicity of the relational algebra and the expressive power of algorithmically complete programming languages. The authors suggest that this conceptual unification of queries and functional programs will contribute to the optimization problem.

The work which is most closely related to ours is described in [Poulovassilis and Small 1994]. This work investigates algebraic query optimization techniques for database programming languages in the context of a purely declarative functional language which supports sets as first-class objects. Within the language, it is possible to use user-defined functions as query predicates and as target expressions. Since the language is computationally complete, the possibility of non-termination and the construction of infinite data structures must be taken into account, while problems concerning side-effects are avoided. As in our framework, all optimizations can be fully exploited for all subexpressions of a query since no distinct languages are used to represent query trees and programming language expressions. In contrast to our work, the query language is not integrated into a general-purpose persistent programming environment.

6 Concluding Remarks

We have presented the syntax and generic rewrite rules for TML, a persistent intermediate code representation. We also reported on our experience building 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. Due to its parameterization by user-defined primitive procedures, TML is virtually independent of the Tycoon language TL and Tycoon's bulk data library and can be tailored with little effort to other program or query languages. By utilizing TML, innovative optimizers like reflective code optimizers and integrated query and program optimizers can be constructed systematically.

The current version of the Tycoon system fully implements dynamic reflective optimization across abstraction barriers based on CPS representations as described in this paper. In particular the static and dynamic optimizers share the same code for TML analysis and rewriting. As described in more detail in [Kiradjiev 1994], 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 runtime) 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). On the down side, due to the space requirements for the additional persistent encoding of the TML tree for each function, the code size doubles at the same time (1.2MB vs. 600kB for the complete Tycoon system). We are currently investigating techniques to reconstruct a TML representation by examining the persistent executable code representation of a procedure, effectively inverting the target machine code generation process. In general, the TML tree reconstructed this way will not be isomorphic to the original TML tree which we currently encode in PTML. The interesting question is whether this has an impact on the possible optimizations, in particular in the presence of nested recursive function bindings.

More work is required to evaluate the effectiveness of query optimization exploiting the availability of a uniform program and query representation at runtime. We are also very interested in exploiting TML for other tasks in data-intensive applications, like code shipping in distributed systems [Mathiske *et al.* 1995], synchronization of persistent threads [Matthes and Schmidt 1994], access control and security issues [Rudloff *et al.* 1995].

References

Agrawal and Gehani 1989: Agrawal, R. and Gehani, N.H. Rationale for the design of persistence and query processing facilities in the database programming language O++. In *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, June 1989.

Appel 1992: Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

Böttcher et al. *1986:* Böttcher, S., Jarke, M., and Schmidt, J.W. Adaptive predicate managers in database systems. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, 1986.

Breazu-Tannen et al. *1991:* Breazu-Tannen, V., Buneman, P., and Naqvi, S. Structural recursion as a query language. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece.* Morgan Kaufmann Publishers, September 1991.

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

Eder et al. *1991:* Eder, J., Rudloff, A., Matthes, F., and Schmidt, J.W. Data construction with recursive set expressions in DBPL. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, April 1991.

Fegaras 1994: 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.

Freytag and Goodman 1989: Freytag, J.C. and Goodman, N. On the translation of relational queries into iterative programs. *ACM Transactions on Database Systems*, 14(1), March 1989.

Gawecki and Matthes 1994: 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.

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

Gifford and Lucassen 1986: Gifford, David K. and Lucassen, John M. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Cambridge, Massachusetts, August 4-6, 1986*, pages 28–38, 1986.

Jarke and Koch 1984: Jarke, M. and Koch, J. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

Jarke and Schmidt 1982: Jarke, M. and Schmidt, J.W. Query processing strategies in the Pascal/R relational database management system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1982.

Jarke et al. *1982:* Jarke, M., Koch, J., Mall, M., and Schmidt, J.W. Query optimization research in the database programming languages (DBPL) project. *IEEE – Data Engineering*, pages 11–14, September 1982.

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

Kiradjiev 1994: Kiradjiev, P. Dynamische Optimierung in CPS-orientierten Zwischensprachen. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, December 1994.

Kranz et al. *1986:* 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.

Lieuwen and DeWitt 1991: Lieuwen, Daniel F. and DeWitt, David J. Optimizing loops in database programming languages. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*, Nafplion, Greece, September 1991. Morgan Kaufmann Publishers.

Mall et al. *1984:* Mall, M., Reimer, M., and Schmidt, J.W. Data selection, sharing and access control in a relational scenario. In Brodie, M.L., Myopoulos, J.L., and Schmidt, J.W., editors, *On Conceptual Modelling*. Springer-Verlag, 1984.

Mathiske et al. *1995:* 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).

Matthes and Schmidt 1991: Matthes, F. and Schmidt, J.W. Bulk types: Built-in or add-on? In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufmann Publishers, September 1991.

Matthes and Schmidt 1992: Matthes, F. and Schmidt, J.W. Definition of the Tycoon Language TL – a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes and Schmidt 1994: 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.

Poulovassilis and Small 1994: Poulovassilis, A. and Small, C. Investigation of algebraic query optimisation for database programming languages. In *Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile*, September 1994.

Rudloff et al. *1995:* 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. (to appear).

Schmidt and Matthes 1994: Schmidt, J.W. and Matthes, F. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.

Schmidt 1977: Schmidt, J.W. Some high level language constructs for data of type relation. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Toronto, Canada*, August 1977.

Steele 1978: Steele, Guy L. Rabbit: A compiler for SCHEME. Technical report, Massachusetts Institute of Technology, May 1978.

Steele 1986: Steele, Guy L. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

Teodosiu 1991: Teodosiu, Dan. Hare: An optimizing portable compiler for Scheme. *ACM SIGPLAN Notices*, 26(1):109–120, January 1991.

Trinder 1991: 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.

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