The Tycoon Machine Language TML

An Optimizable Persistent Program Representation

Andreas Gawecki Florian Matthes

Universität Hamburg
Vogt-Kölln-Straße 30
D-22527 Hamburg

{gawecki,matthes}@dbis1.informatik.uni-hamburg.de

July 29, 1994

Abstract

This document provides a brief overview of the Tycoon Machine Language TML which is used as a persistent intermediate program representation within the Tycoon system. TML representations of Tycoon programs provide the basis for extensive tree analysis and rewriting aiming at various optimizations which are independent of source languages and target machines.

TML representations are used for both static (compile-time) and dynamic (runtime) optimizations. Dynamic optimizations are of particular importance for performance improvement of large and modular programs.

TML is based on Continuation Passing Style (CPS), a powerful yet simple program representation technique developed as a framework for optimizing compilers. The advantage of CPS lies in the significant reduction in number of program constructs to be handled by the Tycoon static and dynamic (reflective) optimizer.

We propose that in modern persistent data-intensive applications traditional database query optimization be integrated into a more generalized framework for extensive program analysis and transformation.

Contents

1 Introduction **1**

2 The Tycoon Machine Language TML **2**
2.1 TML Abstract Syntax 2
2.2 Primitive Procedures 5
2.3 Exceptions 6
2.4 Compilation Phases 7

3 Translating TL Source Programs into TML **8**
3.1 Constants 9
3.2 Bindings 9
3.3 Expressions 9
3.4 Functions 9
3.5 Aggregates 10
3.6 Recursive Bindings 11
3.7 Conditionals 11
3.8 Loops 11
3.9 Case Statements 12
3.10 Exception Handling 12
3.11 C Language Function Calls 13
3.12 Builtin Functionality 14
3.13 L-Value Bindings 14

4 Optimization **15**
4.1 TML Rewrite Rules 16
4.2 Extensions for Reflective Runtime Optimization 19

5 Target Machine Code Generation **20**
5.1 Exception Conversion 21
5.2 Environment Analysis and Closure Conversion 22
5.3 Generating ANSI C Code 24

6 Concluding Remarks **25**

A TML Data Structures **28**

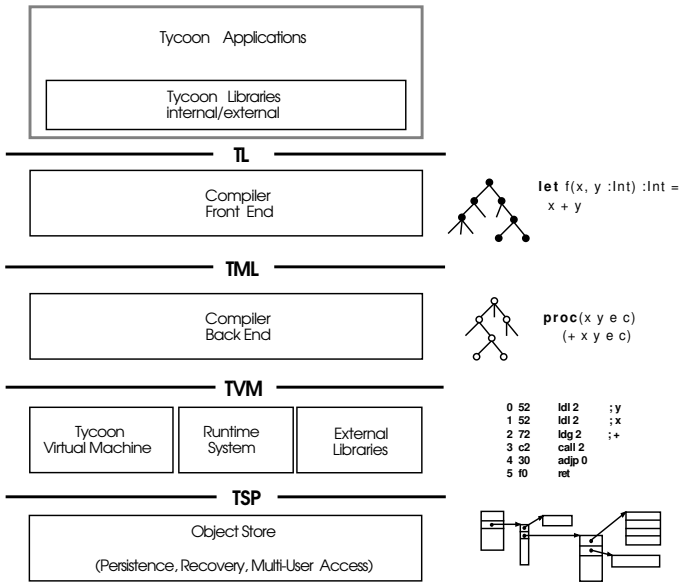

Figure 1: Tycoon System Layers

1 Introduction

The *Tycoon* project aims at substantially improving the productivity of database programmers by providing an integrated linguistic and architectural framework for the flexible integration of external services into an open database programming environment [Matthes and Schmidt 1993]. Tycoon applications are written in a strongly typed, higher-order polymorphic programming language (TL, *Tycoon Language*) that offers a minimal and orthogonal set of naming, binding and typing concepts to the application and system programmer [Matthes and Schmidt 1992].

The Tycoon system is organized as a layered architecture as outlined in figure 1. Tycoon programs are parsed by the compiler front end and an abstract syntax tree representation with annotated type information is built. Subsequently, the program is passed to the compiler back end in the form of a tree-structured, untyped intermediate program representation (TML, *Tycoon Machine Language*). This intermediate representation is used to perform various source language- and target machine-independent optimizations by extensive tree rewriting. The TML tree is further translated into a byte-coded sequence of abstract machine instructions (TVM, *Tycoon Virtual Machine*) that permits efficient code execution at runtime. The Tycoon virtual machine and the runtime system communicate with the underlying object store via a data-model independent software protocol (TSP, *Tycoon Store Protocol*) that encapsulates different object store implementations and their components for access optimization, storage reclamation, persistence, concurrency control and distribution.

This document is intended to provide an overview over the intermediate program representation TML and the translation process from typed TL source code into untyped TML trees. The abstract syntax of TML and the set of available primitive operations are presented in

sections 2.1 and 2.2. Two alternatives in handling runtime exceptions in the intermediate program representation are discussed in section 2.3. Section 2.4 gives an overview of the different passes of the Tycoon compiler back end which works on the TML intermediate representation. The rest of the paper describes the various passes of the compiler back end in more detail. The TL data structures used to represent TML trees within the Tycoon system are given in the appendix.

2 The Tycoon Machine Language TML

Building on the experience gained with the first TML version that was designed to be executed directly [Matthes 1993; Mathiske 1992], we have developed a new intermediate representation that emphasizes easy program compilation, transformation and analysis. These goals are achieved by representing programs in *Continuation Passing Style* (CPS) which has been used successfully in several optimizing compilers for functional languages [Appel 1992; Kranz *et al.* 1986; Kelsey 1989; Teodosiu 1991] as well as for object-oriented languages [Gawecki 1992].

CPS is a powerful yet simple program representation technique. The advantage of CPS lies in the great reduction of the number of program constructs that have to be handled by the Tycoon compile-time and runtime (reflective) optimizer.

CPS is particularly well-suited for data flow analysis by making the flow of control 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 a 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 2.2) operate on an implicit, hidden store.

By representing programs in CPS, many well-known optimization techniques become special cases of a few simple and general λ-calculus transformations. Due to certain syntactical restrictions on the CPS tree, these transformations can be applied freely even in the presence of side-effecting calls to primitive procedures.

Six different node types are sufficient to represent the data structures for a TML tree (cf. appendix A). This simplicity makes it possible to build compact language processors like compiler front ends, back ends and optimizers.

2.1 TML Abstract Syntax

The abstract syntax of TML is given in figure 2. The set of literal constants (*Lit*) includes simple values such as integers, characters and boolean values, as well as references (*object identifiers, OIDs)* to complex objects in the store. These *values* can be bound to *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):

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

Figure 2: Abstract Syntax for TML

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

Abstractions are also values in TML. This means that they may be bound to variables and that these variables may be used in the functional position (val_0 on the right hand side of the production for *app* in figure 2) of an application. 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[1]:

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, and it must expect them in the same order[2]. This property is statically enforced by the compiler front end that performs the necessary type checking on the input to the TML code generator, rejecting any program that contains an application which might violate this rule.

[1] It is important to note that these contraints are never violated by any of the TML rewrite rules introduced in chapter 4.

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

2. Similarly, an application of a primitive procedure must obey the calling conventions of the primitive. Again, the compiler front end (that 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 no first class values in TML. It is not possible to store continuations in data structures where they might subsequently be retrieved and applied. This restriction allows TML procedure calls to be compiled into efficient (stack based) procedure calls and returns on stock hardware, so the main motivation lies in the target code generator techniques we use. If we built a somewhat more complex code generator that is able to deal with first-class continuations, we could simply drop this restriction. The rest of the compiler (e.g. the optimizer) would not be affected.

   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 that capture the current continuation, for example, the *call/cc* of SCHEME [Steele 1986], or the built-in polymorphic function *callcc* of ML [Reppy 1990]).

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 4) and the target code generator (section 5.3) rely heavily on this property.

5. Abstractions that 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 (that receives the computed value) and one for the *exception* continuation (that is invoked iff a runtime exception occurs, cf. section 5.1).

Abstractions that are used as values correspond to first-class, user level procedures. In order to make the printed TML representation used in this document more readable, these procedures (**proc** abstractions) are differentiated from continuations (**cont** abstractions) even though both have the same internal representation (cf. appendix A) and the same semantics (λ-abstractions). The differentiation is based on a purely syntactic property of abstractions: a continuation does not take any 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 ($n \geq 0$):

$$\begin{aligned} \lambda(t_1..t_n)app &\equiv \mathbf{cont}(t_1..t_n)app \\ \lambda(t_1..t_n\ c_e\ c_c)app &\equiv \mathbf{proc}(t_1..t_n\ c_e\ c_c)app \end{aligned}$$

$(p\ x\ y\ c_e\ c_c)$	integer arithmetic, $p \in \{+, -, *, /, \%\}$
$(p\ x\ y\ c_1\ c_2)$	integer comparison, $p \in \{<, >, <=, >=\}$
$(p\ x\ y\ c)$	bit operations on integers, $p \in \{<<, >>, \&, \mid, \hat{}, \tilde{}\}$
(char2int $x\ c$)	convert a character to an integer value
(int2char $x\ c$)	convert an integer to a character 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 n *init* c)	create a mutable array holding n object references, initialized with *init*
($new n *init* c)	create a mutable byte array holding n simple byte values, initialized with *init*
([ ] $x\ i\ c$)	array access: indirect indexed load
([ ]:= $x\ i\ v\ c$)	array update: indirect indexed store
($[ ] $x\ i\ c$)	byte array access
($[ ]:= $x\ i\ v\ c$)	byte array update
(== x	case analysis based on object identity with...
$v_1 \ldots v_n$	values and...
$c_1 \ldots c_n\ [c_{n+1}]$)	branches (optional else branch)
(Y $\lambda(v_1 \ldots v_n\ c)app$)	the Y combinator
(size $v\ c$)	array or byte array size (in slots)
(move n *src srcOffset dst dstOffset* c)	move array contents
($move n *src srcOffset dst dstOffset* c)	move bytearray contents
(ccall *fmt 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 x)	Raise exception x

Figure 3: TML Primitives supporting TL source program compilation

2.2 Primitive Procedures

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

The set of primitive procedures handled by the current back end is listed in figure 3. Some examples of their usage will be given in sections 3 and 5. 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 failes due to overflow.

Although this set of predefined primitive procedures is rather small, the set is not minimal due to efficiency tradeoffs. Moreover, it is possible to define new primitive procedures in order

to meet the specific needs of more specialized source languages, for example languages with a rich built-in object-oriented data model like Fibonacci [Albano *et al.* 1993]. These languages are often built on top of abstract machines with complex instruction sets. The easiest way to support such complex instructions in TML is to define new primitives that are mapped directly to the 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.
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. To give an example, the primitive procedure '+' has an associated function that 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.

2.3 Exceptions

The following two approaches to handle exceptions in CPS have been described by [Appel 1992]:

1. A distinguished location that holds the current exception handler is kept in the store. This handler is simply a continuation taking one argument, the exception object. Two primitive procedures manipulate that location:

   sethandler installs a new handler (a continuation function), and

   gethandler retrieves it and passes it to its continuation argument, where it can subsequently be invoked.

2. Every function accepts an additional argument, the exception continuation, which is normally passed through to other functions called. To install a new handler, however, a new continuation function is passed which handles exceptions in the callee's body. The 'old' handler is remembered automatically within the lexical environment.

Obviously, the second scheme is more in the spirit of continuation passing style since it makes control flow explicit even in the presence of exceptions. This approach has the advantage that exception handling can be optimized immediately with ordinary optimization rules. This becomes important when the optimizer is inlining functions that do extensive exception handling.

In [Appel 1992] it is claimed that the second scheme has an important drawback which forced him to use the first variant in his compiler for ML: the runtime system might not be able to locate the current exception handler to raise a built-in exception like arithmetic overflow.

However, we think this drawback can easily be avoided by treating the exception argument in a special way during target code generation. In our compiler, this is done anyway with the continuation argument, which is passed as a return address on the stack at runtime with ordinary procedure calls and returns. Thus, the implementation has the same freedom as in the first scheme. Moreover, built-in exceptions like arithmetic overflow are handled in TML simply by using primitive procedures that accept an additional exception continuation.

Another drawback of the second scheme is that it does not handle asynchronous exceptions like interupts, but we think that this kind of exceptions should be handled somewhat different anyway, since a thread's stack can be unwound only when the evaluator is in a consistent state [Reppy 1990]. The same problem arises (and must be solved) by a special asynchronous exception: memory overflow, signaled by the underlying object store. When the garbage collector is invoked, every thread must be in a well-defined state.

We will transform the second scheme into a variant of first prior to target code generation (see section 5.1 about exception conversion).

2.4 Compilation Phases

The Tycoon compiler front end performs syntactic analysis, TL abstract syntax tree generation and semantic analysis (scoping and type checking). The remaining compilation tasks are performed by the comiler back end and are organized into the following phases:

1. TML code generation: translation of typed TL abstract syntax trees into untyped TML trees, including cell conversion for mutable variable bindings (see section 3.13).
2. α-conversion: renaming of variables to avoid name clashes in later passes, in particular for the extensive tree rewriting performed by the optimizer in the next phase.
3. TML code optimization (optional): this includes various optimizations like constant folding, dead code elimination and procedure inlining.
4. Exception conversion: removing exception continuation arguments from procedures and procedure calls.

5. Environment analysis: gathering information on the use of each variable, i.e. whether it is referenced by closures.

6. Closure conversion: eliminating non-local variable references.

7. Target machine code generation.

Note that these are only *conceptually* different phases. For efficiency reasons, several of them may be combined into a single pass of the compiler. Phases 1 to 2 as well as Phases 5 to 6 are currently merged into a single pass, respectively.

The translation of TL into TML (step 1) is the topic of the next section. During this pass, the α-conversion (step 2) of variable names is done as a side-effect of the TML tree construction. Since a new TML variable node is created for each bound variable, variables with the same name are distinguished by their different object identities (object identifiers) of the corresponding TML tree nodes.

After the TML tree is built, the (optional) optimization pass (step 3) performs various source language and target machine independent optimizations by extensive TML tree rewriting. We describe the generic re-write rules for TML in section 4.

The phases 4 through 6 rewrite the TML tree into a somewhat simpler form that is easier to handle by the code generator (step 7). The exception conversion pass (step 4) is described in section 5.1, whereas environment analysis (step 5) and closure conversion (step 6) are summarized in section 5.2. Finally, the generation of executable target machine instructions (step 7) is discussed in section 5.3.

3 Translating TL Source Programs into TML

In this section, we discuss some TL source code examples and their corresponding TML equivalents in order to give an idea of the overall translation process. For an introduction to TL see [Matthes and Müßig 1993]. When a given TL expression is translated, there is always a *current continuation* that denotes the continuation after evaluation of the expression. This *normal* continuation will eventually receive the computed value of the expression. Similarly, there is always a *current exception continuation* that denotes the continuation if a runtime exception is raised. Both continuations are initially given by the top level loop of the Tycoon system that interactively reads a TL expression, executes it and prints the result.

We will introduce the normal (current) continuation and the exception continuation as formal parameters *cc* and *ce*, respectively, to the translation function from TL to TML. We denote the translation function with

$$\ll E, ce, cc \gg \qquad\qquad T$$

in the following examples. It takes a TL expression E in addition to the two continuations *ce* and *cc*, and produces a TML tree T. This tree might in turn contain embedded recursive calls of the translation function itself.

3.1 Constants

Simple constants like numbers and string literals evaluate to themselves. This is expressed in CPS by an application of the current continuation, passing the constant as a parameter:

```
≪1,ce,cc≫                    (cc 1)
```

3.2 Bindings

Sequential and parallel bindings are translated into nested abstractions. Each value obtained by a recursive application of the translation function (passing a new continuation) is bound to a TML value identifier with the same name as the original TL identifier. We indicate the sequential evaluation of the two subexpressions A and B by printing them at the same indentation level, even though they are properly nested within the TML data structure:

```
≪let a = A let b = B,ce,cc≫   ≪A,ce, cont(a)      ; a is bound to the value of A
                              ≪B,ce, cont(b)      ; b is bound to the value of B
                              (cc b)≫≫            ; return the value of b
```

The translation of reecursive bindings is somewhat more comlicated and will be discussed in section 3.6. Mutable bindings will be discussed in section 3.13.

3.3 Expressions

Expressions that may contain embedded function calls are translated into CPS by *flattening out* nested subexpressions. Recall that, in CPS, arguments to procedure calls are restricted to be literal constants, temporary variables, procedures or continuations (cf. figure 2 on page 3). No nested procedure calls (function applications) are allowed:

```
≪F(A, B),ce,cc≫              ≪F,ce, cont(f)
                             ≪A,ce, cont(a)
                             ≪B,ce, cont(b)
                             (f a b cc)≫≫≫
```

In the following example, the result of the whole expression is bound to the temporary variable *t3*:

```
≪1 + {2 * 3} - 4,ce,cc≫      (* 2 3 cont(t1)
                             (+ 1 t1 cont(t2)
                             (- t2 4 cont(t3)
                             (cc t3))))
```

3.4 Functions

Each TL function is translated into a TML **proc** abstraction by adding two additional formal arguments to hold the two possible continuations of a function call: the continuation that

does exception handling, and the continuation that accepts the return value of the function. For example:

```
≪fun(x,y :Int) S,ce,cc≫          (cc
                                  proc(x y e c)
                                    ≪S,e,c≫)
```

Mutually recursive function bindings are translated into calls to the primitive procedure *Y*. This is the (multiple-value-return) CPS version of the λ-calculus fixed point operator. The abstraction given to the *Y*-primitive (cf. figure 2.2 on page 5) takes n abstraction arguments $v_1..v_n$ and a continuation abstraction c, 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:

```
(Y λ(v1..vn c)
    (c
     abs1
     ..
     absn
     cont()app))
```

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 neat effect of the *Y*-primitive is that the $n+1$ abstractions $abs_1..abs_n$ and **cont**()*app* are bound to the variables $v_1..v_n$ and c, respectively, and that these bindings are *visible within the abstractions themselves*. Moreover, the continuation **cont**()*app* that is bound to c is invoked tail-recursively after all the recursive bindings have been established.

A similar primitive is used in the ORBIT compiler [Kranz *et al.* 1986]. 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 [Steele 1986].

In the following example, two mutually recursive functions are bound to the identifiers f and g, and g is finally invoked:

```
≪ let rec f(a :Int):Int = F      (Y λ(f g c)
and g(b :Int) :Int = G               (c
g(13),                                proc(a e c) ≪F,e,c≫      ; function f
ce,                                   proc(b e c) ≪G,e,c≫      ; function g
cc≫                                   cont() (g 13 ce cc)))     ; invoke g
```

3.5 Aggregates

Aggregate value constructors are translated into appropriate calls to primitive procedures. The following code fragment creates a mutable array with two slots and binds the result to the temporary variable t:

```
≪array A B end, ce, cc≫          ≪A,ce,cont(a)
                                 ≪B,ce,cont(b)
                                 (array a b cont(t)
                                 (cc t))≫≫
```

3.6 Recursive Bindings

The translation of parallel recursive bindings is done in four consecutive steps:

1. Allocation of aggregates.
2. Initialization of variables for simple bindings.
3. Binding of variables for functions using the primitive procedure *Y*.
4. Initialization of aggregates.

Here is an example of a mutually recursive value binding:

```
≪let rec t = tuple f end
and f() = g(x f)
and x = h(99)
and g(a :Int b :Fun():Int) = t,
ce,
cc≫
```

```
(new 2 nil cont(t)    ; 1. Allocate aggregates
(h 99 cont(x)         ; 2. Inititalize simple bindings
(Y λ(f g c)           ; 3. Bind function variables
    (c
      proc(e c) (g x f e c)  ; function f
      proc(a b e c) (c t)    ; function g
      cont()                 ; 4. Initialize aggregates:
        ([]:= t 0 1 cont()   ; tuple variant
        ([]:= t 1 f cont()   ; tuple field
        (cc g)))))))         ; return the value of g
```

3.7 Conditionals

Conditionals are translated into calls to the primitive procedure '==', which does simple case analysis based on object identity similar to the C language **switch** statement (cf. figure 2.2 on page 5). Note how the current continuation is bound to the variable *join* in order to avoid duplicating code:

```
≪if A then
  B
else
  C
end,
ce,
cc≫
```

```
≪A,ce,cont(a)
(λ(join)
   (== a true
     cont() ≪B,ce,join≫
     cont() ≪C,ce,join≫)
 cc)≫
```

3.8 Loops

Loops are translated into tail recursive procedure calls. This is another application of the *Y* primitive. Note that, in TL, any form of loop statement (**loop**, **while**, **for**) yields the value **ok** which is similar to **nil** in other languages:

```
≪loop
  A
  if B then
    exit
  end
  C
end,
ce,
cc≫
```

```
(λ(exit)
   (Y λ(loop c)
        (c
          cont()              ; loop head (bound to loop)
            ≪A,ce,cont(ignore)
            ≪B,ce,cont(b)
            (== b true exit       ; exit test
              cont() ≪C,ce,cont(ignore)
              (loop))≫)≫≫     ; continue loop
          cont() (loop))          ; loop entry (bound to c)
  cont()                          ; loop exit label (bound to exit)
   (cc ok)) ; yield ok
```

Note that *loop* always denotes the same, statically determinable continuation, and that *loop* is used in functional position only (i.e. it does not escape [Appel 1992]). This property is also an invariant of the optimization rules. So this translation of loops into recursive continuations does not imply the introduction of first-class continuations in TML with associated runtime cost. There is no need to actually allocate a variable cell holding a reference to a "continuation object" for *loop*, and, since there are no **proc** abstractions between the loop head and the exit instructions (enforced by source level restrictions and optimization invariants), the loop exit and loop continuation instructions will be translated into simple jumps (C language gotos) without special tricks, even if the optimization phase is skipped.

3.9 Case Statements

Case statements are translated into calls to the primitive procedure '==' applied to the tuple variant code (an integer value). Note again that the current continuation as well as the case branches are encapsulated using continuations to avoid duplicating code:

```
≪Let T = Tuple
  case mon,tue,wed,thu,fri,sa,so
end
let t :T = X
case t
  when mon,tue then A
  when wed, thu, fri then B
  else C
end,
ce,
cc≫
```

```
(λ(join)
   ≪X,ce,cont(t)    ; evaluate tuple value
   ([] t 0 cont(v)    ; extract tuple variant
   (λ(c1 c2 c3)
        (== v 1 2 3 4 5 c1 c1 c2 c2 c2 c3)
     cont() ≪A,ce,join≫
     cont() ≪B,ce,join≫
     cont() ≪C,ce,join≫))≫
 cc)
```

3.10 Exception Handling

Exception handlers are translated into appropriate continuation passings: a function 'installs' a new handler by binding it to a continuation variable (denoted by *e* in the following example) and passing it to subsequent function calls:

```
≪try                                (λ(join)
  S                                   (λ(ce2) ≪S,ce2,join≫
  when E1 then V1                      cont(exc)              ; handler, bound to ce2
  when E2 then V2                       ≪E1,ce,cont(e1)
  else reraise                          ≪E2,ce,cont(e2)
  end,                                  ([] exc 0 cont(excId)  ; extract the exception id
ce,                                     (== excId e1 e2
cc≫                                      cont() ≪V1,ce,join≫      ; excId == e1
                                         cont() ≪V2,ce,join≫      ; excId == e2
                                         cont() (ce exc))))≫≫     ; else reraise
                                  cc)  ; current continuation, bound to join
```

A **raise** statement is translated into an application of the current exception continuation, passing an exception object as the argument:

```
≪raise E with V end,            ≪E,ce,cont(e)
ce,                             ≪V,ce,cont(v)
cc≫                             (vector e v cont(exc)  ; create exception object
                                (ce exc))≫≫            ; raise
```

3.11 C Language Function Calls

It is often necessary to be able to use existing external services written in other programming languages. The *ccall* primitive provides a way to call C language functions from within TL programs[3]. The inverse direction, i.e. calling TL functions from within C code, is implemented in C within the Tycoon runtime system, accessible for the Tycoon programmer via the ordinary *ccall* mechanism.

The *ccall* primitive takes a format string describing the parameter and result types (used for data representation conversion), an array object describing the desired C language function (essentially two strings describing the library and entry point where the compiled function code can be found), and a continuation which receives the return value.

The predefined polymorphic TL function *bind* can be used to establish a binding to a C language function. The following code fragment defines a TL function that calls the standard C function *strlen*. The function *raiseCCallError* is called to raise an exception if the *ccall* primitive fails. This may be the case when the given library does not exist, or it does not contain a function with the given name:

```
≪let strlen =                   (array "/usr/local/lib/libc.so" "strlen" cont(cfn)
  bind(:Fun(s :String):Int      (λ(strlen) (cc strlen)
    "/usr/local/lib/libc.so"     proc(s e c)
    "strlen"                      (λ (fail) (ccall "si" cfn s fail c)
    "si"),                         cont()
ce,                                 (raiseCCallError
cc≫                                  "/usr/local/lib/libc.so"
                                     "strlen" "si" e c))))
```

[3]These functions are typically bundled within a dynamic link library, even though it is possible to call C language functions that are statically linked to the Tycoon runtime system.

3.12 Builtin Functionality

Some TML primitives can be utilized explicitely from within TL source programs by means of the predefined polymorphic function *builtin*. This is useful for some frequently-used low-level operations like integer arithmetic for which an equivalent implementation using C calls would be too inefficient. The function *builtin* is a mapping of strings and TL functions of a given type to a function of the same type:

```
builtin :Fun(Dyn FctType <:Ok name :String ifFail :FctType) :FctType
```

This mapping is implemented within the Tycoon compiler. The name string tells the compiler about the desired semantics of the result, whereas the function parameter *ifFail* provides a convenient hook in the case the TML primitive fails. This leaves the relatively complex task of exception handling to the higher language level, rather than beeing hard-wired into the compiler.

To give an example, a TL function performing integer division can be defined using the "*int /*" builtin:

```
≪let / =
  builtin(:Fun(:Int :Int) :Int
    "Int /"
    fun(x,y :Int) raise intError),
ce,
cc≫
```

```
(λ(/) (cc /)
  proc(x y e c)
    (λ(fail) (/ x y fail c)
      cont()
        (vector intError cont(exc)
        (e exc))))
```

In this code fragment, the (globally defined) exception *intError* will be raised in case of a division-by-zero error.

3.13 L-Value Bindings

TML does not allow lambda variables to be modified once they are bound. This means that mutable variable bindings must be translated into explicit manipulation of the store by means of TML primitives. A store object (heap memory cell) is allocated for each mutable TL variable that holds the variables value at runtime, introducing a level of indirection for variable access (*Cell Conversion* [Kranz *et al.* 1986]):

```
≪let var a = A,ce,cc≫
```

```
≪A,ce,cont(t)      ; t is bound to the value of A
(array t cont(a)   ; allocate memory cell for a
([] a 0 cc))≫      ; return the value of a
```

In TL, it is also possible to pass a reference to a mutable variable binding (i.e. an *L-value*) to a user-defined function. Such a *call-by-reference* parameter is translated into two *call-by-value* parameters in the TML representation, passing base address and offset of the memory cell (or, in Tycoon Store Protocol terminology [Matthes 1993], the object identifier (OID) and the slot index):

```
≪let inc(var x :Int) = x := x+1
inc(a), ce, cc≫
```

```
(λ(inc) (inc a 0) proc(xBase
xOffset e c) ([] xBase xOffset cont(t1) ; retrieve value of
x (+ t1 1 cont(t2) ([]:= xBase xOffset t2 c)))) ; change
value of x
```

The offset of the mutable memory cell might become non-zero when mutable tuple components or array slots are passed as L-values:

```
≪let k = array 1 2 3 end
inc(k[2]),
ce,
cc≫
```

```
(array 1 2 3 cont(k)
(inc k 2 ce cc))
```

The TL assignment operator is defined as a polymorphic function within the programming environment. A call-by-reference parameter and the predefined function *builtin* is used in the definition. Note that the exception handling code will be removed by the optimizer as dead code:

```
≪let := = builtin(
    :Fun(A <: Ok var lhs: A rhs :A ):Ok
    ":="
    fun(A <: Ok var lhs: A rhs :A ):Ok
      raise exception "none"),
ce,
cc≫
```

```
(λ(:=) (cc :=)
  proc(lhsBase lhsOffset rhs e c)
    (λ(fail)
       ([]:= lhsBase lhsOffset rhs cont()
       (c ok))
      cont()
       (vector "none" cont(exc)
       (e exc))))
```

4 Optimization

The translation algorithm described in the previous section produces TML code that is far from 'optimal' in terms of code size and execution speed. A great deal of this inefficency stems from the translation process itself. We have tried to keep the translation process as simple as possible, so it generates a lot of unnecessary abstractions and λ-bindings. It is much simpler to run a separate optimization pass on the resulting TML tree rather than trying hard to improve the translation algorithm itself. This has the additional advantage that certain inefficiencies introduced by the TL programmer (in order to keep the TL program more readable, for example) will be removed also. And, as we will see in section 4.2, a small extension will enable us to optimize programs at runtime where a lot of valuable information is available to the optimizer.

We have organized the optimization phase into two separate passes, namely the 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 rule is applicable. Termination is guaranteed because each of the rewrite rules reduces the size of the TML tree if it is applied (therefore the name *reduction* for this pass).

The subsequent expansion pass tries to subsitute bound λ-abstractions (procedures or continuations) at the places they are applied. Effectively, this CPS transformation performs *procedure inlining* in terms of traditional compiler optimization. The decision whether a given use of a bound abstracion 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 we perform procedure inlining at all), so each expansion pass is followed by a reduction pass. Likewise, the reduction pass may reveal new opportunities to perform procedure inlining, 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.

4.1 TML Rewrite Rules

In this section, we give a formal definition of the generic TML rewrite rules. We will present them using the notation

$$A \xrightarrow{\textit{rule name}} B \qquad (\textit{precondition})$$

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

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:

$$\begin{aligned}
|v|_v &= 1 \\
|l|_v &= 0 \\
|p|_v &= 0 \\
|v_1|_{v_2} &= 0 \quad (v_1 \neq v_2) \\
|\lambda(v_1..v_n)\, app|_v &= |app|_v \\
|(val_0\ val_1..val_n)|_v &= \textstyle\sum_{i=0}^{n} |val_i|_v
\end{aligned}$$

Similarly, on the right side of a TML rewrite rule, we use the notation $E[val/v]$ that denotes the expression E where every occurence 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: 1. at the original place, namely as a parameter to the λ-application, and 2. at the place the abstraction was substituted for the variable. However, this does not do any harm because the first occurence 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:

$$
\begin{array}{rcl}
v[val/v] & = & val \\
v'[val/v] & = & v' \quad (v \neq v') \\
l[val/v] & = & l \\
p[val/v] & = & p \\
\{\lambda(v_1..v_n)\, app\}[val/v] & = & \lambda(v_1..v_n)\, \{app[val/v]\} \\
(val_0\ val_1..val_n)[val/v] & = & (val_0[val/v]\ val_1[val/v]..val_n[val/v])
\end{array}
$$

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 that may cause side effects in the store.

The complete set of the TML rewrite rules that 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 and procedure inlining are just special cases of these general λ-calculus transformations.

The *subst* rewrite rule replaces each occurence 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 arbitrarily large:

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

The *remove* rewrite rule strikes out a bound variable v_i that 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 2 on page 3), val_i cannot be an application, and, therfore, cannot contain any calls to side-effecting primitive procedures:

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

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

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

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

$$
\lambda(v_1..v_n)(val\ \ v_1..v_n) \xrightarrow{\eta\text{-}reduce} val
$$

The *fold* rewrite rule uses an evaluation function *eval* that 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:

$$(==\ \ v\ \ val_1..val_n\ \ val_1^c..val_n^c) \xrightarrow{case\text{-}subst} (==\ \ v\ \ val_1..val_n\ \ val_1^c[val_1/v]..val_n^c[val_n/v])$$

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

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

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

$$(Y\ \ \lambda(c)(c\ \ \mathbf{cont}()app)) \xrightarrow{Y\text{-}reduce} app \qquad (|app|_c = 0)$$

4.2 Extensions for Reflective Runtime Optimization

In the Tycoon system, the compiler (and, therefore, the optimizer) is an integral part of the persistent programming environment. This enables us to call the optimizer *statically*, during normal program compilation, or *dynamically* at runtime.

For each TL function *f*, the target code generator emits code that is augmented with a reference to a compact persistent representation of the TML tree for *f*. This persistent CPS representation is called PTML (Persistent Tycoon Machine Language) and is isomorphic to TML.

At runtime, it is possible to map PTML expressions back into TML expressions, re-invoke the optimizer and code-generator. Since this compilation takes place during program evaluation, we call our approach *reflective optimization*.

The mapping from PTML back to TML also returns the set of R-value bindings established at runtime. These bindings correspond to free variables (module names, function names, constant names) 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, this is a Tycoon interface *complex* that exports an abstract data type *complex.T* and some 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* that uses these exported functions:

```
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 availabe. 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* that is equivalent to the original function *abs* but that (hopefully) executes faster than the original by explicitly invoking the optimizer on *abs*:

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

In our current implementation, the compiler would inline 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 can be applied:

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

The main extension that 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* (cf. section 5.2) of the runtime representation of a given procedure. For the example above, this means that the values of the variables *complex*, '+', '*' which are global to the function *abs* globaland *sqrt* are fetched from the closure record of *abs* and are bound to the corresponding identifiers *before* the (original) body of *abs* is executed[4]:

```
proc(c_10 c_11)
  (λ(complex_6 *_7 +_8 sqrt_9)
     ([] complex_6 2 cont(t_12)          ; here begins 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))))))))))))          ; here ends 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
```

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

5 Target Machine Code Generation

The tree-structured intermediate TML code generated so far is easy to re-write (by the optimizer), but cannot be executed efficiently. We need some linear program representation

[4] 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.

that is easy to process by some (real or abstract) target machine.

Currently, we have implemented a code generator for a stack-based abstract machine (TVM, *Tycoon Virtual Machine*) using byte-coded instructions, and an experimental ANSI C code generator. We will sketch the C code generator at the end of this section in order to discuss the major topics. But before the code generator is called, three additional passes are made through the TML tree: exception conversion, environment analysis and closure conversion. These passes simplify the task of the code generator by rewriting the TML tree into a form more biased towards stack-based target machines.

5.1 Exception Conversion

The exception conversion pass removes the exception continuation argument from every *proc* abstraction and application and inserts an appropriate sequence of calls to the primitive procedures *pushHandler* and *popHandler*. This isolates the target code generator from exception handling details.

The primitives *pushHandler* and *popHandler* maintain a runtime stack of exception handlers (cf. section 2.3). These exception handlers are represented in TML as continuations that receive the exception object as an argument. At the target machine level, these continuations will be represented as assembly language labels. A *pushHandler* call will push the value of this label (i.e. a code pointer) onto the exception handler stack, and a corresponding *popHandler* call will eventually remove it. If an exception is raised, control will be transferred immediately to the continuation on top of the exception handler stack.

In the following example, an exception handler is bound to the continuation variable *e*. Subsequently, the exception *intError* is raised by passing a newly created exception object to the exception handler bound to e. This handler is also passed in a nested call to the procedure *p*. The exception handler itself issues a *reraise* statement after processing the exception in the statement *D*. After exception conversion, the exception continuation argument to the procedure call is removed and passed implicitely on the exception handler stack:

```
proc(ce cc)
 (λ(e)
    A
    (vector intError cont(err)
    (e err))      ; raise err
    B
    (p e cc)      ; call p
    C
  cont (exc) ; handler e
    D
    (ce exc)      ; reraise exc
```

```
proc(cc)
 (λ(c e)
    (pushHandler e cont()
    A
    (vector intError cont(err)
    (raise err))
    B
    (p c)       ; call p (no exception argument any more)
    C
  cont(t)       ; continuation c
    (popHandler cont()
    (cc t))
  cont (exc)    ; handler e
    D
    (raise exc))    ; reraise exc
```

5.2 Environment Analysis and Closure Conversion

Similar to Scheme [Steele 1986] and ML [Milner *et al.* 1990], the Tycoon language TL and its intermediate representation language TML are block structured languages with functions (procedures) as first class values. In such a language, a function f may be lexically nested within another function g. Function f may access its formal parameters and local variables, but it may also reference a variable v defined in the lexically enclosing function g. We say that v is a *bound* or *local* variable with respect to g, and a *free* or *global* variable with respect to f:

```
≪let g(v :Int) = begin                (λ(g) (cc g)
  let f() = v                           proc(v e c)          ; function g
  f                                       (λ(f) (c f)        ; return the value of f
end,                                        proc(e c)        ; function f
cc,                                           (c v)))        ; return the value of v
ce≫
```

The Tycoon compiler has to generate efficient code for this kind of variable access. The task of the compiler is further complicated by the fact that the lifetime of v bound in function g may exceed the activation of g, since function f may be stored in data structures and be applied later, after the enclosing function g has already returned. Therefore, the values of free variables cannot be allocated on the stack, but must be allocated on the heap.

In the Tycoon system, the runtime representation of a function value is a heap data structure called a *closure record*. A closure record is essentially an array containing a pointer to the executable code of the function, and the value of each free variable referenced in the body of the function. The executable code knows the position of each free variable within the closure record. This is the *flat closure* approach to function representation [Appel 1992]. Flat closures trade closure creation time for fast access of free variables. This is important in persistent object systems where object slot access typically imposes some overhead [Mathiske 1992].

The flat closure approach requires that variable values may be copied freely and spread out into multiple locations. This is possible in TML since variables are never modified once they are bound. During TML code generation, any references to mutable TL variables have been cell-converted into explicit operations on the store (see section 3.13).

The *environment analysis* pass of the Tycoon compiler determines the set of free variables for each *proc* abstraction, and allocates a slot within the closure record for each. This pass also identifies references to literal objects that cannot be represented directly within the byte sequence of the executable code, but must be fetched from the heap at run time. A slot within the *literal vector* of the function is allocated for such an object. To illustrate this, the above example function f is slightly modified to return a pair consisting of the global variable v and a literal string object:

```
≪let g(v :Int) = begin                (λ(g) (cc g)
  let f() =                             proc(v e c)              ; function g
    array v "string literal" end           (λ(f) (c f)
  f                                          proc(e c)          ; function f
end,                                           (array v "string literal" c)))
cc,
ce≫
```

Following environment analysis, the *closure conversion* pass removes every reference to free variables by explicit closure allocation, passing and access (*Closure Passing Style* [Appel 1989]). This isolates the later passes from closure format details.

At runtime, each function will receive it's closure record as an additional parameter, making the stored free variable values accessible to the executable code. Moreover, every reference to a literal object is translated into an indexed access to the literal vector. The runtime representation of f is depicted in figure 5.2. We say that the resulting TML code is *closed* because it does not contain any free variables any more:

```
(λ(g) (cc g)
  proc(v e c)                    ; function g
   (λ(f) (c f)
     proc (e c)                  ; function f
       (array v "string literal" c)))
```

```
(array                           ; create closure record for g
  proc(env v e c)                ; function g
    (array                       ; create closure record for f
      proc(env e c)              ; function f
       ([] env 1 cont (t1)       ; fetch value of v into t1
       ([] env 0 cont (t2)       ; fetch literal vector into t2
       ([] t2 1 cont (t3)        ; fetch string literal into t3
       (array t1 t3 c))))        ; return array
      v                          ; free variable value
      c)
  cc)
```

Traditionally, the formal parameter for the closure record is given the name *env* because it actually contains the *lexical environment* of the function at the time the closure was created, i.e. at the time the TL source code of the function was evaluated. Note that function g does also receive a closure record even though it does not contain any free variables. This is necessary because g may be called from sites that do not know this property of g. Therefore, the standard Tycoon calling sequence must be used, passing an explicit closure record.

A somewhat more complex translation scheme is necessary for the closure conversion of mutually recursive function bindings in an application of the Y primitive. Essentially, additional TML code is generated that, after all closure records have been created, patches each closure record with the references to the closure records of all the other functions. This code generation actually implements the semantics of the Y primitive, i.e. it *computes* the least fixed point, a vector of mutually recursive functions.

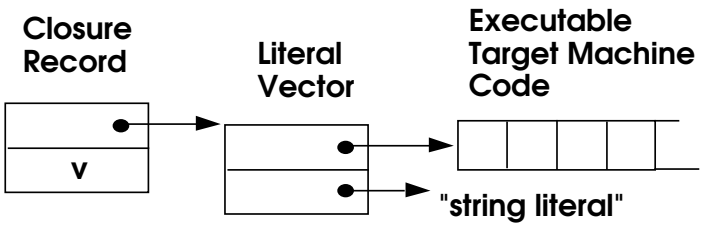

Figure 4: Closure Representation

5.3 Generating ANSI C Code

After all the processing described in the previous sections, C code generation is almost a trivial task. Since there are no more free variable references, each **proc** abstraction can be translated separately into an equivalent C language function, and continuations are 'flattened out' by translating them into a label followed by C code implementing the body of the continuation. References to continuation variables are translated into C language gotos, optionally preceeded by an assignment of the continuation argument to a temporary variable. Live variable analysis as described in [Wulf *et al.* 1973; Gawecki 1992] helps reducing the number of temporary variables actually needed[5].

Note that the last argument of a **proc** abstraction (user level procedure) is always the continuation argument:

```
proc (x y c)                    OID f(OID x, OID y)
  (λ(c2)                          {
      ...(c2 88)...               OID t;
    cont(t)                       ...
      (c t))                      t = MAKE_INT(88);
                                  goto c2;
                                  ...
                                c2:
                                  return t;
                                  }
```

Ordinary bindings are translated into local blocks:

```
(λ(x y) ...                     {
  88                            OID x = MAKE_INT(88);
  99)                           OID y = MAKE_INT(99);
                                ...
                                }
```

Procedure calls are translated into C language function calls, storing the result in the temporary variable allocated for the continuation, and passing control to the continuation:

```
(f x y c)                       OID t;
                                ...
                                t = f(x y);
                                goto c;
```

The translation of calls to primitive procedures is fairly straightforward using equivalent C language expressions and/or appropriate runtime system calls.

As an example, consider a call to the primitive procedure '=='. If all the values of the different branches are simple integer constants (or can be encoded as such, like character values), the C language *switch* statement can be used. C compilers typically generate very efficient code (jump tables) for this kind of statement:

[5]Since this optimization is usually done by optimizing C compilers anyway, we have not implemented it as a part of the C code generator.

```
(== x
  1  2  3  4  5
  c1 c1 c2 c2 c2
  c3)
```

```
switch (x)
  {
  case 1:
  case 2:  goto c1;
  case 3:
  case 4:
  case 5:  goto c2;
  default: goto c3;
  }
```

If some of the branch values are object identifiers, a cascaded *if* statement must be used, fetching the object identifiers form the literal vector of the current function. Note that object identifiers are not constant values since they may be changed at any time by the underlying object store. This might happen if the object store uses a compacting garbage collector.

6 Concluding Remarks

The concept of using the same intermediate program representation for both static *and* dynamic optimization has proven to be feasable. We have a running protoptye that performs all the optimizations presented in this paper. Our compiler has successfully optimized itself, i.e. the complete TL compiler and library management code spread out in 98 modules containing more than 29,000 lines of TL source code. Initial performance measurements indicate that the speed of such non-trivial modular programs as the TL compiler could (at least) be doubled by dynamic, inter-module code optimization.

On the other hand, the space requirements of TL modules have nearly doubled. The main reason for this space overhead is the additional persistent encoding of the TML tree (in a data structure called PTML, Persistent TML) for each TL source code function. We are currently investigating techniques to re-construct a TML prepresentation by examining the persistent executable code representation of a procedure, effectively inverting the target machine code generation process outlined in section 5. In general, the TML tree re-constructed this way will not be isomorphic to the original TML tree that we currently encode in PTML. The interesting question is whether this has an impact on the possible optimizations, especially in the presence of nested recursive function bindings.

We will investigate further more global optimizations like code motion (hoisting), data flow analysis and strength reduction in order to estimate the potential and the limitations of general-purpose program optimization in open database programming environments. This approach may be regarded as an alternative to the well-understood yet limited algebraic optimization rules of the relational algebra. We conclude by emphasizing that in our framework a main target for optimization is represented by libraries of higher-order functions that encapsulate iteration abstractions (SQL-like *select-from-where*, i.e. *loops*) and by operations over bulk data structures (sets, lists, bags, ...).

References

Albano et al. *1993:* Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. An introduction to the database programming language Fibonacci. FIDE Technical Report Series FIDE/92/64, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.

Appel 1989: Appel, A. W. Continuation-passing, closure-passing style. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, January 1989.

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

Gawecki 1992: Gawecki, A. Ein optimierender Übersetzer für Smalltalk. Bericht FBI-HH-B-152/92, Fachbereich Informatik, Universität Hamburg, Germany, September 1992.

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.

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

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.

Mathiske 1992: Mathiske, B. Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, December 1992.

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

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

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

Milner et al. *1990:* Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

Reppy 1990: Reppy, H. J. Asynchronous signals in Standard ML. TR 90–1144, Computer Science Department, Cornell University, 1990.

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.

Wulf et al. *1973:* Wulf, William A., Johnsson, Richard K., Weinstock, Charles B., and Hobbs, Steven O. The design of an optimizing compiler. Technical Report AFOSR-TR-74-0096, Carnegie Mellon University, Air Force Office of Scientific Research, 1973.

A TML Data Structures

```
(* TML Tree: *)

  infiniteCost :Int            (* Value representing infinite evaluation cost
                                  for a TML tree *)
  defaultInfo :Int             (* Value representing usefule default information
  for the optimizer *)

  Let Rec EvalFn = All(:T)T    (* a meta-evaluation function *)
  and CostFn = All(:T)Int      (* cost estimate for a call *)

  and T = Tuple            (* a TML node *)
    case primitive with    (* a primitive procedure *)
      name :String         (* user name, printing only *)
      info :PrimitiveInfo  (* attributes useful for the optimizer *)
      eval :EvalFn         (* a meta-evaluation function *)
      cost :CostFn         (* cost estimate *)
    end

    (* Note that the functional position of apply nodes (node!apply.fn)
       is the only place where primitive nodes may appear.
       I.e., they cannot be passed around as values.
    *)

    case variable with
      name :String
      fCont :Bool          (* true iff this variable names a continuation *)
    end

    case literal with
      value :Literal
    end

    case readRef with     (* variable reference *)
      variable :T
    end

    case lambda with
      variables :list.T(T)      (* linked list of formal paramaters *)
      body :T                   (* must be an appliation *)
    end

    case apply with
      fn :T                     (* primitive, readRef, literal or lambda node *)
      args :list.T(T)           (* linked list of actual parameters *)
    end
  end (* T *)

  and Literal = Tuple
    case nil with end
    case literals with value :Array(Literal) end
```

```
  case tml with value :T end
  case bool with value :Bool end
  case char with value :Char end
  case int with value :Int end
  case real with value :Real end
  case string with value :String end
  case byteArray with value :OID end
end
```