Threaded Language As a Form of Partial Evaluator

Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University
Bangkok 10330, Thailand
Email: prabhas@chula.ac.th

Abstract

This work describes a class of language called Threaded Language and its implementation. An interpreter for this language allows it to be easily extended. Higher order functions and local variables are introduced into the language as extensions. Exploiting the dynamic nature of an interpreter, it is shown how to do a program transformation. Using the "unfold" technique, a partial evaluation in this language is demonstrated. The partial evaluation generates a specialised version of the input program that runs faster. Finally, the implication of using this language to bootstrap a language on a new environment is discussed.

Key Words: threaded language, partial evaluation

1. Introduction

Writing programs to generate another programs are very useful. The examples for such programs are a compiler and a code generator. A compiler translates a source language into a target language. A code generator translates parse trees or an intermediate language into machine codes. The process of compiling a program by a compiler usually consisted of many stages. These stages work in a batch-like manner. The final stage produces executable codes. In contrast to a compiler, an interpreter reads a source program and executes it immediately.

There are two main distinctions that are of interest between a compiler and an interpreter. Firstly, the output of an interpreter is executed immediately. This fact makes it possible to use the action to participate in the translation process of the source program. Secondly, an interpreter is much easier to write than a compiler. The initial effort to bring up a working interpreter is much less than writing and debugging a compiler. This is because an interpreter works directly with tokens of the source program rather than handling intermediate representation usually required by a compiler. Of course, there is a penalty associated with using an interpreter, the speed of execution. An executable code produced by a compiler runs much faster than using an interpreter on the same source program.

I have two aims in this paper. First, I want to explore a class of language, called *Threaded Language*, which has a structure that fits very well with an interpretive implementation. Second, I want to illustrate the application of a program transformation, called *partial evaluation*, to generate a faster version of a program produced by an interpreter. This is done using the Threaded Language itself.

The presentation is divided into 4 sections. Section 2 defines Threaded Language including the details of its interpreter. Section 3 explores simple extensions of the language by means of an interpretive technique. Section 4 show a simple scheme of partial evaluation written in Threaded Language. The next section discusses the implication of Threaded Language and finally, I relate the idea in this presentation to other works.

2. Threaded language

2.1 Definition

A threaded language (TL) is composed of functions. There are two types of functions in TL: primitives, and definitions. A primitive contains the actual machine codes. A definition contains a list of "threads" (or the pointers to functions). A thread can be either a primitive or a definition.

TL = definition
definition = primitive | definition

where
primitive contains the actual machine codes.
definition is list of pointers to functions

The data structure ("form") that represents primitives and definitions is designed so that the evaluation (execution) of both kinds is uniform. The first location of a function is called a "head". The last location is called a "tail" (see Fig. 1). As the form is dependent on the evaluation, the evaluator of a TL will be explained first.

An evaluator of a TL implements the flow of threads by exploiting one stack (called "return stack", R) for storing the continuation points of function calls and using one machine code routine, “enter” and one primitive, “exit”. The flow of control employs two pointers: the instruction pointer (IP) and the function pointer (FP). The instruction pointer, IP, is a kind of "program counter" that points to the definition being evaluated. A program in TL is a list of pointers to functions. So, the actual "instruction" of a program is reached by dereferencing IP. This value is kept in FP. To reach an actual machine code, FP is dereferencing once again. Let `@A++ -> B` denotes a dereference of A put to B and post-incrementing A, pc denotes the actual machine code program counter.

To evaluate (execute) a function:

```
@IP++ -> FP, @FP++ -> pc
```

Now, we must present the "form" of TL. The head of a primitive is the pointer to the next address (denoted by *). The tail of a primitive is a machine instruction to jump to the "next" function. The body of a primitive consisted entirely of machine codes. The head of a definition is the pointer to the "enter" routine. The tail of a definition is the pointer to a primitive "exit". To avoid confusion between names that denote difference objects, let us use the following notation. let **“name”** be the name of a function as it appears in the source program (printed name), let **&name** be a reference to a functon (a primitive or a definition), let ***name** be a reference to a machine code routine. The Arial font will be used for the text that is the source program.

```
       definition            primitive

IP ->  *enter    +--> FP ->   *
       &FP1    __|           machine
       &FP2                  codes
       ...                   ...
       &exit                 jmp next
```

* denote the next address

Figure 1 The structure of definitions and primitives

The head is a special place, it contains the machine code to set up the evaluation. The "enter" is a machine routine, it saves the current IP and enters a function call pointed to by FP.

```
enter:
push IP -> R, FP -> IP, jmp next
```

The "next" machine code performs fetching of a thread and execute it.

```
next:
@IP++ -> FP, @FP++ -> pc
```

The tail of a definition performs a "return" to the caller, hence "exiting" a definition. The "exit" is a primitive. It restores the IP from the return stack and continue to evaluate the next function.

```
exit:
* , pop R -> IP, jmp next
```

The head of a primitive contains the pointer to the actual machine code in its body. All primitives end with a jump to "next" hence evaluating the next function. The FP is important, before the evaluation it points to the head of a function to be evaluated, once the evaluation starts, it is then advanced to point to the body of function. The actual machine code is executed by the second half of "next" using FP, `@FP++ -> pc.`

The "enter", "exit" and "next" are the crux of the evaluator of TL. It allows a higher form of language to exist as definitions.

2.2 Compiling

Compiling a "source" language into a correct form (threads) can be achieved by an evaluator (an interpreter). The evaluator can be very small as most of the work is distributed to definitions themselves which are actively participate in generating threads.

The process of compiling is unlike a conventional compiler. It is more like applying a function to generate definitions which will be evaluated to create more definitions. Primitives are pre-existing routines to build the first evaluator. The process is dynamic. The code is generated from the source, some of which is executed to generate more code. This is in contrast with a conventional compiler where the process of compiling is more batch-like.

An evaluator employs an evaluation stack (E) to store all the intermediate results. It has a state called "mode". The evaluator can be in either mode: a

compile mode or an execution mode. The compile mode is established when a type of definition, called defining functions, appears in the source. When a definition is ended, the mode is changed to execution. For example, here is the source of a postfix form TL:

```
def add2
 2 +
end
11 add2 print
```

The "def" and "end" are defining functions. They define the token "add2" as a new function. Its body is the thread of "2" and "+" functions. "11 add2 print" is in the execution mode. It is immediately executed. Please note that the source is a mix of both defining new functions and executing them.

In the compile mode, most tokens read from the source will be translated into pointers to functions. In the execution mode, a token will be searched in a list of existing definitions (store in a symbol table) to find its reference (FP) and then it will be executed (using the "next"). Some definitions are executed in the compile mode. They are the part of the compiler. There are five possible actions in an evaluator. An evaluator can be described as follows:

```
eval
  read a token from source
  if it is a number
    if execute mode
      push it to E   (1)
    else
      include its handler in def_n (2)
  else
    if search is not found
      stop with error  (3)
    if execute md or compiler function
      execute it  (4)
    else
      include its ref in def_n(5)
  eval
```

The example of source above will be compiled into the following definition.

symbol table

```
"add2"  $2
```

definition

```
$2  *enter
    &number
    2
    &add
    &exit
```

Where &number, &add are the pointers to the appropriate functions (either a primitive of a definition). The tokens "11", "add2" and "print" will be executed immediately under the execution mode. With this basic evaluator, now simple extensions will be demontrated.

3. Language extension

3.1 Higher order function

Compiler functions are examples of higher order functions. Collectively they create another function which can be executed. A "def" and an "end" do:

```
def
  change mode to "compile"
  read the next token from source (1)
  create entry in the symbol table (2)
  alloc an open-end space for storing
    the definition (3)
  put "*enter" at the head

end
  put "&exit" at the tail
  change mode to "execute"
```

Between "def" and "end" is the body of a function. The body of definition will be filled in by the evaluator evaluating the source.

A generic defining function can be defined as follow.

```
def def-generic
  create pack machine
  <machine code>
end
```

There are three functions to be explained: "create", "pack" and "machine".

```
create
  do the (1) (2) (3) of "def" above

pack
  pop a value from E,
  put it in the body
```

```
machine
  similar to "end" but
  put "mcode" at tail instead of "exit"
  read the <machine code> until "end"
  fill them in the body of def-generic
  change mode to "execute"
```

The function "machine" is a compiler function. The <machine code> that followed is not executed. It is put into the body. The definition of "def-generic" is as follows:

symbol table

```
"def-generic" $3
```

definition

```
$3     *enter
       &create
       &pack
       &mcode
       <machine code>
```

The primitive "mcode" will be explained after we show how this "def-generic" is used to create a new function. As an example, we will create a new defining function "variable" which is used to create a global variable.

```
def variable
  create pack machine
  <machine code for variable>
end
```

The definition of "variable" is:

symbol table

```
"variable" $4
```

definition

```
$4        *enter
          &create
          &pack
          &mcode
$5        <machine code for variable>
```

The "variable" is used (executed) as this:

```
10 variable xyz
```

Using "create" the "variable" function will create the following definition:

symbol table

```
"xyz"  $6
```

definition

```
$6        *
          10
```

The "pack" includes a value from E to the body. Then, the "machine" primitive does placing the reference to the <machine code for variable>, $5, in the head of this new definition. When "xyz" is evaluated, its machine code is activated (by the pointer in the head). The machine code for "xyz" will just leave its value on E. "xyz" acts like a global variable. The value storing in its body is an initialised value. A compiler function "**->**" is defined to set a value of a variable.

```
->
  read next token from source
  search its reference
  include "&set reference"

set
  *
  pop a value from E
  store it to reference+1
```

"xyz 1 + -> xyz" will increment xyz. Please note that "`->`" must be used in the compile mode only. A "type" can be implemented by using different <machine code> appropriate to the desired type of function.

3.2 Local variables

We have defined a language where everything are functions where all intermediate values passing between them are in the evaluation stack, S. A function to create a global variable is defined, "variable". How to include local variables in the language? Local variables can be stored in R (similar to a stack frame in a conventional compiler). A pointer "Local Stack" (LS) stores the base address of this frame. A reference to a local variable is an offset from LS. Two primitives are defined for local variables: lget, lput.

```
lget i
  push LS[i] to S

lput i
  LS[i] = pop S
```

where i denotes a reference to a local variable. A compiler function, "[" is used to create local variable of a function. It reads the source and build a local symbol table with references 1..n until the token "]" is found. All tokens between "[" and "]" are local variable names. The behaviour of a local variable is defined as follows. When a local variable name appears in the source, it is compiled into "lget". To update a value to a local variable, a function "->" is defined. It must be modified slightly from the previous definition that deals with a global variable only.

```
->
  read next token from source
  if search local symbol table is found
  then
    include "&lput ref" in def_n
  else
    search global symbol table
    include "&set ref" in def_n
```

The following is an example of a function with local variables.

```
def haslocal [ x y ]
  1 -> x
  2 -> y
  x y +
end
```

The above function is compiled into the following definition:

local symbol table
```
"x"   1
"y"   2
```

symbol table
```
"haslocal"  $7
```

definition

```
$7      *enter
        &frame 2
        &number 1
        &lput 1
        &number 2
        &lget 1
        &lget 2
        &add
        &remove
        &exit
```

where "frame n" and "remove" are primitives that create a frame of size n, and remove the frame.

The evaluator has to be modified slightly to search a local symbol table first then the global symbol table. When a local name is found it includes "lget ref" into the definition (modifying the (5) action of the evaluator). The "def" and "end" needed to be changed slightly too, to handle "frame" and "remove".

A new evaluator can replace the old evaluator using the observation that an evaluator is a function hence its definition is like this:

definition

```
&eval ->  *enter
$8        *eval
          *jmp .-1
          &exit
```

Please note that an evaluator runs in an infinite loop. Its "exit" is never executed. We define a function "neweval" to replace the "*eval" (at $8) with the reference to a new evaluator "&eval2" (assuming it just has been defined).

```
def eval2
  ...
end
neweval
```

3.3 Example of a TL

A simple language in postfix form is defined using a few defining functions. These are compiler functions:

```
def end if else { } while ->
```

The control flow functions: if else { } while, compile the appropriate handlers (jmp, jmp-if-false) into the body of definition. Other functions are of general use. They are simple to be implemented as primitives ("variable" is as defined previously).

```
variable + <= print
```

For the evaluator we need these machine code routines:

```
enter next exit number search set
```

For local variables the following functions are needed:

compiler functions: [ ] ->
machine code: lget lput frame remove

Here are some example of language use:

```
0 variable count

def one-to-ten
  1 -> count
  while count 10 <= {
    count print
    count 1 + -> count
  }
end

def rec-one-to-ten [ n ]
  -> n
  if n 10 <= {
     n print
     n 1 + rec-one-to-ten
  }
end

one-to-ten
1 rec-one-to-ten
```

The beauty of this example is that the compilation of this source and its execution is contained in the evaluator (eval function) and all pre-defined functions above. There is no need for any other function to compile and run this source!

4. Partial Evaluation

A partial evaluation [1] is program optimisation technique, so called "program specialisation". It is a method to generate a specialised version of a program. It takes two inputs: the source program, and a part of its input, it then outputs a program that is usually longer than the original but runs faster. Fig 2 shows a partial evaluator takes two input: a program p, and its input in1. It constructs a new program p_in1, which will yield the same result that p would have produced given both inputs.

```
                  static input in1
                       |
                       v
a program
 p          ->  partial evaluator
                       |
                       |
                       v
dynamic
input in2   ->  specialised   -> output
                program p_in1
```

Figure 2. A partial evaluation due to [1]

We can apply a partial evaluation with TL. We need to execute a function as usual but a "trace" of execution is recorded. This can be achieved with a modified evaluator. With this "trace" a new version of program can be generated using a defining function. We give a simple example of the process. Given a program "power" that computes x^y where x is a global variable and y is a local variable and a static input "3 power".

```
 def power [ y ]
1   -> y
2   if y 1 == {
3      x
4   else
5      x  y 1 - power *
    }
 end
```

The specialised version of "power" will be

```
 def power3
  x x x * *
 end
```

To achieve a partial evaluation, the technique of "unfolding" function calls [2] is used. We record the binding of the original program with respect to its input "3 power", hence the syntactic entities of interest are: "x" and "*". We modify the evaluator to record these two entities, the record is called a "trace". Here is the trace of executing "3 power".

```
(y = 3)
1 2 4 5
(y = 2)
1 2 4 5
(y = 1)
1 2 3
```

At line 3 and 5, the "x" and "*" are recorded. Here is it.

```
"&x" "&x" "&x" "&*" "&*"
```

We can define a defining function to take this trace and generate a function.

```
 def power3
  x x x * *
 end
```

Another use of a partial evaluation is to generate an "unfold" version of a function. In our case, a definition can be unfolded into primitives. This will speed up the execution of a program because the overhead associated with "threading", the "enter", "next" and "exit" can be reduced or bypass. This is similar to compile to machine codes. In order to do this, the evaluator must be able to understand the

meaning of some machine codes so that it will be able to take appropriate "execution" of those instructions and records its effect.

A defining function to generate a primitive is as follows.

```
def primitive
  create
  packcode
end
```

Where "packcode" is similar to "machine" but does not include "mcode". It reads the source until "end" and packs the machine code into the body of the function being defined. Here is a sample of its use.

```
primitive power3p <machine code> end
```

The part <machine code> is generated by applying partial evaluation to "power3" and record its primitives execution.

5. Discussion

I have explore the design of a Threaded language and show its implementation. Because of its dynamic execution it is easily applied to generate programs. I will discuss two aims I set out from the beginning of this presentation.

First, the language, the structure of TL as definitions allows the representation to be uniformly interpreted. The flow control of TL is encapsulated into a few mechanisms ("enter", "next", "exit"). This structure gives rise to a "higher level" language, i.e. the definition. Once the first evaluator (interpreter) is written, the subsequent programming can be done with this "higher level" language (define new functions).

Second, the application of TL as a partial evaluator can be easily achieved. This is because the nature of TL as an interpreter. The defining function plays an important role in the flexibility. The ability to take some appropriate input and generates a program out of it. This allows a partial evaluator to be constructed naturally in TL.

Now I will elaborate on the implication of this presentation. The implementation effort to bring up a TL language is very minimal. Each primitive can be implemented in around 10 lines of assembly code. The "eval" function is implemented in 50 lines of assembly code (it can be done as a definition but something has to be pre-existed for it to work). The accompany symbol table search and the scanner for a source can be implemented conventionally. The crux of TL is just a few machine code instructions (as it should be as it is the bottleneck of the performance of a TL). To contrast this with a compiler. I have a compiler system written in the same machine code that I wrote TL (s-code [3]). The source is publicly available (Som version 2.0 [4]). The compiler source code is about 2000 lines. Of course, this is not a very fair comparison because the compiler and tke interpreter do not "compile" the same language (one being Som language, another is TL). But, both languages are not too different as they are simple languages for teaching purpose.

This use of TL can be very beneficial to "boostrapping" a language in a new environment. For example, to build a new embedded system, an engineer usually selects a processor which has a rich set of accompanying compiler, assembler, monitor to create and debug software for a new platform. This is appropriate. However, if the engineer wants to choose a custom-made processor, then he or she will face with an enormous task of creating an appropriate tool chain. Here is where a TL style can be of help.

The result of partial evaluation is quite impressive. Here are the comparison of "power", "power3" and "power3p" on the same dynamic input. The speed is measured in terms of the number of machine instructions executed to complete the task.

power	492 instructions
power3	196 instructions
power3p	21 instructions

6. Related work

The term "threaded code" was used as a technique to write an interpreter. It can be traced far back many years ([5] in 1973). A. Ertl [6] discussed many methods to implement "threaded code". My threaded language is influenced heavily by this history. The implementation presented here is partly similar in style to the one in the book [7] which was also influenced by the FORTH language invented by Charles H. Moore [8, 9]. The discussion on the topic of boostrapping a new system is explored in depth in my earlier work [10].

The code and the executable system for this experiment is available publicly on my website, at the project topic "Threaded Language" [11].

7. References

[1] N. Jones, "An introduction to partial evaluation", ACM Computing Surveys, Vol 28, Issue 3, September 1996.

[2] R. Burstall, and J. Darlington, “A transformation system for developing recursive programs,” J ACM, 24, 1, Jan. 1977, pp.44-67.

[3] http://www.cp.eng.chula.ac.th/~piak/project/som/s-code.htm

[4] http://www.cp.eng.chula.ac.th/~piak/project/som/index.htm

[5] J. Bell, "Threaded code", Communications of the ACM, Volume 16 Issue 6, June 1973.

[6] M. Anton Ertl, Implementation of Stack-Based Languages on Register Machines, PhD thesis, Technische Universitat Wien, Austria, 1996.

[7] R. Loeliger, "Threaded Interpretive Languages", Byte book, 1981.

[8] C. Moore and G. Leach, Forth : A language for interactive computing, 1970.

[9] R. Wexelblat, History of programming languages, ACM Press, 1978.

[10] P. Chongstitvatana, "Self-Generating Systems: How a $10{,}000{,}000_2$-line Compiler Assembles Itself," Proc. of National Computer Science and Engineering Conference, Bangkok, 2005.

[11] http://www.cp.eng.chula.ac.th/~piak/project/tl/