The MT-Scheme Compiler

Kristine Joy S. Apon and Marco T. Morazán
Seton Hall University
Department of Mathematics and Computer Science
400 South Orange Avenue
South Orange, NJ 07079 USA
E-mail: {aponkris,morazanm}@shu.edu

ABSTRACT

The MT system is being developed to study different memory design and implementation choices for functional languages. Previous studies in MT have empirically demonstrated that the FIFO page replacement algorithm is better than the LRU page replacement algorithm for heap pages while LRU is superior to FIFO for stack pages. The current goal of the MT initiative is to study how memory that stores program code is accessed. A compiler for a subset of Scheme has been designed and implemented. The MT-Scheme compiler is a homomorphism between Scheme programs and sequences of MT primitive instructions. We present the architecture and implementation of the MT-Scheme compiler along with a brief description of the MT system.

1. INTRODUCTION

Functional programming languages such as LISP, Scheme, ML, Haskell, Erlang and Clean emphasize the evaluation of functions, rather than the execution of commands[6]. These programming languages have successfully liberated programmers from the von Neumann style of programming delivering ease of analysis and greater flexibility in program design by providing a higher level of abstraction than imperative programming languages such as C, Pascal, and Fortran[3]. They have done so by hiding the details of the machine programs are executed on. That is, programmers do not have to reason about the machine the program will be executed on. The existence of a fetch-execute cycle in the underlying architecture does not contaminate functional languages. For example, functional programmers do not have to reason about how to correctly sequence commands to meet the specification of the program they are writing. Instead, they reason about functions and function application much like mathematicians do. This has lead to a significant reduction in development time[5] and to the creation of more reliable code.

For these reasons the use of functional programming languages has become successful in the academic world. However, functional languages have failed to become part of the industrial mainstream because they are regarded as slow and inefficient. High memory consumption and poor interaction with virtual memory has contributed to the sluggish reputation of functional languages[4, 8, 12]. This reputation is beginning to be challenged empirically. In a study comparing Lisp, C/C++, and Java, Gat presents empirical evidence suggesting that Lisp is superior to Java and C/C++ in both speed and development time [5]. The same study also highlights that memory consumption in Lisp is comparable to that of Java and much higher than that used by C/C++. Thus, based on the numbers reported in the literature we can safely conclude that the implementation of functional languages has matured to the point that they are competitive with languages popular in the mainstream of computing, but there is still a need to do more research to understand and improve their interaction with memory systems.

In addition to improving memory performance, researchers have applied parallelism to functional languages in order to improve performance. These efforts can be classified into two genres. In the first, researchers have attempted to parallelize user code. This approach depends heavily on the type of code users write and results to date have not met the theoretical gains expected [7]. One of the major problems is that there does not exist a good set of heuristics to decide when to evaluate expressions in parallel. In the second genre, researchers have developed parallel garbage collection algorithms that run in tandem with program execution. It is difficult to judge the effectiveness of these algorithms, because many parallel garbage collection algorithms have not been implemented[1].

In order to understand and improve the performance of functional languages, the MT system is being developed. The goal is to apply parallelism to the engine that evaluates programs instead of trying to parallelize user code. The MT initiative proposes the creation of an intelligent backing store for functional languages based on the major components needed to evaluate functional programs. This intelligent backing store will be implemented using a distributed virtual memory system allowing for the management of different memory spaces to proceed in parallel with program evaluation. In this paper, we will first briefly describe the MT System and its architecture. Then we describe the implementation of the MT-Scheme compiler which can trans-

Proceedings of MASPLAS'03
Mid-Atlantic Student Workshop on Programming Languages and Systems
Haverford College, April 26, 2003

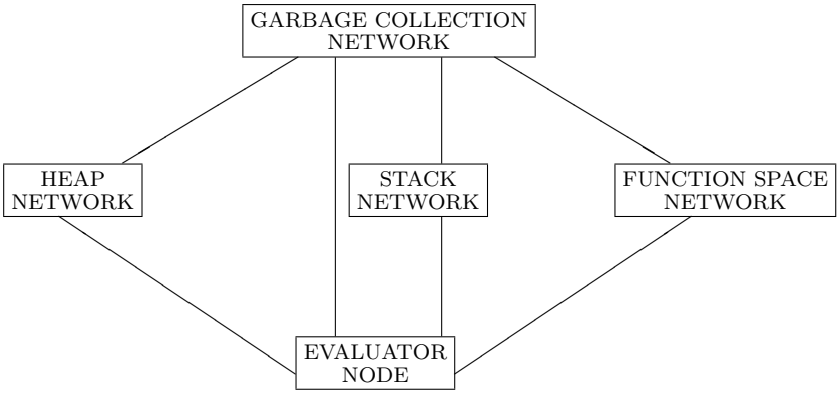

Figure 1: Abstract view of the architecture of an MT Node

form a pure subset of Scheme into sequences of MT primitive instructions. The paper ends with some conclusions and directions of future research.

2. THE MT SYSTEM

The MT system separates program evaluation from memory management into different processes running on different processors. The MT system architecture is based on an all-software distributed virtual memory system (DVM) tailored to the needs of functional languages. Memory is divided into separate spaces that allow memory management to occur in parallel with program execution. The MT system's fundamental computational unit is an MT node, which is made up of an evaluator (which is responsible for program execution) and four management networks: the MT heap, the MT stack, the MT function space, and the MT garbage collector. Figure 1 displays an abstract view of an MT node.

2.1 The MT Spaces

The MT Heap is used exclusively to store dynamically allocated lists. Heap memory is allocated using the MT allocation algorithm which only allocates new heap memory to execute the MT primitive **cons**[9]. Morazán et. al. have demonstrated that the MT allocation algorithm compactly stores list objects by reducing the intermingling of live data and garbage (i.e. heap memory that is no longer useful to the computation of the program[9]. They have also empirically suggested that the FIFO[1] page replacement algorithm performs as well as the LRU[2] page replacement policy.

The MT stack is primarily used for parameter passing and flow control. The stack holds the context in which functions are evaluated. When a function is to be applied to a set of arguments an activation record is pushed onto the MT stack.

[1]The *first-in first-out* page replacement policy always swaps out from main memory the page that has been memory resident for the longest period of time

[2]The *least recently used* page replacement policy always swaps out from main memory the page that has not been accessed for the longest period of time

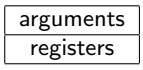

Figure 2: An MT activation record.

An activation record consists of the state of the machine (i.e. the values stored in certain registers) before the function is called and the arguments to the function[3]. Figure 2 displays the structure of an activation record in MT. After an activation record has been created on the stack, the function is called and applied to its arguments. Upon termination of the function, its activation record is popped off the MT stack, the state of the machine is restored, and the value computed by the function is pushed onto the stack. The paging algorithm used for the MT Stack is LRU. Morazán et. al. have mathematically demonstrated and empirically verified that LRU is optimal as a page replacement algorithm for the MT stack[10, 11]. Based on their mathematical proof, they designed and implemented an optimal page replacement algorithm that does incur the costly per access overhead associated with LRU[10, 11].

The MT function space stores the function table, the label table, and code space. The function table stores the name of all user-defined functions and the virtual address in code space of their first instruction. The label table stores all labels generated by a compiler and the virtual address in code space of the first instruction associated with each label. Labels and functions are kept in separate tables to avoid name collisions. Furthermore, only the function table is accessible from an interpreter for any language built on MT system. Code space stores all compiled instruction sequences. Each word in code space holds one MT primitive instruction. Currently, only code space is to be distributed while the function and label tables are to be held at the evaluator. Code space is broken into pages and a demand paging algorithm is implemented to fetch pages when a page

[3]For primitive functions the state of the machine is not saved as part of an activation record.

fault occurs. To date, no studies on the paging behavior of code space have been conducted. The MT-Scheme compiler described later in this article was partly motivated by the desire to study the paging behavior of code space.

The MT Garbage Collector is not yet implemented and is an area of active research. It is being designed to recycle the MT heap. It will need access to both MT heap and MT stack pages to perform its work. The collector will execute in parallel with program evaluation attempting to minimize synchronization time between the two processes. Current designs do not envision a stop-the-world collector, because of the amount of time the evaluator may have to be interrupted. The garbage collector itself may be a parallel process exploiting the intelligence available in the distributed virtual memory system.

2.2 The MT Evaluator Virtual Machine

The current version of the MT Evaluator Virtual Machine is implemented in C++. The MT Evaluator Virtual Machine includes a heap object, a stack object, a function space object, a symbol table object and a set of registers. The MT Evaluator Virtual Machine implements a fetch-execute cycle and has a set of primitive instructions that compute values and control the state of the machine. Figure 3 displays an abstract view of the MT evaluator virtual machine controller.

2.2.1 MT Registers

The MT Virtual Machine Evaluator has 7 registers:

- **PC**: The *program counter* register stores the virtual address of the next instruction to be executed.
- **CONT**: The *continuation* register stores the virtual address of the next instruction to be executed upon returning from a subroutine or function call.
- **ENV**: The *environment* register stores the virtual stack address of the activation record for the user-defined function currently being evaluated.
- **NEWENV**: The *new environment* register stores the virtual stack address of the activation record that is being created in preparation to call a user-defined function.
- **VAL**: The *value* register is used to store the result of applying any primitive or user-defined procedure to a set of arguments[4].
- **FLAG**: The *flag* register stores a boolean value that is used by branch instructions.
- **TEMP**: The *temporary* register is used as a temporary storage during the evaluation of user-defined functions.

[4]The value of functions that return a boolean value are stored in the flag register

2.2.2 S-expression

The basic unit of data in the MT Evaluator Virtual Machine is an S-expression. An S-expression is a structure that can either have two or three components depending on the type of data that is created. The first component, called the tag, indicates the type of data that the S-expression holds. It may be an integer, a real number, a symbol, a function, an operator, a boolean, a label, or a list. The tag may also indicate the register value that is stored on the stack. For all types of S-expressions except list, a single value is also stored as the second component. A list tag indicates a piece of compound data and two pointers into the heap (car and cdr) are stored.

3. THE MT-SCHEME COMPILER

The MT-Scheme compiler is written in Scheme and transforms code for a pure subset of Scheme into the equivalent sequence of MT machine instructions and outputs the sequence to a file. It is based on the compiler by Abelson and Sussman[2]. The current version of the MT-Scheme compiler is capable of compiling self-evaluating expressions, variables, quoted expressions, applications, if-expressions and definitions.

To compile a Scheme expression the procedure MTCompile must be invoked passing in four arguments:

(MTCompile **exp target linkage filename**), where

- **exp** is a list representing the Scheme expression to be compiled.
- **target** is a register that indicates where the value computed by the compiled code should be stored.
- **linkage** describes how the compiled code proceeds after execution. It can have one of the three values: next, return or a label. The symbol 'next signals the compiler that execution should continue at the next instruction in the sequence. The symbol 'return signals the compiler that the expression being compiled needs to return from a function call. A label is passed in to signal the compiler where to branch to after the code for the expression is executed.
- **filename** is a string naming a file where the compiled code is outputted.

A file containing compiled code for the MT evaluator virtual machine is composed of symbols, function names, label names and the machine instruction sequence. Symbols, function names and label names are located at the top of the compiled code and preceded by s_, f_, and l_ respectively. Before the compiled code is executed at run-time, the loader must load the symbols, the function names and the labels into their respective tables.

3.1 Expression Compilation

To exemplify how Scheme expressions are compiled we will use the recursive definition of $n!$. The Scheme code we will refer to is displayed in Figure 4.

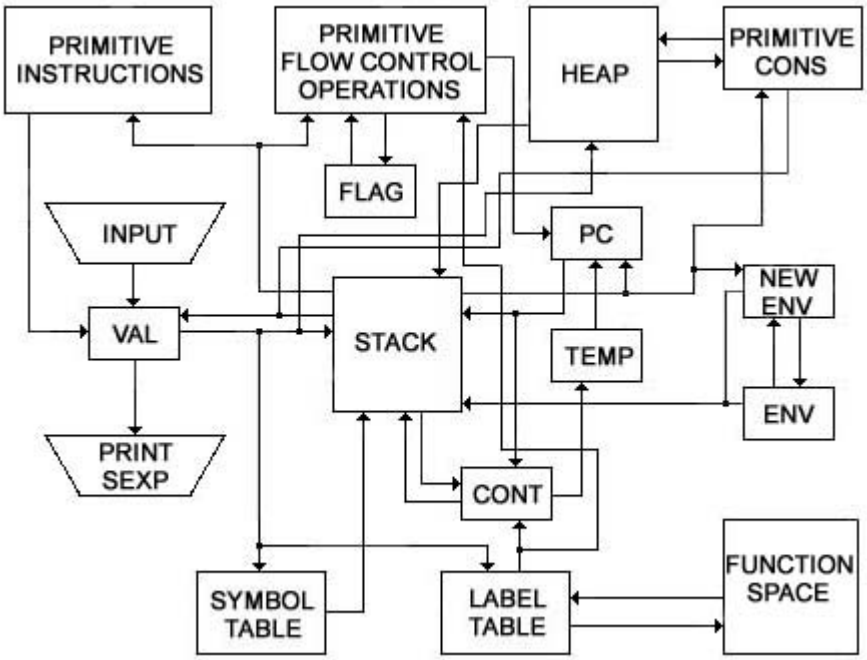

Figure 3: The Controller for the MT Evaluator Virtual Machine

```
(define (fact n)
  (if (≤ n 1)
    1
    (* n (fact (- n 1)))))
```

Figure 4: The recursive definition of $n!$ in Scheme.

3.1.1 Self-Evaluating Expressions

In the current version of the MT evaluator virtual machine the only self-evaluating expressions are numbers. Strings are not yet supported by the virtual machine. The MT primitives that create numbers are MKINTSEXP to create an integer and MKREALSEXP to create a real number. Each of these primitives takes as input an argument representing the desired number and pushes an S-expression it creates to represent the input number onto the MT stack. Figure 5 highlights how the integers in the definition of $fact$ are compiled.

3.1.2 Variable Expressions

Variables are symbols used as parameters in a definition. When a function is called arguments are passed in and the variables in the parameter list are said to be bound to the corresponding values of the expressions in the argument list. During execution variables are accessed by lexical addressing. In MT the lexical address of a variable is the position of the variable in the definition contour. For example, the lexical address of the first parameter is 1 while the second parameter is 2 and so on. The MT primitive that accesses variable values on the stack is PACC (parameter access). PACC accesses the value of a variable by adding the value in the environment register (i.e. the address of the current activation record), the number of saved registers, the lexical address of the variable minus one. Figure 6 displays how the variables in the definition of $fact$ are compiled.

```
f_fact 0
l_false-branch2 3
l_true-branch1 15
l_after-if3 17
PACC 1
MKINTSEXP 1
BRLTE l_true-branch1
PACC 1
FCALL
PACC 1
MKINTSEXP 1
MINUS
SAVEVAL
NENV2ENV
PCCONT 1
GOTO f_fact
MULT
SAVEVAL
FRETURN 1
MKINTSEXP 1
FRETURN 1
```

Figure 5: Self-evaluating expressions in $fact$ are compiled to the code inside the rectangles.

3.1.3 Quoted Expressions

Scheme Expressions that are preceded by a single quote may either be symbols or lists. 'hello is a symbol while '(1) is a list with just one element (i.e. 1). Each type of quoted expression is compiled differently as follows:

- **Symbols** are compiled by performing two actions. The first is to add new symbols to a symbol list where each symbol is prefixed with 's_'. The list of symbols will be placed on the top of the output file for the compiled

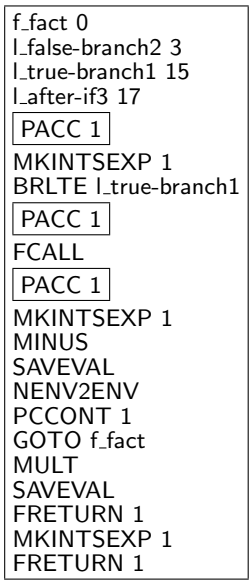

Figure 6: Variable expressions, n, in $fact$ are compiled by passing the lexical address of the variable to the MT primitive SACC.

code. The second action is to generate a MKSYMBOLSEXP primitive instruction in the proper place in the compiled code to create a symbol S-expression representing the symbol. This instruction takes as input the symbol prefixed by 's_'. At load time this prefixed symbol is replaced with the address of the symbol in the symbol table.

- **Lists** are compiled by first compiling each element and having them pushed on the stack. After generating the code to push all the elements onto the MT stack, the code for **cons**-ing them together is generated. At the end of each list (and sublist) the MKNILSEXP primitive is automatically generated since nil marks the end of a list. Figure 7 displays the compiled code for the list '((a 1) b), where a and b are symbols and (a 1) is a sublist.

3.1.4 Application Expressions

There are two kinds of functions that can be applied to a set of arguments: a primitive function and a user-defined function. The application of a primitive function, f_p, is compiled by first compiling the expressions for the arguments to f_p and then generating the MT primitive for f_p. For example, for the Scheme primitive $*$ in ($*$ 3 x) the code for 3 and x is generated first and then the MT primitive MULT is generated for the multiplication.

The application of a user-defined function, f_u, is compiled by generating the sequence of MT instructions to call f_u. The generated code to apply f_u must setup an activation record for f_u, set the continuation register to the point where execution should continue after f_u is executed, and then transfer control to f_u. The MT primitive that pushes the state part of an activation record onto the stack is FCALL. FCALL executes three MT primitives: STACK2NEWENV, SCONT,

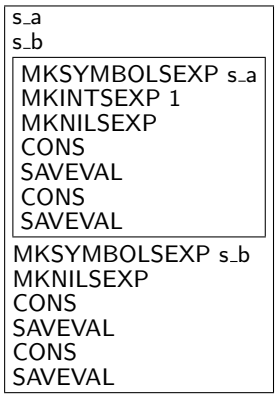

Figure 7: The list '((a 1) b) is compiled by first generating the code for the sublist '(a 1) (boxed) and then generating the code for 'b and for the consing of the elements.

and SENV. STACK2NEWENV saves the virtual address of the stacktop in the new environment register to mark the address of f_u's activation record. SCONT saves the value of the continue register on the MT stack which will be restored after control returns from the call to f_u. SENV saves the value of the environment register on the MT stack. The value of the environment register is restored after f_u is executed. To complete the activation record, the code to push the arguments of f_u onto the MT stack is generated. This code is then followed by the MT primitives, NENV2ENV and PCCONT. NENV2ENV is used to set the address of f_u's activation record by assigning the value of the new environment register to the environment register. PCCONT is used to set the continuation register to the sum of the value at the PC register, 1, and the integer constant it takes as input. This integer constant is the difference between the instruction following the instruction pointed to by the PC register and the point where execution should continue after the call to f_u. Figure 8 highlights the compiled code for - and the recursive call to *fact* for the function definition of Figure 4.

3.1.5 If Expressions

The general form of an if-expression is:

(if (predicate) (consequent) (alternative))

The code generated for the predicate must set the flag register which will be used to decide if the consequent or the alternative is executed. If the flag register is set to true the consequent will be executed. Otherwise, the compiled code for the alternative is executed.

For an if expression, the MT-Scheme compiler first generates the code for the predicate which will always end with a branching instruction to the beginning of the consequent code. The branching instruction will transfer control to the consequent code if the flag is set to true. Otherwise, execution continues with the first instruction after the branching instruction. The compiled code for the predicate is followed by generating a false-branch label to indicate the beginning of the alternative code. This label is followed by the MT code generated for the alternative. The alternative code al-

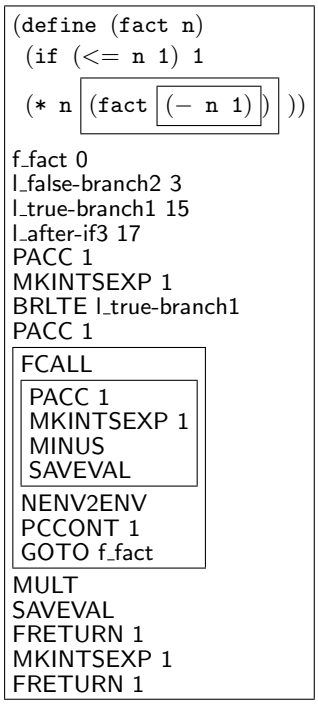

Figure 8: The compiled code for − is highlighted in a small box and the recursive call to $fact$ is highlighted in a large box.

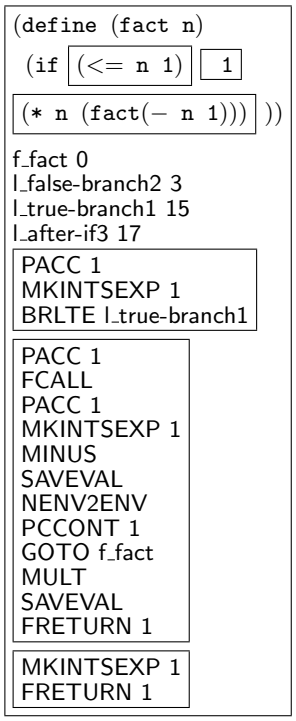

Figure 9: Compiled if statements always contain the predicate code, the alternative code, and the consequent code highlighted by the boxes.

ways end with a GOTO primitive, which transfers control to the point after the code generated for the if statement (marked by an after-if label), or an FRETURN primitive, which is used to return from a function call to a user-defined function. A GOTO or FRETURN primitive is necessary because if at run-time the alternative code is executed then the consequent code should not be executed. The alternative code is followed by a true-branch label marking the beginning of the consequent code. Following a true-branch label the code generated for the consequent is found. The code for the if statement ends with an after-if label. The structure of a compiled if statement can be visualized as follows:

predicate code
false-branch label
alternative code
true-branch label
consequent code
after-if label.

The code generated to compile the if statement in the definition in Figure 4 is highlighted in Figure 9.

Nested if statements are handled by branching to the appropriate label generated by the compiler. For example, if the predicate is itself an if-statement then the after-if label for the nested if statement (i.e. the predicate) will be followed a branch instruction to the true-branch of the outer if-statement. Similarly, nested if-statements contained in the consequent or alternative are followed by an appropriate branch instruction.

3.1.6 Function Definition Expressions

Function definition expressions have the form:

(define (< $functionname$ > < $parameters$ >) (< $body$ >))

The MT-Scheme compiler generates the MT instructions for a function definition, f_u, assuming that at runtime the machine will properly set-up an activation record for f_u before calling it. This means that the state of the machine and the arguments to f_u have been pushed onto the stack. For calls to f_u made from compiled functions, we know that this is the case as described in the section for the compilation of application expressions. If f_u is called from an interpreter then the interpreter must push f_u's activation record onto the stack before calling f_u.

For f_u the compiled code for its body is first generated as described in the above sections by doing a syntactic analysis on the type of expressions it contains. The generated code must end with code that pops its activation record off the stack, restores the state of the machine, and pushes the value computed by f_u onto the MT stack. These operations are captured in the FRETURN primitive instruction. FRETURN takes as input an integer that equals the number of parameters of f_u. The compiled code for the definition in Figure 4 is displayed in Figure 9. The last instruction in the code for both the consequent and the alternative for the if statement is FRETURN since the machine must return from the call to fact when it reaches that point in the code.

4. CONCLUDING REMARKS AND FUTURE WORK

The current version of the MT-Scheme compiler is capable of compiling self-evaluating expressions, variables, quoted expressions, applications, if-expressions and definitions. It generates code by doing a syntactic analysis of the expression being compiled generating a list of symbols, a list of defined functions, a list of compiler generated labels, and a sequence of MT primitive instructions. The current version of the MT-Scheme compiler is still not an industrial-strength compiler. For example, optimizations to evaluate tail-recursive functions using a constant amount of memory space are still not implemented. This optimization is currently being implemented.

Future work will also focus on the implementation of first-class functions in the MT system and on the compilation of related special forms like lambda and let in Scheme. In addition, we are looking at implementing lambda-lifting and deforestation and to empirically study their effect on the paging behavior of the different MT spaces.

5. REFERENCES

[1] S.E. Abdullahi, E.E. Miranda, and G.A. Ringwood. Collection Schemes for Distributed Garbage. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 43–81, Saint-Malo, France, 1992. Springer-Verlag.

[2] H. Abelson and J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, 1990.

[3] John Backus. Can Programming be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):165–180, 1978.

[4] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12:611–612, 1969.

[5] Erann Gat. Lisp as an alternative to Java. *Intelligence*, 11(4):21–24, 2000.

[6] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International, Englewood, NJ, USA, 1980.

[7] Santos Martins. Parallel Implementations of Functional Languages. In Herbert Kuchen and Rita Loogen, editors, *Fourth International Workshop on the Parallel Implementation of Functional Languages*, RWTH Aachen, Germany, 1992. Aachener Informatik-Berichte.

[8] David A. Moon. Garbage Collection in a Large Lisp System. *Proc. of the 1984 ACM Symp. on Lisp and Functional Programming*, pages 235–246, 1984.

[9] Marco T. Morazán and Douglas R. Troeger. The MT Architecture and Allocation Algorithm. In Phil Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, volume 1, pages 97–104, Bristol, UK, 2000. Intellect.

[10] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Designing an All-Software Based Distributed Virtual Memory: The Paging Performance of The MT Stack. In Silvia Teresita Acuna and Cecilia Maria Laserre, editors, *Proc. of The* 1^{st} *Iberoamerican Conference on Software Engineering and Knowledge Engineering*, pages 109–120, Buenos Aires, Argentina, 2001. Universidad Nacional de Jujuy (Editorial UNJU).

[11] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. The MT Stack: Paging Algorithm and Performance in a Distributed Virtual Memory System. *CLEI Electronic Journal*, 5(1), 2002.

[12] Robert A. Shaw. *Empirical Analysis of a Lisp System*. PhD thesis, Department of Computer Science, Computer Systems Laboratory, Stanford University, Stanford, California, 1988.