A Mechanically Verified Code Generator

William D. Young

Technical Report 37 January, 1989

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was supported in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Chapter 1
INTRODUCTION

A compiler provides the ability to program/specify/design in a notation which is more elegant, abstract, or expressive than any which is native to a given piece of hardware. The concomitant gains are real only if the compilation process provides a *correct* translation from the source language to the target language—that is, one which generates semantically equivalent target language code for any given source language program. An incorrect compiler frustrates the effort a programmer invests in writing, debugging, and even verifying his source language program.

The compiler is one of a number of modules in a system of software and hardware support upon which the reliability of program execution depends. Others include the assembler, linker, loader, and runtime software support. Each of these can be designed with fair reliability following the best current software engineering practices. It is our contention that a higher degree of system reliability can be attained by modeling system software within mathematical logic and formally proving its correctness. The most highly reliable software systems will be those in which all of the software components have been proved correct. If additionally we can apply formal verification techniques to user-level software, we gain the ability to build applications on top of our verified support system with a much higher level of assurance than is available from current software engineering techniques.

In this paper we describe the implementation and proof of a code generator, a major component of a compiler. The source language is a subset of Gypsy (version 2.05) [10] and the target language is the Piton [21] assembly level language. Our code generator is one level of a *stack* of verified system components including an assembler and linking loader for Piton and a microprocessor design verified at the register transfer level [13]. Parallel research addresses the issue of verified operating system functionality [2]. The integration of these components into a *vertically verified system* is addressed in a companion paper [3].

Because our source language is a subset of Gypsy, we have the option of verifying the correctness of user-level programs in the Gypsy Verification Environment [11]. Verified programs are compiled into Piton using the code generator; the resulting programs are then assembled into a load image for the FM8502 microprocessor. Thus the semantics of the verified high-level program is provably preserved through several translation steps to an implementation at the hardware level.

The official arbiter of our claim that we have provided a rigorous formal proof of a code generator for a significant subset of Gypsy is a list of "events" in the computational logic of Boyer and Moore [4, 5]. That list is sufficient to lead the Boyer-Moore theorem prover enhanced with an interactive interface by Matt Kaufmann [14] to the proof of our main theorem. This paper is a summary of a much longer report [26] which contains that list and in which we

- present a language recognizer and operational semantics for a subset of Gypsy which we call Micro-Gypsy,
- describe the operational semantics for a subset of the Piton assembly level language,
- implement as functions in the Boyer-Moore logic a code generator translating Micro-Gypsy programs and data into Piton,
- state formally the correctness of the translator, and
- describe a mechanically checked proof of the correctness of the translator.

In the current paper we briefly discuss each of these issues. Our major goal is to give the reader enough information to understand and access our claim that we have verified a code generator for a subset of Gypsy.

In the following chapter we describe the Micro-Gypsy programming language and the abstract prefix syntax which is the input of our verified code generator. In subsequent chapters we sketch the Piton assembly language which is the target language of our code generator, state and explain the correctness theorem for the Micro-Gypsy code generator, and explain the implementation and proof of the correctness result. A significant and necessary byproduct of our proof is a formal semantics for our Micro-Gypsy source language. We illustrate how this semantics can be used to prove the correctness of Micro-Gypsy programs and contrast this to proofs of these programs using the Gypsy Verification Environment.

Chapter 2
MICRO-GYPSY

The source language for our code generator is an abstract syntax form of Micro-Gypsy, a subset of Gypsy. Gypsy is a combined programming and specification language descended from Pascal. It includes dynamic types such as sequences and sets, permits procedural and data abstraction, and supports concurrency. The specification component of Gypsy contains the full first order predicate calculus and the ability to define recursive functions. Programs can be specified with Hoare style annotations, algebraically, or axiomatically. Proof rules exist for each component of the language. The Gypsy Verification Environment (GVE) is a collection of software tools which allow Gypsy programs to be developed, specified, and proved.

Micro-Gypsy contains a significant portion of the executable component of Gypsy including the simple data types and arrays, most of the Gypsy flow of control operations, and predefined and user-defined procedures including recursive procedures. It does not include Gypsy's dynamic data types, data abstraction, or concurrency. Figure 2-1 summarizes features of the language. Our intent in defining the Micro-Gypsy subset was a language sufficient for coding simple applications of the variety for which Gypsy has been used yet of a manageable size to permit constructing a mechanically verified code generator. We believe that the subset is adequate to validate our overall approach to proving the correctness of a code generator but needs to be extended in a variety of ways to make it a useful programming tool. Future research will be directed toward extending Micro-Gypsy.

Gypsy programs may be annotated using the specification component of the language and verified in the GVE. The Micro-Gypsy Lisp-like syntax which is the input to the code generator is not acceptable to the GVE. However, this abstract syntax could easily be generated from Gypsy-style syntax by a preprocessor.[1] Thus, Micro-Gypsy programs can be annotated and verified using all of the mechanical

[1]Such a preprocessor was written for an earlier version of Micro-Gypsy by Ann Siebert. There is no such preprocessor for the current version of Micro-Gypsy, however; the abstract syntax is generated by hand from Gypsy-style text.

Types	`BOOLEAN, INTEGER, CHARACTER, one-dimensional ARRAY`
Control Abstraction	`IF, LOOP, BEGIN-WHEN, SIGNAL`
Procedural Abstraction	`user-defined PROCEDURE, predefined PROCEDURE`

Figure 2-1: Features of Micro-Gypsy

tools available for verifying programs in Gypsy. Verified programs are then preprocessed into a form acceptable to our verified code generator and translated into semantically equivalent Piton programs. An alternative approach to proving the correctness of Micro-Gypsy programs is to verify them directly in the Boyer-Moore logic using the semantics defined by the Micro-Gypsy interpreter. Both approaches are discussed and illustrated in chapter 7.

Micro-Gypsy is characterized by a recognizer and an interpreter. The recognizer is a predicate which ensures that the language satisfies a minimal set of syntactic constraints required for our proof. The interpreter provides an operational semantics for the language. The formal definition of Micro-Gypsy is embodied in a collection of function definitions written in the Boyer-Moore logic defining the recognizer and interpreter.

The abstract syntax of Micro-Gypsy is a simple and expressive, but inelegant program description language. Much of the inelegance arises from the fact the abstract syntax allows only variables and simple literals as expressions. Complex expressions are translated in preprocessing into a sequence of calls to predefined procedures. The syntax is fully described in [26]. Figure 2-2 displays an annotated Micro-Gypsy program for computing the product of two numbers. The translation of this into the Micro-Gypsy abstract syntax form yields the two procedures shown in figure 2-3.

The semantics of Micro-Gypsy programs is defined with respect to an *execution environment* consisting of:

- a sequence of procedures,
- an entry point (a Micro-Gypsy statement), and
- a *Micro-Gypsy state* or *mg-state*.

An mg-state bundles together the *dynamic* components of the execution environment which can be

```
scope MULTIPLICATION_ROUTINES =
begin

   const MININT := -2147483648;

   const MAXINT := 2147483647;

   type INT = integer [MININT .. MAXINT];

   procedure MULTIPLY (var ANS: INT;
                       I, J: INT) =
   begin
      exit ANS = I * J;
      if J ge 0
         then MULTIPLY_BY_POSITIVE (ANS, I, J)
         else MULTIPLY_BY_POSITIVE (ANS, I, -J);
              ANS := - ANS
      end; {if}
   end; {multiply}

   procedure MULTIPLY_BY_POSITIVE (var ANS: INT;
                                   I, J: INT) =
   begin
      entry J ge 0;
      exit  ANS = I * J;
      var K: INT := 0;
      K := J;
      ANS := 0;
      loop
         assert J ge 0
                & K in [0 .. j]
                & ANS = (J-K) * I;
         if K le 0 then leave end;
         ANS := ANS + I;
         K := K - 1;
      end;
   end; {multiply_by_positive}

end; {scope multiplication_routines}
```

Figure 2-2: A Micro-Gypsy Multiplication Routine

affected by program execution. It contains a list of variable bindings, a single condition variable containing the *current condition*, and a program status word or psw. The psw is normally **RUN** but may also be **RESOURCE-ERROR** or **TIMED-OUT**, indicating some aberrant condition which cannot be handled by the program. A sample mg-state is

```
(MG-STATE 'NORMAL                                  ; current condition
          '((B BOOLEAN-MG (BOOLEAN-MG TRUE-MG))     ; variable alist
            (I INT-MG (INT-MG -24))
            (J INT-MG (INT-MG 20))
            (K INT-MG (INT-MG 0))
            (A (ARRAY-MG CHARACTER-MG 3) ((CHARACTER-MG 78)
                                          (CHARACTER-MG 73)
                                          (CHARACTER-MG 76))))
          'RUN)                                    ; psw
```

With respect to this mg-state, a potential entry point for our multiplication routine is the Micro-Gypsy statement **(PROC-CALL-MG MG_MULTIPLY (K I J) NIL)**.

```
(MG_MULTIPLY
  ((ANS  INT-MG)
    (I    INT-MG)
    (J    INT-MG))
   NIL
   ((K    INT-MG    (INT-MG 0))
    (ZERO INT-MG    (INT-MG 0))
    (B  BOOLEAN-MG  (BOOLEAN-MG FALSE-MG)))
   NIL
   (PROG2-MG
    (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B ZERO J))
    (IF-MG B
      (PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS I J) NIL)
      (PROG2-MG
       (PREDEFINED-PROC-CALL-MG MG-INTEGER-UNARY-MINUS (K J))
       (PROG2-MG
        (PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS I K) NIL)
        (PREDEFINED-PROC-CALL-MG
         MG-INTEGER-UNARY-MINUS (ANS ANS)))))))

(MG_MULTIPLY_BY_POSITIVE
  ((ANS INT-MG)
   (I   INT-MG)
   (J   INT-MG))
  NIL
  ((K    INT-MG     (INT-MG 0))
   (ZERO INT-MG     (INT-MG 0))
   (ONE  INT-MG     (INT-MG 1))
   (B    BOOLEAN-MG (BOOLEAN-MG FALSE-MG)))
  NIL
  (PROG2-MG
   (PREDEFINED-PROC-CALL-MG
           MG-SIMPLE-CONSTANT-ASSIGNMENT (ANS (INT-MG 0)))
   (PROG2-MG
    (PREDEFINED-PROC-CALL-MG
             MG-SIMPLE-VARIABLE-ASSIGNMENT (K J))
    (LOOP-MG
     (PROG2-MG
      (PREDEFINED-PROC-CALL-MG MG-INTEGER-LE (B K ZERO))
      (PROG2-MG
       (IF-MG B (SIGNAL-MG LEAVE) (NO-OP-MG))
        (PROG2-MG
         (PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (ANS ANS I))
         (PREDEFINED-PROC-CALL-MG
               MG-INTEGER-SUBTRACT (K K ONE))))))))
```

Figure 2-3: The Multiplication Routine in Abstract Prefix

The semantics of Micro-Gypsy statements is given via an interpreter expressed as a recursive function in the Boyer-Moore logic. This function, called **MG-MEANING**, takes four arguments—the three components of the execution environment and a *clock* argument **N**. **N** bears a rather complicated and unintuitive relation to the number of "steps" executed; it is primarily an artiface to assure that all computations terminate.[2] Figure 2-4 summarizes the semantics of the eight statement types and one of the

[2]Because Micro-Gypsy programs may be non-terminating, the most natural Micro-Gypsy interpreter would not be a total function. The version of the Boyer-Moore logic in which **MG-MEANING** was constructed requires that all functions be total. "Ticking down" the clock argument on each recursive call in the interpreter forces termination.

predefined procedures of the Micro-Gypsy abstract syntax. Much of the expressive power of the language resides in the collection of predefined procedures. The prototype code generator handles only 13 predefined's, though adding additional ones is straightforward.

Micro-Gypsy Statement Types:

`(NO-OP-MG)`
: No effect on the state.

`(SIGNAL-MG condition)`
: Set the current-condition to **`condition`**.

`(PROG2-MG left right)`
: Execute **`right`** in the state resulting from executing **`left`**.

`(LOOP-MG body)`
: Execute **`body`** and then execute **`(LOOP-MG body)`** in the resulting state. The loop is exited if the condition **`LEAVE`**, is ever signaled.

`(IF-MG test left right)`
: If **`test`** is true in the current state, execute **`left`**; otherwise execute **`right`**.

`(BEGIN-MG body cond-list handler)`
: Execute **`body`**; if any of the conditions in **`cond-list`** is signalled, reset the current-condition and execute **`handler`**.

`(PROC-CALL-MG name actuals conds)`
: Execute the body of procedure **`name`** in the state created by bindings actuals for formals. Then copy out the var parameters into the calling environment.

`(PREDEFINED-PROC-CALL-MG name actuals)`
: Apply the semantics of the predefined operation **`name`**.

Example of Predefined:

`(PREDEFINED-PROC-CALL-MG MG-INTEGER-ADD (X Y Z))`
: if **`(Y + Z)`** is a representable integer, then **`X := Y + Z`**; otherwise, set current condition to **`ROUTINEERROR`**.

Figure 2-4: Semantics of Micro-Gypsy Statement Types

There are actually *two* interpreter functions defined for Micro-Gypsy. **`MG-MEANING`** provides a semantics of Micro-Gypsy which is largely independent of its implementation in Piton. However, because a Piton abstract machine has explicit resource limitations, it is trivial to write Micro-Gypsy programs whose translations will not execute correctly because they exhaust the available resources in the implementation. The interpreter function **`MG-MEANING-R`** is structurally identical to **`MG-MEANING`** except that it "reflects" the resource limitations of the Piton implementation up into the Micro-Gypsy world. **MG-MEANING-R** terminates with psw set equal to **RESOURCE-ERROR** if the resources of the underlying Piton machine are inadequate for the computation. Our proof of correctness of the implementation is predicated on the assumption that no resource error occurs. A key lemma which we have proved about our interpreters is that in the absence of resource errors **MG-MEANING** and **MG-MEANING-R** return identical results.

Given the sample Micro-Gypsy state above, the Micro-Gypsy procedure list consisting of the two procedures in figure 2-3, and a sufficiently large clock value, the meaning assigned by our interpreter to the entry point statement **(PROC-CALL-MG MG_MULTIPLY (K I J) NIL)** is the following Micro-Gypsy state.

```
(MG-STATE 'NORMAL
          '((B BOOLEAN-MG (BOOLEAN-MG TRUE-MG))
            (I INT-MG (INT-MG -24))
            (J INT-MG (INT-MG 20))
            (K INT-MG (INT-MG -480))
            (A (ARRAY-MG CHARACTER-MG 3) ((CHARACTER-MG 78)
                                          (CHARACTER-MG 73)
                                          (CHARACTER-MG 76))))
          'RUN)
```

Notice that this is identical to our initial state except that the value of the variable **K** has been set equal to the product of the values of variables **I** and **J**, as expected. It is possible to specify formally what it means for such a program in our abstract syntax to be correct and to prove this result. We discuss this further in chapter 7.

Chapter 3
PITON

Piton is a high-level assembly language designed for verified applications and as the target language for high-level language compilers. It provides execute-only programs, recursive subroutine call and return, stack based parameter passing, local variables, global variables and arrays, a user-visible stack for intermediate computations, and seven abstract data types including integers, data addresses, program addresses and subroutine names. The feature which perhaps most clearly distinguishes Piton from conventional assembly languages is the distinct program and data spaces; it is impossible for a Piton program to overwrite itself.

The Micro-Gypsy code generator uses only a portion of the 65 available Piton instructions. Some data types—bit vectors and program addresses, for example—are not used at all. Figure 3-1 lists the Piton instructions used by the code generator.

The semantics of Piton is defined with respect to a nine-component Piton state or *p-state*. A p-state contains:

- a *program segment*, defining a system of Piton programs or subroutines;
- a *data segment*, defining a collection of disjoint named indexed data spaces (i.e., global arrays);
- a *temporary stack*;
- three control fields, consisting of
    - a *control stack*, consisting of a stack of *frames*, the top-most frame describing the currently active subroutine invocation and the successive frames describing the hierarchy of suspended invocations;
    - a *program counter*, indicating which instruction in which subroutine is the next to be executed;
    - a *program status word* (*psw*); and
- three resource limitation fields,
    - a *word size*, which governs the size of numeric constants and bit vectors,
    - a *maximum control stack size*, and
    - a *maximum temporary stack size*.

Control	Integers	Natural Numbers
CALL	**ADD-INT**	**ADD-NAT**
JUMP	**ADD1-INT**	**ADD-NAT-WITH-CARRY**
JUMP-CASE	**EQ**	**ADD1-NAT**
NO-OP	**INT-TO-NAT**	**EQ**
RET	**LT-INT**	**LT-NAT**
TEST-BOOL-AND-JUMP	**NEG-INT**	**SUB-NAT**
TEST-INT-AND-JUMP	**SUB-INT**	**SUB-NAT-WITH-CARRY**
TEST-NAT-AND-JUMP	**SUB-INT-WITH-CARRY**	**SUB1-NAT**
	SUB1-INT	

Variables	Booleans	Data Addresses
POP-GLOBAL	**AND-BOOL**	**DEPOSIT**
POP-LOCAL	**EQ**	**FETCH**
PUSH-GLOBAL	**NOT-BOOL**	**SUB-ADDR**
PUSH-LOCAL	**OR-BOOL**	

Stack
DEPOSIT-TEMP-STK
FETCH-TEMP-STK
POP
POP*
POPN
PUSH-CONSTANT
PUSH-TEMP-STK-INDEX

Figure 3-1: Piton Instructions Used by Code Generator

The semantics is given via an interpreter function **P** in the Boyer-Moore logic which takes as arguments a p-state and natural number **N**. The value of **P** applied to its arguments is the p-state obtained by beginning in our input p-state and "stepping" forward **N** Piton instructions (or until the psw is no longer **RUN**). At each step, a Piton instruction is fetched from the program segment according to the current value of the program counter and executed in the current state.

The Piton instruction set, its semantics, its implementation on the FM8502 microprocessor, and the proof of the correctness of this implementation is fully described in [22] and summarized in [21].

Chapter 4
THE CORRECTNESS OF THE MICRO-GYPSY CODE GENERATOR

The translator from Micro-Gypsy to Piton takes a Micro-Gypsy execution environment (including the program) and creates a Piton state. This is implemented as a function **`MAP-DOWN`** in the Boyer-Moore logic.

The correctness of the translation is stated as a formalization of the following commutative diagram.

$$\begin{tikzcd}
\text{MG-Initial} \arrow[r, "\textit{MG-Meaning}"] \arrow[d, "\textit{Map-Down}"] & \text{MG-Final} \\
\text{P-Initial} \arrow[r, "\textit{k Piton steps}"'] & \text{P-Final} \arrow[u, "\textit{Map-Up}"']
\end{tikzcd}$$

The intuition is simple. Given a Micro-Gypsy program (execution environment), we wish to discover the result of executing that program. We can invoke our Micro-Gypsy interpreter **`MG-MEANING`** directly. However, if our translation to Piton is correct, we can also translate our initial Micro-Gypsy state to a equivalent p-state, execute this p-state with the Piton interpreter, and extract from the final Piton state the same information we would get from our final mg-state. The general strategy for defining such *interpreter equivalence* theorems and some potential pitfalls are discussed in the companion paper [3].

The actual correctness result for the Micro-Gypsy translator is the following theorem in the Boyer-Moore logic:

Theorem. TRANSLATION-IS-CORRECT

```
(IMPLIES
 (AND (OK-EXEC-ENVIRONMENT STMT COND-LIST PROC-LIST MG-STATE SUBR N)
      (NOT (RESOURCE-ERRORP
            (MG-MEANING-R STMT PROC-LIST MG-STATE N
                  (LIST (DATA-LENGTH (MG-ALIST MG-STATE))
                        (PLUS 2 (LENGTH (MG-ALIST MG-STATE)))))))))
 (EQUAL (MAP-UP (P (MAP-DOWN MG-STATE PROC-LIST COND-LIST SUBR STMT)
                   (CLOCK STMT PROC-LIST MG-STATE N))
                (SIGNATURE (MG-ALIST MG-STATE))
                COND-LIST)
        (MG-MEANING STMT PROC-LIST MG-STATE N)))).
```

This can be interpreted as follows: if the initial Micro-Gypsy execution is well-formed and if we have provided adequate resources for execution of our Micro-Gypsy program, then the diagram "commutes". That is, it is possible to obtain the final Micro-Gypsy state either by running the Micro-Gypsy interpreter **MG-MEANING** directly or via Piton as follows.

- *Down*. Create an initial Piton state with the translator function **MAP-DOWN**.
- *Across*. Obtain the final p-state by running the Piton interpreter for **k** steps, where **k** is a number obtained from the Micro-Gypsy execution environment and clock by the constructive function **CLOCK**.
- *Up*. Extract from the final Piton state using the constructive function **MAP-UP** values corresponding to the values of variables in the Micro-Gypsy variable association list and the Piton representation of the Micro-Gypsy current condition.

The hypotheses of our correctness theorem place several requirements on our initial Micro-Gypsy state and on its execution including the following.

1. The entry point **STMT** is a Micro-Gypsy statement legal in the current execution environment.
2. **PROC-LIST** is a legal list of Micro-Gypsy user-defined procedures.
3. The current Micro-Gypsy state **MG-STATE** has appropriate structure and its various components have legal values.
4. **COND-LIST** is a list of legal Micro-Gypsy condition names and is not more than a fixed maximum length.
5. **SUBR** is a legal procedure name not used in the user program.[3]
6. Execution of our Micro-Gypsy program does not exhaust either time or space resources; i.e., the final Micro-Gypsy psw is **RUN**.

Several aspects of the correctness theorem are rather subtle. First, our **MAP-UP** function extracts only that part of the final Piton state corresponding to the Micro-Gypsy variable values and current condition. These are, in fact, the only *dynamic* aspects of the Micro-Gypsy execution environment and hence the only ones which could have been changed by execution.[4]

[3]**MAP-DOWN** constructs a Micro-Gypsy procedure whose body is just the entry point statement. **SUBR** becomes the name of this new procedure and is required to be distinct from any predefined or user-defined procedure name.

[4]The psw can also change but our hypotheses assume that the final psw is **RUN**.

Also, notice that the call to **`MAP-UP`** takes as one argument the expression **`(SIGNATURE (MG-ALIST MG-STATE))`**. This is a list associating the Micro-Gypsy variable names with their types. Why should the **`MAP-UP`** function need information about the Micro-Gypsy level? The map *down* from Micro-Gypsy data to Piton data is not injective. Hence, there is not enough information in the Piton state to recover the final Micro-Gypsy variable alist without some additional type information. This is typical of compilers; all high-level data is ultimately represented as bit strings which may have a variety of interpretations. It is necessary to supply an interpretation to be able to extract the data.

Finally, we are concerned only with computations in which there are no resource errors, i.e., in which the final psw is **`RUN`**. But, in general the user of Micro-Gypsy cannot determine whether an error occurs without running the interpreter **`MG-MEANING-R`**; this seems like an unreasonable restriction to place on the useability of a code generator. The answer is that we can apply the Micro-Gypsy code generator to programs which have been *proved* to execute without run-time errors. Consider the multiplication program **`MG_MULTIPLY_BY_POSITIVE`** discussed above. We can show formally that, for arbitrary inputs, execution of the program will not cause resource errors if we allocate amounts of storage and time above a certain threshold parameterized by the input values. This is part of a proof of the correctness of the **`MG_MULTIPLY_BY_POSITIVE`** program. This result, along with the general correctness result for the code generator, allows us to conclude that we can use our Piton translation of **`MG_MULTIPLY_BY_POSITIVE`** to perform multiplication by a non-negative integer and that the result extracted from the final Piton state will be the correct product of the input multiplicands. The correctness theorem for **`MG_MULTIPLY_BY_POSITIVE`** is discussed further in chapter 7.

Chapter 5
THE IMPLEMENTATION

In this section we discuss the implementation of Micro-Gypsy in Piton. This mainly involves explaining the function **`MAP-DOWN`**. **`MAP-DOWN`** has the task not only of translating Micro-Gypsy code into Piton, but also of translating data structures, establishing the initial program counter, temporary and control stacks, and establishing the resource limitations of the new Piton state.

We construct our p-state with an awareness that Piton is ultimately implemented on the FM8502 verified microprocessor [13]. We fix the word size at 32, for example, in Micro-Gypsy and in the p-states we generate because that is the word size of the verified Piton implementation. Such considerations are needed to assure that our implementation of Micro-Gypsy on Piton will *stack* atop the implementation of Piton on the FM8502.

We now describe how the nine fields of the Piton state are constructed from the Micro-Gypsy initial execution environment.

5.1 Data Storage

All data items in the Piton implementation of a Micro-Gypsy program are accessed by *reference*. For each data structure to which it has access, a program maintains a local pointer to the beginning of a block of storage on the Piton temporary stack. This permits a very uniform treatment of data; each data parameter to a procedure expects a single pointer value, even if the Micro-Gypsy formal parameter is of array type.

An interesting feature of our project is that the formal semantics of Micro-Gypsy uses *call by value-result* parameter passing and the implementation uses *call by reference*. To our knowledge, there has been no previous compiler proof in which the formal semantics and implementation used different parameter passing mechanisms.

To implement call by reference **MAP-DOWN** represents the Micro-Gypsy data structures as values on the initial temporary stack and stores pointers to these structures in the bindings component of the topmost frame on the Piton control stack.

5.1.1 The Temporary Stack

Suppose that our Micro-Gypsy execution environment contains the following variable bindings.

```
((B1 BOOLEAN-MG (BOOLEAN-MG FALSE-MG))
 (CH CHARACTER-MG (CHARACTER-MG 25))
 (A (ARRAY-MG INT-MG 5) ((INT-MG -294)
                         (INT-MG 38)
                         (INT-MG 0)
                         (INT-MG 12)
                         (INT-MG 25)))
 (B2 BOOLEAN-MG (BOOLEAN-MG TRUE-MG))
 (I INT-MG (INT-MG 4202))).
```

The initial value of the temporary stack in our Piton execution environment would be:

```
index          contents           represents

  9                               unused above here
  8            (INT 4202)             I
  7            (BOOL T)               B2
  6            (INT 25)               A[4]
  5            (INT 12)               A[3]
  4            (INT 0)                A[2]
  3            (INT 38)               A[1]
  2            (INT -294)             A[0]
  1            (INT 25)               CH
  0            (BOOL F)               B1
```

Piton contains instructions for storing and retrieving elements at arbitrary positions in the temporary stack. Thus the temporary stack serves as a random access "memory" in the implementation. Notice that the values of both character variable **CH** and the integer array element **A[4]** are represented by the same Piton **INT** value at the Piton level. This illustrates the loss of information in mapping down; this requires that the **MAP-UP** function have information about the types of Micro-Gypsy variables as mentioned in Section 4 above.

5.1.2 The Control Stack

The Piton control stack is a list of *frames* containing the local variable bindings and return program counter (pc) for each procedure invocation in the current call tree. Piton execution terminates when the last frame is popped, i.e., when control returns from the main program. Because of this the final return pc is irrelevant. **MAP-DOWN** creates a control stack containing a single frame with a dummy pc and list of bindings containing pointers to all of the data in the temporary stack.

For the example above, the bindings component created is

```
((B1 . 0) (CH . 1) (A  . 2) (B2 . 7) (I  . 8)).
```

Thus, the program can access elements of array **A** at an appropriate offset from the value of local variable **A** (in this case the natural number 2) in the temporary stack.

Program execution uses the top of the temporary stack for temporary storage, computation, and as storage for local variables at procedure call time.

5.1.3 The Data Segment

The Piton data segment is a list of global arrays. Two uses are made of the data segment. During program execution, the Piton representation of the Micro-Gypsy current condition is maintained in the global variable (data segment element) **C-C**. The current condition in Micro-Gypsy is a literal atom which may be **NORMAL**, **LEAVE**, **ROUTINEERROR**, or any element of an initial list of conditions **COND-LIST**. This is translated into a Piton natural number (essentially using the index of the condition in **COND-LIST**) and stored in the Piton Data Segment element **C-C**.

Another use is made of the data segment. At the end of program execution, the final values of the Piton analogues of the Micro-Gypsy data structures are copied from the temporary stack into the data segment. This is necessary because the proof of the implementation of Piton on FM8502 guarantees only the value of the final data segment. Therefore, to make the Micro-Gypsy proof "stack" on top of the Piton proof, it is necessary that the final answers be in the data segment rather than on the temporary stack. **MAP-DOWN** creates dummy locations with zero values for all of the Micro-Gypsy data structures to accommodate this final move.

An initial data segment corresponding to the variable bindings above would be

```
((C-C (NAT 2)) (B1 (NAT 0)) (CH (NAT 0))
 (A (NAT 0) (NAT 0) (NAT 0) (NAT 0) (NAT 0))
 (B2 (NAT 0)) (I (NAT 0)))
```

At the end of execution, this would contain representation of the final values of each of the Micro-Gypsy data structures and the final value of the current condition.

5.2 Code Generation

The translation of Micro-Gypsy statements into Piton is very naive; no optimization is attempted.[5] Figure 5-1 displays the Piton procedure which results from translating the

[5]We are currently investigating verifying an optimizer for a subset of Piton. Optimizing full Piton would be difficult because it is possible in Piton to dynamically create program addresses and jump to them. Since no use is made of this feature by the Micro-Gypsy code generator, we could insert a verified optimizer into our stack between the Micro-Gypsy and Piton implementation levels.

`MG_MULTIPLY_BY_POSITIVE` routine in figure 2-3. The reader interested in the full details of translation should consult [26].

```
(MG_MULTIPLY_BY_POSITIVE
     (K ZERO ONE B ANS I J)                              ; formals
     NIL                                                 ; locals
     (PUSH-LOCAL ANS)                                    ; ans := 0;
     (PUSH-CONSTANT (INT 0))
     (CALL MG-SIMPLE-CONSTANT-ASSIGNMENT)
     (PUSH-LOCAL K)                                      ; k := j;
     (PUSH-LOCAL J)
     (CALL MG-SIMPLE-VARIABLE-ASSIGNMENT)
     (DL 1 NIL (NO-OP))                                  ; loop
     (PUSH-LOCAL B)                                      ;    b := k le 0
     (PUSH-LOCAL K)
     (PUSH-LOCAL ZERO)
     (CALL MG-INTEGER-LE)
     (PUSH-LOCAL B)                                      ;    if b then leave
     (FETCH-TEMP-STK)
     (TEST-BOOL-AND-JUMP FALSE 3)
     (PUSH-CONSTANT (NAT 0))
     (POP-GLOBAL C-C)
     (JUMP 2)
     (JUMP 4)
     (DL 3 NIL (NO-OP))
     (DL 4 NIL (NO-OP))
     (PUSH-LOCAL ANS)                                    ;       ans := ans + i;
     (PUSH-LOCAL ANS)
     (PUSH-LOCAL I)
     (CALL MG-INTEGER-ADD)
     (PUSH-GLOBAL C-C)
     (SUB1-NAT)
     (TEST-NAT-AND-JUMP ZERO 0)
     (PUSH-LOCAL K)                                      ;       k := k - 1;
     (PUSH-LOCAL K)
     (PUSH-LOCAL ONE)
     (CALL MG-INTEGER-SUBTRACT)
     (PUSH-GLOBAL C-C)
     (SUB1-NAT)
     (TEST-NAT-AND-JUMP ZERO 0)
     (JUMP 1)
     (DL 2 NIL (PUSH-CONSTANT (NAT 2)))
     (POP-GLOBAL C-C)                                    ; end; {loop}
     (DL 0 NIL (NO-OP))
     (POP* 4)
     (RET)))
```

Figure 5-1: The Translation of the Procedure

5.2.1 The Program Segment

The Micro-Gypsy execution environment contains a list of Micro-Gypsy procedures and a Micro-Gypsy statement which is the *entry point*. Piton does not have a similar notion of an entry point statement; it is the program counter which gives the current point of execution in the Piton program segment. To handle this disparity, we create a new Micro-Gypsy procedure whose body is the entry point, compile the extended procedure list into a list of Piton procedures, and set the program counter to begin execution at the first statement in the translation of our entry point procedure. It is also necessary to affix to our Piton procedure list the list of (hand-coded) implementations of the Micro-Gypsy predefined operations. The structure of the resulting Piton program segment for our multiplication example is given below:

```
'(<translation of MG-SIMPLE-VARIABLE-ASSIGNMENT>
  <translation of MG-SIMPLE-CONSTANT-ASSIGNMENT>
   ...                                        ;; 11 more predefineds
  (SUBR (B I J K A) NIL ....)                 ;; entry point procedure
  (MG_MULTIPLY (K ZERO B ANS I J) NIL ...)
  (MG_MULTIPLY_BY_POSITIVE (K ZERO ONE B ANS I J) ...))
```

5.2.2 The Program Counter

Execution should begin at the point in the Piton program segment corresponding to the Micro-Gypsy entry point. This is the first statement of the special procedure constructed around the entry point. The name of this procedure, say **SUBR**, is given as a parameter by the user. Hence, the initial value of the Piton program counter is **(PC (SUBR . 0))**.

5.3 The Resource Limits

The fields **P-MAX-CTRL-STK-SIZE** and **P-MAX-TEMP-STK-SIZE** define the resource limitations of the Piton machine. Recall that we reflected the resource limitations of the Piton machine up to the Micro-Gypsy level so that we could track resource errors. This meant adopting some specific values—we actually use unspecified constants—for these constants in the Micro-Gypsy definition. These same values are used in the Piton state. The FM8502 implementation of Piton allows the programmer to allocate available resources among several structures. In general, it is not possible to know until load time whether particular choices for these two constants will be acceptable.

5.4 The Word Size

Cognizant of the implementation of Piton on the FM8502, the word size is fixed at 32 in the Micro-Gypsy definition. The **P-WORD-SIZE** field in the p-state is set to 32 as well.

5.5 The Program Status Word

The initial psw in the Piton state is **RUN**. It is shown as part of the code generator proof that exceptional conditions cannot be raised by the target program. Consequently, the psw remains **RUN** during the entire execution.

Figure 5-2 illustrates the p-state generated by a call to **MAP-DOWN** on our multiplication program with an appropriate initial state. The resource limits (**P-MAX-TEMP-STK-SIZE** and **P-MAX-CTRL-STK-SIZE**) are each set to 100. **SUBR** is a new procedure name supplied by the user.

```
(P-STATE '(PC (SUBR . 0))          ;; pc
         '((((B NAT 0) (I NAT 1)   ;; ctrl-stk
             (J NAT 2) (K NAT 3)
             (A NAT 4))
            (PC (SUBR . 0))))
         '((INT 76)                ;; temp-stk
           (INT 73)
           (INT 78)
           (INT 0)
           (INT 20)
           (INT -24)
           (BOOL T))
         '(<13 predefined ops>      ;; prog-segment
           (SUBR (B I J K A) NIL ....)
           (MG_MULTIPLY (K ZERO B ANS I J) NIL ...)
           (MG_MULTIPLY_BY_POSITIVE (K ZERO ONE B ANS I J) ...))
         '((C-C (NAT 1))           ;; data-segment
           (B (NAT 0))
           (I (NAT 0))
           (J (NAT 0))
           (K (NAT 0))
           (A (NAT 0) (NAT 0) (NAT 0) (NAT 0) (NAT 0)))
         100 100                   ;; resource limitations
         32 'RUN)                  ;; word size and psw
```

Figure 5-2: Piton State Generated by the Code Generator

Chapter 6
THE PROOF

The correctness theorem for the Micro-Gypsy code generator has been checked mechanically by the Boyer-Moore theorem prover enhanced with an interactive interface by Matt Kaufmann [14]. In this chapter we describe the overall structure of the proof. The interested reader is directed to [26] for more details.

Recall that our main theorem is a formalization of the following commuting diagram.

$$\begin{tikzcd}
\text{MG-Initial} \arrow[r, "\textit{MG-Meaning}"] \arrow[d, "\textit{Map-Down}"] & \text{MG-Final} \\
\text{P-Initial} \arrow[r, "k\ \textit{Piton steps}"'] & \text{P-Final} \arrow[u, "\textit{Map-Up}"']
\end{tikzcd}$$

If the **MAP-UP** function is a left inverse of **MAP-DOWN**, it is easy to see that the theorem follows whenever the following diagram commutes.

$$\begin{tikzcd}
\text{MG-Initial} \arrow[r, "\textit{MG-Meaning}"] \arrow[d, "\textit{Map-Down}"] & \text{MG-Final} \arrow[d, "\textit{Map-Down}"] \\
\text{P-Initial} \arrow[r, "k\ \textit{Piton steps}"'] & \text{P-Final}
\end{tikzcd}$$

The formal analog of this diagram is much more accessible to an inductive proof. Our proof stategy then was:

1. Show that **MAP-UP** is an appropriate left inverse[6] for **MAP-DOWN**.

[6]This assumes that the additional signature information is available.

2. Establish the formal analog of this second commuting diagram. This was stated in a conjecture we called the **`EXACT-TIME-LEMMA`**.

The first step was straightforward from the definitions of **`MAP-DOWN`** and **`MAP-UP`**.

The second step was more difficult because the version of **`MAP-DOWN`** described in chapter 5 is a very specific function unsuitable for inductive proof; it refers to an *initial* Micro-Gypsy execution environment, not to an *arbitrary* execution environment which might arise during program execution. We formulated a more general version called **`MAP-DOWN1`** which allows us to refer to the translation of an *arbitrary* Micro-Gypsy statement whereever it happens to fall within the Micro-Gypsy procedure list. **`EXACT-TIME-LEMMA`** is stated in terms of this more general mapping.

`EXACT-TIME-LEMMA` may be paraphrased as follows. Suppose that **`STMT`** is an arbitrary Micro-Gypsy statement somewhere within one of the procedures in **`PROC-LIST`**. Now consider an execution environment at just the moment at which we're to begin executing **`STMT`**. We should obtain identical results if we

1. **`MAP-DOWN1`** the execution environment and step forward in Piton **`k`** steps, or
2. interpret the environment with the Micro-Gypsy interpreter and **`MAP-DOWN1`** the result.

Here **`k`** is exactly the number of steps required for the translation of **`STMT`**.

Because the statement of the result is very general, it is amenable to proof by induction on the structure of Micro-Gypsy statements. Consider, for example, a Micro-Gypsy **`(PROG2 left right)`** statement somewhere within a Micro-Gypsy program. The commuting diagram for the **`PROG2`** statement is really the composition of the commuting diagrams for **`left`** and **`right`** as illustrated below.

$$\begin{tikzcd}
\text{MG-Initial} \arrow[r] \arrow[d] & \text{MG-State1} \arrow[d] & \text{MG-State2} \arrow[r] \arrow[d] & \text{MG-Final} \arrow[d] \\
\text{P-Initial} \arrow[r] & \text{P-State1} & \text{P-State2} \arrow[r] & \text{P-Final}
\end{tikzcd}$$

Prog2 Left *Prog2 Right*

In the **`PROG2`** case of the inductive proof of **`EXACT-TIME-LEMMA`**, we have inductive hypotheses which characterize the commuting diagrams for **`left`** and **`right`**. The key to the proof is formulating the induction such that the inductive hypotheses fit together to yield a proof of the theorem for **`PROG2`**. Some measure of the complexity of the induction is that the induction hint, given in the form of a definition in the Boyer-Moore logic, has 12 parameters and is over 250 lines long. See interested reader should see [26] for the details.

We use **EXACT-TIME-LEMMA** by showing that our **MAP-DOWN** function is a special instance of **MAP-DOWN1** and that the various structures in our initial state satisfy the various hypotheses of the **EXACT-TIME-LEMMA**.

Chapter 7
PROVING CORRECTNESS OF MICRO-GYPSY PROGRAMS

In this chapter we return to our simple Micro-Gypsy multiplication program and demonstate how it can be proved correct in either of two ways.

1. We can verify the annotated program in Gypsy-style syntax using the GVE.
2. We can prove the program in abstract prefix form directly using the interpreter semantics.

We discuss the advantages and disadvantages of each approach.

Recall our **`MULTIPLY_BY_POSITIVE`** routine in figure 2-2. The formal specification of the program is given in the form of program annotations indicated by the keywords **`entry`**, **`exit`**, and **`assert`**. The external specification asserts that, for non-negative values of parameter **`I`**, the final value of var parameter **`ANS`** will be the product of **`I`** and **`J`**. Notice that this specification is weaker than it might be. In particular, the **`exit`** specification is

```
exit ans = i * j
```

This is really an abbreviation for

```
exit case (is normal: ans = i * j;
           is routineerror: true);
```

indicating that the only cases of interest to the programmer are those in which no condition is raised. Such incomplete specifications are quite common for Gypsy programs. Gypsy is quite flexible in allowing the programmer to specify the program in more or less detail. We deliberately chose an incomplete specification to see how this is reflected in our two proofs.

7.1 The GVE Approach

Ours is a legal Gypsy program and hence we have available all of the facilities of the Gypsy Verification Environment for reasoning about it. The GVE implements the traditional Floyd-Hoare approach to verification. Verification conditions sufficient to guarantee the conformance of the program with its specification are generated and these are then proven interactively using the GVE proof checker.

For the routine **`multiply_by_positive`** four verification conditions (vc's) are generated, two of which are proven automatically by the vc generator. The remaining two vc's are show below.

```
Verification condition MULTIPLY_BY_POSITIVE#3
  H1: ANS + I * K = I * J
  H2: ANS + I in [MININT..MAXINT]
  H3: K - 1 in [MININT..MAXINT]
  H4: K in [0..J]
  H5: 1 le K
  H6: 0 le J
 -->
  C1: K - 1 in [0..J]

Verification condition MULTIPLY_BY_POSITIVE#4
  H1: ANS + I * K = I * J
  H2: K in [0..J]
  H3: 0 le - K
  H4: 0 le J
 -->
  C1: ANS = I * J
```

These are proven very easily in the GVE proof checker.

Using the GVE in this fashion has several advantages. Mechanical assistance is available for constructing and maintaining annotated Gypsy programs, generating verification conditions, proving them with the assistance of a proof checker with specialized knowledge of Gypsy structures, and maintaining a database of proved lemmas, vc's, and units. This renders the proof of a routine such as **`MULTIPLY_BY_POSITIVE`** extremely easy. From starting the GVE to having a completely verified routine took no more than 10 minutes on this example.

The strong disadvantage of this approach is that the semantics of Gypsy embodied in the GVE verification condition generator and the GVE prover's algebraic simplifier may be different than the interpreter semantics used in the code generator proof. Care was taken to assure that our interpreter semantics is matched the "official" semantics of Gypsy, but we have no formal assurance that this is so. The upshot is that if we consider the verified program as being yet another level in our verified stack there is an assurance gap arising from the potential disparity in the semantics at the interface of the two levels.

7.2 Verifying Against the Interpreter Semantics

We can avoid this potential disparity by verifying our program directly using the semantics provided by our interpreter definition. This essentially involves generating the appropriate verification conditions by hand. An abbreviated version of the correctness theorem for **`MULTIPLY_BY_POSITIVE`** is shown in figure 7-1. The conclusion asserts that a call of the form

```
       '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS X Y) NIL)
```

has the effect of setting **`ANS`** to the Micro-Gypsy representation of the integer product of the values of integer variables **`X`** and **`Y`** in the current state.

```
Theorem.  MG-MULTIPLY-BY-POSITIVE-CORRECTNESS
(IMPLIES
 (AND
  (OK-MG-STATEMENT                                                    ; 1
   '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE (ANS X Y) NIL)
   COND-LIST (MG-ALIST MG-STATE) PROC-LIST)
  (OK-MG-STATEP MG-STATE NIL)                                         ; 2
  (NORMAL MG-STATE)                                                   ; 3
  (EQUAL                                                              ; 4
   (FETCH-DEF 'MG_MULTIPLY_BY_POSITIVE PROC-LIST)
   '(MG_MULTIPLY_BY_POSITIVE ...
        <abstract prefix version of the routine>))
  (LESSP (PLUS 4                                                      ; 5
           (TIMES 4 (ADD1 (UNTAG (GET-M-VALUE
                                  'Y (MG-ALIST MG-STATE))))))
         N)
  (NUMBERP (UNTAG (GET-M-VALUE 'Y (MG-ALIST MG-STATE))))              ; 6
  (SMALL-INTEGERP                                                     ; 7
     (ITIMES (UNTAG (GET-M-VALUE 'Y (MG-ALIST MG-STATE)))
             (UNTAG (GET-M-VALUE 'X (MG-ALIST MG-STATE))))
     (MG-WORD-SIZE)))

 (EQUAL
  (MG-MEANING '(PROC-CALL-MG MG_MULTIPLY_BY_POSITIVE
                             (ANS X Y) NIL)
              PROC-LIST MG-STATE N)
  (MG-STATE
     'NORMAL
     (SET-ALIST-VALUE
      'ANS
      (TAG 'INT-MG
           (ITIMES (UNTAG (get-m-value 'Y (MG-ALIST MG-STATE)))
                   (UNTAG (get-m-value 'X (MG-ALIST MG-STATE)))))
      (MG-ALIST MG-STATE))
     (MG-PSW MG-STATE))))
```

Figure 7-1: MG-MULTIPLY-BY-POSITIVE-CORRECTNESS

The hypotheses formalize the following assumptions.

1. Our call statement is a legal Micro-Gypsy statement.
2. The state in which we are interpreting is well-formed.
3. The **CC** is **'NORMAL**.
4. The definition of **MG_MULTIPLY_BY_POSITIVE** in the procedure list is exactly that given in figure 2-3.
5. The clock parameter **N** to the interpreter is adequate to carry out the multiplication without timing out. Calculating an appropriate lower bound is one of the most difficult aspects of formulating this theorem.
6. The multiplicand **Y** has a non-negative value. This corresponds to the entry specification **entry j ge 0** of **MULTIPLY_BY_POSITIVE**.
7. The product of **X** and **Y** is representable as a Micro-Gypsy **INT-MG** value and can be stored in the variable **ANS**.

The lemma **MG-MULTIPLY-BY-POSITIVE-CORRECTNESS** has been proven in the Kaufmann-enhanced Boyer-Moore theorem prover. Refining the lemma, proving the supporting lemmas, and completing the proof took approximately two days of fairly intense effort.

Notice that the specification provided by this theorem is much more complete than that supplied by the annotated program.

1. Proof of the Gypsy specification only assures *partial* correctness; the proof might still succeed in cases where the routine was non-terminating. Our proofs in the Boyer-Moore framework are easier if we require totality and perform the awkward computation of an adequate clock value.
2. The **`MULTIPLY_BY_POSITIVE`** exit specification says (implicitly) that the programmer is not concerned with cases in which conditions are signaled. In the GVE proof, paths corresponding to such cases are handled by the vc generator and generate trivial vc's. In the Boyer-Moore formalism, it is necessary to characterize explicitly the situation in which conditions will not arise.

Despite the apparent extra complexity, proving the program with respect to the semantics provided by **`MG-MEANING`** has three very strong advantages over using the GVE to prove Micro-Gypsy programs.

1. The semantics assumed in the proof is exactly the semantics assumed in the compiler.
2. The GVE accepts programs in standard Gypsy syntax. Compiling these programs with our verified translator means an error-prone hand translation to our abstract prefix form.[7]
3. Soundness of the logic and the care with which it is implemented in the theorem prover are strong advantages of the Boyer-Moore proof system over the GVE. There is empirical evidence over 15 years for virtually bug-free performance of the Boyer-Moore prover that has not been matched by the Gypsy implementation.[8]

Some of the benefits of a GVE-style proof could be gained while retaining these advantages by writing a verification condition generator for Micro-Gypsy within the Boyer-Moore framework. This is a separate research topic which we have not investigated in detail, though some research has been aimed in this direction [24].

[7]We intend to write a verified preprocessor, but this has not yet been done.

[8]We in fact used Matt Kaufmann's interactive interface to the Boyer-Moore prover in our proofs. However, this uses the same logic and is a relatively benign set of enhancements.

Chapter 8
RELATED WORK

We have followed a long tradition in defining our languages in an operational style. McCarthy [17] seems to have been the first to define a language (LISP) operationally. It was realized quite early that operational (interpreter) style definitions provided a means of investigating a variety of implementations and opened the possibility of proving the equivalence of interpreters [15, 12].

The first attempt to prove compilation correct via an interpreter equivalence proof seems to be the proof of McCarthy and Painter [18] of a simple expression compiler. Various extensions to this work have been reported [6, 19, 7, 1, 4]. Other interpreter equivalence proofs of direct relevance to ours are reported in [13, 21, 2].

Several compiler proofs have used axiomatic semantics [8, 16] and much work has been directed toward specifying and proving compilers using denotational semantics [20, 9],

The most notable previous mechanical compiler proof is by Polak [23]. Polak uses denotational semantics to describe both the source and target languages. His work is less rigorous than ours; his proof has as a basis a large collection of unproved assumptions within the formal theory. Polak's work also does not have the larger context of the verified stack.

See [26] for a more extensive discussion of the literature of compiler verification.

Chapter 9
CONCLUSIONS

The principle contributions of this project are two-fold. We have demonstrated the feasibility of providing a rigorous mechanically-checked proof of a code generator. We have also contributed to providing a framework for constructing highly reliable application programs.

9.1 A Rigorous Proof

Micro-Gypsy is a small language but contains enough functionality to illustrate the viability of our approach. We feel that there is no conceptual difficulty in extending our work in various ways, some of which we outline below. Because Micro-Gypsy is a subset of an existing language, Gypsy 2.05, we were not free to tailor our source language to make the compilation process trivial. Because our target language was also pre-existing, we were not free to choose assembly language features which would narrow the semantic gap between source and target language.

We believe that the implementation is realistic. In particular, though our Micro-Gypsy interpreter uses *call by value-result* semantics, the implementation uses the more efficient *call by reference*. We believe that our project is the first instance of a mechanically checked proof involving a call by reference implementation of a call by value-result semantics.

Our translator was entirely specified in the Boyer-Moore logic and the proof carried out using the Kaufmann-enhanced Boyer-Moore theorem prover. The proof is fully mechanically checked. There are two explicit axioms assumed in the proof; these insure that the two declared constants representing the maximum sizes for the Piton temporary stack and control stack are numbers less than 2^{32}. The only other assumptions of the proof are the axioms provided by the Boyer-Moore implementation and the axioms added as a result of function and shell definitions. We have been extremely careful in formulating definitions, but there is always a possibility that our definitions do not reflect the desired intuition. This, unfortunately, is a hazard of program verification which can be minimized but never entirely eliminated.

9.2 Trusted Systems

A frequent criticism of program verification is that there is a large semantic gap between verified high-level language programs and their ultimate representation in machine language running on real hardware. The Micro-Gypsy project is a component of a larger project designed to partially bridge this gap. We have illustrated two ways in which Micro-Gypsy programs could be proved. These programs and accompanying Micro-Gypsy execution environments can then be compiled using the verified Micro-Gypsy code generator into Piton execution environments in such a way that the program semantics are provably preserved. Piton has been proven to be correctly implemented on the FM8502 microprocessor which, in turn, has been verified down to the gate level.

The "stack" of verified components--the Micro-Gypsy code generator, Piton assembler, and FM8502--provides an environment for building verified applications with a much higher degree of reliability than any methodology previously available. This environment could be extended in various ways discussed below, but we believe the Micro-Gypsy code generator to be a significant tool for furthering our ultimate goal of building highly reliable programs.

9.3 Future Work

There are a number of potentially useful directions for future research building upon the current work.

The Micro-Gypsy code generator, sitting as it does on the Piton and FM8502 work, provides the capability of building extremely reliable applications. However, the usability of the language in its current form is questionable. Building some small applications in which correctness is critical would both show the viability of the methodology and point out deficiencies of the language.

Some deficiencies in the current subset are apparent. The language was pared down to illustrate the feasibility of constructing a rigorously verified code generator while keeping the project manageable. The result is a language with limited functionality but one which we believe can be extended in a number of ways--additional types, a real expression language, I/O facilities.

There is currently a gap between the syntax of the Micro-Gypsy programs which can be handled in the Gypsy Verification Environment and the abstract prefix syntax acceptable to the code generator. We envision this gap being bridged by a preprocessor. This should be defined and verified.

The code generator does not currently have an optimization phase. It would be relatively easy to define and prove a peephole optimizer that operated on the code generator output. This would be an interpreter equivalence proof where both the source and target language interpreters were the Piton interpreter. An initial study has been done on this [25].

References

1. R. Aubin. *Mechanizing Structural Induction*. Ph.D. Th., University of Edinburgh, Edinburgh, Scotland, 1976.

2. W.R. Bevier. Kit: A Study in Operating System Verification. Tech. Rept. CLI-28, Computational Logic, Inc., August, 1988.

3. W.R. Bevier, W.A. Hunt, J S. Moore, W.D. Young. Overview of the Short Stack. Tech. Rept. In Progress, Computational Logic, Inc., 1989.

4. R.S. Boyer, J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

5. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

6. R.M. Burstall. "Proving Properties of Programs by Structural Induction". *Computer Journal 12*, 1 (February 1969), 41-48.

7. R. Cartwright. *A Practical Formal Semantic Definition and Verification System for Typed LISP*. Ph.D. Th., Stanford University, 1976.

8. L.M. Chirica and D.F. Martin. An Approach to Compiler Correctness. Proceedings of the International Conference on Reliable Software, April, 1975, pp. 96-103.

9. A. Cohn. *Machine Assisted Proofs of Recursion Implementation*. Ph.D. Th., University of Edinburgh, Edinburgh, Scotland, 1979.

10. D.I. Good, R.L. Akers, L.M. Smith. Report on Gypsy 2.05. Tech. Rept. CLI-1, Computational Logic, Inc., October, 1986.

11. D.I. Good, B.L. Divito, M.K. Smith. Using The Gypsy Methodology. Tech. Rept. Draft CLI-2, Computational Logic, Inc., January, 1988.

12. W. Henhapl and C.B. Jones. The Block Structure Concept and Some Possible Implementations. Tech. Rept. 25.104, IBM Laboratories, Vienna, 1970.

13. W.A. Hunt. The Mechanical Verification of a Microprocessor Design. Tech. Rept. CLI-6, Computational Logic, Inc., September, 1986.

14. Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. CLI-19, Computational Logic, Inc., May, 1988.

15. P. Lucas. Two Constructive Realizations of the Block Concept and Their Realization. Tech. Rept. 25.082, IBM Laboratories, Vienna, 1968.

16. D.S. Lynn. Interactive Compiler Proving Using Hoare Proof Rules. Tech. Rept. ISI/RR-78-70, Information Sciences Institute, January, 1978.

17. John McCarthy. Towards a Mathematical Science of Computation. Proceedings of the IFIP Congress, Amsterdam, 1962.

18. John McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. Proceeding of Symposium on Applied Mathematics, American Mathematical Society, 1967.

19. R. Milner and R. Weyhrauch. Proving Compiler Correctness in a Mechanized Logic. In *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, Scotland, 1972, pp. 51-70.

20. R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, 1976.

21. J S. Moore. A Mechanically Verified Language Implementation. Tech. Rept. CLI-30, Computational Logic, Inc., September, 1988.

22. J S. Moore. PITON: A Verified Assembly Level Language. Tech. Rept. CLI-22, Computational Logic, Inc., June, 1988.

23. W. Polak. *Compiler Specification and Verification.* Springer-Verlag, Berlin, 1981.

24. L.C. Ragland. *A Verified Program Verifier*. Ph.D. Th., University of Texas at Austin, 1973.

25. W.D. Young. A Verified Optimizer for Pico-Piton. Internal Note 107, December, 1988, Computational Logic, Inc., Austin, Texas.

26. W.D. Young. *A Verified Code Generator for a Subset of Gypsy*. Ph.D. Th., The University of Texas at Austin, December 1988.

Table of Contents

List of Figures

List of Tables