Functional Programming on a Stack-Based Embedded Processor

Andrew J. Harris
andrew.harris@jhuapl.edu

John R. Hayes
john.hayes@jhuapl.edu

The Johns Hopkins University
Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel, MD 20723

Abstract

This paper describes the implementation of a functional programming language interpreter. The interpreter is written in Forth and runs on any Forth system, including a novel processor called the Scalable Configurable Instrument Processor (SCIP), designed at the Johns Hopkins Applied Physics Laboratory. The combination of this novel processor, Forth, and functional programming provides a layering of simple technologies that yields a unique programming and execution environment. The SCIP processor also has a clear transition path to use in space-borne applications.

1. Overview

1.1. Motivation

The goal of this work is to provide an ability for embedded flight software applications to execute small functional programs in a programming language similar to Haskell [4]. However, large general purpose functional programming language systems like Haskell provide many features that aren't accessible to space qualified processing environments that generally provide only limited resources and typically run a real time operating system such as VxWorks. We illustrate a demonstrative evaluator that enables the evaluation of small functional programs within a constrained embedded environment.

The motivation for looking specifically at a functional programming language is threefold. First, as will be discussed briefly in section 1.4, functional programs have a mathematical style to them, which can increase the ability to reason and prove things about the behavior of programs. Secondly, because of this style, functional programming languages are expressive, meaning more functionality can be expressed in fewer lines of code. Many details are abstracted away from the application programmer. Lastly, functional programming languages like Haskell are statically typed. Programs can be guaranteed to be type-safe at compile time. This is at odds with many current interpreted languages.

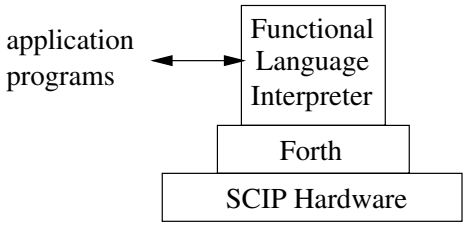

Figure 1. Software and Hardware Layering

The ability to interpret functional programs within a spacecraft would enable execution of dynamically uploadable maintenance and autonomy procedures without requiring an entire flight software upload. On-board procedure execution would also help to alleviate the burden of long round trip light times or missions that are out of Earth contact for long periods of time. For example many open loop operations requiring contact with mission operators could become closed loop operations by encoding the requisite verification procedure as an on-board script.

Figure 1 illustrates the general architecture of the work described in this paper.

1.2. SCIP – A Stack Based Processor

Computer systems designed for space flight require a radiation-hardened processor that provides reasonable performance with minimal power consumption. The number of

2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06)
0-7695-2644-6/06 $20.00 © 2006 IEEE

processors that meet these requirements is small and diminishing. Given the increasing capability of rad-hard Field-Programmable Gate Arrays (FPGAs) and the maturity of hardware synthesis tools, we decided to develop our own processor: the SCIP.

The SCIP [2] is a stack processor. Stack processors allow efficient instruction encoding. The majority of SCIP's instruction opcodes are 16 bits, and this instruction encoding keeps program volume small for instrument applications. Stack processors are also good targets for simple compilers. SCIP's data path is scalable: 16-bit or 32-bit versions of SCIP can be instantiated. The processor is easily configured with mission-specific peripherals.

SCIP's instruction set is designed to efficiently implement Forth [1]. Forth programs typically define many short functions, and consequently, function call and return are the most frequently executed language primitives. SCIP executes a call in one cycle and most returns take zero cycles. Also, SCIP uses stack caches to provide the program with the illusion of arbitrarily large on-chip stacks. Access to the stack is as fast as a register access.

1.3. Forth Programming

Forth was designed as an embedded systems language, so techniques required by embedded systems programmers are readily available in Forth. It is straightforward to access memory by address, so shared and I/O mapped memory areas are easily accessible. Forth can be extended to provide support for concurrent programming with tasks, mutual exclusion semaphores, and event timers.

When programming in Forth, functions are defined starting with the function definition operator "`:`". This operator is followed by the name of the function to be defined, the list of Forth instructions comprising the function, and by the end of function marker "`;`". For example, one possible definition of a function that expects a number on the stack and returns the product of the number multiplied by itself could be written:

```
: square
   dup *
;
```

This function, called `square` duplicates the top element of the data stack using the Forth `dup` instruction and then multiplies the top two stack elements with the multiplication operator `*`. Forth is postfix, so the `square` function defined above would be used in the following way:

```
9 square .
81 ok
```

The operator "`.`" used above simply pops and prints the value at the top of the data stack (in this case, 81). The symbol `ok` is returned from the Forth interpreter to signify it is ready to accept more input.

1.4. Functional Programming

Functional programming gets its name from the inclination to treat functions as first class objects. A first class object is one that can be generated programmatically. The C language doesn't treat functions as first class objects, because the C language doesn't support generation of new functions at run-time. In a functional programming language, function applications are created in the heap and are garbage-collected when no longer needed, just like any other program construct. While this may seem at first to be an esoteric feature, it is pervasive in a functional programming environment.

A functional program is a set of definitions of functions. In general there is a special definition called `main` that represents the first function application evaluated in the program. For example, a simple functional program (that evaluates to 3) contains definitions for the identity function `I` and `main`:

```
I x = x;
main = I 3
```

Since each definition closely resembles a mathematical statement of equality, it is said that functional programming is declarative – a program resembles a specification more than a step by step algorithmic procedure. This likeness of functional programs to specifications carries benefits in reasoning about the correctness of programs.

2 Approach

Note that the simple functional program described in Section 1.4 looks nothing like Forth. This paper illustrates several of the key steps by which functional programs can be transformed for execution on Forth-based systems. The particular execution model discussed is an interpreter that interprets instructions for an abstract machine called the G-Machine [7].

Functional programs are compiled to a sequence of G-Machine instructions. Compilation permits some details to be moved from execution time to compile time, resulting in better execution time performance.

Using the approach described in [7], the execution of a functional program can be considered a four step transformation:

1. Develop the program in a high level functional programming language.

2. Transform this high level functional program into a simpler restricted functional programming language called the *Core Language*, henceforth called Core. Syntactic sugar provided by the high level functional language is removed, and type-checking is done during this step.

3. Transform the Core program into a sequence of G-Machine instructions. Apply any suitable optimizing transformations.

4. Load and execute the G-Machine instruction sequence on a G-Machine based run-time system.

This section will briefly discuss one important aspect of step 2: how the Core language supports pattern matching – a useful feature of high level functional programming languages. Then, the bulk of the discussion will focus on steps 3 and 4, including a description of the components of the G-Machine state machine, a simple example of the state machine in operation with the `Mkap` instruction, and a discussion of a few of the important data structures useful during compilation of text-based Core programs.

2.1. Pattern Matching

Modern functional programming languages support a concept called pattern matching. For example, the standard definition of the `len` function, which computes the length of an arbitrary list is:

```
len []     =  0
len (x:xs) =  1 + (len xs)
```
(1)

The first line of Equation 1 represents the definition of `len` used when the list passed as an argument is the empty list (denoted by `[]`). The second line represents the definition used when the argument is not the empty list. The notation `(x:xs)` binds `x` to the first element of the list, and binds `xs` to the remainder of the list. The first and second lines are definitions of the `len` function with different argument patterns. When the function is called with a specific argument, the proper definition is invoked based on the structure of the argument. This is called pattern matching.

Figure 2 illustrates a canonical list, a list containing the elements 1, 2, and 3. Lists are constructed out of two structural elements, called constructors: `cons` and `nil`. The G-Machine supports constructors by providing a G-Machine instruction called `Pack` (listed in Figure 3) that supports the definition of data structures. As such, `cons` and `nil` can be defined:

```
cons head tail  =  Pack{2,2} head tail;
nil             =  Pack{1,0};
```
(2)

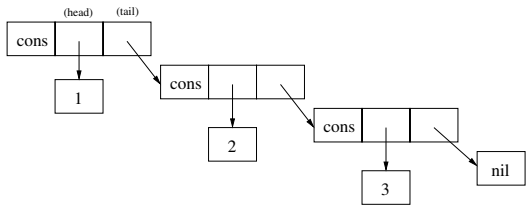

Figure 2. Structure of a List

The expression `Pack`$\{x, y\}$ contains enough information about the data structure to describe it. The x value is a tag: a unique number that is used to differentiate each constructor from all others. The y value is the arity of the constructor, or the number of associated arguments to the constructor. For example `cons` has arity two, while `nil` has arity zero. These are in accordance with Figure 2, where each `cons` node contains two additional pointers: a "head" pointer followed by a "tail" pointer. The `nil` node contains no additional pointers.

Once the constructors are defined, they may be used as targets of pattern matching. Pattern matching is supported using a Core language construct called `case`. In general, `case` is used as such:

```
f z  =  case z of
        <tag_1> bindings_1 -> expr_1
        <tag_2> bindings_2 -> expr_2
        ⋮
        <tag_n> bindings_n -> expr_n
```
(3)

For example, the `len` function from Equation 1 can be written in the Core language in the following way:

```
len list  =  case list of
             <1>       -> 0,
             <2> x xs  -> 1 + (len xs);
```
(4)

The first possibility, containing `<1>` as the target tag matches the `nil` constructor defined with a tag of `1` in Equation 2. The second possibility, `<2>`, matches the `cons` constructor. Note `cons` requires two arguments, as specified by its arity from Equation 2. Both `cons` arguments are bound to names in the $bindings_2$ section, and (only) one of them is used in the target expression $expr_2$.

Based on this method, the Core language supports pattern matching, a useful feature of functional programming languages.

2.2. G-Machine State

The G-Machine is a state machine. The state of the G-Machine is a 6-tuple containing the following elements:

```
Unwind      Pushglobal  Pushint
Push        Mkap        Update
Pop         Alloc       Slide
Eval        Add         Sub
Mul         Div         Neg
Eq          Ne          Lt
Le          Gt          Ge
Purge       Pack        Casejump
Cond        Split       Print
Pushbasic   Mkbool      Mkint
Get
```

Figure 3. G-Machine Instructions

- i – The current instruction stream. Contains the G-Machine instructions to be executed.
- s – A stack of heap addresses. This stack is operated on by many of the G-Machine instructions listed in Figure 3. This stack is called the s-stack.
- d – The current dump stack. Used to store the current machine state when an `Eval` instruction is executed. When the dump stack is empty, the evaluation is complete.
- v – Allows primitive arithmetic operations to not require heap allocated operands. The use of the v-stack is an optimization to the standard G-Machine. In this implementation the Forth data stack serves as the v-stack.
- h – The contents of the heap.
- m – The set of predefined function definitions. These are the source for *i*, the instruction stream.

2.3. Run-time System

Each G-Machine instruction in Figure 3 interacts with the G-Machine state in a prescribed way. For example, the `Mkap` instruction is used to make a function application from the top two heap addresses in the s-stack. The `Mkap` instruction is typically used in a sequence of G-Machine instructions as follows:

```
[Pushint 3, Pushglobal I, Mkap, ...]
```
(5)

The above set of G-Machine instructions pushes the number 3 and the definition of `I` onto the s-stack, makes a function application applying `I` to 3 in the heap, pushes this heap address onto the s-stack, and then execution continues with subsequent instructions. Figure 4 is a graphical illustration of the s-stack during these operations. Since `Mkap` only supports applying a function to a single argument, functions that take multiple arguments are supported via currying [6].

To more formally describe the operation of the `Mkap` instruction, a state transition definition is shown, using notation identical to that used in [7]. The first line illustrates the state of the machine before the instruction, and the second line illustrates the state after execution:

$$\begin{array}{cccccccc} \texttt{Mkap} : i & s_1 : s_2 : s & d & v & h & & m & \\ i & h_1 : s & d & v & \underbrace{\texttt{NAp}\ s_1\ s_2}_{h_1} : h & & m & (6) \end{array}$$

The Forth implementation is shown below. The two calls to Forth function `sStackPop` pop the top two values from the s-stack. The `hAlloc` function is used to allocate a new heap node, in this case a node of type `NAp` (illustrated in Figure 5). `hAlloc` returns the heap address of the newly allocated element. This heap address is pushed onto the s-stack using `sStackPush`:

```
: domkap
    sStackPop
    sStackPop
    NAp
    hAlloc
    sStackPush
;
```

Note the above Forth code and Equation 6 are equivalent.

2.4. Core To G-Machine Instructions

Step 3 from the list defined in Section 2 transforms, or compiles, the Core program into a sequence of G-Machine instructions. The Core to G-Machine instruction compiler has been implemented in the Haskell programming language [9], due to its support of many useful libraries, including a powerful parsing library called *Parsec*.

The first step of the transformation from Core to G-Machine instructions involves transforming the text based Core program into an internal representation that is suitable for algorithmic manipulation. Closely mirroring the text based structure, the internal representation is a list of definitions:

```
type CoreProgram = [ ScDefn Name ]
```

Each definition is a 3-tuple, consisting of a name, a list of named arguments, and an expression:

```
type ScDefn Name =
   ( Name, [Name], Expr Name )
```

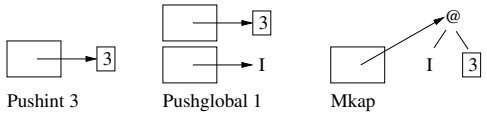

Figure 4. The `Mkap` Instruction

The `Expr Name` component of the above definition can represent an arbitrary expression: a variable (`EVar`), a function application (`EAp`), or any of the possibilities specified in the `Expr` data type below (again specified in Haskell):

```
data Expr a
=  EVar Name                  -- Variables
 | ENum Int                     -- Numbers
 | EConstr Int Int -- Constr, Tag, Arity
 | EAp (Expr a) (Expr a)  -- Application
 | ELet                 -- Let Expression
     IsRec     -- Boolean, True = letrec
     [(a, Expr a)]        -- Definitions
     (Expr a)         -- Body of let(rec)
 | ECase              -- Case Expressions
     (Expr a)               -- Predicate
     [Alter a]            -- Alternatives
```

As an example, the following Core program:

```
I x = x;
main = I 3
```

is converted into the following internal representation:

```
[("I",["x"],EVar "x"),
 ("main",[],EAp (EVar "I") (ENum 3))]
```

This internal representation is subsequently converted into a series of G-Machine instructions in preparation for execution on the G-Machine run-time system. The definition of `main`:

```
("main",[],EAp (EVar "I") (ENum 3))
```

is compiled into the set of instructions illustrated in Section 2.3, equation 5.

3 Heap Management

The heap is largely independent of the G-Machine runtime system. While several instructions from Figure 3 use the `hAlloc` Forth function to request heap space for a new heap node, the run-time system leaves most heap node management to the garbage collector. The garbage collector is a separate function, also written in Forth.

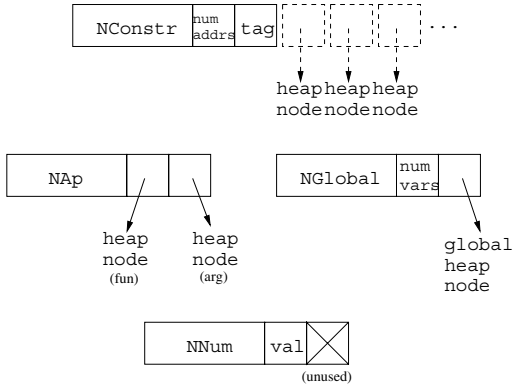

Figure 5. Heap Node Structures

3.1. Heap Structure

There are four types of heap nodes, each illustrated in Figure 5. `NConstr` nodes provide support for data types. Since user defined data structures can be arbitrary, `NConstr` nodes must support the aggregation of a variable number of heap addresses. This complicates garbage collection, but eliminates the need to chain multiple fixed sized heap nodes together to simulate variable sized nodes. `NAp` nodes contain function applications. The function to be applied is in the left node and its argument is in the right node. `NNum` nodes hold heap allocated numbers, required when the compiler can't deduce the ability to use the v-stack optimization.

`NGlobal` nodes point to predefined function definitions which are stored in the global heap. The global heap is a section of the heap that is not garbage collected, because it holds the G-Machine instruction sequences. The global heap can be considered the "text segment" of the application. To keep track of the contents of the global heap, a mapping is provided that maps the names of each of the predefined function definitions to heap addresses of associated `NGlobal` nodes (described in Figure 5). The non-dotted-line portion of Figure 6 shows the mapping structure. This mapping is used by the `Pushglobal` G-Machine instruction from Figure 3 in order to push the heap address of the requested function definition's `NGlobal` heap node onto the s-stack.

3.2. Garbage Collector

In order to run larger programs, it is useful to implement a garbage collector that reclaims heap memory discarded as the program executes. A two space stop and copy garbage collector is attractive for several reasons. For one, it compacts the heap at each collection, avoiding memory fragmentation issues. Stop and copy is also immune to the

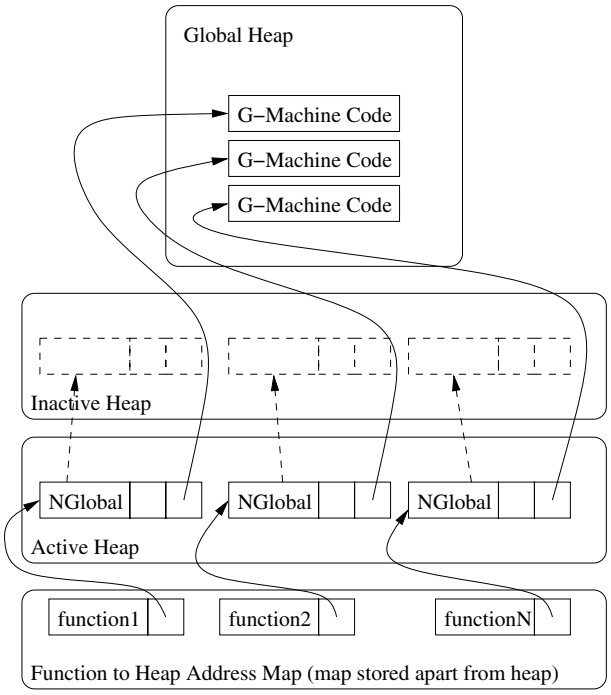

Figure 6. Step 1 of Garbage Collection

problem of unfreeable mutually referential cycles plaguing many implementations that rely on reference counting. Last but not least, stop and copy is relatively easy to implement.

The goal of the collector is to move all relevant nodes in the active heap over to the inactive heap. Then the inactive heap is made the new active heap, and evaluation continues. A general description of the process of garbage collection is discussed below:

1. The mapping of predefined function definitions to heap addresses is moved first – each heap element in the mapping is moved to the inactive heap. (note the element in the global heap is not touched). This activity is shown through dotted lines and arrows in Figure 6.

2. Next, each heap address in the s-stack is visited and each node is moved to the inactive heap.

3. The inactive heap is scanned, and all nodes that are reachable that still reside in the active heap are brought over to the inactive heap.

4. Finally, the inactive heap becomes the active heap. At this point, the new active heap has no references to the old heap, as all reachable elements have been brought over. In addition, the newly active heap has been compacted.

Since the garbage collector is independent of the G-Machine run-time system, the current stop and copy collector could be replaced with any of a variety of different collectors that have different performance characteristics.

4. Results

One may develop simple functional programs and download and execute them on Forth based embedded platforms. A more substantial example than that described in Section 1.4 is shown below. Note the `map` function is not defined for brevity but assumes its standard definition of applying a function to a list, similar to its definition in other functional programming languages.

```
double x   = x + x;
downfrom n = if (n == 0)
               nil
               (cons n (downfrom (n-1)));
main       = map double (downfrom 4)
```

The result of execution of `main` in the above example results in the list `[8,6,4,2]`. The execution of this program consumes, as of this writing, 888 bytes of heap, with the result occupying 264 bytes of heap. This implies 624 bytes of garbage were generated.

The performance characteristics of this specific implementation are not the primary result of this work. The significant result is that the ability to develop a working interpreter for functional programs is within the grasp of a typical embedded software developer: this implementation was built in approximately 2 staff months.

Note the expressiveness of the above 5 line program. The program involves the construction and iteration over a linked list data structure – a set of operations that would take at least an order of magnitude more lines of code if it were written in C.

5. Related Work

There are other ways to evaluate functional programs. There are closure-reducers such as the TIM machine [7], and specialized reducers such as the TIGRE machine [5] that treats the program graph itself as an executable program and executes the graph itself to perform the reduction. Indeed there is a Forth implementation of the TIGRE evaluator. We opted to perform our work using the standard G-Machine because it is well documented and well understood. Haskell, a modern functional programming language uses a variant of the G-Machine model, called an STG-Machine, or spineless, tagless G-Machine [8].

6. Future Work

Ideally, programs aren't written directly in the Core language, the language which is the focus of this paper. Programs are instead written in a high level language and subsequently compiled to Core. Access to a high level language would provide more familiarity and features than programming directly in the Core language. This would be a useful future addition.

Adding a compile-time type checker would increase the robustness of programs by allowing type discrepancies to be detected before execution.

Adding support for stateful computations using monads or arrows [3] would provide the ability to write programs that require a method to preserve the result of a computation and allow this result to be carried between computations.

7. Conclusion

We don't intend for an evaluator of this type to be used for time critical real-time software such as an interface to a MIL-STD-1553-B bus, or attitude control software. However, there are many areas of flight software that are not time critical that would benefit from having access to an evaluator capable of evaluating small functional programs.

A Core language to G-Machine instruction compiler and a G-Machine run time system have been built. This has been done in a reasonable amount of time and is within the reach of a typical embedded software developer. The compiler, discussed briefly in Section 2.4, is written in Haskell. The G-Machine run time system, described in Section 2.3, is written in Forth and can run on a variety of implementations of Forth, including one developed for a novel stack processor developed at JHU/APL.

References

[1] E. K. Conklin and E. D. Rather. *Forth Programmer's Handbook*. FORTH Inc., 1998.

[2] J. R. Hayes. The architecture of the scalable configurable instrument processor. Technical Report SRI-05-030, The Johns Hopkins Applied Physics Laboratory, 2005.

[3] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[4] S. P. Jones, editor. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[5] P. Koopman Jr. *An Architecture for Combinator Graph Reduction*. Academic Press, 1990.

[6] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[7] S. L. Peyton-Jones and D. R. Lester. *Implementing Functional Languages: A Tutorial*. Prentice Hall, 1992.

[8] S. L. Peyton-Jones and S. Marlow. The stg runtime system (revised), 1999. http://www.haskell.org/ghc/docs/papers/runtime-system.ps.gz.

[9] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.