The CURRY Chip

John D. Ramsdell*
The MITRE Corporation
Bedford, MA 01730

Abstract

The CURRY chip is a combinator reduction machine in VLSI. Normal order evaluation is implemented using a pointer reversal scheme that stores the stack in the cells representing the function. Program evaluation maps an input stream to an output stream. Methods used to write sisable programs for the chip are given, along with experience gained using super combinators.

1 Introduction

Turner [Tur79b] implemented a functional programming language based on combinators, a small finite set of functions that form the basis for defining all control structures in his language. His system can be used to represent functions that compute numbers in a form in which all bound variables are removed. The absence of variables allows an extremely simple mode of function evaluation.

The combinator programming system described within builds on the above work by demonstrating a system that can be implemented in VLSI, and which executes interesting programs, such as a compiler that translates abstractions into combinators. A novel feature of the VLSI implementation is an evaluation method that requires no additional storage for an evaluation stack. This method is equally useful in software implementations of combinator reduction machines.

The paper is divided into two main topics. The first describes the VLSI implementation of a combinator machine that has been fabricated using a silicon compiler. The second gives the experience gained by writing a compiler of abstractions into combinators for the CURRY chip, using a simulator. In this paper, programs are denoted using the syntax of the λ-calculus[Bar84, page 22], even though a slightly more sophisticated notation was used in writing the actual programs.

2 Sequential Combinator Machines

The CURRY chip implements a sequential combinator machine in silicon[Ram85]. Similar to many previous works, a sequential combinator machine represents a function as a graph in computer memory and the process of evaluation consists of overwriting functions with simpler but equivalent representations of the functions. A novel handling of the stack, and the selection of a fixed set of combinators that can fit on a chip, differentiate the CURRY chip from previous efforts.

2.1 Combinators

A function is represented as one of the twelve combinators in Table 1 or as an application. The combinators are atomic, but an application is a pointer to a pair of functions. The cell which contains the pair of functions is divided into a head and a tail. The cell represents the function obtained by applying the function in the head to the one in the tail.

Combinators direct the replacement of functions by simpler but equivalent representations of functions using the rules in Table 1. For example, the rule for

*Supported by MITRE IR&D 90580.

I	$\lambda x.x$
J	$\lambda x.I$
Y	Fix
R	Read
K	$\lambda xy.x$
T	$\lambda xy.yx$
S'	$\lambda xy.S(Bxy)$
C'	$\lambda xy.C(Bxy)$
S	$\lambda xyz.xz(yz)$
B	$\lambda xyz.x(yz)$
C	$\lambda xyz.xzy$
P	$\lambda xyz.zxy$

Table 1: CURRY Chip Combinators

K states that the second argument is ignored and the first is the value.

$$\forall x \forall y (Kxy) = x.$$

The rule for K is implemented by overwriting (Kxy) by (Ix) as shown in Figure 1. Figures 2 and 3 give the rules for combinators S and Y respectively. See [Tur79b] for more on these rules.

2.2 Evaluation

Reduction is the process of replacing a function's representation by a simpler, but equivalent representation. Normal order evaluation[1] is the name for the order in which applications are reduced. Roughly speaking, functions are given their arguments unevaluated in this order. This order may be implemented with the following evaluation algorithm: When a combinator is in the head of the application being reduced, use the rule given in Table 1. When the head of the application contains another application, recursively reduce the head. If the result of such a reduction is a combinator, use the rule given in Table 1. Otherwise, recursively reduce the head again. The reduction of the application is complete when no rules apply.

When the reduction of an application leads to a request to reduce its function part, a stack is often used to store the original application. CURRY is unique in storing the stack in the cells that are on the stack. Motivated by [SW67], the idea is an extension of the method used to evaluate strict functions in [SCN84].

Normal order evaluation is implemented by maintaining two pointers, *fun* and *stack*. *Fun* points to the function being evaluated, and *stack* points to the stack which contains the arguments of the function. The stack is a linked list of cells in which the function part contains a pointer to the rest of the stack. When *fun* points to a combinator, the arguments to the combinator are obtained from *stack* and the combinator's rule is applied, resulting in new values for *fun* and *stack*. Functions are often popped from the stack at this time. When *fun* points to an application, the function part is put into *fun*, *stack* is put into the function part, and *stack* is made to point to the application. As a result, the application is pushed onto the stack. The application's function cell is used to hold the stack. See Figure 4 for a step-by-step display of the reduction of $(SKIJ) \Rightarrow J$.

[1]More precisely, head normal order evaluation.

2.3 Input and Output

Reduction of the function (RI) initiates a request for an input bit. That function is replaced with $(PK(RI))$ when the input is low, and $(PJ(RI))$ when the input is high.

Output is obtained from a stream. A stream S is recursively defined to be a function of the form $(P\beta S)$ where β is K or J. The output is low when β is K and high when β is J.

A program for the CURRY chip consists of constants combined by function application. A program is further restricted to those functions having a signature of a map from a stream to a stream. The program is applied to (RI) to obtain the stream for output. The top level loop for the CURRY chip involves reducing the first element of the stream to K or J, then replacing the stream by the next stream and looping, as shown is Figure 5. Thus, the entire evaluation process is driven by requests for output.

```
stream ← (fun (R I));
loop
    print (eval (head stream));
    stream ← (tail stream);
repeat.
```

Figure 5: CURRY Top Level Loop

3 Hardware Support for Reduction

3.1 Data Types

A function is represented as a 23-bit object with three fields: a 21-bit data field, a 1-bit atom field, and another 1-bit field used only by the garbage collector. The atom field determines how the content of the data field is to be interpreted. When the atom field is false,

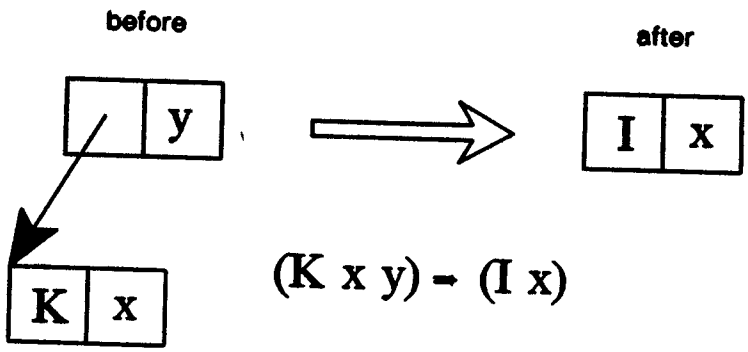

Figure 1: *K* Reduction

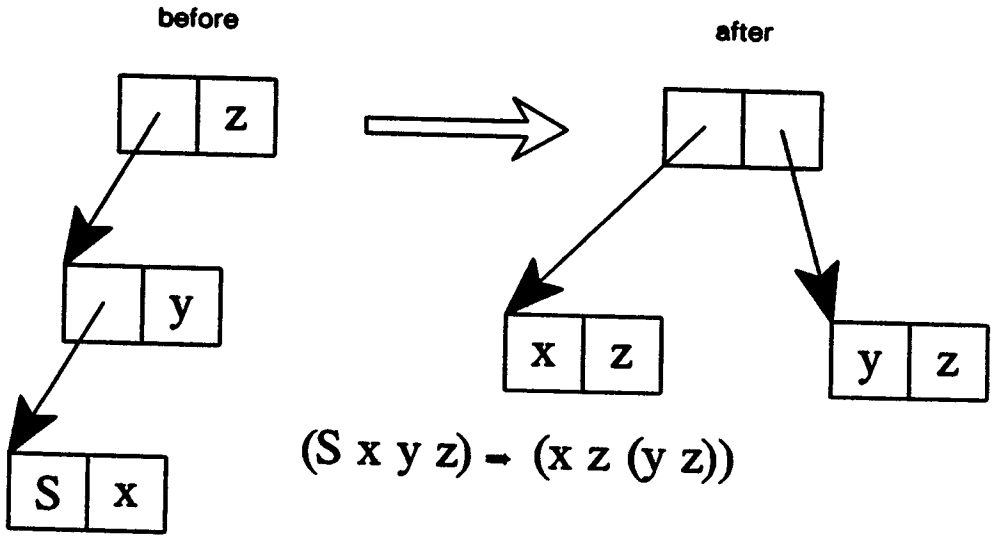

Figure 2: *S* Reduction

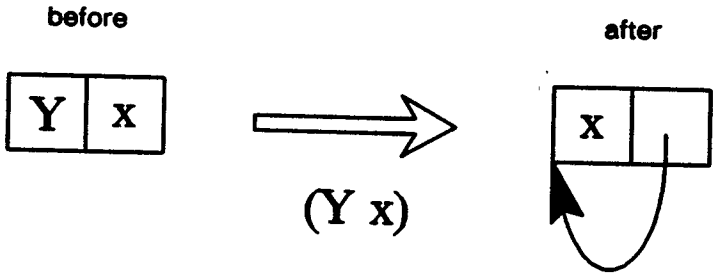

Figure 3: *Y* Reduction

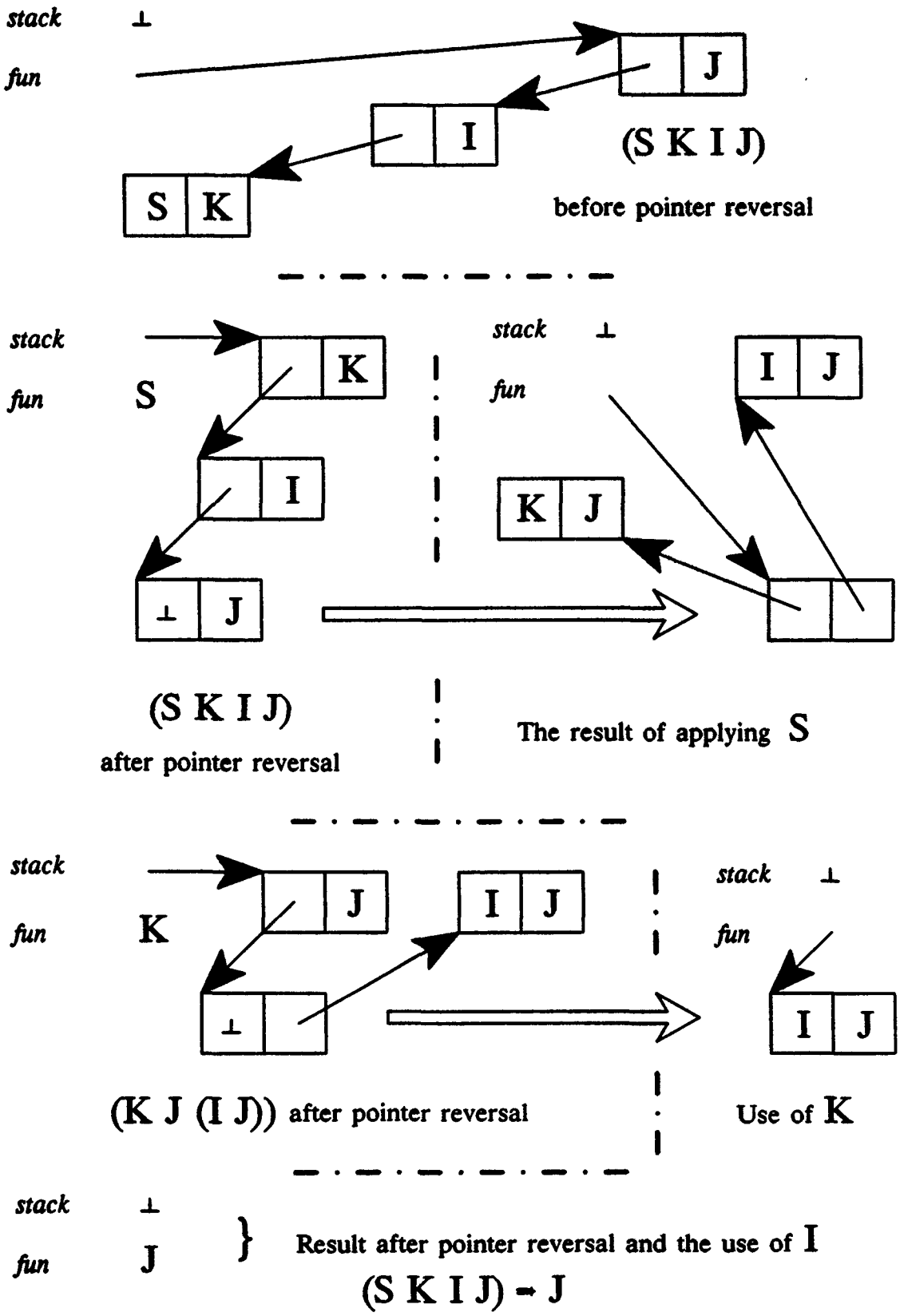

Figure 4: Reduction via Pointer Reversal

the function is an application, and the data is an address pointing to an application cell which contains a pair of functions as shown in Figure 6. When the atom field is true, the function is one of the combinators from Table 1, and the data field identifies which combinator. Alternatively, the function is the stack bottom symbol ($\perp$). The codes for all atoms known by the CURRY chip are given in Table 2. The atomic functions are grouped into three categories based on the number of arguments used during a reduction.

	21 bits	1 bit	1 bit	1 bit
HEAD	DATA	UNUSED	ATOM	GC
TAIL	DATA	UNUSED	ATOM	GC

Figure 6: An Application Cell

Atom	Binary Value
$\perp$	00000001
I	00010010
J	00100010
Y	01000010
R	10000010
K	00010100
T	00100100
S'	01000100
C'	10000100
S	00011000
B	00101000
C	01001000
P	10001000

Table 2: Representation of atoms

The choice of the sizes of the objects was motivated by the following two considerations: The CURRY chip has $22 + 2 *$ (address size) pins; 21-bit addresses allow the use of a 64 pin case. 21-bit addresses allow access to 2 million 6-byte application cells. The half million application cells used in the C implementation of CURRY has been adequate[Ram85].

Given byte addressable memory, all parts of applications can be addressed using 24-bit addresses with the restriction that the two low order bits must never be simultaneously on. This amounts to reducing the effective address space by 25 percent and is caused by the interest in 6 byte objects rather than 8 byte objects.

Notice that only 23 bits of the 24-bit data words are used. A possible use of the remaining bit is for one-bit reference counts. One-bit reference counts offer a simple method for reducing the amount of garbage generated by evaluation [SCN84]. Cells that are referenced by only one pointer are marked with a flag. When that pointer is discarded, the cell can be safely returned to free storage. When another pointer points to the cell, the cell must be unmarked and can only be returned to free storage using the usual garbage collector. Unfortunately, the inclusion of one-bit reference counts would have made the size of the CURRY chip too large. Hardware memory that keeps track of reference counts[Wis85] is another attractive option for reducing garbage collection costs as long the CURRY chip is not slowed down by that memory.

Another possible use of the extra bit in a data word is to differentiate between two types of pointers. In addition to the existing type of pointer, one would add a list pointer type. A cell pointed to by a list pointer would be defined to have the same meaning as a pointer to the pair of cells used to represent $(P\mathcal{E}\mathcal{F})$. List structure would then be represented using half the number of cells used before. The extra bit could also be used to differentiate between two types of atomic data, allowing the addition of an integer data type. As before, adding more data types would have made the size of the CURRY chip too large.

3.2 Computing with the CURRY Chip

The CURRY chip is one of four units required to make a computing system. The other units consist of memory, a garbage collector, and a controller used to link the system to the outside world. The four units are tied together with a 24-bit address bus and a 24-bit data bus. There is nothing special about the memory except for the fact that 24-bit addresses with the two low order bits on, are invalid. The garbage collector is a chip fabricated in the same manner as the CURRY chip and uses the algorithm given in [HSSB80] for an implementation of SCHEME[Cli85] on a chip[SS80]. The marking algorithm used is described in [SW67]. Its implementation is straightforward and described in [Ram86].

The computing system has two modes, and the controller has a different task in both modes. When the system is in *RUN* mode, the CURRY chip and the garbage collector work to evaluate functions in memory, and the controller mediates the input and output of the CURRY chip. When the system is in *BOOT* mode, the CURRY chip and the garbage collector are disabled, and the controller can be directed to load a memory image into the system or copy the existing memory image out. Figure 7 gives a block diagram showing how the four units are connected to make a computing system.

There is no bus contention in this design. When the system is in *BOOT* mode, only the controller

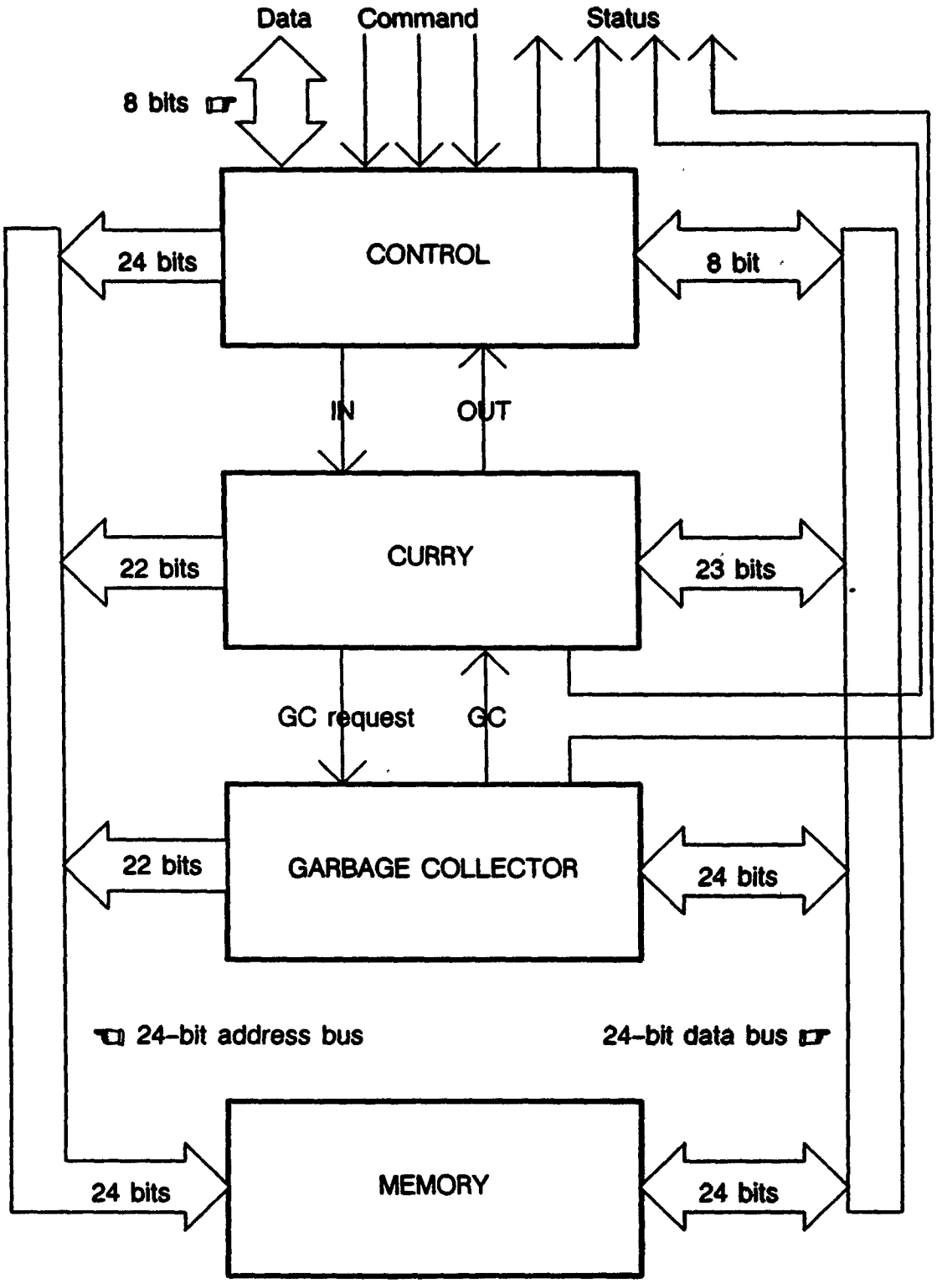

Figure 7: A Complete Computing System

can use the bus. When the system is in *RUN* mode, a simple protocol between the garbage collector and the CURRY chip decides which will use the bus.

The interface to the outside world is via an 8-bit bidirectional port, and some command and status signals. The controller accesses memory in one-byte units and thus requires 24-bit addressing. Since the garbage collector and the CURRY chip access memory in three-byte units, 22-bit addressing suffices.

The CURRY chip and the garbage collector chip were synthesized using a silicon compiler called MetaSyn[2] [Sou83]. The CURRY chip has about nine thousand transistors, consumes 1.11 watts and is 5.6 mm by 7.5 mm when 3 micron nMOS technology is used. The timing predictor conservatively estimated a clocking frequency of 0.4 Mhz. The garbage collector chip is 5.25 mm by 6.24 mm, consumes 0.91 watts, and clocks at a predicted frequency of 3.2 Mhz. It was also implemented in 3 micron nMOS. Both chips have been received from fabrication, and more than a majority of both chips have passed low speed simulation tests. High speed tests of the chips have yet to be completed, but we estimate a speed of at least 20 thousand reductions of applications per second (RAPS) at 0.4 Mhz. Subsequent work has demonstrated a chip design capable of running at 1.2 Mhz, corresponding to 60 thousand RAPS. A detailed algorithmic description of the CURRY chip is in [Ram86].

4 Software

Two sizable programs have been written for the CURRY chip. CCP is a compiler that translates abstractions into combinators. LCP is a loader that converts textual representations of combinators into running programs.

A simulator of the CURRY chip has been coded in C[KR78], and was used to test the above programs. The simulator performed 18 thousand RAPS on a VAX-11/780 as measured by CPU time, while providing enough memory for half a million application cells. The times used to calculate this speed include the time spent in pointer reversal and garbage collection. When run on machines using a virtual memory operating system, the garbage collector described in [Cla76] proved superior compared with [SW67].

[2]MetaLogic, MacPitts, and MetaSyn are trademarks of Metalogic, Inc.

1. $\lambda x.\mathcal{E}\mathcal{F} \Rightarrow (S(\lambda x.\mathcal{E})\lambda x.\mathcal{F})$.
2. $\lambda x.x \Rightarrow I$.
3. $\lambda x.y \Rightarrow (Ky)$; variable $y \neq x$.
4. $(I\mathcal{E}) \Rightarrow \mathcal{E}$.
5. $(J\mathcal{E}) \Rightarrow I$.
6. $(K\mathcal{E}\mathcal{F}) \Rightarrow \mathcal{E}$.
7. $(T\mathcal{E}\mathcal{F}) \Rightarrow (\mathcal{F}\mathcal{E})$.
8. $(S(K\mathcal{E})) \Rightarrow (B\mathcal{E})$.
9. $(S\mathcal{E}(K\mathcal{F})) \Rightarrow (C\mathcal{E}\mathcal{F})$.
10. $(S(B\mathcal{E}\mathcal{F})\mathcal{G}) \Rightarrow (S'\mathcal{E}\mathcal{F}\mathcal{G})$.
11. $(BI) \Rightarrow I$.
12. $(BCT) \Rightarrow P$.
13. $(B\mathcal{E}I) \Rightarrow \mathcal{E}$.
14. $(B\mathcal{E}(K\mathcal{F})) \Rightarrow (K(\mathcal{E}\mathcal{F}))$.
15. $(C(B\mathcal{E}\mathcal{F})) \Rightarrow (C'\mathcal{E}\mathcal{F})$.

Table 3: Compilation Rules

4.1 Compilation

Programs for the CURRY chip were written in a language called CHURCH. A C program translates a CHURCH source into abstractions. The program CCP translates the abstractions into combinators. Since CCP was written in the language CHURCH, its performance, as estimated using the simulator, became an interesting object of study.

The abstraction compilation algorithm simply implements the rules in Table 3. Earlier rules take precedence over later rules so there is no ambiguity in the algorithm. Rules 1–3 are due to Schönfinkel[Sch24], the first to show that variables could be eliminated from abstractions. Rules 1–3, 8, 9, 13 and 14 comprise one formulation of Curry's algorithm [CF58, page 190]. Turner further refined the compilation algorithm by adding rules 10 and 15 [Tur79a]. This algorithm produces expressions that are related to the size of the input by a polynomial, whereas the previous algorithms were exponential. Rules 11 and 13 implement η-conversion, which is characterized by the rule $\lambda x.\mathcal{E}x \Rightarrow \mathcal{E}$ when x is not in $\mathcal{E}$ [Bar84, page 160]. This rule is valid when all objects are functions as is the case in CHURCH. Rule 11 was shown to be important in one experiment. Removal of the rule resulted in functions that were 30% larger. The remaining rules 4–7 and 12[3], implement some simple conversions that reduce the size of the final function.

CCP starts with an environment in which a symbol is bound to Y.

[3]Rule 12 was not always used in the reported experiments.

4.2 Programming Methodology

As mentioned above, two nontrivial programs have been written for the CURRY chip. CCP, the program that translates abstractions into combinators, is 750 lines of code, the loader is 400 lines of code. Many of the usual techniques for writing programs using combinators were applied, such as the identification of K with true, J with false and P with pair [Bar84, pages 132–135]. Two programming practices proved useful during the development of the programs. The first practice allows the input stream to determine how program units are composed, while the second practice allows separate compilation.

4.2.1 Routines

While the overall signature of a CHURCH program is a map from a stream to a stream, it was found useful to divide the program into units called routines. The signature of a routine is a map from some optional input values, a stream and a continuation, giving a stream. Henceforth, a finite part of a stream will be called a string. The output associated with the routine consists of concatenating the string produced by the routine to the stream constructed by applying the continuation to values computed by the routine, continuations being used as described in [Sto77].

CCP contains many examples of routines. The `printer` routine takes a value to be printed, a stream, and continuation. It concatenates a string, representing the value, to the result of applying the continuation to the stream. The `get_token` routine takes only a stream and a continuation. It produces no string; instead its value is the result of applying the continuation to both the token it produces and to the new input stream. The `printer` is an example of a routine that does not compute a value. Hence, its continuation is applied only to a stream. In contrast, `get_token` is an example of a routine that computes one value. Hence, its continuation is applied to that value as well as a stream.

Routines can be used in the same manner as subroutines in other programming languages by following the convention that the continuation represents a return address. Routines can also be used to implement more general control structures[Wad85], since continuations are not restricted to be functions defined in the same section of a program that invokes the routine.

4.2.2 Modules

Given the size of the programs CCP and LCP, it became necessary to break these programs into units that could be compiled separately. Each compilation unit is a function represented by constants combined by function application. Compilation units are combined into a loadable function by applying to the last unit, the result of making the proceeding units into a loadable function. The first compilation unit is responsible for combining the remaining units into a running program. One useful method for combining the units involves modules.

Modules are used to share common definitions. Modules are functions that export their definitions to other functions. Definitions within a module are imported to a function by applying the module to the function. An example of a module that exports $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$ is $\lambda f.f\mathcal{E}\mathcal{F}\mathcal{G}$. A function that binds the three imports to a, b and c is $\lambda abc.\mathcal{H}$.

4.3 Super Combinators

Super combinators were introduced in [Hug82] as a means of speeding up combinator reduction. Instead of using a fixed set of combinators in the representation of a function, the compiler carefully chooses the set of combinators used for each function. These super combinators promote sharing of common subexpressions and lessen the total number of reductions.

On machines that use a fixed set of combinators, such as is the CURRY chip, one must further compile the super combinators into the machine's fixed set of combinators. One reduction of a super combinator is replaced by many reductions, but the total number of reductions could be lessened due to the offsetting influence of increased sharing of common subexpressions.

A super combinator version of CCP was created by applying Hughes' algorithm to produce a super combinator version of the abstractions. These were compiled into combinators using the old version of CCP, noting the number of reductions required to perform the translation. The super combinator version of CCP was loaded, and also used to compile the super combinator version of the abstractions. As expected, the super combinator version required less storage (2.5%) in the combinator machine. The super combinator version of CCP required 11% more reductions to translate a compilation unit independent of the size of the unit being compiled. Super combinators appear not to be helpful for CCP on the CURRY chip.

5 Conclusion

A combinator programming system that has been implemented in VLSI, has been shown to support the

writing of two sizable programs for the hardware. A novel feature of the CURRY chip is an evaluation method that requires no additional storage for an evaluation stack. Future planned topics of research include the implementation of this system on a parallel processor[BCHP86], while adjoining a nondeterministic combinator[ODo85] to the existing fixed set, which will allow the specification of interrupt driven programs.

[Note added in proof: Kevin Greene[Gre85] reports that D. A. Turner, A. Norman, and M. Scheevel have also independently discovered the pointer reversal evaluation scheme.]

6 Acknowledgement

I am grateful to Leonard Monk and Thom Brando for their comments on an earlier draft. Keith Gerhardt, Joel Harris and John Sawyer were helpful sources of information on MITRE VLSI CAD tools. John Kemeny, Jeff Siskind and Jay Southard made the use of MetaSyn possible. DARPA funded Metal Oxide Semiconductor Implementation System (MOSIS) fabricated the chips.

References

[Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[BCHP86] T. J. Brando, H. E. T. Connell, J. D. Harris, and M. J. Prelle. A massively parallel artificial intelligence processor. In *Proceedings of the Phoenix Conference on Computers and Communications*, pages 638–645, IEEE, Scottsdale, AZ, March 1986.

[CF58] Haskell Curry and Robert Feys. *Combinatory Logic.* Volume 1, North-Holland, Amsterdam, 1958.

[Cla76] Douglas W. Clark. An efficient list-moving algorithm using constant workspace. *Communications of the ACM*, 19(6):352–354, June 1976.

[Cli85] William Clinger ed. *The Revised Revised Report on Scheme or An UnCommon Lisp.* AI-Memo 848, MIT, Cambridge, MA, August 1985.

[Gre85] Kevin J. Greene. *A Fully Lazy Higher Order Purely Functional Programming Language with Reduction Semantics.* CASE 8505, Syracuse University, Syracuse, NY, December 1985.

[HSSB80] Jack Holloway, Guy Lewis Steele Jr., Gerald Jay Sussman, and Alan Bell. *The SCHEME-79 Chip.* AI-Memo 559, MIT, Cambridge, MA, January 1980.

[Hug82] R. J. M. Hughes. Super combinators: a new implementation method for applicative languages. In *1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, Pittsburg, PA, August 1982.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[ODo85] Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, MA, 1985. Chapter 19.

[Ram85] John D. Ramsdell. *Combinator Programming.* Technical Report M85–43, MITRE Corp., Bedford, MA, September 1985.

[Ram86] John D. Ramsdell. *The CURRY Chip.* Technical Report M86–23, MITRE Corp., Bedford, MA, April 1986.

[Sch24] M. Schönfinkel. Uber die bausteine der mathematischen logik. *Math. Annalen*, 92(305), 1924.

[SCN84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *1984 ACM Symposium on LISP and Functional Programming*, pages 159–166, Austin, TX, August 1984.

[Sou83] Jay R. Southard. MacPitts: an approach to silicon compilation. *Computer Magazine*, 16(12):74–82, December 1983.

[SS80] Guy Lewis Steele Jr. and Gerald Jay Sussman. Design of a LISP-based microprocessor. *Communications of the ACM*, 23(11):628–645, November 1980.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Tur79a] D. A. Turner. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 44(2):267–270, June 1979.

[Tur79b] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9:31–49, 1979.

[Wad85] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, pages 113–127, Springer-Verlag, Berlin, 1985.

[Wis85] D. S. Wise. Design for a multiprocessing heap with on-board reference counting. In *Functional Programming Languages and Computer Architecture*, pages 289–303, Springer-Verlag, Berlin, 1985.