A Parallel Intermediate Representation based on Lambda Expressions

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
budd@cs.orst.edu

July 10, 1998

Abstract

The lambda calculus has frequently been used as an intermediate representation for programming languages, particularly for functional programming language systems. We introduce two simple extensions to the lambda calculus that describe potentially parallel computations. These extensions permit us to use the lambda calculus as an intermediate form for languages that operate on large data items as single entities, such as FP or APL. We conclude by discussing how this intermediate representation can facilitate the generation of code for different types of parallel systems.

The Von Neumann Bottleneck

In his 1977 Turing Award lecture [Bac78] John Backus argued that traditional languages suffered from the "von Neumann bottleneck." This bottleneck is caused by processors which access words of memory one at a time. This results in programming languages that treat data in small units, to be processed individually. This in turn finally causes programmers to spend an inordinate amount of time dealing with the breaking up of large data items into small pieces, and the gathering together of small pieces of data into larger conceptual units. If languages dealt with data in larger conceptual units, Backus argued, not only would the programming task be simplified, but architects might be encouraged to develop machine organizations which overcome the basic word-at-a-time limitation of the conventional von Neumann design.

Backus proposed a new language, FP, which dealt with lists as large data items. Other languages, such as APL and LISP, have also been noted for dealing with large data items as a single unit, rather than one element at a time. It is certainly true that programs in these languages are often surprizingly succinct; for example Backus' two line definition of matrix multiplication, or the infamous APL one-liners [PeR79]. Some would argue, however, that brevity is not always synonymous with clarity. Regardless of how one stands on this issue, it is intuitive that a language, such as APL, that deals with operations on large data items implicitly in parallel should somehow be more easily amenable to parallel processing than, say, a language such as FORTRAN where parallelism is much less obvious in the original source code. While the intuition is clear, the practical matter of how one goes about making effective use of this parallelism is much more difficult.

Our research has been concerned with the development of compiler techniques that can be applied to languages, such as FP and APL, in which the basic elements of computation are large structures. Our approach is in many ways conventional. The compiler is divided into a number is separate stages or passes; a parser which translates the source code into an intermediate representation, various transformations on the intermediate representation, and a code generator which produces the final object code from the intermediate representation. In this paper we wish to concentrate on describing our intermediate representation, with a few brief words near the end showing how this intermediate representation simplifies the task of producing code for various types of parallel machines.

Lambda notation

The λ calculus has been studied extensively since the 1930s as a simple and elegant model of the notion of a function [Bar84]. A lambda expression is written as the symbol λ, one or more identifier (argument) names, and an expression:

λ x, y, z . *expresison*

It is important to note that the λ expression is a value, and can be treated as such (passed as an argument to another function for example). Various transformations can be performed on λ expressions, such as converting a λ expression of multiple arguments into a series of λ expressions each of one argument (called *curring*), applying a λ to values (so called β transformations), and so on. We will not describe these here; readers unfamiliar with λ notation are referred to any number of sources, such as [Pey86] or [Bar84].

Recently, λ notation has been adopted as a useful abstraction technique, and intermediate representation, for a number of functional languages [Pey86, Dil88]. Used in this manner various extensions to the bare λ forms are often

employed. Examples of such extensions include single assignment variables and conditionals. Usually these extensions can be defined in the basic λ calculus, although this is often only of theoretical interest, since the definition may introduce unnecessary duplication or inefficiencies. This extended lambda calculus is then used as part of the internal language for describing computation.

Before describing our own extensions to the λ calculus, we note that the traditional formulation suffers from the same "von Neumann" bottleneck as conventional languages. A λ abstraction describes a function, but it is a function in which the user feeds in *one* value and obtains *one* result. If multiple values are desired, multiple invocations of the λ function must be performed. To overcome this, and to permit us to describe the computation of large data items in a convenient formal framework, we introduce two extensions to the λ calculus.

Collection (κ) forms

Our first extension is called a collection, or κ form. The collection form describes an entire group of values. It is formed by the symbol κ followed by two fields.

κ *size* (λ x . *expression*)

The first field is the *size* field, and describes the size of the object being described. The structure of the size field is different depending upon the language being compiled; in FP it can be simply a single scalar value representing the number of elements in a list, while in APL it is a pair representing a rank (number of dimensions) and a shape (extent along each dimension) for an arbitrary size array. In whatever form it takes, the expression can either be determined as a constant at compile time, or it may be maintained as a symbolic expression to be evaluated and determined at run time.

The second portion of the collection form is a λ expression. This expression, when evaluated with integer argument i, where i is between 0 and the number of elements of the item, yields the i^{th} element of the resulting structure.

As a simple example, consider the characterization of the APL expression $\iota 200$. The APL function ι takes a scalar argument and yields a vector where the elements run from 1 upto the argument value. Thus the result of this expression would be a vector containing 200 elements. To obtain any particular element, it is sufficient to add one (since ι is one-based, and the κ form zero based) to the argument value. This is described as follows:

κ (1 (200)) (λ p . p + 1)

The κ form is a single value, and can be manipulated as a value in much the same manner as a λ form. Just as there are transformations that can be applied to λ forms, we have defined various transformations on κ forms. These

include extracting the size and/or λ portion, and computing the result of the λ portion when given a specific (perhaps symbolic) argument.

It is also important to note that the κ form does not imply any specific ordering on the evaluation of each element of the result, or indeed that each element will ever be evaluated. This fact will become important when we consider the generation of parallel code for expressions containing a κ form.

Reduction (σ) forms

Our second extension to the λ calculus attempts to capture the common notion of a reduction of a vector to a single scalar quantity. The reduction form consists of the symbol σ followed by three fields. The first field is the name of a built-in function, such as addition ($+$), multiplication ($\times$) or the like. The second field is an expression (perhaps symbolic) representing a scalar value indicating the size of the vector being reduced. The third field is once more a lambda expression, indicating the value to be used in the reduction.

σ *fun limit* . (λ x . *expression*)

The scalar value of a σ expression is determined by evaluating the nested λ expression on the values 0 to one less than the limit, and combining the results using the given function. It is important to note that for communtative and associatative functions no ordering on the evaluations is implied. This fact will be of importance when we describe our code generation strategy. The next section will give an example illustrating the use of a σ form.

While the σ form always produces a scalar value, this does not preclude the description of larger values being generated by a reduction, for example a two dimensional vector being reduced to a vector. Such objects can be described as a combination of κ and σ forms, as shown in the example in the next section.

An APL Compiler

Space does not permit a detailed description of the APL compiler we are constructing based around out intermediate representation; interested readers are referred to [Bud88b] and [Bud88c]. The general technique employed, however, is as follows.

- First we have defined characterizations, in terms of our extended λ calculus, for each of the many APL operations. Each of these characterizations is represented as a λ function which takes a κ form as input and yields a κ form as result.
- A parser then translates an APL expression into a concatenation of these characterizations, yielding a single large expression in our extended λ

language. (Typically each statement of APL produces a single large κ form).

- Transformation rules, chiefly β conversion on λ forms, are then applied to simplify this expression. Any computations and simplifications that can be applied at compile time are performed.
- The simplified form of the statement is then given to a code generator, producing the final object code.

While we have chosen to implement an APL system for our initial investigations, the techniques we use would be similar in any language that manipulated large data objects as single entities, such as FP (see [Bud88b]).

For the remainder of the paper we will restrict ourselves to considering only code generation for assignment statements, as this is the principle action in APL. Furthermore we will ignore details such as memory management and the storage of shape information, and consider only code that computes the value of an expression.

A typical APL assignment statement is the following, which computes a vector of length 200 containing one values in positions corresponding to primes and zero everywhere else. It accomplishes this by computing, for each value, the set of divisors for the value. If the number of even divisors is equal to two, then the value is considered to be prime. (Readers unfamiliar with APL may wish to consult a description of the language, such as [PoP75]).

$$\text{primes} \leftarrow 2 = +\not{.}\, 0 = (\iota N) \circ .| \, \iota N$$

A detailed description of the compilation process for this expression is presented in [Bud88c]. For out purposes it is sufficient to note that this is eventually transformed into the following expression.

```
primes ← κ (1, (200)) . (λ p .
                2 = σ +, 200 .
                      (λ q . 0 = (q+1) mod (p+1)))
```

The form of this intermediate representation is typical, consisting of an outermost $\kappa - \lambda$ pair surrounding zero or more $\sigma - \lambda$ or $\kappa - \lambda$ pairs. The major task during code generation is the translation of these forms into more conventional code sequences.

Code Generation

In this section we will discuss the issues involved in generating code from our extended λ expressions. Note that our basic underlying computational model corresponds to a conventional machine, perhaps augmented with parallel instructions, and thus we have little in common with systems that generate code from λ expressions using graph reduction [Pey86, Dil88].

Scalar Processors

When generating code for a conventional scalar processor, both κ and σ forms become loops. A κ form loops around a subscripted assignment generating each of the individual elements, where the variable being assigned is a temporary or, when a κ is combined with an assignment statement, the variable being assigned.

```
for i := 0 to 199 do begin
    ...
    primes[i] := ...;
end
```

A σ form becomes a loop modifying a reduction variable.

```
redvar := 0;
for i := 0 to 199 do begin
    ...
    redvar := redvar + ...;
end
```

The code is similar to that produced by other techniques [Bud88a].

Vector Machines

In the absence of conditional instructions or nested λ forms, each argument in a λ expression is acted upon in the same manner. On machines that possess vector instructions, if we can make a vector consisting of all the potential input values (that is, a vector running from zero up to the limit of the surrounding σ or κ form), then we can produce all values in parallel. This input vector can be either generated or copied from a static constant location. For example, the values generated by a innermost λ in the example of the last section might all be placed into a temporary variable t in parallel by the following straight-line code.

$t(0{:}199) := \imath\ 200$; *all inputs*
$t(0{:}199) := t(0{:}199) + 1$;
$t(0{:}199) := t(0{:}199)$ **mod** (p+1);
$t(0{:}199) := (0 = t(0{:}199))$;

Conditional (**if**) forms can often be handled by generating both halves the the conditional and combining them with a mask.

SIMD Machines

Our model for a SIMD machine will be something like a Connection Machine, which has slightly more general capabilities than simple vector instructions [HiS86]. As with vector instructions, parallelism is utilized first in the innermost λ forms; with the surrounding forms either being incorporated into the parallel stream or generated as scalar code (see [Bud88c]).

On such a machine all elements of a κ form can be computed in parallel (assuming the number of elements of the array does not exceed the number of processors), since the target of each assignment is independent in each iteration.

```
for i := 0 to 199 in parallel do begin
    ...
    primes[i] := ...;
end
```

To perform a reduction a variation of Hillis' and Steels' logarithmic array summing algorithm can be employed. This produces the scalar result in time proportional to the log of the number of elements being reduced, leaving the answer in the first position of the array.

```
for j := 1 to ceil(log2 extent) do
    for all k to extent in parallel do
            if k mod 2^j = 0 then
              if k + 2^(j-1) < extent then
                 x[k] := x[k] op x[k + 2^(j-1)]
              fi
            fi
    od
od
```

MIMD Machines

We are considering only shared memory MIMD machines. On such systems, the goal is to divide a computation into a number of roughly equivalent parallel tasks each involving a nontrivial amount of computation. Thus, in contrast to the technique used for SIMD machines, we want to parallelise around the outermost κ form of an assignment statement, giving each processor the task of generating a section of the final result. Within each processor scalar code is used to generate the values for the result. If we let *numberProcessors* represent the number of processors, and *extent* the size of an outermost loop generated for a κ form, the high level pseudo-code we could generate for a κ form would be as follows:

```
for i := 1 to numberProcessors do begin
    fork off process doing the following
    for j := i * (extent/numberProcessors)
        to (i+1)*(extent/numberProcessors) - 1 do begin
        primes[j] := ...
wait for all processors to complete
```

We have not considered the issues involved in generating code for non-shared memory MIMD machines.

Conclusions

We have introduced two simple extensions to the λ formalism for describing functional computations. These extensions permit us to describe the manipulation of large data objects as if they were single values. Our extended λ formalism is at the heart of an experimental APL system we are constructing, although the techniques are also applicable to FP or other languages that manipulate large values as single entities. Having introduced our extensions, we have discussed how these constructs, by eliminating explicit control flow requirements from the description of computations, facilitate the generation of code for a variety of parallel processors.

It is interesting to note that the technique we have outlined works best on large and complex expressions. Thus the infamous "one-liner", considered almost an art form among supports of APL [PeR79], and strongly denounced by detractors of the language [Dij72], is shown to have a useful and practical benefit.

References

[Bac78] Backus, John, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, Vol 21(8): 613-641, (August 1978).

[Bar84] Barendregt, H.P., *The Lambda Calculus: its syntax and semantics* (2nd ed), North-Holland, 1984.

[Bud88a] Budd, Timothy A., *An APL Compiler*, Springer-Verlag, 1988.

[Bud88b] Budd, Timothy A., "Composition and Compilation in Functional Programming Languages", Technical Report 88-60-14, Computer Science Department, Oregon State University, June 1988.

[Bud88c] Budd, Timothy A., "A New Approach to Vector Code Generation for Applicative Languages", Technical Report 88-60-18, Computer Science Department, Oregon State University, August 1988.

[Dij72] Dijkstra, Edsger W., "The Humble Programmer", *Communications of the ACM*, Vol 15(10):859-866 (October 1972).

[Dil88] Diller, Antoni, *Compiling Functional Languages*, Wiley, New York, 1988.

[HiS86] Hillis, W. D. and Steele, G. L., "Data Parallel Algorithms", *Communications of the ACM*, Vol 29(12):1170-1183 (December 1986).

[Pey86] Peyton Jones, Simon L., *The Implementation of Functional Programming Languages*, Prentice-Hall, 1986.

[PeR79] Perlis, Alan J. and Rugaber, Spencer, "Programming with Idioms in APL", APL Quote Quad, Vol 9(4):232-235 (June 1979).

[PoP75] Polivka, R. and Pakin, S., *APL: The Language and Its Usage*, Prentice Hall, 1975.