Composition and Compilation in Functional Programming Languages

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

July 10, 1998

Abstract

Functional programming languages, such as Backus' FP, and high level expression oriented languages, such as APL, are examples of programming languages in which the primary method of program construction is the process of composition. In this paper we describe an approach to generating code for languages based on compositions. The approach involves finding an intermediate representation which grows in size very slowly as additional terms are composed. In particular, the size of the intermediate representation of a composed object should be considerably smaller, and easier to interpret, than the sum of the sizes of the internal representations of the individual elements. We illustrate this technique by showing how to generate conventional code for Backus' language FP. The general technique, however, is applicable to other languages, as well as other architectures.

1 Introduction

The purposes of this paper are two-fold. First, we want to describe an approach to compilation and code generation that is applicable to any language for which the primary method of program construction is composition. Examples of such languages include functional programming languages, such as Backus' FP [Bac78], and high level expression languages, such as APL [Ive80]. In the former case the objects being composed are system and user defined functions, while in the latter the objects being composed are expressions built out of a rich set of system provided operations.

Having described our general approach, our second task will then be to illustrate the technique by showing how it is possible to produce conventional imperative style code from functions written in Backus' FP language. It is important to note, however, that while we choose to illustrate the technique with this language, the general approach may be applicable to other languages and architectures.

2 Compositions and Intermediate Representations

The approach to compilation described in this paper hinges on finding an intermediate representation with certain properties. We will cite three such characteristics. The first two are simply properties that any intermediate representation should possess. Namely, it should be easy to translate from the source language (in our case, FP) into the intermediate representation, and it should be easy to translate from the intermediate representation into the target language (say, machine code for a conventional von Neumann style processor).

In our example, the intermediate form representations of system provided functions will simply be given as part of the definition of those functions, and that of user defined functions will be stored as part of the representation of those functions. As to the latter property, we will argue in the sequel (section 5) that it is relatively easy to take our representation and from it produce code for a conventional processor.

The third and most important property we would like our representation to possess is particular to languages based on compositions. This is that the size of composed objects should grow very slowly as the number of compositions increases. In particular, the size of an object formed from compositions should be significantly smaller than the sum of the sizes of the representations of the individual components. It may at first seem surprising that any representation should satisfy this property, however an examination of compositions from other problem domains reveal many instances in which this has been achieved. Probably the most common example is that of linear algebra. In linear algebra transformations are represented by matrices, and functional application by matrix multiplication. If one is interested in the final result of multiple transformations, such as F:G:x, one can construct this first by forming the composition, ($\mathsf{F}\circ\mathsf{G}$). This composition is also formed by matrix multiplication. The important properties are that this composition is independent of any values the transformation may be applied to (a property similar to the Church-Rosser [Bac78] property for functions), and the size of the composition is often no larger than the size of either of the individual components taken separately[1]. By repeated multiplications entire strings of compositions can be reduced to a single matrix.

In the area of programming languages, in [GuW75], Guibas and Wyatt demonstrated how to produce very efficient code for a subset of the APL functions. They did this by finding an intermediate representation, the *stepper*, which could represent each of the functions and which formed compositions in a manner similar to matrix algebra. This technique was extended in [BuT82], and eventually incorporated into a compiler for APL [Bud88].

In the next section we will describe our intermediate representation. Subsequent sections will give examples showing how compositions are reflected in the intermediate form. These examples will illustrate this third property. We will then argue that this intermediate representation can also be used as a basis for generating the final code.

[1]There are, of course, degenerate cases in which the composition is significantly larger than the pieces. We, however, refer here to more common cases.

3 Descriptors and Descriptor functions

FP is a language for performing transformations on lists. For our purposes it is necessary, therefore, to find a representation that can not only be used to describe lists, but can also be used to describe the lists which will result as a consequence of some functional application to an unspecified argument. We first describe our representation for lists, which we call a list *descriptor*. This is followed by our representation for functions which map descriptors to other descriptors.

3.1 Descriptors

The conventional representation for lists is that used historically by most Lisp systems [McC60]. Here lists are represented as a linked list of elements, which may be scalars or may themselves be lists. If we extend this representation, however, to include unevaluated (or lazy evaluated) functions, several difficulties present themselves. The functions in FP for the most part can be described as applying uniformly to each element of a list. If f is such a function, and we wish to describe the effect of f being applied to each element of the list (a, b, c), we can only do it by repeating the description of f, as in $((fa), (fb), (fc))$. An even more taxing problem is the difficulty of describing the effect of f applied to a list of unknown length.

In order to circumvent these problems, we will describe lists using a different, less conventional, notational device. This notation is less useful for describing lists of values, but will be very useful for describing transformations on lists. Our notation will represent a list as a pair. Since this is the only data structure we will use, we will simply use parenthesis to represent this pair. For non-scalars, the first element of this pair is an expression which, when evaluated, will yield the length of the list. The second element is a function of one integer argument, the range of legal values being the numbers between 1 and the size of the list (that is, the value of the first argument). When presented with a value i, the function will yield the description of the ith element of the list. We will use lambda notation [Bar84] to describe the functions.

We will use a value which cannot represent the size of a list, such as ⇔1, as a special indication for scalar values. In this case the second element will be an expression which, when evaluated, returns the scalar value. Thus the following are legal descriptors and have the meaning shown:

(⇔1, 4) the scalar value 4
(1, λ p . (⇔1, 4)) a list containing the scalar value 4
(3, λ p . (⇔1, (2*p) + 1)) a list containing the three values 3, 5 and 7

In order to clarify the accessing of the separate parts of a descriptor, we will use a notation similar to Pascal or C style field names. If x is a descriptor, then x.size will represent the expression yielding the size of the list represented by x, and x.item the function which returns the individual elements.

3.2 Function Descriptions

An FP function is a mapping between a list as input and a list as result. Internally then, we will describe each FP function, as well as the compositions we will subsequently construct, as a function taking a descriptor as input and yielding a descriptor as result. In almost all cases (the one exception being the constant functions) the result depends in some manner on the arguments. For example, the transpose function Trans is described by Backus [Bac78] as follows:

$$\text{Trans}{:}< x_1, ...x_n > \rightarrow < y_1, ..., y_m >,$$
$$\text{where } x_i = < x_{i1}, ..., x_{im} > \text{ and } y_j = < x_{1j}, ..., x_{nj} >, 1 \leq i \leq n, 1 \leq j \leq m$$

It is clear from this description that the input must be a list of lists, all of which must have the same number of elements. Insuring that this is the case is the task of *conformability checking.* Since including conformability checking code would only obscure the intent of the functions we present, we will ignore this issue. We state, however, that this omission is not inherent in the technique[2].

Our internal representation for the transpose is as follows:

Trans $\Leftrightarrow$ λ a . ((a.item 1).size, λ i . (a.size, λ j . ((a.item j).item i)))

Transpose is represented internally by a λ function which takes a list, represented by the descriptor a, then returns a descriptor representing the result. The size of the result list is the size of the first element of a. Each element of the result is a list, the size of which is the size of a, and for which the elements are found by indexing into a first with the argument j, then with the argument i.

Notice that the λ function representing transpose does not contain any free variables. Functions having this characteristic are known as combinators [Bar84]. The representations of all the basic FP functions which we will present will possess the combinator property. Since this is preserved under composition and function application, all the representations described in this paper will also be combinators.

4 Examples

Rather than presenting descriptor characterizations of all the FP forms at once, and then showing how we use these, we will instead proceed by analyzing three examples from Backus' paper [Bac78]. This organization allows us to motivate the introduction of new notation by describing first the problems to be solved, then the solution provided.

4.1 Example 1 - Inner Product

The first FP definition we will consider is the inner product function:

$$\text{Def IP} \equiv (/+) \circ (\alpha\times) \circ \text{Trans}$$

When applied to a pair of lists, such as $<< 1, 2, 3 >, < 6, 5, 4 >>$, the IP function computes the sum of the paired products, in this case 28. We have already noted the descriptor characterization of transpose (section 3.2), thus it remains to discuss the descriptor characterization of insertions, apply to all, and their composition.

[2]Using Backus' definitions, the only difference between an arithmetic function, such as +, and the insertion function formed from + is the lack of conformability checking in the latter. We will therefore use the insertion form in the examples we present.

4.1.1 Descriptor Characterization of Insertion

In order to simplify the presentation, we treat here only the important special case of insertion of a commutative function which possesses a left identity. The most common situations (and the only two which we will use in this paper) are insertion of addition (+) and of multiplication (×). In these two special cases the size of the result will always be a scalar, regardless of the input values (which must, in fact, be a list of scalars; however see our earlier remarks on conformability checking). Thus the first (size) field is always simply a constant $\Leftrightarrow 1$.

The second field presents us with more of a problem. Our intermediate representation is intended to make it easy to both translate from FP into the form and translate from the form into code for a conventional computer. Because of the latter requirement it is perhaps natural that our representation is not a pure functional language. Nevertheless, we wish to retain certain properties of the pure functional languages, the most important being the absence of free variables within functions (the so-called *combinator* property [Bar84]). In order to preserve this property, we introduce concepts such as variables and assignments, and loops, but in a very restricted form. Let us treat variables and assignments first. New variables can be created only within a certain type of expression, called the ν expression. The ν expression has the following form

$$\nu \text{ x . } \textit{statement list}$$

Only a single variable can be created in any ν expression; multiple variables require nested expressions. The statement list will be surrounded by a pair of parentheses to avoid ambiguity. The value of a ν expression is defined operationally; it is the value of the associated variable following execution of the given statement list. Assignment statements can appear in the statement list, but the only variables which can be assigned are those which appear in a surrounding ν expression. This implies, conversely, that if we wish to determine all the places where a ν variable is modified, we need consider only the statements in the statement list.

Statements in the statement list are separated by semicolons. There are three types of statements; expressions, assignment statements, and σ forms. With the exception of conditional expressions, which will be introduced in section 4.2, expressions are self explanatory. The preceding paragraph described the essential restrictions to assignment statements. The third statement, the σ statement, is our looping construct. It has the following form:

$$\sigma \text{ x, } \textit{limit} \text{ . } \textit{statement list}$$

The σ form specifies a loop running from 1 up to and including the value given by the expression *limit*. The statement list will be repeated this number of times, each time with the variable x being assigned a successive value. Note that this is a statement, executed for its side effects, and not an expression. There is no value associated with a σ statement. (Backus' while function requires a more general looping statement, which will not be described here).

We note that neither the variables defined in ν nor σ forms have any meaning outside the forms. Thus, as with λ functions, in order to avoid ambiguity we can perform α transformations [Pey86] which uniformly change the variable to another name, without affecting the meaning in any way. That is, a statement such as

$$\sigma \text{ i, 10 . j} \leftarrow \text{j} + \text{i}$$

Can be changed into

$$\sigma\ \mathrm{k},\ 10.\ \mathrm{j} \leftarrow \mathrm{j} + \mathrm{k}$$

with no change in meaning.

Having defined ν and σ forms, the descriptor function characterization of insertion can now be presented. We give here the characterization of $\times$ insertion; the characterization of $+$ insertion is similar.

$$/\times \Leftrightarrow \lambda\ \mathrm{a}\ .\ (\ \Leftrightarrow\!1,\ \nu\ \mathrm{r}\ .\ (\ \mathrm{r} \leftarrow 1\ ;\ \sigma\ \mathrm{i},\ \mathrm{a.size}\ .\ (\ \mathrm{r} \leftarrow \mathrm{r} * (\mathrm{a.item\ i}).\mathrm{item})))$$

In the item part the code creates a new variable r, then loops over the argument values computing their product. The result is the value of the ν expression, which is the product in the variable r.

4.1.2 Characterization of Apply to All

Apply to all is not a function in the sense of insertion, rather it is a higher order functional form. In our notation, apply to all takes as argument a descriptor function, and yields a descriptor function. It can be given as follows:

$$\alpha \Leftrightarrow \lambda\ \mathrm{f}\ .\ (\ \lambda\ \mathrm{a}\ .\ (\ \mathrm{a.size},\ \lambda\ \mathrm{p}\ .\ \mathrm{f}\ (\mathrm{a.item\ p})))$$

As one can see from this definition, the size of the result of an apply to all is the size of the argument, and the values are found by applying the function f to each element of the argument.

As an example of how the higher order functional form α yields a function, we show the derivation of the function which applies a multiplicative insertion to each of its members.

$$\begin{aligned}(\alpha/\times) \Leftrightarrow\ & (\lambda\ \mathrm{f}\ .\ (\ \lambda\ \mathrm{a}\ .\ (\ \mathrm{a.size},\ \lambda\ \mathrm{p}\ .\ \mathrm{f}\ (\mathrm{a.item\ p}))))\\ & \quad (\lambda\ \mathrm{a}\ .\ (\ \Leftrightarrow\!1,\ \nu\ \mathrm{r}\ .\ (\ \mathrm{r} \leftarrow 1\ ;\ \sigma\ \mathrm{i},\ \mathrm{a.size}\ .\ (\ \mathrm{r} \leftarrow \mathrm{r} * (\mathrm{a.item\ i}).\mathrm{item}))))\end{aligned}$$

We first apply an α transform to the argument (second line), changing the variable a into b. We then perform a β transform, substituting the argument for f. This yields:

$$\begin{aligned}&\lambda\ \mathrm{a}\ .\ (\ \mathrm{a.size},\ \lambda\ \mathrm{p}\ .\\ &\quad (\lambda\ \mathrm{b}\ .\ (\ \Leftrightarrow\!1,\ \nu\ \mathrm{r}\ .\ (\ \mathrm{r} \leftarrow 1\ ;\ \sigma\ \mathrm{i},\ \mathrm{b.size}\ .\ (\ \mathrm{r} \leftarrow \mathrm{r} * (\mathrm{b.item\ i}).\mathrm{item})))\\ &\qquad (\mathrm{a.item\ p})))\end{aligned}$$

A further β transform replaces b with (a.item p), yielding our final form:

$$\begin{aligned}&\lambda\ \mathrm{a}\ .\ (\ \mathrm{a.size},\\ &\quad \lambda\ \mathrm{p}\ .\ (\ \Leftrightarrow\!1,\ \nu\ \mathrm{r}\ .\ (\ \mathrm{r} \leftarrow 1\ ;\ \sigma\ \mathrm{i},\ (\mathrm{a.item\ p}).\mathrm{size}\ .\ (\ \mathrm{r} \leftarrow \mathrm{r} * ((\mathrm{a.item\ p}).\mathrm{item\ i}).\mathrm{item})))))\end{aligned}$$

Note the duplicated appearance of the expression (a.item p). Unlike other approaches, such as graph rewriting [Pey86], our technique makes no attempt to recognize or exploit common expressions during composition. Avoiding doing so is useful, since often these expressions will have different subsequent uses in later compositions, any may be vastly transformed or vanish altogether in the final representation. Those common subexpressions that remain can be easily detected by conventional means during final code generation (section 5).

4.1.3 Continuation of Example

Having presented the characterizations of the constituent parts, we now return to the question of characterizing the inner product function, which is defined by the composition $(/+) \circ (\alpha\ \times) \circ \mathsf{Trans}$. The first question which comes up is in what order the composition should be performed; left to right or right to left. In particular, one would like to assert a Church-Rosser like property [Pey86] to the effect that the order has no effect on the result. It is beyond the scope of this paper to show that this does indeed hold; we merely assert it here and carry on accordingly.

Leaving out the details of the derivation, we assert that the composition of $(\alpha/\times) \circ \mathsf{Trans}$ yields the following:

$$\begin{array}{l} \lambda \text{ a . ((a.item 1).size, } \lambda \text{ p,} \\ \quad \text{(-1, } \nu \text{ r . (r} \leftarrow 1 \text{ ; } \sigma \text{ k, a.size . r} \leftarrow \text{r} * \text{((a.item k).item p).item)))} \end{array}$$

The next step is then to compose $(/+)$ with this function. Doing so yields the final characterization of the IP function:

$$\begin{array}{l} \mathsf{IP} \Leftrightarrow \lambda \text{ a . (-1 , } \nu \text{ s . (s} \leftarrow 0 \text{ ;} \\ \qquad\qquad\qquad\qquad \sigma \text{ i, ((a.item 1).size). s} \leftarrow \text{s} + \nu \text{ r . (r} \leftarrow 1; \\ \qquad\qquad\qquad\qquad\quad \sigma \text{ k, a.size. r} \leftarrow \text{r} * \text{((a.item k).item i).item)))} \end{array}$$

4.2 Example 2 - Cross Product

The second example we will look at appears as part of a matrix multiplication function in Backus' paper. It computes a form of cross product. It can be defined as follows:

$$\text{Def } \mathsf{CP} \equiv (\alpha\mathsf{Distl}) \circ \mathsf{Distr}$$

For example, given the input $<< 1,2,3 >, < 4,5 >>$, the function CP computes $<< 1,4 >, < 1,5 >>, << 2,4 >, < 2,5 >>, << 3,4 >, < 3,5 >>$. In order to produce the descriptor function characterization of Distr, we need to introduce a conditional expression, similar to the cond expression in Lisp. We write the conditional as a generalized **if** statement, similar to Dijkstra's guarded command **if** statement [Dij76]. The syntax is the keyword **if**, followed by a sequence of one or more condition, action pairs. A right arrow ($\rightarrow$) separates the conditions from the actions, and a box ($\Box$) separates pairs. The conditions must always be disjoint, and one of the conditions must always be satisfied. In order to help ensure this, the condition on the final pair can be omitted with the understanding that it represents all other conditions not previously covered. The value of the **if** statement is the value associated with the condition evaluating true.

Using the **if** statement, the semantics of the Distl function can be described as follows:

$$\begin{array}{l} \mathsf{Distl} \Leftrightarrow \lambda \text{ a . ((a.item 2).size,} \\ \qquad\qquad \lambda \text{ p . (2, } \lambda \text{ q . if q} = 1 \rightarrow \text{(a.item 1)} \\ \qquad\qquad\qquad\qquad\qquad\quad \Box\!\rightarrow \text{(a.item 2).item p))} \end{array}$$

The definition of Distr is similar, differing only in the values of the **if** expression.

```
Distr ⇔ λ a . ((a.item 2).size,
              λ p . (2, λ q . if q = 1 → (a.item 1).item p
                                  □→ (a.item 2) ))
```

We state, again without going through the derivation, that (α Distl) is characterized as follows:

```
(α Distl) ⇔ λ a . (a.size,
                λ p . ( ( (a.item p).item 2), size,
                      λ r . (2, λ q . if q = 1 → ((a.item p).item 1
                                           □ → ((a.item p) 2).item r ) ) )
```

and that consequently the function CP can be described as follows:

```
CP ⇔ λ a . ((a.item 1).size,
           λ s . ((a.item 2).size, λ r . (2,
                 λ q . if q = 1 → (a.item 1).item s
                         □ → (a.item 2).item r ) ) )
```

We note one important observation concerning this characterization. The definitions of both Distl and Distr included conditional expressions, thus it may be surprising that the composition only contains a single **if** expression. The reason is that in the course of performing the composition, several λ arguments become bound to constants. This permits the evaluation of conditional parts of the **if** statements, which allows the selection of a single expression to be made at compile time, rather than at run time. This is a form of conventional dead code elimination [ASU86].

This illustrates a general feature of our representation, and a major reason why it is successful. During the process of composition, because of the combinator property of our functions, analysis can be performed easily to recognize special cases, reducing the size of the representation. Thus the size of the descriptor representation grows very slowly (sometimes even shrinking) as the number of terms of a composition increases.

4.3 Example 3 - Matrix Multiplication

The third example combines the first two examples to provide a function for matrix multiplication. It can be described as follows:

$$\text{Def } \mathsf{MM} \equiv (\alpha\alpha\mathsf{IP}) \circ \mathsf{CP} \circ [1, \mathsf{Trans} \circ 2]$$

Of the functional forms shown, the only one we have not yet discussed is the constructor $[f_1, f_2, ..., f_n]$. Like the apply to all, we describe this as a function, but a function of n arguments.

```
[f_1, ..., f_n] ⇔ λ f_1, ..., f_n .
             λ a . ( n,
                   λ p . if p = 1 → f_1 a
                         □ p = 2 → f_2 a
                         □ ...
                         □ p = n → f_n a)
```

Note that this characterization differs from Backus' in being non-strict, which is to say it is not $\perp$-preserving.

As an example of this form, the characterization of [1, Trans ∘ 2] turns out to be:

```
λ a . (2, λ p . if p = 1 → a.item 1
                □ p = 2 → (((a.item 2).item 1).size,
                           λ i . ((a.item 2).size, λ j . (a.item 2).item j) i)))
```

The computation of ($\alpha\alpha$ IP) yields what is among the largest of the intermediate representations we will discuss:

```
λ a . (a.size,
    λ p . ((a.item p).size,
        λ q . (-1, ν s . (s ← 0;
                         σ i, ((((a.item p).item q).item 1).size,
                           s ← s + ν r . (r ← 1;
                               σ k,((a.item p).item 2).size .
                                   r ← r * ((((a.item p).item q).item k).item i).item))))
```

The composition of this function with the CP function previously described illustrates the many simplifications that frequently occur, and which are responsible for the success of this technique. The expression (((a.item p).item q).item 1).size becomes simply ((a.item 1).item p).size, and the expression ((a.item p).item q).size becomes the simple constant 2. The result is thus:

```
λ a . ((a.item 1).size,
    λ p . ((a.item 2).size,
        λ q . (-1, ν s . (s ← 0;
                         σ i, ((a.item 1).item p).size .
                         s ← s + ν r. (r ← 1;
                         σ k, 2 .
                         r ← r * (((2, λ w . if w = 1 → (a.item 1).item p
                                                   □ → (a.item 2).item q).item k).item)))
```

This example illustrates how several conventional optimization techniques can profitably be applied to this representation. In particular, we have here a loop with a fixed and small upper bound. In such cases (say, where the constant is three or less), it is usually better to unroll the loop. Doing so in this case yields the following sequence.

```
ν r . (r ← 1;
    r ← r * ((a.item 1).item p).item;
    r ← r * (((a.item 1).item q).item 2).item )
```

Simple analysis, such as constant propagation coupled with data flow analysis, permits us to simplify this and remove the ν expression altogether, yielding a much simpler form for the composition. Combining this with the representation for composition previously described yields the following characterization of the MM program:

```
$\lambda$ a . ((a.item 1).size,
        $\lambda$ i . (((a.item 2).item 1).size,
                $\lambda$ j . (⇔1, $\nu$ c . ( c $\leftarrow$ 0;
                                        $\sigma$ k, ((a.item 1).item i).size .
                                            c $\leftarrow$ c + ( ((a.item 1).item i).item k) $*$
                                                        ((a.item 2).item k).item j)))))
```

There are several observations to make concerning the transformation from Backus' functional description to this form. The most important fact to keep in mind is that it was produced by a simple sequence of rewritings directly from the original functional form.

The second point to make concerns the size of this representation. If we, for example, count individual tokens as a measure of size, then despite the fact that the original definition contained over a dozen separate functions, the size of the result is only slightly larger than any one of the original functions, and is indeed smaller than some of the intermediate compositions. It is this slow growth in size which is, we feel, the most important feature of this representation.

The final point concerns the ease of producing conventional style code from this representation. This is the subject of the next section.

5 Generating code from the Representation

There are several inferences an intelligent code generator can derive from an examination of a function described in our intermediate representation. By considering the depth of λ expressions, for example, one can infer the nesting depth of the result, in terms of the arguments. That is, a function represented as three nested lambda expressions will return a value of at least nesting level two, such as $<< x_{11}, x_{12} >, < x_{21}, x_{22} >>$. The actual depth of nesting may be greater, depending upon the input values. However, we know x_{11} and so on are taken directly from the input arguments, and are not modified by this function.

In the particular case of the MM function, we can determine even more. Since the size field of the object returned by the innermost λ function shows it to represent a scalar, we know the result is exactly of nesting depth 2; that is a list of lists.

On the input side, the depth of *item* subscripting provides a lower bound on the necessary nesting of the input values. In the MM example, the deepest subscripting is found in the expression ((a.item 2).item r).item q). This indicates that the input must have nesting depth of at least three; that is lists of lists of lists. The fact that this value is being used in a scalar operator (multiplication) implies that the nesting level of the input must be exactly this.

Argument conformability checking aside, generating code for a conventional processor from this representation is straightforward. λ expressions (with the exception of the outermost) become loops, as do σ forms. ν expressions become a sequence of assignment statements. If we assume the result of a function is to be an array of arrays of scalars, compare, for example, the representation of the MM program given in the last section with the following:

```
$l_1$ := allocate((a.item 1).size);
**for** i := 1 **to** (a.item 1).size **do begin**
    $l_2$ := allocate(((a.item 2).item 1).size);
    $l_1$.item 1 := $l_2$;
    for j := 1 to ((a.item 2).item 1).size do begin
        c := 0;
        for k := 1 to ((a.item 1).item i).size do
            c := c + (((a.item 1).item i).item k) * ((a.item 2).item k).item j)
        $l_2$.item j := c;
        end
    end
return $l_1$;
```

There are clearly a number of conventional optimizations, such as detection of loop invariant expressions (a.item 1 and a.item 2, for example), which can profitably be applied to this code.

If one is generating code for a vector processor, yet another possibility is to detect patterns representing expressions which can be computed by vector expressions, in a manner similar to that done in other compilers for very high level languages [Bud84]. That is, given an expression such as (*size*, λ p . (*expression*)), one could attempt to compute all *size* elements in parallel.

6 Conclusions

In this paper we have outlined an approach to the problem of finding an internal (or intermediate) representation for a language based on compositions, and presented a specific example of that approach. Broadly speaking, the approach is to try to find a representation which meets the following objectives:

- It should be easy to translate from the source language into the intermediate representation.
- It should be easy to translate from the intermediate representation into whatever target language (virtual or actual machine code) is being used.
- Most important, the size of the representation should grow very slowly as the number of compositions increases. In particular, the size of a composed object should be significantly smaller than the sum of the sizes of the representations of the individual components.

The last point is the key to the success of this approach. In classical mathematics the representation of linear transformations by matrices, and the composition of transforms by matrix multiplication, is a good example of this objective. There have also been previous examples of exploiting such a representation in the generation of code for computer languages [GuW75].

In this paper we have presented such a representation for Backus' language FP, and illustrated how compositions are formed and how code can finally be generated from this representation. While our examples have been specific to FP, this technique should be applicable to any programming language based largely on compositions. Examples of such languages include other functional languages, such as Hope [Eis87] or Miranda [Pey86], as well as languages based upon the composition of expressions, such as APL [Ive80].

Similarly, while our examples of code generation have been directed at conventional von Neumann style processors, this is not inherent in the technique and there is no reason why this representation, or a different representation possessing the characteristics we have outlined, could not be profitably applied to other architectures.

7 Acknowledgments

Bruce D'ambrosio and Jim Rudolf read and made several useful comments on earlier drafts of this paper.

References

[ASU86] Aho, Alfred V., Sethi, Ravi, and Ulman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[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.

[BuT82] Budd, Timothy A. and Treat, Joseph., "Extensions to Grid Selector Composition and Compilation in APL", *Information Processing Letters*, Vol 19(3): 117-123, (Oct 1984).

[Bud84] Budd, Timothy A., "An APL Compiler for a Vector Processor", *ACM Transactions on Programming Languages and Systems*, Vol 6(3):297-313, (July 1984).

[Bud88] Budd, Timothy A., *An APL Compiler*, Springer-Verlag, 1988.

[Dij76] Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Eis87] Eisenback, Susan, *Functional Programming: Languages, Tools and Architectures*, Wiley, 1987.

[GuW75] Guibas, L. and Wyatt, D., "Compilation and Delayed Evaluation in APL", *Conference Record of the Fifth Principles of Programming Languages Conference*, Tucson, AZ, 1978.

[Ive80] Iverson, Kenneth E., "Notation as a Tool of Thought", *Communications of the ACM*, Vol 23(8):444-465 (August 1980).

[McC60] McCarthy, John, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM*, Vol 3(4):184-195 (April 1960).

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