Improved Dimensionality Analysis for Lucid

Gord Brown (gdbrown@uvic.ca), University of Victoria
Bill Wadge (wwadge@lucy.uvic.ca), University of Victoria

We describe work in progress on a method for determining the dimensionality of Lucid terms. This method gives a better (i.e. lower) upper bound on expressions involving function calls than the method currently in use.

1.0 Introduction

Dimensional analysis is used in the Lucid compiler to determine the dimensionality of terms in a program. Knowing the dimensionality of a term allows the runtime system to avoid unnecessary computation and caching of instances of a term, when the term does not, in fact, vary in some given dimension. In general, it is impossible to determine the dimensionality of a term exactly at compile time, but an upper bound can be determined. The lower the upper bound is, the greater the efficiency gain will be. This paper briefly describes the method of dimensional analysis currently used in the GLU (Granular Lucid) system, then describes an improvement on this method which provides better (i.e. lower) upper bounds for expressions involving user-defined functions and dimensional abstraction.

The general approach used is abstract interpretation: abstract from the (many-dimensional) value of a variable to the set of dimensions in which it varies. We then consider each operator in the language as a set operation on its operand(s), and compute the set of dimensions which result from the evaluation of each term in the program. The weakness of the existing method is that functions are considered to have a single dimensionality, computed once and for all, based on all possible invocations of the function, anywhere in the program. Each invocation of the function is assumed to result in the same dimensionality. If a function is invoked from many places in the program, with arguments of many different dimensionalities, the resulting function dimensionality will significantly overestimate the actual dimensionality at each invocation site. The improved method (originally suggested by Ed Ashcroft) is to derive an abstract dimensionality function corresponding to each function definition, and apply that function at each invocation site of the real function, to determine the dimensionality of the result at that site. This method will still overestimate the dimensionality of the result in some cases, but it is not sensitive to the number of places a function is called from.
2.0 Existing Approach

This description is based on reference [2], Chris Dodd’s paper on dimensionality analysis in the current GLU system. The approach is to compute iteratively the dimensionality for all terms in the program, until a fixed point is found (no change from one iteration to the next of the dimensionality of any term in the program).

Notation:
- \( \delta(X) \) denotes the dimensionality of an expression \( X \)
- \( \cup \) denotes the set union operator
- \( - \) denotes the set difference operator
- \( \{x\} \) denotes the singleton set containing dimension \( x \)
- \( \{} \) denotes the empty set

The basis of computation is an expression for each operator in the language, as well as some basic handling of user-defined functions. The expressions for the operators are:

<table>
<thead>
<tr>
<th>Operators</th>
<th>Lucid Expression</th>
<th>Set Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>pointwise unary operators</td>
<td>( \textit{op \ x} )</td>
<td>( \delta(x) )</td>
</tr>
<tr>
<td>pointwise binary operators</td>
<td>( \textit{x \ op \ y} )</td>
<td>( \delta(x) \cup \delta(y) )</td>
</tr>
<tr>
<td>constant</td>
<td>( \textit{&lt;some constant expression&gt;} )</td>
<td>( {} )</td>
</tr>
<tr>
<td>if-then-else</td>
<td>( \textit{if \ x \ then \ y \ else \ z \ fi} )</td>
<td>( \delta(x) \cup \delta(y) \cup \delta(z) )</td>
</tr>
<tr>
<td>#</td>
<td>#.(d)</td>
<td>( {d} )</td>
</tr>
<tr>
<td>@</td>
<td>( \textit{x \ @.d \ y} )</td>
<td>( \delta(x) \setminus {d} \cup \delta(y) )</td>
</tr>
<tr>
<td>first</td>
<td>( \textit{first.d \ x} )</td>
<td>( \delta(x) \setminus {d} )</td>
</tr>
<tr>
<td>fby, before, asa, wvr, upon</td>
<td>( \textit{x \ op.d \ y} )</td>
<td>( \delta(x) \setminus {d} \cup \delta(y) )</td>
</tr>
<tr>
<td>next, prev</td>
<td>( \textit{op.d \ x} )</td>
<td>( {d} \cup \delta(x) )</td>
</tr>
</tbody>
</table>

Starting with an initial assumption that each term’s dimensionality is \( \{} \), the above rules are applied in a bottom-up fashion to all the expressions in the program, repeatedly until the fixed point is reached.

User-defined functions and abstract dimensions are handled by assigning them dimensionalities based on the dimensionality of all their invocation sites, then assuming that dimensionality for the value of the function at all invocation sites. This approach gives a higher upper bound than necessary, since there is no reason to think that the result of evaluating a function on one input will have the same dimensionality as the result of evaluating it on some completely different input, if those inputs vary in different dimensions.

3.0 Improved Handling of Functions

The improvement described in this paper is the generation of a dimensionality function for each function in a program. The dimensionality function is used at each site at which
the function is invoked, to compute, based on the dimensionality of the arguments, abstract dimensions, and whatever dimensions the function itself might contribute, the dimensionality of the result of the function call.

The dimensionality function is an expression composed of set union, set difference, constant dimension sets, variable dimension sets (corresponding to the arguments of the function and any abstract dimensions used in the function), and function calls (as well as references to global dimensions and variables). It is generated in a straightforward manner from the function definition, using the same rules as are used in normal dimensionality determination, but building an expression tree instead of immediately evaluating it.

For example, the function definition:

\[ f \cdot d(a, b) = a \oplus d b; \]

would generate the following dimensionality function, corresponding to the effect of the function arguments on the dimensionality of the result:

\[ f' \cdot d(a, b) = (\delta(a) - \{d\}) \cup \delta(b) \]

When evaluating the dimensionality of a term, as described in the previous section, this abstract function \( f' \) is used at the invocation sites of function \( f \), to determine the dimensionality of the result at that invocation site.

Dimensionally abstract variable definitions are considered to be functions with no arguments (or, rather, with only dimensional arguments).

Some benefits of this approach, beyond the basic one of treating each function invocation independently, are that the handling of dimensional abstraction is extremely simple, and that the technique deals naturally with the situation where some variable's dimensionality does not contribute to the dimensionality of the result (i.e. an argument is not used in the function body). Also, the method easily deals with local dimensions, since the set expressions work with whatever is in the local scope at the invocation site. Some care must be taken to handle duplicate dimension names (say, the same dimension name is found in the scope of the function definition and the invocation site, but referring to different dimensions), but no real problem arises.

3.1 Recursion

Recursion makes the construction of the dimensionality function more complicated. An initial cut at a dimensionality function results in a set expression defined in terms of itself (or of some other mutually-recursive function).

A solution is to compute the dimensionality function itself in much the same way as the dimensionality of a simple expression is computed. An initial approximation of the dimensionality function is assumed: \( f' (x) = \{\} \). Then, if a recursive call to \( f \) shows up in the body of \( f \), it is replaced with \( \{\} \). When the definition of \( f' \) has been generated, it is recomputed with the pass-0 edition of \( f' \) inserted in the body where the recursive call to \( f \) occurs. Generation is repeated until there is no change from one iteration to the next. Note that reduction of the computed dimensionality function to a normal form is
Improved Handling of Functions

essential for detection of the fixed point. To handle mutual recursion, all functions have
their dimensionality functions computed together, using the usual iterative process until
the fixed point is found. Because the rules used for computing the set expressions are
additive, a fixed point will eventually be reached.

Consider a simple example (granted, an iterative solution to this problem would be more
natural in Lucid):

\[ f(x) = \text{if } (x < 2) \text{ then } x \text{ else } f(x-1) + f(x-2); \]

The first pass would produce the following set expression (after some straightforward
reduction):

\[ f'(x) = \delta(x) \cup f'(x) \]

Substituting \( \emptyset \) for \( f'(x) \), the expression becomes:

\[ f'(x) = \delta(x) \]

The next iteration will produce:

\[ f'(x) = \delta(x) \cup \delta(x) \]

which reduces to

\[ f'(x) = \delta(x) \]

and the process halts, having found the fixed point. In general, of course, it could take
several iterations before the fixed point is found.

3.2 Normal Form

The following normal form for dimensionality functions was proposed by Bill Wadge.
The dimensionality function is represented by a matrix for each argument to the func-
tion: the indices of the matrix are the relevant dimensions of the argument; a 1 at a par-
ticular row and column indicates that if the argument varies in the row dimension, the
result of the function will vary in the column dimension, while a 0 indicates no effect.
An additional flag per input variable is needed to indicate whether, for that argument,
dimensions not explicitly listed in the matrix contribute to the result or not.

The set of relevant dimensions for a given function consists of those dimensions which
are defined in the scope in which the function is defined and appear explicitly in the
function, plus any abstract dimensions declared in the function definition.

At the implementation level, abstract dimensions must be distinguished from other
dimensions, since they will be replaced by real dimensions when the dimensionality
function is applied at some particular invocation site. However, this does not appear to
create any problems.

This representation is actually more expressive than the set representation, in its ability
to handle correctly expressions that swap dimensions. For example, consider the
"realign" program described in Multidimensional Programming [1, page 14]:

Improved Dimensionality Analysis for Lucid
An Example

realign.a,b(X) = X @ a #: b;

If X varies in a, the result will vary in b, and the existing approach will reflect that. However, if X does not vary in a, the result should not vary in b, either. The set expressions will not correctly represent that, but instead will insist that the result varies in b, due to the #: b component of the expression.

The details of transforming the set expressions to this normal form are left as an exercise for the reader.

3.3 Efficiency

One concern is the efficiency of the proposed approach. More computation may necessary at each invocation site to determine the dimensionality of the result. However, the gain in efficiency from not having to compute and cache values for dimensions in which a term does not actually vary should offset this additional effort. At any rate, the amount of computation necessary is minor, particularly considering that it is compile time.

4.0 An Example

Consider the following (somewhat contrived) Lucid program:

res where
    dimension w,x,y,z;
    res1 = f.z(a);
    res2 = f.w(b);
    f.q(k) = k fby.q k + 1;
    a = 0 fby.x a + 1;
    b = 0 fby.y b + 1;
    res = res1 + res2;

end;

The first step is to generate the dimensionality function for f. Building the set expression from the bottom up, the result will be:

f'.q(k) = U (δ(k), {q}, U (δ(k), {}))

Reduction to the simplest form (δ(k) U {q}) is straightforward, although the prototype implementation does not do it. The current implementation in the GLU compiler would compute a single dimensionality of U (δ(a), δ(b), {x}, {w}) for the function.

The next step is to repeatedly compute the dimensionality of the top-level term, bottom up, until there is no change from one iteration to the next. The system assumes an initial dimensionality of {} for all terms. In the first iteration, the system computes the following, in the order given:

δ(res1) = U (δ(a), {z}) = {z}
Further Work

\[ \delta(\text{res2}) = \bigcup(\delta(b), \{w\}) = \{w\} \]
\[ \delta(a) = \bigcup(\delta(a), \{x\}) = \{x\} \]
\[ \delta(b) = \bigcup(\delta(b), \{y\}) = \{y\} \]
\[ \delta(\text{res}) = \bigcup(\delta(\text{res1}), \delta(\text{res2})) = \{w, z\} \]

Since the initial assumption is that \(\delta(a)\) and \(\delta(b)\) are \(\emptyset\), the first guess of dimensionality for \(\text{res1}\) and \(\text{res2}\) is incomplete.

Second iteration:
\[ \delta(\text{res1}) = \bigcup(\delta(a), \{z\}) = \{x, z\} \]
\[ \delta(\text{res2}) = \bigcup(\delta(b), \{w\}) = \{w, y\} \]
\[ \delta(a) = \bigcup(\delta(a), \{x\}) = \{x\} \]
\[ \delta(b) = \bigcup(\delta(b), \{y\}) = \{y\} \]
\[ \delta(\text{res}) = \bigcup(\delta(\text{res1}), \delta(\text{res2})) = \{w, x, y, z\} \]

Third iteration:
\[ \delta(\text{res1}) = \bigcup(\delta(a), \{z\}) = \{x, z\} \]
\[ \delta(\text{res2}) = \bigcup(\delta(b), \{w\}) = \{w, y\} \]
\[ \delta(a) = \bigcup(\delta(a), \{x\}) = \{x\} \]
\[ \delta(b) = \bigcup(\delta(b), \{y\}) = \{y\} \]
\[ \delta(\text{res}) = \bigcup(\delta(\text{res1}), \delta(\text{res2})) = \{w, x, y, z\} \]

Since the third iteration is the same as the second, the computation is complete. The key point is that the computed dimensionalities of \(\text{res1}\) and \(\text{res2}\) are smaller than they would have been with the current method (and, in fact, are exactly correct, although that will not always be the case). The method used in the current GLU implementation would compute a dimensionality of \(\{w, x, y, z\}\) for both \(\text{res1}\) and \(\text{res2}\), and would further inflate their dimensionalities if \(f\) were used other places (with arguments having different dimensionalities). Using the proposed method, additional invocations of function \(f\) will not inflate the computed dimensionalities of these invocations.

### 5.0 Further Work

A prototype implementation of the scheme described in this paper has been implemented, but it is incomplete in several respects. It does not correctly implement the recursion technique, or the normal form. It does not handle local dimensions. However, there appear to be no non-trivial roadblocks to implementing them.

A more advanced approach (suggested by Bill Wadge) is demand-driven abstract interpretation. The idea is to use the same eductive evaluation for abstract interpretation as for evaluation of the program itself. This technique promises to be a convenient solution to the question of how to transform dimensionality functions into the proposed normal form, as well as being an interesting application of eduction itself.
6.0 Summary

The enhanced dimensionality analysis method described in this paper computes a better upper bound for expressions involving dimensional abstraction and user-defined functions. In particular, each invocation of a function is treated separately, so many uses of a function do not inflate the upper bound determined at each site. Other benefits are ease of handling dimensional abstraction and recursion, and a lack of sensitivity to the actual number of dimensions in the system. Implementation of the transformation of dimensionality functions to the proposed normal form will allow the technique to handle recursion correctly. Further work on eductive abstract interpretation may provide a technique for achieving the transformation (or perhaps avoiding the need for it).

7.0 References

