different contexts. The weakness noted above could be reduced by somehow using actual dimensions passed to a dimensionally abstract function within the function. It is probably impractical to do a separate rank analysis of the function body for each call site, however. Ed Ashcroft has suggested in private correspondence finding a "rank function" for each function and applying the function to the ranks of the actual arguments to find the rank of any call site. Bill Wadge[1] has suggested finding the rank itself eductively.

1. E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional Programming.* Oxford University Press, 1995.
2. R. Jagannathan and C. Dodd. GLU programmer's guide (version 0.9). Technical Report SRI-CSL-94-06, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, July 1994.
3. W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

The last piece is to find the rank of the results of the program. This is actually done on the first pass, but won't change on later passes, as Rank [`tot`] is unchanged.

$$\begin{aligned}
\text{Rank } [\texttt{linear_tree.i(data, fn, size)}] &= \text{Rank } [\texttt{tot}] - \{\texttt{i}\} - \{\texttt{t}\} \\
&= \{\texttt{d}\} \\
\text{Rank } [\texttt{result}] &= \text{Rank } [\texttt{linear_tree}] = \{\texttt{d}\} \\
\text{Rank } [\texttt{result @.d 0}] &= \emptyset
\end{aligned}$$

This reveals one weakness in the analysis — the rank of `linear_tree` should be empty due to the `first.i`. Unfortunately the analysis only removes the abstract dimension `i` from the rank, not the actual dimension `d` to which it refers. In a program with multiple uses of a function such as `linear_tree` with differing actual dimension, the problem becomes even worse.

4. Ranks in optimization

The most important optimization enabled by rank analysis is the elimination of redundant `@`–expressions.

$$E_1 \ \texttt{@.d} \ E_2 \Rightarrow E_1 \qquad \text{if d} \notin \text{Rank } [E_1]$$

This optimization is complicated by the fact that it can't be done if eliminated term (E_2) might contain an infinite loop. We can still use this optimization as long as we can prove that E_2 will terminate. This comes up most frequently in the form of a `first` or `next` expression on an irrelevant dimension.

Related to this, the rank of the actual parameters to a function call can be used to eliminate save and restore of local dimensions which occurs around a function call.

5. Ranks in value caching

The calculated ranks are used by the GLU runtime system to control the variable value cache. Each variable is cached based solely upon the dimensions in its rank, rather than the entire dimension set tag.

The rank is also important in calculating usage counts. Any term whose rank is not a superset of every parent term has essentially an infinite usage count. This is an effect of basing the cache on the ranks, as such a term will get many demands which differ only in the dimensions not in its rank.

6. Improvements

The main limitations of the current GLU rank analysis scheme are due to the compromises dealing with functions, particularly when they have many call sites with

$$
\begin{aligned}
\text{Rank}\ [\texttt{tmp}] &= \text{Rank}\ [\texttt{s == 1}] \cup \text{Rank}\ [\texttt{\#.t}] \cup \ldots = \{\texttt{t}\} \\
\text{Rank}\ [\texttt{tot}] &= \text{Rank}\ [\texttt{\#.t <= 0}] \cup \text{Rank}\ [\texttt{data}] \cup \text{Rank}\ [\texttt{tot1 @.t \#.t-1}] \\
&= \{\texttt{t}\} \cup \{\texttt{d}\} \cup (\text{Rank}\ [\texttt{tot1}] - \{\texttt{t}\} \cup \{\texttt{t}\}) \\
&= \{\texttt{d, t}\} \cup \text{Rank}\ [\texttt{tot1}]
\end{aligned}
$$

To find the rank of `tot1` it is first necessary to find the rank of `add`. There is only one call site for `add`, so it first finds the dimensions which are `local` to the call site and the rank of the actual arguments. Once this has been done, it can find the rank of the entire function, and add the `local` dimensions back to get the rank of the call site.

On the first pass, Rank [`tot`] will not have been determined yet and so will still be $\emptyset$.

$$
\begin{aligned}
\texttt{local} &= \{\texttt{i, t}\} \\
\text{Rank}\ [\texttt{x}] &= \text{Rank}\ [\texttt{tot}] - \texttt{local} = \emptyset \\
\text{Rank}\ [\texttt{y}] &= \text{Rank}\ [\texttt{tot @.i \#.i+1}] - \texttt{local} = \emptyset \\
\text{Rank}\ [\texttt{add(x, y)}] &= \text{Rank}\ [\texttt{x}] \cup \text{Rank}\ [()x) = \emptyset \\
\text{Rank}\ [\texttt{fn(...)}] &= \text{Rank}\ [\texttt{add(x,y)}] \cup \texttt{local} = \{\texttt{i, t}\} \\
\text{Rank}\ [\texttt{tot1}] &= (\text{Rank}\ [\texttt{fn(...)}] - \{\texttt{i}\}) \cup \text{Rank}\ [\texttt{2 * \#.i}] \\
&= (\{\texttt{i, t}\} - \{\texttt{i}\}) \cup \{\texttt{i}\} = \{\texttt{i, t}\} \\
\text{Rank}\ [\texttt{tot}] &= \{\texttt{d, i, t}\}
\end{aligned}
$$

On the second pass, Rank [`tot`] will have the value determined for it on the first pass.

$$
\begin{aligned}
\text{Rank}\ [\texttt{x}] &= \text{Rank}\ [\texttt{tot}] - \texttt{local} = \{\texttt{d}\} \\
\text{Rank}\ [\texttt{y}] &= \text{Rank}\ [\texttt{tot @.i \#.i+1}] - \texttt{local} = \{\texttt{d}\} \\
\text{Rank}\ [\texttt{add(x, y)}] &= \text{Rank}\ [\texttt{x}] \cup \text{Rank}\ [()x) = \{\texttt{d}\} \\
\text{Rank}\ [\texttt{fn(...)}] &= \text{Rank}\ [\texttt{add(x,y)}] \cup \texttt{local} = \{\texttt{d, i, t}\} \\
\text{Rank}\ [\texttt{tot1}] &= (\{\texttt{d, i, t}\} - \{\texttt{i}\}) \cup \{\texttt{i}\} = \{\texttt{d, i, t}\} \\
\text{Rank}\ [\texttt{tot}] &= \{\texttt{d, i, t}\}
\end{aligned}
$$

The third pass will not change the ranks of any term and the rank analysis will be complete.

of rank overestimates.

3. A Complete Example

Here's a simple example program to add 32 numbers in a tournament, to illustrate the rank analysis algorithm.

```
result where
     dimension d;
     result = linear_tree.d(input(#.d), add, 32);
     add(x,y) = x + y;
     linear_tree.i(data, fn, size) = first.i(tot asa.t s == 1) where
          dimension t;
          s = size fby.t s/2;
          tot = data fby.t fn(tot, next.i tot) @.i (2 * #.i);
     end;
end
```

First, the compiler transforms all indexical builtins to the equivalent `@`–expression.

```
result @.d 0 where
     dimension d;
     result = linear_tree.d(input(#.d), add, 32);
     add(x,y) = x + y;
     linear_tree.i(data, fn, size) = ((tot @.t tmp) @.i 0) @.t 0 where
          dimension t;
          tmp = if s == 1 then #.t else tmp @.t (#.t+1) fi;
          s = if #.t <= 0 then size else (s/2) @.t (#.t-1) fi;
          tot = if #.t <= 0 then data else tot1 @.t (#.t-1) fi;
          tot1 = fn(tot, tot @.i #.i+1) @.i (2 * #.i);
     end;
end
```

The compiler initializes the rank of every term to be $\emptyset$, and then does bottom-up passes of the entire program applying the rules.

The first bottom-up traversal will first determine the ranks of the arguments to linear_tree.

$$\text{Rank } [\texttt{data}] = \text{Rank } [\texttt{input(\#.d)}] = \{\texttt{d}\}$$
$$\text{Rank } [\texttt{size}] = \text{Rank } [\texttt{32}] = \emptyset$$

Next, it finds the ranks of the various definitions in linear_tree.

$$\begin{aligned} \text{Rank } [\texttt{s}] &= \text{Rank } [\texttt{\#.t <= 0}] \cup \text{Rank } [\texttt{size}] \cup \text{Rank } [\texttt{(s/2) @.t (\#.t-1)}] \\ &= \{\texttt{t}\} \cup \emptyset \cup \{\texttt{t}\} = \{\texttt{t}\} \end{aligned}$$

A second bottom up pass will determine the same ranks for every term, so rank analysis ends.

2.2. Local dimensions

When a local dimension is introduced by a where clause, the GLU compiler inserts a `first.`*dim* operation around the top term of the where clause. This will be translated into the equivalent `@`–epxression and handled by the above rule.

2.3. Functions

Dealing with function calls is the most difficult part of rank analysis. When dealing with a function body, the GLU compiler first finds all the call sites for the function. For each call site, the compiler finds the set of dimensions which are "local" to the call site. This is the syntactic rank of the call site minus the syntactic rank of the definition site, and minus the actual dimensional parameters. The local dimension set is also the set of dimension which needs to be saved across a call at runtime.

The rank of each formal parameter becomes the union of the ranks of all the corresponding actual parameters minus the local dimensions from all the call sites. The rank of the call site is the rank of the function definition plus the intersection of the local dimensions and the union of the actual parameters.

$$
\begin{aligned}
\texttt{local}\,[F.\text{dims}(\ldots)] &= \text{SyntaxRank}\,[\textit{here}] - \text{SyntaxRank}\,[F] - \text{dims} \\
\text{Rank}\,[\textit{formal}] &= \bigcup_{\text{callsites}} \text{Rank}\,[\textit{actual}] - \texttt{local} \\
\text{Rank}\,[F.\texttt{dims}(\ldots)] &= \text{Rank}\,[F] \cup \left(\texttt{local} \cap \bigcup_{\text{args}} \text{Rank}\,[\textit{arg}] \right)
\end{aligned}
$$

The use of the Syntactic Rank (rather than the calculated rank) when computing the local dimensions is important to maintain the purely additive nature of the rules. Since the `local` set is constant throughout the rank analysis, it can safely be subtracted from the rank set without causing an infinite loop.

2.4. Abstract dimensions

A final pass over the calculated ranks is used to add all possible actual dimensions to any rank that contains an abstract dimension. This is probably the biggest source

2. Calculating the ranks

The GLU compiler determines the rank of every term in the program with a simple iterative algorithm. Initially, the rank of every term is set to the empty set. Then, a set of simple rules are used to determine the rank of each term based on the ranks of it operands. The rules are applied to every term in the parse tree of the program using a simple bottom-up recursive descent algorithm. They are applied repeatedly until a fixed point is reached; once the compiler finds that further application of the rules no longer changes the rank of any term, it is finished.

The rules are designed to be simple and fast and, in some cases, are overly conservative, leading to an overestimate of the rank. The rules are designed to be purely additive, so that once a dimension is added to the rank of a term, further iterations will not remove it. This insures that eventually the compiler will reach a point where no more dimensions can be added to any rank, which is the termination condition.

While parsing, the GLU compiler also maintains a Syntactic Rank for every term. This Syntactic Rank is simply the set of all dimensions which are "in scope" while parsing the term.

2.1. Simple Expressions

Before rank analysis begins, all Lucid indexical builtin operators are translated to an equivalent form using `@`[3]. Pointwise operators, including externally declared C functions, are handled by simple rules.

$$
\begin{aligned}
\text{Rank}\,[\textit{constant}] &= \emptyset \\
\text{Rank}\,[\texttt{\#.d}] &= \{\texttt{d}\} \\
\text{Rank}\,[E_1\ \textit{op}\ E_2] &= \text{Rank}\,[E_1] \cup \text{Rank}\,[E_2] \\
\text{Rank}\,[\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \texttt{fi}] &= \text{Rank}\,[E_1] \cup \text{Rank}\,[E_2] \cup \text{Rank}\,[E_3] \\
\text{Rank}\,[E_1\ \texttt{@.d}\ E_2] &= (\text{Rank}\,[E_1] - \{\texttt{d}\}) \cup \text{Rank}\,[E_2]
\end{aligned}
$$

For example, consider the one-line program "`time + 5`". The compiler requires one bottom-up pass to determine the ranks of every term:

$$
\begin{aligned}
\text{Rank}\,[\texttt{time}] &= \{\texttt{time}\} \\
\text{Rank}\,[\texttt{5}] &= \emptyset \\
\text{Rank}\,[\texttt{time + 5}] &= \{\texttt{time}\} \cup \emptyset = \{\texttt{time}\}
\end{aligned}
$$

Rank Analysis in the GLU Compiler

Chris Dodd
SRI International
333 Ravenswood Ave
Menlo Park CA 94025
E-mail: dodd@csl.sri.com

ABSTRACT

Rank analysis in Lucid is crucial for efficient implementation. The GLU compiler uses a set of simple rules for determining the rank of every term, and iterates these rules until it reaches a fixed point. As it is impossible to exactly determine the rank in some cases, the GLU compiler deliberately overestimates the rank where it cannot determine it exactly.

1. Motivation

When the first version of GLU to support multiple dimensions was built, we observed that many seemingly simple programs ran either very slowly or not at all. Careful inspection of the running program revealed that it was recomputing many terms, even though the values of those terms remained the same, as it traversed some non-related dimension. The following simple program to compute squares by addition illustrates the problem.

```
squares @.x time where
     dimension x;
     time = 1 fby.time time+1;     // this program predates ``#''
     squares = time fby.x squares+time;
end
```

When this program was run, we expected the amount of computation required to increase linearly with subsequent iteration (increasing `time`). Instead, we found it required quadratic time. Each step in `x` while computing `squares` caused it to recompute `time` from scratch. Even worse than the increased computation time was the increased space required for the value cache. More complex programs were unable to complete as the value cache rapidly expanded to use up all available memory on the machine.

The solution to this problem is to compute some set of dimensions which are relevant to each term, and base value cache access only on those dimensions. This set is the "rank" of the term, and the rank is determined through Rank Analysis.