Induction Operators and Intermediate Forms

Christopher W. Fraser, AT&T Bell Laboratories

600 Mountain Avenue 2C-300, Murray Hill, NJ 07974-0636, USA. Internet: `cwf@research.att.com`.

Todd A. Proebsting, University of Arizona

Department of Computer Science, Tucson, AZ 85721, USA. Internet: `todd@cs.arizona.edu`.

This paper shows significant benefits from small extensions to typical compiler intermediate representations (IRs). We have added to such an IR two features that operate on sequences and thus hide simple loops. For example, we represent matrix multiplication with an expression dag of about 20 nodes and no explicit control flow. This design simplifies many optimizations. Examples include a 200-line program that partially evaluates dags with respect to any subset of their inputs and a tree-matching code generator that can exploit tuned linear algebra packages. It should be possible for loop analyzers to retrofit these operators to existing IRs.

1.0 Introduction

Many compiler optimizations that are simple on expression trees — and, in some cases, even on dags — become much harder in the presence of explicit control flow. For example, partial evaluation, which took years to develop for general programs, reduces to textbook constant folding when control flow isn't involved.

We have been experimenting with a more economical approach. Rather than invest heavily in optimizations that handle explicit control flow, we've been investigating what can be gained by minimal changes to the intermediate representation (IR) on which the compiler operates. We've extended a typical IR to include two operations on sequences. The IR uses dags over the usual arithmetic operators, but it adds two *induction operators* adapted from APL [Falkoff and Iverson, in Wexelblat]. One generates a sequence of values and the other reduces a sequence. It should be possible to introduce induction operators into the IR of an existing compiler if the language supports array operations or if the compiler does some loop analysis.

Induction operators don't hide all control flow, but they can hide simple loops, which suffices to simplify some important optimizations. We support this claim with one sample optimization and outlines of several others. Examples include partially evaluating a series of dags with respect to any subset of their inputs, and automatically identifying subtrees best implemented by calling tuned math libraries.

This paper is about a language design. The language happens to involve language *implementation*, but the paper is mainly about the *design* of the language and the rationale for the design. Like most papers about new language designs, this one can't offer many users, programs, or measurements, at least when compared with the more typical papers about new implementation work for established languages, but even the initial evidence below makes the case for induction operators.

2.0 The Intermediate Language

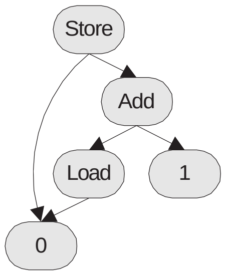
The demonstration optimizer uses a program that reads dags encoded in a functional notation. The usual arithmetic operators work on values that originate in a series of numbered memory cells. `Load(x)` returns the value in cell `x`, and `Store(y,z)` copies the value `z` into cell `y`, so the input

```
Store(0,Add(Load(0),1))
```

represents a tree that increments cell 0. The input `x=y` directs the program to use `y` for `x` in what follows. For example,

```
N=0
Store(N,Add(Load(N),1))
```

generates the dag



and is another input that increments cell 0. Temporaries like `N` must be set at most once, so functional, single-assignment semantics apply. The “=” operator above exists to allow one to create shared dag nodes, though it may also be used to present large trees in pieces. The IR uses a single numeric type; more base types are trivially added, but for now they’d only clutter experiments that focus on other matters, namely control flow.

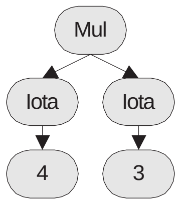
Each dag is interpreted in isolation. Each dag must have a single root. Each node in each dag must from be reachable from its root. `Store` may appear only as the root. These restrictions collaborate to give the dags functional semantics. Arguments can thus be evaluated in any order, including in parallel.

The operator `Iota(N)` (the name is APL’s) generates the values 0 through `N-1`, inclusive. That is, if a dag includes an `Iota(N)`, then the dag is effectively evaluated `N` times. `Iota` may be used only to represent code that yields the same value regardless of the order in which `Iota` yields its values, including evaluation in parallel. To simplify matters for the reader, all examples in this paper have `Iota` yield its values sequentially.

If a dag includes multiple distinct `Iota` nodes, it is evaluated once for each combination of the values of each generator. For example,

```
Mul(Iota(3),Itoa(4))
```

generates the dag



and yields not just one value but rather generates the sequence

```
0 0 0 0 0 1 2 3 0 2 4 6
```

which is easier to read like this:

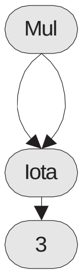
```
0 0 0 0
0 1 2 3
0 2 4 6
```

Multiple `Iota` nodes may be used only to represent code that yields the same value regardless of the order in which the `Iota`s combine. For example, the code that uses the sequence above must accept any permutation of the sequence.

If a generator is used more than once in a dag, it is generated only once, and each user gets the values as if in parallel. For example,

```
I=Iota(3)
Mul(I,I)
```

builds the dag



and generates the sequence

```
0 1 4
```

Our generator semantics come from Icon [Griswold and Griswold]. Indeed, replace each “=” above with “:=”, separate the assignments with “&”, embed the result in a “every write(...)”, and link in the primitives (e.g., `Mul`), and the resulting Icon program generates and emits the values requested. Outside Icon, `Iota` could be implemented with coroutines, but the restrictions on the semantics exist to permit simpler choices. Options include a stack of live generators or a single loop nest, with each generator in the dag defining one loop.

`Iota` can’t replace arbitrary control flow, but it can replace simple loops. For example, the inner loop of a standard matrix multiplier is

```
A=0
B=4
N=2
K=Iota(N)
Aik=Load(Add(A,Add(K,Mul(N,I))))
Bkj=Load(Add(B,Add(J,Mul(N,K))))
Addends=Mul(Aik,Bkj)
```

which generates in `Addends` the sequence of values that, when summed, yield position (`I`,`J`) in the product of the 2 by 2 (row-major, zero-origin) matrices A and B, which are stored at locations 0 and 4. For the time being, I and J are free variables.

The dag operator `Reduce(F,X)` exhausts all generators in `X` and yields a scalar. `F` must be a binary operator with an identity. If `X` generates nothing, `Reduce` yields `F`’s identity. Otherwise, `Reduce(F,X) = F(First(X),Reduce(F,Rest(X)))`. For example,

```
Reduce(Add,Iota(4))
```

yields the scalar value 0+1+2+3=6. Appending

```
Cij=Reduce(Add,Addends))
```

to the matrix multiplier begun above completes the inner loop, and appending

```
I=Iota(N)
J=Iota(N)
C=8
Dest=Add(C,Add(Mul(I,N),J))
Store(Dest,Cij)
```

completes the matrix multiplier.

The IR above starts with conventional low-level operators and adds two operators. Without `Iota` and `Reduce`, each dag represents a constant number of instructions. With `Iota` and `Reduce`, a dag can easily replace many nested loops. They can't represent all control flow, and they can't represent even a simple loop with one conditional inside, so many loops must still be represented conventionally. They can, however, represent many inner loops in many computationally intensive codes and trivialize some otherwise costly optimizations.

3.0 Partial Evaluation

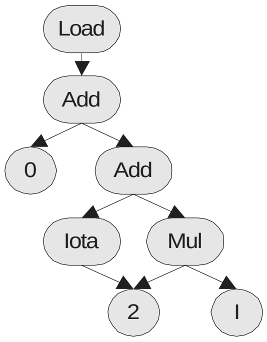
Partial evaluation is, at least for the purposes of the augmented dags in this paper, roughly equivalent to symbolic simulation and to constant propagation plus constant folding and loop unrolling.

Partial evaluation of dags is simple. `Iota` and `Reduce` complicate matters a little, but the cost is far less than implementing full partial evaluation. A 200-line Icon program reads the input described in the last section, compiles it into dags, partially evaluates them, and emits an Icon program that performs any remaining computations when the rest of the inputs become known. Indeed, the partial evaluator could be regarded as a partial evaluator for a limited subset of Icon expressions.

Consider the definition of `Aik` in the previous section. If we present

```
A=0
N=2
K=Iota(N)
Aik=Load(Add(A,Add(K,Mul(N,I))))
```

the partial evaluator builds the dag below. (In practice, the program needs to know the identity for each operator, so it replaces the `Add(0,X)` with `X`.) As long as `I` remains a free variable, simple examination of the dag shows that the only subdag that can be completely evaluated is the one rooted at the `Iota`. That the `Iota` generates multiple values is immaterial so long as the complete set of values is known. The partial evaluator emits



```
Load(Add(0,Mul(2,I)))
Load(Add(1,Mul(2,I)))
```

Again, the redundant addition of zero is removed but included above because the parallel construction makes the example easier to understand.

The two-dag input below helps demonstrate the full power of this simple partial evaluator. The first dag copies 0-7 into the first 8 cells of memory, and the second multiplies the 2-by-2 matrices starting at locations 0 and 4 and leaves the result in the cells starting at 8.

```
Mem=Iota(8)
Store(Mem,Add(1,Mem))

N=2
k=Iota(N)
Aik=Load(Add(0,Add(Mul(N,I),K)))
Bkj=Load(Add(4,Add(Mul(N,K),J)))
Cij=Reduce(Add,Mul(Aik,Bkj))
I=Iota(N)
J=Iota(N)
Store(Add(8,Add(Mul(N,I),J)),Cij)
```

With this input, all arguments to the matrix multiplier are known, so the partial evaluator does the complete computation at compile time and emits

```
Store(8,19)
Store(9,22)
Store(10,43)
Store(11,50)
```

If we replace the digit 8 with 7 in the first line for the first dag, however, the lower right corner of the second matrix is unknown, which means that the partial evaluator can now compute only the first result column at compile time. It emits code to finish the calculation when the last part of the input becomes known at execution time:

```
Store(8,19)
Store(9,Add(6,Mul(2,Load(7))))
Store(10,43)
Store(11,Add(18,Mul(4,Load(7))))
```

4.0 Math Libraries as “Instructions”

It is well known that a tree-parsing, dynamic-programming code generator can emit optimal local code for expression trees [Aho and Johnson], but few applications have used expression trees with implicit control flow because few target machine instructions offer implicit loops. Math libraries tuned for the memory hierarchy or pipeline or both, however, present larger “primitives” that are increasingly important. Without induction operators, no simple way is known to automatically introduce such library calls into generated code, unless the programmer had the knowledge to place them there in the first place. *With* induction operators, this problem has a textbook solution.

For example, the tree grammar [Aho, Ganapathi, and Tjiang; Fraser, Henry, and Proebsting] below recognizes loops that can be implemented by the inner-product routines in the Level 1 BLAS library [Lawson, Hanson, Kincaid, and Krogh].

```
induct: Iota(scalar)
induct: Add(induct,constant)
induct: Mul(induct,constant)
scalar: Reduce(Add,Mul(Load(induct),Load(induct)))
```

The first line notes that an `Iota` node generates a sequence corresponding to an induction variable. The next two lines note that one induction variable plus or times a constant yields another induction variable. The last line identifies inputs that fit the dot-product template. Each rule would need an attribute equation or semantic action. For example, the last rule above would need an action to emit the call on an inner-product routine.

Some rules would also need a cost function. For example, the last rule above would use a cost function that confirms that the two embedded induction variables are “dagged” and share one `Iota`, because dot products use one loop, not two. If they are, then the cost function would estimate the cost of the tuned implementation of the library routine. If, for example, software pipelining has reduced the average cost per iteration to, say, six cycles, then the cost for this rule should be six times the number of iterations or or some compile-time estimate of it.

The cost functions can reject the rule — for example, if the two embedded induction variables above aren’t shared — by returning an effectively infinite value, which forces the tree parser to choose other, presumably more general and thus more costly, rules to cover these nodes. For example, the rules

```
scalar: Reduce(Add, expr)
expr: induct
expr: Mul(expr, expr)
expr: Load(expr)
```

combine to give an alternate match for dot products, but the costs associated with these rules should add up to more than six cycles per iteration, because this loop is assembled from individual instructions and doesn't benefit from tuning.

It may be possible to recast the partial evaluator as a tree parser in the style above.

5.0 Other Applications

Without induction operators, a typical dag represents perhaps a few tens of instructions between side-effects. With induction operators, dags can represent thousands of instructions without explicit side-effects. These properties should benefit a variety of optimizations:

- Cache optimizations such as blocking benefit from advance information about the coming sequence of memory references and from permission to reorder some references. Dags with induction operators represent this data compactly and functionally.
- Instruction scheduling and software pipelining might be simpler on a single, uniform representation for loop nests *and* basic blocks.
- Optimizers that automatically introduce parallelism need to partition and reorder computations. Induction operators make some dags "big enough" to partition profitably, and they explicitly identify at least some permitted reorderings.

6.0 Related Work

IR induction operators borrow heavily from APL [Falkoff and Iverson, in Wexelblat]. They also appear in some later languages (e.g., Matlab, HPF) that may be better known in some communities.

Vcode [Blelloch and Chatterjee] is another IR that borrows from APL, but it differs from ours in many important ways. For example, Vcode targets the full range of modern supercomputers and thus supports far more vector operators, dynamic allocation of vectors, and even nested parallelism. It is a complete IR for vectors, where ours is among the smallest plausible subsets and can be of use even in less ambitious compilers for uniprocessors. Both ends of the spectrum merit study. Vcode is far more powerful, but ours might be easier to work into an existing C or Fortran compiler, because it is small, it adds no datatypes, and existing compiler analysis phases can identify and introduce its limited extensions.

7.0 Bibliography

Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems 11*(4):491-516, October 1989.

Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. *Processings of the Third Symposium on the Frontiers of Massively Parallel Computation*:471-480, 1990.

Adin D. Falkoff and Kenneth E. Iverson. The evolution of APL. In [Wexelblat], 661-674.

Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — Fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 4/92.

Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, second edition, 1990.

Stephen C. Johnson and Cleve Moler. Compiling Matlab. *USENIX Very High Level Languages Symposium*: 119-127, 1994.

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5*:308-325, 1979.

Richard L. Wexelblat. *History of Programming Languages*. Academic Press, 1981.