A Family of Data Parallel Derivations

Maurice Clint
Stephen Fitzpatrick
Terence J. Harmer
Peter L. Kilpatrick
Department of Computer Science
The Queen's University of Belfast
Belfast, Northern Ireland, UK

James M. Boyle
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, USA

This work is supported by SERC Grant GR/G 57970, by a research studentship from the Department of Education for Northern Ireland and by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38

Clarity V Efficiency

High-level, architecture-independent programs

- Easier to construct
- Easier to understand
- Portable

Efficient programs

- Tailored to particular machine: non-portable
- Awash with details
- Difficult to construct
- Difficult to understand

Example: Transpose of a Matrix

Definition: transpose A^T of $m{\times}n$ matrix A is an $n{\times}m$ matrix such that

$$\forall i,j : A^T[i,j]{\equiv}A[j,i]$$

High-level implementation

```
function transpose(A,m,n)
  = generate([n,m],fn(i,j)=>A[j,i])
```

Efficient sequential implementation for square matrix ($m = n$)

```
SUBROUTINE transpose(A,n)
DO i=1,n
  DO j=i+1,n
    t := A[i,j]
    A[i,j] := A[j,i]
    A[j,i] := t
  END
END
```

Our resolution

Programmer constructs specification and implementation *automatically* derived.

Specification language

Functional programming language

- Mathematically based
- Simple semantics: easily understood
- Useful mathematical properties
- Executable prototypes

Implementation language

Whatever required by implementation environment; usually version of Fortran or C.

- Efficient
- Good vendor support
- More convenient than machine language

Derivation by program transformation

Program Transformations

Program rewrite rules:

$$pattern \rightarrow replacement$$

All occurrences of $pattern$ in program changed to $replacement$.

- Achieves a small, local change
- Based on formal properties
  Clearly preserves meaning of program
- Formally defined in wide spectrum grammar
- Formal proof possible

Derivations

Sequences of transformations

- Complete metamorphosis through many applications of many transformations
- Automatically applied by TAMPR system

Family of Derivations

Derivation performed in steps

- *Sub-derivations*
- *Intermediate forms* between specification and implementation languages

For example:

$$\text{SML} \longrightarrow \lambda\text{-calculus} \longrightarrow \text{Fortran77}$$

Same intermediate form for:

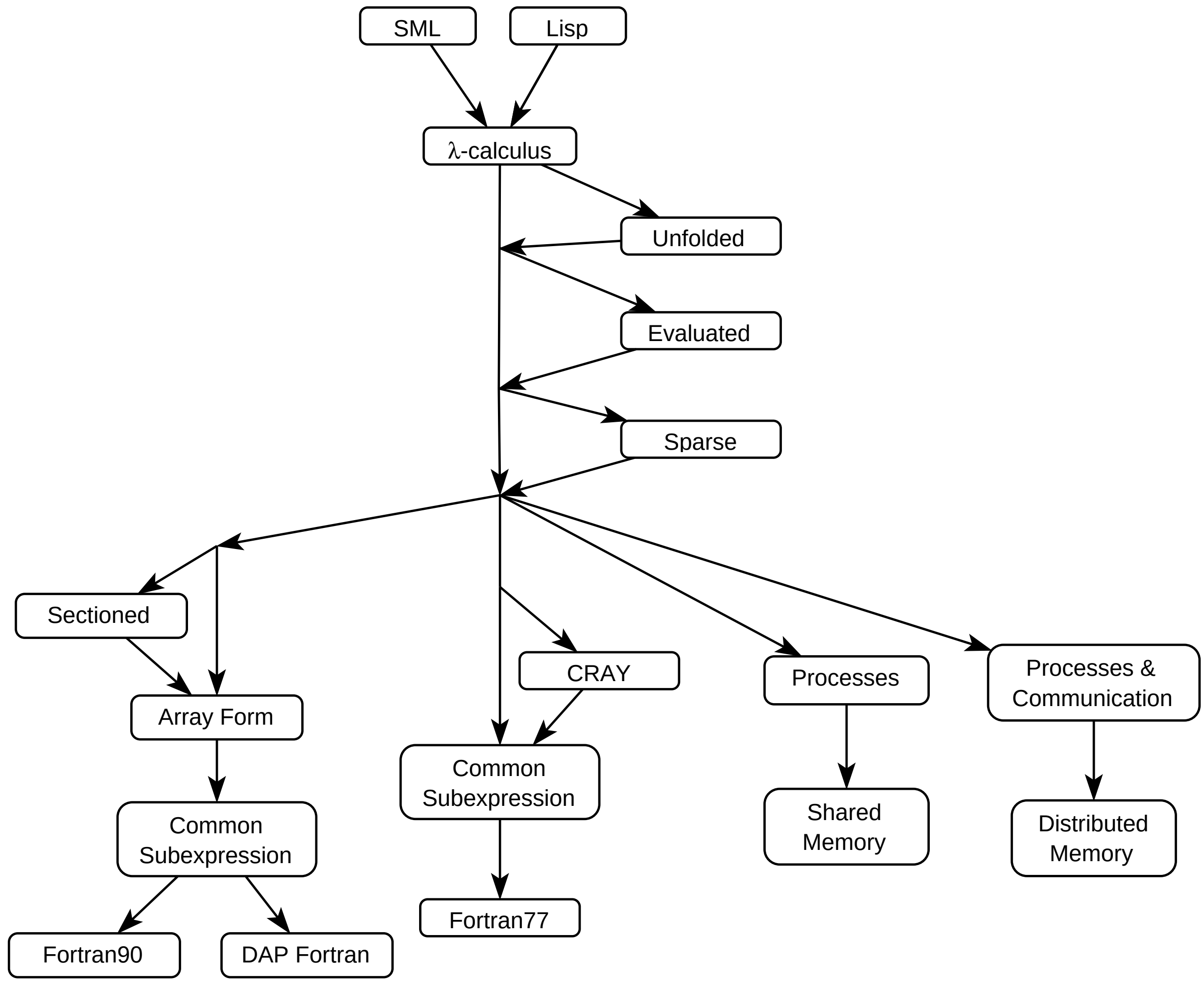
- other specification languages
- other architectures/implementation languages

Combinations have included:

$$\left.\begin{array}{c}\text{SML}\\ \text{Lisp}\\ \text{Miranda}\end{array}\right\} \longrightarrow \lambda\text{-calculus} \longrightarrow \left\{\begin{array}{c}\text{Fortran}\\ \text{CRAY Fortran}\\ \text{DAP Fortran}\\ \text{C}\end{array}\right.$$

Other sub-derivations/intermediate forms for:

- Optimization e.g.
  function unfolding
  common subexpression elimination
- Tailoring for particular forms of data
  e.g. sparse matrices



Example

Matrix-vector multiplication

$$\left[\begin{array}{cccc} & \cdots & & \\ & \cdots & & \\ \hline 1 & 2 & 3 & 4 \\ \hline & \cdots & & \\ & \cdots & & \\ & \cdots & & \end{array}\right] \times \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \left[\begin{array}{c} \cdots \\ \cdots \\ \hline 1a + 2b + 3c + 4d \\ \hline \cdots \\ \cdots \\ \cdots \end{array}\right]$$

```
fun times(U:vector,V:vector):vector
  = generate(size(U),fn(i:int)=>U@[i]*V@[i])
fun sum(U:vector):real
  = reduce(U,+,0.0)
fun innerproduct(U:vector,V:vector):real
  = sum(times(U,V))

fun mvmult(A:matrix,V:vector):vector
  = generate(size(A,0),
     fn(i:int)=>innerproduct(row(A,i),V))
```

SML specification

Data parallel functions

- `generate` defines vector/matrix
- `reduce` combines elements of vector/matrix into single value

Optimize:

```
generate([n],λi·
  reduce([n],λj·real.times(
    element(A,[i,j]),
    element(V,[j])),
  real.plus,0.0)
)
```

Sequential/CRAY implementation:

`generate` and `reduce` implemented as loops

```
DO i=1,n,1
 AV(i)=0.0
 DO j=1,n,1
  AV(i)=AV(i)+A(i,j)*V(j)
 ENDDO
ENDDO
```

DAP implementation: whole-array operations

```
AV=sumc(A*matr(V,n))
```

Assessment

Techniques have been applied to more complex algorithms for sequential,vector, array and shared-memory architectures.

Comparing with independent, manually constructed implementations:

- Derived implementations similar.
- Execution performance equal or better.

Techniques are being extended for yet more complex algorithms, for distributed and shared memory parallel architectures and for further special data structures.

With derivational approach, programmer

- develops implementation techniques
- encodes techniques as derivations

Reusability

Multiple specifications

Multiple implementations of each

Algorithm modified: modify specification and re-apply derivation

Extensibility

New optimization technique
or new architecture
or new data representation:
'slot in' new sub-derivation

Transferability

Sub-derivation requires no expertise to use

One programmer may use another's work

Correctness

Correctness of transformations implies correctness of implementation