A Subspace Optimizing Data Parallel Compiler

by

Todd O. Dampier

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May, 1994

Copyright Todd O. Dampier, 1994. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce
and to distribute copies of this thesis document in whole or in part,
and to grant others the right to do so.

Author

Department of Electrical Engineering and Computer Science

May 16, 1994

Certified by

J. Dally

Supervisor

Accepted by

F. R. Morgenthaler

Chairman, Department Committee on Graduate Theses
A Subspace Optimizing Data Parallel Compiler

by

Todd O. Dampier

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 1994

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Scientific programs with large data sets are an important class of computer application, requiring large amounts of memory and computational power. Massively parallel processing hardware and data parallel programming techniques are increasingly used to meet these requirements. A new approach to data parallel compilation, the Subspace compilation model, is introduced. This model is based on the idea that the shapes of data objects and how these shapes change represent higher-level performance considerations that the alignment of individual data elements. This model also removes the ad hoc restrictions of the prevalent Single Program, Multiple Data (SPMD) model. A Subspace compiler is designed based on the Subspace model, employing subspace trees as its program representation. A significant part of this compiler is implemented, with the CM-5 CM Fortran platform as its target. The implementation is tested on benchmark code, and the results discussed.

Thesis Supervisor: William J. Dally
Title: Associate Professor, M.I.T. Artificial Intelligence Laboratory
Acknowledgements

I would like to thank

Kathy Knobe, who planted and nurtured this thesis, provided constant and
sane guidance and support, and saw it through to the very end,

Bill Dally, who placed enough faith in me to let me go it alone,

William E. and Dolores C. Dampier, my parents, who footed the bill and called
every weekend,

Darrin Jewell, who kept me alive and brought me lunch, and

Brenda Pendleton, who believed it could be done, and made it so.
# Table of Contents

1. Introduction .............................................................................1

2. The Subspace Model ..................................................................3

3. The Subspace Tree Program Representation ...............................6
   3.1. Declarations ....................................................................7
   3.2. Object references..............................................................7
   3.3. Local operations...............................................................8
   3.4. Expansions......................................................................10
       Replication.....................................................................10
       Hopping.........................................................................11
       Scans............................................................................12
   3.5. Reductions ......................................................................13
   3.6. Control flow operations .....................................................14
   3.7. Missing nodes..................................................................15
   3.8. Summary of SST nodes .....................................................15

4. Compiler Overview....................................................................17
   4.1. The compilation target......................................................17
   4.2. Compiler design and implementation.................................18
   4.3. The Front End..................................................................19
   4.4. Natural Space Determination............................................20

5. Subspace Reconciliation.............................................................22
   5.1. An example fragment.......................................................22
   5.2. Owner computes..............................................................24
   5.3. Union of children.............................................................25
   5.4. Subspace minimization.....................................................26
       What are we minimizing, and how?.................................26
1. Introduction

Scientific programs with large data sets represent an important class of computer applications. Typically, these programs require large amounts of memory and computational power. One way of meeting these requirements is with parallel computers, employing multiple processors to cooperate on solving the same problem.

A prominent method of applying parallel computing to these problems is data parallelism. In data parallel computation, the work each processor performs is dependent upon the data held by that processor. A processor communicates with other processors only when it requires values held by those processors, or when it holds values those processors require. A figure of merit in data parallel computing is the required quantity of such communication.

One approach to data parallelism is the Single Program, Multiple Data (SPMD) model of compilation. This model has become quite popular, and is now almost synonymous with data parallelism [4]. However, the SPMD model has several arbitrary rules [9] that can lead to a significantly less than optimal translation of the program — that is, a translation which performs more communication and computation than is necessary. These rules are:

- For each statement in the code, a processor determines the role it will play in executing that statement based on the data it holds and the following two rules.
- A processor which owns a variable or array element computes all right hand sides of assignments to that location and stores their results. This is the owner computes rule.
- A processor which owns a variable or array element must communicate its value to any processor which needs that value to compute a right hand side.

The Subspace model of compilation attempts to abstract away from physical location of variables and iteration. Instead, it concerns itself with the dimensionality of data objects, as manifested not only in their declarations, but also in their use in expressions in iteration contexts.
The Subspace model is a different approach to data parallelism which removes some of the ad hoc restrictions present in the SPMD model. The Subspace model can uncover more parallelism in programs than is possible by direct application of the SPMD rules.

This thesis outlines a compiler that employs the Subspace model, and describes the construction of a significant part of such a compiler. This compiler uses subspace trees as its internal program representation, and will serve as a platform to experiment with the Subspace model and to learn about the performance benefits it enables.

The remainder of this document describes the Subspace model (Section 2), the subspace tree program representation (Section 3), the design and implementation of the Subspace compiler (Sections 4 through 7), and some benchmark experiments (Section 8). Sections 9 and 10 conclude and discuss directions for future research.
2. The Subspace Model

The Subspace model is a data parallel compilation model formulated around the shape, or dimensionality, of data [6]. The model is based on the idea that how data changes shape is a higher-level consideration than how data elements align. Accordingly, shape changes should be dealt with as a separate phase earlier in the compiler.

Large scientific applications of the kind the Subspace model is designed to benefit typically declare large arrays, and access their contents iteratively using loops. Many existing compilers, such as those employing the SPMD model, assign a processor to hold each declared unit of memory, and proceed from there. The Subspace model, on the other hand, is not so rigidly bound by the programmer’s declarations. A set of nested loops form a space — the iteration space — and the references to data objects within those loops occupy a subspace of that space, whose size and shape are not completely determined by the programmer’s variable declarations.

In the subspace model, each reference to an object has a natural shape determined by the context in which it is used. The shape of a definition is computed from the loops that surround it and the expression that defines the object. The shape of a use is computed from the shapes of the definitions that reach it. Thus, references to an object may have natural shapes that are different from the declared dimensions of the object, and that are different from the shape of the iteration space. A couple of examples suffice to demonstrate why this is a useful view to take.

\[
\begin{align*}
&\text{do } i = 1, \text{imax} \\
&\quad \text{do } j = 1, \text{jmax} \\
&\quad \quad a(i) = c(i) \ast d(j) \\
&\quad \quad \ldots \\
&\text{ enddo} \\
&\text{ enddo}
\end{align*}
\]

In this example, the natural shape of the right hand side of the assignment is two-dimensional, since it takes on potentially different values at every point in the \(ij\) plane. (We assume that the ellipsis here hides at least one use of \(a(i)\), so that all these values really do get used in the loop.) Therefore, the subspace model concludes that the natural shape of the left hand side is also two-dimen-
sional. If, as the programmer has written it, we were to use a one-dimen-
sional object as the target of the assignment, we would unnecessarily serialize
the computation along the missing axis (j in this case). That is, we would
have to wait until \( a(1) \) was computed before we could begin computing \( a(2) \),
even though there is no true dependence between the two locations.

\[
\begin{align*}
\text{do } & i = 1, \text{imax} \\
\text{do } & j = 1, \text{jmax} \\
\text{do } & k = 1, \text{kmax} \\
\text{a}(i,j,k) &= \text{c}(i) \times \text{d}(j) \\
\end{align*}
\]

Again in this example, the right hand side is two-dimensional, and so again
the natural shape of the left hand side is two-dimensional. Using a three-di-
mensional object as the target of the assignment might imply performing the
same computation \( \text{kmax} \) times (if we performed \( \text{c}(i) \times \text{d}(j) \) at every point
along the \( k \) axis, as the owner computes rule prescribes), and would certainly
imply more communication to expand either \( \text{c} \) and \( \text{d} \) or their product across
the extra axis. Figure 1 illustrates this situation.

\[\text{Figure 1. The target is too large. The target } a, \text{ as declared, is larger than the}
\text{object being assigned to it, } c \times d. \text{ There are two ways to assign } c \times d \text{ to all of } a. \]
\text{One is to create } c \times d \text{ in its natural space and enlarge it to fill } a. \text{ The other is}
\text{to enlarge both } c \text{ and } d \text{ to the size of } a \text{ and multiply them together. The}
\text{former wastes communication, the latter wastes computation and more}
\text{communication.}

Determining the natural shapes of references and the corresponding move-
ments of data between shapes has the effect of eliminating anti-dependences
and output dependences resulting from assignment targets of insufficient di-
dimensionality, as in the first example above. This optimization potentially benefits any target that exhibits parallelism, and only fails to benefit a completely sequential uniprocessor. Natural shape information also allows the elimination of unnecessary computation and communication resulting from assignment targets of excess dimensionality, as in the second example. This elimination of computation can benefit any target. Thus, these transformations within the subspace model have the advantage of being independent of the compilation target.
3. The Subspace Tree Program Representation

The compiler’s internal representation of the program, used by the subspace analysis phases, is a subspace tree (SST). A subspace tree program representation consists of a set of declarations together with a subspace tree expressing the computation to be performed. The subspace tree portion of the program representation contains nodes which can be broadly classified as object references and operations. Operation nodes can be further decomposed into local operations, expansions, reductions, and control flow nodes.

The following, for example, is a subspace tree program which is a translation of the examples from the last section. The first three lines contain the declarations, and the second three lines contain the subspace tree — a single assignment node — which is the body of the code.

```
(((parameter imax 100) (parameter jmax 100) (parameter kmax 100)
 (range i 1 imax 1) (range j 1 jmax 1) (range k 1 kmax 1)
 (sym a (i j) real) (sym c (i) real) (sym d (j) real))

{[i j] = exp ([i j] aref (a i j))

{[i j] * ([i j] rep ([j] ([i] aref (c i))))

{[i j] rep ([i] ([j] aref (d j)))))
```

Each subspace tree node, when fully annotated, contains a space in which it is interpreted, a node type (operator, or rator for short), and a list of arguments (operands, or rands). When it is omitted, the space is assumed to be null. The exact meaning of these fields depends on the type of node. The general template for a subspace tree node, then, is the following:

```
(s op (a1 ... an))
```

A general description of subspace tree nodes is found in the appendix of [7]. The rest of this section is devoted to describing the components of an SST program, with particular attention to how they are supported in the compiler. Section 3.1 discusses the declaration section, and Sections 3.2 through 3.6 discuss the various types of SST nodes. A summary of all node types is given in Table 1 at the end of this section.

Where code appears in italics in this section, it represents a Fortran analogue of the subspace tree code which follows it. The variables i, j, and k in such code are assumed to be the control variables of surrounding loops.
3.1. Declarations

There are three types of declarations in the declaration section of an SST program: parameters, ranges, and symbols.

A parameter declaration associates an identifier with a constant.

\[ \text{(parameter zero 0.0e0)} \]
\[ \text{(parameter imax 100)} \]

A range declaration associates an identifier with a range of integer values by specifying an initial value, a final value, and a stride. This name may be used as an axis in the spaces of SST nodes, or as an axis argument in those nodes that have such arguments. The three values that define a range are late-binding, and may represent different values at different points in a program. In the example below, for instance, the second range is defined in terms of the first. Thus, \( \{i, j\} \) would denote a two-dimensional, triangular-shaped space.

\[ \text{(range i 1 imax 1)} \]
\[ \text{(range j 1 i 2)} \]

A symbol declaration associates an identifier with an object of a given type and dimensionality. The set of valid types is that of Fortran 77. The dimensionality of the object is specified by a list of ranges. These ranges must have been previously defined in the declaration section, and must have used only constants and previously declared parameters in their definitions. For example, the declaration below declares a one-dimensional object with real-typed values.

\[ \text{(sym a (i) real)} \]

3.2. Object references

An object reference node represents a constant, a scalar, or an array reference. Constants and scalars have null spaces, since they only have a single value. Their single argument is their value, which is a constant or scalar variable name, respectively. The type of a constant is determined by the format of its argument, and the type of a scalar is determined by its entry in the declaration section. For example,
The space of an array reference serves two roles. First, it indicates the shape of the array reference, by indicating a set of axes along which the reference can take different values. Second, it binds the variables corresponding to these axes. Axis identifiers appearing in the space, when used in indices in the argument list, will assume all values contained in the corresponding range. Note that the axes, and therefore the space, are properties of the reference, not of the object, which may have been declared with a totally different set of axes.

The first argument in an array reference is the name of the array object, which must appear in the declaration section. The type of the array reference is the declared type of the array object. The remaining arguments are the indices of the reference. Each index is a simple expression which is an affine function of at most one axis identifier. In addition to the axis identifier, the index expression may involve constants, scalar variables, and the operations +, *, /, and abs.

\[
a(i, j+1, k)
\]

\[
((i \ j \ k) \ \text{aref} \ (a \ i \ (+ \ j \ 1) \ k))
\]

A \text{baref} is a more general form of array reference. It is like an \text{aref}, except it may include arbitrary expressions and even SST nodes as indices. This allows indirect and arbitrary references, such as references with vector-valued or array-valued subscripts. The compiler does not currently support this type of node.

\[
a(i, j+1, v(i))
\]

\[
((i \ j) \ \text{baref} \ (a \ i \ (+ \ j \ 1) \ ((i) \ \text{aref} \ (v \ i))))
\]

**3.3. Local operations**

Local operation nodes manipulate object references within a single space. A prominent local operation is the assignment node. The space indicates the space in which the assignment is performed; both arguments must be in this space. The first argument is the left hand side, or target, of the assignment.
It must be a scalar or an array reference. The second argument is the right
hand side, or value to be assigned. Its semantics require it to be an expression
SST. That is, it may contain no control flow or assignment nodes.

\[(s = (\text{lhs rhs}))\] ; General form.

\[a(i, j) = b(i+1, j)\]

\[\{i j\} = (\{i j\} \text{aref } (a i j))\]
\[\{i j\} \text{aref } (b (+ i 1) j))\]

Arithmetic operations such as addition and subtraction are also in the cate-
gory of local operations. Again, the space indicates the space in which the op-
eration is performed, and is required to be the space of all operands. All ar-
guments are expression SSTs. Some operators place restrictions on the num-
ber of arguments, whereas others do not. All arguments should evaluate to
numeric types. A complete list of currently supported operators is given in
Table 1. Some examples are given below.

\[(s \text{ op } (r_1 \ldots r_n))\] ; General form.

\[c(i) + d(i+2) + e(i)\]

\[\{i\} + (\{i\} \text{aref } (c i))\]
\[\{i\} \text{aref } (d (+ i 2))\]
\[\{i\} \text{aref } (e i))\]

\[\text{ABS}(a(i,j))\]

\[\{i j\} \text{abs } (\{i j\} \text{aref } (a i j)))\]

Logical and comparison operations are also supported as local operation
nodes. They are just like arithmetic operators, except they return logical-typed
values, and the argument expressions for logical operations should be of logi-
cal type. Some examples follow.

\[c(i,j) \text{ .EQ. } d(i,j)\]

\[\{i j\} \text{ eq } (\{i j\} \text{aref } (c i j))\]
\[\{i j\} \text{aref } (d i j)))\]

\[x(i) \text{ .LT. } y(i) \text{ .AND. } (.NOT. } b(i))\]
Finally, a particularly interesting local operation node is the \texttt{phi} operation \cite{7}. This operator represents a conditional selection between several alternatives. Its format is much like that of the LISP \texttt{cond} construct. All arguments must be in the space indicated by the space field. There must be an even number of arguments, organized in condition-consequent pairs. The former member of each pair should be a logical-type expression. When used as an expression, the latter member of each pair should be an expression SST containing no control flow or assignment nodes. The value of the \texttt{phi} node at a point in its space is the value of the consequent member of the first condition-consequent pair whose condition member is true at that point. The special logical value \texttt{T}, which is true at all points in all spaces, may be used as a default condition.

The value of the following example is a two-dimensional object containing at each point the lesser of the value of \( c \) at that point and the value of \( d \) at that point.

\[
\begin{bmatrix}
\{i\} \phi \begin{bmatrix}
\{i\} \{j\} \lt \begin{bmatrix}
\{i\} \{j\} \text{aref} (c \begin{bmatrix} i \{j\} \end{bmatrix})
\{i\} \{j\} \text{aref} (d \begin{bmatrix} i \{j\} \end{bmatrix}))
\{i\} \{j\} \text{aref} (c \begin{bmatrix} i \{j\} \end{bmatrix})
\{i\} \{j\} \text{aref} (d \begin{bmatrix} i \{j\} \end{bmatrix}))
\{i\} \{j\} \text{aref} (c \begin{bmatrix} i \{j\} \end{bmatrix})
\{i\} \{j\} \text{aref} (d \begin{bmatrix} i \{j\} \end{bmatrix}))
\{i\} \{j\} \text{aref} (c \begin{bmatrix} i \{j\} \end{bmatrix})
\end{bmatrix}
\end{bmatrix}
\]

In addition to its use as an expression as described here, the \texttt{phi} operation may be used as a control flow construct. This use of the operator is covered in Section 3.6 below.

\subsection*{3.4. Expansions}
Expansions are operations which create new objects by adding spatial axes to their arguments.

\textbf{Replication}

The simplest type of expansion is a replication. The value of the replication expansion is the result of making copies of the original object along the new axes. The general form of this node is

\[
(s \text{ rep } (k \text{ obj}))
\]

;General form.
Here, $s$ is the target space, or the shape of the resulting object. The SST $\text{obj}$ is the expression SST being replicated, and $k$ is the set of axes along which it is being copied. Thus, $s$ is the union of $k$ and the space of $\text{obj}$, and $k$ and the space of $\text{obj}$ should be disjoint. The example below shows a one-dimensional object being replicated to produce a three-dimensional object.

$$e(i,j,k) = x(i)$$

$$\{i \ j \ k\} = (((i \ j \ k) \ \text{aref} \ (e \ i \ j \ k))$$

$$\{i \ j \ k\} \ \text{rep} \ ((j \ k)$$

$$((i) \ \text{aref} \ (x \ i))))$$

**Hopping**

Another type of expansion is the hopping expansion. The hop node captures the semantics of a value depending on a previous value. Its general format is

$$(s \ \text{hop} \ (k \ \text{init-val} \ \text{hopping-form})) \ ; \text{General form.}$$

Here, $s$ is the target space of the expansion, and is also the space of the third argument, $\text{hopping-form}$. The first argument, $k$, is the expansion axis along which the hop is being performed. The result is formed by sequentially instantiating the $\text{hopping-form}$ at each point along the expansion axis. The special value (!) may be used within the $\text{hopping-form}$ to refer to the previous value in the expansion. The second argument, $\text{init-val}$, is the value of the resulting object at the lower bound of the expansion axis (i.e., at the point specified by the first value in the declaration of the corresponding range, whether or not it is the smallest value in the range), and is also the first value of (!). The space of $\text{init-val}$ is equal to $s$ with $k$ removed.

Consider the following Fortran fragment:

```fortran
do i = 1, imax
    a = a + x(i)
    ...
enddo
```

The natural shape of $a$ is one-dimensional, since it takes on $imax$ distinct values in the course of the loop. It can be expanded to its one-dimensional shape in the loop from its scalar shape at the entry to the loop by using a hopping expansion such as the following.
Of course, in this example, the relationship between the current value and the value at the previous iteration (i.e., at the previous point along the expansion axis) is very specific, and would in practice be implemented by an add-scan (described below). However, the generality of a hop node would be required for something less readily understood by the compiler, such as the following example, in which a two-dimensional object is formed by modifying a one-dimensional object a number of times in succession.

There is currently no support in the compiler for the hop node. It should be clear that any construct that requires a hopping expansion to be correctly expressed, can also be expressed using an iter construct (see Section 3.6 below). Since a hop necessarily serializes the computation it expresses, the only reason not to unify the two mechanisms in practice is the very specific, restricted form of dependence the hop captures. The compiler could not currently take advantage of that specificity. Nonetheless, we include the concept of the hopping expansion to more consistently solidify the relationship between spatial dimensions and the role of iteration.

**Scans**

Another type of expansion is a scan, a familiar motif in data parallel programming. It is different from other expansions, in that it does not add a new axis, but rather performs a computation along an existing axis. The notion is of accumulating under some operation the values of an object along one of its axes. The general format of a scan operation is

```
(s scan-op (k obj)) ; General form.
```
Here, $s$ is both the target space of the operation and the space of the source object $obj$. The first argument $k$ is the axis along which the scan is being performed. The operator scan-op is one of add-scan, mul-scan, max-scan, min-scan, and-scan, and or-scan. The result is formed by accumulating the elements of $obj$ along the axis $k$ under the appropriate operation.

Let us reconsider our first example from the discussion of hopping above. An add-scan node would be used to express the definition of $a$ in this example. Note that the reference to $a$ within the loop has a one-dimensional natural shape.

\begin{verbatim}
a = 0.0
do i = 1, imax
  a = a + x(i)
  ...
  ... = a ...
  ...
enddo
\end{verbatim}

We would represent the first assignment as

\begin{verbatim}
({i} = (({i} aref (a i))
  ({i} add-scan (i ({i} aref (x i)))))
\end{verbatim}

3.5. Reductions

Reductions create new objects by removing spatial axes from their arguments. In the following three types of reductions, $s$ is the space of the resulting object, $k$ is the axis of the object which is being removed, and $obj$ is the object, an expression SST, from which that axis is being removed. The space of $obj$ is $s$ with $k$ added to it, and $k$ should not exist in $s$.

The last-true reduction forms its result by taking the value of $obj$ at the last point along $k$ where its second argument, $b$, is true. The object $b$ should be in the same space as $obj$, and should be of logical type. The last-true node has the form

\begin{verbatim}
(s last-true (k b obj)) ;General form.
\end{verbatim}

This node would be useful, for instance, in the following example.

\begin{verbatim}
if b(i) then
  a = c(i)
endif
\end{verbatim}
The `any` reduction forms its result by nondeterministically choosing a point along \( k \), and taking the corresponding values of \( \text{obj} \) at that point.

\[
(s \text{ any } (k \text{ obj})) ;\text{General form.}
\]

The specific reduction (spec) node forms its result by choosing values from \( \text{obj} \) at a specific point along \( k \). This point is indicated by the second argument, \( \text{val} \). This argument may be in any subspace of \( s \), and should be an integer-typed object.

\[
(s \text{ spec } (k \text{ val } \text{ obj})) ;\text{General form.}
\]

### 3.6. Control flow operations

Control flow operations are subspace tree nodes which agglomerate smaller subspace trees, and which may control when or in what order they execute. Control flow nodes do not have return values associated with them, which makes them ineligible for use in expression SSTs.

The `iter` node provides a basic iteration construct. It sequentially executes its second argument, an arbitrary SST, at every point along the axis \( k \). Inside this body SST, \( k \) may appear as an axis in SST node spaces (though it will not increase the degree of those spaces) and as a scalar in expressions. The `iter` node has the form

\[
() \text{ iter } (k \text{ sst}) ;\text{General form.}
\]

The sequence and block construct group arbitrary SSTs into a single SST. They provide no guarantees regarding the execution order of their arguments, except that data availability constraints among them will be honored.

\[
() \text{ seq } (\text{sst}_1 \ldots \text{sst}_n) ;\text{General form.}
\]

\[
() \text{ block } (\text{sst}_1 \ldots \text{sst}_n) ;\text{General form.}
\]

As mentioned above, the `phi` construct can also be used as a control flow node. In this role, the logical expressions should be scalar, and the consequent SSTs may be arbitrary nodes. This is not implemented in the current compiler.
3.7. Missing nodes

There are a number of important omissions from this compiler’s catalog of SST nodes if a subspace tree is to be able to represent a typical application’s code. The nodes we’ve included are those we feel are necessary to represent common expressions and control flow. That is, these represent a set of features appropriate for a minimal, experimental compiler, not for a complete programming system.

Subroutine and function definitions, as well as calls to such modules, are not expressible in a subspace tree using these nodes alone. For the purposes of this research, a module’s code can be inlined whenever the module is called. Input and output primitives are also not included.

Features of Fortran such as COMMON blocks have no analogue in this program representation. Declarations, again, are only complex enough to capture the minimum expressiveness required.

3.8. Summary of SST nodes

Table 1 summarizes the various types of SST nodes that have been defined in this section.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>local</td>
<td>2</td>
<td>Assignment.</td>
</tr>
<tr>
<td>+, −, *, /</td>
<td>local, arithmetic</td>
<td>unlimited</td>
<td>Arithmetic operations.</td>
</tr>
<tr>
<td>abs</td>
<td>local, arithmetic</td>
<td>1</td>
<td>Arith. functions of 1 arg.</td>
</tr>
<tr>
<td>and, or</td>
<td>local, logical</td>
<td>unlimited</td>
<td>Logical combiners.</td>
</tr>
<tr>
<td>lt, le, gt, ge, eq, ne</td>
<td>local, logical</td>
<td>2</td>
<td>Arithmetic comparison ops.</td>
</tr>
<tr>
<td>not</td>
<td>local, logical</td>
<td>1</td>
<td>Logical negation.</td>
</tr>
<tr>
<td>phi</td>
<td>local</td>
<td>even</td>
<td>Conditional expression.</td>
</tr>
<tr>
<td>rep</td>
<td>expansion</td>
<td>2</td>
<td>Copying expansion.</td>
</tr>
<tr>
<td>hop</td>
<td>expansion</td>
<td>3</td>
<td>Serialized expansion.</td>
</tr>
<tr>
<td>add-scan, mul-scan, max-scan, min-scan</td>
<td>expansion, arithmetic</td>
<td>2</td>
<td>Arithmetic accumulation expansions.</td>
</tr>
<tr>
<td>and-scan, or-scan</td>
<td>expansion, logical</td>
<td>2</td>
<td>Logical acc. expansions.</td>
</tr>
<tr>
<td>last-true</td>
<td>reduction</td>
<td>3</td>
<td>Logical reduction.</td>
</tr>
<tr>
<td>any</td>
<td>reduction</td>
<td>2</td>
<td>Nondeterministic reduction.</td>
</tr>
<tr>
<td>spec</td>
<td>reduction</td>
<td>3</td>
<td>Specific value reduction.</td>
</tr>
<tr>
<td>iter</td>
<td>control flow</td>
<td>2</td>
<td>Iteration construct.</td>
</tr>
<tr>
<td>seq</td>
<td>control flow</td>
<td>unlimited</td>
<td>Aggregation construct.</td>
</tr>
<tr>
<td>block</td>
<td>control flow</td>
<td>unlimited</td>
<td>Aggregation construct.</td>
</tr>
<tr>
<td>phi</td>
<td>control flow</td>
<td>even</td>
<td>if-then-else flow control.</td>
</tr>
</tbody>
</table>

Table 1. Summary of SST nodes.
4. Compiler Overview

This thesis describes the construction of a significant part of a compiler that translates Fortran 77 code into Connection Machine (CM) Fortran code, using subspace trees as the internal program representation and supporting the Subspace compilation model. Thus, the compiler performs a parallelizing source-to-source program transformation. Some phases of the compiler are simulated for the purposes of this research.

The remainder of this section discusses this choice of target, and the overall design of the compiler. The simulated phases are sketched also. Subsequent sections will discuss the implemented phases in greater detail.

4.1. The compilation target

The CM-5 and CM Fortran were selected as the target platform because they constitute a widely used and well supported system [10]. Using this target facilitates measurement and meaningful comparison of benchmark examples, without necessitating the development of an entire commercial quality compiler, or the use of any ad hoc scaling factors to compensate for the effects of different architectures.

This choice of target also serves to demonstrate a point about the subspace model: it is an early, high-level phase of a parallel compiler, and can be used in addition to traditional analyses. A fully fleshed-out subspace compiler would begin with a front end, include a subspace phase which determines shapes and shape changing operations, and then would follow this with analyses which considered alignment of data, and mapping onto a physical processor set, in addition to standard sequential program optimizations. By performing a source-to-source translation, we leverage the traditional analyses already in place in an established parallel compiler.

Unfortunately, using CM Fortran as a target prevents full realization of the potential speedups offered by Subspace analysis. Subspace analysis removes dependences, so that expressions at each particular data point can be computed as soon as their operands are ready at that data point. However, CM Fortran imposes a globally synchronized programming model that does not allow full
advantage to be taken of this fact. CM Fortran turns out to pose other difficulties for this research as well. These are discussed further in Section 8.

4.2. Compiler design and implementation

The overall structure of the compiler is illustrated in Figure 2. Modules indicated with dashed lines are those which are simulated rather than implemented in this thesis work.

The subspace tree utilities and all the phases of the compiler except the front end are implemented in Common LISP, which is well suited to traversing and manipulating the list-oriented, recursive structure of the SST representation described in the previous section.

In the full design, the front end parses the Fortran source and produces a naive SST representation of the program. This naive SST is a closely literal translation, and contains only iteration nodes and local operation nodes, none of them annotated. The natural space determination phase performs the analysis necessary to determine the natural spaces of the object references (leaves) in this SST, and thereby annotates them. It eliminates iteration and serialization where possible, and inserts the communication (expansion and reduction) nodes necessary to replace them. These phases are currently simulated by hand, but their design will be sketched in the remainder of this section.

The subspace reconciliation phase propagates the leaf annotations created by the previous phase to all the nodes in the SST, reconciling them when subtrees in disparate spaces must be combined by a single operation. Inserting replications when they're required in reconciliation expands the operands from their natural spaces to their useful spaces.
The final two phases are target dependent. The temp allocation phase identifies and assigns to temporary variables any subexpression trees which prevent their parent from being translated into a single construct in the target language. Finally, the back end takes the result of this phase and produces a program in the target language.

The design and implementation of these final three compiler phases are described in detail in the following three sections. A complete listing of the source code for these phases is given in Appendix B.

4.3. The Front End

The front end phase accepts a Fortran 77 file as input, and produces a naive subspace tree representation of the program. This representation has no more parallelism than the source program; the program is still expressed in terms of iterations and object references as they appeared in the source. As such, this subspace tree has no space annotations, and contains only local operation nodes and control flow nodes.

For the purposes of this research, this phase is simulated by hand-translation of the input file.

There are two main challenges facing an actual front end implementation. The first is the notorious difficulty of writing an elegant parser for Fortran, especially one complete enough to accept all the constructs of the language as they are specified. [2]

One solution to this challenge is to use an established, sufficiently general Fortran parser. This approach has the advantage that many of the needed software modules already exist, have proven themselves elsewhere, are independently tested and maintained, and have a user community stabilizing their development.

Several grammars and parsers for Fortran exist in the public domain. One of the more promising among these is the Sage++ package developed at the University of Indiana [3]. This package parses Fortran source into a program representation data structure, and provides access to this representation through a set of C++ classes and methods. The interface is well-designed and
flexible, allowing the construction of arbitrary compiler tools. A front end that is little more than a parser seems like a trivial use of these capabilities.

The other challenge is deciding on an appropriate translation of the Fortran constructs. This is mostly a design challenge. Some constructs, such as do loops and assignments, are straightforward to translate into the set of SST nodes described in the previous section. As mentioned in that section, some constructs, such as subroutines, calls, and common blocks, require extensions to SST program representation. These extensions vary in their degree of difficulty. For instance, a CALL seems straightforward, but a distinction must be made between a call that can be made independently at each point in a space, and one that may not. Some constructs, such as arbitrary GOTO statements, are extremely difficult to translate and analyze in the context of the subspace model. These should probably just be forbidden. The solidification of this part of the compiler design will be resolved largely in the decisions made in designing the front end.

4.4. Natural Space Determination

Natural Space Determination, in the most basic view, is the phase of the compiler which performs the conversion of the program from scalar sequential code, to parallel code. It accepts as input a subspace tree whose nodes have no space annotations, and include only local operation nodes and control flow nodes.

This phase produces a semantically equivalent subspace tree whose leaves are annotated with their natural spaces, and which contains expansion and reduction nodes necessary to create objects in their natural shapes as they are referenced. Thus, for example, an iter may be completely eliminated, and replaced with assignments involving scans and hops to effect the expansions it produced. This phase also introduces the multiple internal objects that may be used to represent a single source object, and inserts the code necessary to manage them.

The subspace analyses required to perform this transformation are described in detail in [7], and only briefly summarized here.

The natural space of a definition is the union of the natural spaces of the references used in defining it. In this definition of a
\[ a(i, j, k) = c(i) + d(j) \]

if we know that the reference to \( c \) has natural space \{i\}, and the reference to \( d \) has natural space \{j\}, we can conclude that the reference to \( a \) has natural shape \{i j\}, the union of the two spaces occurring on the right hand side.

The natural space of a reference is determined by the natural shapes of the definitions that reach it.

\[
\text{if } b \text{ then } \\
\quad a(i, j, k) = c(i) \\
\text{else } \\
\quad a(i, j, k) = c(i) \times d(j) \\
\text{endif} \\
\text{s} = a(i, j, k) \times 3.0
\]

In this example, making the same assumptions about references to \( c \) and \( d \), the first definition of \( a \) has natural space \{i\}, and the second \{i j\}. Both of these definitions potentially reach the use of \( a \) in the last statement, so its natural space is \{i j\}.

If a reference is part of a cycle of references carried by a loop, the axis corresponding to the loop must be added to the reference's space. For instance, in the following code,

\[
\text{do } i = 1, \text{imax} \\
\quad s = s + 1.0 \\
\text{enddo}
\]

both the definition and the use of \( s \) would seem to have natural space \{\}. However, because of the dependence carried by the \( i \) loop, \( s \) takes on a different value for every value of \( i \). The axis \( i \) must be added to the natural space of every reference in this cycle, which for this example is simply both references to \( s \).

The Natural Space Determination module is omitted from the scope of this research, and its effect on the program tree is simulated in this research by manual translation and annotation.
5. **Subspace Reconciliation**

The Subspace Reconciliation phase begins with a subspace tree whose leaves are annotated with their natural spaces as determined by the Natural Space Determination phase. These leaves are exactly the object reference nodes of the subspace tree.

This phase is responsible for determining the space annotation of each operation node. As these spaces are assigned, this phase also inserts replications to expand operands to the space assigned to their parent operation. This is called the *useful space* of the operand.

How an operation node's space is assigned represents a policy decision. Three such reconciliation policies are considered: owner computes, union of children, and subspace minimization. Section 5.1 introduces a code fragment which will be used to illustrate the action of the subspace reconciliation phase. The details of the three reconciliation policies and their implementations within the compiler are discussed in the Sections 5.2 through 5.4. The final subsection of this section considers these policies in comparison to loop invariant code motion, a related classical optimization.

### 5.1. An example fragment

Consider the following statement within a Fortran loop nest.

```fortran
  do i = 1, imax
    do j = 1, jmax
      do k = 1, kmax
        ... 
        c(i, j, k) = c(i, j, k) + s1 * a(i, j) * s3 * b(i, j)
        ... 
      enddo
    enddo
  enddo
enddo
```

This is not unlike statements found in many typical Fortran codes. If it seems a bit unlike the way a Fortran programmer would express herself, we might consider this fragment to be the result of a transformation based on natural spaces. For instance, $a$ and $b$ might be scalars that have been promoted to two-dimensional objects because their natural space is two-dimensional.
Let us assume that the natural space of $s_1$ and $s_3$ is null, since they are scalar; that the natural space of $a$ and $b$ is $\{i, j\}$; and that the natural space of $c$ is $\{i, j, k\}$.

Shown below is the subspace tree program representation of this fragment, annotating only the leaves (i.e., object references) of the tree with their natural spaces. This is the form of the program presented as input to the subspace reconciliation phase. Note that the operation nodes have no space annotations.

(((range i 1 50 1)  
  (range j 1 60 1)  
  (range k 1 70 1)  
  (sym s1 () real)  
  (sym s3 () real)  
  (sym a (i j) real)  
  (sym b (i j) real)  
  (sym c (i j k) real)  
)  
(= ({{i j k} aref (c i j k)})  
  (+ ({{i j k} aref (c i j k)})  
    (* (* (* ((({{} scalar (s1)}  
              ({{i j} aref (a i j)}))  
              ({{} scalar (s2)}))  
              ({{i j} aref (b i j)}))))))))

The subspace tree portion of this program — a single assignment node — is depicted graphically in Figure 3. Boldface text near a tree node represents that node's space annotation.

![Figure 3. Fragment SST with leaf annotations.](image)

We will revisit this example as we consider the three subspace reconciliation policies that assign useful spaces to these operation nodes.
5.2. Owner computes

Under the *owner computes* policy, the space of an operation node is the natural space of the left hand side of the assignment in which it is involved (has as an ancestor). That is, every operation in an expression is performed in the space of the "owner," or target, of the assignment of which that expression is a part. This policy takes its name from the SPMD rule of the same name [9], which dictates that all values participating in an assignment must travel to the processor which holds the target of the assignment, and the operations on them must occur there.

The implementation of this policy is correspondingly simple. The reconciliation procedure traverses the program tree. For each assignment node, it annotates the assignment node with the space of the target of the assignment. It then traverses the descendents of the assignment node — the expression nodes of the "right hand side" — also annotating them with the space of the target. For each leaf in the right hand side tree, it inserts the necessary replication node to expand that leaf to the space of the operation it’s involved in, which is the space of the assignment’s target. Thus, only leaves are replicated.

Figure 4 shows the result of reconciling the example fragment under this policy. Note that all nodes are now annotated with a space, and that replications occur only at the leaves. There are four replications in all, across a total of eight new axes.

![Figure 4. Fragment SST with owner computes reconciliation.](image-url)
5.3. Union of children

Under the *union of children* policy, the space of an operation node is the union of the spaces of its children (operands). This is exactly the natural space of the operation.

This policy might be viewed as the owner computes policy applied at a finer grain: at the operation level rather than at the assignment level. That is, each operation is an "owner" of its operands, whereas under the owner computes policy, only assignment nodes are eligible to be "owners."

To implement this policy, the reconciliation procedure traverses the SST. For each leaf node, it returns the space with which the leaf is already annotated. For each assignment or expression node, it recursively calls itself on each SST child of that node to determine its space. It then computes the space that is the union of the spaces of the children, inserts the appropriate replication nodes to expand each SST child to this space, and makes this the space of the node.

Figure 5 shows the result of reconciling the example fragment under the union of children policy. Note that there are three replication nodes, they occur throughout the tree, and they expand their operands across a total of five new axes.

![Figure 5. Fragment SST with union of children reconciliation.](image-url)
5.4. Subspace minimization

The subspace minimization policy allows rearrangement of expression trees to create operations with the smallest possible natural spaces. These natural spaces are then assigned as the useful spaces. Thus, this policy can be viewed as the union of children policy, applied after rearrangement of the expression tree.

What are we minimizing, and how?

How the expression tree is rearranged is the key to the optimization performed under this reconciliation policy. First, it is necessary to make clear what we are trying to optimize, and the assumptions underlying this goal. Based on the idea that communication is the greatest expense in a parallel program, we assume that, to the first order, the communication associated with an expression determines its execution time. The communication that we have the ability to affect in the subspace reconciliation phase is that arising from replication. We assume that replication along any axis incurs unit expense in terms of communication — we say it incurs a replication cost of 1. Thus, our goal is to minimize each expression's replication cost by minimizing the number of axes across which replications take place in an expression.

The smallest space an operation can be performed in is the union of the spaces of its operand children. However, commutative, associative operation nodes with more than two operands can be decomposed into trees whose internal nodes are all that operation, and whose leaves are the operands of the original operator. Figure 6 shows such a transformation.

![Figure 6](image)

Figure 6. A decomposition of a single commutative, associative operation (a) into a tree of operations with the same leaves (b).

This kind of transformation can be used to reduce the replication cost of the expression. This is accomplished by grouping the operands such that the min-
imum number of expansions are required. Figure 7 shows an example of using this transformation to this effect. In this figure, the edges of the expression tree are annotated with the replication cost they incur.

![Diagram](image)

Figure 7. Decomposing the operation in (a), which has a replication cost of 4, can result in a tree (b) with a replication cost of 2.

**Minimization of an operation — the marble game**

But what is the minimum replication cost of an expression tree? Answering this question entails the entire problem of assigning a space to every non-leaf node. Let us answer a simpler version: What is the minimum replication cost of an operation whose operands’ spaces are known?

For an operation node with one or two children, or whose operator is not both commutative and associative, the answer is trivial: the replication cost is the total number of axes which operands must be replicated across in order to expand each operand to the union of the operand spaces.

For a commutative, associative operation node with more than two children, the compiler is free to group the operands in any way. So what is the minimum-cost grouping? We can view the problem in the following way: Each operand begins in a space, and must eventually “reach” the target space — which is still the union of all the operand spaces — through a series of replications. Along the way, it may be combined with other operands, expanded or in their natural space, under the operation. Only the expansions incur a unit of replication cost. We will evolve an algorithm for solving this problem through an example.

Consider an addition with operands in \{i\}, \{j\}, and \{i j k\}. Note that it is not important how many operands are in each space; adding operands in the same space is free in our model. The target space of the operation is \{i j k\}, so the operands in \{i j k\} do not need to expand. This initial situation is shown in
Figure 8(a) as a graph whose vertices are subspaces of the target space, and whose edges connect spaces which are separated by an expansion across a single axis. Dark "marbles" rest in spaces where there are operands, and a light "footprint" is left by a marble when, its expansion decided, it leaves a space along an edge.\(^1\) When two marbles are met in a space, they combine into one (the operands are added together). When a marble lands in a space where there is a footprint, it can "follow" the footprints — that is, the operand will be added with whatever operands formerly existed in that space.

To make a step toward the solution, we choose any marble, consider all its possible expansions, and select one which yields the cheapest solution. In Figure 8(b), we choose the marble at \{i\}, for which there are two possible expansions, \{i j\} and \{i k\}. The latter will yield a solution requiring four expansions, whereas the former will yield three; the former is selected. In Figure 8(c), we choose the marble which entered \{i j\} in the last step; we select its only possible expansion, to \{i j k\}. In this space it will be added with the operands originally in \{i j k\}. Finally, in Figure 8(d), we choose the only remaining marble, in \{j\}. As in step (b), from its two possible expansions, we choose the one yielding the three-replication solution, \{i j\}. In \{i j\}, this operand is added with the operand formerly occupying \{i j\}, which is in fact the operand we expanded into \{i j\} from \{i\} in step (b).

The pattern of expansions which remains when all the marbles are gone, is the pattern which produces the grouping of operands with the minimum replication cost. In this example, the remaining pattern indicates that the optimal grouping is that of the following expression.

\[
((i \ j \ k) + (ijk-rand

((i \ j \ k) \ rep \ ((k)

((i \ j) + (((i \ j) \ rep \ ((j) \ i-rand))

((i \ j) \ rep \ ((i) \ j-rand)))))))))
\]

\(^1\) No, marbles don't have feet, and so don't leave footprints. The metaphor must be indulged. We considered using Boy Scouts finding their way to a campsite, but stayed with marbles to avoid the possible associations.
Expression coalescing

It seems like we can formulate this marble game into a formal algorithm, which in fact we will do shortly. But even so, we have only answered the limited question of how to minimize a single operation, not the full question of how to minimize a whole expression.

Considering a single operation at a time is sufficient if nothing can be gained by looking at the expression tree more deeply or more broadly. The algorithm which implements this reconciliaton policy, before playing the marble game on a per-operation basis, coalesces each expression tree. The goal of coalescing the tree is to maximize its branching factor, so that it will offer the most opportunities for optimized grouping of operands.
The coalescing algorithm accomplishes this goal by finding addition operators which have addition operations as operands. The operands of the child additions are made part of the operands of the parent addition. In searching for nodes which match this pattern, it considers a subtraction to be an addition of its first operand and the negatives of its other operands. Addition is not the only commutative, associative operation; the coalescing algorithm handles multiplications the same way as additions (and, in this context, divisions the same way as subtractions, with reciprocation taking the place of negation).

Only expression semantics constrain the transformations the coalescing algorithm can perform on the expression tree. In addition to the regrouping it currently performs, it might be extended to perform factoring or apply the distributive law of multiplication over addition. However, either of these transformations can potentially increase or decrease the minimum replication cost of the expression, whereas regrouping can only decrease this cost, as Figure 9 shows. To find the truly minimal version of an expression, the coalescing algorithm would have to speculatively regroup or factor wherever possible, compute the cost of the resulting expression, and choose the version which turned out least expensive. This was deemed too time-consuming an optimization for the limited benefits it might provide.
Dynamic programming for the marble game

The marble game above was a good illustration of the problem of finding a grouping of arguments with a minimum replication cost, but it leaves a lot to be desired as a description of an algorithm. The problem of the marble game clearly has a recursive structure, as we are left with a new marble game after each move. Also, the part about “consider all its possible expansions, and select one which yields the cheapest solution” suggests the solution to the smaller marble problem guides the choice of expansion.

This structure leads us to formulate the marble game — that is to say, the problem of finding the minimum replication cost grouping for the operands of an associative, commutative operation — as a dynamic programming algorithm [5].

We can formulate the minimum replication cost of an operation formally as $c[M, F]$, where $M$ is the set of spaces for which there are operands in the ex-
pression whose expansions are as yet undecided (the marbles), and $F$ is the set of spaces whose expansions are decided (the footprints). We always place a footprint in the target space, since we know what to do with any operands which end up there. Thus, the expansion cost of an expression with operands in spaces $m_1, m_2, ..., m_n$ and target space $t$ is $c[[m_1, m_2, ..., m_n], [t]]$.

This function is defined recursively as follows, where $v$ is an arbitrary member of $M$, and the function $\text{parents}(s)$ for any space $s$ returns the set of spaces which can be reached in a single-axis expansion from $s$.

$$c[M, F] = \begin{cases} 
0 & \text{if } M = \emptyset \\
\{v\in F\} & \text{if } v \in F \\
\min_{p \in \text{parents}(v)} c[M \cup \{p\} - \{v\}, F \cup \{v\}] & \text{otherwise}
\end{cases}$$

The first condition in this formula occurs when there are no marbles left. This corresponds to all expansions having been decided, in which case, no additional expansions are needed. This is the base case.

The second condition occurs when an operand exists in a space whose expansion is already decided — a marble is in the same space as a footprint. In this case, no additional expansions need take place in order for that operand to arrive at the target space; it can be combined with the other operands as they pass through this space, and the result expanded in the same manner as the operand which left the footprint was. Thus, the operand is eliminated from the set of spaces whose expansions must be decided. This is called the operand merge case.

The third and most interesting condition, called the operand expansion case, corresponds to making a move in the marble game. Given an operand in space $v$, it finds the optimal expansion for the operand by finding the cost corresponding to each possible expansion (the recursive uses of $c$), and selecting for $v$ the expansion which produces the cheapest solution. The recursive invocations of $c$ have the prospective parent space (to which the marble may move) added to $M$, and the marble's current space taken away from $M$ and added to $F$ (as it will leave a footprint in $v$ when it moves). Moving the marble incurs a replication cost of one expansion axis over the cost of the problem which is left after the marble is moved.
The operand expansion case must record \( v \) and the expansion target selected for it in a data structure. This data structure corresponds to the arrows of Figure 8, and is used in constructing the grouped expression after the cheapest solution is found.

**The subspace minimization algorithm**

From the elements we have now discussed, it is relatively straightforward to construct the complete algorithm which implements the subspace minimization reconciliation policy.

This reconciliation algorithm traverses the program tree. When it encounters an assignment node, it coalesces the expression on the right hand side of the expression, as discussed above. It then reconciles the coalesced expression in the following way: For each node in the expression tree which is not one of the commutative, associative operators, the procedure is the same as with the union of children policy. For each commutative operator node, the operands are first reconciled, and then the dynamic programming algorithm described above is invoked. This algorithm returns the data structure representing the optimal expansion pattern. From this data structure, a correctly reconciled expression tree corresponding to that optimum operand grouping is produced.

Figure 10 shows the total effect of this reconciliation policy applied to the example fragment from above. Note that only two replication nodes have been inserted, expanding their operands across a total of only three axes. Also note that the expression tree has been rearranged, and one of the multiplications has three operands.
5.5. Comparison with loop invariant code motion

As a way of understanding the optimizations represented by these reconciliation policies, it is instructive to compare them to a similar type of optimization, loop invariant code motion.

Invariant code motion (IVCM) is an optimization which hoists expressions out of as many enclosing loops as possible. An expression can be hoisted out of a loop enclosing it if its value does not vary from one iteration of that loop to the next (i.e., is invariant within that loop). This kind of optimization is typical within existing compilers [1].

The union of children and the subspace minimization policies completely subsume the type of optimization performed by IVCM. IVCM can eliminate some combinations of unnecessary “axes” from a computation, but cannot always eliminate all of them. From within a nest of \(N\) loops, there are \(N-1\) places to which an expression can be hoisted, as the hoisting must be considered loop by loop in the order they are nested. That is to say, there are \(2^{N-1}\) combinations of axes that can be eliminated from the computation of the expression.

In contrast, performing a computation in its natural space will always eliminate all unneeded axes. In other words, there are \(2^N-1\) combinations of axes that can be eliminated. An example demonstrates.
In this example, the second term in the RHS sum is invariant with respect to both the $i$ and the $k$ loop. IVCM can move the second term out of the $k$ loop, but then cannot move it past the $j$ loop:

```fortran
do i = 1, imax
  do j = 1, jmax
    do k = 1, kmax
      a(i, j, k) = a(i, j, k) + b(j) * c(j)
    enddo
  enddo
enddo
```

However, in the subspace model, we observe that the multiplication operation has a one-dimensional natural space (with $j$ being its axis). Thus, both of the irrelevant "loops" are eliminated in computing it.

The argument might be made, at least for this example, that the same effect could be achieved using loop invariant code motion together with loop reordering transformations. While this is true, we believe the subspace view is more elegant, omitting considerations of loop bounds and other details.
6. Temp Allocation

The Temp Allocation phase accepts a fully-annotated subspace tree, and breaks up complex expressions in this tree for easier digestion by the Back End. In particular, this phase allocates temporary variables for holding subexpression computations, and breaks off branches of expression trees to assign to these variables in a statement-level assignment. The broken-off branches are replaced by uses of the corresponding temporary variable. This is intended to be a “smart” phase which breaks up the trees in such a manner that the following Back End can be straightforward. It allows the Back End to make the simplifying assumption that a single tree can be translated as a single construct (statement) in the target language.

Thus, temp allocation is the first target-specific phase of the compiler, as it must contain knowledge of what can and cannot be included in a single expression in the target.

The temp allocation routines traverse the subspace tree supplied as an argument, searching for expression nodes which require separate assignments to temporaries. Each routine returns a replacement for the tree it is called on — the original tree if there are no substitutions, otherwise a tree containing new temporary variables — together with a list of assignment nodes which must precede the assignment node containing that tree. The assignments in this list are those which give the temporaries their values, and the list is empty if no substitutions were performed. Any new temporaries are also added to the list of declarations in the SST program. Below are the three cases which require temporaries for the CM Fortran target:

A scan cannot be an argument in an expression; it may only occur as the “right hand side” argument of an assignment. This is because scans are translated into CALLs to CM Fortran Utility Library routines, which take as arguments a source and a target array. The “left hand side” must be able to be given as the target array argument. Thus, when a scan is found, it is assigned to a temporary, which is then substituted in its place. Temp allocation is invoked on the argument being expanded in the scan node.
A scan cannot have an expression as its argument; it can only expand an object reference. Again, this restriction arises from the fact that the scan will be translated into a CM Fortran CALL, in which the source of the expansion must be able to be supplied as the source array argument. When the argument to a scan is not an object reference, the argument expression is assigned to a temporary, and the temporary used in its place. Temp allocation is invoked on the argument being assigned to the temporary.

A rep, like a scan, cannot have an expression as its argument. The CM Fortran SPREAD intrinsic, into which all reps are translated, only allows an array section to be expanded. Thus, when the argument to a scan is not an object reference, the argument expression is assigned to a temporary, and the temporary used in its place. Temp allocation is invoked on the argument being assigned to the temporary.

Returning to our example code as it was after subspace minimization in Section 5.4, we observe that there are two places where the third pattern listed above — a rep with an expression as an argument — exists. Thus, two temporary variables are introduced: rep_temp_164 and rep_temp_163. Thus, the original fragment, which was a single assignment node, becomes three assignment nodes, grouped in a block. These assignments are shown in Figure 11.
A possible improvement that might be made to the temp allocation phase is to enable the reuse of its temporary variables. Temporary variables are live from their single assignment to their single use, and since both this definition and this use arise from a single expression, it is guaranteed that there will be no dependence between them carried by any loop. By virtue of this information, a temporary of a particular size and shape could be reused in a subsequent expression. Thus, a single routine would only need a number of temporaries proportional to the complexity of its expressions (greatest number of temporaries required to be simultaneously live), rather than to its length.

In addition to the slight complexity it would add to the compiler, part of the rationale for not incorporating this optimization was the assumption that the CM Fortran compiler would detect the short-lived variables and perform the optimization itself. This assumption may not be justified.
Common subexpression elimination might also be performed in this phase, but is not in the current compiler implementation for similar reasons.
7. The Back End

The Back End is the final phase of the compiler, and as such is closely tied to the compilation target. It accepts as input a fully annotated subspace tree containing expressions that are easily translated into the target language.

The compilation target in this research is Connection Machine Fortran for the CM-5 [10]. This section describes the structure of the back end constructed for this target.

The back end routines are built on top of an output apparatus tailored to the formatting requirements of Fortran. These output routines can correctly create continuation lines and provide an interface which supports column-oriented program formatting and pretty printing.

Since the compiler does not yet support nodes for transfer of control between procedures and functions, or for definitions of separate program modules, the SST program passed to the back end is translated as a single routine — the main routine of the Fortran program that is the output of this phase.

There are two sections in the back end phase. The first deals with the declarations section of the SST program. For each sym declaration, the corresponding identifier is declared with the indicated type, dimensionality, and array extents. Furthermore, if the symbol corresponds to an array, a CM Fortran compiler directive is generated which specifies that the array should reside in CM, rather than front-end, memory. For each range declaration, an integer variable with that name is declared, in case it needs to be used as an index variable in a loop across the elements of that axis.

The second section of the back end phase deals with the SST portion of the SST program. The subspace tree input is translated in a single pass to a valid CM Fortran program, which is written to a file. There are two main issues of interest in performing this translation. The first issue is how the various nodes are translated — what CM Fortran constructs or utility library calls are used. The second issue, which is related to the first, is how the idea of an SST node's space is handled. That is, how does the SST come to have a value at every point in its space?
Many of the nodes, such as arithmetic operations, have straightforward translations in CM Fortran. Others are less obvious. As mentioned in the previous section, scan operations are translated into CALLs to routines in the CM Fortran Utility Library. Replication nodes are translated into SPREAD intrinsic functions. The phi node is translated using the Fortran WHERE construct. Furthermore, since WHERE constructs cannot be nested, whereas phi nodes can, the back end keeps track of the controlling conditions, and combines them with logical operations to “flatten” this type of nesting.

Where possible, the computation of an expression’s value at every point in its space is represented using Fortran 90 array notation. For example, the SST assignment

\[
\{ij\} = (\{ij\} \text{aref} (a \ {ij}) \{ij\} \text{aref} (b (+ i 2) \ j))
\]

could be translated into the following Fortran assignment, if \(i\) is declared as a range from 1 to 10 and \(j\) declared as a range from 1 to 20:

\[
A(1:10:1,1:20:1) = \& B(3:12:1,1:20:1)
\]

There are some cases in which such a clear translation is not possible. Consider, for example, the following assignment.

\[
\{ij\} = (\{ij\} \text{aref} (a \ {ij}) \{ij\} \text{aref} (b \ j (+ i 2)))
\]

Since the the two sides of a Fortran array assignment must conform in shape and extent, this cannot be translated as an assignment using array notation, since the LHS is a 10x20 object and the RHS is a 20x10 object. In cases like these, the back end uses the CM Fortran FORALL construct. It would translate the above example as follows.

\[
\text{forall } (I=1:10:1, j=1:20:1) \\
& A(I,J) = \\
& B(J,I+2)
\]

The final result of compiling the example fragment from the previous two sections — that is, the subspace minimized version — is given below.
One other detail that one might notice when looking at this program is the calls to the routines whose names begin with “cm_timer”. The back end has the ability to time regions of code using the CM timer facility. Currently, the entire body of any compiled code is timed; however, provisions exist for using the timers at a finer grain.

program synth

INCLUDE '/usr/include/cm/CMF_defs.h'

integer I
integer J
integer K
REAL S1
REAL S3
REAL A(1:50, 1:60)
CMF$ LAYOUT A( :NEWS, :NEWS )
REAL B(1:50, 1:60)
CMF$ LAYOUT B( :NEWS, :NEWS )
REAL C(1:50, 1:60, 1:70)
CMF$ LAYOUT C( :NEWS, :NEWS, :NEWS )
REAL REP_TEMP_1264(1:50, 1:60)
CMF$ LAYOUT REP_TEMP_1264( :NEWS, :NEWS )
REAL REP_TEMP_1265

call cm_timer_clear(0)
call cm_timer_start(0)
REP_TEMP_1265 =
& ( S3
& * S1)
REP_TEMP_1264(1:50:1,1:60:1) =
& ( B(1:50:1,1:60:1)
& * A(1:50:1,1:60:1)
& * SPREAD(SPREAD(REP_TEMP_1265,1, (1+ABS((50-1))),
& 2, (1+ABS((60-1))))
C(1:50:1,1:60:1,1:70:1) =
& ( C(1:50:1,1:60:1,1:70:1)
& + SPREAD(REP_TEMP_1264(1:50:1,1:60:1),3, (1+ABS((70-1))))

call cm_timer_stop(0)
call cm_timer_print(0)

end
8. Benchmark Experiments

The main thrust of this research is to develop a platform for exploiting and experiment- ing with the Subspace data parallelism model. As such, the construction of the platform itself is the paramount task. However, some quantifiable experiments are included to provide incentive and evidence that this endeavor is indeed interesting.

The remainder of this section describes the experiments, discusses the results for a small program, and finally discusses the results from a moderately sized routine taken from the NAS Parallel Benchmark suite [8].

8.1. Experimental design

These experiments consist of changing compiler parameters, and observing the effect on the execution times of benchmarks. The parameter that will be varied is the choice of policy within the Subspace Reconciliation phase. We choose from among the three possibilities enumerated in Section 5: (O) Owner computes, (U) union of children, and (M) subspace minimization. This produces three sets of experimental outputs:

- (O) is our representative of the code produced by a naive parallelizing compiler. No loop optimizations have been performed, and all expressions have been assigned a useful space via the Owner Computes rule.

- (U) employs a straightforward version of the Subspace model, performing all intermediate computations in their natural shapes.

- (M) is more aggressive, and tries to improve over (U) by rearranging expression trees.

Putting this information together, we would thus expect to find, in execution time, that

\[(M) \leq (U) \leq (O).\]

That is, in general, we would expect that subspace analysis can be leveraged into better performance in some cases, and in the worst case will not degrade it.
8.2. Example fragment

To begin our experiments with a small, easily understood program, let us turn one last time to the code fragment we fabricated in Section 5, and which has served as an example ever since.

We have seen this example numerous times, in all three versions as produced by the compiler's three subspace reconciliation policies. In the previous section, we saw one version's generated code, including CM timer commands, ready to be submitted to the CM Fortran compiler (cmf) and run on the CM-5.

Table 2 contains the execution times of the three versions of this program, run on a CM-5 with 32 processors in a timesharing configuration. The numbers in the table are the CM elapsed time and the CM busy time as reported by the CM timer functions included in the CM Fortran libraries. According to the documentation, the former represents total system time expended by both front end host and CM (i.e., not wall time), and the latter represents only time spent in CM cycles. The numbers in each case are the average of five trials, where each trial contains a hundred iterations of the program body. The "busy time" figures are much more reproducible, with each set of trials having a standard deviation below 0.002 seconds; the elapsed time trials have standard deviations just below 0.03 seconds.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Elapsed time (sec)</th>
<th>CM Busy time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Computes</td>
<td>0.665</td>
<td>0.553</td>
</tr>
<tr>
<td>Union of Children</td>
<td>0.365</td>
<td>0.286</td>
</tr>
<tr>
<td>Subspace Minimization</td>
<td>0.351</td>
<td>0.286</td>
</tr>
</tbody>
</table>

Table 2. Example fragment execution times for 100 iterations.

This small benchmark seems to bear out our earlier predictions. The two more aggressively optimizing policies produce code that runs about a factor of two faster, in "busy" time, than the owner computes version. However, the subspace minimized candidate does not seem to do any better than the union of children version, even though each successive degree of optimization saves us a single replication — from four replications under owner computes down to two replications under subspace minimization — and at least two expansion
axes — from eight axes under owner computes, to five under union of children, to three expansion axes under subspace minimization.

8.3. NAS Benchmarks routine

The second, and much larger, code fragment used for the purposes of this experimentation is a routine taken from the NAS suite of parallel benchmarks. The source is the JACZ subroutine in the APPSP benchmark. The Fortran code for the original routine, the hand-translated SST version with leaf annotations, and the three results of compiling under the various subspace reconciliation policies can all be found in Appendix A.

According to the profile gathered by gprof, the JACZ routine is responsible for about 5% of the execution time of the APPSP benchmark on a Sparc workstation, and as such is one of the four program modules in which the most time is spent. The routine contains a great number of expressions combining three-dimensional arrays and scalar variables. Subspace analysis reveals that these references have natural shapes ranging from zero- to three-dimensional. As such, it would seem a prime target for the optimizations that can take place in the subspace reconciliation phase.

As in the previous experiment, execution times were collected from five sets of trials on the 32 processor CM-5 in a timeshared environment, again performing one hundred iterations of the program body in each trial. The averages of the results obtained are given in Table 3. Again, the busy time figures are more consistent, with standard deviations below 0.01 seconds; the elapsed time data sets have standard deviations in the neighborhood of 0.6 seconds.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Elapsed time (sec)</th>
<th>CM Busy time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Computes</td>
<td>3.087</td>
<td>1.984</td>
</tr>
<tr>
<td>Union of Children</td>
<td>3.678</td>
<td>2.273</td>
</tr>
<tr>
<td>Subspace Minimization</td>
<td>3.803</td>
<td>2.229</td>
</tr>
</tbody>
</table>

Table 3. JACZ subroutine execution times, 100 iterations.

These result fly in the face of the expected outcome and of the results obtained for the smaller example. The more we optimize, the longer the program seems to run! What is causing this?
One observation is that larger data sets seem to lessen the differences in execution times among the three versions. In the JACZ runs above, all array dimensions had extent 12 (NX = NY = NZ = 12). Increasing these numbers to 72 (i.e., by a factor of 6), produces the CM busy time figures in Table 4. This effect might suggest there is a high cost to initiating communication, but that the subspace optimized versions have less expensive communication patterns once this initial cost is paid. This would mean that extremely large data sets would be required for the Subspace model’s assumptions concerning communication cost to be valid.

<table>
<thead>
<tr>
<th>Policy</th>
<th>NX=NY=NZ=12</th>
<th>NX=NY=NZ=72</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner Computes</td>
<td>1.984</td>
<td>184.975</td>
</tr>
<tr>
<td>Union of Children</td>
<td>2.273</td>
<td>200.777</td>
</tr>
<tr>
<td>Subspace Minimization</td>
<td>2.229</td>
<td>200.880</td>
</tr>
</tbody>
</table>

Table 4. Effect of data set size on JACZ subroutine execution time. There are 100 iterations of the program body in each trial. Units are seconds.

Another observation is that the CM Fortran compiler does best with a programming idiom other than the one that allows the Subspace back end to express the specific desired expansions. Consider the following two fragments of CM Fortran code.

```fortran
REP_TEMP_388 = 5.0 * DSSP * DT
C(2:NX-1:1,2:NY-1:1,2) =
& C(2:NX-1:1,2:NY-1:1,2)
& + SPREAD(SPREAD(REP_TEMP_388, DIM=1, NCOPIES=NX-2),
& DIM=2, NCOPIES=NY-2)
forall (I=2:NX-1, J=2:NY-1)
& C(I,J,2) = C(I,J,2) + 5.0 * DSSP * DT
```

The first fragment is a slight simplification of a line taken from the subspace minimized version of the JACZ routine. Note the use of the temporary variable containing a scalar product, and the Fortran 90 SPREAD functions which explicitly replicate this to create a two-dimensional version of that product.

The second fragment performs exactly the same assignment to C, but uses the CM Fortran FORALL construct. It is impossible to explicitly dictate the how a data object expands in this construct; the semantics of the construct evaluate a
scalar statement at all points in a space specified by the controlling ranges. Thus, this construct is not very useful to the Subspace back end.

For \( NX = NY = 12 \), one thousand iterations of the first fragment consume 0.513 seconds of CM busy time. For the same data size, one thousand iterations of the second fragment consume 0.116 seconds of CM busy time. It seems clear that there is better support in the cmf compiler for the second kind of construct. This seems natural, since it is the sort of code a CM Fortran programmer would be more likely to write; but it is unfortunate for the Subspace compiler.

A final observation, closely related to the previous one, is that the semantics CM Fortran make it a bad fit as the Subspace compiler's target. We have already seen how using SPREAD to specify expansions incurs execution time penalties (an issue more of compiler implementation than of semantics). But looking at the example above suggests another problem: We had to use a temporary variable and an extra assignment statement to express our intentions in the first fragment, because a SPREAD accepts only a single identifier as its source argument. We would like a semantics that allows us to expand an expression. An ideal compiler would recognize that it was really expanding the expression on the right hand side of the previous assignment, but even so, this information cannot be explicitly expressed to the cmf compiler by the programmer (or our back end). And in practice, the cmf compiler seems to perform many of its optimizations on the statement level, rather than between statements; the assignments to temporaries are the most significant factor in slowing down the "optimized" versions.

A final possible explanation for the execution time effect of "optimization" in the reconciliation phase, is that the changes that the Subspace compiler makes in the flow and expansion of data actually lengthen the critical path of this program in some sense that is meaningful to its compilation model. For instance, if it creates parallel execution of SPREADs where it can, and the Subspace compiler is introducing dependences through its temporary variables, then a longer critical path is created.

Looking at the "listing" file generated by the cmf compiler — a digested summary of the compilation — it is indeed the case that the union of children and
subspace minimized versions of the program, when compiled with cmf, contain more CM communication primitives (SEND and BLOCKMOVE) than does the owner computes version. Yet reducing the program to a single statement which produces communication in one version and not in the other, can cause this phenomenon to vanish in some cases. That is, when the program becomes small, the results are similar to those of the small benchmark of Section 8.2. This would seem to lend credence to the hypothesis that the cmf compiler is somehow batching and/or parallelizing its communication and computation.

There are two main conclusions to be drawn from these observations and speculations. The first is that CM Fortran does not bear a close enough resemblance to the machine model for which the Subspace compilation model is designed, both in terms of its semantic expressiveness and in terms of the implementation of its compiler. Second, a high level language with a complex compiler is very likely not the best target for an experimental compiler, since the actions of the complex compiler can easily obscure the intent of the compiler being developed. This would seem clear from the difficulty that arises in trying to understand the performance variation in these relatively simple benchmark programs.


9. Conclusions

In the course of this research, a portion of a compiler was constructed which translates subspace tree programs with annotated leaves into correct CM Fortran code. This compiler uses subspace tree data structures and routines, performs some of the stages of subspace analysis — namely, reconciling the spaces of expressions and replicating operands appropriately — and lowers subspace tree constructs to a particular target, CM Fortran.

A complete set of subspace tree nodes is not yet supported by the compiler — more are needed to correctly express typical application code — but enough are present to express basic expression computations and control flow. This apparatus represents enough to perform experiments to further understanding of optimizations on basic code idioms.

Quantitative experiments were performed to evaluate the type of subspace analysis performed in this compiler: reconciliation. These experiments validated the assumptions underlying the reconciliation algorithms for small programs, but the results were inconclusive for a larger block of code, with the optimizations resulting in longer execution times.

The ambiguity of these results, and the frustration encountered in the attempt to understand what phenomenon was really behind these numbers, might suggest that the methodology of this research is questionable. In particular, it may not be a good idea to build a compiler which uses another sophisticated compiler as a further back end, at least if one of the goals is to gain a better understanding of the optimizations being introduced in the new compiler. Either better understanding of the complex compiler being employed, or complete implementation of a new compiler and run time system which embody the assumptions underlying the innovations, is probably necessary to perform thoughtful research and obtain meaningful results.

In conclusion, the software developed in this research will provide a useful nucleus for a more complete Subspace compiler. However, the value of the ideas behind a Subspace compiler have yet to be proved by quantitative experiment.
10. Future Work

The Subspace model provides an elegant framework for thinking about data parallel programs, and the implementation of a compiler which embodies it is a worthwhile ongoing task. Further work in this line of research can be broken down into three areas: refinement of this software, completion of the compiler, and a development more complete Subspace compiler.

The current partial compiler can be refined primarily by the addition of the missing nodes mentioned in Section 3.7. These nodes — especially some minimal support for subroutines and functions — are necessary to represent serious application code. Also, the existing compiler can be refined by more intelligently choosing the target constructs it uses to represent the program — striving for run time performance rather than simply correctness.

Completion of this compiler involves creating these missing phases, hinted at in the dashed lines of Figure 2. A front end must be present to parse Fortran 77 — and even Fortran 90 or HPF — into subspace tree form. And algorithms must be present to determine the natural spaces of nodes, inserting along the way the necessary merges, expansions, and reductions of objects, and recognizing the patterns of communication (e.g., scans) present in the code. This work is already underway.

Finally, a more complete Subspace compiler is probably worthwhile — one which includes a temp allocation phase and a back end targeted to a custom run time system which is implemented on top of innovative hardware. The MIT J-Machine, a lower level interface to the CM-5, and the forthcoming MIT M-Machine suggest themselves as possible targets for this effort.
Appendix A: NAS Benchmark Routine

This appendix contains the code for the larger benchmark used in Section 8, the JACZ subroutine of the APPSP benchmark from the NAS Parallel Benchmarks. What follows is the original Fortran 77 code, the hand-translation to SST program form, and the CM Fortran output of the Subspace compiler under each of the three reconciliation policies.

Original Fortran 77 source

```fortran
subroutine jacz ( m )
c c***form the zeta-direction pentadiagonal system.
c c Author: Sisira Weeratunga
 c NASA Ames Research Center
 c (10/25/90)
c c include 'appsp.incl'
c c dimension cv(isiz3), aa(isiz3), rhos(isiz3)
c c print *, "Entering jacz"
OPEN(UNIT=20, FILE='jacz-arrays', STATUS='replace',
& FORM='formatted')
WRITE(20, *)cl, c2, c3, c4, c5, dt, dssp, u, dzl,
& dz2, dz3, dz4, dz5
CLOSE(UNIT=20)
c print *, "Done with file dump."
c r43 = 4.0d+00 / 3.0d+00
c34 = c3 * c4
c1345 = cl * c3 * c4 * c5
c if ( m .eq. 3 ) then
  sn = 0.0d+00
else if ( m .eq. 4 ) then
  sn = 1.0d+00
else if ( m .eq. 5 ) then
  sn = -1.0d+00
end if
do j = 2, ny - 1
do i = 2, nx - 1
```
do k = 1, nz

c  
  rul = 1.0d+00 / u(1,i,j,k)
  uu = rul * u(2,i,j,k)
  vv = rul * u(3,i,j,k)
  ww = rul * u(4,i,j,k)

c  
  q = 0.50d+00 *( uu**2  
                 + vv**2  
                 + ww**2 )

c  
  cv(k) = ww
  aa(k) = sqrt ( c1 * c2 * ( rul * u(5,i,j,k) - q ) )

c  
  rhos(k) = max ( dz1,  
                 dz2 + c34 * rul,  
                 dz3 + c34 * rul,  
                 dz4 + r43 * c34 * rul,  
                 dz5 + c1345 * rul )

c  
end do

c  
  a(i,j,1) = 0.0d+00
  b(i,j,1) = 0.0d+00
  c(i,j,1) = 1.0d+00
  d(i,j,1) = 0.0d+00
  e(i,j,1) = 0.0d+00

c  
end do

c  
  a(i,j,nz) = 0.0d+00
  b(i,j,nz) = 0.0d+00
  c(i,j,nz) = 1.0d+00
  d(i,j,nz) = 0.0d+00
  e(i,j,nz) = 0.0d+00

c  
***fourth order dissipation

c  
  c(i,j,2) = c(i,j,2) + dt * dssp * ( + 5.0d+00 )
  d(i,j,2) = d(i,j,2) + dt * dssp * ( - 4.0d+00 )
  e(i,j,2) = e(i,j,2) + dt * dssp * ( + 1.0d+00 )

c  
  b(i,j,3) = b(i,j,3) + dt * dssp * ( - 4.0d+00 )
  c(i,j,3) = c(i,j,3) + dt * dssp * ( + 6.0d+00 )
  d(i,j,3) = d(i,j,3) + dt * dssp * ( - 4.0d+00 )
  e(i,j,3) = e(i,j,3) + dt * dssp * ( + 1.0d+00 )

c  
end do

c  
  a(i,j,4) = a(i,j,4) + dt * dssp * ( + 1.0d+00 )
  b(i,j,4) = b(i,j,4) + dt * dssp * ( - 4.0d+00 )
\[ c(i,j,k) = c(i,j,k) + dt \times dssp \times ( + 6.0d+00 ) \]
\[ d(i,j,k) = d(i,j,k) + dt \times dssp \times ( - 4.0d+00 ) \]
\[ e(i,j,k) = e(i,j,k) + dt \times dssp \times ( + 1.0d+00 ) \]

```
c   end do
c   a(i,j,nz-2) = a(i,j,nz-2) + dt \times dssp \times ( + 1.0d+00 )
b(i,j,nz-2) = b(i,j,nz-2) + dt \times dssp \times ( - 4.0d+00 )
c(i,j,nz-2) = c(i,j,nz-2) + dt \times dssp \times ( + 6.0d+00 )
d(i,j,nz-2) = d(i,j,nz-2) + dt \times dssp \times ( - 4.0d+00 )
c   a(i,j,nz-1) = a(i,j,nz-1) + dt \times dssp \times ( + 1.0d+00 )
b(i,j,nz-1) = b(i,j,nz-1) + dt \times dssp \times ( - 4.0d+00 )
c(i,j,nz-1) = c(i,j,nz-1) + dt \times dssp \times ( + 5.0d+00 )
```

```
c   end do
c   end do
c
return
eend
```
(sym uu (i_decl j_decl k_decl) real)
(sym vv (i_decl j_decl k_decl) real)
(sym ww (i_decl j_decl k_decl) real)
(sym q (i_decl j_decl k_decl) real)

; cv, aa, rhos should probably have one_to_isiz3 instead of k_decl.
(sym cv (i_decl j_decl k_decl) real)
(sym aa (i_decl j_decl k_decl) real)
(sym rhos (i_decl j_decl k_decl) real)

(seq 
  ; This stuff is set in the main routine by reading from a file.
  (= ((scalar (nx)) (number (12))))
  (= ((scalar (ny)) (number (12))))
  (= ((scalar (nz)) (number (12))))
  (= ((scalar (dt)) (number (1.5e-2))))

  ; This stuff is actually in the jacz routine.
  (= ((scalar (r43)) (/ (number (4.0)) (number (3.0))))))
  (= ((scalar (c34)) (* ((scalar (c3)) (scalar (c4))))))
  (= ((scalar (c1345)) (* ( (scalar (c1)) (scalar (c3)))))
      (scalar (c4))))
      (scalar (c5))))))

  ;; We need to add .EQ. as an operator!!!!
  (= ((scalar (sn)) (phi ((.eq. ((scalar (m)) (number (3))))
                      (number (0.0))))
      (number (1.0)))
      (number (5))))
      (- (number (1.0))))))

  (= ((scalar (sn)) (number (1.0))))
  (= ((i j k) aref (rul i j k))
      (number (1.0)))
  (= ((i j k) aref (u1 i j k))))

  (= ((i j k) aref (uu i j k))
      (i j k) aref (u2 i j k))))

  (= ((i j k) aref (vv i j k))
      (i j k) aref (v1 i j k))))

  (= ((i j k) aref (ww i j k))
      (i j k) aref (w1 i j k))))

  (= ((i j k) aref (q i j k))
      (number (0.50))
      (+ ( ( (i j k) aref (uu i j k))
         (i j k) aref (uu i j k)))
      (+ ( (i j k) aref (vv i j k))
         (i j k) aref (vv i j k)))
      (+ ( (i j k) aref (ww i j k))
         (i j k) aref (ww i j k))))))

  (= ((i j k) aref (cv i j k))
      (i j k) aref (cv i j k)))))

  ; Should also add ** operator! but for now...
  (= ((i j k) aref (q i j k))
      (* (number (0.50))
         (+ ( ( (i j k) aref (uu i j k))
             (i j k) aref (uu i j k)))
         (+ ( (i j k) aref (vv i j k))
             (i j k) aref (vv i j k)))
         (+ ( (i j k) aref (ww i j k))
             (i j k) aref (ww i j k))))))

  (= ((i j k) aref (cv i j k))
      (i j k) aref (cv i j k))))))

  ; Must add SQRT operator!
  (sqrt ( (scalar (c1))
      (scalar (c2))
      ( (i j k) aref (rul i j k))
      (i j k) aref (rul i j k))))

  ; Must add MAX operator!
  (max ( (scalar (dz1))
      (scalar (dz2))
      ( (i j k) aref (rul i j k))
      (i j k) aref (rul i j k))))

); This stuff is actually in the jacz routine.
)}
; (* ((scalar (c34)) (i j k) aref (rul i j k))))
; (+ ((scalar (dz4))
; (* ((scalar (r43)) (scalar (c34))
; ((i j k) aref (rul i j k))))
; (+ ((scalar (dz5))
; (* ((scalar (c1345)) (i j k) aref (rul i j k))))
; )))
; End of first k loop

= (((i j k) aref (a i j 1)) (number (0.0)))
= (((i j) aref (b i j 1)) (number (0.0)))
= (((i j) aref (c i j 1)) (number (1.0)))
= (((i j) aref (d i j 1)) (number (0.0)))
= (((i j) aref (e i j 1)) (number (0.0)))

= (((i j k2) aref (a i j k2)) (number (0.0)))
= (((i j k2) aref (b i j k2))
- (*((- ((scalar (dt)))
(scalar (tz2))
(+ (((i j k2) aref (cv i j (- k2 1))))
 (* ((scalar (sn)) ((i j k2) aref (aa i j (- k2 1))))))))
))
= (((i j k2) aref (c i j k2))
+ ((number (1.0))
(* ((scalar (dt))
((i j k2) aref (rhos i j k2))
(scalar (tz1))))))
= (((i j k2) aref (d i j k2))
- (* ((scalar (dt))
(scalar (tz2))
(+ (((i j k2) aref (cv i j (+ k2 1))))
 (* ((scalar (sn)) ((i j k2) aref (aa i j (+ k2 1))))))))
))
= (((i j k2) aref (e i j k2)) (number (0.0)))
; End second k loop.

= (((i j) aref (a i j nz)) (number (0.0)))
= (((i j) aref (b i j nz)) (number (0.0)))
= (((i j) aref (c i j nz)) (number (1.0)))
= (((i j) aref (d i j nz)) (number (0.0)))
= (((i j) aref (e i j nz)) (number (0.0)))

= (((i j) aref (c i j 2))
 (* ((i j) aref (c i j 2))
 (* ((scalar (dt)) (scalar (dssp)) (number (5.0))))))
= (((i j) aref (d i j 2))
 (* ((i j) aref (d i j 2))
 (* ((scalar (dt)) (scalar (dssp)) (number (-4.0))))))
= (((i j) aref (e i j 2))
 (* ((i j) aref (e i j 2))
 (* ((scalar (dt)) (scalar (dssp)) (number (1.0))))))

= (((i j) aref (b i j 3))
 (* ((i j) aref (b i j 3))
 (* ((scalar (dt)) (scalar (dssp)) (number (-4.0))))))
= (((i j) aref (c i j 3))
 (* ((i j) aref (c i j 3))
 (* ((scalar (dt)) (scalar (dssp)) (number (6.0))))))
= (((i j) aref (d i j 3))

55
Compiler output under Owner Computes

program jacz

INCLUDE '/usr/include/cm/CMFdefs.h'

integer ONE TO 5
integer ONETOISIZ1
integer ONE TO ISIZ2
integer ONETO ISIZ3
integer I DECL
integer JDECL
integer KDECL
integer I
integer J
integer K
integer K2
integer K3
REAL C1
REAL C3
REAL C4
REAL CS
REAL D2
REAL D2
REAL D2
REAL D2
REAL D2
REAL D2
REAL D2
REAL D2
REAL C34
REAL C34
REAL C1345
REAL RHOS(2:11,2:11,1:12)
REAL VV(2:11,2:11,1:12)
REAL D(1:12,1:12)
REAL B(1:12,1:12)
REAL U(1:5,1:5)
REAL DZ5
REAL DZ4
REAL DZ2
REAL DZ1

CV(I,J,K) = WW(I,J,K) * SPREAD(SPREAD(SPREAD(TZ2, 1, (1+ABS(((NX-1)-2))),(1+ABS(((NY-1)-2)))), 3, (1+ABS(((NZ-1)-2)))), 2, (1+ABS(((NY-1)-2)))), 1, (1+ABS(((NZ-1)-2))))

forall (I=2:(NX-1):1,forall (J=2:(NY-1):1,forall (K=1:NZ:1))

SN
SN

call cm__timer start (0)
call cm timer clear(0)
program jacx

include '/usr/include/cm/cm_defs.h'
include cm_lib.h
include cm_common.h
include cm_timer.h

integer ONE_TO_5
integer ONE_TO_121
integer ONE_TO_122
integer ONE_TO_123
integer ONE_TO_124
integer ONE_TO_125
integer 1
integer J
integer K
integer I

REAL C1
REAL C3
REAL C4
REAL C5
REAL DZ1
REAL DZ2
REAL DZ3
REAL DZ4
REAL DZ5
REAL DT
REAL DSSP
REAL R43
REAL C34
REAL C1345
REAL SN
REAL U(1:5,1:12,1:12,1:12)

cmf$ layout u(:news, :news, :news, :news)

REAL A(1:12,1:12,1:12)

cmf$ layout a(:news, :news, :news)

REAL B(1:12,1:12,1:12)

cmf$ layout b(:news, :news, :news)

REAL C(1:12,1:12,1:12)

cmf$ layout c(:news, :news, :news)

REAL D(1:12,1:12,1:12)

cmf$ layout d(:news, :news, :news)

REAL E(1:12,1:12,1:12)

cmf$ layout e(:news, :news, :news)

REAL RUI(2:11,2:11,1:12)

cmf$ layout rui(:news, :news, :news)

REAL UU(2:11,2:11,1:12)

cmf$ layout uu(:news, :news, :news)

REAL VV(2:11,2:11,1:12)

cmf$ layout vv(:news, :news, :news)

REAL WW(2:11,2:11,1:12)

cmf$ layout ww(:news, :news, :news)

REAL Q(2:11,2:11,1:12)

cmf$ layout q(:news, :news, :news)

REAL CV(2:11,2:11,1:12)

cmf$ layout cv(:news, :news, :news)

REAL AA(2:11,2:11,1:12)

cmf$ layout aa(:news, :news, :news)

REAL RHOS(2:11,2:11,1:12)

cmf$ layout rhos(:news, :news, :news)

REAL REPTEMP 646
REAL REP_TEMP_645
REAL REP_TEWMP 644
REAL REP_TEMP_643
REAL REP_TEMP_642
REAL REP_TEMP_641
REAL REPTEMP_640
REAL REP_TEMP_639
REAL REPTEMP_638
REAL REP_TEMP_637
REAL REP_TEMP_636
REAL REPTEMP_635
REAL REP_TEMP_634
REAL REPTEMP_633
REAL REPTEMP_632
REAL REP_TEMP_631
REAL REPTEMP_630
REAL REP_TEMP_629
REAL REP_TEMP_628
REAL REPTEMP_627
REAL REP_TEMP_626
REAL REPTEMP_625
REAL REP_TEMP_624
REAL REPTEMP_623
REAL REP_TEMP_622
REAL REPTEMP_621
REAL REP_TEMP_620
REAL REPTEMP_619
REAL REP_TEMP_618
REAL REPTEMP_617
REAL REP_TEMP_616
REAL REPTEMP_615
REAL REP_TEMP_614
REAL REPTEMP_613
REAL REP_TEMP_612
REAL REPTEMP_611
REAL REP_TEMP_610
REAL REPTEMP_609
REAL REP_TEMP_608
REAL REPTEMP_607
REAL REP_TEMP_606
REAL REPTEMP_605
REAL REP_TEMP_604
REAL REPTEMP_603
REAL REP_TEMP_602
REAL REPTEMP_601
REAL REP_TEMP_600
REAL REPTEMP_599
REAL REP_TEMP_598
REAL REPTEMP_597

call cm_timer_clear(0)
call cm_timer_start(0)

nx = 12
ny = 12
nz = 12
dt = 0.015

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  uu(i,j,k) = ru(i,j,k) * u(2,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  vv(i,j,k) = rui(1,j,k) * u(3,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  ww(i,j,k) = ru(i,j,k) * u(4,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  qq(i,j,k) = rui(1,j,k) * u(5,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  cv(i,j,k) = rui(1,j,k) * u(6,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  aa(i,j,k) = rui(1,j,k) * u(7,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  rep(i,j,k) = rui(1,j,k) * u(8,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  ws(i,j,k) = rui(1,j,k) * u(9,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  rsi(i,j,k) = rui(1,j,k) * u(10,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  rep(i,j,k) = rui(1,j,k) * u(11,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  ws(i,j,k) = rui(1,j,k) * u(12,i,j,k)

forall (i=2:(nx-1), j=2:(ny-1), k=1:nz)
  rsi(i,j,k) = rui(1,j,k) * u(13,i,j,k)
A(2:(NX-1):1, 2:(NY-1):i, 4:(NZ-3):i) = 

E(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

D(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

REP_TEMP = 

REP_TEMP = 607, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 629, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 633, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 635, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

C(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

A(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

D(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

E(2:(NX-1):1, 2:(NY-1):1, 2:(NZ-1):1) = 

REP_TEMP = 607 

REP_TEMP = 629 

REP_TEMP = 633 

REP_TEMP = 635 

REP_TEMP = 607, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 629, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 633, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

REP_TEMP = 635, 1, (1+ABS(((NX-1)-2))}, 2, (1+ABS(((NY-1)-2))), 3, (1+ABS(((NZ-1)-2)))) 

"
Compiler output under Subspace Minimization
call cm_timer_clear(0)
call cm_timer_start(0)

NX = & 12
NY = & 12
NZ = & 12
DT = & 0.015
R43 = ( 4.0 * S (1.0 / U(1,2:(NX-1):1,2:(NY-1):1,1:NZ:1)) - 1.0 )

Q(2:(NX-1):1,2:(NY-1):1,1:NZ:1) = A0.5


REP TEMP_647 = ( Tz2 + DT)

REP TEMP_648 = ( Tz1 - DT)

REP TEMP_649 = ( Tz1 - DT)

REP TEMP_650 = ( Tz1 - DT)

REP TEMP_651 = ( Tz1 - DT)

REP TEMP_652 = ( Tz1 - DT)

REP TEMP_653 = ( Tz1 - DT)

REP TEMP_654 = ( Tz1 - DT)

REP TEMP_655 = ( Tz1 - DT)

REP TEMP_656 = ( Tz1 - DT)

REP TEMP_657 = ( Tz1 - DT)

REP TEMP_658 = ( Tz1 - DT)

REP TEMP_659 = ( Tz1 - DT)

REP TEMP_660 = ( Tz1 - DT)

REP TEMP_661 = ( Tz1 - DT)

REP TEMP_662 = ( Tz1 - DT)

REP TEMP_663 = ( Tz1 - DT)

REP TEMP_664 = ( Tz1 - DT)

REP TEMP_665 = ( Tz1 - DT)

REP TEMP_666 = ( Tz1 - DT)

REP TEMP_667 = ( Tz1 - DT)

REP TEMP_668 = ( Tz1 - DT)

REP TEMP_669 = ( Tz1 - DT)

REP TEMP_670 = ( Tz1 - DT)

REP TEMP_671 = ( Tz1 - DT)

REP TEMP_672 = ( Tz1 - DT)
\[ B(2:(NX-1):1,2:(NY-1):1,NZ-2) = B(2:(NX-1):1,2:(NY-1):1,NZ-2) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_665, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{REPTEMP}_666 = 0 \]
\[ C(2:(NX-1):1,2:(NY-1):1,NZ-2) = C(2:(NX-1):1,2:(NY-1):1,NZ-2) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_666, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{REPTEMP}_667 = 6.0 \]
\[ \text{D(2:(NX-1):1,2:(NY-1):1,NZ-2) = D(2:(NX-1):1,2:(NY-1):1,NZ-2) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_667, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{REPTEMP}_668 = -4.0 \]
\[ \text{A(2:(NX-1):1,2:(NY-1):1,NZ-1) = A(2:(NX-1):1,2:(NY-1):1,NZ-1) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_668, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{REPTEMP}_669 = 1.0 \]
\[ \text{B(2:(NX-1):1,2:(NY-1):1,NZ-1) = B(2:(NX-1):1,2:(NY-1):1,NZ-1) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_669, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{REPTEMP}_670 = -4.0 \]
\[ \text{C(2:(NX-1):1,2:(NY-1):1,NZ-1) = C(2:(NX-1):1,2:(NY-1):1,NZ-1) + \text{SPREAD(}\text{SPREAD(}\text{REPTEMP}_670, 1, (1+ABS((\text{NX}-1)-2))), 2, (1+ABS((\text{NY}-1)-2))) \]
\[ \text{cali} \text{ cm} \text{ timer} \text{ stop}(0) \]
\[ \text{call} \text{ cnLm} \text{ timer} \text{ print}(0) \]

end
Appendix B: Subspace Compiler Source Code

This appendix contains the Common LISP source code for the implemented portions of the Subspace compiler described in this thesis. The source files are included in alphabetical order. To summarize, they are:

back-end.lisp Parts of the Back End phase dealing with code.
be-decls.lisp Parts of the Back End phase dealing with declarations.
eval.lisp Evaluator the Back End uses to generate infix expressions.
main.lisp Top level compiler interface.
misc.lisp Various patches.
output.lisp Fortran output apparatus used by the Back End.
reconcile.lisp Code for the Subspace Reconciliation phase.
subspace.lisp Subspace tree utilities, including SST parsing/printing.
temp-alloc.lisp Code for the Temp Allocation phase.
timer.lisp Interface to CM timing facilities used by Back End.

back-end.lisp

;;;*******************************************************************************
;;; Subspace Compiler
;;; Version 1.0
;;; M.Eng. thesis under Professor William J. Dally
;;; Copyright (c) Todd O. Dampier, 1994.
;;;*******************************************************************************
;; back-end.lisp
;; This file contains code for the Back End phase of the compiler.
;; Specifically, these are the routine which produce CM FORTRAN code from
;; SST nodes.


(in-package 'user)

;; Code generation
;; Da brains -- recursive descent unparsing

(defun f-unparse-sst-node-list (sst-node-list &key (col 7) (in-space nil))
  (mapcar #'(lambda (sst-node) (f-unparse-sst-node sst-node :col col :in-space in-space))
    sst-node-list))

(defun f-unparse-sst-node (sst &key (col 7) (in-space nil) (lhs nil))
  (cond ((eq sst 'T) (f-output ".TRUE." :col col)) ; Kinda kludgy.
    ((not (sst-node-p sst))
     (f-error "f-unparse-sst-node: Not an sst node -- ~s~q" sst))
    ))
(not (all (mapcar #'defined-space (sst-node-space sst))))
(f-error "f-unparse-sst-node: Space is undefined -- ~S~%"
 (sst-node-space sst)))
(t
(case (sst-node-rator sst)
  (=
    (cond ((not (sst-node-p (rhs sst)))
          (f-error "f-unparse-sst-node: RHS of = is not an sst.- thru +")
          ((is-scan (rhs sst))
           (f-unparse-scan (lhs sst) (rhs sst)))
          ((is-phi (rhs sst))
           (f-unparse-phi (lhs sst) (rhs sst) nil :col col))
          (t
           (let* ((my-space (sst-node-space sst))
                   (generate-forall (and (not (spaces-subset my-space in-space))
                                        (needs-forall (lhs sst) (rhs sst))))
                (new-col (+ col (if generate-forall 3 0)))
                (new-space (if generate-forall
                             (spaces-union my-space in-space)
                             in-space)))
               (:f-output "C Space ~S~%" in-space :col 1)
               (:f-output " ; Should be the above, but I think spaces-minus is buggy, Kath!
               (if generate-forall (emit-loops (set-difference my-space in-space)
                              :col col))
               (f-unparse-sst-node (lhs sst) :col new-col :in-space new-space)
               (f-output "="
               (f-continuation-line)
               (f-unparse-sst-node (rhs sst) :col (+ new-col 3)
                              :in-space new-space
                              :lhs (lhs sst))
               (f-output "~S")
               ))
      )
    (+ - / *)
    (let ((first-rand (first (sst-node-rands sst)))
          (rest-rands (cdr (sst-node-rands sst))))
     (f-output "(" :col col)
     (if (null rest-rands)
       (case (sst-node-rator sst)
       (())
       ((f-output "-") ;Unary -
        ((f-output "1.0") ;Unary /
         (+) (f-error "Binary op used with one argument.- thru +")
         (f-unparse-sst-node first-rand :lhs lhs :in-space in-space :col (+ col 3))
         (dolist (rand rest-rands)
          (f-continuation-line)
          (f-output "~S " (sst-node-rator sst) :col (+ col 1))
          (f-unparse-sst-node rand :lhs lhs :in-space in-space :col (+ col 3))
          (f-output ")")
         ))
       (abs)
       (if (> (length (sst-node-rands))) 1)
       (f-error "f-unparse-sst-node: Too many arguments to ABS.- thru +")
       (progn
        (f-output "abs(" :col col)
        (f-unparse-sst-node (first (sst-node-rands sst))
                       :lhs lhs :in-space in-space)
        (f-output "))")
       )
       (lt gt le ge ne and or)
       (if (> (length (sst-node-rands))) 2)
       (f-error "f-unparse-sst-node: Too many arguments to comparison.- thru +")
       (progn
        (f-output "(" :col col)
        (f-unparse-sst-node (first (sst-node-rands sst))
                       :lhs lhs :in-space in-space)
        (f-output ")")
        (f-if (member (sst-node-rator sst) '(and))
          (f-continuation-line)
          (f-unparse-sst-node (second (sst-node-rands sst)) :col (+ col 3)
                      :in-space new-space
                      :lhs (lhs sst))
          (f-output ")")
        ))
       )
       (not)
       (if (> (length (sst-node-rands))) 1)
       (f-error "f-unparse-sst-node: Too many arguments to negation.- thru +")
       (progn
        (f-output "(" :col col)
        (f-unparse-sst-node (first (sst-node-rands sst))
                       :lhs lhs :in-space in-space)
        (f-output ")")
        (f-continuation-line)
        (f-unparse-sst-node (second (sst-node-rands sst)) :col (+ col 3)
                      :lhs lhs :in-space in-space)
        (f-output ")")
        ))
       )
       (rep)
       (let ((target-space (sst-node-space sst))
             (expansion-axes (first (sst-node-rands sst)))
             (expr-to-expand (second (sst-node-rands sst))))
       (if (sst-node-p expr-to-expand)
         (case (sst-node-rator expr-to-expand)
         ())))
((number)
  (f-output "~S" (first (sst-node-rands expr-to-expand)) :col col))
((scalar aref)
  (f-unparse-replicate expr-to-expand target-space expansion-axes
   lhs (col col :in-space in-space))
  (t ; If source is not simple, we must use a temp.
   (f-error "Complex argument to REP not normalized! -S-% expr-to-expand"]))
(cond
  ((numberp expr-to-expand) (f-output "~S" expr-to-expand :col col))
  ((symbolp expr-to-expand)
   (f-unparse-replicate (create-scalar 'nil expr-to-expand)
                        target-space expansion-axes
                        lhs (col col :in-space in-space))
   (t (f-error "Bad replicated expression: ~S-% expr-to-expand")))
  ((hop)
   (f-output "hop")
   (iter)
   (let* ((rands (sst-node-rands sst))
          (indexvar (first rands))
          (statements (rest rands))
          (indexrange (get-range indexvar)))
     (if (null indexrange)
         (f-output "do -S-=S, -A, ~A, ~A-% %" indexvar
                   (eval-to-str (range-lo indexrange))
                   (eval-to-str (range-hi indexrange))
                   (eval-to-str (range-stride indexrange))
                   :col col))
     (f-unparse-sst-node-list statements :col (+ col 3)
                                         :in-space (spaces-union (list indexvar)
                                                                 in-space))
     (f-output "enddo-%" :col col))
  ((add-scan mul-scan max-scan min-scan and-scan or-scan)
   (f-output "scan")
   (spec)
   (let ((axis (first (sst-node-rands sst)))
          (loc (second (sst-node-rands sst)))
          (expr (spec-exp sst)))
     (if (sst-node-p loc) 0 1))
   (f-output "specific")
  (any)
   (f-output "any")
  (last-true)
   (f-output "last-true")
  (phi)
   (do* ((clauses (sst-node-rands sst) (cddr clauses))
         (test (first clauses) (first clauses))
         (expr (second clauses) (second clauses))
         (null clauses))
     (f-output "phi")
   (block seq)
   (f-unparse-sst-node-list (sst-node-rands sst) :col col :in-space in-space))
  (aref)  ; arefs must be simple -- only simple exprs as indices
   (let* ((rands (sst-node-rands sst))
          (arr-name (first rands))
          (indices (cdr rands)))
     (f-output "-S(" arr-name :col col)
     (f-output (eval-to-str-expand-triplets (first indices) :in-space in-space))
     (dolist (index (rest indices)
                 (f-output ",")
                 (f-output (eval-to-str-expand-triplets index :in-space in-space))
                 (f-output ")")
     (f-output ")")
   (baref)  ; barefs may have arefs, etc., in their indices.
     (let* ((rands (sst-node-rands sst))
            (arr-name (first rands))
            (indices (cdr rands)))
     (f-output "-S(" arr-name :col col)
     (if (sst-node-p (first indices))
         (f-unparse-sst-node (first indices))
         (f-output (eval-to-str (first indices)))
     (dolist (index (rest indices)
                 (f-output ",")
                 (f-output ")")
     (f-output ")")
     (if (sst-node-p index)
(f-unparse-sst-node index)
    (f-output (eval-to-str index)))

(f-output '*)

(scalar)
    (f-output "-S" (first (sst-node-rands sst)) :col col)
;; To avoid having any floating point variables living on the front end,
;; the following lines can be substituted for the one above.
;; Modifications must also be made in be-decls.lisp.
;; (if (eq (find-type sst) 'real)
    ; (f-output "-S(l,l,l)" (first (sst-node-rands sst)) :col col)
    (f-output "-S" (first (sst-node-rands sst)) :col col))

(number)
    (f-output "-S" (first (sst-node-rands sst)) :col col)
(t (f-error "f-unparse-sst-node: Unrecognized operator ~S-%" 
    (sst-node-rator sst) )))

(defun f-unparse-phi (target phi-sst encl-pred &key (in-space nil) (col 7))
;; We assume all phis are either statments or top-level rhs exprs.
;; target: nil if this phi is a stmt. The lhs if this phi is an rhs.
;; phi-sst: the sst that is this phi.
;; encl-pred: the enclosing predicate. For instance, if phis are nested.
;; nil if none.
    (do* ((clauses (sst-node-rands phi-sst) (cddr clauses))
          (old-tests encl-pred
             (if (null old-tests)
                (create-local (sst-node-space expr) 'not
                             (list new-test))
                (create-local (sst-node-space old-tests) 'and
                              (list old-tests
                                   (create-local (sst-node-space old-tests) 'not
                                     (list new-test)))))))
          (new-test (first clauses) (first clauses))
          (expr (second clauses) (second clauses)))
          (null clauses))
    (let ((new-condition (if (null old-tests)
                              new-test
                              (create-local (sst-node-space old-tests) 'and
                                            (list old-tests new-test))))
          (cond ((is-phi expr)
                 (f-unparse-phi target expr new-condition :in-space in-space :col col))
                 (null target)
                 ;; This phi is used for control flow.
                 t)
          ;; This phi is an expression.
          (let ((assignment (create-local (sst-node-space expr)
                                     (list target expr)))
                (assg-space (sst-node-space expr)))
                (if (needs-forall (lhs assignment) (rhs assignment))
                    (progn
                      (emit-loops
                       (set-difference assg-space in-space)
                       :in-space (spaces-union assg-space in-space)
                       :cond new-condition :col col))
                    (f-unparse-sst-node assignment :col (+ col 3)
                                   :cond new-condition :col (+ col 3)))))))

(defun f-unparse-scan (target scan-sst)
    (let* ((target-name (cond
                          ((symbolp target) target)
                          ((and (sst-node-p target) (is-aref target))
                           (aref-sym target))
                          (t (f-error "f-unparse-scan: Target is not a symbol or sst-node -- ~S-%" target)))))
        ((source (scan-exp scan-sst))
         (source-name (cond
                        ((symbolp source) source)
                        ((and (sst-node-p source) (is-aref source))
                         (aref-sym source))
                        (t (f-error "f-unparse-scan: Source is not a symbol or sst-node -- ~S-%" source)))))
        (axis (xpand-axis scan-sst))
        (space (order-axes (sst-node-space scan-sst)))
        (segment (f-make-segment-array space))
        (mask (f-make-mask-array space)))
(cond ((listp axis) (f-error "f-unparse-scan: Scan axis is a list -- ~S-#" axis))
  (not (spaces-subset axis space))
  (f-error "f-unparse-scan: Scan axis -S is not in result space -S-#" axis space))
   ;; Should also check that scan axis is in source space.
  (t
   (f-output "CALL CMF_SCAN_-S(~S(" (combiner (sst-node-rator scan-sst)))
   (f-output ~S, ~S, ~S, CMF_UPWARD, CMF_INCLUSIVE, CMF_NONE, ~S"
     target-name
     source-name
     segment
     (1+ (position axis space))
     mask)
   (f-output "{)-#")
   ;; CALL CMF_SCAN_combiner ( DEST, SOURCE, SEGMENT, AXIS
   ;; & DIRECTION, INCLUSION, SEGMENT_MODE, MASK )
  )))

(defun f-make-segment-array (space)
  (let ((name (apply #'concatenate
                      (append '(string "SEGMENT_")
                              (mapcar #'princ-to-string space))))
        (f-output "-A = .TRUE.-#" name)
        (intern name)))

(defun f-make-mask-array (space)
  (let ((name (apply #'concatenate
                     (append '(string "MASK_")
                             (mapcar #'princ-to-string space))))
        (f-output "-A = .TRUE.-#" name)
        (intern name)))

(defun combiner (rator)
  (case rator
    ((add-scan) 'add)
    ((mul-scan) '???)
    ((max-scan) 'max)
    ((min-scan) 'min)
    ((and-scan) 'iand)
    ((or-scan) 'ior)))

(defun needs-forall (lhs rhs)
  ;; Question: when is it okay not to use a forall?
  ;; e.g.: a(1:10) = b(25:34) + 3
  (if (sst-node-p rhs)
      (cond ((member (sst-node-rator rhs) '(number scalar)) nil)
            ((contains-rep rhs) '(number scalar))
            (t t))
      (t nil))

(defun contains-rep (expr)
  (if (sst-node-p expr)
      (case (sst-node-rator expr)
        ((+ - / * abs lt gt le ge ne and or not phi)
         (not (none (mapcar #'contains-rep (sst-node-rands expr))))))
      (t t))

(defun conformable (expr)
  ;; NOTE: May incomplete.
  (if (sst-node-p expr)
      (value t -'())
      (case (sst-node-rator expr)
        ((aref scalar) (values t (aref-indices expr)))
        ((+ - * / abs) nil))))

(defun f-unparse-lhs (sst &key (col 7) (in-space nil))
  (cond ((not (lhs-p sst))
        (f-error "f-unparse-lhs: Not a valid lhs -- -S-#" sst))
        ((sympobj sst)
         (f-output "$S" sst))
        ((sst-node-p sst)
         (f-error "f-unparse-lhs: Space is undefined -- -S-#" sst))
        (case (sst-node-rator sst)
          ((aref)
           (let* ((rands (sst-node-rands sst))
                   (arr-name (first rands))
                   (indices (cdr rands))
                   (f-output "$I" arr-name :col col)
                   (f-output (eval-to-str (first indices)))
                   (dolist (index (rest indices))
                     (f-output ",")
                     (f-output (eval-to-str index))
                   (f-output ")")
           )))
  )
;;; The replicate stuff doesn't really do anything with the in-space
;;; argument -- just passes it through to any spawned calls to
;;; f-unparse-sst-node...
(defun f-unparse-replicate (source-expr target-space expansion-axes lhs &key (col 7) (in-space nil))
  (let* ((lhs-indices (case (sst-node-rator lhs)
                              ((aref) (rest (sst-node-rands lhs)))
                              (T
                               (f-error "Suspicious lhs with REP rhs: -S-%" lhs) nil)))
         (lhs-base-indices (extract-base-indices lhs-indices target-space))
         (ordering-pred #'(lambda (a b) (< (position a lhs-base-indices)
                                         (position b lhs-base-indices))))
         (ordered-expansion-axes (reverse (sort expansion-axes ordering-pred))))
    (if (all (mapcar #'(lambda (axis)
                        (if (not (member axis target-space))
                            (f-error "Replication axis ~S does not appear in target space -S-%" axis target-space))
                            (member axis target-space))
                        expansion-axes))
      (f-unparse-replicate-helper source-expr
                                 lhs-base-indices
                                 ordered-expansion-axes
                                 col
                                 in-space)
      (f-error "Replication unable to be performed.~%")))
(defun extract-base-indices (indices axes)
  (remove nil
           (mapcar #'(lambda (expr) (extract-base-index expr axes)) indices)))
(defun extract-base-index (expr axes)
  (cond ((member expr axes) expr)
        ((atom expr) nil)
        ((listp expr) (first (set-difference (extract-base-indices expr axes) (list nil))))))
(defun f-unparse-replicate-helper (source-expr target-space expansion-axes col in-space)
  (if (null expansion-axes)
      (f-unparse-sst-node source-expr :in-space in-space)
      (progn
       (f-output "SPREAD(" :col col)
       (f-unparse-replicate-helper source-expr target-space
                                (cdr expansion-axes)
                                (+ col 7)
                                in-space)
       (f-output ",")
       (if (> (length expansion-axes) 1) {f-continuation-line})
       (f-output "-S, ~A"
                (1+ (position (car expansion-axes) target-space))
                (range-length (get-range (car expansion-axes)))
                :col (+ col 7)))))
(defun range-length (rng)
  (let ((hi (range-hi rng))
        (lo (range-lo rng))
        (stride (range-stride rng)))
    (if (numberp stride)
        (if (= stride 1)
            (if (and (numberp lo) (numberp hi))
                (progn
                 (f-warning "Warning: replicating with non-unity stride.")
                 (return-from range-length
                              (eval-to-str (ceiling (/ (+ 1 (abs (- hi lo)))
                                                 (abs stride))))))
                 (f-warning "Warning: ignoring nonunity stride in replication.")
                 (eval-to-str '(+ 1 (abs (- ,hi ,lo)))))))
        (f-warning "Warning: ignoring possible nonunity stride in replication.")))
(defun order-axes (axis-list)
  (sort axis-list #'(lambda (a b) (string-lessp (princ-to-string a)
                                          (princ-to-string b)))))
(defun f-do-triplet (range)
  (f-output "-A" (triplet-to-str range)))
(defun emit-loops (space-spec &key ((:cond condition) nil) (in-space nil) (col 7))
be-decls.lisp

;;============================================
;; Subspace Compiler
;; Version 1.0
;; M.Eng. thesis under Professor William J. Dally
;; Copyright (c) Todd O. Dampier, 1994.
;;============================================

;; be-decls.lisp
;; -----------
;; This file contains code for the Back End phase of the compiler.
;; Specifically, these routines deal with declarations, producing CM FORTRAN
;; declarations from SST program decl-lists, and other parts of the CM FORTRAN
;; subroutine/function/program headers and footers.
;;============================================


(in-package 'user)

;; Back End phase:
;; Declarations
;;============================================

(defun f-program-header (&key (prog-name "jignesh"))
  (f-output "&-program -A-~\%-%" prog-name :col 7)
  (f-output "& INCLUDE '/usr/include/cm/CMFdefs.h'~\%-%" :col 7))

(defun f-program-footer ()
  (f-output "\% end-%\%" :col 7))

(defun f-unparse-decl-list (decl-list)
  (mapcar #* f-unparse-decl decl-list))

(defun f-unparse-decl (decl)
  ;; A decl is either a range declaration or a symbol declaration.
  ;; This function unparses either to a FORTRAN declaration (e.g., integer a(1:10)).
  ;; Note: if a range is used to define a sym, it must have already been defined.
  (cond ((range-p decl)
    ;; For a range, declare an integer in case we must use it as an index.
    (f-output "integer S-\%-%" (range-axis decl) :col 7))
    ((sym-entry-p decl)
      (let ((sym-name (sym-entry-name decl))
        (sym-axes (sym-entry-ranges decl))
        (sym-type (sym-entry-type decl)))
        ;; Make sure all the ranges used to define the symbol, have been declared.
        (if (all (mapcar #\%defined-axis sym-axes))
          ;; If so, declare the symbol.
          (progn
            (if (null sym-axes)
              ;; Add in commented-out lines to force real scalars to
              ;; live on the CM. Make sure to change back-end.lisp also.
              ;; (if (eq sym-type 'real)
              ;;   (progn
              ;;     (f-output "-S S(10, 10, 10)-\%-% sym-name :col 7)
              ;;   ;; (f-output "CMPS$ LAYOUT S(\%NEWS, \%NEWS, \%NEWS)-\%-% sym-name :col 1))
              ;; (f-output "-S S(\%-% sym-type sym-name :col 7)
              ;;   )
              ;; (progn
              ;;   ;; Issue warning if any range has stride other than one.
            )
          )))
    )))
  )
eval.lisp

;; Subspace Compiler
;; Version 1.0
;; M.Eng. thesis under Professor William J. Dally
;; Copyright (c) Todd O. Dampier, 1994.
;;
;;******************************************************************************
;;
;; eval.lisp
;;******************************************************************************
;; This file contains a mini-evaluator which is used to convert the prefix-style
;; notation within SSTs (mainly AREF indices) to FORTRAN infix expressions. There
;; are two versions of these routines, one which deals with triplets (e.g., A(1:n:2))
;; and one which does not.
;;
;;******************************************************************************
;; Header: /home/gn/dampier/Thesis/src/RCS/eval.lisp,v 1.2 1994/05/19 19:00:53 dampier Exp $

(in-package 'user)

;;******************************************************************************
;;
;; eval-to-str (expr)
;; A mini-evaluator to turn lisp prefix into fortran infix
;;******************************************************************************
(defun eval-to-str (expr)
  (cond ((null expr) (f-error "null expr passed to eval-to-str"))
    ((numberp expr) (format nil "~S") expr))
; Arithmetic functions
(op-list (car expr) (cdr expr)))
((eq (car expr) 'abs) (format nil "ABS(-A)" (eval-to-str (cadr expr))))
(t (f-error "nasty expr passed to eval-to-str: -S expr")))

(defun op-list (rator rands &key (spaces-round nil))
; Returns a string that is the eval-to-string result for each
; rand, separated from each other by the rator.
; e.g. (op-list + ' (a b c)) => "a + b + c"
(let ((result (eval-to-str (first rands))))
  (dolist (rand (rest rands))
    (setq result (concatenate 'string result
      (if spaces-round
        (format nil "-S " rator)
        (format nil "-S" rator))
      (eval-to-str rand))))
  (concatenate 'string "(" result ")")))

(defun eval-to-str-expand-triplets (expr &key (root t) (in-space nil))
; If root is true, returns an infix string corresponding to expr
; (e.g., "5", "3:5:1", "2-1:n-1:1"). Otherwise, returns either
; a string (for a non-triplet expr) or a list of three elements --
; lo, hi, stride (for a triplet expr).
(cond ((null expr) "")
  ((numberp expr) (format nil "~S" expr))
  ((symbolp expr) (if (member expr in-space)
    ; If it's already bound in the space, just output symbol.
    (format nil "~S" expr)
    ; Otherwise, see if it's got an associated triplet.
    (let ((e-range (get-range expr)))
      (if (null e-range)
        ; Not a range, so just output symbol.
        (format nil "~S" expr)
        ; Is a range, with a triplet.
        (if root
          ; triplet-to-str e-range
          (list (eval-to-str (range-lo e-range))
            (eval-to-str (range-hi e-range))
            (eval-to-str (range-stride e-range))))))))
  ((member (car expr) '(+ - */))
    ; Arithmetic functions
    ; (let ((result (op-list-expand-triplets (car expr) (cdr expr)
    ; :in-space in-space)))
    (if (and root (listp result))
      (format nil "-A:~A:~A" (first result) (second result) (third result))
      (eq (car expr) 'abs) (format nil "ABS(-A)" (eval-to-str (cadr expr))))
    (t (f-error "nasty expr passed to eval-to-str: -S expr")))

(defun op-list-expand-triplets (rator rands &key (spaces-round nil) (in-space nil))
; Returns a string that is the eval-to-string result for each
; rand, separated from each other by the rator.
; e.g. (op-list + ' (a b c)) => "a + b + c"
(if (null (rest rands))
  ; Only one operand -- better be unary minus.
  (if (eq ' - rator)
    ; It is unary minus -- force a null first operand: -a => null - a.
    (op-list-expand-triplets rator (cons nil rands) :spaces-round spaces-round
      :in-space in-space)
    (f-error "op-list-expand-triplets: Not enough operands for -S-% rator")
  ; More than one operand.
  (let ((result (eval-to-str-expand-triplets (first rands)))
    ; root nil :in-space in-space)
    (dolist (rand (rest rands))
      (let ((next-rand (eval-to-str-expand-triplets rand :root nil
        :in-space in-space)))
        (if (listp next-rand)
          (if (listp result)
            (f-error "Two ranges in one expression!")
            (setq result (list (concatenate 'string result
              (pad rator spaces-round)
              (first next-rand))
              (concatenate 'string result
                (pad rator spaces-round)
                (second next-rand))
              (third next-rand))))
          (if (listp result)
            (setq result (list (concatenate 'string (first result)
              (pad rator spaces-round)
              next-rand)
              (concatenate 'string (second result)
                (pad rator spaces-round)
                next-rand)
              (third result)))
            (setq result (concatenate 'string result
              (pad rator spaces-round)
              next-rand))))
        (concatenate 'string result
          (pad rator spaces-round)
          next-rand)
        (concatenate 'string result
          (pad rator spaces-round)
          next-rand)
        (third result)))
      (concatenate 'string result
        (pad rator spaces-round)
        next-rand)))
    result)))

73
(defun pad (symbol do-pad)
  (if do-pad (format nil "-S " symbol) (format nil "-S" symbol)))

(defun triplet-to-str (range)
  (format nil "-A:-A:-A"
    (eval-to-str (range-lo range))
    (eval-to-str (range-hi range))
    (eval-to-str (range-stride range))))

main.lisp

;;; Subspace Compiler
;;; Version 1.0
;;; M.Eng. thesis under Professor William J. Dally
;;; Copyright (c) Todd O. Dampier, 1994.

;; This file contains the compiler's top-level routines.


(in-package 'user)

;; An "sst program" has the form
;; ((d1 d2 ...) s)
;; where di is a declaration and
;; s is a subspace tree node.
;; A declaration is either a range or a sym. If a range is used
;; in the declaration of a sym, the declaration of the range must
;; precede it.

(defun f-unparse-sst-program (prog &key (file nil) (prog-name "jignesh"))
  ;; Unparse a program that is expressed as a subspace tree (sst), returning a string.
  ;; An sst *program* has the form (<list-of-decls> <sst>).
  ;; Modifies: Fortran output buffer << this unparsing.
  ;; Destroys old range and symbol table.
  (setf *range-tab* (make-hash-table))
  (setf *sym-tab* (make-hash-table))
  (f-zap-all-timers)
  (f-flush-output)
  (if (valid-sst-program? prog)
      (let* ((garbage (enter-decl-list (first prog)))
              (reconciled-prog (reconcile-sst-program prog))
              (normalized-prog (normalize-sst-program reconciled-prog)))
        (f-program-header :prog-name prog-name)
        (f-unparse-decl-list (first normalized-prog))
        (f-open-timer 0)
        (f-unparse-sst-node (second normalized-prog))
        (f-close-timer 0)
        (f-program-footer))
      (f-error "Not a well-formed sst program like ((decl1 decl2...) sst).")
    (f-unparsing-result file))

(defun valid-sst-program? (sstp)
  (and (eq 2 (length sstp)) ; An sst pgm is a list of two things:
       (list (first sstp)) ; 1. a list of decls, and
       (sst-node-p (second sstp)))) ; 2. an sst node.

(defun load-compiler ()
  (cd "~dampier/Thesis/src")
  (load "main.lisp")
  (load "subspace.lisp")
  (load "be-decls.lisp")
  (load "eval.lisp")
  (load "misc.lisp")
  (load "output.lisp")
  (load "reconcile.lisp")
  (load "temp-alloc.lisp")
  (load "test.lisp")
  (load "timer.lisp")
)

(defun compile-sst-benches (sst name)
  (format 't "&Compiling owner computes version..
        (set-reconcile 'owner-computes)
        (f-unparse-sst-program sst :file (concatenate 'string name "-oc.fcm")
        :prog-name name)
        (format 't "&Compiling union of children version..."))
(set-reconcile 'union-of-children)
(f-unparse-sst-program sst :file (concatenate 'string name "-uc.fcm")
:prog-name name)
(format 't "~&Compiling subspace minimized version...")
(set-reconcile 'subspace-minimize)
(f-unparse-sst-program sst :file (concatenate 'string name "-sm.fcm")
:prog-name name)
(format 't "~&Done.")
'done)

misc.lisp

;; This file contains miscellaneous code, patches, etc.
;; Subspace Compiler
;; Version 1.0
;; M.Eng. thesis under Professor William J. Dally
;; Copyright (c) Todd O. Dampier, 1994.
;;
(defvar *print-gensym* nil) ;Suppresses "#:" before gensyms.

(defun is-phi (exp)
  (eq 'phi (sst-node-rator exp)))
(defun spaces-diff (space-a space-b)
  ;; spaces-diff is like spaces-minus, only it doesn't complain
  ;; (e.g., when an element of space-b is not in space-a).
  (set-difference space-a space-b))

output.lisp

;; This file contains the FORTRAN output apparatus for the compiler.
;; It also contains routines for the compiler's error and warning messages.
;;
(defvar *print-gensym* nil) ;Suppresses "#:" before gensyms.

(defun f-error (ctl-string &rest args)
  (apply #'format (append (list '#t ctl-string) args)))
(defun f-warning (ctl-string &rest args)
  (apply #'format (append (list '#t ctl-string) args)))
(defvar *f-output-string* "")
(defvar *f-output-stream* (make-string-output-stream))
(defun f-flush-output ()
  (setq *f-output-string* (concatenate 'string *f-output-string*
    (get-output-stream-string *f-output-stream*))))
(defun f-new-output ()
  (setq *f-output-string* ""))
(defun f-continuation-line ()
  (f-output-helper "&" & *))
(defun f-fresh-line ()
  (f-output-helper "&"))
(defun f-new-line ()
  (f-output-helper ":"))
(defun f-comment (ctl-string &rest args)
  (apply #'f-output (cons (concatenate 'string "&c " ctl-string "&") args))))
(defun f-output (ctl-string &rest args)
  ; Since f-output-helper has keyword arguments in its lambda list
  ; (to accommodate the :col option) and also a rest specifier
  ; (to accommodate the varargs to format), we must guarantee that
  ; after ctl-string, there are an even number of args remaining.
  ; Gross, non?
  (if (evenp (length args))
    (apply #'f-output-helper (cons ctl-string args))
    (apply #'f-output-helper (append (list (concatenate 'string *"* ;ignore arg
      "garbage-arg") ;the ignored arg
      ctl-string) args))))
(defun f-output-helper (ctl-string &rest args &key (col 7) &allow-other-keys)
  (apply #'format (append (list *f-output-stream* (concatenate 'string
    "",0T
    ctl-string)) args)))
(defun f-unparsing-result (file)
  (f-flush-output) ;Flush fortran output buffer to *f-output-string*.
  (if (not (null file)) ;If file is not nil, then
    (with-open-file (ofile file ;If file is not nil, then
      ;write string to file.
      ;direction :output
      :if-exists :supersede)
      (write-string *f-output-string* ofile)))
  *f-output-string*)

reconcile.lisp

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ;; Subspace Compiler
;; ;; Version 1.0
;; ;; M.Eng. thesis under Professor William J. Dally
;; ;; Copyright (c) Todd O. Dampier, 1994.
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; reconcile.lisp
;; -------------------
;; This file contains code for the Subspace Reconciliation phase.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(in-package 'user)
;; NOTE: nothing in this file deals with the condition field yet!

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *subspace-reconciliation-setting* 'owner-computes)
(defun set-reconcile-owner-computes ()
  (setq *subspace-reconciliation-setting* 'owner-computes))
(defun set-reconcile-union-of-children ()
  (setq *subspace-reconciliation-setting* 'union-of-children))
(defun set-reconcile-subspace-minimize ()
  (setq *subspace-reconciliation-setting* 'subspace-minimize))
(setq *subspace-reconciliation-setting* 'subspace-minimize))

(defun set-reconcile (arg)
  (case arg
    ((owner-computes 1)
     (set-reconcile-owner-computes))
    ((union-of-children 2)
     (set-reconcile-union-of-children))
    ((subspace-minimize 3)
     (set-reconcile-subspace-minimize))
    (t (format 't "set-reconcile: Bad reconciliation policy -S-%" arg))))

(defun get-reconcile () *subspace-reconciliation-setting*)

;;; Top level reconciliation function
;;; Top level reconciliation function
(defun reconcile-sst-program (prog)
  (let ((decls (first prog))
         (sst (second prog))
         (new-sst nil))
    (setq new-sst (case (get-reconcile)
                       ((owner-computes)
                        (oc-reconcile-toplevel sst))
                       ((union-of-children)
                        (uc-reconcile-toplevel sst))
                       ((subspace-minimize)
                        (sm-reconcile-toplevel sst))
                       (t sst))
    (list decls new-sst)))

(defun oc-reconcile-toplevel (sst)
  ;; Returns an sst with its spaces normalized in the "owner computes" style.
  (if (not (sst-node-p sst))
      sst
      (case (sst-node-rator sst)
        (=)
        (create-local (sst-node-space (lhs sst))
          (list (lhs sst)
            (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
        (iter)
        (create-iter nil (xpand-axis sst)
          (oc-reconcile-toplevel (iter-exp sst)))
        (block seq phi)
        (create-sst-node nil (sst-node-rator sst)
          (mapcar #'oc-reconcile-toplevel (sst-node-rands sst)))
        (t sst))))

(defun expand-to-space (target-space sst)
  (if (leaf-p sst)
      (create-local (sst-node-space (lhs sst))
        (list (lhs sst)
          (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
      (case (sst-node-rator sst)
        (+ - * / abs lt gt le ge ne and or not phi)
        (create-local (sst-node-space (lhs sst))
          (list (spaces-minus target-space (sst-node-space sst))
            sst))
        (case (sst-node-rator sst)
          (+ - *)
          (create-local (sst-node-space (lhs sst))
            (spaces-minus target-space (sst-node-space sst))
            sst))
        (block seq phi)
        (create-local (sst-node-space (lhs sst))
          (mapcar #'(lambda (arg) (expand-to-space target-space arg))
            (sst-node-rands sst)))
        (t sst)))))

(defun reconcile-sst-program (prog)
  (let ((decls (first prog))
         (sst (second prog))
         (new-sst nil))
    (setq new-sst (case (get-reconcile)
                       ((owner-computes)
                        (oc-reconcile-toplevel sst))
                       ((union-of-children)
                        (uc-reconcile-toplevel sst))
                       ((subspace-minimize)
                        (sm-reconcile-toplevel sst))
                       (t sst))
    (list decls new-sst)))

;;; Owner computes
;;; Owner computes
(defun oc-reconcile-toplevel (sst)
  ;; Returns an sst with its spaces normalized in the "owner computes" style.
  (if (not (sst-node-p sst))
      sst
      (case (sst-node-rator sst)
        (=)
        (create-local (sst-node-space (lhs sst))
          (list (lhs sst)
            (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
        (iter)
        (create-iter nil (xpand-axis sst)
          (oc-reconcile-toplevel (iter-exp sst)))
        (block seq phi)
        (create-sst-node nil (sst-node-rator sst)
          (mapcar #'oc-reconcile-toplevel (sst-node-rands sst)))
        (t sst)))))

(defun expand-to-space (target-space sst)
  (if (leaf-p sst)
      (create-local (sst-node-space (lhs sst))
        (list (lhs sst)
          (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
      (case (sst-node-rator sst)
        (+ - * / abs lt gt le ge ne and or not phi)
        (create-local (sst-node-space (lhs sst))
          (list (spaces-minus target-space (sst-node-space sst))
            sst))
        (case (sst-node-rator sst)
          (+ - *)
          (create-local (sst-node-space (lhs sst))
            (spaces-minus target-space (sst-node-space sst))
            sst))
        (block seq phi)
        (create-local (sst-node-space (lhs sst))
          (mapcar #'(lambda (arg) (expand-to-space target-space arg))
            (sst-node-rands sst)))
        (t sst)))))

(defun reconcile-sst-program (prog)
  (let ((decls (first prog))
         (sst (second prog))
         (new-sst nil))
    (setq new-sst (case (get-reconcile)
                       ((owner-computes)
                        (oc-reconcile-toplevel sst))
                       ((union-of-children)
                        (uc-reconcile-toplevel sst))
                       ((subspace-minimize)
                        (sm-reconcile-toplevel sst))
                       (t sst))
    (list decls new-sst)))

;;; Owner computes
;;; Owner computes
(defun oc-reconcile-toplevel (sst)
  ;; Returns an sst with its spaces normalized in the "owner computes" style.
  (if (not (sst-node-p sst))
      sst
      (case (sst-node-rator sst)
        (=)
        (create-local (sst-node-space (lhs sst))
          (list (lhs sst)
            (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
        (iter)
        (create-iter nil (xpand-axis sst)
          (oc-reconcile-toplevel (iter-exp sst)))
        (block seq phi)
        (create-sst-node nil (sst-node-rator sst)
          (mapcar #'oc-reconcile-toplevel (sst-node-rands sst)))
        (t sst)))))

(defun expand-to-space (target-space sst)
  (if (leaf-p sst)
      (create-local (sst-node-space (lhs sst))
        (list (lhs sst)
          (expand-to-space (sst-node-space (lhs sst)) (rhs sst))))
      (case (sst-node-rator sst)
        (+ - * / abs lt gt le ge ne and or not phi)
        (create-local (sst-node-space (lhs sst))
          (list (spaces-minus target-space (sst-node-space sst))
            sst))
        (case (sst-node-rator sst)
          (+ - *)
          (create-local (sst-node-space (lhs sst))
            (spaces-minus target-space (sst-node-space sst))
            sst))
        (block seq phi)
        (create-local (sst-node-space (lhs sst))
          (mapcar #'(lambda (arg) (expand-to-space target-space arg))
            (sst-node-rands sst)))
        (t sst)))))

(defun reconcile-sst-program (prog)
  (let ((decls (first prog))
         (sst (second prog))
         (new-sst nil))
    (setq new-sst (case (get-reconcile)
                       ((owner-computes)
                        (oc-reconcile-toplevel sst))
                       ((union-of-children)
                        (uc-reconcile-toplevel sst))
                       ((subspace-minimize)
                        (sm-reconcile-toplevel sst))
                       (t sst))
    (list decls new-sst))))
*all-scan-ops*
;; Scans do not change the space, even though they're "expansions".
(create-sst-node target-space (sst-node-rator sst)
  (list (xpand-axis sst)
    (expand-to-space target-space
      (scan-exp sst))))

(defun leaf-p (sst)
  (if (sst-node-p sst)
      (member (sst-node-rator sst) 'aref scalar number)
      t))

;;; Union of children

(defun uc-reconcile-toplevel (sst)
  ;; Returns an sst with its spaces normalized in the "union of children" style.
  (if (not (sst-node-p sst))
      sst
      (case (sst-node-rator sst)
        (= (uc-reconcile-expr sst))
        (iter)
          ;; NOTE: the expressions inside an iter cannot have the iter-axis
          ;; in their spaces. (????)
          (create-iter nil (xpand-axis sst)
            (uc-reconcile-toplevel (iter-exp sst)))
        (block seq phi)
          ;; For blocks, seqs, and control-flow phis.
          ;; NOTE: Is this right for a phi?
          (create-sst-node nil (sst-node-rator sst)
            (mapcar #'uc-reconcile-toplevel (sst-node-rands sst)))
        (t
          sst)))))

(defun uc-reconcile-expr (sst)
  (if (leaf-p sst)
      sst
      (case (sst-node-rator sst)
        C (=W
          (let* ((new-rhs (uc-reconcile-expr (rhs sst)))
            (uc-space (spaces-union (sst-node-space (lhs sst))
              (sst-node-space new-rhs))))
            (create-local uc-space '=(list (lhs sst)
              (uc-expand uc-space new-rhs))))
          (let* ((new-rands (mapcar #'uc-reconcile-expr (sst-node-rands sst)))
            (uc-space '())
            (garbage (dolist (spc (mapcar #'sst-node-space new-rands)
              (setq uc-space (spaces-union uc-space spc))))))
            (create-local uc-space (sst-node-rator sst)
              (mapcar #'(lambda (rand) (uc-expand uc-space rand))
                new-rands)))
        (+ - * / abs lt ge le ge ne and or not phi)
          (let* ((new-rands (mapcar #'uc-reconcile-expr (sst-node-rands sst)))
            ;; NOTE: There's gotta be a better way. Consult lisp book.
            (uc-space '()))
            (garbage (dolist (spc (mapcar #'sst-node-space new-rands)
              (setq uc-space (spaces-union uc-space spc))))))
            (create-local uc-space (sst-node-rator sst)
              (mapcar #'(lambda (rand) (uc-expand uc-space rand))
                new-rands)))
        (rep)
          (let ((new-rep-exp (uc-reconcile-expr (rep-exp sst))))
            (create-rep (spaces-union (sst-node-space (sst-node-space new-rep-exp))
              (xpand-axis sst))
              (list (xpand-axis sst) new-rep-exp)))
        (*all-scan-ops*
          ;; Scans do not change the space, even though they're "expansions".
          (let ((expansion-axes (spaces-diff target-space (sst-node-space sst))))
            (cond ((null expansion-axes) sst)
              ((lis-rep sst) (uc-expand target-space (rep-exp sst)))
              (t
                (create-rep target-space (list expansion-axes sst))))))))

;;; Subspace minimization

(defun uc-expand (target-space sst)
  ;; Returns an sst in which the second argument is expanded via
  ;; replication to target-space.
  (let ((expansion-axes (spaces-diff target-space (sst-node-space sst))))
    (cond ((null expansion-axes) sst)
      ((lis-rep sst) (uc-expand target-space (rep-exp sst)))
      (t
        (create-rep target-space (list expansion-axes sst))))))

;;; Subspace minimization

78
(defun sm-reconcile-toplevel (sst)
  ;; Returns an sst with its spaces normalized in the "subspace minimization" style.
  (if (not (sst-node-p sst))
      sst
    (case (sst-node-rator sst)
      (= (sm-reconcile-expr (create-local '() '=' (list (lhs sst) (coalesce-expr (rhs sst))))))
      (iter)
        ;; NOTE: the expressions inside an iter cannot have the iter-axis
        ;; in their spaces. (????)
        (create-iter nil (xpand-axis sst)
          (sm-reconcile-toplevel (iter-exp sst))))
      (block seq phi)
        ;; For blocks, ses, and control-flow phis.
        ;; NOTE: Is this right for a phi?
        (create-sst-node nil (sst-node-rator sst)
          (mapcar #'sm-reconcile-toplevel (sst-node-rands sst))))
      (t
       sst))))

(defun coalesce-expr (sst)
  ;; Returns an expression with maximum branching factor.
  (if (not (sst-node-p sst))
      sst
    (case (sst-node-rator sst)
      (+ (let ((coalesced-rands (mapcar #'coalesce-expr (sst-node-rands sst))))
            (create-local '() '+ (promote-rands coalesced-rands '+ '-))))
      (*)
        (let ((coalesced-rands (mapcar #'coalesce-expr (sst-node-rands sst))))
          (create-local '() '* (promote-rands coalesced-rands '* '/)))
      (-)
        (if (= (length (sst-node-rands sst)) 1)
            ;; Negation of its single operand.
            (let ((coalesced-rand (coalesce-expr (first (sst-node-rands sst))))
                  (sst-node-rator coalesced-rand))
              (**)
                ;; Negation of a product = negation of one multiplicand
                (create-local '() '**
                  (cons (negate (first (sst-node-rands coalesced-rand)) '-)
                        (rest (sst-node-rands coalesced-rand))))
              ('+)
                ;; Negation of a sum = negation of all addends
                (create-local '() '+
                  (mapcar #'(lambda (r) (negate r '-))
                        (sst-node-rands coalesced-rand))))
              (t
                (create-local '() '-' (list coalesced-rand))))
        ;; A subtraction.
        (let ((coalesced-rands (mapcar #'coalesce-expr (sst-node-rands sst))))
              (create-local '() '-
                (append (promote-rands (list (first coalesced-rands)) '+ '-)
                        (mapcar #'(lambda (r) (negate r '-))
                                (promote-rands (rest coalesced-rands)
                                              '+ '-))))))
      (abs)
        ;; NOTE: Might be worth using the rule (not used for now):
        ;; abs(a*b) = abs(a) * abs(b).
        (create-local '() 'abs (list (coalesce-expr (first (sst-node-rands sst))))))

)
;; NOTE: Pretty much ignores logical ops for now.
((lt gt le ge ne and or not phi)
 (create-local '() (sst-node-rator sst)
 (mapcar #'coalesce-expr (sst-node-rands sst))))
)

(rep)
 (create-rep '() (list (xpand-axis sst) (coalesce-expr (rep-exp sst))))

(*all-scan-ops*
 (create-sst-node '() (sst-node-rator sst)
 (list (xpand-axis sst) (coalesce-expr (scan-exp sst))))
)
(t sst)))

(defun negate (sst neg-op)
 (if (and (eq (sst-node-rator sst) neg-op)
 (= (length (sst-node-rands sst)) 1))
 (first (sst-node-rands sst))
 (create-local '() neg-op (list sst))))

(defun promote-rands (rand-list op inv-op)
 (let ((new-rands '()))
 (dolist (rand rand-list)
   (cond ((eq (sst-node-rator rand) op)
     (setq new-rands (append (sst-node-rands rand)
 new-rands)))
   ((eq (sst-node-rator rand) inv-op)
     (if (= (length (sst-node-rands rand)) 1)
       (setq new-rands (cons rand new-rands))
       (setq new-rands
         (append (list (first (sst-node-rands rand)))
         (mapcar #'(lambda (r) (negate r inv-op))
         (rest (sst-node-rands rand)))
 new-rands))))
   (t (setq new-rands (cons rand new-rands))))
 new-rands))

(defun sm-reconcile-expr (sst)
 (if (leaf-p sst)
 sst
 (case (sst-node-rator sst)
 ((=)
   (let* ((new-rhs (sm-reconcile-expr (rhs sst)))
   (sm-space (spaces-union (sst-node-space (lhs sst))
 (sst-node-space new-rhs))))
 (create-local sm-space '=' (list (lhs sst)
   (uc-expand sm-space new-rhs))))
 )
 (+)
 (sm-commutative-expr sst)
 (-) / (length (sst-node-rands sst))
 (let* ((new-rands (mapcar #'sm-reconcile-expr (sst-node-rands sst)))
   (sm-space '())
 (sm-space (spaces-union sm-space sm-space spc) )))
 (create-local sm-space (sst-node-rator sst)
 (mapcar #’(lambda (rand) (uc-expand sm-space rand))
 new-rands))
 )
 (rep)
 (let ((new-rep-exp (sm-reconcile-expr (rep-exp sst)))
 (create-rep (spaces-union (sst-node-space new-rep-exp) (xpand-axis sst))
 (list (xpand-axis sst) new-rep-exp)))
 (*all-scan-ops*
  ;; Scans do not change the space, even though they're "expansions".
  (create-sst-node (sst-node-space new-scan-exp) (sst-node-rator sst)
   (list (xpand-axis sst) new-scan-exp)))
 )
)))

(defun mv-expansion-cost (dest spaces marbles footprints)
 ;; Returns an a-list of spaces and the spaces they're promoted to
 ;; in an optimal reconciliation of the spaces in marbles.
 ;; Returns (in second return value) the cost (# of expansions)
 ;; in such a solution.
 ;; dest = ultimate target space
 ;; spaces = the collection of possible spaces which operands may occupy
 ;; marbles = spaces containing operands
 ;; footprints = spaces that operands have been promoted out of
 ;; Implementation: This uses a dynamic programming algorithm.
 (cond ((null marbles)
   ;; Base case: no arguments involve
   )

80
no promotions and incur no cost.

(values nil 0)

((member (first marbles) footprints :test #'spaces-equal)
   ; Marble piggybacking case:
   ; if the operand is in a space from
   ; which another operand has been promoted,
   ; they can be joined in that space and
   ; promoted together for the same cost.
   ; We can "forget about" the first marble.
   (values nil 0))

((member (first marbles) footprints :test #'spaces-equal)
   ; Marble promotion case:
   ; the operand must be promoted at a
   ; cost of 1 promotion to the parent space
   ; that yields the cheapest solution.
   (let* ((parents (parent-spaces (first marbles) spaces))
          (costs-and-promotions
           (mapcar #' (lambda (p) (multiple-value-list
                                  (mv-expansion-cost dest spaces
                                  (adjoin p (rest marbles))
                                  (adjoin (first marbles) footprints))))
          parents))
          (costs (mapcar #'cadr costs-and-promotions))
          (promotions (mapcar #'car costs-and-promotions))
          (min-cost (apply #'min costs))
          (min-cost-pos (position min-cost costs))
          (min-cost-parent (nth min-cost-pos parents))
          (min-cost-promotions (nth min-cost-pos promotions)))
   (values (acons (first marbles) min-cost-parent min-cost-promotions)
           (+ 1 min-cost))))

(defun parent-spaces (space spaces)
  ;; Spaces is a collection of possible spaces. The parent spaces are those
  ;; spaces which are one basis axis larger than space. They are returned as
  ;; a set of spaces.
  ;; Rep: A collection of spaces is represented by a list of basis axes.
  (let ((result nil))
    (dolist (axis spaces)
      (if (not (member axis space))
        (setq result (adjoin (cons axis space) result))))
    result))

(defun make-spaces (largest-space)
  ;; Returns an object representing all spaces that are (nonstrict) subspaces
  ;; of largest-space.
  ;; Rep: A collection of spaces is represented by a list of basis axes.
  ;; largest-space)

(defun sm-commutative-expr (sst)
  ;; Requires: sst is a commutative, associative operation sst.
  ;; Returns an sst equivalent to the argument, but grouped such that
  ;; expansions take place in an optimal fashion.
  (let* ((rator (sst-node-rator sst))
         (rands (mapcar #'sm-reconcile-expr (sst-node-rands sst)))
         (rand-spaces (mapcar #'sst-node-space rands))
         (target-space (reduce #,union rand-spaces :initial-value '()))
         (xpand-alist (mv-expansion-cost target-space
                                         (make-spaces target-space)
                                         rand-spaces
                                         (list target-space)))
         (sm-group rator rands target-space xpand-alist))

(defun sm-group (rator rands target-space xpand-alist)
  ;; Returns an sst which combines rands under the (commutative, associative)
  ;; rator by grouping the rands together by space, and expanding smaller-space
  ;; rands to larger spaces as specified by xpand-alist, an a-list whose
  ;; pairs specify spaces (car) and the spaces to which they're expanded (cdr).
  ;; The top-level result is in target-space.
  (let* ((child-xpands (remove-if-not #'(lambda (pair) (spaces-equal (cdr pair)
                                               target-space))
                                  xpand-alist))
         (child-spaces (mapcar #'car child-xpands))
         (raw-args (mapcar #'(lambda (spc) (sm-group rator rands spc xpand-alist))
                  child-spaces))
         (cooked-args (remove-if #'null raw-args))
         (xpanded-args (mapcar #'(lambda (rand) (uc-expand target-space rand))
                               cooked-args))
         (these-are-rands (remove-if-not #'(lambda (rand) (spaces-equal rand
                                                        target-space)
                                (xspaced-args (sm-group rator rand target-space)
                                              xpand-alist))
                            child-xpands)))
(sst-node-space rand))

(all-rands (append this-space-args xpanded-args))
(if (> (length all-rands) 1)
  (create-local target-space rator all-rands)
  (first all-rands))))

;; Unused stuff

(defun expansion-cost (dest spaces marbles footprints)
  ;; Cost-only version of mv-expansion-cost which doesn't record
  ;; promotions used.
  (cond ((null marbles) 0)
        ((member (first marbles) footprints :test #'spaces-equal)
         (expansion-cost dest spaces (rest marbles) footprints))
        (t
         (let* ((parents (parent-spaces (first marbles) spaces))
                (costs
                 (mapcar #'(lambda
                             p)
                          (expansion-cost dest spaces
                                        (adjoin p (rest marbles))
                                        (adjoin (first marbles) footprints)))
                         parents))
          (min-cost (apply #'min costs))
          (min-cost-pos (position min-cost costs))
          (+ 1 min-cost))))

(defun expr-cost (sst)
  ;; Calculates the cost (number of promotions in an optimal solution)
  ;; of the commutative, associative operation sst.
  (let* ((expr-spaces (mapcar #'sst-node-space (sst-node-rands sst)))
         (target-space (reduce #'union expr-spaces :initial-value '()))
         (expansion-cost target-space
                         make-spaces target-space
                         expr-spaces
                         (list target-space))))

;;; Module tests

(defvar rec-j (in-sst "((i j k) araf (d i j k))
  ((i) + (((i) araf (a i))
            ((i) araf (b j))))
  (k araf (c k)))))

subspace.lisp

;; Utilities for subspace trees.
;; Under development by Kathy Knobe.
;;
;; Utilities for subspace trees.
;; Under development by Kathy Knobe.

;; Package Setup

(in-package 'user)

;; Initialization

(defvar *all-indices* nil)
(defvar *range-tab* (make-hash-table))
(defvar *sym-tab* (make-hash-table))
(defvar *iteration-ss* nil)
(defvar *all-sst-nodes* (union *all-scan-ops* (add-scan mul-scan max-scan min-scan and-scan or-scan spec last-true any phi aref scalar number block seq))
(defvar *local-sst-nodes* (add-scan mul-scan max-scan min-scan and-scan or-scan))
(defvar *all-expansion-ops* (add-scan mul-scan max-scan min-scan and-scan or-scan))
(defvar *all-reduction-ops* (spec last-true any))
(defvar loadall ()

(fsetall)

(defun loadall ()

(fsetall)
(load "subspace.lisp")
(load "sst-bench.lisp")

(defun start ()
  (global-init)
  (phase-init)
  (loadall)
  (control-these 'printout (list analyze-p build-sst-p breakup-p gen-p)))

(defun global-init ()
  (setq *mods* (make-hash-table))
  (setq *xpansions* 'hop-only)
  (setq *num-phases* 4)
  (setq *print-level* nil)
  (setq *full-ilist* '(0 1 2 3 4 5 6 7 8 9 10 11 12))
  (setq *sym-table* (make-hash-table))
  (setq *procedures* (make-hash-table))
  (setq *sym-id* 0)
  (setq *unique-id* 0)
  (setq *print-array* 2)
  (setq *print-readably* nil)
  (setq *supress-printing-slots* nil)
  (setq *supress-printing-predicates* nil)
  (setq *supress-except* nil))

(defun init ()
  (analyze-init)
  (build-sst-init)
  (breakup-init)
  (gen-init))
(defstruct sst-node
  space rator rands &optional (cond t) (xrefs nil) (def nil))

(defun create-sst-node (space rator rands
  &optional (cond t) (xrefs nil) (def nil))
  (make-sst-node :space (canon-space space) :rator rator
  :rands rands :cond cond :xrefs xrefs :def def))

;;; SST nodes by type
;;  ------------------------
(defun create-local (space rator rands &optional (cond t))
  (create-sst-node space rator rands cond))
(defun create-rep (space rands &optional (cond t))
  (create-sst-node space 'rep rands cond))
(defun create-hop (space rands &optional (cond t))
  (create-sst-node space 'hop rands cond))
(defun create-scan (space rator rand &optional (cond t))
  (create-sst-node space (scan-op rator) (list rand) cond))
(defun scan-op (op)
  (case op
    ((add) 'add-scan)
    ((mul) 'mul-scan)
    ((max) 'max-scan)
    ((min) 'min-scan)
    ((and) 'and-scan)
    ((or) 'or-scan)
    (t (format t "^error: scan op ~S is unrecognizable" op))))
(defun create-aref (space name indices)
  (create-sst-node space 'aref (cons name indices)))
(defun create-scalar (space name)
  (create-sst-node space 'scalar (list name)))
(defun create-number (space num)
  (create-sst-node space 'number (list num)))
(defun create-iter (space axis rand &optional (cond t))
  (create-sst-node space 'iter (list axis rand) cond))
(defun create-seq (space rands &optional (cond t))
  (create-sst-node space 'seq rands cond))
(defun create-block (space rands &optional (cond t))
  (create-sst-node space 'block rands cond))
(defun create-spec (space scalar rand &optional (cond t))
  (create-sst-node space 'spec (list scalar rand) cond))
(defun create-last-true (space bool rand &optional (cond t))
  (create-sst-node space 'last-true (list bool rand) cond))
(defun create-any (space rand &optional (cond t))
  (create-sst-node space 'any (list rand) cond))
(defun create-phi (space rands &optional (cond t))
  (create-sst-node space 'phi rands cond))

;;; SST nodes type predicate
;;  ------------------------
(defun is-range (exp)
  (eq 'range (sst-node-rator exp)))
(defun is-local (exp)
  (member (sst-node-rator exp) *local-sst-nodes*))
(defun is-assign (exp)
  (eq '=' (sst-node-rator exp)))
(defun is-rep (exp)
  (eq 'rep (sst-node-rator exp)))
(defun is-hop (exp)
  (eq 'hop (sst-node-rator exp)))
(defun is-iter (exp)
  (eq 'iter (sst-node-rator exp)))
(defun is-scan (exp)
  (member (sst-node-rator exp) *all-scan-ops*)
(defun is-expansion (exp)
  (member (sst-node-rator exp) *all-scan-ops*)))
(defun is-reduction (exp)

  (member (sst-node-rator exp)
          "all-reduction-ops")
)

(defun is-spec (exp)

  (eq 'spec (sst-node-rator exp))
)

(defun is-last-true (exp)

  (eq 'last-true (sst-node-rator exp))
)

(defun is-any (exp)

  (eq 'any (sst-node-rator exp))
)

(defun is-scalar (exp)

  (eq 'scalar (sst-node-rator exp))
)

(defun is-aref (exp)

  (eq 'aref (sst-node-rator exp))
)

(defun is-number (exp)

  (eq 'number (sst-node-rator exp))
)

(defun is-leaf (exp)

  (or (is-number exp)
      (is-scalar exp)
      (is-aref exp))
)

(defun is-seq (exp)

  (eq 'seq (sst-node-rator exp))
)

(defun is-block (exp)

  (eq 'block (sst-node-rator exp)))

;; SST nodes slot accessor
;; ------------------------

(defun loc-rand1 (exp)

  (if (is-local exp)
      (first (sst-node-rands exp))
      (format t "error in loc-rand1: called on ~S which is not ~a local operation" exp))
)

(defun loc-rand2 (exp)

  (if (is-local exp)
      (second (sst-node-rands exp))
      (format t "error in loc-rand2: called on ~S which is not ~a local operation" exp))
)

(defun rep-exp (exp)

  (if (is-rep exp)
      (second (sst-node-rands exp))
      (format t "error in rep-exp: called on ~S which is not a rep node" exp))
)

(defun hop-exp (exp)

  (if (is-hop exp)
      (second (sst-node-rands exp))
      (format t "error in hop-exp: called on ~S which is not a hop node" exp))
)

(defun iter-exp (exp)

  (if (is-iter exp)
      (second (sst-node-rands exp))
      (format t "error in iter-exp: called on ~S which is not an ~iter node" exp))
)

(defun seq-list (exp)

  (if (is-seq exp)
      (sst-node-rands exp)
      (format t "error in seq-list: called on ~S which is not a seq node" exp))
)

(defun block-stmmts (exp)

  (if (is-block exp)
      (sst-node-rands exp)
      (format t "error in block-stmmts: called on ~S which is not a block node" exp))
)

(defun block-uses (exp)

  (if (is-block exp)
      (sst-node-xrefs exp)
      (format t "error in block-uses: called on ~S which is not a block node" exp))
)

(defun block-defs (exp)

  (if (is-block exp)
      (sst-node-xrefs exp)
      (format t "error in block-defs: called on ~S which is not a block node" exp)))
(defun scan-exp (exp)
  (if (is-scan exp)
      (second (sst-node-rands exp))
      (format t "-error in scan-exp: called on ~S which is not a scan node" exp)))

(defun spec-exp (exp)
  (if (is-spec exp)
      (third (sst-node-rands exp))
      (format t "-error in spec-exp: called on ~S which is not a spec node" exp)))

(defun last-true-bool (exp)
  (if (is-last-true exp)
      (second (sst-node-rands exp))
      (format t "-error in last-true-bool: called on ~S which is not a last-true node" exp)))

(defun last-true-exp (exp)
  (if (is-last-true exp)
      (third (sst-node-rands exp))
      (format t "-error in last-true-exp: called on ~S which is not a last-true node" exp)))

(defun any-exp (exp)
  (if (is-any exp)
      (second (sst-node-rands exp))
      (format t "-error in any-exp: called on ~S which is not an any node" exp)))

(defun xpand-axis (exp)
  (if (is-xpansion exp)
      (first (sst-node-rands exp))
      (format t "-error in xpand-axis: called on ~S which is not an xpansion node" exp)))

(defun reduce-axis (exp)
  (if (is-reduction exp)
      (first (sst-node-rands exp))
      (format t "-error in reduce-axis: called on ~S which is not a reduction node" exp)))

(defun scalar-sym (exp)
  (if (is-scalar exp)
      (first (sst-node-rands exp))
      (format t "-error in scalar-sym: called on ~S which is not a scalar node" exp)))

(defun aref-sym (exp)
  (if (is-aref exp)
      (first (sst-node-rands exp))
      (format t "-error in aref-sym: called on ~S which is not an aref node" exp)))

(defun leaf-sym (exp)
  (cond ((not (sst-node-p exp))
          (format t "-error in leaf-sym: called on ~S which is not an sst node" exp))
        ((is-aref exp) (aref-sym exp))
        ((is-scalar exp) (scalar-sym exp))
        (t (format t "-error in leaf-sym: called on ~S which is neither an aref nor a scalar node" exp)))

(defun aref-indices (exp)
  (cond (((is-aref exp)
          (cdr (sst-node-rands exp)))
          ((is-scalar exp) nil)
          (t (format t "-error in aref-indices: called on ~S which is neither an aref nor a scalar node" exp))))

(defun number-val (exp)
  (if (is-number exp)
      (first (sst-node-rands exp))
      (format t "-error in number-val: called on ~S which is not a number node" exp)))

(defun lhs (exp)
  (if (is-assign exp)
      (first (sst-node-rands exp))
      (format t "-error in lhs: called on ~S which is not an assign node" exp)))

(defun rhs (exp)
  (if (is-assign exp)
      (second (sst-node-rands exp))
      (format t "-error in rhs: called on ~S which is not an assign node" exp)))

(defun lhs-p (exp)
  (cond)}
((symbolp exp) t)
((sst-node-p exp) (eq 'aref (sst-node-rator exp))
(t nil)))

;; SST nodes setf operations
;; ---------------------------

(defun setf-aref-indices (sst indices)
  (if (is-aref sst)
      (setf (cdr (sst-node-rands sst)) indices)
      (format t "error in setf-aref-indices: called with "~S which ~
is not an aref")))

(defun setf-block-stmmts (prog stmmts)
  (if (is-block prog)
      (setf (sst-node-rands prog) stmmts)
      (format t "error in setf-block-stmmts: called with ~S which ~
is not a block" prog)))

(defun setf-iter-exp (prog exp)
  (if (is-iter prog)
      (setf (second (sst-node-rands prog)) exp)
      (format t "error in setf-iter-exp: called with ~S which ~
is not an iter" prog)))

(defun setf-seq-list (prog 1st)
  (if (is-seq prog)
      (setf (sst-node-rands prog) 1st)
      (format t "error in setf-seq-list: called with ~S which ~
is not a list") prog)))

;; Subspaces set utilities
;; ------------------------

(defun canon-space (sl)
  (sort sl string-lessp))

(defun spaces-subset (sl s2)
  (cond ((null sl) t)
        ((and (atom sl) (atom s2)) (eq sl s2))
        ((eq 'number sl) t)
        ((atom s2) nil)
        ((atom sl) (member sl s2))
        (t (subsetp sl s2))))

(defun spaces-equal (sl s2)
  (cond ((and (atom sl) (atom s2)) (eq sl s2))
        ((or (atom sl) (atom s2)) nil)
        ((not (eq (length sl) (length s2))) nil)
        (t (and (subsetp sl s2) (subsetp s2 sl)))))

(defun spaces-disjoint (sl s2)
  (cond ((and (atom sl) (atom s2)) (not (eq sl s2)))
        ((atom sl) (not (member sl s2)))
        ((atom s2) (not (member s2 sl)))
        (t (null (intersection sl s2)))))

(defun spaces-union (tl t2)
  (let ((sl (if (not (atom tl)) (copy tl) tl))
        (s2 (if (not (atom t2)) (copy t2) t2)))
    (canon-space
      (remove nil
        (cond ((and (atom sl) (atom s2))
          (union (list sl) (list s2)))
          ((atom sl)
            (un (list sl) s2))
          ((atom s2)
            (union (list s2) sl))
          (t (union sl s2)))))

(defun spaces-minus (sl s2)
  (cond ((null s2) sl)
        ((and (atom sl) (atom s2))
          (if (eq sl s2) nil
            (format t "error in spaces-minus: ~
              trying to remove ~S from ~S," sl s2)))
          ((atom s2) (if (member s2 sl)
            (remove s2 sl)
            (format t "error in spaces-minus: ~
              trying to remove ~S from ~S," sl s2)))
          ((atom sl) (format t "error in spaces-minus: ~
            trying to remove ~S from ~S," sl s2)))
        (t (if (subsetp s2 sl)
            (set-difference sl s2)
(format t "&error in spaces-minus:  
trying to remove ~S from ~S." s2 s1)))

(defun ss-subset (expl exp2)
  (if (not (sst-node-p expl))
      (format t "&error in ss-subset: called on ~S which is not an
sst-node" expl))
  (if (not (sst-node-p exp2))
      (format t "&error in ss-subset: called on ~S which is not an
sst-node" exp2))
  (subsetp (ss-of-sst expl) (ss-of-sst exp2)))

(defun ss-disjoint (expl exp2)
  (if (not (sst-node-p expl))
      (format t "&error in ss-disjoint: called on ~S which is not an sst-node" expl))
  (if (not (sst-node-p exp2))
      (format t "&error in ss-disjoint: called on ~S which is not an sst-node" exp2))
  (null (intersection (ss-of-sst expl) (ss-of-sst exp2))))

(defun ss-eq (expl exp2)
  (let ((spacel (if (symbolp expl) nil (sst-node-space expl)))
        (space2 (if (symbolp exp2) nil (sst-node-space exp2))))
    (if (and (or (symbolp expl) (sst-node-p expl))
             (or (symbolp exp2) (sst-node-p exp2)))
        (equal spacel space2)
        (format t "&error in ss-eq: arg is neither a symbol nor a subspace tree")))

(defun ss-of-sst (exp)
  (if (not (sst-node-p exp))
      (format t",-&error in ss-of-sst: called on
sst which is not an sst-node" exp))
  (sst-node-space exp))

(defun ss-of (exp)
  (cond ((null exp) nil)
         ((symbolp exp) 'ss-number)
         ((numberp exp) 'ss-number)
         ((sst-node-p exp)
          (typecase (sst-node-space exp)
                    (list (sst-node-space exp))
                    (space-min-max (if (eq (max-space-min-max (sst-node-space exp))
                                          (min-space-min-max (sst-node-space exp)))
                                   (max-space-min-max (sst-node-space exp))
                                   (format t "&error in ss-of: called on ~S but ~
                                            min and max are not the same"))))
         (sst-node (ss-of (sst-node-space exp)))
         (t (format t "&error in ss-of: called on ~S which is not handled ~
                    * exp"))))
  (t (format t "&error in ss-of: called on ~S which is not handled ~
             * exp"))))

(defun defined-space (ss)
  (if (atom ss)
      (defined-axis ss)
      (all (mapcar #'defined-axis ss))))

(defun defined-axis (ss)
  (if (null ss)
      t
      (get-range ss)))

(defun add-to-iteration-ss (axis)
  (setq *iteration-ss* (spaces-union *iteration-ss* axis))

(defun subtract-from-iteration-ss (axis)
  (setq *iteration-ss* (spaces-minus *iteration-ss* axis)))

;;; Converting between text and sst
;;;************************************************************************

(defun in-sst (x)
  (let* ((whole-pgm (read-sst x))
          (decls (first whole-pgm))
          (sst (second whole-pgm)))
    (list (mapcar #'text-to-decl decls)
           (text-to-sst sst))))

(defun read-sst (str)
  (setq x (read-from-string (fix-parens str)))
  x)

(defun fix-parens (str)
  (fix-parens str)
(map 'string #'fix-parens-aux str))

(defun fix-parens-aux (c)
  ;; Turns curly braces into parens, angle brackets into spaces.
  (case c
    ;;
    ((#\{) #\()
     ((#\}) #\))
    ((#\<) #\Space)
    ((#\>) #\Space)
    (t c)))

(defun old-fix-parens-aux (c)
  (if (or (equal #\{ c) (equal #\< c))
    (if (or (equal #\} c) (equal #\> c))
      '#\)
      (if (equal #\c) (equal #\c)))))

(defun text-to-decl (exp)
  (cond
    ((eq 'range (car exp))
     (set-range (create-range (axis exp) (lo exp) (hi exp)
         (if (not (null (stride exp)))
           (stride exp)
           1))))
    ((eq 'sym (car exp))
     (set-sym-entry
      (second exp) (third exp) (fourth exp) (fifth exp) (sixth exp)))
    (t (format t "~S-\text-to-decl: Not a valid declaration: -S-\" exp))))

(defun text-to-sst (exp)
  (cond
    ((numberp exp) (create-number '(i) exp))
    ((symbolp exp) (create-scalar '(i) exp))
    (t (format 't "\text-to-sst: atom -S is neither number nor symbol.-%" exp))
    ((member (car exp) *all-sst-nodes*) ; No space indicated. car is a rator.
     (text-to-sst-aux (space-to-string (sst-node-space n)) (sst-node-rator n))
     (if (null (sst-node-rands n))
       (list (space-to-string (sst-node-space n)) (sst-node-rator n))
       (list (space-to-string (sst-node-space n)) (sst-node-rator n)
           (print-xdefs n) (print-xuses n)))))

(defun text-to-sst-aux (space exp)
  (cond
    ((eq (car exp) 'phi)
     (create-phi space (mapcar #'text-to-sst (cadr exp)))))
    ((eq (car exp) 'aref)
     (create-aref space (caadr exp) (cdadr exp)))))

(defun print-sst (node)
  (print-full)
  (pprint (print-sst-el node)))

(defun print-sst-el (n)
  (cond
    ((listp n) (mapcar #'print-sst-el n))
    (t (if (sst-node-p n)
         (list (space-to-string (sst-node-space n)) (sst-node-rator n))
         (list (space-to-string (sst-node-space n)) (sst-node-rator n)
               (print-xdefs n) (print-xuses n))))

(defun print-sst-node (space exp)
  (cond
    ((eq (car exp) 'phi)
     (create-phi space (mapcar #'text-to-sst (cdr exp)))))
    ((eq (car exp) 'aref)
     (create-aref space (cadadr exp)))
    (t (format 't "\text-to-sst-nod: called with exp -S \" exp))))

(defun range-p n)
  (list 'range (range-axis n) (range-lo n) (range-hi n) (range-stride n))
(triplet-p n)
(triplet-to-string n)
(t n))

(defun triplet-to-string (trip)
  (intern (concatenate 'string (prinl-to-string (triplet-lo trip)) ":" 
  (prinl-to-string (triplet-hi trip)) ":" 
  (prinl-to-string (triplet-stride trip))))

(defun space-to-string (space)
  (cond ((null space) 
    (intern (concatenate 'string "{}"))
  ((listp space) 
    (mapcar #'intern 
      (mapcar #'prinl-to-string 
        (append (list '{) space (list '}}}))))
  ((space-min-max-p space) 
    (intern (concatenate 'string 
      "(min: " (prinl-to-string (min-space-min-max space)) 
      " max: " 
      (prinl-to-string (max-space-min-max space)) 
      ")"))
  (t (intern (concatenate 'string "(" (prinl-to-string space) ")"))))

(defun is-sst (n)
  (cond
    ((symbolp n) nil)
    ((numberp n) nil)
    ((and (= 2 (length n)) nil) 
      (member (first n) *all-sst-nodes*) t)
    ((and (= 3 (length n)) nil) 
      (member (second n) *all-sst-nodes*) t)
    (t nil))

(defun axis (exp)
  (second exp))

(defun lo (exp)
  (third exp))

(defun hi (exp)
  (fourth exp))

(defun stride (exp)
  (fifth exp))

;;; Subspace tree I/O
(defun p-node (node stream)
  (format stream "~-S" 
    (print-sst-el node)))

(defun output-sst (exp file-name &optional flag)
  (with-open-file (stream file-name :direction :output)
    (format stream "-& ~S" exp)))

(defun input-sst (file-name)
  (with-open-file (stream file-name :direction :input)
    (read stream)))

;;; General Utilities
(defun print-full ()
  (setq "print-level" nil)
  (setq "print-length" nil))

(defstruct (my-hash 
  (:print-function print-hash)
  (:constructor make-hash)))

(defun print-hash (table)
  (maphash #P'(lambda (key val)
    (pprint key) (pprint val))
    table))

(defun empty-hash (h-table)
  (let ((empty t))
    (block blockl
      (maphash #'(lambda (key val)
        (setq empty nil)
        (return-from blockl))
        table)
    empty))

90
(defun lg (num)
  (log num 2))

(defun all (x) (not (member nil x)))

(defun none (x) (all (mapcar #'not x)))

(defun set-array (array-name index value)
  (setf (aref array-name index) value))

(defun set-bits (array-name index value)
  (setf (bit array-name index) value))

(defun get-array (array-name index)
  (aref array-name index))

(defun lookup-hash-table (table entry)
  (gethash entry table))

(defun write-hash-table (table name new-entry)
  (setf (gethash name table) new-entry))

(defun setprop (obj prop val)
  (setf (get obj prop) val))

(defun first-n (n parsed-seq)
  (let ((result nil))
    (dotimes (index n result)
      (setq result (append result (list (nth index parsed-seq))))))

(defun sizeof (type)
  (case type
    ((int bool)
     1)
    ((float) 1)))

(defun flat-1 (stmt-list)
  ;; flattens list one level
  (setq q (mapcar #'(lambda(s)
                      (if (listp s) s (list
                                       s)))
                 stmt-list)
        q (reduce #'append q)))

(defun new-temp ()
  (incf *current-temp*)
  (intern (concatenate 'string "TEMP" (prinl-to-string *current-temp*))))

(defun this-temp ()
  (intern (concatenate 'string "TEMP" (prinl-to-string *current-temp*))))

(defun copy (n)
  (cond
    ((listp n) (copy-list n))
    ((symbolp n) (copy-symbol n))
    ((hash-table-p n) (copy-hash n))
    (t (format t "error in copy: ~S is not a recognized type" n))))

(defun xcopy-sst (n)
  (create-sst-node (copy (sst-node-space n))
                   (copy (sst-node-rator n))
                   (mapcar #'copy (sst-node-rands n))
                   (copy (sst-node-xrefs n))))

(defun copy-hash (n)
  (let ((temphash (make-hash-table))
        (maphash #'(lambda (key val)
                    (write-hash-table temphash key val)))
        n)
    temphash))

---

temp-alloc.lisp

This file contains code for the Temp Allocation phase of the compiler.
Specifically, these are the routine which convert an arbitrary SST program
into an SST program containing at the top level only nodes which can be
translated into a single CM FORTRAN statement.
Temp allocation phase

Also called "Normalization pass".

(in-package 'user)

(defvar *normalization-more-decls* nil)

(defun normalize-sst-program (prog)
  (let ((decls (first prog))
        (sst (second prog))
        (new-sst nil))
    (setq *normalization-more-decls* nil)
    (setq new-sst (normalize-toplevel-sst-node sst))
    (list (append decls *normalization-more-decls* new-sst))))

(defun normalize-toplevel-sst-node (sst &key (in-progn nil))
  (if (sst-node-p sst)
      (case (sst-node-rator sst)
        (let ((new-sst-list
               (normalize-assign (sst-node-space sst) (lhs sst) (rhs sst))))
          (if in-progn
              new-sst-list
              (create-block (sst-node-space sst) new-sst-list (sst-node-cond sst))))
        ((phi) ;; Better be a control-flow phi.
           ;; Just ignore for now.
          (if in-progn (list sst) sst))
        ((iter)
          (let ((result (create-iter (sst-node-space sst)
                                      (first (sst-node-rands sst))
                                      (normalize-toplevel-sst-node (iter-exp sst) :in-progn nil)
                                      (sst-node-cond sst))))
            (if in-progn (list result) result))
        ((block)
          (let* ((result (mapcar #'(lambda (sst)
                                      (normalize-toplevel-sst-node sst :in-progn t))
                     (sst-node-rands sst)))
                  (result-block (create-block (sst-node-space sst) (apply #'append result)
                                             (sst-node-cond sst))))
              (if in-progn (list result-block) result-block))
        ((seq)
          (let* ((result (mapcar #'(lambda (sst)
                                      (normalize-toplevel-sst-node sst :in-progn nil)
                                      (sst-node-rands sst)))
                    (result-seq (create-seq (sst-node-space sst) (apply #'append result)
                                             (sst-node-cond sst))))
              (if in-progn (list result-seq) result-seq))
          (t (if in-progn (list sst) sst)))
            (if in-progn (list sst) sst))
      (normalize-expand (sst-node-rator sst) sst lhs-base-indices)
      ((+ - / *) abs)
      (let ((result (normalize-expr rhs (make-base-indices-list lhs space)))))
      (append (first result)
              (list (create-local space 'r (list lhs (second result)) cond))
            (create-local space 'r (list lhs rhs) cond)))
    (let ((result (normalize-expr-list lhs rhs-base-indices))
      (append (first result)
              (list (create-local space 'r (list lhs (second result)) cond))
            (create-local space 'r (list lhs rhs) cond)))
    (normalize-expand (sst-node-rator sst) sst lhs-base-indices)
    ((+ - / *) abs)
    (let ((result (normalize-expr-list lhs rhs-base-indices)))
      (append (first result)
              (list (create-local space 'r (list lhs (second result)) cond))
            (create-local space 'r (list lhs rhs) cond)))
    (defun normalize-assign (space lhs rhs &optional cond)
      ;; Returns a list of ssts. It is up to caller to patch them
      ;; into a prog, or create a seq/block, appropriately.
      (if (sst-node-p rhs)
          (let ((result (normalize-expr rhs (make-base-indices-list lhs space))))
            (append (first result)
                    (list (create-local space 'r (list lhs (second result)) cond))
                  (create-local space 'r (list lhs rhs) cond)))
        (defun normalize-expr (lst lhs-base-indices)
          ;; ssts to precede the one in which sst occurs, and whose second element
          ;; is the replacement for sst.
          ;; is the replacement for sst.
          (case (sst-node-rator sst)
            (rep add-scan nil-scan max-scan min-scan and-scan or-scan)
            (normalize-expand (sst-node-rator sst) sst lhs-base-indices)
            ((+ - / *) abs)
            (let ((new-rands nil)
                   (new-defs nil))
              (dolist (old-rand (sst-node-rands sst))
                (let ((result (normalize-rand old-rand lhs-base-indices)))
                  (setq new-defs (append new-defs (first result))))))
            (t (if in-progn (list sst) sst))
          (if (sst-node-p rhs)
              (let ((result (normalize-expr rhs (make-base-indices-list lhs space))))
                (append (first result)
                        (list (create-local space 'r (list lhs (second result)) cond))
                      (create-local space 'r (list lhs rhs) cond))))
        (defun normalize-expr-list (lhs rhs)
          ;; Returns a list of ssts. It is up to caller to patch them
          ;; into a prog, or create a seq/block, appropriately.
          (if (sst-node-p rhs)
              (let ((result (normalize-expr rhs (make-base-indices-list lhs space))))
                (append (first result)
                        (list (create-local space 'r (list lhs (second result)) cond))
                      (create-local space 'r (list lhs rhs) cond))))
            (t (if in-progn (list sst) sst)))
          (if (sst-node-p rhs)
              (let ((result (normalize-expr rhs (make-base-indices-list lhs space))))
                (append (first result)
                        (list (create-local space 'r (list lhs (second result)) cond))
                      (create-local space 'r (list lhs rhs) cond))))
        (defun normalize-assign (space lhs rhs &optional cond)
          ;; Returns a list of ssts. It is up to caller to patch them
          ;; into a prog, or create a seq/block, appropriately.
          (if (sst-node-p rhs)
              (let ((result (normalize-expr rhs (make-base-indices-list lhs space))))
                (append (first result)
                        (list (create-local space 'r (list lhs (second result)) cond))
                      (create-local space 'r (list lhs rhs) cond))))
        (t (if in-progn (list sst) sst)))
      (if in-progn (list result) result))
    (if in-progn (list sst) sst)))
    (if in-progn (list sst) sst)))
(setq new-rands (append new-rands (cdr result))))
(list new-defs
  (create-local (sst-node-space sst)
    (sst-node-rator sst)
    new-rands
    (sst-node-cond sst)))
)
(t (list '() sst)))

(defun normalize-rand (rand lhs-base-indices)
  ;; Returns a list whose first element is a list of new ssts to precede
  ;; the use of rand, and whose second element is the replacement for rand.
  (if (not (sst-node-p rand))
    (list '() rand)
    (case (sst-node-rator rand)
      ((add-scan mul-scan max-scan min-scan or-scan)
        ;; A scan is not a valid rand, since it becomes a utility
        ;; library call. It must be replaced with a temp.
        (let* ((temp-name (gensym "RAND_TEMP_"))
          (rand-space (sst-node-space rand))
          (temp-def (if (null lhs-base-indices)
                        (create-scalar rand-space temp-name)
                        (create-local rand-space 'aref
                                      (cons temp-name lhs-base-indices))))
          (temp-use (if (null lhs-base-indices)
                        (create-scalar rand-space temp-name)
                        (create-local rand-space 'aref
                                      (cons temp-name lhs-base-indices))))
          (norm-temp-assign (normalize-assign rand-space
temp-def
rand
(sst-node-cond rand))))
        (setq *normalization-more-decls*
          (cons (set-sym-entry temp-name
          (sst-node-space rand)
          (find-type rand))
          *normalization-more-decls*)))
          (list temp-def
          temp-use)
        )
      ((rep)
        (normalize-expand (sst-node-rator rand) rand
          lhs-base-indices (sst-node-cond rand))
        )
      ((+ - / * = abs lt gt le ge ne and or phi)
        (normalize-expr rand lhs-base-indices)
        )
      (t (list '() rand))))
)

(defun normalize-expand (thing xpand-sst lhs-base-indices &optional cond)
  ;; thing is either REP or a scan (ADD_SCAN, etc.).
  ;; Returns a list whose first element is a list of stmt-level ssts to precede
  ;; the one where xpand-sst occurs, and whose second element is the
  ;; replacement for xpand-sst.
  (let ((expr-to-expand (case thing
                            ((rep) (rep-exp xpand-sst))
                            ((add-scan mul-scan max-scan min-scan or-scan)
                            (scan-exp xpand-sst)))))
    (temp-name-root (case thing
                      ((rep) "REPTEMP_")
                      ((add-scan mul-scan max-scan min-scan and-scan or-scan)
                      "SCAN_TEMP_"))))
  (if (sst-node-p expr-to-expand)
    (case (sst-node-rator expr-to-expand)
      ((number scalar aref)
        (list '() xpand-sst))
      (t (let* ((temp-name (gensym temp-name-root))
          (temp-indices (remove-if #"\(^\(\lambda\) axis\)
          (member axis (xpand-axis xpand-sst))))
          (lhs-base-indices))
          (temp-def (if (null temp-indices)
                      (create-scalar (sst-node-space expr-to-expand)
temp-name)
                      (create-local (sst-node-space expr-to-expand)
                        'aref
                        (cons temp-name temp-indices))))
          (temp-use (if (null temp-indices)
                      (create-scalar (sst-node-space expr-to-expand)
temp-name)
                      (create-local (sst-node-space expr-to-expand)
                        'aref
                        (cons temp-name temp-indices))))
          (norm-temp-assign-sst (create-local (sst-node-space expr-to-expand)
                        (list temp-def expr-to-expand)
                        (sst-node-cond expr-to-expand))
            (norm-temp-assign-sst (normalize-assign (sst-node-space temp-assign-sst)
                        (rhs temp-assign-sst)
                        (lhs temp-assign-sst)
                        (sst-node-cond temp-assign-sst))))
          )))
      )))

93
(new-xpand-sst (create-sst-node (sst-node-space xpand-sst) thing (list (xpand-axis xpand-sst) temp-use) (sst-node-cond xpand-sst))))

;; Declare a temp to hold the expression, simultaneously
;; entering it in the symbol table.
(setq *normalization-more-decls*
  (cons (set-sym-entry temp-name
           (sst-node-space expr-to-expand)
           (find-type expr-to-expand)
           *normalization-more-decls*)
        (list norm-temp-assign-sst new-xpand-sst))))

(list '()
(list xpand-sst))

(defun find-type (expr)
  (cond ((sst-node-p expr)
         (case (sst-node-rator expr)
             ((aref)
              (let ((expr-sym-entry (get-sym-entry (aref-sym expr))
                        (sym-entry-type expr-sym-entry)))
               (if (null expr-sym-entry) 'some-damn-type
                   (sym-entry-type expr-sym-entry))))
             ((scalar)
              (let ((expr-sym-entry (get-sym-entry (scalar-sym expr))
                        (sym-entry-type expr-sym-entry)))
               (if (null expr-sym-entry) 'some-damn-type
                   (sym-entry-type expr-sym-entry))))
             ((number)
              (let ((the-number (number-val expr)))
               (cond ((floatp the-number) 'real)
                     ((integerp the-number) 'integer)
                     (t 'some-damn-type))
               (cond ((merge-types (mapcar #'find-type (sst-node-rands expr))))
                     ((add-scan mul-scan max-scan min-scan and-scan or-scan)
                      (find-type (scan-exp expr)))
                     ((rep)
                      (find-type (rep-exp expr)))))
               (cond ((eq expr 'T) 'logical)
                     (t 'some-damn-type)))))
    (merge-types type-list)
  (cond ((member 'real*8 type-list) 'real*8)
        ((member 'real type-list) 'real)
        ((member 'integer type-list) 'integer)
        ((member 'logical type-list) 'logical)
        (t 'some-damn-type))))

(defun make-base-indices-list (lhs space)
  (if (not (and (sst-node-p lhs) (is-aref lhs)))
      nil
      (let* ((lhs-indices (rest (sst-node-rands lhs)))
              (lhs-base-indices (extract-base-indices lhs-indices space)))
        (lhs-base-indices)))

(defun make-index-order-list (lhs space)
  (if (not (and (sst-node-p lhs) (is-aref lhs)))
      nil
      (let* ((lhs-indices (rest (sst-node-rands lhs)))
              (lhs-base-indices (extract-base-indices lhs-indices space))
              (order-list nil))
       (do (if (list (index lhs-base-indices)
                      (setq order-list (acons index (position index lhs-base-indices) order-list)))
            order-list))))

timer.lisp

;;; # "Subspace Compiler
;;; Version 1.0
;;; M.Eng. thesis under Professor William J. Dally
;;; Copyright (C) Todd O. Dampier, 1994.
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; This file contains code for dealing with timers.
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
;;; #
(defvar *f-timer-list* '())

(defun f-zap-all-timers () (setq *f-timer-list* ()))

(defun f-open-timer (timer-num &key (iter-var nil) (iters 1) (col 7))
  (if (or (< timer-num 0) (>= timer-num 64))
      (return-from f-open-timer
       (f-error "f-open-timer: Bad timer number -- ~S~%" timer-num)))
  (if (member timer-num (mapcar #'car *f-timer-list*))
      (progn
       (f-warning "f-open-timer: Timer -S already open. Resetting.~" timer-num)
       (f-close-timer timer-num :print nil)))
  (f-fresh-line)
  (f-output "call cm_timer_clear(~S)-%" timer-num :col col)
  (f-output "call cmtimerstart(~S)-%" timer-num :col col)
  (if (not (null iter-var))
      (if (member iter-var (mapcar #'cadr *f-timer-list*))
          (progn
           (f-warning "f-open-timer: Timing loop variable ~S already in use! No loop inserted.~" iter-var)
           (setf iter-var nil))
          (f-output "do ~S = 1," iter-var iters :col col)))
  (setf *f-timer-list* (cons (list timer-num iter-var iters) *f-timer-list*)))

(defun f-close-timer (timer-num &key (print t) (col 7))
  (if (or (< timer-num 0) (>= timer-num 64))
      (return-from f-close-timer
       (f-error "f-close-timer: Bad timer number -- ~S~%" timer-num)))
  (let ((timer-pos (position timer-num *f-timer-list* :key #'car :test #'equal)))
    (if (null timer-pos)
        (f-warning "f-close-timer: Timer ~S not open." timer-num)
      (let* ((timer-entry (nth timer-pos *f-timer-list*))
         (iter-var (cadr timer-entry)))
        (setf *f-timer-list* (remove timer-entry *f-timer-list*))
        (if (not (null iter-var))
            (f-output "enddo~%" :col col))))
  (f-fresh-line)
  (f-output "call cm_timer_stop(~S)-%" timer-num :col col)
  (if print
      (f-output "call cm_timer_print(~S)-%" timer-num :col col))
  (f-output "~%")))
References


