SZIN: An Object-Oriented Macro-Based System for Lattice Field Theory

Robert G. Edwards A. D. Kennedy Claus Vohwinkel

August 23, 1996

Contents

Chapter 1

Objectives and Goals

1.1 Overview

In order to explain the motivation for the SZIN macro system we need to discuss three topics: what is the long-term goal, how far does the current macro-based version get us towards this goal, and what does the name mean?

1.1.1 What's in a name?

The answer to the third question is the briefest: SZIN is the Hungarian word for "color," and one line of ancestry of the system is to a program called SZINHUR ("color string" in Hungarian) which measured Polyakov loops.

1.1.2 Long-term goals

Our long-term goal is to develop a lattice field theory system using a category theory approach, using a language called A#. In the context of programming languages a categorical system is similar to an object-oriented system, but more so. Let us explain first what such a system is, and then why it is desirable.

We are all familiar with the concept of a "data type" or "domain" from their use in languages such as C and FORTRAN. A sequence of bits may be interpreted as a floating point number (`float`) or a fixed precision integer (`int`), and the appropriate machine operations or subroutines are selected for addition, multiplication, trigonometric functions, printing the value, etc. The benefits of using such domains in the language are

- *Type safety.* The compiler will not let you implicitly carry out undefined operations: for instance, you cannot add a floating point number to a characted string.[1] If there is a meaningful conversion from one domain into another then some languages will sometimes convert the value automatically, while others require an explicit conversion operation. This implicit or explicit conversion usually changes the binary representation of the value: for example, compare the conversions (or "coercions") for i and j

  ```
  float x = 3.14;
  int i = (int) x;
  int j = * (int *) &x;
  ```

  in C.

- *Overloading.* In machine language the operations for integer addition and floating point addition are quite separate things, with different opcodes which are represented by different assembler mnemonics. It is convenient, intuitive, and in accord with common mathematical usage to be able to represent both operations with the symbol $+$ in expressions, and to select the intended operation on the basis of the types of the arguments. Observe that there is an inherent conflict between such overloading and automatic domain conversion. In most languages some built in operators and functions are overloaded[2] but user-defined functions cannot be. A little thought leads to the conclusion that a function is really identified by its *signature* rather than its name: for example the following are all distinguishable entities

  ```
  float f(float);
  float f(float, float);
  float f(int);
  int f(int);
  ```

[1] Except in PL/I, where `"20" + "1"` is `"21"`, but `"<20" + "1>"` is `"<201>"`.

[2] They are called "generic functions" in FORTRAN.

although C will not select amongst them on the basis of the argument and return types, and C++ refuses to allow overloading on the return type (so the last two are in conflict).

- *Portability.* One of the main advantages of not writing programs in assembler is that there is a small chance that the programs can be executed on more than one machine architecture. In particular, it is useful to distinguish a incorrect program from an incorrect implementation; it is a very useful debugging technique to compare the results of running a program using several different compilers on several different architectures. It is also very pleasant not to have to rewrite one's software from scratch everytime one uses a new machine architecture. Beyond this, it is also frustrating to have to make major changes in the details of a program just in order to change the domains used from `float` to `double` in some part of the program.

It is useful to consider in what sense it is natural to expect that converting a program from `float` to `double` should be an essentially trivial task, whereas converting it from `float` to `char` is not even a meaningful one. In a sense both `float` and `double` are approximate representations of the underlying mathematical entity of real numbers, and our programs are only making use of the properties of real numbers and not of the representation-specific details (e.g., the number of fraction bits). A *category* is thus defined to be a class of domains which have a set of operations defined on them, and satisfy a set of (usual implicit) axioms. We could define a category which has the properties needed for a typical program which makes use of `float`s as

```
define Field: Category == BasicType with

  1: %                        ++ Identity for multiplication.
  0: %                        ++ Identity for addition.

  +: % -> %                   ++ Identity function.
  +: (%, %) -> %              ++ Addition.

  -: % -> %                   ++ Negation.
  -: (%, %) -> %              ++ Subtraction.
```

```
  *: (%, %) -> %               ++ Multiplication.
  ^: (%, Integer) -> %         ++ Powers.

  inv: % -> %                  ++ Multiplicative inverse.
  /: (%, %) -> %;              ++ Division.

  default
    (a: %) - (b: %): % == a + (-b)
    +(x: %): % == x
    (a: %) / (b: %): % == a * inv b
```

in A#. What this means is that any domain (data type) which belongs to the category Field must provide an implementation of all the operations whose signature are listed. The symbol % stands for the whichever domain is currently being considered as belonging to Field, so if we specify that the domain float belongs to Field then we are requiring that float "knows" how to add two values of type float.[3]

We have no idea how the entities 0 or +: % -> % are represented, all we know is that they must exist and must satisfy all the (implicit) axioms, such as the fact that addition and multiplication are associative and commutative, and that multiplication is distributive over addition, and so forth. A typical function we may define is

```
f(F: Field, x: F, y: F): F == (x-y) / (x+y)
```

which may be invoked by

```
float a == 2.7
float b == 3.1
float c == f(float, a, b)
```

or alternatively one could globally parameterize the domain,

```
F == float
f(x: F, y: F): F == (x-y) / (x+y)
F a == 2.7
```

[3]Note that if float and double both belong to Field it is not required that addition be defined between a float and a double. The categorical structure implies that each of these domains has the structure of a Field, not that they are special cases of a Field.

```
F b == 3.1
F c = f(a, b)
```

To see the power of this approach for lattice field theory let us consider a more interesting example, a conjugate gradient linear equation solver.

```
--
-- Solve the linear equations  v = A * x  for  x
-- with residual < epsilon
--
conjugateGradient(F: Field, H: HilbertSpace(F),
  A: LinearOperator(H, F), v: B): B ==
    x: H := 0
    r: H := v - A * x                       -- residual
    rNorm: F := norm(r)
    p: H := r                               -- Kryllov vector
    while rNorm > epsilon repeat
      a: F := rNorm^2 / p*A*p
      x := x + a*p
      r := r - a*A*p
      (rNorm, oldRNorm) := (norm(r), rNorm)
      p := r + (rNorm/oldRNorm)^2 * p
    x
```

Consider just the statement

```
a: F := rNorm^2 / p*A*p
```

This says that the variable a, which belongs to the domain F, which in turn belongs to the category Field, is assigned the value of $||r||^2/\langle p, Ap\rangle$. The types of the quantities occuring are

```
rNorm: F                    -- by explicit declaration
2: Integer                  -- literal value
^: (F, Integer) -> F        -- from F because F: Field
p: H                        -- by explicit declaration
A: LinearOperator(H, F)     -- by explicit declaration
*: (LinearOperator(H, F), H) -> H
                            -- from LinearOperator(H, F)
```

```
  *: (H, H) -> F              -- from H because
                              --  H: HilbertSpace(F)
  /: (F, F) -> F              -- from F because F: Field
```

We see that the conjugate gradient function is expressed in terms of the operations which are guaranteed to exist and have the right properties because of the categories to which the argument types belong. It is most important that the categorical structure not only syntactically provides the declaration of the required operations, but also that is (implicitly) provides the axioms which enable one to prove that the algorithm is valid.

1.1.3 Summary

Just as subroutines provide a was of abstracting the functional behavior of an algorithm, categories provide a way of abstracting the minimum properties required of the objects upon which subroutines act.

We are all familiar with using a `sin` function from a library, and not wanting to know exactly how it works. The definition of what it does is much simpler than the details of how it does it. Likewise, we should write programs using an interface which specifies that an argument to some function must have the property of being `LinearOperator` over a `HilbertSpace(F)`, without knowing whether the linear operator is represented as a matrix, or is represented lazily as a function.

The specification of a category as opposed to a domain is useful because the former specifies the minimum properties required of its members, and users of it cannot make use of any other properties.

1.1.4 What exists at present?

A full implementation of a lattice field theory system using `A#` does not yet exist for a variety of reasons.[4] However we can persue the same ideas in simpler ways if we are willing to accept a less flexible system on grounds of expedience. We propose to use the macro-based system `SZIN` to develop the physics software for the 0.8 Teraflop/s machine, as we believe it provides significant advantages over the alternative approaches, and allows us to move towards the full categorical system described above.

[4]Principally being related to compiler bugs.

Chapter 2

Comparison with Other Approaches

2.1 Introduction

There are several packages available for lattice field theory programming, Canopy, MILC, APE, GIBBS[1], and C++-based systems, to name but a few. What are the advantages of using this macro system for lattice field theory programming?

One of the main differences between the various programming approaches is that of abstraction versus efficiency. On the one hand, proponents of a "high level" approach want to provide a system in which one can directly manipulate the "physical" objects one thinks of in the formulation of the physical problem. For example, in such as system applying the Dirac operator to a fermion field might be a "primitive operation." The intention is that

- It is much faster to write physics programs as one does not have to get involved in the low-level details such as explicit loops over color or space indices.
- The chance of making errors is much reduced (empirical studies supposedly show that the number of programming errors is proportional to the number of lines of code written, regardless of how much each line

[1] Whose main disadvantage is lack of existence.

actually does). Tools should be provided to manipulate and display high level data objects so that low-level debugging is not necessary.

- It is easy to reorganize the low-level data layout, which may be desirable when introducing new algorithms or otherwise extending the code. This is because the high-level code does not contain explicit reference to such structures, and indeed is only permitted to manipulate the "data objects" through a well-defined set of operations.

- It is impossible to construct meaningless expressions, such as multiplying a scalar field by just the first component of a color vector field. Such errors are flagged at compile time. Of course, it is still possible to write expressions which are built of valid operations but are still erroneous!

On the other hand, proponents of the "low level" approach claim that it is preferable to write lattice QCD programs directly in a low-level language, such as assembler, C, or Fortran. The arguments for this are that

- The "high level" systems produce inefficient code, and one cannot afford to squander 50% of the machine's performance in an application like QCD.

- The "high level" systems are complex, buggy, and not well-documented.

- The systems either produce incomprehensible low-level code as output, or spend most of their time executing code from some complicated run-time library. This makes low-level debugging very painful.

The approach outline in this document is intended to be a step in the direction of producing a "high level" system which will nevertheless produce efficient code. There are several reasons why this can be done.

- *Minimalism.* In antithesis to a system such as Canopy, which aims to give you the "richest" structure possible, our approach is to define the least structure which is absolutely necessary to specify algorithms. For example, Canopy lets you specify a very general lattice by specifying lists of neighbouring sites, whereas our approach is that high-level algorithms must be specified in terms of operations on fields such as pointwise operations, global sums, and application of the Dirac operator. We are thus able to implement these operations in a manner that is efficient for a specific computer (e.g., making use of a grid-like communications network). If it were desired to work on a BCC or random lattice then the appropriate "layer" of our hierarchical system would have to be modified, but the high-level code should remain untouched.

- *Macros.* One of the problems introduced by defining such a minimalist interface to the data objects provided is that the cost of an operation might depend on the specific operation performed. For example, an operation like summing up the values of a scalar field over all sites might be expensive on a scalar implementation for a random lattice, but might be very cheap on a parallel implementation of a regular hypercubic grid. In order to avoid a large overhead when using a very cheap operation the operations are expressed as (compile time) macros rather than (run time) subroutines.

2.2 Summary

Here is an outline of the advantages of our approach:

- *Data abstraction.*
  - *Type checking.* The macros will complain if meaningless operations are attempted. While macro systems are infamous for being bad at giving useful error messages[2] compared to a real compiler, our macros do a great deal of consistency checking.
  - *Operations on abstract data types.* The macros act on a family of data types, which can efficiently represent field of a variety of kinds on a variety of architectures.
  - *Operator overloading.*
  - *Hierarchical data structure.* The basic datatype used in the macro system is the following hierarchy:

[2] As anyone who uses TEX will testify to.

  * *Explicit array indices.* This level is used to provide explicit indices, and is used for example for the direction index of gauge fields. This is in violation of the spirit of the system, where the natural operations on such indices should be encapsulated within the macro system itself.
  * *Outer grid.* This level provides for looping over blocks of sites. On a serial computer this is where the site loop would live because it maximizes the locality of reference to memory. For the 0.8 Teraflop/s machine this is where the loop over sites on each node will live, for the same reason.
  * *Primitive.* This is where color indices live. The primitive can be `scalar`, `vector`, or `matrix`, specifying colorless objects, matter fields, and gauge fields respectively. Spinors are expressed as matrices, which is a poor man's way of handling the spin and color indices. We plan to switch spinors to their proper formulation of an outer product of a color vector and a spin vector.
  * *Reality.* This level handles complex fields.
  * *Inner grid.* This level allows looping of sites in the way needed for vector machines (i.e., the site loop is innermost), or machines with explicit parallel instructions. For the 0.8 Teraflop/s machine you can either view this level as doing nothing, or as being where the parallelism over all the nodes in the machine lives.
  * *Word.* This level specifies the underlying hardware data type to be used – `int`, `float`, etc.
  * *Expression optimizations.* While the user cannot construct expressions of type objects, several important optimizations like communications during a multiply are provided. The categorical properties of the operations allow for general applicability without the need for worrying about specific cases like a matrix or scalar multiply.

- *Memory management.*

- *Heap versus Stacks.* All `C` and `C++` based systems allocate storage using `malloc` and related functions.[3] This heap allocation strategy is not in general optimal either in time or space[4] which can be a serious problem for QCD where the amount of memory available is important both in determining the cost of the hardware and the largest problems that can be solved. `SZIN` uses a stack allocator (separate from the call frame stack) which is powerful enough for QCD, where the number and size of objects is fixed and known in advance. Another disease of the `C` and `C++` heap allocators is that they do not provide garbage collection, and thus it is easy to forget to free some object and thus gradually reduce the amount of available free memory.
- *Temporaries.* Because the macro syntax does not allow one to construct expressions, all temporaries must be allocated by the top-level program itself. While this is clear undesirable in the long term, it does allow for efficient and controlled storage management and reuse of temporaries where this is critical for fitting the problem into memory. Many of the macros do allocate there own temporaries, however. This is especially useful in cases where two macros have been merged into one like communications during a multiply.
- *Defects of* `C++`. `C++` does allow one to write expressions built out of operators acting on objects belonging to arbitrary classes.[5], and this means that one has to worry about where one saves intermediate results. Unless the compiler allocates space for all the temporaries required (which means that it has to know in advance how big they are) then they cannot be returned on the call frame stack. In `C++` the semantics of the language requires temporary values to be copied unecessarily. See 3.3 for a more detailed discussion.

[3] It is possible to allocate storage on the call-frame stack using `alloca`, but this is not a standard feature of either language.

[4] It is however necessary in general for applications which have to return dynamically sized objects, a behavior which is typical of `LISP` and computer algebra applications.

[5] A `class` is `C++` terminology for a type or domain.

- *Subroutine generation.* The default implementation of the package is to write the `C` code generated by the macros into unique files (modulo hash name collisions). This feature allows for compact code size on machines were memory is scarce. Also, the generated subroutines can be easily replaced with hand written highly optimized versions for additional speed.
- *Orthogonality.*
  - *Ease of debugging.*
  - *Speed of getting correct code.*
  - *Ease of modification.*
- *Macro implementation.*
- *Existence of lots of debugged code.*

Chapter 3

SZIN Reference Manual

3.1 History

The system evolved from a set of macros developed by Pietro Rossi at UCSD which used the standard `UNIX` macro processor `m4` and `FORTRAN` preprocessor `ratfor`. This system was used to generate most of the code used for the `CYBER-205` and `ETA-10` vector computers as well as compatible versions for workstations and `CRAY`s. It was completely rewritten by Tony Kennedy and Robert Edwards, and in its new(er) form was used to transfer our codes to the `CM-2`. At the same time most of the configuration generation code using the macros was rewritten by Tony Kennedy and Robert Edwards. It has been maintained by Robert Edwards, and large amounts of measurment code have been written using it by Robert Edwards and Urs Heller. Robert Edwards and Claus Vohwinkl modified it to produce `C` and generate subroutines; this has the desirable feature of obviating the need for `ratfor`, whose job was essentially only to satisfy `FORTRAN`'s arcane syntax requirements. The `C` code generation allows for easier internal optimizations like code motion hence more efficient code. The macro code generation was extended to include `C` subroutine generation producing reduced code size, ease of replacement, and allowing M4 time assembly code output.

3.2 Overview

The basic requirements for a high-level QCD programming system are that it allows fairly complicated programs to be written easily without sacrificing the ability to achieve maximal performance. To realize this we have been led to the following design philosophy:

- Innermost kernels are hand coded in assembler or microcode;
- Data structures are determined by the requirements of these kernels;
- High-level programs are expressed in a data-parallel way;
- No optimization is performed by the "compiler."

We do not attempt to impose a particular model of parallelism in the way that CANOPY does, or the MILC code uses PVM. Instead we provide a set of high-level primitives which allow the physics code to be written independently of the underlying hardware architecture, and the knowledge of the particular hardware being used is located in a few "machine specific" macros.

An example may serve to make this clearer. Consider the following fragment of high-level code

```
LATTICE_COMPLEX_COLOUR_COLOUR_CB_DIRECTION('U');
LATTICE_COMPLEX_COLOUR_COLOUR('V');
LATTICE_COMPLEX_COLOUR_COLOUR('W');

MULTIPLY(V, CONJUGATE(U(c,nu)), W, REPLACE);
```

This generates the code to do the matrix multiplication $W \leftarrow V \cdot U^{\dagger}_{c\mu}$. The declarations specify that U and V are fields (have a value on each lattice site) which are complex-valued, are $N_c \times N_c$ matrices in colour space, are partitioned into even and odd "checkerboard" sites, and live on each of the N_d links emanating from the site. With this information the `MULTIPLY` macro can verify that its arguments are compatible, and generate the appropriate code to carry out its operation. Exactly what this code is depends upon the target machine. For a scalar workstation if might generate something like

```
for ( x = 0 ; x < volume ; x++ )
  for ( c1 = 1 ; c1 <= 3 ; c1++ )
    for ( c2 = 1 ; c2 <= 3 ; c2++ )
    {
      W[x,c1,c2,0] = + V[x,c1,c3,0] * U[x,mu,c,c2,c3,0]
                     + V[x,c1,c3,1] * U[x,mu,c,c2,c3,1];
      W[x,c1,c2,1] = + V[x,c1,c3,1] * U[x,mu,c,c2,c3,0]
                     - V[x,c1,c3,0] * U[x,mu,c,c2,c3,1];
      for ( c3 = 2 ; c3 <= 3 ; c3++ )
      {
        W[x,c1,c2,0] += + V[x,c1,c3,0] * U[x,mu,c,c2,c3,0]
                        + V[x,c1,c3,1] * U[x,mu,c,c2,c3,1];
        W[x,c1,c2,1] += + V[x,c1,c3,1] * U[x,mu,c,c2,c3,0]
                        - V[x,c1,c3,0] * U[x,mu,c,c2,c3,1];
      }
    }
```

where there is an outer loop over lattice sites and a bunch of inner colour loops. We have written in an idealized `C` where we have simple multidimensional arrays solely in order to simplify the presentation. Notice that the macro has "optimized" the matrix multiplication by assigning the result of the first term in the colour multiplication to W and then incremented it for each successive term, rather than initializing W with zero and the adding each term to it. This "optimization" does not involve any clever analysis of the source program by the "compiler," it is directly specified by the `REPLACE` option specified as the last argument to the `MULTIPLY` macro.

On a vector machine the generated code might look like

```
for ( c1 = 1 ; c1 <= 3 ; c1++ )
  for ( c2 = 1 ; c2 <= 3 ; c2++ )
  {
    for ( x = 0 ; x < volume ; x++ )
      W[c1,c2,0,x] = + V[c1,c3,0,x] * U[mu,c,c2,c3,0,x]
                     + V[c1,c3,1,x] * U[mu,c,c2,c3,1,x];
    for ( x = 0 ; x < volume ; x++ )
      W[c1,c2,1,x] = + V[c1,c3,1,x] * U[mu,c,c2,c3,0,x]
                     - V[c1,c3,0,x] * U[mu,c,c2,c3,1,x];
    for ( c3 = 2 ; c3 <= 3 ; c3++ )
```

```
    {
      for ( x = 0 ; x < volume ; x++ )
        W[c1,c2,0,x] += + V[c1,c3,0,x] * U[mu,c,c2,c3,0,x]
                        + V[c1,c3,1,x] * U[mu,c,c2,c3,1,x];
      for ( x = 0 ; x < volume ; x++ )
        W[c1,c2,1,x] += + V[c1,c3,1,x] * U[mu,c,c2,c3,0,x]
                        - V[c1,c3,0,x] * U[mu,c,c2,c3,1,x];
    }
  }
```

where the site loop is now innermost and the site subscript is the last (fastest varying). This would allow a dumb vectorizer to recognize that vector instructions could be used to vectorize over sites in an efficient manner. It would be trivial, of course, to get the macros to generate explicit vector or parallel syntax for this case, or to insert the appropriate vectorization directives.

The important points to glean from this example are

- The knowledge of the data layout and preferred loop ordering is localized in the `MULTIPLY` macro, and does not modify the "physics" source code in any way;

- In order to produce efficient code for the workstation (which wants all the data associated with a given site to be nearby in memory so as to avoid cache misses) and the vector computer (which wants all the data organized vectors of contiguous sites so it can stream them through its vector units) not only does the order of loops have to be altered, but the data layout has to be modified also.

This last observation gets to the heart of OOPS. In languages such as `FORTRAN` and `C` you can use a subroutine to hide the details of the implementation of a particular function — we are all happy to use a built-in exp function and assume that someone has written it in a way that is accurate and efficient for the taget machine — but when the interface to the subroutine is more complicated that a simple number then the subroutine hides only the *operations* done to implement the function, but does not hide the *data structures* used by this implementation. The goal of an OOPS is to provide a *class* or *domain* which — to the outside world — is a "black box" with a set of public functions available to manipulate it.

In our macro system the declaration macro

```
LATTICE_COMPLEX_COLOUR_COLOUR('W')
```

declares that `W` will be the name of a member of the class of complex matrix-valued fields over the lattice. The only thing you can do with `W` is to give it as an argument to a macro; for instance you can multiply it by something, take its trace, or even write it into a binary or ascii file. What you cannot do is to get the imaginary part of the $(3,3)$ component of `W` at the site with coordinates $(1,1,1,2)$ (unless a suitable "indexing" macro has been provided). One reason for this is that the low level implementation may not explicitly store this component: it might have chosen to represent $SU(3)$ matrices by their first two rows, and thus the putative indexing macro would have to do some computation to deduce what the $(3,3)$ component should be. Another reason is that the low level implementation by be expressed in terms of parallel or vector primitives, and accessing individual elements might be orders of magnitude slower than the high level macro operations provided.

The data structure obfuscation provided by the macro system, and by OOPS in general, not only allows the high level code to be portable between different machines but also allows the low level implementation to be changed *post facto*. Even if our intention is to generate code for a single target machine it has proved valuable in the past to be able to generate scalar code to run on a workstation for purposes of debugging the logic of the high level program on tiny lattices. We have also had occasion to choose very strange ways of representing lattice fields in order to make the hand-coded kernels as efficient as possible: on the `CM-2` we chose to store the $SU(3)$ matrices as `imaginary part, real part, negated imaginary part` because this allowed us to produce much longer unbroken pipelines for complex matrix-vector multiplication where changing from an add operation to a `subtract` operation would have entailed breaking the pipeline. Clearly, we do not want such perverse data layouts to infect the high level code. Furthermore, we want the freedom to change the underlying data layout in the future when someone thinks of a better way of doing things (or we want to use a "clover inverter" without rewriting all the measurement code which uses the inverter).

The difficult part of writing an OOPS system for field theory is in choosing a "natural" set of primitive operations to act on the abstract "objects" belonging to the "classes" we have defined. We need operations which are flexible enough to allow the top level programs to implement complicated lattice operators, but which are sufficiently opaque so as not to preclude efficient implementation.

In summary, the goals of the system are

- Data structure independence;
- High-level operations;
- Abstraction of parallelism;
- Portability.

3.3 Comparison with C++

We have seriously considered reimplementing the macro system in C++. There appear to be too many limitations in the C++ language to make this approach useful — in practice one always seems forced to lose the advantages of the OO nature of the language in order to implement the mathematics of field theories in a natural fashion.

3.3.1 Inefficiency

Suppose we try to define a class of matrices which implements the natural linear algebra operations. We would naïvely write

```
#include <complex.h>

class Matrix
{
  private:

    // For this example we use 3x3 complex matrices

    complex array[3][3];

  public:
```

```
    ...

    Matrix& operator*(const Matrix& y) const
    {
      Matrix& x = new Matrix();
      for ( int r = 0 ; r < 3 ; r++ )
        for ( int c = 0 ; c < 3 ; c++ )
          {
            x.array[r][c] = array[r][0] * y.array[0][c];
            for ( int rc = 1 ; rc < n ; rc++ )
              x.array[r][c] += array[r][rc] * y.array[rc][c];
          }
      return x;
    }

    ...
}

Matrix a, b, c;
...
a = b*c;
...
```

but in standard C++ this would generate a temporary of type `Matrix`, multiply `b` by `c` placing the result in the temporary, and then copy the entire matrix temporary into its final resting place `a`. This is required because C++ does not know what the possibly user-defined copy constructor (i.e., the definition of the assignment operator `=`) will do [see also *The Annotated* C++ *Reference Manual* by Stroustrup and Ellis, §13.4.2 item 4 (p. 333)].

One can circumvent this is many ways, but none of them are satisfactory.

- Defining an object of class `Matrix` to be a pointer to the actual array of data values makes the extraneous copy operation cheap, but now the matrix multiplication function overloading the operator `+` is forced to allocate the storage for the product matrix itself. Since it has no way of knowing the extent of the product (i.e., how long a live pointer to the product matrix will be around) it has to allocate the space on the heap, and this space will never be recovered until the program ends. Of course, one could recover the space using a garbage collector, but the overheads of this are most undesirable.

- Defining a function `MatrixMultiply(a,b,c)` where a pointer to the space preallocated for the result is explicitly passed in as an argument. This is the method used by `SZIN`, but one cannot then use the operator `*` and construct expressions from it.

- One could make `operator*` just build a small object on the heap which indicates that its arguments are to be multiplied, and then arrange that these objects are linked together into an expression tree. The final assigment operator then does all the evaluation. This is quite nice, as it allows some "automatic" optimizations to be carried out (e.g., only calculating the diagonal elements of a product if we know that the result is only going to be used as the argument of a `trace` operation), but the expression parsing is done at run time instead of compile time, and the expression tree on the heap has to be cleaned up with all the usual difficulties of working out when this can be done safely.

3.4 Comparison with `A#`

We shall just briefly indicate some of the features of `A#` which `C++` is lacking.

The concept of *inheritance* in `C++` is an approximation to the concept of domains belonging to categories in `A#`. Suppose we want classes `SU2` and `E8` both to inherit class `group`, where `group` provides the operations of multiplication, inverse, and so forth. We now want to write a routine which implements an algorithm which just needs its arguments to be group elements, and works equally well for $SU(2)$ and $E(8)$. In `C++` the type system does not allow us to check at compile time that all the arguments belong to the *same* group. It also does not allow us to return a result which belongs to the same group as the arguments.

3.5 Datatype structure

The basic type declaration in SZIN is DEFINE_TYPE, which defines a hierarchical type. Its general form is

$$\texttt{DEFINE\_TYPE(}\langle name\rangle, \langle word\ type\rangle, \langle inner\ grid\rangle, \langle reality\rangle, \langle primitive\rangle,$$

$$\langle outer\ grid\rangle, \langle array\ index\rangle, \ldots)$$

and a type declaration of the form

```
LATTICE_COMPLEX_COLOUR_COLOUR_CB_DIRECTION(`U')
```

just expands into

```
DEFINE_TYPE(`U',`REAL_32',`LATTICE',`COMPLEX_64',
              `MATRIX(`Nc',`Nc')',`SCALAR',`2',`Nd')
```

for a vector machine and

```
DEFINE_TYPE(`U',`REAL_32',`SCALAR',`COMPLEX_64',
              `MATRIX(`Nc',`Nc')',`LATTICE',`2',`Nd')
```

for the 0.8 Teraflop/s machine.

The possible arguments for DEFINE_TYPE are

- *Word types.*

  ```
  UNSIGNED_32
  INTEGER_32
  REAL_32
  REAL_64
  BOOLEAN
  ```

- *Inner grid types.*

  ```
  SCALAR
  ILATTICE
  ```

- *Reality types.*

```
SCALAR
COMPLEX_64
```

- *Primitive types.*

```
SCALAR
SEED
TRIANG(<dimension>)
VECTOR(<dimension>)
MATRIX(<dimension1>,<dimension2>)
```

- *Outer grid types.*

```
SCALAR
OLATTICE
```

- *Array types.* Up to 5[1] array indices are allowed, they are 0-based, and the arguments to `DEFINE_TYPE` are their dimensions.

3.6 Operations

The currently defined macro operations are

```
ALLOCATE
FREE

#+
# BINARYOP operations:
#   BINARYOP ( <source1>, <source2>, <target>, <option> )
#
```

[1] This is easily extended

```
# <option> :== ADD | SUBTRACT
#-

#+
# BINOPS operations:
#   <function> ( <source1>, <source2>, <target> )
#
# where <function> := MOD, MIN, MAX, DIVIDE, AND, OR, XOR
#-

#+
# CAST operations: CAST ( <source>, <target> )
#
# Makes explicit the indices of a primitive type
# or as an overloading hides the indices.
#
# Ex.  Matrix(Nc,Nc) -> Array(Nc,Nc)
#  or  Array(Nc,Nc)  -> Matrix(Nc,Nc)
#-

#+
# CMPLX operations:
#   CMPLX ( <source1>, <source2>, <target>, <option> )
#
# where <option> := REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# COERCE operations:
#   <function> ( <source>, <target> )
#
# where <function> := FLOAT, INT, BOOL, FLOOR,
#                     CEILING, TRUNCATE, ROUND
#-

#+
# COMPARE operations:
```

```
#   <function> ( <source1>, <source2>, <target> )
#
# where <function> := LT, GT, LE, GE, EQ, NE
#

#+
# CONJG operations:
#   CONJG ( <source>, <target>, <option> )
#
# where <option> := REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# COORD operations: COORD ( <source>, <target> )
#
# where <source> := scalar integer indicating
#                   direction (between 0 and Nd-1)
# and   <target> := lattice integer filled with
#                   coord. along <source> dir.
#-

#+
# COPY operations:
#   COPY ( <source>, <target>, <option> )
#
# where <option> := REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# COPYMASK operations:
#   COPYMASK ( <source>, <mask>, <target>, <option> )
#
# where <option> := REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# FILL operations: FILL ( <target>, <value> )
```

```
#
# Ex. <value> can be a lattice scalar of the same
#     primitive type as <target> (a lattice type).
# Or  <value> can be a lattice scalar of a lower
#     primitive type of <target> (a lattice type),
#     so fill a matrix with a diagonal of <value>
# Or  <value> can be a special symbol like
#     ZERO,ONE,RANDOM,GAUSSIAN.  If <target> is a
#     Matrix, then filled with a diagonal matrix.
#     If <target> has Matrix primitive, then
#     RANDOM,GAUSSIAN fill the entire matrix
#-

#+
# INDEXING operations:
#   INDEXING ( <source1>, <coordinate>, <target> )
#-

#+
# IO operations: these come in three varieties,
# binary (machine-dependent configuration files),
# NAMELIST (portable machine-readable data files), and
# terminal (human but not machine readable).
#
# CREATE_BINARY(unit_num, "filename"),
# OPEN_BINARY(integer, "filename"),
# WRITE_BINARY, READ_BINARY, CLOSE_BINARY(unit_num)
#
# WRITE_NAMELIST
#
# READ_TERMINAL, WRITE_TERMINAL
#-

#+
# MULTIPLY
#
# operations: MULTIPLY ( <source1>, <source2>, <target>, <option> )
```

```
#
# The allowed forms of source1 and source2 are
#   source
#   CONJUGATE(source)
#   COMMUNICATE(source,<cb>,<sgn>,<dir>)
#   CONJUGATE(COMMUNICATE(source,<cb>,<sgn>,<dir>))
#   COMMUNICATE(CONJUGATE(source),<cb>,<sgn>,<dir>)
#
# If a COMMUNICATE is used, then  <cb>  must be specified with the corresponding
#   sign and direction
#
# If both sources use a COMMUNICATE, then the CB must be the same
# between them. The sgn and dir are allowed to be different.
#
# CONJUGATE or COMMUNICATE is not allowed on the target
#
# The option must always be specified and is
# <option> :== REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# NEIGHBOUR_ALL ( <source>, <target>,
#                 <checker board>, <duplex> )
#
# If <duplex> = FORWARD or BACKWARD this is logically
# equivalent to
#
#   FOR <direction> FROM 0 THROUGH Nd-1 DO
#     { NEIGHBOUR ( <source>(<direction>),
#                   <target>(<direction>),
#                   <checker board>, <duplex>, <direction> );
#
# whereas if <duplex> = BOTH it is logically equivalent to
#
#   FOR <direction> FROM 0 THROUGH Nd-1 DO
#     { NEIGHBOUR ( <source>(<direction>,F_SWAP),
#                   <target>(<direction>,F_SWAP),
```

```
#                   <checker board>, FORWARD, <direction> );
#       NEIGHBOUR ( <source>(<direction>,B_SWAP),
#                   <target>(<direction>,B_SWAP),
#                   <checker board>, BACKWARD, <direction> ); }
#
# This allows optimized communication by getting the neighbours
# in all directions at the same time (e.g., by using "Stencils"
# on the CM-2). The <checker board> argument is as for
# NEIGHBOUR, but the <source> and <target> must now be
# declared to be of the same PRIMITIVE type as <source>
# but with an ARRAY index of size Nd. For <duplex> = BOTH
# they must also have an array index of size 2.
#
# NEIGHBOUR ( <source>, <target>, <checker board>,
#             <duplex>, <direction>[,<log2 distance>] )
#
# This carries out the operation
#
#   FORALL x DO
#     <target>[x, <checker board>] :=
#         <source>[x' <duplex> <direction>,
#                  1 - <checker board>]
#
# where <direction> stands for a unit vector or index as
# appropriate (the usual abuse of notation), and
# (anti)periodic boundary conditions are implied. <source>
# and <target> must have trivial ARRAY types (i.e., either
# they have no ARRAY dimensions at all, or they are completely
# indexed). Here, x' is the source coordinate after shifting
# by 2^distance given in log2_distance. This means that if
# log2_distance = 0, then only nearest neighbour communication
# is used and for log2_distance > 0, neighbours from a power
# of 2 are used. Note, in this case, then the source and
# target are on the same checkerboard. Arguments <duplex>,
# <checker board>, <direction> and <log2 distance> are all
# simple FORTRAN INTEGER variables, with <checker board> = 0|1,
# <duplex> = FORWARD|BACKWARD, and 0 <= j < Nd.
```

```
#
# All WORD, REALITY and PRIMITIVE types are supported.
# Obviously, only LATTICE GRID type is supported
#-

#+
# PARTOF operations: PARTOF ( <source>, <target>, <function> )
#
# where <function> := REAL_PART_OF and IMAG_PART_OF
#-

#+
# SEED_TO_FLOAT operations:
#   SEED_TO_FLOAT ( <source>, <target> )
#
# Converts seeds to floating point numbers
#-

#+
# SLICE_SUMSQ operations:
#   SLICE_SUMSQ ( <source>, <orthog direction>,
#                 <target>, <option> )
#   SLICE_SUM ( <source>, <orthog_direction>,
#               <target>, <option> )
#
# where <option> ::= REPLACE | NEGATE | ADD | SUBTRACT
# NOTE: this macro could be generalized to a list or
# orthogonal directions (a plane, a cube, etc)


#+
# Special purpose macros for manipulating spinors
#-
#+
# SPIN_PROJECT ( <source>, <direction>,
#                <PLUS/MINUS>, <target> )
#
```

```
# This carries out the operation
#
#   <target> := (1 <+/-> gamma[<direction>]) * <source>
#
# where <source> is of PRIMITIVE type MATRIX[<colour>,Ns]
# and <target> is of PRIMITIVE type MATRIX[<colour>,Ns/2]
# (i.e, COLOUR_SPIN and COLOUR_HALFSPIN respectively).
# <source> and <target> must have REALITY type COMPLEX_64,
# and conformant GRID and WORD types.
#
# The projection operator (1 <+/-> gamma[<direction>])
# is specified by the arguments <+/-> and <direction>
# both of which are simple FORTRAN INTEGER variables,
# with <+/-> = +/-1 and 0 <= j < Nd.
#-

#+
# SPIN_PRODUCT ( <source>, <Dirac algebra element>,
#                <target> )
#
# This carries out the operation
#
#   <target> := Gamma[<Dirac algebra element>] * <source>
#
# where <source> and <target> are of PRIMITIVE type
# MATRIX[<colour>,Ns] and (i.e, COLOUR_SPIN). <source>
# and <target> must have REALITY type COMPLEX_64, and
# conformant GRID and WORD types.
#
# The argument <Dirac algebra element> is a simple
# FORTRAN INTEGER variable, lying between 0 and 15 inclusive.
#-


#+
# TRICK ( <source>, <direction>, <PLUS/MINUS>,
#         <target>, <option> )
```

```
#
# <source> is of PRIMITIVE type MATRIX[<colour>,Ns/2]
# and <target> is of PRIMITIVE type MATRIX[<colour>,Ns]
# (i.e, COLOUR_HALFSPIN and COLOUR_SPIN respectively).
# <source> and <target> must have REALITY type COMPLEX_64, and
# conformant GRID and WORD types. Arguments <+/-> and
# <direction> are both simple FORTRAN INTEGER variables,
# with <+/-> = +/-1 and 0 <= j < Nd. The last argument
# specifies how <target> is to be modified; the choices are
#
#     <option> :== REPLACE | NEGATE | ADD | SUBTRACT
#
# The TRICK operation makes use of the fact that
# (1 <+/-> gamma[<direction>]) is a projection operator,
# and thus in a suitable spinor basis can be implemented
# using only the top two components of the spinors
# (SPIN_PROJECT) with the bottom two components being
# reconstructed by TRICK:
#
#     <target>[1] = <source>[1]
#     <target>[2] = <source>[2]
#     <target>[3] = a * <source>[1] + b * <source>[2]
#     <target>[4] = c * <source>[1] + d * <source>[4]
#
# with a and b being suitable constants from the set
# {+/-1,+/-i}.
#-

#+
# SUMSQ operations: SUMSQ ( <source>, <target>, <option> )
#                   SUM ( <source>, <target>, <option> )
#
# where <option> ::= REPLACE | NEGATE | ADD | SUBTRACT
#-

#+
# Special purpose macros for manipulating SU(N) matrices
```

```
#
# SU2_EXTRACT ( <source>, <su2 index>, <b0>, <b1>, <b2>, <b3> )
#
# Compute the b(k) of A_SU(2) = b0 + i sum_k bk sigma_k */
#
#
# SUN_FILL ( <b0>, <b1>, <b2>, <b3>, <su2 index>, <target> )
#
# Fill in B from B_SU(2) = b0 + i sum_k bk sigma_k
#-

#+
# TRACE operations: TRACE ( <source>, <target>, <option> )
#
# where <option> ::=
#   REAL_PART | IMAGINARY_PART | COMPLEX_PART
#-

#+
# TRACE_MULTIPLY operations:
#   TRACE_MULTIPLY ( <source1>, <source2>,
#                    <target>, <option> )
#
# where <option> ::= REAL_PART | IMAGINARY_PART | COMPLEX_PART
#-

#+
# UNARY operations: <function> ( <source>, <target> )
#
# where <function> := ABS, EXP, COS, SIN, LOG, ACOS,
#                     SQRT, ASIN, TAN, ATAN, NOT, SIGN
#-
```

3.7 Example

```
/* @(#)polylp.m 11.2 (8/22/96) */
```

```
/* POLYLP -- calculate the global normalized sum of the Polyakov loop */
/* u -- gauge field (Read) */
/* poly_loop -- Polyakov loop average in direction mu (Write) */
/* mu  -- direction of Polyakov loop (Read) */

include(types.mh)

SUBROUTINE(polylp, u, poly_loop, mu)

LATTICE_COMPLEX_COLOUR_COLOUR_CB_DIRECTION(u);
DCOMPLEX(poly_loop);
INTEGER(mu);
{ /* Local Variables */
  include(COMMON_DECLARATIONS);

  LATTICE_COMPLEX_COLOUR_COLOUR(u_dble);
  LATTICE_COMPLEX_COLOUR_COLOUR(poly);
  LATTICE_COMPLEX_COLOUR_COLOUR(tmp_1);
  LATTICE_COMPLEX(poly_trace);
  INTEGER(cb);
  INTEGER(n);
  DOUBLE(dummy);

  START_CODE;

  ALLOCATE(u_dble);
  ALLOCATE(poly);
  ALLOCATE(tmp_1);
  ALLOCATE(poly_trace);

  cb = 0; /* Start from even checkerboard */

  /* Construct double links */
  MULTIPLY(u(cb,mu), COMMUNICATE(u(1-cb,mu),cb,FORWARD,mu), u_dble, REPLACE);

  /* Multiply together to get the Polyakov loop */
  COPY(u_dble, poly, REPLACE);
```

```
  for(n = 2; n <= VALUE(nrow(mu))/2; ++n) /* runs over half the linear size */
  {
    NEIGHBOUR(poly, tmp_1, cb, FORWARD, mu, 1);
    MULTIPLY(u_dble, tmp_1, poly, REPLACE);
  }

  /* Take the trace and sum up */
  TRACE(poly, poly_trace, COMPLEX_PART);
  SUM(poly_trace, poly_loop, REPLACE);

  FREE(poly_trace);
  FREE(tmp_1);
  FREE(poly);
  FREE(u_dble);

  dummy = TO_DOUBLE(1) / TO_DOUBLE(Nc*vol_cb);
  MULTIPLY(poly_loop, dummy, poly_loop, REPLACE);

  END_CODE;
}
```