Comparing Id and Haskell in a Monte Carlo photon transport code

JEFFREY HAMMES
Colorado State University, CO, USA
(e-mail: hammes@cs.colostate.edu, jhammes@lanl.gov*)*

OLAF LUBECK
Los Alamos National Laboratory, CA, USA
(e-mail: oml@lanl.gov*)*

WIM BÖHM
Colorado State University, CO, USA
(e-mail: bohm@cs.colostate.edu*)*

Abstract

In this paper we present functional Id and Haskell versions of a large Monte Carlo radiation transport code, and compare the two languages with respect to their expressiveness. Monte Carlo transport simulation exercises such abilities as parsing, input/output, recursive data structures and traditional number crunching, which makes it a good test problem for languages and compilers. Using some code examples, we compare the programming styles encouraged by the two languages. In particular, we discuss the effect of laziness on programming style. We point out that resource management problems currently prevent running realistically large problem sizes in the functional versions of the code.

Capsule Review

The Monte Carlo technique has a long history. Its importance has grown in tandem with the availability of cheap computing power. The authors outline the functionality of a large Monte Carlo simulation program, and demonstrate that a simplified kernel version can be cleanly coded in a functional style. They illustrate some effects of functional language implementation on programming style.

It is characteristic of the Monte Carlo method that code validation and debugging depend on high-statistics results. The authors frankly describe the problems encountered in obtaining such results from the functional codes. Their experiences highlight the need for future research to address specific implementation problems. Chief among these needs are effective debugging tools for inspecting partial results and efficient yet unobtrusive methods of memory management. Reports of this kind provide important empirical data on the practice of functional programming that can help guide both application development and language support research.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

1 Introduction

This paper presents to the functional language community a large Monte Carlo radiation transport code for language comparison and benchmarking. We also compare the functional languages Id (Nikhil, 1990) and Haskell (Hudak *et al.*, 1992) with respect to their expressiveness when used to write a complex scientific code, and point out the deficiencies which currently prevent this program from executing realistically large problems. We describe purely functional Id and Haskell versions of the photon transport simulation code based on the MCNP Fortran code of Los Alamos National Laboratory (LANL) (Briesmeister, 1993). The Id code discussed here uses no explicit I- or M-structures. Its only side-effecting is that which is required to do input/output.

Monte Carlo photon transport simulation is a good test problem for languages and compilers. It exercises such abilities as parsing, input/output, recursive data structures and traditional number crunching. While it contains high task parallelism, its Monte Carlo nature makes execution irregular and highly dependent upon the user's problem specification. Our codes should therefore be useful as benchmarks for compilers, run-time systems and machines, and they reveal the capabilities and deficiencies of a programming language and its implementation.

The MCP-Id code was developed using MIT's Id-World version 115 (Morais, 1986), which includes a compiler, a simulator called 'MINT', and the ability to execute compiled code on the Motorola/MIT Monsoon dataflow machine (Hicks *et al.*, 1993); the later stages of development were done on the 16-node Monsoon at LANL. The MCP-Haskell code was developed using the Chalmers Haskell B compiler, version 0.999.5, and was run on SUN SPARC machines.

The Id and Haskell codes are structured very similarly. The main differences lie in the way input/output is handled. The programming styles differ primarily in that Id encourages the use of loops and arrays, because of efficiency and resource management reasons, whereas Haskell programs tend to use higher order function composition and lists. Resource management problems severely limit the problem size that both the Haskell and Id versions of the code can run.

The rest of the paper is organized as follows. In section 2 we describe the Monte Carlo photon transport simulation problem. In section 3 the program structure is introduced. In section 4 the Haskell and Id programming styles are discussed. In section 5 the Haskell and Id codes are compared using some example code fragments. Section 6 deals with resource management issues. In section 7 we conclude and discuss future work.

2 Problem description

Throughout this paper, the name 'MCNP' refers to the original Fortran code developed at Los Alamos over many years, which simulates the transport of photons, neutrons and electrons. MCNP is a very general code which allows a myriad of parameter settings and applications, and is used at over 100 sites worldwide. The name 'MCP-Functional' refers to the photon transport codes which are based on

MCNP and written in a pure functional way, called 'MCP-Id' for the functional Id code and 'MCP-Haskell' for the Haskell version.

2.1 Overview of Monte Carlo photon transport

The Monte Carlo radiation transport problem involves simulating the statistical behavior of certain particles while they travel through objects of specified shapes, consisting of certain materials. It is used to model problems in areas such as nuclear reactors, radiation shielding and medical physics (Whalen *et al.*, 1991). MCP-Functional deals only with photon radiation. Simulating the behaviour of particles and their collisions with the nuclei of the material that the particles travel through is called *tracking*. A *nucleus* is the kernel of a specific atom, and a *nuclide* is a certain element. While tracking takes place, statistical information about certain events is gathered in histograms in a process called *tallying*. The user of a Monte Carlo transport code describes a problem geometry, a radiation source and the information that needs to be tallied. The program simulates individual source photons as they collide with the nuclei of the materials in the problem geometry, and each photon contributes information to the tallies being collected. The accuracy of the tally results is proportional to the square root of the number of source particles simulated. Figure 1 shows an example of a photon track. Each intermediate event is a collision or a surface crossing. The exact nature of these events is described in the following sections. Because photon *splitting* can occur, a source photon's track forms an irregular tree. Track lengths and the degree of splitting are highly dependent on the the user's problem specification.

2.2 Photon tracking

The essence of photon simulation is the tracking of a photon through a series of movements and collisions with nuclei. Each collision has the potential for absorbing some of the photon's energy and changing the photon's direction of travel. It is also possible for a photon to split into multiple photons, each of which must be separately tracked. Splitting can be either physical (see section 2.2.1) or statistical (see section 2.2.2). Eventually, every photon's track ends due to one of three reasons: its energy or weight falls below a specified threshold, the photon reaches a region of little or no interest with regard to the tallies being collected, or the photon is absorbed by a nucleus.

2.2.1 Collisions

There are four kinds of collision possible when a photon interacts with a nucleus:

1. Compton (incoherent) scatter, in which a new angle and energy result.
2. Thomson (coherent) scatter, in which a new angle but unchanged energy result.
3. Pair production, in which the photon is absorbed and two new photons are produced, each with 0.511008 Mev of energy and opposite directions.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

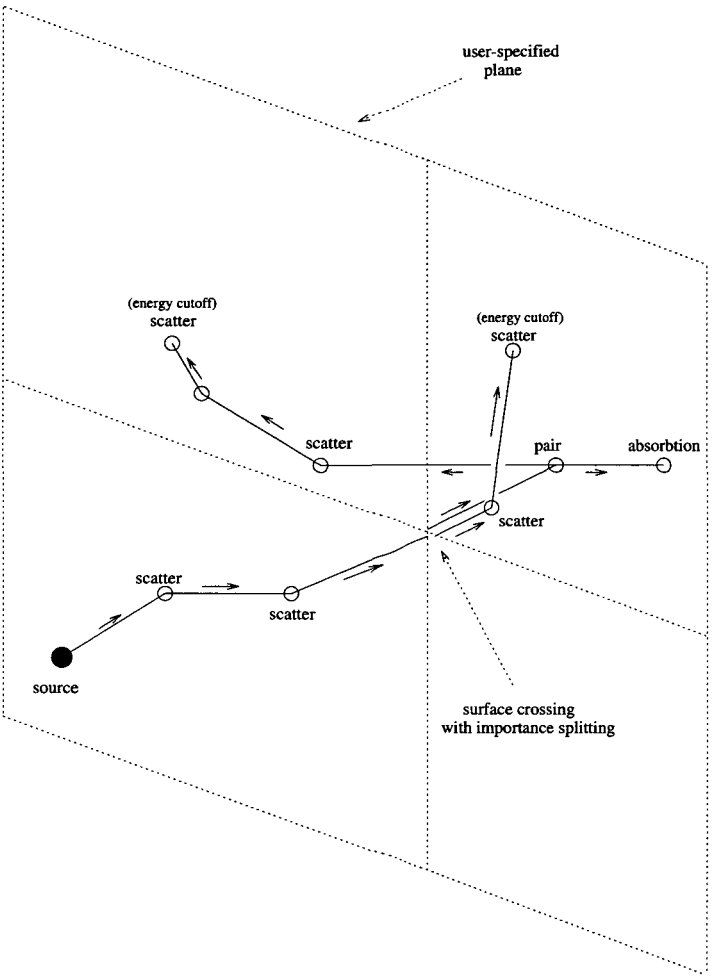


Fig. 1. Example of a source photon's track

4. Photoelectric effect, in which zero, one or two new photons may be produced by photofluorescence.

It is possible to specify different physics treatments, *simple* and *detailed*, to handle collisions. The simple model ignores coherent scattering and photofluorescence, and uses rougher (and computationally faster) approximations in computing the new energy and scatter angle for an incoherent scatter. This simple treatment can be useful for high-energy photons but is not accurate enough for low energies or heavy nuclides. The user can specify, based on an understanding of the problem being simulated, a threshold energy below which the simulation will switch to detailed physics routines.

For each of the 94 nuclides, empirically derived *cross sections* have been collected and refined over the years. Determining accurate cross section data is an ongoing activity. "The main limitation in MCNP's ability to model problems correctly is the lack of well known cross sections" (Whalen *et al.*, 1991). The cross sections of a

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

nuclide are stored in a table, indexed by energy, with values (in units called *barns*) for each of the four collision types. For an arbitrary photon energy, the cross sections are to be interpolated. The sum of the four cross section values for a given energy, and the density of the material, allow the *mean free path* length to be calculated; this is the average distance to the next collision. This value can then be used, along with a random number, to select the actual distance to the next collision. The ratios of the four cross section values indicate the relative likelihoods of the four kinds of collision, allowing a random selection of the collision type. Also, in the case of a collision in a composite material, a random selection is made to determine which of the material's nuclides was involved in the collision.

In addition to the cross section tables, each nuclide has two tables of data which relate to interpolations in scatter collisions when selecting a new energy and angle of deflection. Finally, there are four small tables which relate to probabilities of photofluorescence for the heavier nuclides. Since the cross section data for all 94 nuclides is large, MCP-Functional expects to read a cross section file which contains only the data for the nuclides used in the problem being run.

A user describes geometric *cells* for a problem, each with a specified shape and material composition. Every step in a photon's track is a move either to a cell boundary or to a collision location. To select the photon's next move, two distances are computed: the distance to the cell boundary given the photon's current direction of travel, and the distance to the next collision given the cell's material composition and the photon's energy. The smaller of the two selects the one which actually takes place. If it is a collision, the photon is moved along its trajectory by the collision distance, and a new energy and direction are computed. Otherwise, the photon is simply moved to the cell's boundary.

2.2.2 *Variance reduction*

To get better accuracy with less compute time, *variance reduction* is used. Cells are assigned importance values which the user must devise in a somewhat *ad hoc* way. Each time a photon crosses from one cell to another, one of three things happens:

- If the two cell importances are equal, the photon simply crosses over into the new cell.
- If the new importance is higher, the photon *splits* into multiple photons. Each has the same energy, location and direction of travel, but each is seeded with a different random number, and the weight of the old photon is equally apportioned among the new ones.
- If the new importance is lower, then *Russian roulette* takes place: a random choice (weighted by the cell importance ratio) is made to decide whether the photon will continue to be tracked or whether it will 'die'. If it survives, its weight is increased proportionate to the cell importance ratio. Crossing into a zero importance cell simply means that the probability of death by Russian roulette is 100%, i.e. that a photon's track ends when it crosses into a zero importance cell.

It is important to understand that a photon's *weight* is statistical, not physical. The weight tells what fraction of its source the photon represents. All source photons begin with a weight of one. A photon's contributions to tallies are scaled down by its weight. Intuitively, when importance splitting occurs, the apportioning of a photon's weight among its split daughter photons maintains a 'conservation of photons'.

Typically, a user will represent a physical object by multiple cells, rather than one cell, and will assign importances to those cells in such a way as to keep a reasonable number of photons alive as they travel away from the source. The idea is that every source photon should 'push' some photon or photons (which by then have reduced weights) across the tallied surfaces or through the tallied cells. The number of cells, their spacing, and the importances are chosen so that the degree of photon splitting approximately compensates for the photons which die due to absorption or energy cutoff as they migrate toward the tally locations. The exact techniques for choosing importances and cell specifications are learned by experience. A user will experiment with the problem specification, doing small runs and noting the rate of statistical convergence of the tallies. When the user is satisfied with the behaviour of the simulation, a full run will be performed. In general, the cell partitioning and importances should affect the speed of convergence but not the answer itself. However, it is possible for some specifications to result in badly-behaved simulations. The documentation of LANL's MCNP describes the selection of importances as something of an art.

It is possible for the user to set up a geometry such that a cell may be a vacuum of non-zero importance. If a photon enters such a cell and there is no surface to be found on its present trajectory, then it simply travels forever. Our code detects this and aborts with an error message. Such errors can be avoided simply by surrounding the problem geometry by a cell of zero importance.

2.3 User's specification of problem

We now describe how a user sets up and describes a problem to be simulated. This is aided by an example which will be followed through the rest of the paper.

2.3.1 Geometry

A three-dimensional geometry is built up from a number of *cells*, each of a certain material and shape. Our running example's geometry consists of a sheet of carbon steel with an aluminum half sphere shell on top of it, and a gamma radiation source just above its center. A two dimensional slice is shown in figure 2, while figure 3 shows its MCP-Haskell specification file with comments explaining the information in the file. The Haskell-style comments are ignored because the *lex* function in the standard prelude was used in parsing the input.

In MCP-Functional each cell has associated with it the following information:

1. Its region, describing the space the cell occupies.
2. Its material composition.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

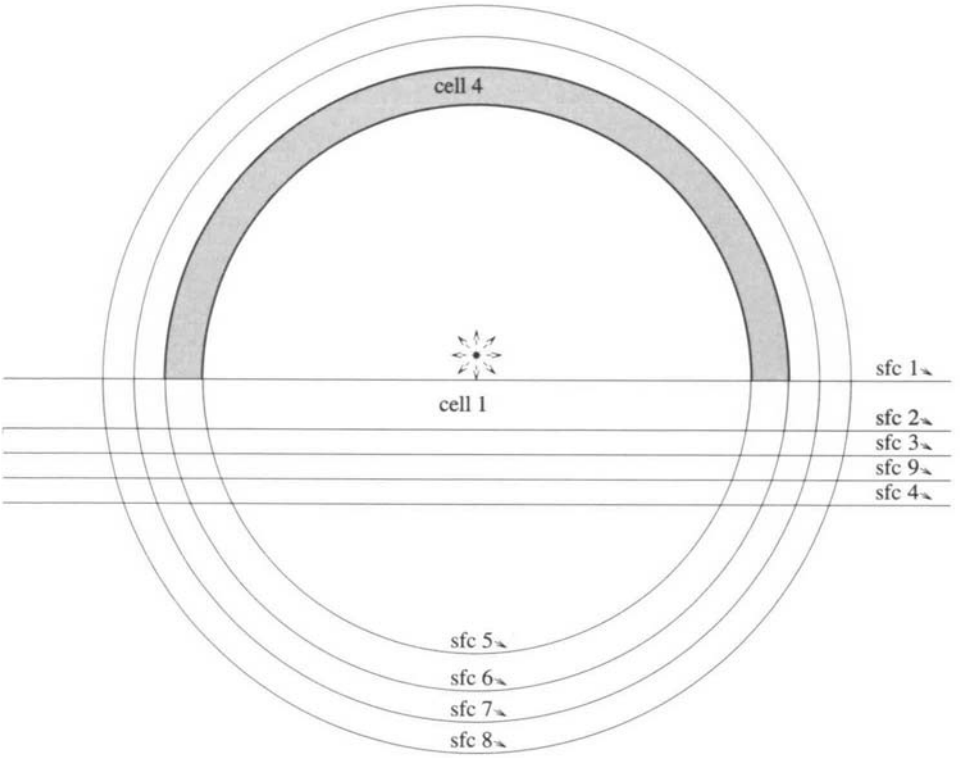


Fig. 2. Vertical slice (at y=0) through the example geometry.

3. Its material density.
4. Its statistical importance (see section 2.2.2).

Regions are defined using *surfaces* such as planes, spheres and cylinders. A surface defines two primitive regions, one positive and one negative, such as the two half spaces defined by a plane or the inside and outside regions of a cylinder or sphere. More complex regions can be defined using *intersection*, *union* and *complement* operators. In defining the regions for the cells, the user has the responsibility to ensure that no point in the three-dimensional space is in more than one cell. Also, it is a run time error for a photon to reach a location which is not in any of the defined cells. If it is impossible for a photon to reach a given point, then that point does not have to be in a cell. However, most users will specify a zero-importance cell whose region contains all points not covered by the other cells.

To describe the material composition of the cells, a separate array of *materials* is defined and referenced by the cells. Each material consists of a list of (*nuclide, fraction*) pairs where the nuclide is in the range 1 (hydrogen) through 94 (plutonium). The user must ensure that the fractions for each material add up to 1. The atom density of the cell's material is expressed in $10^{24} atoms/cm^3$. The statistical importance is a float value; its use is described in section 2.2.2 on variance reduction.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

```
500                     -- number of photons
1.0                     -- simple/detailed threshold
0.005                   -- energy cutoff
0.000001                -- weight cutoff
(0.0, 0.0, 0.01)        -- source location
iso                     -- isotropic source
7                       -- source is in cell 7
-- source energy histogram
[(1.0,    0.00),        -- 1.0    Mev
 (1.0001, 0.4999),      -- 1.0001 Mev
 (1.4999, 0.5001),      -- 1.4999 Mev
 (1.5,    1.00)]        -- 1.5    Mev

-- surface definitions:
[(Pz 0.0),              -- sfc 1 : plane at z =  0.0
 (Pz -1.0),             -- sfc 2 : plane at z = -1.0
 (Pz -1.5),             -- sfc 3 : plane at z = -1.5
 (Pz -2.5),             -- sfc 4 : plane at z = -2.5
 (So 100.0),            -- sfc 5 : sphere (origin), dia = 100.0
 (So 100.5),            -- sfc 6 : sphere (origin), dia = 100.5
 (So 100.7),            -- sfc 7 : sphere (origin), dia = 100.7
 (So 101.0),            -- sfc 8 : sphere (origin), dia = 101.0
 (Pz -2.0)]             -- sfc 9 : plane at z = -2.0

-- cells:
--    geometry        mat'l  importance  density
[*[#-1,  #2]           1       1.0       0.125,        -- cell 1
 *[#-2,  #3]           1       2.0       0.125,        -- cell 2
 *[#-3,  #9]           1       4.0       0.125,        -- cell 3
 *[#5,   #-6,   #1]    2       1.0       0.0892765,    -- cell 4
 *[#6,   #-7,   #1]    2       2.0       0.0892765,    -- cell 5
 *[#7,   #-8,   #1]    2       3.0       0.0892765,    -- cell 6
 *[#-5,  #1]           0       1.0       0.0,          -- cell 7
 +[#-4,  *[#8,  #1]]   0       0.0       0.0,          -- cell 8
 *[#-9,  #4]           1       7.0       0.125 ]       -- cell 9
-- materials:
[[(0.6, 2), (0.4, 1)],          -- matl 1 : 60% iron, 40% carbon
 [(1.0, 3)]]                    -- matl 2 : 100% aluminum
-- z list:
[6, 26, 13]     -- carbon, iron, aluminum
-- cross sections file:
( phot_bench_3 )

-- tallies:
[ collisions   [1, 2, 3, 4, 5, 6, 9]     [1.5],
  flux         [8]                       [0.8,  1.25,  1.5]]

-- random seed:
300506
```

Fig. 3. Example input file

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

In our running example the sheet of carbon steel is defined using boundary planes parallel to the $z = 0$ plane, and it consists of multiple cells so that they can have different statistical importances. Cell 1 is defined as the intersection (*) of the region below surface 1 (#-1) and the region above surface 2 (#2), and has an importance of 1.0. The aluminum half sphere is defined by a number of cells using sphere and plane surfaces. For example, cell 4 is the intersection of the region outside surface 5 (the smallest sphere), the region inside surface 6 (the next smallest sphere), and the region above surface 1 (the plane at $z = 0$). The importances of the cells grow as they are further from the source, to compensate for photons which are absorbed in the closer cells.

2.3.2 Photon source distribution

The photon source is described by three properties:

1. Location.
2. Direction – unidirectional or isotropic.
3. Energy distribution in the form of a cumulative histogram.

An isotropic source emits photons evenly in all directions. In the example we have an isotropic gamma (high energy photon) source just above the center of the sphere, at (0, 0, 0.01). The energy histogram produces half of the source photons with 1.0 Mev and half with 1.5 Mev of energy.

2.3.3 Tallies

The user specifies a list of desired tallies, which describe the histograms which will be gathered during simulation and output at the end of the program execution. Some tally types are inherently related to surface crossings; others are related to collisions in the interior of cells. It can be useful to partition a tally by energy bands, or *bins*. Each tally consists of the following information:

1. Kind of tally (current, flux, collision count, etc.).
2. List of locations (surfaces or cells, depending on the kind of tally).
3. A list of energy bin partition values.

Every time a photon passes through a tallied surface or collides with a nucleus in a tallied cell, it contributes information to the appropriate tally. The nature of the information depends on the kind of tally being performed. For example, a *current* tally simply counts the number of photons crossing a surface, so the tally is incremented by the photon's weight. If the tally was specified with more than one bin, then the appropriate bin is chosen to be incremented based on the photon's energy (i.e. it chooses the first bin whose partition value is $\geq$ the photon's energy). It is an error if a photon's energy exceeds the largest bin, so the user should always make the largest bin value at least equal to the largest possible source energy.

Our example shows two tallies being performed. The first is a collision count (which does not use the photon's weight) in cells 1, 2, 3, 4, 5, 6 and 9. There is

one all-inclusive bin for this tally, set equal to the largest photon energy which can occur. The second tally measures flux (which takes into account photon weight and crossing angle) across surface 8. It is partitioned into three energy bins.

2.3.4 Other information

The remaining pieces of input information are:

1. Simple/detailed threshold (described in Section 2.2.1).
2. The energy threshold below which a photon will no longer be tracked.
3. The weight threshold below which a photon will no longer be tracked.
4. The starting random seed.
5. The name of the file containing the cross section information for the nuclides used in the simulation.

2.4 Output

The output of MCP-Functional reports the average source photon energy, the various kinds of photon *creates* and *losses*, and the values and error estimates for each of the user's tallies. The creates and losses give the user some insight into the kinds of tracks which are taking place, and demonstrate that all photons have been accounted for (i.e. the number of creates equals the number of losses).

Figure 4 shows the output for our running example, with 500 source photons. As expected, the average source energy is almost exactly 1.25 Mev, since about half of the photons should be 1.0 and half 1.5 Mev. The photon creates and losses are equal. (Inequality would indicate a program bug.) The first tally gives the number of collisions per source photon in each of the non-vacuum cells (cells 1, 2, 3, 4, 5, 6 and 9). The second tally gives the photon flux across the outside of the aluminum shell (surface 8), subdivided into three energy bins. For each tally line, the first number is the value and the second number is the error estimate. All tallies are normalized per source photon.

2.5 Comparison of MCP-Functional and MCNP

2.5.1 Capabilities

MCP-Functional implements a subset of MCNP's photon part. Here some of the differences between the programs are noted. LANL's MCNP is an impressive code, with over 400 person-years of effort already devoted to it. It simulates the transport of photons, neutrons and electrons. Its primitive surfaces include cones and toruses in addition to those in MCP-Functional. It also has a great deal of user flexibility, including six photon tally types and 13 kinds of variance reduction. Tally bins can be specified based on a variety of parameters, not just energy. Photon sources can have probability distributions for energy, position and direction as well as energy as a function of angle, etc. Flux can be tallied on point and ring detectors as well as across surfaces. Each tally bin gets ten pass/no-pass tests applied to determine

```
average energy = 1.2489999512730641

photon_creation
    source            500
    cell importance   1518
    p_annihilation    0
    first fluor       62
    second fluor      0
photon loss
    escape            1627
    energy cutoff     0
    weight cutoff     0
    cell importance   166
    capture           287
    pair              0

tally 1:
    [1]  9.3200000000000005e-1  7.7363059728494610e-2
    [2]  5.6800000000000006e-1  1.1467677941655713e-1
    [3]  9.7399999999999998e-1  9.2940754703784670e-2
    [4]  1.5400000000000000e-1  1.5966210737350792e-1
    [5]  9.4000000000000000e-2  1.9335029558804326e-1
    [6]  2.5000000000000000e-1  1.9689591158782346e-1
    [9]  1.1380000000000001  1.0599562042694273e-1
tally 2:
  bin 8.0000000000000004e-1
    [8]  3.2259257224815835e-1  1.2769247563478950e-1
  bin 1.2500000000000000
    [8]  2.0607260608766872e-1  8.6165062944350668e-2
  bin 1.5000000000000000
    [8]  2.1157410637032456e-1  8.4882237007883735e-2
```

Fig. 4. Example output file

whether the simulation is well-behaved. The hundreds of pages in MCNP's users manual attest to its capabilities.

In creating MCP-Functional it would have taken a prohibitive amount of time to implement all of MCNP's photon capabilities. Instead we have tried to choose representatives of the things which MCNP can do, so that the code's structure and behavior faithfully reflect the job being performed.

2.5.2 Use

An MCNP user tends to handle a problem in four steps:

1. Construct the problem by specifying the geometry, source, tallies, etc. Also specify the variance reduction techniques and parameters, including subdividing cells and assigning importances to them.
2. Do a small run, yielding information about the convergence and general behaviour of the simulation.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

3. Use this information to modify the problem specification, especially the variance reduction parameters.
4. After iterating the previous two steps and obtaining satisfactory behaviour, execute one or more long runs. Such runs may involve millions of source particles.

MCP-Functional's feedback to the user during the middle steps of this process is probably inadequate for real-world use. MCNP produces a large amount of information which helps give the user a picture of what is happening during the simulation. For example, since cell importances are manipulated by the user to keep a reasonable population of photons alive as they migrate toward the tally locations, a user is interested in the number of photons entering and leaving each cell or crossing each surface. That information can be used to change the importances appropriately.

3 MCP-Functional – Program structure and correctness

On the top level, the code consists of input parsing, simulation and output formatting. The simulation part is divided into tracking and tallying, which have a producer–consumer relationship for which the *event list* serves as the interface. The event list provides a clean separation between the tracking and tallying tasks, which means that new kinds of tallies can be added without touching the tracking code in any way. As a photon is tracked, each collision and surface crossing appends an event onto that photon's event list. The event contains all useful information about the photon at that time: energy, weight, location, direction, angle of crossing, etc. The event list is traversed by the tallying code to extract the required information for output.

Throughout this section, MCP-Haskell is used when code fragments are shown; their MCP-Id counterparts are nearly identical except for syntax differences. Figure 5 shows the function *mcp* which generates the source photons, tracks them, tallies the results, and produces the output string; figure 6 shows a flow diagram of that code. A list of random seeds is generated, and the *sample_source* function is mapped on the list, yielding a list of source photons and their seeds. This list is then used to determine the average source energy, and the *track* function is mapped on it, yielding a list of event lists. The user's tally functions are then applied to each of the event lists, yielding a list of tallies for each source photon. Finally, that list of lists is transposed and the accumulations are performed. Also, the list of event lists is used to total up the various photon creates and losses.

3.1 Tracking

The tracking code has two parts: physics and geometry. The physics part of the tracking code bears strong resemblance to the corresponding Fortran code in MCNP after the 'spaghetti' is untangled. Some of the Fortran functions have exact counterparts in MCP-Functional. The routines include interpolations of cross sections, and sampling of statistical distributions through techniques such as direct methods or rejection sampling. (Chapter 2 of Carter and Cashwell (1975) contains a full discussion of statistical sampling.)

```
mcp :: User_spec_info → Srce_spec_info →
                    [Tally_entry] → Seed → Int → String
mcp user_info srce_info all_tallies seed n =
    let
        recip = 1.0/(fromIntegral n)

        particle_list :: [(Particle, Seed)]
        particle_list =
            map (sample_source srce_info) (take n (iterate trand_9973 seed))

        avg_e =
            (foldl (+) 0.0 (map (\((_,_,_,e,_),_)→e) particle_list))*recip

        event_lists :: [[Event]]
        event_lists = map f particle_list
            where
            f (particle@(_,_,_, e, _), sd) =
                (Create_source e):(track user_info particle [] sd)

        tally_list :: [[Array (Int,Int) Double]]
        tally_list = transpose (map g event_lists)
            where
            g evs = map (tally_a_source evs) all_tallies

        accums :: [Array (Int,Int) Double]
        accums = map accum_tally tally_list

        squares :: [Array (Int,Int) Double]
        squares = map accum_tally_squares tally_list

        totals :: [Int]
        totals = tally_bal event_lists

        final_tallies = zip5 all_tallies accums squares (repeat recip) [1..]

    in
    shows_results avg_e totals final_tallies ""
```

Fig. 5. Top level Haskell code

The geometry part of MCP-Functional was developed independently from the original Fortran code of MCNP, as it was much easier to start from scratch and use the more powerful data structuring facilities of functional languages. Recursive data structures are used to express in a direct way the region expressions specified by the user. The *region* is defined in the Haskell code as follows:

```
data Region =
        R_prim Int              -- base case: primitive region
                                -- the absolute value of the Int specifies
                                -- the surface, and its sign indicates
```

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

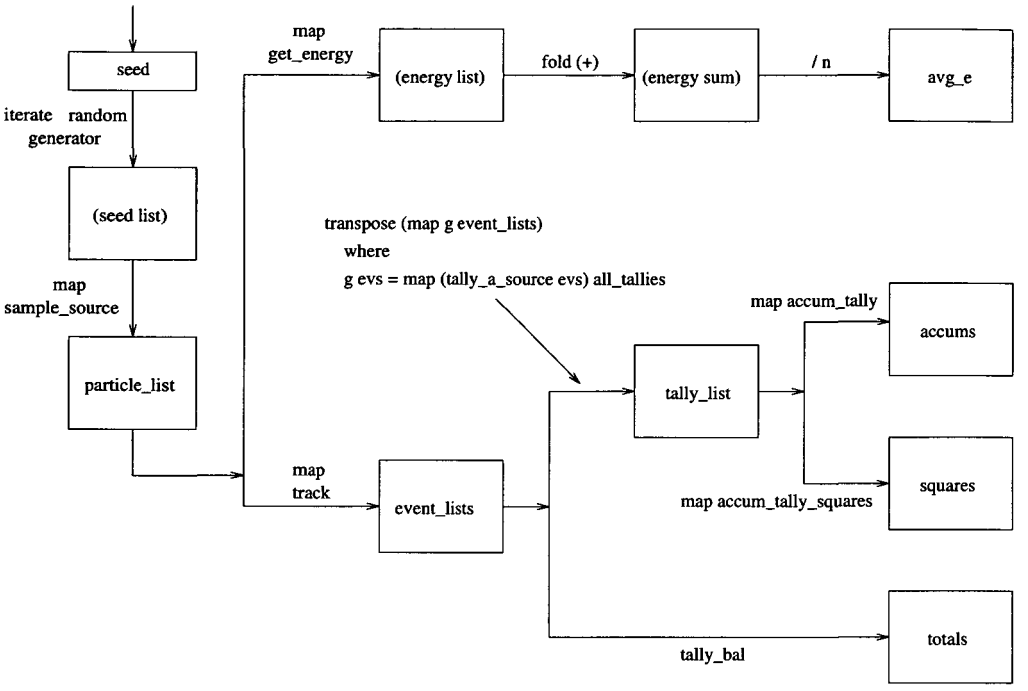


Fig. 6. Dataflow view of MCP

```
                              -  which side of the surface
| R_compl Region              -  complement of a region
| R_intersects [Region]       -  intersection of regions
| R_unions [Region]           -  union of regions
```

The geometry part has two responsibilities. The first is to determine the distance to a cell's boundary, given a photon's location and direction. This is done by calculating the distance to each of the surfaces which form the leaves of the cell's region structure, eliminating those which are cutting through a cell (rather than forming part of its boundary on the half line defining the trajectory of the photon), and selecting the minimum. When calculating the distance to a surface, the direction of the photon is taken into account; surfaces that will never be crossed, because they are 'behind' the photon or are parallel to its trajectory, have an infinite distance. The second responsibility of the geometry part of the code is to determine, after a surface has been crossed, which cell has been entered. This involves testing, for each cell, whether the new location lies within that cell's boundaries. Search lists, one for each surface, are precomputed at the start of the program and devised so that the more likely cells are at the head of the list and are therefore tested first.

The *track* function uses the physics and geometry routines to track a photon. It begins by determining whether energy or weight cutoff terminates the photon's track. If termination does not occur, it determines whether the next event will be a collision or a move to the cell boundary (as described in section 2.2.1). One of three

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

functions is then called: *simple_physics*, *detailed_physics* or *move_to_surface*. These functions are capable of terminating the photon's track (by absorption or Russian roulette), continuing the track, or producing multiple photons (by pair production, photofluorescence, or importance splitting), each of which must be tracked.

The *track* function and these three other functions are mutually recursive. Each call to track handles one collision or one move to a cell boundary, and appends one event to the accumulating event list. If the photon survives the collision or cell crossing, the *track* function is called with the photon's updated information to continue its tracking. In the cases where splitting occurs, *track* is called on each of the split photons and the returned event lists must be concatenated. To avoid the inefficiency of actual list concatenation, an accumulating event list parameter is used.

3.2 Tallying

The tally code takes the event lists produced by the *track* function, extracts each tally, and reduces tally information to averages and standard deviations so that error estimates can be given. When the user's input specification file is read, a list of tallies is created; each tally specification is held in a data structure defined as follows:

```
data Tally_entry =
        Tally
            Tally_function          -  specifies current, flux, etc
            [Int]                   -  list of surfaces/cells to tally on
            (Array Int Int)         -  table mapping sfc/cell nums to array index
            Int                     -  number of surfaces/cells being tallied
            (Array Int Double)      -  energy bins
```

The accumulation of an event list, using a user's tally, is performed by the following function, which takes one event list and one tally, and returns a tally array:

```
tally_a_source :: [Event] → Tally_entry → (Array (Int,Int) Double)
tally_a_source evs (Tally f _ ref n bins) =
    accumArray (+) 0.0 ((1,1),(n,ss))
                [(i,bin_idx):=v | (i, bin_idx, v) ← (map g evs), v>0.0]
    where
    (_,ss) = bounds bins
    g :: Event → (Int, Int, Double)
    g ev =
        let
            (idx, bin_val, val) = f ev
            j = ref!idx
            bin_num =
                let
                    bin_num' =
```

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

```
                    1 + length (takeWhile ( \v→bin_val>v) (elems bins))
            in
            if bin_num' > ss then
                error "no bin for value"
            else
                bin_num'
    in
    if (not (inRange (bounds ref) idx)) || (j == 0) then
        (0, 0, 0.0)
    else
        (j, bin_num, val)
```

Figure 7 shows four of the intermediate data structures represented in figure 6. The first is *particle_list*, which is the list of source photons and their seeds. The second is *event_lists*, which is the result of mapping the *track* function on *particle_list*. The third and fourth are the list of tally arrays, before and after transposition. The transposition is done to prepare for the horizontal accumulations which will be performed to produce the *accums* and *squares* tally results.

In addition to the user's tallies, the program also reports a tally of all photon creates and losses, called *totals* in figure 6, by accumulating into a histogram indexed by the various kinds of births and deaths.

3.3 Assessing correctness

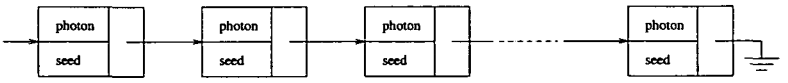
There are two possible approaches to checking the correctness of a Monte Carlo simulation code. The first is to devise simple tests for which an analytical solution is known. If the simulator's result matches the analytical solution within the error estimate, then the simulator can be considered correct for that particular test case. The second approach tests not only that the code faithfully reflects the underlying theory, but tests the theory itself by simulating real world problems and comparing the results with measured physical experiments.

3.3.1 Monte Carlo outputs

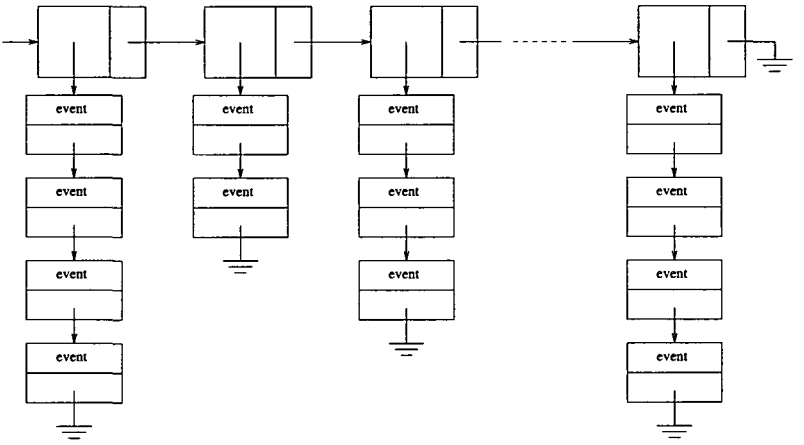
Testing and debugging Monte Carlo codes can be difficult, due to the statistical nature of their outputs. The source of the difficulty comes from the fact that these codes produce not just values, but values with error estimates.

Consider as an example the development of some simple Monte Carlo code with an output of a single scalar value. The code is written with various low level components that are 'glued' together to form the program. The low level routines are individually tested as completely as is practical. Then the program must be tested by running a problem which has a known result, either because it has an analytical solution or because empirical real-world measurements have been made.

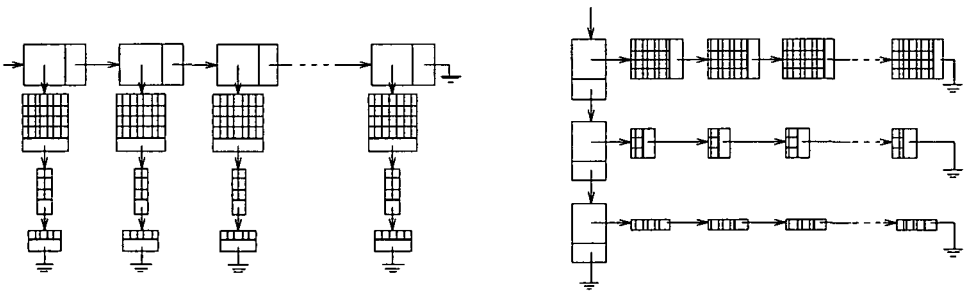
https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press



(a) Source photon list, called *particle_list*



(b) List of event lists, called *event_lists*



(c) List of tally-array lists (d) Transposed for accumulating, called *tally_list*

Fig. 7. Main data structures in MCP

When the program is run, a value and error estimate are produced. If the known correct value is outside of the error range, the probability is high that there is a problem in the simulation. (The problem could be a programming error or an error in the simulation model itself.) But if the correct value lies within the error range, it *cannot* be concluded that the simulation is working correctly. In other words, an error range that contains the correct value is a necessary but not sufficient condition for correctness.

Figure 8 shows how this can happen. Each point represents the output value for a given run, and its vertical bar represents its error range (the value plus or minus the error estimate.) The known correct result is also shown. In both of these scenarios

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

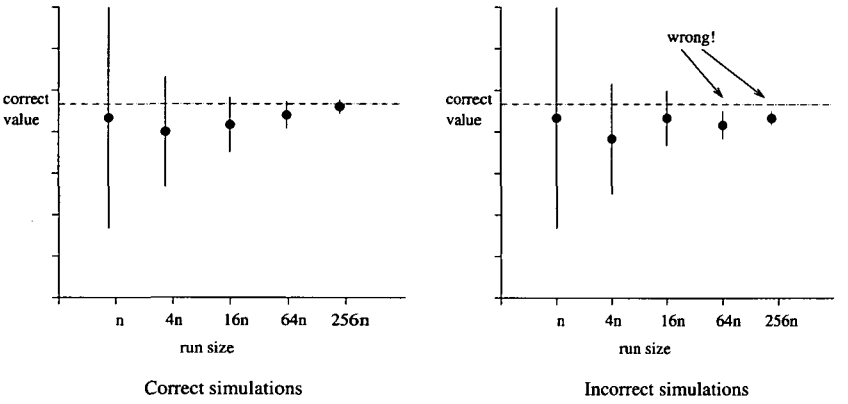


Fig. 8. Two hypothetical convergences

the same error range is produced for a run of size n, but it is impossible after that size run to know whether either simulation is working correctly. Quadrupling a run size cuts its error range in half. As larger runs are performed the first scenario shows convergence on the correct known value, because that value is always within the error bars. In the second scenario a problem is not apparent until a run size of $64n$, where the error bar's range does not contain the correct value. Thus determination of correctness for a given problem can come only from doing large runs, so that sufficiently small error bars can reveal the convergence. Note also that the convergence of the values is not necessarily monotonic; however, each error bar will likely be within the range of the preceding error bar.

If a low level routine is itself statistical in nature, then the same problem exists not only in testing the program as a whole but in testing the individual routine during the early stages of program development. Once a bug is known to exist, finding its source can also be difficult since its manifestation is statistical. Dumping traces of the simulation is often of little use since the individual steps of the simulation can look feasible yet the aggregate of the steps is known to be wrong.

3.3.2 Determining correctness of MCP-Functional

An important activity of the Radiation Transport Group (X6) which maintains the MCNP code at LANL involves validation of MCNP using both analytically solvable problems and empirical measurements. For MCP-Functional, validation is considerably easier: we assume that MCNP is correct and simply seek to duplicate its results. However, comparing for exactly identical outputs turns out to be impractical, since it would require both that the random number streams be identical and that each value from the stream be used at the same decision point in both executions.

In both the Fortran and the functional codes the random stream is a sequence of unsigned integers; each is used to generate the next one in the sequence, and is also converted to a float in the range zero to one to be used at a decision point. MCNP's designers have gone to some trouble to use 48-bit seeds (with 96-bit intermediate results in multiplies) while maintaining machine portability (Hendricks,

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

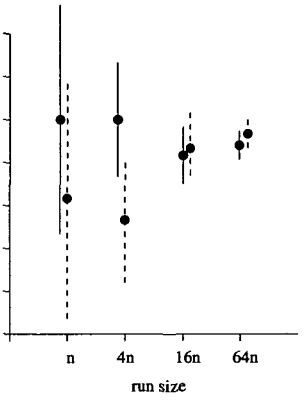


Fig. 9. Hypothetical comparison of two Monte Carlo programs

1991). But using 48-bit seeds in the functional versions would have been awkward since integers on the Monsoons and SPARCs are 32 bits. Even if MCNP's random stream had been duplicated, it would have been challenging and confining to force MCP-Functional to use the stream in the same order as it is used by MCNP. Instead, MCP-Functional uses 30-bit integers in its stream, with 60-bit intermediate results in multiplies.

Because MCNP and MCP-Functional generate and use the random streams in different ways, the comparison of their results must be statistical, and since MCNP gives error estimates, our comparisons must be between MCNP's error bars and MCP-Functional's error bars. Figure 9 shows a hypothetical comparison of the outputs of two Monte Carlo programs for various run sizes. Here we consider the two programs to be in agreement if their error bars always show some amount of overlap. Note that the amount of overlap often can change with different run sizes because of variations in the random streams and their uses.

Because of resource management problems neither MCP-Id nor MCP-Haskell are able to run problem sizes that allow meaningful comparisons of outputs with MCNP (see section 6), so it was necessary first to develop a code in which resource usage was adequately constrained, thus allowing detection of bugs during development and validation of the final program. Because Id allows explicit synchronization and memory releases, and because the Id code preceded the Haskell code in development, an 'impure' version of MCP-Id was created that imposed no limit on the number of photons that could be run, allowing meaningful statistical comparisons of their outputs.

Using this impure code, various problems have been simulated and all have given statistically comparable results to those of MCNP. The problems include three photon benchmarks, with analytical solutions, produced by LANL's X6 group and used by them to validate MCNP. Our running example also has been run on both the modified MCP-Id and MCNP. Figure 10 shows, in graphical form, the outputs of both programs on 160,000 source photons. A horizontal bar's width indicates the range of the energy bin, its y-coordinate is the tally value, and the vertical bar shows the error range. Note that the sizes of the error ranges agree, and that the error

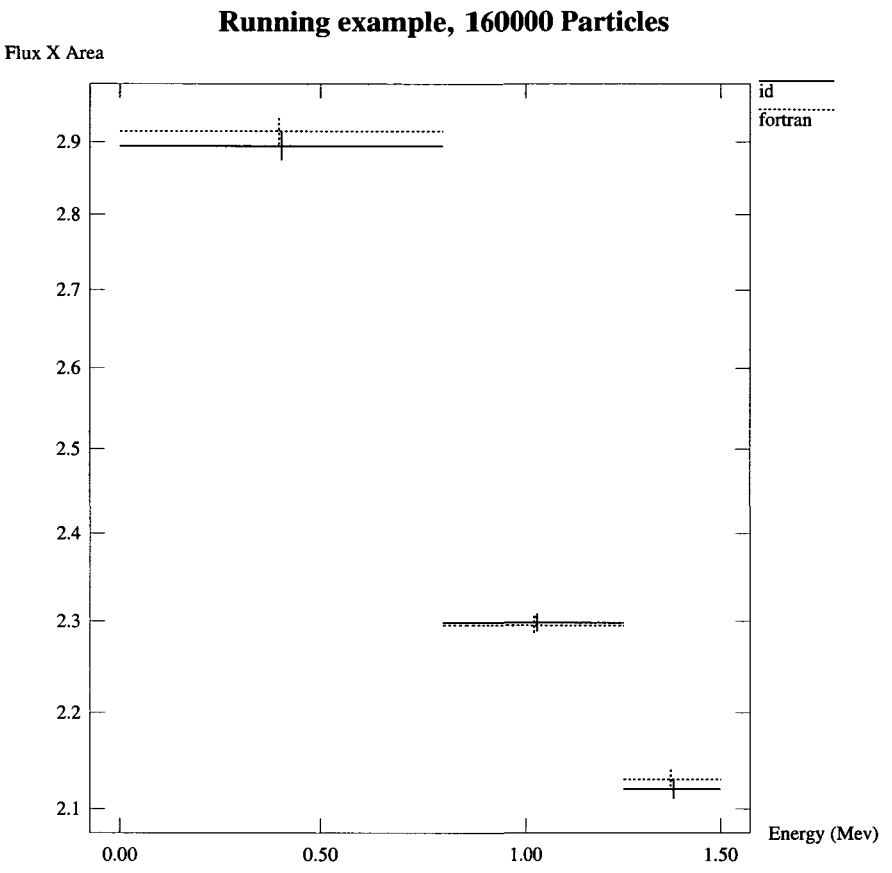


Fig. 10. Comparison of MCNP and MCP-Id output

ranges on the tallies overlap. The extremely close agreement of the middle energy bin is coincidence; smaller runs show less overlap in this band.

4 Programming style

The Fortran MCNP code is quite large, consisting of about 40,000 lines of code, including comments, and counting the replicated COMMON blocks just once. It is coded to be as compact as possible; local variable names are all no more than two characters long. It reflects a coding style which "...is clearly counter to modern computer science programming philosophies..." (Briesmeister, 1993). Except in some cases where bottom-level physics subroutines could be isolated and understood, it was hopeless to use the Fortran code in any significant way when writing MCP-Functional because of its side-effecting and the differences in overall code structure.

4.1 Development approach

Rather than attempt to use the Fortran code as a point of departure, we started with a fresh, structured view of the problem. Strong parsing capabilities coupled with

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

recursive data structures allow the geometry code to be expressed in a straightforward way. The event list in MCP-Functional serves as an interface between the tracking and tallying parts of the program, whereas in MCNP the geometry and collision parts of the code call routines which side-effect into the tally arrays.

One of the most important differences between the Fortran and the functional codes occurs in the handling of random numbers. Since functions in a pure functional program have no state, the random seeds must propagate through the code as function parameters. At split points, a seed must be able to 'fork', i.e. one seed must be able to generate two or more seeds with no correlations among them. The easiest solution to this forking was found in the MCNP code itself. A linear stream of seeds is used, and the stream has the characteristic that it is possible, in constant time, to leap forward in the sequence by any desired number of steps. Each source photon is given a fixed segment of the stream, and when the track of a photon splits, the split photons are seeded by taking different sub-segments of that photon's segment.

This *strided* approach to random number generation is not ideal. It is possible for long photon tracks to cross into other photons' segments, making it theoretically possible for correlations to occur. It is also possible for the entire random stream to loop around if a large number of source photons is run. However, experiments performed by X6 show that the effect on the answers is usually negligible, probably due to the fact that even when a part of the random stream is being reused, each value's decision point is completely different (Hendricks, 1991).

As noted in section 3.3.1, debugging Monte Carlo codes requires reasonably large problems to be run. The development process was therefore seriously altered: instead of developing a pure functional code first, and then making necessary adjustments to allow it to run large problems under hardware resource limits, an impure version had to be developed and used throughout the development process. Only when that version was fully debugged could the pure Id and Haskell codes be derived from it. Since Id was used for the impure version, the development sequence was:

1. Impure Id.
2. Pure Id (MCP-Id).
3. Pure Haskell (MCP-Haskell).

Once a correct impure Id code was in hand, the pure versions came easily since the codes are deterministic and are expected to produce identical results.

4.2 Id's non-functional part

It is useful to note some relevant aspects of Id's non-functional language features. The language is built around a pure, higher order functional core which is augmented with side effecting capabilities. Its semantics are parallel, and the exact execution order of concurrent operations can vary at run-time. The programmer is able to exercise some control over execution order by the use of loop bounds and barriers. Each loop can be given a bound that specifies how many concurrent iterations are allowed to take place. Barriers create synchronization points; within a code block, concurrent operations preceding a barrier are guaranteed to complete before any

operations after the barrier can start. (Note that the barrier syntax is three dashes; Haskell programmers should not confuse this with Haskell's comment syntax.)

Id allows the expression of explicit reads and writes with two kinds of data access behavior: I- and M-structures. Both kinds are synchronized, on a word by word basis, for readers; if a reader attempts to read an empty memory location it will block until some writer fills it. I-structure memory imposes a 'single-assignment' restriction in its use; a second write to a location will cause a run time error. An M-structure memory cell is emptied by a read, making it possible for another writer to fill the cell with new data, and making it possible for programs using M-structures to produce non-deterministic results.

Heap memory is implicitly allocated in Id whenever a data structure is created, but the Monsoon run-time system has no garbage collection. Explicit releases are required in the source code to return a structure to the heap. The release is placed in a block of concurrent operations or statements, and it is guaranteed to take place only after all operations in the block have completed. It is the programmer's responsibility to place a release in such a way that it can occur only after all references to its structure are complete.

Each function call during execution of an Id program on Monsoon requires frame memory. It remains allocated for the lifetime of the function call and is released when the function returns its value and all side-effecting statements within the function have completed. Because Monsoon has limited frame memory, it cannot tolerate deep recursion, so many of Id's library functions have been written in loop rather than recursive form. For example, the well-known *map* function loops through its source list, creating its return list as it does so. Since the return list has to be built from head to tail, the iterative form uses side-effecting: each iteration creates a cons cell with an unwritten I-structure tail pointer, and the succeeding iteration writes the tail.

It is common practice for functional programmers to use an 'accumulating parameter' to avoid list concatenation (Bird and Wadler, 1988). This technique does not work, however, if the recursion is replaced by iteration. Because iteration is so often necessary to make Id programs execute within Monsoon's resource limits, the Id library defines *open lists*, which allow list concatenation to take place in constant time. Similar to the map implementation, an open list maintains a tail whose tail pointer is an unwritten I-structure. The list itself is represented as a pair of pointers, one to the head and one to the tail cell. Concatenation of two lists is done by writing the tail of the first list with a pointer to the head of the second list. A special *close* operation writes a null pointer to the tail of an open list, turning it into a normal list. Because they work through side-effecting, open lists must be used with care. Specifically, great care must be taken if an open list is shared, since concatenate and close operations will affect all references to the list.

4.3 Comparing Id and Haskell

Id and Haskell have strong similarities: they are both functional or have a functional subset, with currying and higher order functions. In creating MCP-Id and MCP-

Haskell, we have attempted both to use the languages' features and to reflect the way experienced programmers write in each of the languages. Here we discuss the style differences between the two languages and the reasons for those differences.

Coding styles are influenced not only by language characteristics, but also by the purposes for the codes and the limitations of the machines which will execute them. Id programmers tend to try to balance exploitation of parallelism and efficient space usage, while Haskell programmers are further removed from machine issues and rely on the compiler to do an effective mapping of the program onto the target machine. Because the Id language has been closely associated with the Monsoon dataflow machine, that machine's characteristics (especially its resource limitations) often influence the way Id codes are written.

Id's requirement for explicit heap memory releases can affect coding style in a number of ways:

- In Id, function composition style is discouraged because the intermediate data structures which glue the functions together must be released. Releasing a heap structure in Id requires the structure to be named, so an expression which composes functions must be broken up, with each function application bound to a name. Instead of composing functions, Id programmers tend to write in a 'deforested' style, fusing the functions into one function or loop and eliminating the intermediate structures entirely.
  In contrast, the combination of higher-order functions, partial applications, laziness, and an extensive library of functions in Haskell encourage a compositional style of programming. Laziness in the compositional 'pipeline' insures that only the necessary parts of the intermediate data structures will actually be created, and automatic garbage collection (and possibly a deforestation optimization in the compiler to eliminate intermediate data structures at compile time) can make Haskell programmers less concerned about the costs of this style. Also, they are more insulated from the target machine details, and sequential semantics means that they need not concern themselves with control of parallelism.
- Though the Id language views functions as curried, MIT's compiler optimizes those situations where a function application is given all of its arguments at once. Instead of creating closures (which consume heap memory), it uses a special, more efficient call mechanism which uses no heap. Thus, Id programmers tend to avoid partial applications because they require explicit releasing of closures if heap memory is to be reclaimed. Since Haskell programmers do not directly control reclamation of heap memory (and in fact do not even know the implementation details used by the compiler and run-time system), the costs of partial applications appear to be, and may be, no different than the costs of other functions.
- In Id, arrays are often preferred over lists since an array of scalars can be released in one operation, whereas releasing a list requires traversing the list, releasing the cons cells one by one.

The cumulative effect of all these concerns is that experienced Id programmers

often avoid recursion, higher-order functions, partial applications and function composition, because even though the language itself allows all of them, efficiency and resource issues encourage the use of a more imperative-looking (though well-structured and relatively side-effect free) style with abundant use of loops.

The MCP-Id code shows evidence of these influences. They are more pronounced in the simulation part of the program, since that is where the vast majority of execution time is spent. More liberal use of higher-order, composed functions occurs in the code which reads the input and sets up the simulation since efficiency effects there are negligible. The code is also influenced by the fact that the pure functional MCP-Id version came after a less pure version, and MCP-Haskell was developed after the pure Id version had been written and debugged.

5 Comparison of the codes

The MCP-Haskell and MCP-Id codes are, in the main, structured identically; the differences between them tend to be local. The following discussion highlights the major contrasts between the codes.

5.1 Input/Output

The two languages differ significantly in their handling of input/output. Each code's I/O routines were written in a way that extended the I/O capabilities already found in the language.

The Id library file *stdio.id*, which comes with the Id-world software, holds a large number of I/O functions which work in an imperative way. A file is opened for reading or writing, and a stream pointer is returned to be used as an argument to the many input and output functions which exist for various data types. Since these functions side-effect directly to the file, explicit sequencing (using the *seq* construct or explicit barriers) must be enforced by the programmer to avoid interleaving accesses. Higher order functions are used to allow the building up of I/O functions for more complex data structures. For example, the *format_list* function defined in the library takes as its first argument the format function which is appropriate for the kind of element in the list.

Writing input and output routines for MCP-Id consisted of extending the scan and format functions with routines to recognize the various structure types defined in the program, and then using those routines as parameters in the existing scan and format functions. For example, a new function called *scan~cell* was created and used in reading the array of cells:

```
typeof scan~cell = IO_STREAM → CELL_DEF;
def scan~cell strm =
        { reg = scan~region strm
          - - -
          mtl = scan~int strm
          - - -
```

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

```
        imprt = scan~float strm
        - - -
        density = scan~float strm
        - - -
        in reg, (flatten reg), mtl, imprt, density };

.
.
cells = scan_1d_array scan~cell strm
.
.
```

In the definition of *scan~cell*, barriers force sequential access to the file being read. In obtaining each of the necessary values, the appropriate scan functions must be used. Later, *scan~cell* is used as an argument to the higher order function *scan_1d_array* to read an array of cells.

Haskell's I/O is much different, since an entire file is handled as one string. The Standard Prelude provides various functions for extracting information from strings and building them, and Haskell's overloaded *reads* and *shows* functions are very powerful. For MCP-Haskell the input functions are written in the style described in Section 5.3.1 of Hudak and Fasel (1992). MCP-Haskell's counterpart to MCP-Id's *scan~cell* is shown here:

```
readsCell :: String → [(Cell_def, String)]
readsCell s =
    [ (Cell reg (flatten reg) mat imp density, s') |
          (reg, x)          ←   reads s,
          (mat, y)          ←   reads x,
          (imp, z)          ←   reads y,
          (density, s')     ←   reads z]

instance Text Cell_def where
    readsPrec _ = readsCell      - make "readsCell" a method for "reads"

.
.
(cells, s') ← reads s
.
.
```

Haskell's approach to I/O needs no barriers since its *reads* and *shows* functions are not side-effecting. Also, because of overloading, the programmer does not have to specify which kinds of read are performed in the body of *readsCell*; the type system will infer the correct method in each call of *reads*. This is true not only for the

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

material, importance and density parts (which are scalars), but also for the function which reads a region because *Region*, in earlier code, was made an instance of class Text, and a method for reading regions was created. Similarly, in the code above, *Cell_def* is made an instance of Text and *readsCell* becomes its method for *reads*.

Three type synonyms in MCP-Id (including the CELL_DEF shown above) were made data types in MCP-Haskell with the specific purpose of making it possible to read them with the overloaded *reads* function. This made it necessary to create a constructor function for each. The following example shows the declaration of CELL_DEF in each code:

Id:

```
typesyn CELL_DEF =
    REGION, (list I), I, F, F;
```

Haskell:

```
data Cell_def =
    Cell Region [Int] Int Double Double
```

Haskell also allows a programmer to get a text representation of a data type through the use of a *derived instance*, where a type's constructor names in the program source become the constructors' text representations for input and output. While it would have been possible to use derived instances for all data types in MCP-Haskell, we specified our own syntax (closer to that of MCNP) for all types except *Surface*.

5.2 *Laziness*

Since MCP-Haskell was derived from MCP-Id, it is not surprising that laziness is not used in any deep, fundamental way. Nevertheless, laziness is used in places to simplify code or make it clearer.

Laziness sometimes simplifies conditional expressions. This example computes the distance from a point to a plane which is perpendicular to the x axis, given the direction cosine u to the x axis. It comes from the geometry part of the code where the distances to a cell's boundaries are computed. Two special cases must be tested: trajectory parallel to the plane ($u == 0$), and the plane 'behind' the photon (distance $t < 0$). In the Id code the computation of t must be protected by the outer conditional to avoid a division by zero. The Haskell code requires only a single conditional expression since the value of t will only be demanded if the left side of the || operator evaluates to false.

Id:

```
def dist_to_sfc (PX d) =
    if u == 0.0 then maxfloat
    else
        { t = (d-x)/u
```

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

```
            in
            if t <= 0.0 then maxfloat
            else t }
```

Haskell:

```
dist_to_sfc (Px d) =
    let
        t = (d-x)/u
    in
    if (u == 0.0) || (t <= 0.0) then
        bignum
    else
        t
```

A much more significant situation involving nested conditionals, too large to incorporate in this paper, occurs in the function which handles photofluorescence in *track*. A sequence of tests is made to determine whether zero, one or two photons will be produced when a photon is absorbed. In the Id code, various computations occur in code blocks at different levels of the nested conditionals. In the Haskell code these computations were lifted to the function's top level, eliminating many 'let' blocks and allowing laziness (and particularly the nonstrictness of the second argument of || and &&) to protect against such things as unneeded computations or out-of-bounds array accesses. The result is code which is much easier to read.

Laziness also played a role in the generation of the source photon seeds. The function *trand_9973* takes a seed and returns a new seed by striding forward in the initial random stream by 9973 steps. In MCP-Haskell the list of n seeds, where n is the number of photons to be tracked and *seed* is the initial seed specified by the user, is easy to generate:

```
take n (iterate trand_9973 seed)
```

The *iterate* function has no terminating condition; laziness decouples the problem of generating the seeds from the problem of terminating the list.

Id's library also has an *iterate* function, but it has a third parameter to allow it to terminate the list. Unfortunately there are a variety of ways one might wish to terminate an iterate function. Id's library function *iterate p f x* terminates by applying a predicate to the list members as they are generated; it is the equivalent of the Haskell composition *takeWhile p (iterate f x)*, and it is poorly suited to terminating on the basis of list length. Because of this a special iterate function was written for MCP-Id.

5.3 Function composition and loops

Both MCP-Id and MCP-Haskell made little use of explicit recursion. Instead, MCP-Id used loops, while MCP-Haskell used higher-order functions. A simple example

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

illustrates this:

Id:

```
typeof sample_for_nuclide = (1d_array (F,F,F,F)) → F → I;
def sample_for_nuclide xsecs r =
    {
    i,h = bounds xsecs;
    tot = 0.0
    in
    {while (r > tot) and (i <= h) do
        next i = i+1;
        tphoto, tinc, tcoh, tpair = xsecs[i];
        next tot = tot + tphoto + tinc + tcoh + tpair
        finally i-1}};
```

Haskell:

```
sample_for_nuclide ::
    Double → [(Double, Double, Double, Double)] → Int
sample_for_nuclide r =
    length.(takeWhile (r>)).sums.(map (\(a,b,c,d)→a+b+c+d))
```

Here the cross sections (an array in Id, a list in Haskell) are summed until they cross over the value r. The crossover index selects which nuclide of a composite material is involved in a collision. Note also the use of laziness: the *map* and *sums* functions will stop when the *takeWhile* function is satisfied.

6 Resource management

Resource management problems severely limit the number of source photons which both MCP-Id and MCP-Haskell can run. The consequences are not limited to the usefulness of the codes themselves. As section 3.3.1 points out, large runs are needed during development to detect the presence of bugs. To run large numbers of photons, a separate, impure version of MCP-Id was created first.

6.1 Heap managed Id code

There are four goals in this version of the code:

1. Release all structures to the heap. This must be 100% effective, since even one space leak will eventually use up the heap if many millions of source photons are run.
2. Control parallelism so as to stay within frame memory limits.
3. Force an evaluation order such that the peak requirements for heap and frame memory are constant with respect to the number of source photons.
4. Remove intermediate data structures where possible.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

The most important aspect of this version is a top-level restructuring to more tightly couple producers and consumers. While I-structures (either explicit or implicit) can prevent a consumer from running ahead of its producer, there is nothing to prevent the producer from running far ahead of a consumer and prematurely creating structures which sit waiting for the consumer to catch up. To prevent this, the entire flow diagram of figure 6 was combined into one *for* loop body in which each iteration creates, tracks, tallies, and frees the structures for one source photon:

```
{for i ← 1 to nparticles bound bnd do
    next seed = trand_9973 seed;
    particle, s0 = sample_source srce_info seed;
    _, _, _, e, _ = particle;
    (
    next energy_acc = energy_acc + e
    - - - % make sure 'track' doesn't free particle before 'e' is read
    o_events =
        ol_cons_r (CREATE_SOURCE_EV e) (track user_info particle s0);
    );
    events = ol_close_r o_events;

    % tally the aggregates
    _ = tally_balance events;

    % do the user's tallies
    _ = {for j < - low to high sequential do
        _ = tally_a_source all_tallies[j]
                    events accumulators[j] squares[j];}

    - - - % complete tallying before releasing the event list

    _ = free_struct_list events};
```

This allowed elimination of some intermediate lists, but more importantly, it made the heap requirements constant with regard to the number of source photons: a new iteration (source photon) cannot start until a previous one has completed and returned its memory to the heap. Because the tally accumulations had to become a part of the loop, they could no longer be expressed using Id's accumulator array syntax. Instead, the tallies were explicitly accumulated into M-structure arrays by the *tally_a_source* and *tally_balance* procedures.

To conserve frame memory, sequential processing of each source photon was enforced by using sequential loops and/or barriers in the *track* function so that split photons are not executed concurrently. Thus, parallelism comes only from the bound on the *for* loop. Also, recursions were replaced with loops wherever possible. This included putting a loop in the *track* function to process linear photon tracks without

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

recursion. When splitting occurs, all but one of the split photons are tracked by recursive calls, and the last of the split photons continues to be tracked by the loop. Since this loop form is incompatible with the use of an accumulating parameter for the event list, Id's open list library functions were used to allow appending and concatenation to be done in constant time. This is safe because there is no sharing of these lists in this code.

All heap memory space leaks were plugged by adding releases. This included making a modified version of the open list library functions with releases added. Also, synchronization barriers had to be strategically placed to ensure that the releases did not occur prematurely. For example, consider the situation where a structure is allocated, its fields are written, the consumer reads some (but not all) of the fields, and the structure is released. Without barriers it is possible for the structure to be released and subsequently reallocated before all of the fields have been written, causing I-structure double-write errors to occur. Placing a barrier immediately after the writing of the fields, thereby witholding the structure from readers until it is completely written, solves the problem. Sharing of data structures is eliminated by copying, so that reference counts are not needed and a reader can release a structure when finished with it.

It would be hard to overstate the difficulty involved in producing and working with this version, because the debugging process involved many iterations of plugging memory leaks, running problems, tracking down bugs and fixing them. Each change to the code required the revision of memory releases in the code parts which were modified. Finding space leaks was done by running small numbers of source photons using the MINT simulator and producing a trace of all heap allocations and deallocations. Leaks were then located and isolated by doing repeated runs with different function 'colourings' (user-assigned tags for different code blocks), since the allocations and deallocations are tagged by colour. Once a structure's size and source function were known, it could be identified and its release could be added to the code. It was a long and tedious process.

During debugging, Id's imperative approach to input/output was helpful. It was possible, at the beginning of execution, to open a file for writing and make its stream pointer globally available to the program. An M-structure lock for the file would also be created, to help synchronize the writes to the file and prevent interleaving of writes due to the parallel, asynchronous execution of the program. This allowed print statements to be embedded in the code anywhere. Typically, a suspect function would be given print statements at its entrance and exit points to show the input and output values of the function call.

Finding synchronization problems was very difficult. They typically manifested themselves as I- or M-structure memory errors, but usually only when running on multiple processors. There was no way to know which of the program's data structures was associated with the error, and the error was not even repeatable. Often it would not show up on the MINT simulator, since it simulates single-processor sequential execution whose order may not cause the illegal operation to occur. In the absence of any support for diagnosing synchronization errors, one could only gaze at the code and wait for inspiration. While the modified version of MCP-Id

now runs correctly in all of our tests, it is impossible to rule out the possibility that it still contains some subtle synchronization error which has not yet shown itself.

Eventually, full statistical agreement with MCNP was reached. The longest run consisted of 20,000,000 source photons, taking approximately 50 hours on the 16-node Monsoon machine at LANL. Its correct execution increases our confidence that the memory leak elimination and synchronization have been done correctly. With this code in hand it was easy to 'reverse engineer' a pure functional version, since the outputs of the heap-releasing Id code and the pure functional Id code should always be absolutely identical. Any divergence between them during debugging was easy to detect and isolate, leading directly to a fix. (One might also expect that the output of MCP-Haskell should be identical to that of MCP-Id. However, this is not quite true; differences in the handling of floating point numbers between Id and Haskell produce very small variations in those values, and occasionally a random value will fall on different sides of a decision threshold and cause a divergence between MCP-Id and MCP-Haskell. The difference is statistically insignificant.)

6.2 Heap problems in MCP-Haskell

Heap releases and control of parallelism are not a concern in the Haskell code, since automatic run-time garbage collection can release memory and the execution is sequential. Unfortunately, the evaluation order of MCP-Haskell requires a heap size which grows linearly with the number of source photons. The heart of the problem is data structure sharing interacting with lazy evaluation.

To understand the problem, consider a simple example:

```
pr :: [Integer] → (Integer, Int)
pr xs = (sum xs, length xs)
```

The list *xs* is shared by two function calls. With lazy evaluation, if the first of the return tuple is demanded, the entire list *xs* must be built, yet none of it can be garbage collected as *sum* traverses it because the list may still be needed if the second of the tuple is later demanded. Thus, after *sum* has finished its traversal the entire list *xs* exists in memory at once.

In figure 6 each of the three forks in the diagram represents a sharing of a data structure, analogous to the above example. Consider the sharing of the *event_lists* structure; if millions of source photons are run, many millions of events will live simultaneously in this structure and exhaust heap memory.

Obviously this is a problem of evaluation order. If the two consumers would interleave their accesses, it would not be necessary for the entire list to live in memory at one time. A simple attempt to do this in Haskell might look like this:

```
pr :: [Integer] → (Integer, Int)
pr = f 0 0
```

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

```
where
f v1 v2 [] = (v1, v2)
f v1 v2 (x:xs) = f (v1+x) (v2+1) xs
```

But this is not sufficient. Laziness still prevents the addition to the second parameter of f when the first of the tuple is demanded.

In this simple example the desired behaviour can be obtained by providing strictness information, thus allowing all the additions for each invocation of f to be be performed together:

```
pr :: [Integer] → (Integer, Int)
pr = f 0 0
    where
    f v1 v2 [] = (v1, v2)
    f v1 v2 (x:xs) = f ((v1+x){-#STRICT#-}) ((v2+1){-#STRICT#-}) xs
```

This function uses resources that are constant with respect to the length of the list. The application of this technique to MCP-Haskell will be considerably more complicated than this, and it is questionable whether a programmer should be expected to go to such lengths in modifying source code.

7 Performance

The overriding aspect of the MCP-Functional codes' performance is their inability to run usefully large problems because of resource issues. While the heap-managed Id code is not the primary focus of this paper, its ability to do long runs makes it the only one of our codes that can be compared for performance to the Fortran MCNP. It is important to realize, however, that while MCP-Id is patterned in many ways after MCNP, there are many differences between them which can make blind comparison dangerous. Specifically, the code doing the geometric computations (determining identities of cells, etc.) was not patterned after the MCNP code, and it is likely that significant differences exist in those computations. Also, MCNP collects more information and does analysis of the results to give the user an idea of the degree of success in translating the problem into a well-behaved specification.

The performance of the heap-managed MCP-Id on Monsoon is not embarrassing when compared to MCNP on a Cray YMP. For our running example, with 5000 source photons run on single processors, the performance is: MCP-Id – 108.4 seconds; MCNP – 5.2 seconds.

There are many factors that can affect performance on these machines, including compilers, memory characteristics, instruction differences, CPU design, and clock speed (Monsoon has a 100 ns cycle versus the YMP's 6 ns cycle.) A full performance comparison will require much closer investigation of these issues and will be the subject of future work.

https://doi.org/10.1017/S0956796800001374 Published online by Cambridge University Press

8 Conclusion

The Monte Carlo photon transport problem codes cleanly in a pure functional way, and MCP-Functional produces results which are statistically comparable with those of MCNP. These functional codes should be useful to language implementors since they use many features and coding styles of the languages. Laziness was found to be helpful in that it allows a somewhat cleaner style of coding, but it was found to cause resource management problems in large shared data structures.

Development of MCP-Functional had to proceed in an order one would not have expected. Rather than develop a pure functional version, and then modify it as necessary to let it run large problems, an efficient heap released version had to be done first so that statistically meaningful results could detect bugs. Without the ability to do explicit heap releases in Id (tedious though it was), and to print output traces by side-effecting, it would have been nearly impossible to debug the code due to its statistical nature.

Resource management issues were by far the biggest obstacle to the development of MCP-Functional. The problems of correctly reclaiming heap memory, and managing the order of program evaluation to keep peak requirements within the machine's capacity, must be solved if functional languages are to become accepted for many scientific codes. Requiring the programmer to provide explicit releases of heap memory goes against the grain of functional programming since it introduces imperative and time-dependent operations and requires precise understanding and control of evaluation order in the program. Either compilers must become very smart in their handling of this issue (Hicks, 1993), or languages must give programmers the means to express in a straightforward way the information needed to let an executing program live within its resource limits.

Further research directions include a study of heap usage in the Haskell environment, compilation techniques to avoid the creation of complete structures in multiple-use situations, and an exploration of Sisal's strengths and weaknesses in expressing this code, especially with regard to the use of Sisal's streams to express producer–consumer relationships and to help control heap memory. Some interesting research directions also suggest themselves in the area of compiler-directed program transformations similar to those which were done manually to allow an MCP-Id version to run large problem sizes.

Obtaining source code and documentation

The source codes are available by anonymous ftp from *schubert.cs.colostate.edu*, directory *pub/MCP-functional*. In addition to the Id and Haskell sources, there are also the full cross section files for the 94 nuclides, allowing users to set up other problems with different nuclides.

Acknowledgement

We wish to thank Tom Booth for patiently helping us understand MCNP.

References

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice Hall.

Briesmeister, J. F. ed. (1993) *MCNP–A General Monte Carlo N-Particle Transport Code, Version 4A*. Los Alamos National Laboratory Report LA-12625-M.

Carter, L. L. and Cashwell, E. D. (1975) *Particle-Transport Simulation with the Monte Carlo Method*. ERDA Critical Review Series, TID-26607.

Hendricks, J. S. (1991) Effects of Changing the Random Number Stride in Monte Carlo Calculations. *Nuclear Science and Eng.* **109**(1): 86–91.

Hicks, J. (1993) Experiences with Compiler-Directed Storage Reclamation. In: *Conference on Functional Programming Languages and Computer Architecture*. ACM Press, pp. 95–105.

Hicks, J., Chiou, D., Ang, B. S. and Arvind. (1993) Performance studies of Id on the Monsoon dataflow system. *J. Parallel and Distributed Computing* **18**: 273–300.

Hudak, P. and Fasel, J. (1992) A gentle Introduction to Haskell. *ACM SIGPLAN Notices* **27**(5).

Hudak, P., Peyton Jones, S. and Wadler, P. eds. (1992) Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices* **27**(5).

Morais, D. R. (1986) ID WORLD: An Environment for the Development of Dataflow Programs written in Id. MIT/LCS/TR-365.

Nikhil, R. S. (1990) Id Version 90.0 Reference Manual. Computational Structures Group Memo 284-1, Massachusetts Institute of Technology.

Whalen, D. J., Hollowell, D. E. and Hendricks, J. S. (1991) *MCNP: Photon Benchmark Problems*. Los Alamos National Laboratory Report LA-12196.