A Comparison of Functional and Imperative Programming Techniques for Mathematical Software Development

Scott Frame and John W. Coffey
Department of Computer Science
University of West Florida
Pensacola, FL. 32514

ABSTRACT

Functional programming has traditionally been considered elegant and powerful, but also somewhat impractical for ordinary computing. Proponents of functional programming claim that the evolution of functional languages makes their use feasible in many domains. In this work, a popular imperative language (C++) and the leading functional language (Haskell) are compared in a math-intensive, real-world application using a variety of criteria: ease of implementation, efficiency, and readability. The programming tasks that were used as benchmarks involved mathematical transformations between local and global coordinate systems. Details regarding the application area and how language features of both languages were used to solve critical problems are described. The paper closes with some conclusions regarding applicability of functional programming for mathematical applications.

1. INTRODUCTION

Imperative programming performs computation as a sequence of statements that manipulate stored data until a desired result is achieved. The functional style of programming, in contrast, represents programs as relationships between mathematical expressions which are based on dependencies. Functional programming has been described as powerful and expressive, yet it has never achieved the success and widespread use of imperative programming. An impediment to the growth of functional programming is that many tasks are most naturally attacked by imperative means and cannot be represented as readily in a functional manner. Some functional languages include imperative constructs. These inclusions compromise the functional model, but allow the imperative tasks to be accomplished through the most direct means. In other cases imperative languages have been equipped with some functional tools to make them more expressive.

Functional language advocates argue that functional languages have evolved substantially over the years, making them suitable for a broader range of tasks. For example, one key improvement is the advent of the monad, a programming construct that allows developers to produce code which interfaces easily with the outside world in a sequential manner while preserving a distinct separation between purely functional code and I/O tasks. This development and other advances in functional language design have led advocates in the functional programming community to claim that modern functional languages are as well equipped to deal with real-world programming tasks as any popular imperative language [1]. This work seeks to examine this claim by evaluating the benefits and drawbacks of imperative and functional programming in a side-by-side comparison on a mathematical application.

The remainder of this paper contains a description of the implementation of world coordinate system transformations for time-space-position information (TSPI) in both imperative and functional languages. C++ was used as the imperative language and Haskell [2] was used as the functional language. The two languages are compared using a variety of criteria: ease of implementation, runtime efficiency, readability and correctness. Issues pertaining to data types, language constructs employed in local and global conversions, useful features, and performance are analyzed.

2. THE PROBLEM AREA: COORDINATE SYSTEM TRANSFORMATIONS

The programming task involved the implementation of world coordinate system transformations for time-space-position information (TSPI). Positional information can be represented in a variety of ways, each representation having particular applications for which it is useful. These coordinate systems can be global or local. The most useful coordinate systems for global positioning are the geocentric (earth-centered, earth-fixed, rectilinear) system and the geodetic (latitude, longitude, height) system. The Global Positioning System (GPS) uses the geocentric system internally, although most GPS devices display or report coordinates in a geodetic format, which is easier for humans to read and understand. The most common local systems are the rectilinear topocentric system and the angular topocentric system. The angular topocentric system provides positional information in the form of azimuth, elevation and range values with respect to a fixed origin; this is the format most radar systems use to report the location of radar tracks. Local rectilinear systems are also very useful for representing entities near a fixed origin or range center. Figure 1 shows conversion paths for these conversions.

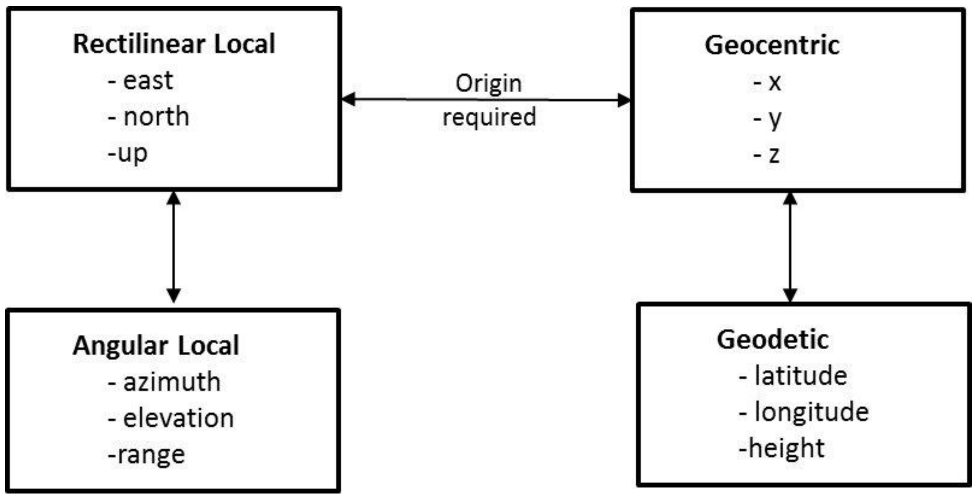

Figure 1. Conversion paths among the various coordinate systems.

Implementation of the various conversions involves trigonometry, linear algebra and iterative estimation. TSPI transformations provide a reasonable application area in which to compare the mathematical capabilities of the two programming paradigms. It does not seem to favor either one of the languages considered in any important manner. The coordinate transformations are the focus of the comparison, and consequently I/O tasks and user interaction are not considered in the current study. Iteration is not extensive in the calculations, which at first may seem to remove an aspect that can be challenging to implement functionally. However, with the modern functional tools that Haskell provides to recurse through lists and to implement list comprehensions, iteration is not a significant issue.

Data Types

An important concern in any programming language is the way in which data is represented. In C++, as with any imperative, object-oriented language, the class is the most basic construct used to create data types. In Haskell, the algebraic data type is the most relevant language construct for the current problem. Algebraic data types possess some characteristics of structures, enumerations and unions from C. These types can be defined in more than one way; if defined using “record syntax” the definition looks similar to a C struct, and automatically creates accessor functions for each component. To use the accessor, which is actually just a normal Haskell function, a value of that type is passed to it.

Although this mechanism appears strange from an imperative programming perspective, it is not drastically different from a class accessor function in an object-oriented language. Algebraic data types can be used to create representations of each coordinate system. Classes are typically defined in the C++ implementation. These various data types are passed into conversion functions and the components of each type are manipulated to obtain a transformation to a new type. The use of compound data types in both languages eliminates the need to pass multiple arguments into each conversion function and provides type-checking for added safety.

Local Conversions

Transformations between local systems are the simplest of the conversions. They require only basic trigonometric operations. Conversions from a rectilinear (east, north, up) system to an angular (azimuth, elevation, range) system are illustrative. The azimuth and elevation values are vector angles and the range is the vector magnitude.

The C++ version calculates the `range` value first since it is used in the calculation for `elevation` later. In the Haskell implementation, the local variables in a `where` clause are calculated based on dependency, not sequence, since the `range` calculation is listed after the `elevation` calculation. Therefore, statement order does not matter.

Another difference between the two implementations was found in the treatment of a conditional. An `if` statement in the C++ source was replaced in the Haskell implementation with an additional function application to handle either alternative. Although they exist in Haskell, `if` statements are rarely used. A preferred (more readable) technique for this sort of selection is a feature called a guard (indicated by a vertical bar), which uses pattern matching to select between available options.

Global Conversions

Transformations between global systems are more involved. The geocentric system is the easier of the two with regard to computations, but the geodetic system is usually much more useful from a user standpoint. Geodetic coordinates are an angular projection onto an ellipsoidal model of the earth, which is most closely approximated by an oblate spheroid. The geodetic characteristics of the earth are obtained by surveying; the most widely used geodetic system is the 1984 World Geodetic System [3] which defines the ellipsoidal characteristics of the earth with two parameters: the semi-major axis (a), and the inverse flattening (1/f). These constants and other parameters derived from these values are used to perform conversions between the geocentric and geodetic systems.

The conversion from geodetic to geocentric was a straightforward set of trigonometric statements; the imperative and functional implementations looked similar. The inverse conversion, from geocentric to geodetic was more interesting. This conversion was achieved by estimation, and there are numerous iterative approaches used to perform this calculation [4]. The approach used here is known as the Hirvonen and Moritz iterative method, which initially sets the height to 0, and uses this value to calculate an initial estimate for latitude. The radius of curvature in the prime vertical, height above ellipsoid (HAE), and the geodetic latitude are then continually refined until the maximum error between successive iterations of height calculations is less than a predetermined acceptable limit.

In the imperative approach, three variables (`lat, hae`, and `primeVerticalRadiusOfCurvature`) are continually refined in a `do .. while` loop. Since the algorithm for this estimation is inherently sequential by nature, it seemed that the conversion might be difficult to implement in a functional manner. In Haskell, the conversion was achieved by using a potentialy infinite recursive *list comprehension*, a novel idea in functional languages. The first three statements of a `where` clause were used as inputs to a function named `hirvonenMoritzIteration` which performed the iterative estimation and returned a `tuple` containing the height and latitude values. List comprehensions use generator expressions to define successive elements of a list, and because Haskell is a pure, lazy functional language, the list elements are not produced until they are evaluated.

Two functions, `computeHae` and `computeLat` were straightforward calculations that depend on the previous values of latitude and height. The computation required three list comprehensions: `HaeList`, `HaeLat`, and `HaeLatList`. The `HaeLatList` defines its first element and then recursively defines all successive elements based on that first element, using the first element as input into the `computeHae` and `computeLat` functions. The result is a list of tuples containing the height and latitude values, with each successive tuple refining one of the two values. The `HaeList` and `HaeLat` extract the respective values from every other entry and then the `HmList` zips up these two lists into tuples at each iteration so that each tuple is a refinement of the height and latitude.

The final task is to scan the `HmList` to determine when the error between successive iterations is small enough to stop. This is achieved with a recursive function `findEstimate` with two pattern guards. The first pattern guard specifies that as long as the difference between the new height and previous height is greater than the `altitudeLimit` and a certain number of iterations has not been exceeded, `findEstimate` is called recursively; when finished `findEstimate` returns the tuple containing the final approximation of the height and latitude.

Local-to-Global and Global-to-Local Conversions

When converting from a local system to a global system or vice-versa, the conversion involves the application of a rotation matrix to the input vector to achieve a translation to the target system. This rotation matrix is built using the origin for the local system. The creation of matrix data types allows for convenient passing of data and computation. Three-by-three and three-by-one matrix data types are needed for the calculations. They are obtained in the C++ implementation with classes and in the Haskell implementation with algebraic data types. With the matrix types in place, building a rotation matrix looks very similar in the two frequently quite different paradigms.

For velocity and acceleration translation, only the rotation must be performed. For positional transformations, the positional offset between the global system and the local system origin must also be resolved. For the C++ implementation, it makes sense to allow the user of the function to supply a rotation matrix independently from the origin if desired. This way the rotation matrix can be stored for repeated conversions without having to rebuild it each time the conversion is performed. Also in the C++ implementation, the matrix classes can easily be equipped with overloaded arithmetic operators for convenient and readable arithmetic operations. For a local to global conversion, the transpose of the rotation matrix must be multiplied with the input vector. The Haskell implementation is similar, except the rotation matrix is built within a function. In place of the overloaded operator in the C++ version is a `transposeMultiply` operation in Haskell. For the global to local conversion, the relationships are reversed. Instead of multiplying the transpose, the regular rotation matrix is multiplied with the positional-offset-adjusted vector.

3. LIBRARY DESIGN ISSUES

Implementation of the basic conversions has been completed in both languages, but an outstanding issue is how the implementations will be packaged into libraries. This section contains a discussion of some of the issues that have been identified pertaining to this packaging.

Perhaps one of the most important characteristics of a reusable software library is ease of use and practicality. The library should make the job of the application developer easy and should abstract away as much implementation detail as possible. Experience using a legacy TSPI conversion library is perhaps the most valuable resource available to an individual developing a new library of this sort. Legacy C libraries are common in many domains; they can provide widely usable resources to a variety of functional areas and languages.

Experience using a legacy C TSPI library, however, can lead one to desire more powerful constructs and abstractions that might be produced with a library developed in an object-oriented or functional approach. An ideal TSPI processing capability should be able to establish multiple layers of abstraction and perform conversions and other essential operations behind the scenes, instead of simply providing a set of functions to chain together for processing data. An advanced object-oriented TSPI processing architecture is proposed with generic interfaces for representing position, velocity, acceleration and orientation.

A goal of the library architecture for both the C++ and Haskell implementations is that an application developer should not have to perform explicit data conversions if positional data is desired in a different format. A library user should be able to set the position using any one of the specific coordinate system representations and then simply ask for any field of any one of the other representations.

The conversions will be performed implicitly by the library only when a field of one of the unset specific formats is requested. Flags can be used to keep track of which formats have been populated by the library and so conversions will only occur when necessary. The use of state is essential in this architecture, since the top class and mid-level classes must keep track of which formats have been populated. Architectures similar to the Position representation are needed for the Velocity, Acceleration and Orientation vectors as well.

From a high-level design standpoint, the TSPI library functionality should contain three levels of detail for entities in time and space. The most basic entity to track is simply a point in space with a unique identifier, a position, and a time associated with that position. This is the most applicable interface for tracks returned from a radar or some other basic surveillance mechanism. A more detailed interface onboard GPS systems should fall into this category, and this provides a useful common interface that can be used to treat TSPI data generically so that specific logic and data manipulations do not have to be applied to each data source. Finally, an even more specific level can be used only when the data needed is not a part of the common interface.

The envisioned architecture is stateful, and as such does not translate to the functional domain. Haskell features a language construct called a `typeclass` that is distinctly different from the classes of object-oriented software development. `Typeclasses` are designed to provide a common interface for functions into which different types can be injected, and it seems that they may be useful for a developing an abstraction that is more useful from a functional domain. Put another way, `typeclasses` may be able to get a functional TSPI library architecture "part of the way there" with regard to the proposed level of abstraction, and there may be other features in the Haskell toolset which can be applied to achieve something closer to the desired capability; if so, these techniques are beyond the scope of this study.

4. RESULTS OF THE STUDY

This section contains a summary of the results obtained in the comparison of the two implementations. Comparisons were made pertaining to relative ease of use and runtime performance.

Ease of Use of the Languages
The software engineer who implemented the programs succeeded in creating all the conversions in both programming languages. As alluded to before, different language constructs, as summarized in Table 1, were used, but in many cases, the code looked similar. Readability might be an initial issue for programmers lacking a functional programming background, but it is anticipated that this issue would quickly recede. The functional nature of Haskell ensures that potential sources of errors such as an incorrect ordering of statements when one variable depends on computation of another inside a loop, is not an issue. The implementation of the functions pertaining to `Position` were analyzed to compare the lines of code needed. An open source tool named SLOCCount [5] was used. The C++ code required 962 lines and the Haskell implementation required 173 lines.

Table 1. A Comparison of Language Features used in the Current Study.

Language	Selectors	Iterators	Data Types	Matrix Operations
C++	`If .. else, switch`	`do .. while while, for`	Classes	Operator overloading
Haskell	Guards and pattern matching	List comprehensions, guards, and recursion	Algebraic data types	Function application

For the conversion involving the iterative estimation, there were some challenges to overcome in the Haskell implementation, but most of the difficulty can be attributed to lack of experience with the functional programming paradigm. When considering the usefulness of the architecture and the desire to have an interface comparable with the object-oriented design proposed, it is difficult to know (without more knowledge of functional library design techniques) whether an elegant abstract solution is possible that is more powerful than the current object-oriented design.

Performance

Both the C++ and Haskell code were executed on a 2.0Ghz, Intel Core 2 processor running Ubuntu 10.10. Both the C++ and the Haskell were natively compiled, with g++ and the Glasgow Haskell Compiler [6] respectively. Table 2 contains results of various conversions showing the ratio of Haskell to C++ performance. For instance, the conversion of geocentric to Geodetic (geocentToGeodetic) took 32.7 times as long to execute in Haskell as in C++. Overall, the Haskell required 30 to 70 times longer than the C++ to execute. No particular performance problems were noted in the routines that required iteration.

Table 2. Performance Comparisons on various conversion functions.

Function	Ratio	Function	Ratio
geocentToGeodetic	32.7:1	geodetToGeocen	35.9:1
geocentToRectilinLocal	65.6:1	geodetToRectilinLoc	50.6:1
geocentToAngLocal	48.7:1	geodetToAngLoc	45.6:1
rectilinLocToGeocen	69.5:1	angLocToGeocen	56.1:1
rectilinLocToGeodet	48.4:1	angLocToGeodetic	42.1:1
rectilinLocToAngLoc	25.5:1	angLocToRectiLoc	40.2:1

5. CONCLUSIONS

The goal of this work was to determine the usefulness of functional programming in a math-intensive real-world problem and to provide a comparison with imperative programming. This study focuses on the implementation of coordinate system conversion routines, but also seeks to examine the ease of implementation and the usability of the languages being compared. With regard to the development of the conversion routines, the imperative and functional implementations employed different features, but each achieved the objective and neither seemed ill-equipped for the task. Some of the functional programming constructs and language features might seem unfamiliar to a programmer coming from the procedural programming world, but the functional program source code was a small fraction of the size of the procedural program. Furthermore, several ease-of-use aspects were identified: guards for selection and the lack of need to consider statement sequence in Haskell simplified implementation. With regard to performance, the procedural language was far more efficient, even though the functional program was natively compiled. In real-time applications, the performance differential could be critical, but in less time-critical applications, the functional language performance is satisfactory. Overall, while it might be concluded that the functional approach has many elegant, highly expressive features that potentially simplify software development, the performance deficiencies probably limit applicability of functional programming languages generally for mathematically intensive applications.

6. REFERENCES

[1] Meijer, E. (2007). Confessions of a Used Programming Language Salesman. Proceedings Of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, 677-694.

[2] Stewart, D. (2009). Real World Haskell. Sebestapol, CA: O'Reilly Media, Inc.

[3] Torge, W. (2001). Geodesy. Berlin, Germany: Walter de Gruyter GmbH & Co.

[4] Burtch, R (2006). A Comparison of Methods used in Rectangular to Geodetic Coordinate Transformations. Orlando, FL: 2006 ACSM Annual Conference and Technology Exhibition

[5] Wheeler, D. (2004). SLOCCount. http://www.dwheeler.com/sloccount/

[6] GHC. (2011). The Glasgow Haskell Compiler. http://www.haskell.org/ghc/