Haskell vs. Ada vs. C++ vs. Awk vs. ...
An Experiment in Software Prototyping Productivity*

Paul Hudak
Mark P. Jones

Yale University
Department of Computer Science
New Haven, CT 06518
{hudak-paul,jones-mark}@cs.yale.edu

July 4, 1994

Abstract

We describe the results of an experiment in which several conventional programming languages, together with the functional language Haskell, were used to prototype a Naval Surface Warfare Center (NSWC) requirement for a *Geometric Region Server*. The resulting programs and development metrics were reviewed by a committee chosen by the Navy. The results indicate that the Haskell prototype took significantly less time to develop and was considerably more concise and easier to understand than the corresponding prototypes written in several different imperative languages, including Ada and C++.

*This work was supported by the Advanced Research Project Agency and the Office of Naval Research under Arpa Order 8888, Contract N00014-92-C-0153.

1 Introduction

The functional programming community has worked long and hard to make functional languages good enough for use in industrial and commercial applications; the community dreams of the day when Haskell or ML will replace C in real applications! Although we are now reaching a point where some of these dreams are becoming reality, there is another front upon which we can hope for success: the use of functional languages in *prototyping*.

This idea has of course been recognized for a long time (see for example [Hen86]), but recent trends in the software industry have increased the importance of this idea: a strong interest in the *disciplined* use of prototyping has arisen in contemporary theories of software engineering. Perhaps ironically, it is the *largest* of software developers, on the *largest* of projects, that have most recognized the value of prototyping, and are beginning to integrate it into their software development process as a legitimate, well thought-out phase in the software life-cycle.

The critical fact that highlights the utility of prototyping is a very simple one: *on most software projects, the overall problem is not well understood*, or perhaps worse, is *wrongly understood*. The primary goal of prototyping is simply to understand better both the problem and the solution space! This has the effect of *reducing risk*: studies have shown that the cost to fix a fault early in the software life-cycle is exponentially less than the cost to fix it later.

In this light, the benefits of software prototyping can be seen in many different ways. Effective prototyping permits rapid exploration of several different spaces, each affecting a different phase of the software life-cycle:

1. Exploration of the *problem* space helps us to better understand the problem itself, thus facilitating the requirements acquisition phase.

2. Exploration of the *solution* space helps us to explore the viability of different solutions, thus facilitating the specification phase.

3. Exploration of the *design* space helps us to experiment with different software structures, thus facilitating the design phase.

4. Exploration of the *resource* space enables us to experiment with specific tool and resource commitments, thus facilitating the implementation phase.

The above sequence reflects our potentially poor understanding of the problem, the solutions, *and* the available tools and resources; there are many dimensions to fully understand and develop a large system. If one of these dimensions is in fact fully understood, that component of prototyping can be de-emphasized or eliminated; but it is extremely rare that all of the dimensions are fully understood for any system of reasonable magnitude and complexity.

Although it seems that the emphasis here is on the use of prototyping only at early phases in systems development, there are several ways in which it can permeate the software life-cycle: First, modern notions of large systems design are iterative in nature; each of these iterations may involve some degree of prototyping. Second, a key product of prototyping is a document that describes what has been learned; often this document is the prototype itself, which serves as an executable specification of the problem and influences not just implementation, but test, integration, and documentation as well. Finally, the prototype itself may be refined for later use as the actual delivered code; this brings us full circle to the use of functional languages in replacing C. The refinement process must ensure that all aspects of performance are achieved, and may involve the use of inter-operability mechanisms to rewrite critical "inner loops" in a lower-level language.

We feel that the modern notion of software prototyping outlined above is a grand opportunity for the functional programming community to have a major impact on the way software is developed in the industrial, commercial, and military sectors. In this paper we describe an experiment that lends weight to this argument.

2 The NSWC Experiment

In recognition of the importance of prototyping in software development, the United States Advanced Research Projects Agency (ARPA) launched a program in 1989 to develop the languages, tools, and infrastructure necessary to enhance the prototyping process. The resulting "ProtoTech" program yielded several useful technologies, most notably a series of programming languages targeted specifically for prototyping applications. Yale University's involvement emphasized the use of the then-new functional language *Haskell* [HPJWe92].

In the fall of 1993, in an effort to measure the success of the ProtoTech program, ARPA conducted an experiment in collaboration with the Office of Naval Research (ONR) and the Naval Surface Warfare Center (NSWC). The experiment was conducted as follows:

1. A simplified version of real-world problem was chosen by NSWC. This problem, a *geometric region server* (geo-server), is one component of a much larger system, NSWC's AEGIS Weapons System (AWS), which NSWC is in the process of redesigning.

2. An informal, English-language description of the geo-server was given to all participants in the experiment. This description was oriented toward black-box behavior, and did not include performance requirements [Car93].

3. A meeting was held at NSWC in Dahlgren, Virginia, at which participants were briefed on the geo-server problem and given additional clarifying material.

4. The participants, each considered an expert programmer in one of the programming languages being tested, was asked to write a fully functional prototype of the geo-server, while keeping track of software development metrics such as development time and lines of code and documentation.

5. The prototypes and metrics were collected by NSWC and distributed to a distinguished panel of computer scientists and software engineers for evaluation along several dimensions.

6. A second meeting was held at NSWC, at which the participants presented their solutions. Discussions also transpired concerning the proper metrics by which a good prototype should be judged.

7. The review panel wrote a final report summarizing their findings [LBK+94].

The remainder of this paper discusses not only the results of the experiment as reported by the review panel, but many other issues relating to the experiment and the prototypes, in particular the Haskell prototype. The views presented reflect those of the authors, and should not be construed as views of ARPA, ONR, NSWC, the review panel, or any other participants in the NSWC experiment.

Although many good things can be said about this experiment, it is important to realize that studies of this sort are *extremely* difficult to design, conduct, and evaluate. We point out in particular the following potential criticisms:

1. The NSWC experiment was conducted in a very short time-frame with very little direct funding. Thus many corners had to be cut, including significant simplification of the problem itself; the largest of the prototypes was only 1200 lines of code.

2. The geo-server specification was by nature ambiguous and imprecise, thus leading to some variation in the functionalities of the developed prototypes. (On the other hand, the specification is probably typical of that found in practice, especially during requirements acquisition.)

3. The participants were trusted entirely to report their own development metrics. In addition, not all of the participants attended the first meeting at NSWC; those who did attend were advantaged.

4. No guidelines were given as to what exactly should be reported. For example, development times usually included documentation time, but there were major variations in the code-size/documentation-size ratios. In addition, different methods were used to count lines of code for each language, and it is not clear how one line of code for one language relates to that of another.

5. The review panel had very little time to conduct a really thorough review. For example, none of the code was actually run by the panel; they relied on the written reports and oral presentations.

Now that we've exposed the criticism, let's consider the virtues of this experiment, especially as they relate to the functional programming community. After all, there have been many reports of factors of 10 or more reduction in code size when using functional languages, so what additional value is gained from the NSWC experiment?

In fact, we believe that the NSWC experiment is unique and noteworthy for several important reasons:

1. The experiment itself was neither designed nor conducted by either functional programmers or academicians. It was conducted by NSWC, whose software development staff is highly-regarded, with many years experience developing large, complex, software systems.

2. Although simplified, the problem itself was not a toy; it is considered to be an important component of a much larger system being developed at NSWC.

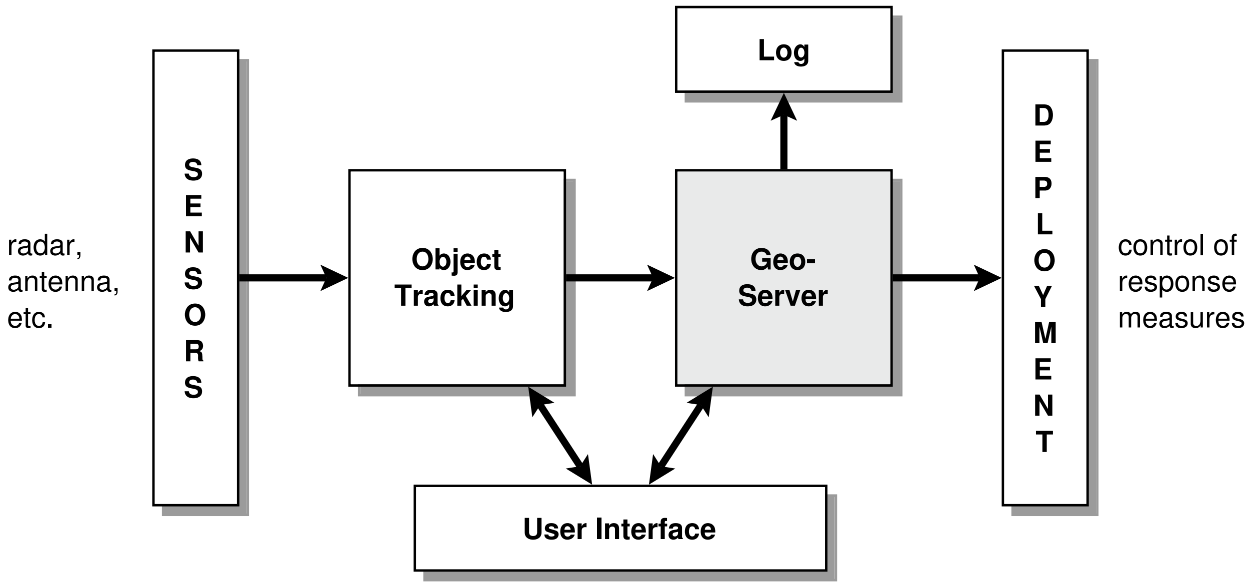

Figure 1: Simplified Aegis Weapons Systems Diagram

3. Imperative code was not rewritten "after the fact" in a functional language and then compared to the original program. Nor were all of the programs written by the same person. Instead, the problem statement was given to an expert programmer in each of several different programming languages.

4. Although difficult to compare, the metrics include the time it took each participant to create his prototype. Short programs have their advantages, but if they take twice as long to develop, the advantages may be nullified.

5. The NSWC experiment represents a military application and customer. Military applications represent a large and critical use of software systems, and their correctness and reliability is of obvious concern.[1]

3 Problem Description

Space limitations preclude our inclusion of the full problem specification. We will instead describe it briefly, giving an example of its behavior, and refer the interested reader to reference [LBK$^+$94] for the full specification.

As mentioned earlier, the geo-server is just one part of a much larger system, as shown in Figure 1. This diagram illustrates the essential role that the geo-server plays, although its oversimplification does not give justice to the size and complexity of the other components.

[1]For those who object to this use of functional languages, we suggest reading the rest of this paper as if it were about a video game. We do not intend this as a joke: video war games must have components in them similar to the "geo-server" described in this paper, and video games are certainly an important commercial application.

The input to the geo-server consists of data that conveys the positions of various ships, airplanes, and other objects on the globe; the output consists of relationships between these objects as computed by the geo-server. To gain some intuition about the functionality of the geo-server, it is helpful to observe a typical input/output pattern. The input data is best conveyed using a *map*, for example as shown in Figure 2. In this diagram:

1. Friendly ships – aircraft carriers, battleships, etc. – are represented as triangles. The geo-server is expected to monitor the position/status of many such ships.

2. Surrounding each ship are several "zones" of interest: an *engageability zone*, an annulus within which engagement with hostile craft is allowed; a *weapon doctrine*, a pie-shaped region which the geo-server must monitor; and *slaved doctrine*, regions surrounding friendly ships that are to be protected.

3. In addition, *tight zones* are fixed regions in space of arbitrary size and shape that represent important regions such as commercial aircraft flight patterns, civilian population areas, etc. These are shown in polygonal form in the figure.

4. Friendly aircraft are represented as squares, and hostile aircraft as hexagons. The geo-server's main task is to determine the presence of these objects in engageability zones, weapon doctrines, and tight zones.

In the NSWC experiment, each participant was given a sequence of 5 maps similar to that in Figure 2, representing a temporal sequence of movements of the various objects. How these maps were represented in the geo-server was up to the participants. Note that one consequence of the problem simplification process is evident here: the input data uses a geometric model that is *two-dimensional* and *cartesian*. An enhanced Haskell geo-server for both three dimensions and a curved-earth model was developed, but is not reported here (see [CHJ93]).

To get a feel for the output required of the geo-server, here is the output generated by the Haskell prototype:

```
Time 0.0:
commercial aircraft: (38.0,25.0)
 -- In tight zone
hostile craft: (258.0,183.0)

Time 20.0:
commercial aircraft: (58.0,30.0)
 -- In tight zone
hostile craft: (239.0,164.0)

Time 40.0:
commercial aircraft: (100.0,43.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (210.0,136.0)
 -- In carrier slave doctrine

Time 60.0:
```

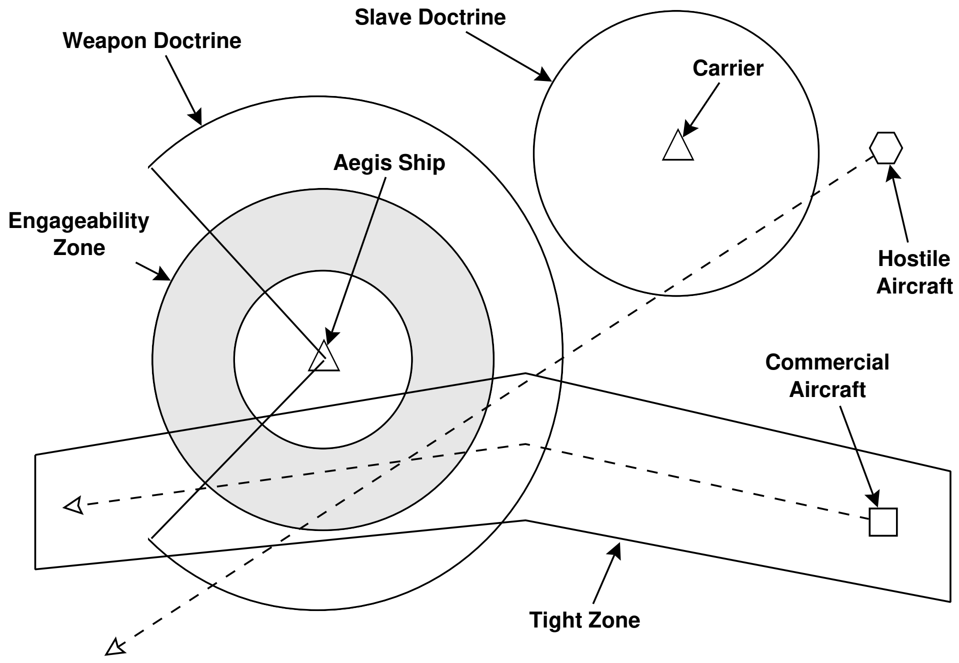

Figure 2: Geo-Server Input Data

```
commercial aircraft: (159.0,36.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (148.0,73.0)
 -- In carrier slave doctrine

Time 80.0:
commercial aircraft: (198.0,27.0)
 -- In engageability zone
 -- In carrier slave doctrine
 -- In tight zone
hostile craft: (110.0,37.0)
 -- In tight zone
```

And to get a feel for the *potential* of the geo-server, here is the output of an enhanced version of the Haskell prototype, which satisfies several "extra credit options" posed in the original specification (but whose development metrics are not included in this paper; see [CHJ93]):

```
Time 0.0:
commercial aircraft: (38.0,25.0) --> (0.0,0.0)
 -- Currently in tight zone
hostile craft: (258.0,183.0) --> (0.0,0.0)

Time 20.0:
commercial aircraft: (58.0,30.0) --> (1.0,0.25)
 -- Expected in weapon doctrine at t+131.3
 -- Expected in engageability zone at t+52.5
 -- Expected in missile range at t+52.5
```

```
 -- Expected in carrier slave doctrine at t+94.5
 -- In tight zone, expected to leave at t+93.2
hostile craft: (239.0,164.0) --> (-0.95,-0.95)
 -- Expected in weapon doctrine at t+55.6
 -- Expected in engageability zone at t+65.0
 -- Expected in missile range at t+65.0
 -- Expected in carrier slave doctrine at t+26.9
 -- Expected in tight zone at t+111.5

Time 40.0:
commercial aircraft: (100.0,43.0) --> (2.1,0.65)
 -- Expected in weapon doctrine at t+22.8
 -- In engageability zone, expected to leave at t+10.2
 -- Expected in missile range at t+34.3
 -- Expected in carrier slave doctrine at t+31.8
 -- In tight zone, expected to leave at t+19.8
hostile craft: (210.0,136.0) --> (-1.45,-1.4)
 -- Expected in weapon doctrine at t+19.6
 -- Expected in engageability zone at t+26.1
 -- Expected in missile range at t+31.6
 -- In carrier slave doctrine, expected to leave at t+44.3
 -- Expected in tight zone at t+55.1

Time 60.0:
commercial aircraft: (159.0,36.0) --> (2.95,-0.35)
 -- Expected in weapon doctrine at t+16.2
 -- In engageability zone, expected to leave at t+20.9
 -- Expected in carrier slave doctrine at t+3.9
 -- In tight zone, expected to leave at t+33.9
hostile craft: (148.0,73.0) --> (-3.1,-3.15)
 -- Expected in engageability zone at t+1.4
 -- In carrier slave doctrine, expected to leave at t+2.1
 -- Expected in tight zone at t+4.8

Time 80.0:
commercial aircraft: (198.0,27.0) --> (1.95,-0.45)
 -- In engageability zone, expected to leave at t+11.9
 -- In carrier slave doctrine, expected to leave at t+4.0
 -- In tight zone, expected to leave at t+31.3
hostile craft: (110.0,37.0) --> (-1.9,-1.8)
 -- In tight zone, expected to leave at t+5.0
```

This output was generated from the same input data, but contains extrapolated ship and aircraft positions as estimates of future times at which various events would occur.

The scope of this experiment can be gleaned somewhat from reference [Car93], from which we quote:

> "The AEGIS Weapons System (AWS) is an extensive array of sensors and weapons designed to defend a battle group against air, surface, and subsurface threats."
>
> . . .
>
> "The AWS is undergoing a major redesign. As part of this redesign several functionally similar algorithms (currently dispersed throughout the system) have been proposed

as targets for consolidation. It is speculated that consolidation of these algorithms can reduce redundant computation and offer a better opportunity to utilize the parallel computation capability of modern architectures."

. . .

". . . the role of AEGIS is changing from one stressing defense against massive raids in open ocean to conflicts fought in the complex and contact rich environments of confined bays and straits. This change requires . . . a refined ability to differentiate and separate friend and foe. This complexity makes the ability to quickly test and fine tune algorithms through prototyping very compelling."

. . .

"The current AEGIS consists of millions of lines of CMS-2 code. This experiment, on the other hand, was to take place during a two week period, and the initial guidance was that about five person days of effort should be devoted to each prototype."

Reference [Car93] also carefully defines the objectives for the experiment:

- prototype "algorithms which compute presence of tracks within geometric regions in space." These are referred to as "doctrine-like capabilities."
- "minimize work to achieve a first delivery"
- "allow room for incremental enhancement."

as well as the criteria for evaluating the prototypes which are produced; we quote the following list:

- Extensibility – ease with which the prototype can be extended to fulfill additional requirements.
- Understandability – rapidity with which the prototype can be absorbed and altered.
- Appropriateness – the expressiveness of the language relative to the task at hand.
- Accuracy – correct functioning of the prototype.
- Compactness – capability per line of code.

4 Participating Languages

The programming languages used in this experiment include ones that should require no explanation – Haskell, Ada, Ada9X, C++, and Awk – as well as ones that were developed as part of the Arpa ProtoTech initiative:

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Figure 3: Summary of Prototype Software Development Metrics

1. Rapide, a language developed at Stanford, uses a partial-ordering-on-events semantics and a high-level-module structure, and is targeted primarily for simulation and software architecture modeling [LVB+92].
2. Griffin, a language designed at NYU, can be seen as somewhat of a blend of Ada, SETL, and ML. It is targeted for the same kinds of applications as Ada [HHLH92].
3. Proteus, a language designed jointly by Duke and the University of North Carolina, is a parallel programming language with high-level, machine-independent notions of various kinds of parallelism [MRNP92].
4. Relational Lisp, a language developed at ISI, is essentially Lisp enhanced with notions of "relational abstraction:" a database-like facility having a logic-programming feel.

5 Results

The table shown in Figure 3 is a concise summary of the development metrics associated with each of the prototypes. This table is somewhat different from that in [LBK+94], and in any case should not be used alone to infer any conclusions about the experiment, especially because some of the prototypes did not actually execute. The following paragraphs describe important information that must be factored in when interpreting the results. The numbered paragraphs below correspond to the numbered languages in the table.

(1) The Haskell prototype was written by the second author of this paper, with minor help from the first; neither attended the kickoff meeting at NSWC. It is interesting to note that of the 85 lines of code, 20 were for encoding of the input data, and 29 were either type synonyms or type signatures (and thus could have been elided, with the compiler inferring the necessary type information). Thus only 36 lines of "dynamic" code were required to generate a fully functional geo-server prototype. The code and generated output were reviewed independently by personnel at Intermetrics, Inc., who "concluded that it was an accurate, extensible, understandable, and compact solution to the NSWC requirements for a prototype GEO Server" [CHJ93]. The very high documentation-to-code ratio is also worth noting, which is at least partially due to using the "literate programming style;" the Haskell solution is actually an executable LaTeXdocument.

(2) The Ada solution was written by a lead programmer at NSWC; in this sense it represents the "control" group. The developer initially reported a line count of only 249; this was the number of lines of *imperative* statements as reported by the Sun Ada compiler, and did not include declarations, all of which were essential for proper execution. The line count of 767 is based on the actual code in [LBK+94], and does not include lines with only termination characters.

(3) The Ada9X solution was written by an independent consultant hired by Intermetrics, Inc. The consultant was given not only the problem specification, but the Haskell solution as well. Although the code was never run (at the time no suitable Ada9X compilers were available), it was subjected to a design review.

(4) The C++ solution was written by an ONR program manager *after* having first written the Awk solution described in the next paragraph. In addition to these 1105 lines of code, the developer also wrote a 595-line "test harness." No development times were reported.

(5) In an attempt to determine the utility of "shell" languages for prototyping, a program manager at ONR wrote a geo-server prototype in Awk; the 250 lines of code indicate some degree of success. The conciseness of expression and pre-defined string and file handling capabilities are the points in favor of Awk. It is interesting to note that the developer originally reported only 101 lines of code, but fully admitted attempting to pack as many statements into each 80-column row as possible! The line count of 250 is based on looking at the actual code in [LBK+94] and estimating a line count based on a more "reasonable" notion of program formatting. No development times were reported.

(6) The Rapide solution was written by several researchers at Stanford and TRW, and is perhaps the most difficult to assess: the prototype was not only not executed (at the time no compiler or interpreter was available), but also did not address the essential functionality of the geo-server. The developers chose instead to address software architecture issues, emphasizing Rapide's utility in that domain. For purposes of comparison, it may be best to ignore this entry, but we did not want to eliminate it from the results, since it is included in [LBK+94].

(7) The Griffin solution was written by two members of the NYU development team. It shares the same difficulty of comparison as Rapide, in that the code was never executed; it did, however, undergo a design review, and was intended to satisfy the problem specification. In addition to the 251 lines of code for the prototype, an additional 200 lines of "library code" were written (which should perhaps be viewed analogously to Haskell's Standard Prelude).

(8) The Proteus solution is perhaps the most "functional" of all solutions other than the Haskell solution; effective use is made of Proteus' list-comprehension-like constructs for expressing parallelism. The code was executed by an interpreter. The 293 lines of code does not include a file containing the input data.

(9) The notable aspect of the Relational Lisp solution is the very low development time. The documentation size is also quite low, which may partially account for this. The developer also spent 4 hours in the kickoff meeting at NSWC.

(10) Intermetrics, independently and without the knowledge of NSWC or Yale University, conducted an experiment of its own: the Haskell Report was given to a newly hired college graduate, who was then given 8 days to learn Haskell. This new hire received no formal training, but was allowed to ask an experienced Haskell programmer questions as issues came up during the self study. After this training period, the new hire was handed the geo-server specification and asked to write a prototype in Haskell. The resulting metrics shown in row 10 of the table are perhaps the most stunning of the lot, suggesting the ease with which Haskell may be learned and effectively utilized.

6 Why Did Haskell Perform So Well?

As mentioned earlier, experiments of this sort are difficult to design and conduct; they are even more difficult to analyze! The data presented in the last section is an attempt to objectively make some comparisons between the various prototypes, but in the end the most value is often gained by looking carefully at the code itself. We can then try to answer questions such as: Why was the Haskell solution so much more concise than the others? Which of the prototypes was more readable? Which would be easiest to maintain? And which would be easiest to extend with added functionality?

Based mostly on the presentations given at NSWC, the review panel did try to answer some of these more subjective questions, including questions about the programming languages themselves. These results are summarized in the table in Figure 4. Although "grade compression" factors probably prevent any declaration of a "clear winner," Haskell did fair better than any of the other languages in this analysis. In the following sections we add our own commentary.

6.1 Conciseness

We believe that the Haskell prototype was most concise for three reasons: (1) Haskell's simple syntax, (2) the use of higher-order functions, and (3) the use of standard list-manipulating primitives in the standard prelude. The syntax issue is a subjective one, but we note, for example, that many of the languages used in this experiment have relatively heavy-weight "begin...end" constructions, whereas Haskell uses a convenient "layout" rule for block structuring.

Evaluation Criterion	Ada	Haskell	Rel. Lisp	Proteus	Rapide	Griffin
Extendability	A-	A	A	A-	B	B
Understandability	A	A+	B	B+	B	B
Appropriateness	A	A	A	A	A	A
Accuracy	A	A	A	A	–	–
Compactness	C	A	B	A	C	B
Modeling Support	C	A	B	A-	B	B
Prototyping Support	C	B-	C	B	–	–
Migration Support	A	B	B	B	–	–
Enhancement Support	A	A	B	B	–	–
High Level Structuring	B+	A	C	C	C	B
Support for Reuse	A	B	B	A-	B	B
Turn-around Time	B-	A	A	A	–	–
Efficient Execution	B-	A	–	B	–	–
Executable Specification	C+	A	A	A	B	B
Burst Speed of Writing	B-	A	A+	A	C	B
Learnability	C	A	B-	B	B-	B
Maturity	A-	A-	B-	C	0	0
Overall	B	A	A-	A		

Figure 4: Subjective Evaluations by Review Panel

The use of higher-order functions is noteworthy. In particular, the key abstraction used in the Haskell prototype was the use of higher-order functions to denote geometric *regions*: the idea is that a region is simply a function that, when applied to a point in space, returns a boolean value indicating whether the point is in the region or not (much like the use of a characteristic function to denote a set). Thus we can define:

```
> type Region  =  Point -> Bool
```

To aid readability, a function was also defined to determine whether a point was is in a particular region:

```
> inRegion       :: Point -> Region -> Bool
> p `inRegion` r = r p
```

This function was strictly unnecessary, and resulted in more lines of code, but was considered to be an elegant form of self-documentation.

Given this representation of regions, it is then desirable to define (1) a set of functions to create primitive regions such as circles and half-planes, and (2) a set of functions that combine simple regions to form other, more complex regions (such as the intersection or union of regions). For example, given suitable definitions of `circle`, `outside`, and `(/\)` with functionalities:

```
> circle   :: Radius -> Region            -- creates a region with given radius
> outside  :: Region -> Region            -- the logical negation of a region
> (/\)     :: Region -> Region -> Region  -- the intersection of two regions
```

we can then define a function to generate an *annulus* (which is used to define an engageability zone):

```
> annulus       :: Radius -> Radius -> Region
> annulus r1 r2 = outside (circle r1) /\ circle r2
```

This simple use of higher-order functions, quite obvious to the experienced functional programmer, was not used in any of the other prototypes, even though some of the languages support higher-order functions. The resulting "region authoring sub-language" is very compact and effective. Higher-order functions are used elsewhere in the prototype as well; for example, the overall program is a simple composition of functions performing various sub-tasks.

The third reason for compactness – the use of standard prelude functions – is most apparent in the "situation assessment" and "report generation" tasks, where various list manipulating functions such as append, zip, etc., and string manipulating functions such as the overloaded `show` operator, are used extensively. These functions also take advantage of the concise notation afforded by list comprehensions, which are used to effect the overall "control."

6.2 Documentation

It is worth noting the high percentage of documentation relative to program code in the Haskell prototypes, even in the trainee's solution; the first Haskell prototype had the highest documentation-to-code ratio of all solutions. The largest amount of time in the creation of this Haskell prototype was spent on documentation, not coding. We view this as an advantage of the literate style of programming, which becomes more like writing a paper than writing a program. Subjectively speaking, we feel that the Haskell prototype could best serve as an executable specification for future systems development.

6.3 Extensibility

There were actually three Haskell prototypes developed, in ascending order of generality and functionality, based on the original problem specification, the amended one, and several enhancements implied by the amended specification. We wrote these with no particular goal in mind other than to convince ourselves that the enhancements were possible. In retrospect, however, they constitute an excellent study of maintainability and evolvability; the modularity and highly abstract nature of the original prototype scaled nicely, with a large amount of code reuse resulting from the evolution. Reading the code and accompanying documentation is the best way to get a feel for this.

6.4 Formal Methods

Although not required by the geo-server specification, we also embarked on a tiny bit of formal methods work. It is an almost trivial matter to use equational reasoning to prove properties such as the additive nature of region translation, distributive and commutative properties of region intersection and union, etc.

7 Lessons Learned

Haskell appeared to do quite well in the NSWC experiment; even better than we had anticipated! The reaction from the other participants, however, in particular those not familiar with the advantages of functional programming, was somewhat surprising, and is worth some discussion. There were two kinds of responses:

In conducting the independent design review at Intermetrics, there was a significance sense of disbelief. We quote from [CHJ93]: "It is significant that Mr. Domanski, Mr. Banowetz and Dr. Brosgol were all surprised and suspicious when we told them that Haskell prototype P1 (see appendix B) is a complete tested executable program. We provided them with a copy of P1 without explaining that it was a program, and based on preconceptions from their past experience, they had studied P1 under the assumption that it was a mixture of requirements specification and top level design. They were convinced it was incomplete because it did not address issues such as data structure design and execution order."

The other kind of response had more to do with the "cleverness" of the solution: it is safe to say that some observers have simply discounted the results because in their minds the use of higher-order functions to capture regions was just a trick that would probably not be useful in other contexts. One observer described the solution as "cute but not extensible" (para-phrasing); this comment slipped its way into an initial draft of the final report, which described the Haskell prototype as being "too cute for its own good" (the phrase was later removed after objection by the first author of this paper).

We mention these responses because they must be anticipated in the future. If functional languages are to become more widely used, various sociological and psychological barriers must be overcome. As a community we should be aware of these barriers and realize that they will not disappear overnight.

8 Conclusions

There is an emerging opinion in the software engineering community that the use of prototyping experiments to reduce risk as early in a project as possible actually reduces cost while improving quality. The paradoxical conclusion that you can lower costs by planning to throw some code away is consistent with numerous studies of software engineering productivity. Defects cost much more to remove late in a process than early, and design problems are much more costly to correct than coding errors. Furthermore, productivity declines exponentially as project size increases, so, for example, five five person projects will produce much more useful code than one twenty-five person project. By using prototyping to finalize subsystem interfaces early and to determine experimentally that the interfaces are well designed, one can substantially reduce the overhead of human communication on a large project. In effect, the architecture and interfaces partition the large project into several smaller ones, thereby increasing productivity substantially.

The NSWC experiment is unique in its structure, and noteworthy for functional programmers in its results. We feel that the results clearly demonstrate the value of functional languages in prototyping, and suggest that this is an area where the functional programming community can have a large and effective impact. Of course, the experiment is also lacking in many ways, and we can only hope that future experiments will remedy these deficiencies.

References

[Car93] J. Caruso. Prototyping demonstration problem for the prototech hiper-d joint prototyping demonstration project. CCB Report 0.2, Naval Surface Warfare Center, August 1993. Last modified October 27, 1993; further changes specified by J. Caruso are described in "Addendum to Prototyping Demonstration Problem for the Prototech HiPer-D Joint Prototyping Demonstration Project," November 9, 1993.

[CHJ93] W.E. Carlson, P. Hudak, and M.P. Jones. An experiment using Haskell to prototype "geometric region servers" for navy command and control. Research Report 1031, Department of Computer Science, Yale University, November 1993.

[Hen86] P. Henderson. Functional programming, formal spepcification, and rapid prototyping. *IEEE Transactions on SW Engineering*, SE-12(2):241–250, 1986.

[HHLH92] M.C. Harrison, C-H. Hsieh, C. Laufer, and F. Henglein. Polymorphism and type abstraction in the Griffin prototyping language. In *Proceedings of Software Technology Conference*, pages 458–470. DARPA, 1992.

[HPJWe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[LBK$^+$94] J.A.N. Lee, B. Blum, P. Kanellakis, H. Crisp, and J.A. Caruso. ProtoTech HiPer-D Joint Prototyping Demonstration Project, February 1994. Unpublished; 400 pages.

[LVB$^+$92] D. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz. Partial orderings of event sets and their application to prototyping concurrent timed systems. In *Proceedings of Software Technology Conference*, pages 443–457. DARPA, 1992.

[MRNP92] P. Mills, J.H. Reif, L.S. Nyland, and J.F. Prins. Prototyping high-performance parallel computing applications in Proteus. In *Proceedings of Software Technology Conference*, pages 433–442. DARPA, 1992.