Algorithm Prototyping and the
Nickle Prototyping Environment

Bart Massey
Computer Science Department
Portland State University
P.O. Box 751 M.S. CMPS
Portland, Oregon USA 97207-0751
bart@cs.pdx.edu

Abstract

Algorithms are rarely produced by a process of pure mathematical reasoning. In practice, the specification and design of new algorithms is almost always accompanied by some sort of Algorithm Prototyping (AP) process, in which alternatives are explored and evaluated. The discardable algorithm prototype is then used as the basis of product implementation, and often as an effective oracle to test that implementation. The AP process has long been a feature of real-world software development, distinct from prototyping activities such as user-interface prototyping and rapid application development that have received more attention in the software engineering community.

The Nickle language-based prototyping environment is a long term project having as one of its principal goals support for easy and effective AP. Nickle has been used to prototype a wide range of numeric and semi-numeric algorithms, with good success.

In this paper, the prerequisites for AP are explored. The Nickle prototyping environment is evaluated as a tool for AP, with several examples of its use in this mode. Finally, suggestions are made as to the direction of future improvements in this area.

1. Introduction

Academic software developers with commercial experience know that academic and commercial software development are often handled quite differently. In general, commercial development tends to be more structured, more rigidly controlled, and to be less ambitious in the use of advanced or complex algorithms and designs.

To some extent, these differences are the result of differing types of software requirements. In academia, functional requirements are focused more on provable correctness, and non-functional requirements on worst-case complexity analysis. In the commercial world: robust correctness and good absolute performance across the full range of cases that could be reasonably encountered in use plays a larger role.

On the other hand, modern software development models such as Extreme Programming [2] adopt some practices that are familiar to academic developers. Design and implementation is interleaved with specification, and requirements elicitation and specification is guided by this fine-grained feedback. These practices have been adopted for good reason: they in essence constitute a Spiral Model [3] of software development, minimizing development risk and increasing software quality.

Software engineers have long understood the important role that software prototyping can play in areas such as requirements gathering, user interface design and system design. In this paper, it is argued that Algorithm Prototyping (AP) is also an important part of many software development projects. Section 2 distinguishes AP from other kinds of prototyping in a number of significant ways. Section 3 identifies some of the requirements for effective AP. Sections 4 and 5 introduce the Nickle prototyping environment as a vehicle facilitating AP and describe some experiences with AP in the Nickle setting. Sections 6 and 7 discuss related and future work. Finally, Section 8 summarizes the work and draws some conclusions.

2. Algorithm Prototyping

When simple or well-understood algorithms are deployed in a software product, the engineering techniques that engender success are relatively well understood. For algorithmically simple tasks, standard software process models suggest a number of techniques for scheduling software development and evaluating the resulting implementation.
For more advanced pre-existing algorithms, the availability of detailed specifications, pseudocode, validated test cases, and other fielded implementations can be extremely helpful in both the implementation and testing phase.

The task of developing a new algorithm for deployment in a software product requires different software engineering perspectives. While academic study of algorithms, data structures and complexity provides a great deal of insight into higher-level algorithm design, it is notoriously difficult to evaluate algorithmic ideas without a straightforward working implementation.

It is tempting to believe that algorithms are developed in the fashion that they are presented in textbooks and scholarly papers: an algorithm meeting requirements is designed on paper, proved correct, and subjected to complexity and performance analyses. This neat package is then ready for deployment in a software product.

By contrast, in practice algorithm design has a highly empirical component. Algorithmic ideas are implemented and evaluated on benchmark instances. The algorithm that seems to best meet correctness and efficiency requirements is selected. Further testing ensues, interleaved with theoretical correctness analyses. The newly-designed algorithm continues to be “tuned” in various ways, such as correctness fixes for corner cases, selection of sub-algorithms with acceptable performance, and adjustment of heuristic parameters. Only when the developer is comfortable with an algorithm and its sample implementation does it become a candidate for software product deployment.

This, then, is the heart of AP: a discardable prototype is constructed that is used as a testbed for algorithm development. This prototype generally consists of the minimal subprogram that can execute the algorithm in question. The interface to the prototype is normally that provided by the language and environment being used. Special purpose I/O code, if necessary, is kept as simple as possible. The resulting prototype provides critical information to the algorithm designer.

The AP process tends to involve a wide variety of standard development tools, including programming languages such as C and C++ and scripting languages such as UNIX shell scripting and Perl code. In addition, more esoteric and less directed tools are often employed: symbolic math packages such as Maple and Mathematica, numerical environments such as Matlab, macro preprocessors such as CPP and M4, and even spreadsheets and simple calculators. A successful algorithm designer often has the ability to prototype using all of these tools and more, selecting among them as the situation warrants. Thus, successful algorithm development depends to some degree on the mastery of a wide range of mundane and arcane skills.

Domain-specific techniques and tools can also be important for AP. For example, discrete event simulators are very useful in AP for parallel processing; statistical tools are a great aid in analysis of complex systems. In addition, these sorts of tools are very useful in the generation of instance data for algorithm evaluation.

It is instructive to contrast AP with several other kinds of prototyping in common use in software engineering.

**System prototyping** is used as an aid in the construction and evaluation of system architectures. The emphasis tends to be on high-level blocks and data flows. Often, some or all system components are stubbed out. Thus, algorithms are rarely the focus here.

**Rapid Application Development** is a close cousin of system prototyping. In this software development model, a reusable prototype is constructed: the normal intent is that the prototype be completed and elaborated into the finished system. Applications built using this model generally involve little algorithm development: the goal of the process is normally quick development, and at any rate reusable prototypes can be more difficult than discardable prototypes to instrument and adapt.

**User Interface prototyping** is quite common in modern software practice, and is helpful in eliciting and specifying requirements, in documenting UI design, and often consists of code intended for reuse in the actual product. Modern development environments often include support for these activities, typically in the form of a “GUI builder” tool. Unfortunately, these tools and techniques are of little assistance in algorithm development. In fact, most GUI builders are designed to build interfaces that are deliberately independent of the underlying algorithmic code.

### 3. Prerequisites For Effective AP

The description of effective AP in section 2 suggests a number of desirable prerequisites. These prerequisites can be divided into several broad areas: form of the problem description, features of the prototyping language, features of the prototyping environment, and facilitation of algorithm implementation. Section 4 describes prototyping features in the context of the Nickle prototyping environment.

AP tends to need a slightly different style of problem specification than most kinds of software development. In particular, it is desirable to have an instance description [10] characterizing the problem instances on which the algorithm will operate. Like any specification, the instance description needs to be clear and detailed; unlike most specifications, an instance description should be expressed in a formal notation, and should be formally sound and complete. Unfortunately, most real-world problem instances are too complicated to easily describe at this level of formality. Thus, it is normal to *abstract* the problem, identi-
ifying key instance features that are expected to impact algorithm development. For example, it is common in describing scheduling problem instances to abstract away real-valued times in favor of fine-grained but discrete intervals. This dodges questions of floating-point equality, and more importantly permits the consideration of algorithms that depend on this discrete interval structure.

Criteria for a good AP environment are to some extent subjective. However, some general conclusions can be drawn as to the sorts of features that support the AP style described previously. In particular, easy interaction appears to be an essential part of the AP process. The algorithm designer should be able to modify the prototype with minimal effort. Debugging and tracing the execution of the algorithm should be easy. “Batch mode” interactions are unacceptable in this setting. While graphics and GUI support are not necessary in the AP environment itself, it is important to be able to input and output data in ways that are intuitive to the user and amenable to scripting: external tools such as spreadsheets or plotting programs are commonly used for data visualization. In short, AP is an experimental process, and a good AP environment will support an experimental style.

Crucial to the success of AP in an SE setting is the ability to carry the prototyped algorithm into production code with minimal effort. It is true that the algorithm prototype is normally disposable. However, if the semantic distance between the prototyping and production language or environment is too large, it may be difficult to transport even a detailed understanding of the algorithm to the production setting.

Another important factor is the ability to use the prototype as an effective oracle for test case generation. Just as running the prototype on test and benchmark instances is a crucial part of the prototyping process, running the prototype on test inputs can be a tremendous aid to test case generation. This implies that the prototype should exhibit reasonable performance, and should be amenable to scripting or some other type of automated data entry.

The next section describes Nickle [22], a language and environment for AP that meets many of these goals. Section 5 outline some positive experiences using Nickle in AP. It is important to emphasize that this is just one possible execution of the AP concept, and that there are many environments that can enable effective AP. For example, many algorithm designers have had success with AP in the Smalltalk-80 and Squeak environments. However, Nickle was designed to be well-suited to the AP task, and appears to be particularly successful at it.

4. Nickle: An AP Environment

The design and implementation of programming environments is always a matter of taste, if not religion. The design space for is large, and the engineering tradeoffs can be difficult. One classic technique for simplifying the design is to identify a set of use cases corresponding to a target domain. Tradeoffs are then made to favor that domain at the expense of others.

Nickle has its roots in ec: a C-like programming language developed about 15 years ago by Keith Packard in response to his need to prototype numeric algorithms. ec was a compiler that converted arbitrary precision integer and rational arithmetic written in a C-like style to byte code for an interpreter. ec became the basis of ic: a fully interpreted desk calculator and simple programming language featuring arbitrary-precision integers and rationals and C-like syntax.

Further development of ic over a 10-year period eventually produced the Nickle programming language and prototyping environment. Throughout this period, the principal emphasis in design and implementation was on algorithm prototyping, numeric scripting, and calculation and prototyping interface. This set of design choices produced a system that combines familiar language and system ideas with unique features to achieve high flexibility and broad applicability.

4.1. AP-Friendly Features

To understand the design tradeoffs in Nickle, it is useful to consider programming language and prototyping environment features that encourage AP.

1. A good AP programming language will be familiar, in a couple of important senses. First, it will support a style that is familiar to the algorithm developers. While Smalltalk, APL, Prolog and Haskell are all good languages for AP, they are all limited to some extent by the difficulty of programmers adapting to their unusual programming models. In addition, algorithms prototyped in these languages are difficult to transfer to production environments that still tend to be built around C, C++ and Java.

Nickle’s programming language is a C-like language. Part of its design philosophy is to never be gratuitously incompatible with C: the syntax and semantics of the language and its libraries should be as close to C’s as possible while still achieving the desired improvements for AP applications. (Additional functionality is partly original, but more often borrowed from a host of languages, notably Java and ML [1].) For example, Nickle’s value initializers look much like C’s, but features such as arbitrary initialization expressions, implicitly uninitialized storage, array fills with an arbitrary value, array comprehensions, and tagged structure initializers greatly increase the usability of initializers without greatly compromising the syntax and
3. A good AP environment should be readily available on a comfortable platform and amenable to change. It is extremely convenient to be able to identify a desired prototyping environment feature, design an implementation, and add it to the existing system as needed.

The Nickle implementation is freely available for UNIX platforms under an Open Source license. The system undergoes constant, though largely backward-compatible, change. This change typically occurs in response to a noted deficiency that can be readily remedied. The implementation currently consists of almost 30,000 lines of C, FLEX, and YACC, and a few thousand lines of Nickle code. The use of a FLEX lexer and YACC parser has greatly eased the process of syntactic modification. The Nickle garbage collector also manages memory for the C portion of the implementation, greatly reducing the chance of difficult-to-debug memory usage errors. The ability to easily provide new functionality in the form of Nickle code has allowed much more compact and readable coding; for example, the floating point library is implemented entirely in Nickle.

4. A good AP environment should have an honest implementation of truly mathematical numbers. Integer overflow and floating point underflow and inaccuracy are common problems plaguing production implementations. It is important to be able to defer dealing with these issues until the underlying algorithm is otherwise sound. Pseudocode usually assumes unbounded integers and accurate rationals, as do formal specification notations such as Z [28]. Irrational numbers are more of a problem; the obvious alternative is to provide floating point as a fallback.

Nickle represents a number internally in one of three forms: as an unbounded integer, a reduced-form rational with unbounded numerator and denominator, or as a floating point number with unbounded exponent and user-specifiable mantissa precision. (The default mantissa precision, 256 bits, is more than adequate for most applications.) For many algorithmic domains (e.g. probabilistic calculations) arbitrary-precision rationals give exact answers in an efficient fashion. These features make Nickle extremely useful for prototyping numerical and semi-numerical algorithms.

Other popular language-based interactive environments that to some extent meet the above criteria include Scheme [7] and ML. These programming languages are syntactically farther from C than Nickle, but not unusually so. While they promote a largely functional programming style, they adequately support a traditional imperative model. Both have seen implementations supporting arbitrary-precision integers: some Scheme implementations (e.g. [5]) also support arbitrary-precision rationals. The author has successfully used each of these environments for AP in the past, and can attest to their utility.

4.2. Non-requirements For AP

In addition to the desirable feature list for AP, there are some common features that are less important to the AP process. For example, cross-platform support is not particularly critical in this environment. An algorithm prototype is typically discardable code in a different language than the production target. It thus suffices to implement the developed algorithm using a cross-platform language.

Similarly, while it is important that the AP system be efficient enough to permit reasonable experimentation and test-case generation, it is not necessary that it meet the performance time and memory requirements of the target software product. Typically, algorithm design efficiencies are considered largely in terms of complexity-theoretic order
bool[*] make_sieve(int size) {
    bool[size] sieve = {true ...};
    for (int k = 2; k < size; k++) {
        if (!sieve[k])
            continue;
        for (int i = 2 * k; i < size; i += k)
            sieve[i] = false;
    }
    return sieve;
}

Figure 1. Sieve of Eratosthenes In Nickle

Java, the for loop index is properly scoped. As in Java, a Boolean data type is available, and is useful in detecting some of the more common C typographical errors (particularly mistyping ‘==' as ‘=’). A bulk initializer is used to set the entire sieve array true. The sieve array is a first class object, returned from the function. The work to port this function to C would be minimal.

Because Nickle is interactively byte compiled, its environment combines features of an interpreted programming language and of a desk calculator. Figure 2 shows a brief transcript of Nickle execution illustrating some of the natural flavor of the Nickle AP environment. (Nickle has a primary prompt of ‘>’.) In the transcript, the code of figure 1 is loaded. Then a small sieve array is made. A sample value is checked, and the size of the array is examined. The sieve array can now be used as a parameter to other programs, etc. As can be seen from even this simple example, the Nickle design and implementation has features intended to encourage effective AP experimentation. Space limitations preclude a full discussion of Nickle programming language features: for a fuller exegesis, see [21].

5. Experiences With AP In Nickle

This section reports several experiences using Nickle for AP. The emphasis here is on the entire process of moving from an algorithmic idea to a robust and efficient product implementation. Several such applications are reported, involving the domains of number theoretic algorithms, combinatorial search in cryptography, and 2D graphic rendering. While most AP experience with Nickle to date has been by the system developers, one of the reported applications is largely the work of a third party; more outside experience will be gained as the system and techniques are more widely adopted.

5.1. AP With Prime Numbers

The RSA public key cryptographic algorithm [27] is a commercially important algorithm involving modular multiplication of large primes. A C implementation typically requires a couple of weeks for a competent graduate student. A Nickle prototype of the RSA algorithm is shown

<load "erat.5c">
<s = make_sieve(10)
[10] {true, true, true, true, false, true, false, true, false, false}
<s[7]
true
<dim(s)
10
>
namespace RSA {
import Numbers;
global int e, n;
global int d = 0;
public int function encrypt(int m) {
    return bigpowmod(m, e, n);
}
public int function decrypt(int c) {
    if (d == 0)
        abort();
    return bigpowmod(c, d, n);
}
public void function set_private_key(int p, int q, int e0) {
    int phi = (p - 1) * (q - 1);
    n = p * q;
    if (e0 % 2 == 0)
        e0++;
    while (gcd(e0, phi) > 1)
        e0 += 2;
    e = e0;
    d = zminv(e, phi);
}
public void function set_public_key(int n0, int e0) {
    n = n0;
    e = e0;
    d = 0;
}
}

Figure 3. RSA In Nickle

in its entirety (excepting comments) in figure 3. Note that ordinary arithmetic expressions are used in all cases; arbitrary precision arithmetic greatly simplifies this implementation. The references to bigpowmod (efficient modular exponentiation) and zminv (multiplicative inverse in the modular integers) call previously prototyped algorithms for these functions. The overall structure of the algorithm is well-described by the code, and interactive experimentation with the prototype is straightforward.

A key component of an effective RSA implementation is a generator of large random prime numbers. The algorithm normally used for this is due to Miller and Rabin [25], and establishes the probable primality of a candidate number to any desired degree of precision in polynomial time. This algorithm is also straightforward to prototype in Nickle. The Nickle implementation is efficient enough that the prototype can generate a 1024-bit random prime with high confidence (failure probability less than $2^{-64}$) in a few seconds on modern hardware. The cryptographically secure pseudorandom number generator used for prime candidate selection is also implemented in Nickle; it was originally prototyped to explore the use of the RC4 stream cipher as a pseudo-random number generator.

It was recently shown that deterministic (as opposed to probabilistic) primality testing can also be performed in polynomial time. While the algorithm is not currently efficient enough to be practical, it is nonetheless interesting. Within a few days of the original announcement, the author had a prototype Nickle implementation of the algorithm; the first public implementation, as far as he is aware. This prototype made heavy use of the same numeric machinery as the previous examples. It also required an implementation of modular polynomial arithmetic, prototyped and tested separately.

(All of the code described in this section is available in the examples/ directory of the Nickle distribution. To download this distribution, see section 10.)

5.2. AP In A Combinatorial Search Application

The winning entry in the 2001 Intel Northwest Science Exposition involved work on cryptanalysis of substitution ciphers [18] using a statistical technique due to Ganesan and Sherman [9] to guide a greedy local search in the key space. While the work was similar to that of Jakobsen [17], both the search technique and the statistical method of the work were somewhat improved. Nonetheless, the C++ implementation was distressingly slow. A more sophisticated search technique was needed to improve performance.

Ganesan and Sherman describe a simple metric for using bigram (letter pair) statistics to estimate the likelihood that the text of a particular document is in the same language as a reference corpus. Using this metric, an automatic decryption algorithm for substitution ciphers was implemented by Bart Massey and Keith Packard in about 700 lines of Nickle code. The solution algorithm compares the distribution of bigrams in the input document with that of the reference corpus. A random substitution key is generated. Each pair of entries in the key is repeatedly tested to see what the change in the overall score would be if the cipher text were decrypted with a key where these two entries are exchanged. At each iteration, the algorithm exchanges the pair that improves the score by the largest amount. This test and swap sequence is repeated until no swaps are found that improve the score. The result is a potential decryption key.

A set of ADTs were constructed for managing the bigrams and encryption functionality. Within this infrastructure, a search algorithm was developed to select the best decryption key. Over a period of two days, several major improvements in the efficiency of the search mechanism were prototyped and tested using Nickle. Each change was regression tested using the existing Nickle implementation to verify correctness. Errors were caught with assertions that invoked the Nickle debugger. This allowed the state of the program to be inspected using Nickle itself; a far more powerful mechanism than an external debugger, especially
as the internal state of the cipher solver was difficult to interpret by examining program-visible variables.

Execution performance was improved from a runtime of six hours to a runtime of 15 minutes through this process, both through identifying values recomputed several times through the execution, and through pruning computations that could be quickly identified as unhelpful.

Once the algorithm was finalized, it was translated to C. This was a largely mechanical process taking approximately 15 minutes to convert the Nickle code to approximately 500 lines of C code. This process was aided by the fact that some Nickle computations, principally the bigram statistics for the underlying corpus, were not performed by the C implementations but were simply read in. Notably, the C version exhibited no obvious algorithmic defects and only one significant error of any sort: this error being due to a trivial and easily corrected mis-translation.

The C version was faster, taking approximately 3 seconds to generate a potential decryption key. With this additional performance, a random restart was added to the algorithm to improve the results. The resulting decryption tool accepted text encrypted with an arbitrary substitution cipher and generated a readable result in approximately 15 seconds on a modern machine. Figure 4 shows an example decryption using a ciphertext [12] and underlying corpus [11] drawn from the Tom Swift novels. (Uppercase and lowercase were folded and duplicate whitespace was removed before processing of both the reference corpus and the plaintext.) In (a), the decryption is not yet complete, but some obvious progress has been made in the search. When completed (b), the only obvious decryption error is the interchange of the period and comma; a larger ciphertext or manual intervention can easily correct this problem.

Figure 4. A Ciphertext Decryption

b) Final Decryption

5.3. AP For 2D Graphics

Two-dimensional (2D) computer graphics is the workhorse of modern personal computing. However, there has been only slow progress in 2D graphics algorithm development over the last few years. When the need arose to modernize rendering in the X Window System [26], it became apparent that some novel 2D graphics algorithms would be necessary. Algorithm prototypes developed in Nickle were a key factor in the success of this endeavor.

5.3.1. Rendering Trapezoids

In early 2001, algorithms were developed for trapezoid rendering in the Render Extension [23] to the X Window System. The specification employed Nickle AP in a number of modes with great success, and the success of the resulting C implementation was a direct result of experience with the Nickle prototype.

The X Render Extension provides a new rendering model for the X Window System. A significant part of the extension is a new design for anti-aliased rendering of geometric objects represented as a collection of trapezoids, each defined by four line segments. Anti-aliasing is approximated by computing the area of each pixel covered by the polygon and using this value to perform image compositing. Like the core X protocol specification, the Render extension specification gives exact pixelization rules—both for ease in verifying an implementation and, more importantly, to ensure that multiple implementations used within the same application can generate matching results.

To solidify the specification and explore implementations a sample implementation of the proposed trapezoid rasterizer was developed using Nickle. Nickle’s rational datatype made this implementation straightforward, taking about 20 minutes to write the 75 lines of code. With a demonstration of correctly rendered trapezoids in hand, design of the C version was begun. The first step was to switch from rational arithmetic to integer arithmetic. A new Nickle implementation was begun in order to prototype the algorithm.

As C provides only fixed-precision integers, analysis of the algorithm was done during the integer Nickle implementation to compute the size of all intermediate results. This uncovered a fundamental flaw in the specification. Rasterizing “self-intersecting” trapezoids (as seen in Figure 5) involved the computation of the intersection point, requiring 192-bit integer arithmetic. The high cost of such a computation would undermine the utility of trapezoid-based rendering.

Realizing that the original specification was only an approximation of anti-aliasing in any case, the correct solution in this case was not to discover a clever way of implement-
Several alternative specifications for polygons were proposed and prototyped in Nickle to examine their behavior. Of these, some were discarded because of undesirable rendering artifacts and some were shown to have computational problems similar to the original specification. A simple Nickle rasterizer was used to render test outputs of the algorithm; this level of visualization was sufficient for effective development.

The final algorithm involves integers no larger than 64 bits, avoids rendering artifacts when several polygons are combined to rasterize a complex shape, and meets a specification that differs little from the original. This trapezoid rendering algorithm has now been implemented and fielded as part of the XFree86 server. Several demonstration applications have been constructed, and have performed admirably, as expected.

5.3.2. Rendering Splines

With the server-side X implementation of trapezoids completed, work began on a client-side implementation of splines rendered using these trapezoids. An arbitrarily precise algorithm was developed for computing the outline formed by stroking a path made of cubic Bezier splines with a circular pen. As before, AP was performed in Nickle, leading to a C production implementation. Most of this work was performed by a developer who was not seriously involved in the Nickle implementation; this provided a good test of the effectiveness of Nickle AP by a relative outsider.

The Nickle prototype turned out to be critical to the development of a successful algorithm. Eventually, the problem specification was revised several times during the exploration of various candidate algorithms. The developer reported that “Nickle features such as arbitrary precision arithmetic, automatic garbage collection, and fully nested functions allowed us to code quickly and to focus on the ideas and algorithms rather than the code,” [32] just as was hoped.

The final Nickle prototype consisted of about 700 lines of code. This prototype was carefully inspected, and then used as the basis for a largely direct port to C. The removal of Nickle features used in the prototype was largely mechanical: for example, all nested functions were extracted from their enclosing functions, and the corresponding argument lists were augmented as necessary. The most serious defects introduced during the porting process were the result of missing C functionality, such as an integer overflow during a multiplication, and troubles with a circular buffer related to the incomplete specification of the C modulus operator.

To quote the developer once again, “In the end the C program was completed very quickly. There is no doubt in my mind that the same program would have taken much longer to write, and would not be as elegant if Nickle had not been used during its development.”

6. Related Work

AP is an activity as old as computer science. A wide variety of tools and environments have been constructed to ease the process; in this section, we touch on a few of the most important examples.

In terms of programming language environments, Scheme and ML were discussed in section 3. Smalltalk-80 [13] and APL [16] are also reasonably attuned to the AP style. Although both are intended as final implementation languages in their own right, both have also been used quite successfully as AP languages for projects implemented in other languages. The same can be said to some extent of Common Lisp [29].

The Icon programming language and environment [14] supports a large number of features intended to ease algorithm development. Its imperative structure combined with implicit iterators leads to a pseudocode-like programming style. However, it can be tricky to master the Icon semantic model, and the semantic distance between the Icon language and more traditional imperative languages is not small.

It is currently popular to prototype in Perl. While Perl is a quite suitable language for system prototyping, it tends to be awkward for AP due to the complexity of the language, the difficulty of debugging programs, and the lack of important features like a good numeric model. All in all, Perl is probably better suited for prototype implementations of well-understood algorithms. While Python [20] is a less complex language, it shares many of the other problems of Perl in AP applications.

Symbolic algebra packages such as Maple [6] and Mathematica [31] can be useful in developing certain kinds of
algorithms. The Macsyma [15] symbolic algebra package is the first system the author is aware of to use rationals as a fundamental datatype. The ability of a symbolic algebra package to manipulate equational forms and the like can help in understanding a problem; however, it can equally well lead to development of an algorithm that is hard to deploy in a traditional setting. In general, these tools appear to be most useful when applied as an adjunct to some other AP environment.

Matlab [30] is an algorithm development environment in the spirit of Nickle. The author is most familiar with the Matlab-inspired but freely available GNU Octave [8] variant. While Octave lacks some Nickle conveniences such as nested functions, and some important Nickle datatypes such as arbitrary precision rationals, it has built-in support for complex arithmetic, various kinds of matrix manipulations, and solving differential equations. Octave also has a large library of predefined functions useful for science and engineering applications. The success of many in using Matlab and Octave for AP attests to the power of this approach.

A number of systems have been constructed for domain-specific AP [19, 4, 24]. Many of the current systems of this type are commercial, and have received little academic evaluation. There is quite a bit of leverage to be gained from using domain-specific techniques for AP where these techniques are available; environments such as Nickle tend to be much more usable outside domain boundaries.

7. Future Work

The work reported here is just the beginning of successful study and development of the AP process in the software engineering context. Future work is of two general types: investigation of AP and work on Nickle.

An empirical evaluation of AP processes and tools should be conducted. More information is needed about the role of AP in the software engineering process. In particular, AP should be better studied in industrial research and deployment settings. These settings are likely to differ in important ways from academic settings in which software is not the principal target product. Algorithm development and AP in the Open Source community is of special interest. This development is much more open than in industrial settings, and tends to lead to novel and synergistic designs, as the work of section 5.3 illustrates.

The programming language portion of the Nickle system is far from finished. Parametric or template polymorphism and abstract datatype (ADT) support facilities will eventually be added to the type system. This will allow the incorporation of more powerful structured datatypes such as lists and dictionaries into the language, and enable development of an ADT library.

The current Nickle implementation suffers from a lack of validation and verification effort. A formal specification should be written for the system, and a test suite developed to debug the implementation. At the least, a more effective regression test suite should be developed.

Finally, Nickle needs to be more widely disseminated in the software engineering community. Experience with a wider variety of developers would greatly improve the Nickle designers’ understanding of AP as it is practiced outside their circle. The other side of the coin is that a wider exposure to Nickle might improve the algorithm development capabilities of the software engineering community.

8. Conclusions

Algorithm prototyping has always been a common practice, and an essential aspect of new algorithm development. By allowing the algorithm designer to explore the requirements and design spaces, an algorithm prototype helps the designer be creative in choices and make necessary engineering tradeoffs. By serving as a blueprint for product implementation and as an effective oracle for testing the implementation, AP helps to ensure that the algorithm is successfully integrated into the software development process.

Algorithm design is one of the more challenging aspects of the software development process. Good AP tools ease the burden on the algorithm designer, allowing more consideration of the algorithm and less of the language and environment in which the prototype is being implemented.

The Nickle prototyping environment, the result of 15 years of design and development, features support for AP as a principle design goal. The C-like syntax and clean imperative semantics of the Nickle programming language mean an easy learning curve for most programmers. Nickle’s many unique features in support of AP are integrated in a clean and comprehensible way that allows focus on the algorithm being constructed rather than infrastructural detail. The Nickle implementation can and does accommodate change; it is freely available, and its developers welcome suggestions and bug reports and respond to them promptly.

A number of successful AP projects have been conducted using Nickle; several of them are reported in section 5. These projects validate Nickle as a fit to many desirable properties of an AP environment. In addition, these projects illustrate the important role the AP process has in the development of some kinds of computer software.

Much work remains to be done in documenting, supporting and evaluating the AP process. By better understanding AP, the software engineering community can improve the overall software development process. By using modern AP tools such as the Nickle prototyping environment, the AP process can be greatly eased.
9. Acknowledgements

Robert Bauer, Dick Hamlet, Warren Harrison, Keith Packard and Carl Worth all read drafts of this manuscript and gave detailed and extremely helpful comments. Jeremy Lennert’s cryptographic search application was an important contribution to the work. Carl Worth and Keith Packard donated a great deal of time to documenting their efforts; Keith has also been largely responsible for the Nickle implementation.

10. Availability

The reference UNIX implementation of the Nickle system, together with many examples of its use, is freely available under an Open Source license at http://nickle.org.

References