Implementation of an A+ Interpreter for .NET

Péter Gál and Ákos Kiss
Department of Software Engineering, University of Szeged, Dugonics tér 13., 6720 Szeged, Hungary

Keywords: A+, .NET, Interoperability.

Abstract: In this paper, we introduce a clean room implementation of an A+ interpreter for the .NET platform. Our goal was to allow interoperability between A+ and .NET programs, thus unlocking existing routines to .NET developers and making .NET class libraries available to A+ developers. In a preliminary experiment, we made the advantage of interoperability visible: we achieved a 5-fold speedup by calling .NET methods from an A+ script. Additionally, we have also shown how to use A+ as an embedded domain-specific language in C# code.

1 INTRODUCTION

A+ is an array programming language (Morgan Stanley, 2008) inspired by APL. It was created more than 20 years ago to suite the needs of real-life financial computations. However, even nowadays, many critical applications are used in computationally-intensive business environments. Unfortunately, the original interpreter-based execution environment of A+ is implemented in C and is officially supported on Unix-like operating systems only.

Our goal was to develop a .NET-based implementation, thus allowing the interoperability between A+ and .NET programs (calling .NET methods from A+ scripts, or executing A+ code from or even embedding – as a domain-specific language – into .NET programs), and the hosting of A+ processes on Windows systems. This would extend the lifetime of existing A+ applications, unlock existing financial routines to .NET developers, and make .NET class libraries available to A+ developers.

In this paper, we introduce a clean room implementation of an A+ runtime[1] for the .NET environment utilizing its Dynamic Language Runtime (DLR) (Chiles and Turner, 2009). We give an overview of its architecture, we present preliminary results on its performance, and most importantly, highlight the potential that lies in the interoperability between A+ and .NET code.

The rest of the paper is organized as follows. Section 2 gives a short introduction to the A+ language, focusing on its specialities, Section 3 discusses briefly the implementation of the .NET A+ runtime, Section 4 details the experiments conducted with the runtime, Section 5 overviews related work, and finally, Section 6 concludes the paper and gives directions for futute work.

[1]Project hosted at: https://code.google.com/p/aplusdotnet/.

2 THE A+ PROGRAMMING LANGUAGE

A+ derives from one of the first array programming languages, APL (Clayton et al., 2000). This legacy of A+ is one of the most notable differences compared to more recent programming languages. While the operations in modern widespread programing languages usually work with scalar values, the data objects in A+ are arrays. Even a number or a character is itself an array, of the most elementary kind. This approach allows the transparent generalization of operations even to higher dimensional arrays.

The other striking speciality of A+ is the vast number of (more than 60) built-in functions, usually called operators in other languages. These functions range from simple arithmetic functions to some very complex ones, like the matrix inverse or inner product calculations. Furthermore, most functions have special, non-ASCII symbols associated. This allows quasi-mathematical notations in the program source code, but may cause reading and writing A+ code to be a challenge for the untrained mind.

Finally, although being a language of mathematical origin, A+ has an unusual rule: all functions have equal precedence and every expression is evaluated

```
(+/a) , ×/a ← 1 + ι 10
```

Listing 1: The computation of the sum and product of the first 10 natural numbers in A+.

from right to left.

The above mentioned specialities are exemplified in Listing 1, which shows how the computation of the sum and product of the first 10 natural numbers can be formalized in A+. The expression `ι 10` (using the symbol *iota*) generates a 10-element array containing values from `0` to `9`, while `1 +` increments each element by one, which is then assigned to the variable `a`. The operator `×/` computes the product of the vector, while `+/` computes the sum. The concatenation function is denoted by comma, which finally results the two-element array `55 3628800`. Note the parentheses around the sum, whithout which concatenation would be applied to variable `a` and the product, and summation would be applied only afterwards because of the right-to-left order of evaluation.

As the above code shows, quite complex computations can be easily expressed in a very compact form in A+. The Language Reference gives further examples, mostly from financial applications, e.g., how to compute present value at various interest rates in a single line (Morgan Stanley, 2008, page 62).

3 IMPLEMENTATION OF THE INTERPRETER

3.1 Problems

Even before the implementation of the .NET-based runtime could have started, we have run into two major problems of the original system. First, it turned out that A+ has no formal grammar. There are only two official sources of information available on the syntax of A+: the Language Reference (Morgan Stanley, 2008), which gives only a textual description of the language, and the source code of the reference implementation, which contains a hand-written tokenizer and parser, from which the formal rules are non-trivial to reverse engineer. Thus, we had to formalize the grammar of A+ first. (The resulting grammar in EBNF form is available from the repository of the project.)

The second major problem was that the language reference and the reference implementation conflicted at quite a few points in semantic matters. Some differences were clearly the result of the lazyness of the implementer (e.g., the system command `$mode` determines whether a string equals to “apl”, “ascii” or

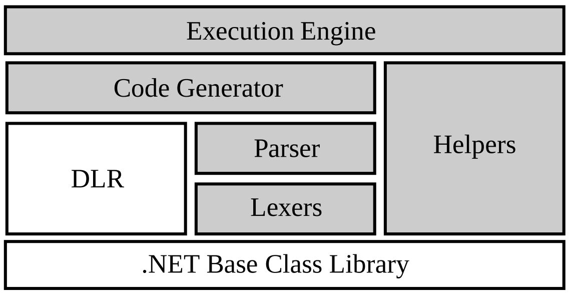

Figure 1: Components of the A+ .NET runtime.

“uni” based on the second character of the string instead of looking for an exact match). Some other differences were, however, the results of extensions to the semantics documented in the language reference (e.g., the implementation of the pick function accepts not only one-dimensional arrays as documented but multi-dimensional ones as well.) In those cases that fell into the first category, we decided not to repeat the same errors but to follow the language reference. However, in the case of the second category, we chose to accept the extensions in the .NET version since existing A+ applications may rely on their existence.

Once we handled the disturbing syntactic and semantic problems of the original system, we became able to work on the adaptation to .NET. The architecture of the resulting system is described in the next subsection.

3.2 Architecture Overview

Figure 1 depicts the architecture of the .NET-based A+ runtime and identifies its main components. The white boxes denote components provided by the .NET framework, including the base class library and the DLR that aids the adaptation of scripting languages to .NET. The shadowed boxes form the system implemented by us. These components build on top of each other as follows.

The lexer and parser modules are automatically generated by ANTLR (Parr, 2007), thanks to the formalization of the A+ grammar. In the current implementation, we have lexer grammars for two input modes of the possible three of A+: one for the APL input mode that is commonly used by the everyday users of the language (but requires a special APL font for proper display) and one for the ASCII input mode, which makes the embedding of A+ scripts into other source code, e.g., into C#, easier. (The system can be extended with the now-missing UNI mode lexer easily if the need arises.)

The output of the generated parser is an abstract syntax tree (AST), which is transformed by the Code

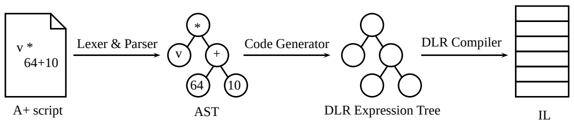

Figure 2: Compilation steps of an A+ script.

Generator module into DLR Expression Trees (ET). During transformation, part of the semantics – especially control structures and the structure of statements – are expressed using ETs, while complex functionalities operating on diverse data structures get usually transformed to calls to helper functions implemented in C#.

The entry point for the execution of an A+ script is the Execution Engine. It glues together the parser, the Code Generator, and the DLR subsystem by feeding the A+ source code into the lexer-parser, giving the resulting AST to the Code Generator, passing the generated ET to the compiler of DLR, and finally, calling the compiled executable IL code. In Figure 2, we show how the different components of the runtime transform an A+ source code until it becomes executable by the .NET framework.

3.3 Usage

The .NET-based A+ execution engine can be used in two ways. Since DLR provides a command line hosting API, it is very easy to implement a command line tool mimicking the behaviour of the reference interpreter implementation. The biggest advantage of the .NET-based implementation is, however, that it can be embedded into other .NET applications. Moreover, by adding .NET methods into the execution scope of the engine, it is possible to achieve interoperation between A+ scripts and the embedding .NET environment.

4 EXPERIMENTS

Although our primary goal for the initial implementation of the .NET-based runtime was to make its observable behaviour as equivalent to the reference implementation as possible and, thus, we did not focus especially on optimizations, we still wanted to get preliminary results on its runtime performance. Thus, we extracted a code fragment from a real-life code base and extended it with some code performing execution time measurement. Listing 2 shows the test A+ script, where lines 1–20 implement URL encoding of strings, and lines 22–28 drive the encoding by feeding a set of URLs to the routines (and repeating

```
 1  uri.AN ← { "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
 2             "abcdefghijklmnopqrstuvwxyz",
 3             "0123456789-_.˜" };
 4
 5  string.join{char; strlist}: {
 6    if (0=#strlist) ← "";
 7   ← (-#char)↓ ⊃ strlist,¨ <char;
 8  }
 9
10  uri.encodechar{char; ignore}: {
11    if ((char=' ') ∧ (' ' ∈ ignore)) ← '+';
12    if (char ∈ uri.AN) ← char;
13    if (char ∈ ignore) ← char;
14   ← '%',(16 16 ⊤ ‘int?char)#"0123456789abcdef";
15  }
16
17  uri.encode{ascii; ignore}: {
18    bts ← uri.encodechar ¨ {ascii; <ignore};
19   ← string.join{''; bts};
20  }
21
22  uri.encodeD{ascii}: ← uri.encode{ascii; ""}
23
24  start ← time{}
25  300 do { uri.encodeD ¨ data; }
26  end ← time{}
27
28  ↓ 'elapsed: ', ⍕ end[2] - start[2]
```

Listing 2: The A+ script used for performance evaluation (written in APL input mode).

this 300 times) and additionally measure the elapsed time. The input for the encoding has been collected during a browsing session and contains 50 URLs of length ranging from 60 to 1439 characters. (Because of length constraints, the test data is not listed here.)

For our experiments, we used a computer equipped with a Dual Core AMD Opteron 275 2.2GHz CPU and 4GB RAM. During the evaluation, the reference implementation of the A+ interpreter (version 4.22) acted as a baseline for comparison, which was executed on a 32-bit Debian Squeeze Linux installation. Our .NET-based implementation (revision 231) was ran on Windows 7 (32-bit) and .NET framework 4.0 by embedding the execution engine into an application utilizing the command line hosting API of DLR.

In our first experiment, we compared the execution time of the A+ script as measured on the reference implementation and on the .NET-based interpreter. The result of the comparison is shown on the first two columns (A, B) of Figure 3. According to the measurements, the reference implementation is about 7 times faster than the .NET port, currently.

However, we also experimented with the interoperability between A+ and .NET. Since the .NET Base Class Library contains equivalents of `string.join` and `url.encode`, we performed two additional ex-

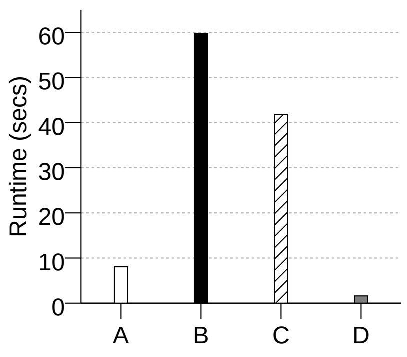

Figure 3: Execution times of the test script A) on the Linux reference implementation, B) on the .NET implementation, C) on the .NET implementation with `string.join` replaced, and D) on the .NET implementation with `uri.encode` replaced.

periments where we removed the A+ implementation of these functions from the test script (lines 5–8 in the first experiment, lines 1–20 in the second) and added their .NET counterparts to the scope of the engine before execution. The result of these experiments is shown on the last two columns (C, D) of Figure 3. The replacement of `string.join` by its .NET equivalent resulted in a nearly 30% speedup, even though this version is still slower than the reference implementation. However, the replacement of `url.encode` yielded a huge performance gain. The execution time in this experiment dropped to 20% of the time measured for the reference implementation, which is equivalent to a 5-fold speedup. This latter result shows one of the possible exploitations of the interoperability, i.e., performance improvement of existing A+ applications by adding .NET implementations for critical code parts.

Speeding up the execution of A+ scripts by calling .NET routines is not the only interesting application of the runtime. Because of the mathematical expressive power of A+, it can be used in .NET code as an embedded domain-specific language. Since A+ is mostly used in financial computations, we illustrate the usefulness of the embedding with the mixed-language implementation of the oft-cited Black-Scholes formula (Black and Scholes, 1973) used to calculate the price of European put and call options. Listing 3 shows a code fragment where mathematical formulas are written in A+ but control structures are in the host C# language.

5 RELATED WORK

Efforts on the integration of script languages into modern object-oriented frameworks did not begin with DLR. One of the first attempts was the implementation of the Python language in/on Java, called JPython (now Jython) (Hugunin, 1997). That approach featured a Python to Java byte code compiler to achive its goals. The developer of the port reported a slowdown by a factor of 1.7 compared to the C implementation.

For the .NET framework, Python was again among the languages to be ported first. However, in the pre-DLR era that was a hard task and the project has been abandoned (Hammond, 2000). This negative result has led some interpreter developers not to reimplement their execution engines in .NET but to provide a bridge between the framework and the existing interpreter (Mascarenhas and Ierusalimschy, 2004).

Then, however, the idea of porting Python to .NET was raised again, which resulted in the successful IronPython project (Hugunin, 2004). In the first publicly announced version (0.2), it was considerably (100-fold) slower on some benchmarks than the C implementation (while faster on some others). The project is still under active development and its performance is regularly compared to the reference implementation (Fugate, 2010). Most importantly, the DLR also grew out from this project (Hugunin, 2007).

The Java community is also putting effort on supporting the execution of script languages on the Java platform. Several related JSRs (Sun Microsystems, 2006; Sun Microsystems, 2011) reached final status and got incorporated into public releases of the platform.

Besides A+, APL has several other derivatives as well. Two of these are even designed to work with the .NET framework. The first version of VisualAPL, an APL-based development platform for .NET, became available in 2003, and it was released as a consumer-ready product in 2009 (Blaze, 2009). However, VisualAPL has departed from the conventional APL and adopted object-orientation, and C# syntax and semantics into the language. More recently, APL# (Kromberg et al., 2010) was introduced based on Dyalog APL. In this case, the language designers have taken careful steps not to introduce language-breaking features. The APL# language is still under active development, with an interpreter in a pre-beta phase.

```
aplusEngine.Execute("d1 := ((log (S % X)) + T * (r + (vˆ2) % 2)) % (v * T ˆ 0.5)", scope);
aplusEngine.Execute("d2 := d1 - v * T ˆ 0.5", scope);

if (option == "call") {
    result = aplusEngine.Execute("(S * CND{d1}) - (X * ˆ(-r * T) * CND{d2})", scope);
} else {
    result = aplusEngine.Execute("(X * ˆ(-r * T) * CND{-d2}) - (S * CND{-d1}) ", scope);
}
```

Listing 3: A C# code fragment embedding A+ expressions (written in ASCII input mode).

6 SUMMARY AND FUTURE WORK

In this paper we have introduced our .NET-based clean room implementation of an A+ runtime. Although the current version of the .NET port is slower than the reference C implementation, its greatest advantage, i.e., the interoperability between A+ and .NET, is already visible. In a preliminary experiment we were able to achieve a 5-fold speedup by substituting A+ functions with equivalent .NET methods. Additionaly, we have also shown how A+ scripts can be embedded into C# code, thus exploiting the expressiveness of A+ in financial computations.

For the future, several steps are already seen ahead. We would like to focus on three major areas, each consisting of several research and/or development topics.

The first area is the enhancement of the runtime. First of all, we would like to improve the runtime to support all language features of A+ (some minor deficiencies still exist). Furthermore, besides extending the compatibility of our implementation, we also wish to investigate the possibilities of improving its performance. This latter topic, i.e., speeding up script engines, is a huge research area on its own.

The second area we would like to focus on is the analysis and comparison of the two implementations, i.e., the reference interpreter and the .NET runtime. By analysing both systems and computing several maintainability and modularizability metrics, and by evaluating their performance on a broader benchmark set, we will be able to assess the costs and gains of porting legacy interpreter implementations to new platforms.

The third area of our interest is language design. We would like to develop A+ into a modern programming language of the .NET framework. This will require the extension of the language with several concepts, like classes and objects, to foster the interoperability between A+ scripts and the host environment. Moreover, we will have to define a restricted but still useful subset of the language that is compilable ahead of execution time. Last but not least, to aid efficient programming, we plan to provide IDE support for the A+ language.

ACKNOWLEDGEMENTS

The authors would like to thank Csaba Bátori for his valuable help in the development of the .NET port of A+.

REFERENCES

Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654.

Blaze, J. (2009). Prior and current versions of VisualAPL. APL2000 Developer Network Forum, `http://forum.apl2000.com/viewtopic.php?t=447` (Accessed 5 June 2012).

Chiles, B. and Turner, A. (2009). *Dynamic Language Runtime*. `http://dlr.codeplex.com/documentation` (Accessed 26 April 2012).

Clayton, L., Eklof, M. D., and McDonnell, E. (2000). *ISO/IEC 13751:2000(E): Programming Language APL, Extended*. Internation Standards Organization.

Fugate, D. (2010). IronPython performance comparisons. `http://ironpython.codeplex.com/wikipage?title=IronPython%20Performance` (Accessed 26 April 2012).

Hammond, M. (2000). *Python for .NET: Lessons learned*. ActiveState Tool Corporation.

Hugunin, J. (1997). Python and Java – the best of both worlds. In *Proceedings of the 6th International Python Conference*, pages 11–20, San Jose, CA, USA.

Hugunin, J. (2004). IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004*, Washington, DC, USA.

Hugunin, J. (2007). A dynamic language runtime (DLR). `http://blogs.msdn.com/b/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx` (Accessed 26 April 2012).

Kromberg, M., Manktelow, J., and Scholes, J. (2010). APL# - an APL for Microsoft.Net. In *Conference USB stick of APL2010*, Berlin, Germany.

Mascarenhas, F. and Ierusalimschy, R. (2004). LuaInterface: Scripting the .NET CLR with Lua. *Journal of Universal Computer Science*, 10(7):892–909.

Morgan Stanley (1995–2008). *A+ Language Reference*. `http://www.aplusdev.org/Documentation/` (Accessed 26 April 2012).

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition.

Sun Microsystems (2006). *JSR-223: Scripting for the Java Platform*.

Sun Microsystems (2011). *JSR-292: Supporting Dynamically Typed Languages on the Java Platform*.