

Extending A+ with Object-Oriented Elements

A Case Study for A+.NET

Péter Gál[1(✉)], Csaba Bátori[1,2], and Ákos Kiss[1]

[1] Department of Software Engineering, University of Szeged, Dugonics ter 13, Szeged 6720, Hungary
{galpeter,akiss}@inf.u-szeged.hu
[2] BlackRock, Vaci ut 47, Budapest 1134, Hungary
csaba.batori@blackrock.com

Abstract. This work presents a language extension to A+ that allows developers to conveniently handle external objects in A+ code. It discusses a set of object-oriented concepts and investigates their implications for the A+ language. For these new language elements, new symbols are added, and each of them represents basic operations (member access, member modification, type casting, and indexing) on objects. For each new operation the underlying mechanism is presented, and simple examples are provided showing their usage. Using these new operations, various .NET classes can be accessed and used from A+ programs without writing any additional code.

The presented language extension in this paper is runtime agnostic and it could enable the interoperation of A+ not only with the .NET framework but with other object-oriented runtimes, too. Additionally this extension adds the basic mechanisms for implementing full object-oriented support for the A+ language in the future.

Keywords: A+ · Object-oriented programming · Language extension · DLR

1 Introduction

A+ is an array programming language [11] based on the APL language. The language was created more than 20 years ago to give an aid in real-life financial computations. Still, even in the present age, critical A+ programs are still used in computationally-intensive business environments. The original interpreter-based execution environment of A+ is implemented in C/C++ and is officially supported on Unix-like operating systems only. However, the development activity of this implementation is very low and the maintainability of the code is questionable, as discussed in the paper [6].

Csaba Bátori was with the Department of Software Engineering at the time of the language extension work.

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12957, pp. 141–153, 2021.
https://doi.org/10.1007/978-3-030-87013-3_11

Thus in a previous work, the A+.NET project [5] was introduced, which is a cleanroom implementation of the original A+ environment for the .NET environment[1]. The goal of the .NET-based implementation was to allow the interoperability between A+ and .NET programs. This allows A+ scripts to call .NET methods, allows .NET applications to embed A+.NET as a domain-specific language, and A+ scripts could be executed just like any other application on Windows systems. In addition this provides a way for existing A+ applications to extend their lifetime, provide .NET developers existing financial routines in form of A+ code, and allow access for .NET libraries accessible for A+ developers.

However, using .NET classes and methods from A+.NET is not straightforward. For each .NET method or class, specially formatted wrapper methods are required that need to be written by the developer. Fortunately, this extra burden can be removed by introducing language extensions that can handle external objects in the A+ language. In this paper a set of required operations for a language which is not object-oriented is presented to cooperate with the object-oriented concepts.

The rest of the paper is organized as follows. In order to make the paper self-contained, in Sect. 2 a short introduction into the A+ language is presented, focusing on distinguishing features. (This does not provide too in depth information, thus for further details the reader should refer to the A+ Language Reference [11] document.) In Sect. 3, we describe the current state of A+ and .NET interoperability. In Sect. 4, the proposed extensions are presented, and in Sect. 5, the implemented extensions for the A+.NET runtime are detailed. In Sect. 6, we overview the related work on extending legacy languages with object-oriented concepts, and finally, in Sect. 7, we conclude the paper and give directions for future work.

2 The A+ Programming Language

A+ derives from one of the first array programming languages, APL [2]. Due to this legacy, A+ have unusual aspects compared to other more recent programming languages. One of the most important one is that, although being a language of mathematical origin, all functions have equal precedence and every expression is evaluated from right to left.

There are more than 60 built-in functions, which are usually treated as operators in other languages. Among these functions there are simple arithmetic functions and also some other complex ones, like inner product or matrix inverse calculations. All built-in and user defined functions are treated as first class-citizens, which means that the programmer can assign functions to variables and is also able to pass functions as arguments to other functions. In addition almost all built-in functions have special non-ASCII symbols associated. This syntax makes the code compact and also allows mathematical-like notations in an A+ code. However, this could pose a challenge for the untrained eye when trying to read and understand the source code.

[1] Project hosted at: https://github.com/elecro/aplusdotnet.

```
1  (×/s) , +/s ← 1 + ⍳ 10
```

Listing 1.1. Calculating the product and sum of the first 10 natural numbers in A+.

The language defines a very narrow set of types, it has integers, floats, characters, symbols – denoted by a starting backtick character –, functions, and a type named box – which is a special type wrapping another type. In A+ even the a single number or character is treated as an array.

Arrays can be placed in an array if they have the same type. This means that it is disallowed to put a character array inside an integer array. The programmer would need to convert the character array into an integer array first to be able to put it into the target array. Currently, the runtime environment only allows the creation of arrays with a maximum of 9 dimensions, as specified by the language reference [11].

In A+, there is a namespace-like concept called context. Each variable and function is in such a context and each context has a name. If one would like to access the `demo` variable from the context named `tst`, one would write `tst.demo` in A+. It is also possible to have a context named `tst.ctx` and then, one can use the `tst.ctx.var` identifier to access or modify the `var` in that context. This `<context name>.<var>` format is called the qualified name – note that there is one dot before the `<var>` part. The format `<var>` is called the unqualified name, when there is no dot character in the name. During the execution of an A+ code, the runtime always knows what context it is in and automatically extends all unqualified names. It is also possible to change context at runtime. There is a default root context that has no name. Thus, if one uses the `var` variable then it will return the value of the `.var` variable if the current context is the root context or it will return the `tst.var` value if the runtime is in the `tst` context.

The specialties mentioned above can be illustrated in Listing 1.1, which shows a way to calculate the product and sum of the first 10 natural numbers. Breaking down the example in smaller parts can help in understand how it works. The `⍳ 10` expression generates an 10-element array containing values from 0 to 9. The expression `1 +` will increment each element in this array by one. After this, the value is assigned to the variable `s`. By using the `+/` operator the sum of the vector is calculated. Similarly, the `*/` will calculate the product of the same vector. Lastly, using the concatenation function – which is denoted by a comma – the two values are merged into a single two-element array containing the `3628800 55` values. The parentheses around the product is important, as without it the variable `s` and the sum would be concatenated forming a two-element array and the product would be calculated on this new two-element vector because of the right-to-left evaluation order.

As the code above 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 the present value at various interest rates in a single line [11, page 62].

```
1 var scope = engine.CreateScope();
2 scope.SetVariable("tst.value", AFloat.Create(3.2);
```

Listing 1.2. Registering the constant 3.2 into an A+.NET scope

3 State of the Art of A+ and .NET Interoperability

A+.NET allows the developers to inject extra variables and functions into the scope of the runtime.

Each value that the developer wants to register into the runtime must be wrapped into an `AType`. There are four types of values that can be registered: numbers – integer and double –, characters, and symbols. The A+.NET types for these are `AInt`, `AFloat`, `AChar`, and `ASymbol` respectively. Naming the double type as float might seem strange at first sight, however, floats represent double precision floating point number in A+ terminology. Thus, `AFloat` adheres to the original naming conventions. Listing 1.2 depicts how someone can add the constant `3.2` into the A+.NET runtime under the name `tst.value`. The first line exemplifies the creation of a new scope in the runtime, which stores, all methods and variables and can be used when executing A+ code from .NET.

Each added method must adhere to some rules: First, the method must be a static method. Second, the return type must be `AType`, which is the base interface type for all types in the runtime. Third, the first argument must be an `Aplus` type, which contains the runtime environment information and can be accessed by the method. And fourth, any other arguments must be of `AType` type and – most importantly – they must be in reverse order. (The reverse order is required because the A+ language evaluates function arguments from right to left while the C# does not. Thus the A+.NET runtime performs a trick and requires all methods to have a reverse order of arguments. So for an A+ function that accepts two parameters, the second argument of the A+ function becomes the first non-environment argument of the registered C# method and the first argument in A+ will be the last parameter in C#. In Listing 1.3, a conforming method is shown, which meets all the requirements for the A+.NET.

There are two ways of adding a compatible method into the A+.NET runtime. The first one is to create scope – which contains the variables – for the A+.NET engine/runtime and then add the required method as a variable into the scope. The function must be wrapped into an `AFunc` which contains information about the callable method for the runtime. This approach is shown in Listing 1.4, where method `Foo` from Listing 1.3 is registered into the scope as `tst.foo`.

The second option is to create a class with the compatible methods and annotate all functions as `AplusContextFunction` and the class as `AplusContext`. The annotation `AplusContext` specifies the context name under which the methods should be registered. The function annotation specifies the name by which the method should be accessible from A+. Such annotated classes can be loaded via the `$load` system command specifying the name used in the annotation of the

```
static AType Foo(Aplus env, AType arg2, AType arg1) {
  // ...
}
```

Listing 1.3. An A+.NET compatible C# method

```
var scope = engine.CreateScope();
scope.SetVariable("tst.foo",
  AFunc.Create("foo", Foo, 2, "Test method"));
```

Listing 1.4. Registering a C# method into an A+.NET scope

class. Listing 1.5 depicts a class with all the annotations. For this specific case, the `$load tst` instruction in A+ will load the method into the context named `tst` with the name of `foo`.

After we have loaded the required function(s) into the A+.NET runtime – in any of the two ways described above –, it is possible to invoke them just like any other A+ functions, as shown in Listing 1.6.

As visible from the explanations and examples above, adding a method to A+.NET runtime requires writing a lot of code. Writing these wrapper methods or classes for each required .NET class is tedious and error prone, not to mention that there could be a lot of copy-paste code in the end.

Instead of writing these wrappers or providing tools to generate them, a better way would be to have a mechanism that allows run-time access for the required classes, methods, properties, and variables. For this, we have reviewed the object-oriented concepts and investigated requirements for the A+ language to handle external objects conveniently. These concepts will be described in the next section.

```
[AplusContext("tst")]
class Demo {
  [AplusContextFunction("foo")]
  static AType Method(Aplus env, AType arg2, AType arg1) {
    // ...
  }
}
```

Listing 1.5. A+.NET conformant C# method with annotations, registered as `tst.foo` in the A+ environment

```
1 tst.foo{1;2}
```

Listing 1.6. Invocation of a registered C# method from A+.NET

4 Language Extension

Due to the fact that the A+ language is not an object-oriented language by design, the investigation of the required operations to handle objects in a language was the first step.

During the investigation the base requirements to handle objects in a language were identified. First, the introduction of a way to represent objects in the runtime was required. This essentially means that a new type should be added to the language.

Second, there are four basic operations that should be supported by a language to handle the most basic tasks on objects. These are the following:

- Accessing members (methods, variables, and properties): this operation provides the means to read variables and properties, and to access methods.
- Modifying variables and properties: the operation makes it possible to assign new values to properties and values.
- Type casting: an important operation to resolve ambiguities.
- Using indexer properties: in .NET indexing objects is done via a special property and there are compound types where there is no other way of accessing elements e.g. in ArrayList. This is mainly required for the .NET binding.

Note that the method invocation is not among the operations. This is because most languages already support it in some way. So the existing method invocation syntax can be improved to handle the invocation of methods on instances and classes.

Also note that with the operations above the language will still not support native classes. That is, it is not possible to write classes in the same language for which the operations are implemented. Fortunately, the described operations take the language closer to the real object-oriented languages. So it becomes possible to build further object-oriented functionalities on top of these new operations. Also note that the concept of these operations is not tied to one specific language, in our case to A+. Thus, these could be added to other non-object-oriented languages as well.

5 Implemented Language Extension

As described in the previous section, four operations are a must for the A+.NET runtime to handle (external) objects. In the case of A+.NET, the implementation of the required operations can take advantage of the reflection capabilities of the .NET framework and also the code generation possibilities of the Dynamic Language Runtime (DLR) framework. The use of reflection provides ways to search

for classes and methods, and the code generation capabilities of DLR enable the building of wrapper method(s) at runtime, thus eliminating the need for writing wrapper methods by hand. With these two techniques combined, it is possible to provide general functions for A+.NET which can perform the required lookups and code generations, thus avoiding unnecessary repetitive manual coding. Additionally, the implementation must take into account, that in A+ the order of evaluation is right to left and any changes to this rule could break existing code.

As mentioned in the previous section, a new type must be added into the runtime. In A+.NET, this new type is internally named `AObject` and its instances can store any .NET object. The subsequent sections will define each previously mentioned operation in more detail.

5.1 Accessing Methods, Variables, and Properties

In A+, functions are first class citizens, thus it must be ensured that each .NET method can also be used accordingly and not just for simple method invocation. Fortunately, this lenience is a win in our case as this will allow us to handle the methods, variables, and properties in a similar way, thus simplifying the implementation. The following algorithm describes the inner workings of the accessor operation. This operation has two input arguments, the name of the property, variable, or method to be looked up and the instance or class on which the lookup should be performed, and can be formulated as `SelectMemeber(x, y)`. Note that, both the member and class names must be passed as an A+ symbol type. This approach was chosen not to interfere with the context concept of the A+ language.

1. Collect all methods, variables, and properties of the instance or class specified by the second parameter (that is `y`).
2. For each method, variable, and property check if it has the same name as specified by the first parameter (that is `x`).
3. If no match is found, return an error, reporting that there is no member accessible with the given name.
4. If a variable or property was found, return it as an A+ type. In the case of a primitive type, such as string, number, or enum value, they be converted into their compatible A+.NET counterpart which are: array of `AChar`, `AInt`, `AFloat`, and `ASymbol` respectively. In any other case, the value must be returned as an `AObject`.
5. If the lookup found a method, construct a lambda function using the DLR capabilities, which accepts a variable number of arguments, performs the type matching algorithm – which is described in Sect. 5.5 – and invokes the method returned by type matching. The constructed lambda method is then returned as an A+ function type, represented by the `AFunc` type in A+.NET.

```
1  constructor ← ⊖ ‘Bar
2  instance ← constructor{}
3  value ← ‘variable ⊖ instance
4  (‘method ⊖ instance){}
```

Listing 1.7. Example A+.NET accessor usage

Constructor access can be thought of as a special case of method access. The main difference, in this case, is that the function name parameter is always that of the class.

When the `SelectMember` returns a lambda function – either a constructor or another method – the A+ function containing it can be called a traditional function in the runtime. Thus, method invocation does not require additional implementation or functionality in the engine. The generated lambda function contains all required information on how to perform the .NET method invocation.

To integrate seamlessly into the syntax of the A+, the existing but hitherto unused ⊖ symbol was chosen for the `SelectMember` operation. The format $x \ominus y$ is treated as `SelectMember(x, y)`. Listing 1.7 shows an example use case for the `SelectMember` operation. In line 1, the constructor lookup for the .NET class named `Bar` is demonstrated, which is invoked in line 2. This invocation looks like any other A+ function invocation. Line 3 shows how variables can be accessed with the new operation. Finally, on line 4, the method access and invocation is demonstrated. As this line shows, the call to the ⊖ function is enclosed in parentheses to ensure the evaluation order dictated by A+, also this way the language grammar does not require radical changes to give the ⊖ function a higher precedence.

The described `SelectMember` operation provides a basic building block for handling the modification of variables and properties which is described in the next section.

5.2 Variable and Property Modification

The modification of variables and properties are similar to the access case, with the addition of a third parameter for the accessor algorithm. This third parameter will serve as the new value for the target variable or property. Additionally, searching for methods is not allowed when performing the member name lookup.

The following algorithm describes how to modify a variable or property on a given instance or class. It has three input parameters. The first is the name of the variable or property to modify. The second is the class or instance on which the variable/property lookup should be performed. And the third argument is the new value for the variable/property. The operation is named `SetMember(x, y, z)`

1. Get all variables and properties of the object or instance specified by the second parameter (that is `y`).

```
1  (‘variable ⊖ instance) ← 42
```

Listing 1.8. Example A+.NET instance variable modification

2. For each variable and property check if it has a name which is specified by the first argument (that is `x`).
3. If no match is found, return an error, reporting that there is no such member to modify.
4. As the lookup found a variable or a property, try to cast the new value – specified by the third argument `z` – to the type of the variable/property.
5. If the cast is not possible, return with an error, stating that it is not possible to update the variable with the supplied new value.
6. If the cast was correctly performed, assign the cast value to the selected variable/property.
7. Return with the new value as specified by the third argument. Note that this is the original value of z, not the cast one.

For the `SetMember` operation the same $\ominus$ symbol was used, but to invoke the algorithm it must be used in the $(x \ominus y) \leftarrow z$ format. This can be detected during the parsing of the code, as the accessor function is on the left side of the assignment function. Note that the parentheses are required because of the strict right-to-left evaluation order of the A+ code and the fact that the assignment function has the same precedence as any other function in the language. Listing 1.8 exhibits a use case for the operation.

5.3 Indexers

In .NET, there are numerous objects that provide access to their individual elements. This is usually performed with the help of indexers. To have better integration with .NET types in the A+.NET runtime, it is important to provide an intuitive way to access such elements.

Fortunately, there is also the notation of indexing elements in A+ and, therefore, there is no need for introducing new syntactic elements. We can leverage the existing indexing mechanism and only improve its internal mechanisms to add the .NET indexer binding.

As the indexers in .NET are special properties that can be queried by reflection, the `SelectMember` algorithm can be used to get the value of an item at a given index from an object or instance. Thus, in the indexer case, the `SelectMember` algorithm will search for the indexer properties. In the case of element assignment at a given index, the `SetMember` algorithm will perform the same search as in the element access case.

An example of indexer usage is shown in Listing 1.9. After creating an instance of `ArrayList` in line 1, the code performs an element addition to the list in line 2. (The equivalent code in C# would be: `list.Add(1);`.) Then the usage of the indexers is shown: first, the access of a single element by its index on line 3, continued by the value modification of a given index in line 4.

```
1  list ← (⊖ ‘ArrayList){}
2  (‘Add ⊖ list){1}
3  list[0]
4  list[0] ← 2
```

Listing 1.9. Example indexer usage for .NET types

```
1  (‘booleanVariable ⊖ instance) ← ‘Boolean ◇ 1
```

Listing 1.10. Example A+.NET type casting for .NET types

5.4 Type Casting

Type casting is an important operation in the .NET world, as it allows developers to resolve ambiguities. A new ◇ symbol is introduced into the language, providing the means to perform the .NET type casting functionality in the runtime from the A+ code. This function can also be used for resolving method call ambiguities, which is detailed in Sect. 5.5.

Listing 1.10 depicts an example of the type casting function, represented by the ◇ symbol. In the example, the number `1` is initially represented as an A+ number type, but using the new type cast function, it will be changed to a `Boolean` type which is from the .NET world. The A+.NET symbol `‘Boolean` is used to specify the target type for the cast function. After the cast is performed, the value can be assigned to a boolean variable or property on a given instance if needed.

If the type cast operation cannot be accomplished, an error is reported to the A+.NET runtime. For example, if the developer wants to cast an integer to a string the error is returned, stating that an invalid cast was attempted.

5.5 Type Matching

As mentioned before in the `SelectMember` operations, .NET methods are looked up by their names. However, just a method name is not always enough to correctly match a method. It is possible that there is more than one method with the same name and the difference is only in the number of arguments or in their types. Thus, to correctly select a method, the types and number of parameters are also required.

The type matching algorithm should be performed after the potential methods based on their names have already been found. First, any method is ignored if it does not have the same number of arguments as the number of arguments supplied for the method invocation. In case there are no methods left to select from, an error is reported during runtime, that the number of parameters are incorrect. Second, as the number of arguments is now correct, a type distance vector calculation is performed. The basic building block of the calculation is the type distance notation.

```
1  class Place {}
2  class Bar : Place {}
3  class Vehicle {}
4  class Car : Vehicle {}
5
6  void Foo(Place arg1, Car arg2) {}
7  void Foo(Bar arg1, Vehicle arg2) {}
```

Listing 1.11. Example for C# method ambiguity

The type distance calculation of non-primitive types (i.e. classes) is based on the inheritance hierarchy of .NET types. If two types are in an inheritance relation, then the type distance of those types is the length of the shortest path between them in the inheritance graph, with the result of `0` if the two types are the same. If the two types are unrelated inheritance-wise, their type distance is specified as infinite. For example: if there is a class named `Bar` which is a subclass of class `Place` then the type distance between `Bar` and `Place` is one.

For primitive types, inheritance hierarchy is not applicable. However, the C# reference documentation [4, § 11.2] specifies conversion tables, which help to define a pseudo-hierarchy between them, which can be used the same way as the real inheritance for non-primitive types. This pseudo-hierarchy is as follows: `Boolean` → `SByte` → `Byte` → `Int16` → `UInt16` → `Int32` → `UInt32` → `Int64` → `UInt64` → `Single` → `Double` → `Decimal` → `AType` → `Object`, the `Char` type connects into the `UInt16` type, the string values are starting with `String` → `Enum` and connecting into the `AType`. Thus, for example if there is an input argument with the type `Byte` and the method requires an `Int32`, the distance between the two types is `3`. The A+ types are using this pseudo-hierarchy to allow interoperability with the .NET methods. A+ symbols are using the chain starting from the `String` type, integers either from `Int32` or `Double`, and characters use the `Char` type as a starting point.

Based on this type of distance information, it is now possible to define the type distance vector. The type distance vector is a vector of `N` elements where `N` is the number of input parameters for the current method invocation and for each element the type distance is calculated between the input parameter and the parameter of the potential method. After the distance calculations, the best method is chosen. A method is considered better than the other if each element of its type distance vector is smaller or equal than the corresponding element in the other method's vector, but at least one element is strictly smaller than its corresponding element. This selection method is based on the better function member selection described in the C# language specification [4, § 12.6.4.3]. Note, however, that there may be incomparable elements in this relation. Only if there is a single method that is better than all other alternatives does the runtime select that method for invocation. Otherwise, the runtime reports an error because of the ambiguity. Such ambiguities can be resolved with the newly introduced type casting operation.

Listing 1.11 depicts a C# method ambiguity. For this use case the `(`Foo ⊖ `class){(⊖ `Bar){}, (⊖ `Car){}}` invocation would not be successful. The calculated type vector distance for the invocation and the first `Foo` method would be – based on the type hierarchy – the vector `[1, 0]`, and for the second `Foo` method it would be `[0, 1]`. Thus, based on the better method selection rules, it is not possible to select one method and the ambiguity error is reported. However – as stated before – it is possible to resolve the ambiguity if e.g. the second parameter for the method invocation is cast to `Vehicle`. Then the type distance vector for the second case would result, in a `[0, 0]` vector and a perfect match would be found.

6 Related Work

Extending an already existing language with object-oriented capabilities is not a new thing in the world of programming languages. Even for APL – from which A+ derives – there are object-oriented extensions [1]. Before the creation of the A+ language, there were already experiments to incorporate object-oriented notations into the APL language [7]. Both of these implementations provide object-oriented notations, allowing the developers to create classes from APL code and not just operations for objects. However, in one case, the authors introduced the member access operation to be read from left-to-right thus making the right-to-left reading mode a bit awkward when such operations are performed. In the other case, the authors introduced a special system function (`□NEW`) to construct new instances from a given class. Additionally, possibility to interact with other objects – that are outside of the APL language – were not mentioned.

Moreover, the APL language is still in active use and there are different companies providing support for its implementations, the most notable one is provided by Dyalog Ltd. [3]. This version also provides object-oriented notations to make the developers' life easier. In addition, it is also gives a syntax to write classes, not just the option to instantiate them, and access its members. The variant took the approach to introduce a new system function to instantiate classes and use the dot syntax to select members on classes/instances [8].

Other array-based programing languages used in mathematics already have object-oriented support, for example the widely used Maple [9] and Matlab [10, 12] software packages.

7 Summary and Future Work

In this work, a set of requirements for a language with no object-oriented programming concepts were presented, to handle external objects. This can be achieved by adding a new type into the language and implementing four basic operations. The case study shows that with the new type and operations it is indeed possible to conveniently handle objects in A+.NET while also eliminating the need to write or generate wrapper methods to expose objects into the runtime. The introduction of only two new language elements into A+ was required

for this extension and the refinement of the indexing, and assignment operations. Additionally, a type distance vector-based approach was presented for matching formal .NET and actual A+ method parameters.

However, there is still work to do as there are constructs in .NET which are not accessible in the current A+.NET extension. In the current approach there is no support for generics, parameter passing by reference, output parameters, and no support for invoking overloaded operators from A+. In addition, adding class support would be beneficial to further improve the language. The extension of the language with these new constructs may facilitate the more widespread use of A+.NET. Furthermore, after having a full object-oriented extension for the A+ language, a comparison with the APL object-oriented extension can be performed.

Acknowledgements. This research was supported by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary.

References

1. Brown, R.G.: Object oriented APL: an introduction and overview. In: Proceedings of the International Conference on APL-Berlin-2000 Conference, APL 2000, pp. 47–54. ACM, New York (2000). https://doi.org/10.1145/570475.570482
2. Clayton, L., Eklof, M.D., McDonnell, E.: ISO/IEC 13751:2000(E): Programming Language APL. Extended, International Standards Organization, June 2000
3. Dyalog Ltd.: Dyalog APL. http://www.dyalog.com/. Accessed 15 June 2021
4. ECMA International: ECMA-334 - C# Language Specification, 5th edn., December 2017. https://www.ecma-international.org/wp-content/uploads/ECMA-334_5th_edition_december_2017.pdf. Accessed 15 June 2021
5. Gál, P., Kiss, Á.: Implementation of an A+ interpreter for .NET. In: Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012), Rome, Italy, pp. 297–302. SciTePress, 24–27 July 2012. https://doi.org/10.5220/0004129202970302
6. Gál, P., Kiss, Á.: A comparison of maintainability metrics of two A+ interpreters. In: Proceedings of the 8th International Joint Conference on Software Technologies - ICSOFT-EA, (ICSOFT 2013), pp. 292–297. INSTICC, SciTePress (2013). https://doi.org/10.5220/0004597702920297
7. Girardot, J.J., Sako, S.: An object oriented extension to APL. In: Proceedings of the International Conference on APL: APL in Transition, APL 1987, pp. 128–137. ACM, New York (1987). https://doi.org/10.1145/28315.28330
8. Kromberg, M.J.: Arrays of objects. In: Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, pp. 20–28. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1297081.1297087
9. Bernardin, L., Chin, P., DeMarco, P., et al.: Maple Programming Guide. Maplesoft (2011)
10. MathWorks: Matlab Object-Oriented Programming (2021). https://www.mathworks.com/help/pdf_doc/matlab/matlab_oop.pdf. Accessed 15 June 2021
11. Morgan Stanley: A+ Language Reference (1995–2008). http://www.aplusdev.org/Documentation/. Accessed 15 June 2021
12. Register, A.H.: A Guide to MATLAB Object-Oriented Programming. Scitech Pub Inc. (2007). https://doi.org/10.5555/1202571