Features to Defer From Proposed ECMAScript 4[1] (position statement)

26 February 2008

Lars T Hansen and Jeff Dyer, Adobe Systems

Introduction

When the ES4 effort restarted two years ago we created a timeline that called for an initial proposal period followed by a period of culling of those proposals that, on reflection, were found not to be suitable after all. At long last TG1 reached the final proposal cut-off this past September, after which the group has published an ES4 language overview and a paper on the compatibility of ES4 with ES3. TG1 continues to work on the reference implementation and the language specification, activities that help to firm up the proposals.

Numerous proposals were rejected before TG1 arrived at the proposal cut-off. It is however our opinion that ES4 as described by the language overview represents a *starting point* for the final culling of proposals, not the *result* of that process. Two observations lead us to this conclusion. First, it is not until now, when we have a complete proposal set and are working on details, that we see how well conceived the proposals actually are and how they fit together. Second, we have to be mindful of how language and implementation complexity affect the timeline for producing a specification and production implementations.

The success of ES4 hinges entirely on adoption, and adoption in turn depends on ES4 being a credible and likeable language that can be implemented widely, correctly, and relatively quickly. We believe that incorporating unproven, strange, or costly features into ES4, or deferring the ES4 spec beyond 2008, will cast significant doubt on the viability of the language, will slow its implementation and adoption, and may cause TG1 problems in later revisions.

In the spirit of ensuring that ES4 gains wide acceptance, this document enumerates some feature proposals for ES4 that we find troublesome, and explains in some detail why that is.[2] We would like TG1 to consider deferring the features we list until a later edition of the language, allowing TG1 members first to accumulate experience with the more fundamental features of ES4, and giving them time to consider the troublesome features' final form. We believe that deferring the features we list in this document from ES4 will significantly increase the likelihood of finishing the spec in 2008, will increase community buy-in, will help keep implementation complexity manageable, will reduce the risk of standardizing something we'll later regret, and will—with plenty of luck—somewhat reduce the discord among TG1 members.

In practical terms, it is more important to standardize the important fundamentals of ES4—namespaces, fixtures, types, and self-hosting libraries—along with all the creature comforts we believe to be sound, than it is to include features that are controversial, hard to implement, hard to use, or likely to delay the specification or commercial implementation.

The themes that led us to our list are:

Premature standardization: Some features do not feel "finished" in the sense that, though TG1 members have a clear (but not always unified!) intuition about why they would like them, it is not clear that the features can be used in the intended way, or indeed that they can be implemented to conform to the intuition. Two examples are `wrap`, which is so far not well specified and with which we have no experience (and on which others have written negatively in the past), and decimal numbers, which after critical review in TG1 and subsequent scaling back no longer can be used in the way we first intended, as an incantation against the evils of binary floating point.

Unclear use cases and usage patterns: Some features appear to be well-defined but have uncertain applicability

[1] Based on the 2007-10-23 Language Overview and later resolutions.

[2] Ours is one of two possible approaches and relies on our intuition and taste. The other approach, which may be better but is more costly, starts with ES3 and the full proposal set and adds (sets of) proposals to the language, all the time considering coherency, compatibility, and complexity. Unlike the process of the last two years, such a process would benefit from the knowledge of the full set of candidate features.

in real programs. Again TG1 members have an intuition about why the feature is good, but no practical experience with using it, and no handle on whether the feature is truly valuable. We risk incurring cost without receiving any benefit. Two examples are reformed `with`, which may or may not make it easy to move programs that use straight `with` to strict mode, and stack marks, which are at best an implementation technique for other, truly desirable, features (and an optional feature of ES4 besides).

High implementation cost or other implementation issues: Some features preclude the use of certain implementation techniques, or make the use of certain implementation techniques very hard. Other features are simply expensive to implement. Both issues are subjective but real, and implementation cost is not irrelevant to the success of the language. An example of the former is that a tree-walking interpreter has problems handling tail calls and generators in a natural, performant way. Two examples of the latter are generic functions and efficient support for `wrap`.

Table 1 briefly summarizes the points made in the remainder of this paper. An "X" in a cell means that we think the feature named in the row head suffers from the problem described in the column head. A "?" means that the feature might suffer from that problem, but that we think it is more a matter of opinion.

A similar table could be made for all the features of Proposed ES4, breaking it down not just by negative attributes but also by positive ones. We suspect such a table would be useful in the discussion that has already started, about the rationale for each feature.

	Premature standardization	**Unclear use cases Uncertain usage**	**Implementation hardship**
Numeric conversion	X	X	
Int and uint	X	X	?
Decimal	X	X	X
Operator overloading	X	X	?
Generic functions	X	?	X
Wrap	X	?	X
Stack marks		X	X
Generators		X	X
Tail calls	X	?	X
Nullability	X	X	
Program units	X	X	
Reformed "with"		X	?
Resurrected "eval"		X	
Namespace filters	X	X	

Table 1: Strikes against certain proposed features of ES4

Though the deferrals proposed in this paper barely reduce the number of tokens, productions in the grammar, or classes and methods in the library, they significantly reduce run-time system and compiler complexity.

Numbers: Conversion, int, uint, and decimal

The situation with numbers feels confused at the moment. We suggest that ES4 scales back to ES3 style numbers by providing a primitive number type ("double"), its wrapper, and thus sidesteps issues of convertibility, optimization around `int` and `uint`, and the use of `decimal`, none of which are clearly thought out at the present.

Conversions

(Premature; uncertain use) Annotating a variable with any numeric type does not provide any guarantee at all against incompatible numeric values being stored in the variable: An annotation with e.g. "`int`" forces a `ToInt32` conversion on assignment, and that operation accepts any numeric value. Graydon has suggested that conversions be disallowed at run-time if the source value can't be represented in the target type, but to us this does not spell good usability. Most languages allow conversions from type A to type B if all values in A can be represented by B. (In ES4, this is true for `int` or `uint` conversion to `double` or `decimal`, but no others.) That kind of restriction may or may not be a hardship for ES programmers, but it would probably make more sense: annotations should have teeth; that's what they're for. Liberal conversions like we currently have them are

appropriate when type annotations are mandatory, but that's not true in ES4.[3]

int and uint

(Premature; uncertain use; hardship?) `int` and `uint` are meant to be "optimization hooks". But what resources are being optimized? `int` and `uint` values overflow to `double` during arithmetic, so any optimization effect is only realized in three cases:

- *Space:* When the implementation optimizes the storage used, mainly in `Vector.<int>` contexts and for `int` instance variables.
- *Speed:* When the compiler knows that it is reading optimized values, that it will truncate the result of an operation to an optimized value, and finally that the computation performed as an optimized value without overflow checking will produce the same result as the computation performed with overflow to `double` followed by a truncation back to the desired representation. To be sure, there are situations like that, but if those overflows are rare, and overflow checking is fast in any case (branch on carry to the overflow handler), implementations will probably be happy to perform the overflow check. Finally, unannotated slots will likely continue to be the norm, and implementations will wish to optimize untyped `int`/`uint` arithmetic as much as possible.
- *Keystrokes and consistency:* The programmer can use the annotation `int` to specify truncation-on-write rather than having to write `int(x)` everywhere.

Since the speed optimization is rarely realized, then except for the fact that they can currently be instantiated, `int` and `uint` are primarily chopper and storage types, like the much-derided `byte` proposal. We think these types should be taken out of ES4 and reconsidered later, perhaps with different semantics (wrapping instead of overflowing? error signaling instead of overflowing?), when we understand what it is that we're really trying to optimize and how to do it.

decimal

(Premature; uncertain use; hardship) As currently envisioned, `decimal` is just a distinguished number type one has to ask to use (by conversions, annotations, or suffixes) and which only results from operations on one or more decimal operands. Yet the primary argument for decimal is that it is The Right Thing; that Joe Programmer, who divides 1 by 10 to get something not quite like 0.1 and is upset about it, would be helped by decimal. This is not true, because Joe won't know to use decimal. Therefore the utility of decimal is strictly limited to those who know how to ask for it. Most people won't care about it, because they don't have to; Joe's world will remain broken.

We suspect the only thing that will fix Joe's world is a big red switch that means "use decimal everywhere unless I specifically ask for anything else". We have no idea how that would fit into ES in its various hosted environments, TG1's previous attempt at this having failed. As a consequence we think decimal should be taken out of ES4.

In addition, implementing decimal may be a hardship for implementations on smaller systems and those that cannot use existing open-source libraries. We also have no practical experience with how to make decimal arithmetic fast in the context of the current `use decimal` pragma form.

Summing up...

In short, we think TG1 should go back to having a single primitive number type based on IEEE doubles, and a single wrapper for it. For future-proofing the single type could be called "double" (even if "number" would be more natural).

Operator overloading

(Premature; uncertain use; hardship?) Operator overloading is great if you're doing things that are a lot like math on data that are a lot like numbers but that are not built in to the language (complexes, fractions, intervals, matrices, bignums, bigfloats; the list is longer). We suspect a fairly small number of ES programmers write programs like that—maybe enough to warrant supporting overloading, maybe not. We are certainly not dismissing the idea.

---

[3] It's not even obvious that variables typed `boolean` should accept anything but boolean values.

TG1 has been through three overloading proposals so far:

- a receiver-based idea (from Michael);
- a static-method based idea (really a cleanup of Michael's idea); and
- a global generic operator function idea (Lars's idea, though not a very original one).

Global generic functions appear to be a good fit for the primary use case for overloading, but global values are highly non-modular. In practice, operator overloading tends to be abused for all sorts of non-math-like things and the non-modularity of global generic functions may be undesirable for that reason.

We think operator overloading should be taken out of ES4 and reconsidered later, based on a more thorough discussion about what the use cases for operator overloading are in ES/JS programming.

We are also concerned about how operator overloading can be made to work well, that is, how it can be extensible and useful in the absence of static type information without impacting the speed of built-in operators on predefined types. It appears that the current generic function design solves the problem (coupled with `final` predefined classes) but there is as of yet no hard evidence to support that.

Generic functions

(Premature; uncertain use?; hardship) Generic functions were added late and made it into the proposed language because they are neat, are well researched, and could be used to provide both operator overloading and overloaded methods.

Unfortunately, the need for operator overloading in ES4 is not clear, as we've seen above, and generic functions as currently realized may not support overloading the way it ought to be supported (because of the global nature of generic functions). John Resig has also expressed doubts about whether generic functions are right for method overloading at present, since generic functions can't be instance methods and don't allow for e.g. parameter lists of varying length, and we think he has a point.

Summing up the weaknesses of the current generic function proposal, we find:

- Not available on instances; and if they were available on instances then we would want to know how they interact with subclassing—can they be overridden, or extended in a subclass, or can one or the other be prevented?
- Not dispatching on structural types except unions, thus leaving out a significant part of the type system; fixing this requires at least experimentation and maybe research.
- We have no experience on how well they actually fit ES.
- The technology is somewhat heavyweight, arguably among the hardest to implement well in ES4.
- There are no provisions for local extensions of generic functions.
- The parameter list congruence rules are not particularly flexible.

Since we think TG1 needs to remove operator overloading from ES4, a large use case for generic functions also goes away, and that is probably the final nail in the coffin for generic functions in ES4. We should experiment with them and see how they are used, can be used, and what is lacking, before standardizing.

wrap

(Premature; uncertain use?; hardship) We think `like` has a lot of uses in typical ES programming, replacing current hand-written verification code. The case for `wrap` is much less clear. Essentially `wrap` exists to allow even module-internal interfaces to be typed without further use of `like`. If the common pattern for ES4 programs ends up being typed API / untyped code, then `wrap` won't see much use. Another use case for `wrap` is to create untyped or `like`-typed façades for strongly typed modules that are being integrated into a program, but Urs Hölzle's ECOOP '93 paper casts real doubt on whether this strategy can succeed.

Another concern is that wrappers, despite passing static type checks, don't guarantee any absence of run-time type errors because the wrapped object can change in arbitrary ways under the feet of the wrapper.

A third concern is that it is apparently hard to implement `wrap` in such a way that it will be efficient.

We think TG1 should defer `wrap` until later, when we might have a better understanding of how structural types are used in practice and will know more about the use cases that can't be met by ES4. As it is, `wrap` is just our hunch about that. (Lars observes that having tried to be honest about its shortcomings in the gradual typing tutorial, he is not sure he would use wrap for anything.)

Stack marks, Generators, Tail calls

The reason for grouping these three together is that they all preclude some common implementation techniques. As outlined in the following we also think they are of dubious utility—they feel unfinished to us. TG1 should rethink them for a later edition to make sure they are solid—this will also help reduce the implementation barrier for ES4.

Stack marks

(Uncertain use; hardship) The main strikes against stack marks are that (a) they're optional and (b) most of their use cases are not obviously applicable to ES. Clearly dynamic binding based on stack marks is more elegant than `try-finally` blocks with assignments to globals; neither is as elegant as a facility that hides the implementation details behind appropriate syntax (such as `PARAMETERIZE` in Scheme). Stack traces in exceptions and debugging are facilities applications want without the programmers having to think about them; having to implement a control inspector and rewrite the code—whether manually or automatically—to obtain a backtrace is not attractive.

Code generators can overcome most of those objections but not the fact that the facility is optional, and the optionality is itself a result of stack marks not being efficiently implementable in all environments (notably in hosted environments like .NET, from what we remember).

Generators

(Uncertain use; hardship) Generators in ES4 are weak because they are shallow; they are shallow because deep generators can't be mapped to all viable implementation strategies (some kind of limited continuation or coroutine or thread is required; a single-threaded tree-walking interpreter cannot implement deep generators in any simple way, nor can single-stack bytecode interpreters). No doubt there is some utility even to the shallow generators, but the use cases are in our opinion few. Most interesting uses for shallow generators can likely be handled by iterators equally well.

We understand there is some experience from Python. PEP255 defines generators like they are in ES4—they're shallow. But it's not clear to us that even the use cases outlined in PEP255 are handled by that design except in a superficial way.

A tree-walking interpreter must implement fairly elaborate backing-out and returning-in strategies even for shallow iterators. The mess is mostly localized, and clearly doable, but it raises the bar for implementation. Given that the utility of the shallow generators is limited, we feel that they are not worth the cost.

Tail calls

(Premature; uncertain use?; hardship) TG1 has discussed briefly the desirability of marking tail calls explicitly, thus paving the way for compile-time or run-time errors when the call can’t be executed as a tail call, but we have not reached agreement. (Lars suggested “goto” sort of as a joke, several people liked it, others hated it, and there it stands.)

A more serious point is that we can't avoid adding tail calls at some point if we cater to the functional programming crowd. However, if we really intend to cater to them then we need to provide data structures that are functional too, like lists and operations on them—unlike ES arrays, which are entirely imperative. Of course users can implement those data structures themselves, if they have tail calls, but right now just adding tail calls “for functional programming” seems like a half solution.

Finally, tail calls preclude the use of straightforward implementation techniques for procedure calls. To be sure they are less limiting than generators, as one-shot continuations or longjumps are sufficient to handle tail calls in a non-tail-calling implementation language, but implementations that want good-performance tail calls must necessarily switch to a code generation technique.

Nullability

(Premature; uncertain use) After looking at C# (which has value types), listening to Guy Steele talk on "Growing a language" (in which he advocates value types), thinking about Modula-3 (which has value types), and recognizing that ES4 has value types (`int`, `uint`, `double`, `decimal`, `string`, and `boolean`), it seems to us that non-nullable reference types may not be quite the right thing for ES4.

Our feeling now about the nullability discussion is that TG1 made some progress, but the progress that was made was based on the assumption that everything is a reference type, and the work that was done was centered on removing `null` from the legal value set for variables of that type. Yet while doing that TG1 introduced default values for built-in types but rejected methods for having default values of user-defined types.

All value types in ES4 are final, non-dynamic classes that have no mutable fields, absorb all sets and gets, are compared by their magic values (not by identity), and have default values. But they can't be defined on the user level because there is no way of providing default values (and if we remove operator overloading, no way of comparing them either). Yet there does not seem to be a good reason why the value types in ES4 should not be definable on the user level as instances of a general value type concept.

Lately there has been a little discussion about whether "`static meta function default() ...`" might not be a good mechanism for providing default values for a non-nullable or value type. TG1 rejected this in the past for performance reasons—fields of non-nullable type would end up being initialized twice, once automatically and then later by the program, and the generated initial value would be discarded—but this is not an issue if the object is a true, immutable value type, because the initial value could be the same every time. Though C# value types may not be natural for ES4, we think TG1 should consider value types more generally before committing to a design. In short, we think TG1 should do some more thinking here, and defer nullability and/or value types until a later edition.

Program units

(Premature; uncertain use) Two of the three stated goals of program units proposal speak to the use of ES4 in web browsers; only the declaring of dependencies between "compilation units" appears to be truly independent of host environment, and then only when unit names are globally unique in some sense (the locator is not used in determining if a unit has been loaded or not). Michael O'Brien has already observed that the locator is of no use to his implementation; he derives the location of the unit from the form of the unit name.

Web browsers benefit from units because they allow source files to be combined; to be loaded once; and it allows browsers to use strict mode because strict mode checking can commence once a unit has been completely loaded. Stand-alone environments will instead solve these problems outside the language, as part of the tool chain or in the loader.

It seems to us that this feature could benefit from more experimentation in diverse host environments before being standardized.

Reformed "with"

(Uncertain use; hardship?) The primary use case for reformed `with` is untyped code that migrates to strict mode. We have the impression that `with` is not heavily used in AS3 programs. We think TG1 should wait and see whether in fact code that migrates to strict mode has a lot of problems with `with`, and in the mean time prohibit `with` in strict mode.

This suggestion is motivated not so much by the semantics of the idea, as it is motivated by the fact that reformed `with` is the only statement in the language that introduces aliases, now that aliasing `import` has been removed. The complexity thereby introduced (in the compiler in sophisticated systems; at run-time in simpler systems) seems to outweigh the benefits of the statement, until we know how much the statement might in fact be needed.

Reformed "eval"

(Uncertain use) As in the case of reformed `with`, the operator form of `eval` can simply be outlawed in strict mode. We feel TG1's thinking comes up short on the problems of `eval`. For example, this program is legal in Proposed ES4 as far as we can tell:

```
use strict
var v: string;
function f()
{
  {
    use standard
    eval("var v:int = 27")
  }
  v = "foo"
}
```

yet there needs to be a run-time type error at the last assignment. Maybe the problem is the ability to turn off strict mode, or the problem is that strict mode really needs to affect the variable object, not the `eval` operator. In either case the current rule seems dubious.

Recall that there is no reason why the programmer can't use `global.eval` even in strict mode, so outlawing the `eval` operator (in the sense that the `eval` operator is illegal inside any strict scope, even inside nested scopes that turn strict mode off) does not make `eval` unavailable, only restricted—and not much more restricted than we already made it. That is not to say that it's trivial for the programmer to move from `eval(s)` to `global.eval(s)`, since the semantics are not the same, but it might be good enough. If it is not good enough, we will know.

Namespace filters

(Premature; uncertain use) We invented the types `NamespaceSet` and `NamespaceSetList` and the `namespaces` expression to be able to perform iteration and enumeration with namespace filtering, thus providing strictly more functionality than iterating/enumerating just public properties. The idea is that values of type `NamespaceSetList` can be passed explicitly to `iterator::get`.

It now seems to us that standard enumeration should just enumerate `public` properties always and that if the programmer needs more elaborate iteration or enumeration then he could provide custom iterators for that. Exactly how that should be done should be the subject of active experimentation.

Appendix: Future evolution

Though not strictly relevant in the context of this paper, we have the following thoughts on the future of ECMAScript and the evolution of the language.

We think ES needs to evolve in a more piecemeal fashion than we're seeing for ES4 so far. The fact that nine years will have passed from the publication of E262-3 to the publication of E262-4 is not in itself a valid reason to introduce a large number of new features at once; each feature must carry its weight, and experience must guide us. That said, this paper not advocating a watered-down "ES3.1" (which should really be called "ES3.01"); we are advocating that we go for the 80% solution "ES3.8" now and then plan to grow to meet new needs in the near future, when those needs are clearer.

The economic importance of ES is now so great that it will be unnatural for TG1 to evolve the language without relying on significant, direct, open input from the user community. Though the TG1 membership includes representatives for most important implementers it also includes representatives for important users or user groups, and some of the implementers represent large user groups as well. We believe the importance of the language will continue to increase, and we think that a future growth process must involve the community in the style of the Python and Java community processes. (Initial proposals for the ES process ought to be taken from among the features deferred from ES4—both those described here and those TG1 has deferred previously.)

The importance of ES, and the intractable problems of backward compatibility, also mean that future evolution will likely be in quite small increments, like the increments from JS1.5 to JS1.6, JS1.7, and JS1.8. We should not expect to make another leap like the one from ES3 to ES4.