Minutes of the: Ecma TC39-TG1

held in: Mountain View (Mozilla)

on: 25th May 2006

Attendees

- Francis Cheng, Adobe Systems
- Michael Daumling, Adobe Systems
- Jeff Dyer, Adobe Systems
- Gary Grossman, Adobe Systems
- Dan Smith, Adobe Systems
- Brendan Eich, Mozilla Foundation
- Dave Herman, Northeastern University
- Lars Thomas Hansen, Opera Software
- Cormac Flanagan, UC Santa Cruz

On phone:

- Graydon Hoare, Mozilla Foundation
- Blake Kaplan, Mozilla Foundation
- Edwin Smith, Adobe Systems

Agenda

- static vs dynamic / phases of compilation
- multiple compilation units
- `let` at top-level vs. incremental compilation
- review all proposals
- contracts
- formal type system
- reflection, or how first-class are types?
- stack inspection
- Date for Oslo meeting / discuss June meeting

Discussion

Phases Of Compilation / Static vs. Dynamic

- Lars wishes to ensure that parse/execution model is compatible with an ahead-of-time compilation solution. Goal is to pin down the semantics of the draft spec, and define invariants on the phases of compilation. For instance, constant folding always before type checking. Forward references can be assumed to be computed in earlier phase. Optimizations are possible, such as constant folding at runtime, but Lars wants to be able to know that his implementation is correct by matching the behavior of a canonical definition.
- The spec specifies that verifying is a separate phase, but doesn't specify when type checking, constant folding happen.
- Jeff figures that type checking and constant folding can be merged.

- Lars wants to know that A can come before B, but B cannot come before A.
- Where are compile-time constants required? Classes, interfaces, const, namespaces, type annotations.
- The spec currently states that you can write

```
const C = 3
const D = C
```

- and then D can be used as a compile-time constant wherever.
- Brendan asks: Is this legal?

```
function f(const T:Type, a:Array<T>)
```

- Lars: No, that should not be legal. It seems needlessly complicated.

```
function f<T>(a:Array<T>)
```

- Dave: It seems awfully intricate for the type checker to be dependent on the constant folding process.

```
class Foo { ... }
const x:Type = Foo;
new x (...)
```

- Dave would be happy giving this the type dynamic, not giving it the type Foo. One of the things that bothered him about Waldemar's spec was the multiple phases.
- Jeff: Is there a usability problem if we know that x is foo?
- Dave: If you have const x defined in some other module that someone provided you, it's hard to mentally understand what the type system is going to do without going through it in detail in your head. If the type system's rules are simple, then it's easy to see what is going to happen, and it's locally readable without having to mentally compile the whole program.
- Namespaces, there are statically known namespaces and dynamic ones. But that is not distinguished by the const keyword. Having namespace declarations inform the type system makes sense, but constant folding, not clear what the value to the user is.
- Lars: My system has no compilation phase; just parse and execute. No constant folding, but there is constant computation to do type checking at the point of function invocation.
- Dave: Constant evaluation vs. constant folding: Constant evaluation is looking at const declarations and determining what is a known constant. Constant folding is looking at 2+4 and determining that it can be replaced with 6.
- Lars: Right, that was a misstatement in the proposal, we're talking about constant evaluation here.
- Brendan: This doesn't change the binding of f, due to function hoisting:

```
alert(f)
function f() { ... }
alert (f)
var f;
if (f)
```

- Do you want to model that as a source transformation?
- const says that the value of a binding will not change once it is initialized, and you can't delete a const.
- A const field doesn't have to be initialized with a constant expression, and same with a const variable. It's a local promise within the scope of a const... a write-once variable.
- You can have forward references to const things, but they may not be initialized yet.
- Dave: Waldemar's spec had some notion of a true compile-time constant which basically came down to textual substitution; an efficiency hack. Is that possible now?

- Jeff: Yes. And it can be done in most cases. But that requires constant evaluation.
- Brendan: The following code prints undefined:

```
var x = y;
const y = y2;
print(x);
```

- It is not an error.
- Michael Daumling: This is a real-world example of incompatible behavior. With constant folding, the if statement will enter the else block.

```
function f() {
  if (typeof c == "undefined")
  ...
  else ...
}
f()
const c = 5;
f()
```

- Dave: Constant evaluation often doesn't seem worth it because to preserve backwards compatibility, the language can't do anything with the information anyway. I am distinguishing between things declared with "const", versus things like "namespace" and "class". It is a given that namespace and class needs to be looked at early, especially for things like mutual recursion. But const doesn't come with the same guarantees.
- Lars: Suppose a namespace is declared with a URI whose value is assembled using const declarations.
- Brendan: Is this what you're talking about?

```
namespace N = p.s
package p {
  ...
  const s = "foo"
}
```

- Lars: Package needs to be above.
- Brendan: There's no doubt, then, that that would have a definite value.
- Dave: What happens if the packages are in separate compilation units?
- Jeff: That is fine as long as you do last-minute verification. As long as the value is available at verification time.
- Cormac: Can we perhaps handle this with thunks?
- Lars: Spec mechanism, or implementation mechanism?
- Jeff: Waldemar had a fairly detailed algorithm in the section called compile-time constants, which could perhaps be a candidate proposal. I don't think we need to be as elaborate as he was. Why don't we leave this as, we need to pin this down soon, which I think was the point.

Multiple Compilation Units

In AS3, every compilation unit gets its own global object and implicit namespace. Toplevel code in a file goes into a file-specific namespace. Unless it is declared public, it would be hidden from outside code. There is an implicit package. It's not quite the same as just putting a block around the file due to hoisting behavior.

```
File 1
package p {
  public var x = 10
}
```

```
File 2
import p.x
print(x)

File 3
package p {
  public var y = x
}

Scope chain
...
F3 <- this.p
F2
F1
```

- The global objects of all other modules are under the current global object, and get searched for imported definitions.
- There are some quirks in the current AS3 implementation, like the global builtins are in a separate G global object by themselves. The global builtins are a separate program, essentially. So this.NaN is undefined, but NaN works. Jeff thinks we need to fix that.
- Brendan: I was hoping that we could achieve universal functioning of the "is" operator, even for types in different "trust domains" that are interoperating.
- Dave: If there were UUID's associated with type names, then we could be assured that types are truly nominal types.
- Gary: In AS3, there is an abstract (type) world and a concrete (object) world, and we instantiate the concrete objects from the abstract type information. But the abstract type information for the builtins is computed at initialization, and then we instantiate that same type information into concrete objects over and over again for each trust domain. But the "is" operator would work fine for "is Number", etc. between two trust domains that are interoperating because they share the same type description for Number.
- Brendan: That is important, we want types to be platonic.

```
class C {
  static public function g() {
    return f;
  }
}
```

- What goes on the right hand of an "is" is a type expression. "is" is used to refer to this world of abstract types, where this is only one copy shared across multiple windows, but not shared across virtual machines.
- Dave: There seems to be an ambiguity here where there could be a type expression or a value expression, if we are to know the type of the result of the new operator statically:

```
class Foo { ... }
const x:Type = Foo
new x (...)
```

- Brendan: The spec cannot be ambiguous on this point.
- Dave: One cannot use a structural type in the new operator, like

```
new {x:String}
```

- It's not a type expression. Although it does evaluate to an object.
- Brendan:

```
new f.<T>(a,b,c)
```

- Graydon had this example in his proposal; a sort function parameterized to a vector-of-int type:

```
g = f.<int>;
```

- In Edition 3, you can do new on lots of crazy things:

```
new g(a,b,c)
```

- Dave: New of any expression is allowed since any expression might turn into a function.
- Brendan: Making sure of one thing. You don't need parens on new:

```
new Date
```

- But you can:

```
new Date()
```

- And then if you add:

```
new Date().getTime()
```

- The precedence is:

```
(new Date()).getTime()
```

- Group confirmed that yes, we still preserve this in the E4 spec.
- Dave: When we have a new, if the thing that follows the new is an expression, then we don't know the type. If we recognize that what follows the new is a type expression, then we know the type of the result of the new operator. When you have a variable that is bound statically to a type, then when you see that expression, you know exactly how it will evaluate. But otherwise, in general, you don't know what type it will evaluate to. So there is a syntactic subset that you can reason about for the benefit of the type system.
- Jeff: Watch out, since * allows a free downcast, but Object does not.
- Brendan: From the AS3 folks, would they be happy with the new operator type being determined or not?
- Gary: They would be pretty insistent that they get errors when the type of a new mismatches with what it's being assigned to, so they would want the type to be determined wherever possible.
- Dave: It seems confusing that the user doesn't know, has to take on the additional cognitive load to see what the types are.
- Gary: We don't currently have const x:Type = Foo, but if we have it, then people would use it to alias types and want the type to be determined.
- Dave: But that's what "type A = B" is for; with const it's not as well-defined. With "type A = B" it is fine to do that.
- Brendan: OK, so this is a motion for simplicity in the type system.
- Lars: Type directives, defining a type T = Blah, the expression Blah needs to have the same rules for evaluation as constant evaluation. So, it doesn't change much.
- Brendan: You don't have hoisting, but you were saying about recursion...
- Lars: We could fix it so that you couldn't do recursive types, and only be able to refer to stuff previously typed.
- Graydon: That brings on the more general question, what are you going to do about recursive nominal types? You absolutely have to support a Node type that has a Node type as a child. So, I don't see why it can't be supported in the structural case.

- Dave: It's possible to do structural recursive types, but it's a more complicated algorithm. Newer Palsberg/Schwartzbach is O(n^2), original Amadio/Cardelli is O(exp(n)). The papers are complicated and difficult to understand, so you're asking a lot of the implementors. Also, when you're doing runtime checks, you're running a N^2 algorithm. The "is" operator could be a N^2 algorithm instead of a simple lookup a chain. My concerns are about efficiency and complexity of implementation. Obviously, you want recursive nominal types.
- Maybe we put in the spec a note that in the future, we'd allow recursive structural types.
- Graydon: Think real language use cases... if someone really needs a recursive type, it's not a big deal to go to recursive nominal types.

Concepts

- Default Package
- Global Object
- Default Namespace

```
window 1  +-- <html>              --+
          |     <script>            |
shared    |       var x = 10        |
global    |     </script>           |
shared    |     <script>            | different
default   |       print(x)          | globals
pkg       +--   </script>           |
                                    | different
          +-- </html>               | default
window 2  |   <html>                | package
          |     <script>            |
          |       print(window1.x)  |
          |     </script>           |
          +-- </html>             --+
```

- Gary: The stacking of global objects is a way of conceiving how AS3 is importing definitions, but we actually are doing more like an imported definition table for performance.
- Brendan: I would like to avoid a normative description of how this works that involves really deep scope chains.

Private and sealed packages

- Jeff: This is related somewhat to Lars's private package proposal.
- Also, the sealed package proposal. My feeling is we punt on "final" just because it becomes too constrained to be useful.
- Lars: You could allow it only on packages that have one source.
- Jeff: But is that going to be useful?
- Dave: We could leave it up to the implementations.
- Cormac: What's the motivation? Security?
- Brendan: Not so much security, since that's not a very comprehensive security mechanism, but it'd be one line of defense. Mostly it is about integrity of programming in the large.
- Private packages are not too controversial, but sealed packages poses some challenges, like imposing some rules on loading modules which we've tried to avoid in the spec effort thus far.
- So, final, sealed packages are out for now.
- But local packages are in.
- We entertained using the keyword "local", since we've been trying to avoid using "private" outside of classes. But, in the end, we concluded that "private" is OK for now.

Namespace shadowing

```
namespace debug
namespace release
debug function trace() { ... }
release function trace() { ... }
use namespace release
{
  use namespace debug
  trace("foo")
}
```

- Today in AS3: ambiguous reference error
- Desired: shadowing; more useful behavior

```
use namespace release
function foo() {
  var trace=10
}
{
  use namespace debug
  trace("foo")
}
```

- Brendan: This is the fix that I wanted when I made my namespaces slide, so thank you!
- Jeff: OK, noncontroversial, I guess.
- Jeff: That would work for imports, too, I guess.

Operators

- Daumling: Should we permit compound assignment operators to be overridden independently?
- Graydon: I think there's a solid argument for allowing them to be overridden separately.
- Brendan: Yes, definitely something like +=, you want to be able to override apart from +.

Public review

- We probably want to set up a listserv/usenet group. Talk pages could get overwhelmed... if you watch Wikipedia talk pages, it's a mess. A Wiki isn't as good as something built for conversation.
- We want the cream to rise to the top in our pitcher, not someone else's pitcher. Lambda the Ultimate can knock itself out but we should have our own forum.
- Graydon doesn't think DokuWiki is particularly well-suited to a large number of editors.
- We could do a Google Group or something.
- We need to move clarification issues off the front of the proposals namespace so that someone coming to this stuff fresh doesn't get a mouthful of clarification issues.
- ACTION ITEM: Graydon will do it later. (Graydon completed this.)
- Graydon mentioned this for exporting the site:

http://wiki.splitbrain.org/wiki:export

Review of the proposal list

- Clarification issues
  - dispatch_rules: Can move to resolved. Putting happy face.
  - code_mixing: Still an unhappy face.

- namespace_shadowing: Jeff working on it.
- abstract_syntax_tree: Broader question... do we even want to talk about the syntax of AST's in this context? Or is that implementation-specific? Lars thinks clear rules for verification, etc. could be written down if we have clear layout of AST. But it also falls under notation to some extent. We can use publication syntax like ALGOL 60 did. We can put this off for a little bit until we are working on notation.
- normative grammar: What does checkable syntax mean? It probably means different things to different people.
  - To Jeff, it meant having a parser available. But that may not be what others were thinking.
  - To Graydon, it means not publishing a spec with a grammar that has never been checked against real code. A surprisingly common feature of language specs is that if you implement it to the letter, it can't parse most programs... you have to tweak it until it works.
  - Dave has a friend from Utrecht who has a tool called SDF which will tell you if the grammar is a CFG.
  - Brendan doesn't think we're LL(2), and have some bottom-up issues. Narcissus switches into LR mode for operator precedence. It had 3 tokens of lookahead.
  - Lars: It is probably really hard to formalize what the grammar is, which can be a real problem.
  - Challenges: XML literals, regular expressions, unary plus, for...in
  - Brendan would like to use the "right" formalism.
  - Dave: It is hard to plug the JavaScript grammar into any formalism.
  - Lars: You need error productions in Bison just to do semicolon insertion. Pinning down rules for semicolon insertion would be a step forward. Jeff: Waldemar tried to tackle that.
- compact_profile: Lars thinks this has all been resolved.
  - Trying to avoid too much profile complexity. Reflection should be in a library. Can't throw out compiler.
  - We don't want compact profile to be a separate spec; want it to be part of the regular spec.
  - Lars says 21% of scripts use eval.
  - Gets a happy face.
- drop_traits: Think Graydon wanted it in some waste-bin. Graydon: Get rid of it!
  - Ed mapped out difficulty of mapping metaslots on to vtables. Traits is the way to go.
- lexical_scope: Was all Dave's bag of neuroses! Which have all been quelled or justified. (A few have been confirmed.)
  - Lars: We fixed a lot of these with use_static to kill some of the other problems.
  - Mostly, Dave was afraid that the places without lexical scope would destroy type soundness, but type dynamic is the escape hatch so he feels fine.
  - Smiley face.
- normative grammar: Jeff will put a pointer to the grammar he has been making, and that will get baked into the spec at some point.
- typeof: Jeff thought that Brendan's latest proposal was that typeof is broken, and let's leave it alone except for null.
  - We don't want to get into expanding typeof with a bunch of new strings... adding typeof xml for E4X probably didn't help.
  - It feels like int and uint should at least return typeof number?
  - For class and interface, we could return function for compatibility, but it feels weird.
  - Summary:
    - typeof(null) == "null"
    - typeof(int/uint) == "number"
    - typeof(class/interface) == "function"
- syntax_for_pragmas
  - Let's use the names strict and standard
  - Get rid of dynamic, get rid of dialect
  - "use default namespace" as means of steering which namespace definitions go into

- package_semantics
    - Package name with no dot is allowed in AS3
    - We should think about how things like Dojo today could migrate into Edition 4 namespace/packages 5 years from now.
- reserved_words
    - Prototyped not reserving identifiers after '.', '..' or initializers
    - Get error via error token and then "unreserve" it based on context in which it falls
    - Crockford's proposal would lead to people being able to say "var var". Brendan only wants to go as far as unreserving in property context.
    - We propose that after '.', you can use any reserved word.
    - 'goto' we will probably never do, contextually reserved word.
    - 'enum' probably will happen one day, and JScript.NET is close to what we want there
    - IE only reserves 4 Java words even though Edition 3 calls for pretty much all Java reserved words. SpiderMonkey backpedaled and unreserved most things, so one can say "var enum = 10"
    - Rhino and ExtendScript reserve everything in Edition 3, but they face not the onslaught that is the Web
    - Future reserved words is no more. We're moving goto, enum, and debugger into one of the two other reserved word categories. And making a comment in the spec on the '.' context, and property initializers.
- type_hierarchy_for_numbers
    - Conclusion is that int/uint should not be subtypes of Number
    - The flat model is easy to deal with and doesn't seem to get in the way
    - Think we're stuck with lowercase int and uint names
    - For double, decimal, probably lowercase as well for symmetry with our pragma identifiers
    - Use union type if you want to use "is" operator

Proposals

- is_as_to: Translation path from service dialect. Type annotations boil away in the core language, although leaving some residue. The annotated static dialect form translates to the dynamic form that the interpreter actually runs.
    - Some objections to wording
    - Graydon: We don't have two dialects; we have two translation paths from one dialect. The dialect always supports colon annotations.
    - The execution environment doesn't have anything like colon annotations, but has inserted code to assert/convert.
    - Lars: var x:T = y does not mean the same as var x:* = y, because one has a cast and the other does not, at least until the execution path is clarified completely.
    - Jeff: At runtime they mean the same thing.

Lars: The meaning of this is what precisely?

```
var x:T = e
```

Presumably, one meaning in both dialects. Yes, one meaning.

The question is when meaning is enforced.

These two have the same meaning:

```
var x:T = e
var x:T; x = e
```

The idea was to map : out of picture using operator to.

```
var x = e to T
var x; x = e to T
```

The bizarre thing is that the "to" operator goes with the assignment as opposed to the declaration.

Imagine you had something like the Edition 3 language and you wanted to convert something like this language to it. You'd have some "to" type helper, __convert.

Day Two

Nullability

- We decided that super() must be the first call in a constructor, and cannot contain any references to fields or methods of "this". This is needed for the soundness of the initializer system.
- When you construct a C, the first thing it does is run initializers and initialize those fields. Then super() is the first thing that happens in the constructor body. In the superclass, you then have the same process to initialize its fields, and so on recursively.
- Same as the way Java does it, and this is a sound system where no non-nullable field will ever be used before having a value.
- This requires a restriction on innitializer and super calls that they do not refer to "this" in any way.
- Could this be done for nullable fields? It could, but makes things more messy.
- Can one do this?

```
var x = 3; var y = x;
```

- Prior art for non-nullable types: C# has them, haXe might.

Non-nullable locals

- Scope of let is easy to understand, with var it's not as easy to understand. We might just not allow non-nullable vars. You'd have to use "let", the reformed var. Variable hoisting of "var" makes things a lot more complicated.
- A var in a function has a well-defined scope. But a var in a global object, may be hard to provide guarantees that it's not used before being referred to.

Complexity of nullability

- Dave: What if we did nullability in a more lightweight way, by having it essentially insert assertions? It becomes a mechanism for doing assertions everywhere very easily, where you would've had to do a null check in the past. Avoids the complexity of the static checking.
- Lars thought that contracts could only be used for enforcement on calls. Dave says that contracts work for fields, too. And locals?
- Brendan: One reason people will want the static check is to avoid the performance penalty of constant null checking.
- Dave: OK, but if we want static check, it has to be sound. The alternative is a syntax where you can express a null check with a single character.
- Brendan: Some of the same problems as const where you might refer to it before initialization and get undefined, due to hoisting.
- Lars: It seems broken that you could refer to an uninitialized const and not get an error.
- Brendan: We probably have an opportunity to make that an error.

- The problem with let, var and const is that you can use these things before initialization due to hoisting. If any are annotated with a non-nullable type, we propose that such uses before initialization are errors. (Let as an alternative to var is hoisted to the top of the block.)
- By the time you mention the name of another class, it has to be completely loaded.
- Brendan has no problem with excluding globals from non-nullable annotations.
- There may be builtins that use non-nullable types since they exist before the world starts, but user code can't do it.
- OK, so globals are out, but statics are in. Locals, we can live with and a reasonable compiler can do a good job.
- Jeff wants to confirm that this error is a runtime error, not compile time.
- Yes, the uninitialized error that you used it before initialization, that is a runtime error.
- Making it a compile time error, even optionally, would require specifying the data flow analysis in the spec.
- As soon as you say you can report the error early, you must be able to say "under what circumstances."
- Issues with "with", even reformed with:

```
with (o : {x:Complex!}) {
}

with (o : {x:Complex!}) {
  var x = 3
  function f(x) {
    if (b) {
      var x
    }
    print (x);
  }
}
```

- Can you ever say that this is an error?

```
with (o : {x:Complex!}) {
  var x = 3
  function f(x) {
    if (b) {
      var x : Complex! = c
    }
    print (x);
  }
}
```

- In here, x is not initialized unless b is true, so print(x) would be an error if b is false. A runtime error.
- At best you can make a conservative approximation, because there is no way to know whether b is true or false. In general, it is undecidable. But a compiler is permitted to be conservative.
- We want this program to run or not run on all systems.

Lvalues

- JScript in IE allows lvalues that are not Reference type in certain circumstances.

```
foo() = 10
```

- foo() returns a reference.
- The Edition 3 grammar does allow this syntax. We've carried forward the syntax, which may be overly general.
- Syntax error if the implementation does not support references.

- Would that make sense in some circumstances?

```
x.foo = 6
```

- How do you tell the difference between that and setter?
- Setter always wins.
- Jeff to start a proposal on tightening the grammar to not allow certain kinds of expressions as lvalues.
- ACTION: Jeff to write lvalue proposal
- A parenthesized expression can be an lvalue, but a comma expression is an rvalue and cannot appear on the lhs of an assignment.
- Behavior of the delete operator doesn't prove anything, as delete is a "big liar" in Edition 3.

```
delete (x, y)
```

doesn't do anything.

```
delete (y)
```

will actually delete y since parens are part of the lvalue grammar.

Numbers

Flat hierarchy. Types are double, decimal, int, uint. Number is an alias for whichever type is specified as the default.

Simple syntax for literals introduced to give literals a type.

- 10.5m is a decimal literal
- 10.5d is a double literal
- 37i is an int literal
- 42u is a uint literal

The "m" suffix is based on C#, not on IEEE. Mike noted that others have issues with the "m" suffix but Graydon says he didn't say he personally dislikes it.

Dave: What about

```
Number = 3
```

Not possible, we've made that read only.

Because of namespace shadowing, we can have a backwards compatible public binding for Number for Edition 3 code.

Weird syntax:

```
10.m.toString()
```

Well,

```
10..toString()
```

was already weird.

The rules are right, but weird code can be constructed. Most of the time people will have a named variable involved.

Not allow this:

```
0x10m
```

But we should allow:

```
0x10u
0x10i
```

What is

```
1.23i
```

It's an error! It's "1." abutted against "23i", or it's "1.23" followed by "i" which is also an error.

That tells Jeff that

```
0x5m
```

is also an error.

Discussing type conversion between different number types. There is no subtype relationship. Lars shows example:

```
function f( g : function (double):double )
f ( function (x:int) : int )
```

This is a type error in the strict language, because int and double have no common supertype except Object.

This is actually fine:

```
function f( g:function(int):* )
f( function (x:*):int )
```

Are these record types compatible?

```
{x:int} {x:double}
```

They aren't, because they're mutable.

It will get nasty with record types with functions inside them.

Two types having the subtype relationship and two types having the coercion relationship are not the same thing. In particular, coercions don't seem to bubble up through structural types, and there seem to be some cases where they should and some cases where they can't.

"to" conversion

We've found a bug in ":" as "to" conversion. It doesn't allow user-defined "to", because Bang doesn't respect the "to" conversion. If you want to write a program in Bang that does the conversion, there's no way to do that, like between Number and Complex. There are two concepts being encoded in one construct.

```
var c : Complex = 10
```

Bang would reject this no matter how you define Complex. It looks at the type lattice, and knows that 10 is not related to Complex. That's how it is detailed in the draft spec.

If you spoof the strict compiler to let this pass, then it will call Complex's function to.

```
var n = 10 var c : Complex = n
```

Free pass to numbers in the type lattice:

```
var n : int = "10"
```

But then we're not declaring numbers using the language anymore, which goes against our desires.

```
var s:String = "10" var n:int = s
```

Bang currently makes that an error, too.

Strict accepts a proper subset of the language. The question is, what subset?

There is no way to specify, in the language, what the behavior will be in Bang.

What would it look like?

```
Complex
  function to Complex (x:Object)
    :Strict(Numeric)
```

"to" called by returning, passing and assignment.

We need a strict compatibility lattice and a way to define it in the language.

If you allow programmers to write their own "to", you are inherently going to have an incoherent type system, since programmers can write their own incompatible conversions.

Even though this is defined this way:

```
class int {
  function to (x:*):int { ... }
}
```

This is a static type error in Bang:

```
var n:int = "10"
```

That seemed absurd until we observed that Strict is a subset of the language.

The question is, do we want there to be some way to do this in the strict mode? Is the subset too small a subset?

The "function to" conversion was introduced for two purposes, to be able to write the library in the language, and to describe conversions in a way more primitive than type annotations.

Jeff: If Bang doesn't respect "to", then the subset becomes too small. So let's add in a way for it to respect "to".

```
Complex {
  function to Complex(x:Object)
    use strict to (Numeric);
    ...
```

A pragma. We need type expressions to work in pragmas, now, we'll work it out.

Static and Dynamic

Reviewed strict and standard modes.

Meta-Objects

Reviewed meta_objects.

Contracts

Contract system in PLT Scheme is there in a way to make up for lack of a type system, so that makes its contract system more complicated.

For us, we can say that a contract is a refinement of a type, and that's it.

Blame annotations, blame assignment should be relatively easy.

Compiler may be able to do efficient implementation by augmenting function calls with additional blame information. Otherwise, implementation using proxies may degrade performance.

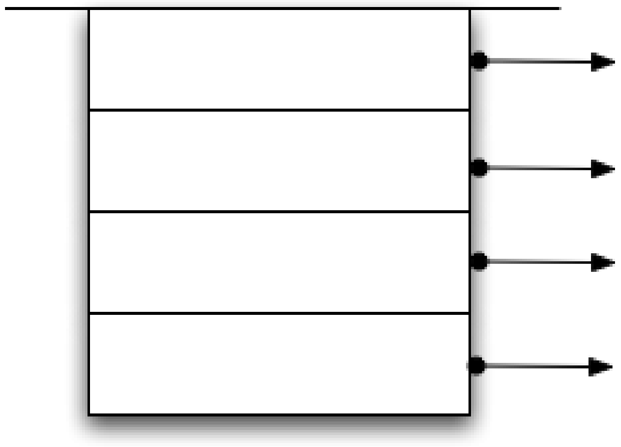
May need debugging helpers that are also at schedule risk, like source coordinates.

Stack Inspection

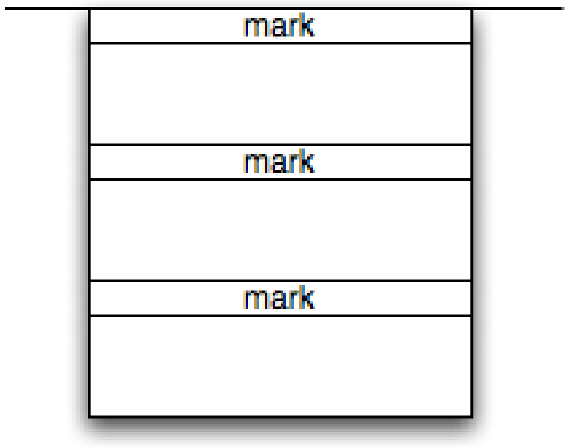
Build a way of saving information on the stack.

Here are permissions I have so far, can overwrite with new permissions depending on your particular algebra of security restrictions.

If every time you make a function call, you save information on the stack about function call you made, and you're trying to do proper tail calls, you end up saving information on the stack with each tail call which nullifies the goal of proper tail calls.



Another strategy is to put marks above the current frame. This is less problematic because you can do a proper tail call without clobbering the marks above the current stack frame.



Parent activation object could have a slot in it, and save in there.

Or save in a separate stack.

These are implementation details but just way Dave conceives mentally of it.

We could select granularity of marks at level we want. Probably most implementations would want it to be at level of procedure activations.

“Try” can be implemented in terms of continuation marks in the abstract, although there are some obstacles for this in JavaScript. One could go to as fine granularity as having every curly brace block introduce a new frame.

API for stack inspection:

```
class ControlInspector {
  annotate
  getCurrentAnnotation
  getAnnotations
}
```

Happy face OK on stack inspection proposal.

Quick review of "!" proposals

Iterators and generators: Brendan to rewrite.

Multiple compilation units: Jeff taking ownership.

Binding of "this":

```
function f() {
  print(this)
  function g() {
    print(this)
  }
  g()
  return g
}

h = f() // prints "global" 2x
o = {m:f}
k = o.m() // prints "o" 2x

function f
o = { m:f }
o.m()
^
|
+-- this

f()
^
|
+-- global
```

All proposals need to be finished up by Wednesday deadline or should be moved to deferred section.

The spec namespace needs a bunch of corrections; Jeff and Francis to work on it.

Meeting on Wednesday, perhaps go live with materials on Friday.

June 1 has been stated deadline so should aim for "goodwill" of delivering in a predictable fashion.

Unicode: Lars will be in Bangkok next week but will try to make progress.

Extend Regex: All agreeable. Brendan is generally happy with what's here but doesn't seem like enough. But lack of useful character classes. However, that introduces a lot of complexity (composition, case folding, etc.) so it might be Hard. Brendan was interested in a "/e" RegEx switch which let direct tap into the NFA, but too much to bite off. Brendan will own this one and make a pass at it.

Date: getTimezoneOffset is insufficient, you can't actually determine your time zone in any standard way although often people parse the result of Date.toString to get it.

```
a[i:j] = b
s[i:j]
s.slice(i, j)
a.splice(i, j-i, b)
```

slice is OK. It is awkward that we have this splice which is Perl-ish instead of Pythonic.

```
interface I { }
I.toString()
var v = I
frob(v)

type T=U

o = { p:42, q:true }
o.toString

type U=(A,B,C)
U.toString()
```

You can say

```
function f(A) { return new A }
f(Date)
f(f)
```

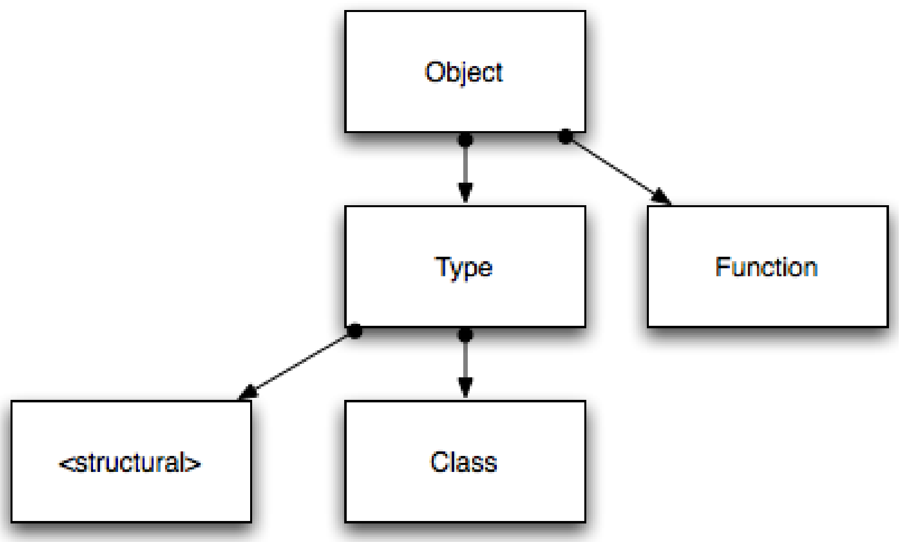
But can you say this, with a structural type?

```
return new { construct:... }
```

Consequences for eval.

```
function frob (T:Type) {
  eval(before + T.toString + after)
}
```

It looks like we need a Type type, which would be the supertype of Class.



http://pugscode.org/images/metamodel.png

Graydon posting a new illustration under meta_objects.

Next Meeting

Some haggling over dates... Planning tentatively on meeting in Oslo on July 28, 29.