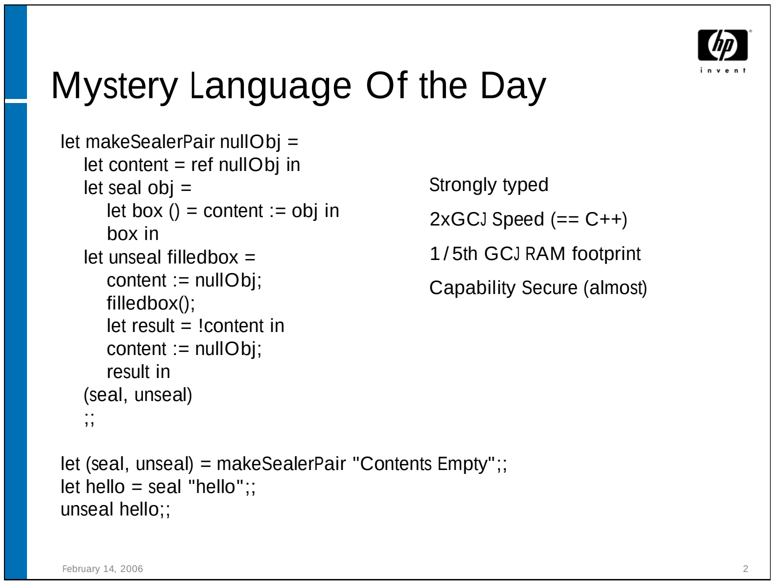
Emily: A High Performance Language For Secure Cooperation

Marc Stiegler
Visiting Scholar, HP



Mystery Language Of the Day

```
let makeSealerPair nullObj =
    let content = ref nullObj in
    let seal obj =
        let box () = content := obj in
        box in
    let unseal filledbox =
        content := nullObj;
        filledbox();
        let result = !content in
        content := nullObj;
        result in
    (seal, unseal)
    ;;

let (seal, unseal) = makeSealerPair "Contents Empty";;
let hello = seal "hello";;
unseal hello;;
```

Strongly typed

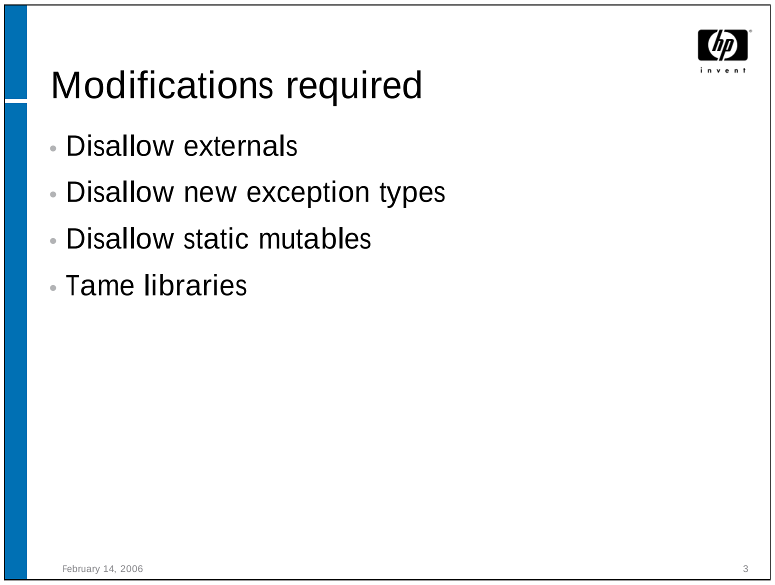
2xGCJ Speed (== C++)

1/5th GCJ RAM footprint

Capability Secure (almost)

February 14, 2006 2

- Recently, while investigating ML-style languages, I learned that OCaml has performance characteristics comparable to those of C++ (see the Great Language Shootout), that it had a relatively small footprint, that the designers had treated imperative-style programming like an important part of programming (thus running against the standard set of prejudices found among functional programming language designers), and most amazingly of all, that it was very nearly an object-capability language already. The code on this page implements the standard sealer/unsealer pair, compactly and understandably, in a fashion quite similar to the way E implements such a pair.
- The similarity to E is more amazing because OCaml is a strongly statically typed language. Usually (in Java/Joe-E, for example) static typing makes it difficult to express such general purpose patterns without mountains of declarations. But since OCaml does its type analysis inferentially, it is often invisible, just as the security is often invisible in object-cap languages. Since the programmer does not have to spend so much time and code writing type declarations, the code is much more compact, and therefore (if one's accepts the results of Fred Brooks and those who have followed him, that the primary driver of programmer productivity is the number of lines of code he must write) programmers are quite productive with this language (compared to java and C++, for example).

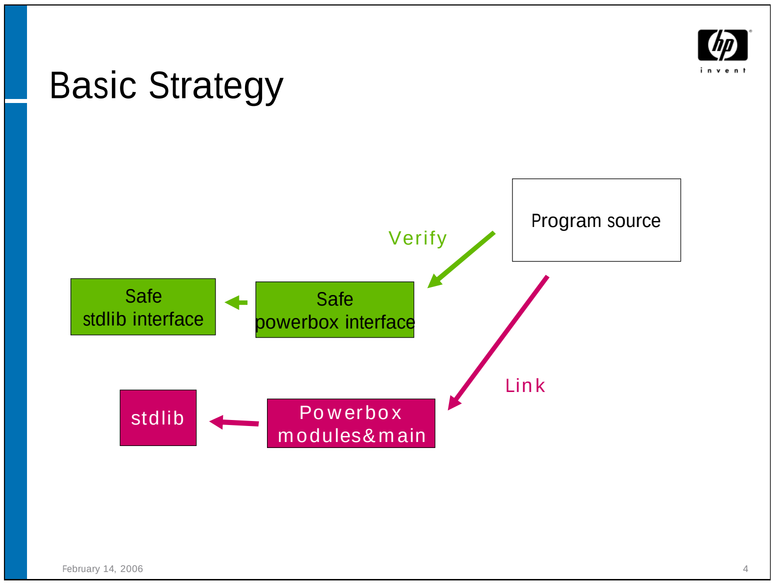


Modifications required

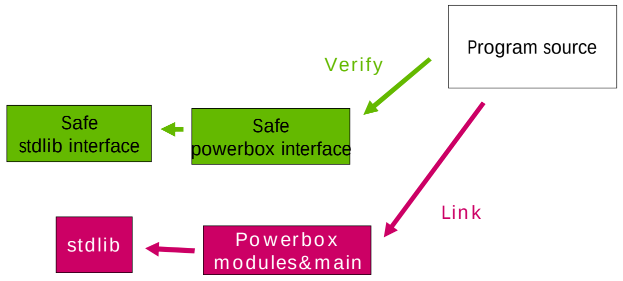
- Disallow externals
- Disallow new exception types
- Disallow static mutables
- Tame libraries

February 14, 2006 3

- To transform OCaml into the object-capability language Emily, one needs only make a handful of changes:
- The keyword “external”, which enables linking to arbitrary binary, must be forbidden
- The keyword “exception”, which enables the construction of arbitrary new exception types that can be used to leak authority, must be forbidden.
- Static mutable state, which can be embodied in OCaml by putting a reference at the top level of a module, must be forbidden.
- The libraries must be “tamed”. Which is to say, the libraries must be reorganized in such a fashion that the authorities embodied in the libraries can be presented to programs in an object-capability disciplined fashion. Functions that convey authority, such as the “open_out” function in the Pervasives module, which takes a string and returns a channel to a file, must not be directly reachable by the application, but must be brought into the scope of the specific objects that need such power and no others.

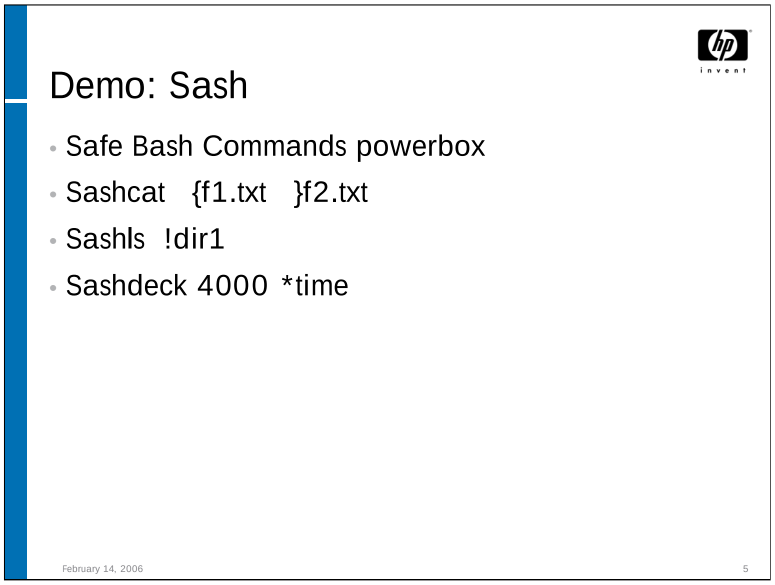


Basic Strategy



February 14, 2006 4

- The basic strategy for Emily is to combine the standard OCaml compiler with a verifier to create an Emily compiler. The Emily compiler follows the following steps:
- The programmer feeds Emily a powerbox folder and a program folder. The powerbox fills the position normally considered to be the position of the "main" function in C or Java. This powerbox is the part of the executable that is activated at launch, and holds full authority, i.e., it is a part of the user's TCB. The powerbox may include some interfaces that the program itself will use, and the powerbox folder includes descriptions of where to look for the safe version of the standard library interface (the safe version of the stdlib is a set of the interfaces --*.mli files -- to those parts of the stdlib that convey no authority).
- The Emily verifier examines all the source files in the program folder and confirms that basic rules of object-capability design are not violated (no "external", "exception", or static mutable state).
- The Emily verifier/compiler then runs the compiles the program sources (using the Ocaml compiler) against the safe version of the standard library and the safe version of the elements of the powerbox. If this compile fails, the program is rejected.
- If the program passes this compilation, then we know that it uses no excess authority, and we compile a second time (again with the Ocaml compiler) with the real stdlib and the real powerbox. The result is linked into an executable. Emily at this time only works with the Ocaml binary compiler, it does not work with the Ocaml byte code virtual machine.

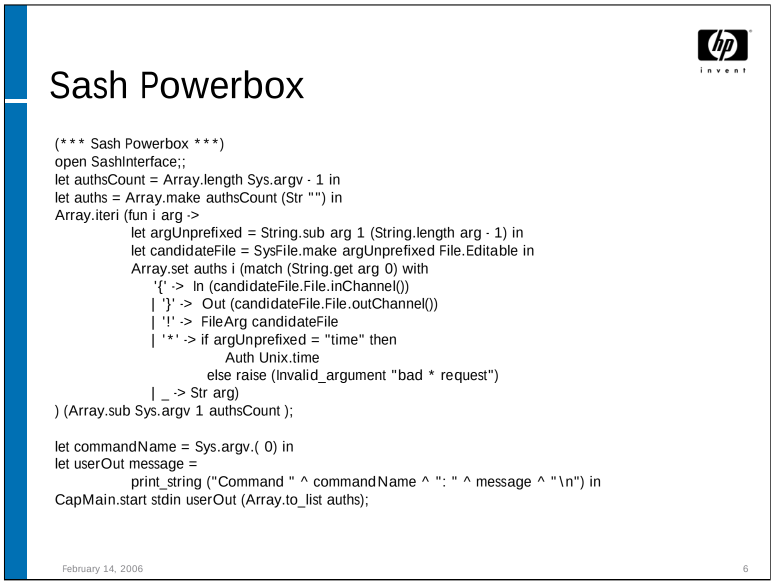


Demo: Sash

- Safe Bash Commands powerbox
- Sashcat {f1.txt }f2.txt
- Sashls !dir1
- Sashdeck 4000 *time

February 14, 2006 5

- To prove that our Emily compiler/verifier was working correctly, we built a simple powerbox for implementing bash commands, and wrote three applications that used this powerbox. The powerbox is called “sash” (for “safe shell commands”). We implemented a cat program (sashcat), an ls program, and a card deck shuffling program.
- The syntax for using sash is a little funky at this time. The powerbox interprets the following special characters:
- If an argument begins with “{“, sash creates an in_channel for the named file.
- If an argument begins with “}”, sash creates an out_channel.
- If an argument begins with “!”, sash creates a read/write file authority. For this powerbox we wrote a wrapper for the file system that works much like the E file system (which in turn is similar to the Java io.File system, but it follows capability discipline).
- If an argument begins with “*”, the powerbox interprets the rest of the argument as a special power to be conferred. Sash only recognizes *time as such a power, and grants a read-only authority on the system clock.
- This syntax and these powers are for demonstration purposes only. Eventually, one would like to see a real sash with a better syntax that embodied a more flexible set of power-granting possibilities. But sash has the demonstration merit that the whole powerbox can be put on a single powerpoint slide.



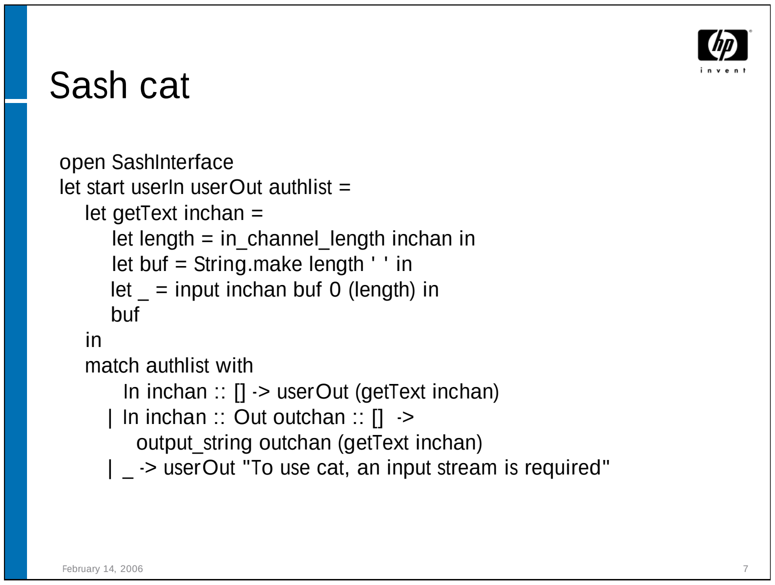
Sash Powerbox

```
(*** Sash Powerbox ***)
open SashInterface;;
let authsCount = Array.length Sys.argv - 1 in
let auths = Array.make authsCount (Str "") in
Array.iteri (fun i arg ->
        let argUnprefixed = String.sub arg 1 (String.length arg - 1) in
        let candidateFile = SysFile.make argUnprefixed File.Editable in
        Array.set auths i (match (String.get arg 0) with
            '{' ->  In (candidateFile.File.inChannel())
           | '}' ->  Out (candidateFile.File.outChannel())
           | '!' ->  FileArg candidateFile
           | '*' -> if argUnprefixed = "time" then
                        Auth Unix.time
                     else raise (Invalid_argument "bad * request")
           | _ -> Str arg)
) (Array.sub Sys.argv 1 authsCount );

let commandName = Sys.argv.( 0) in
let userOut message =
        print_string ("Command " ^ commandName ^ ": " ^ message ^ "\n") in
CapMain.start stdin userOut (Array.to_list auths);
```

February 14, 2006 6

• Emily and the sash powerbox supply a compact example of all the different concerns one needs to address in building and using an object-capability language. Sash, for example, incorporate a petname system: sash applications are granted stdout writing authority by default, but when the app writes to stdout, every print operation is prepended with the name of the executable typed by the user to invoke the application (thus making it difficult for applications to fool users into thinking they are some other application). This is in effect a true petname, the user types this name to invoke the program, and the program uses this name to tell the user about its actions.

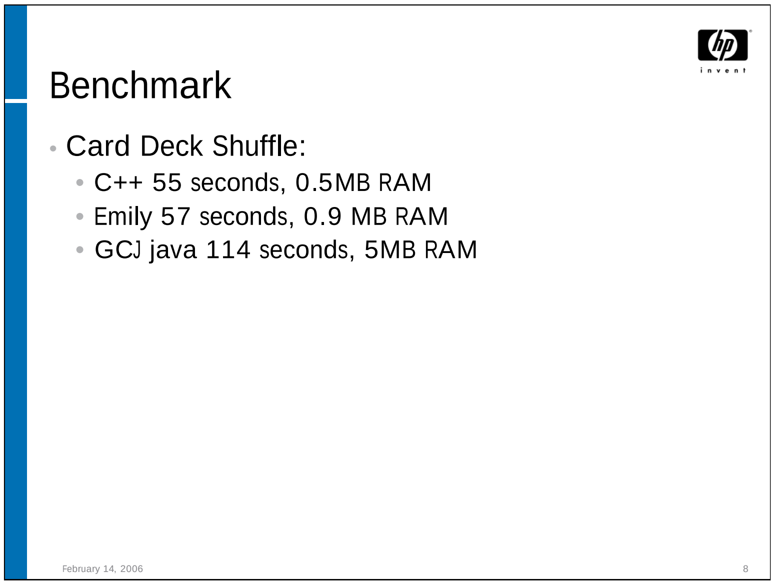


Sash cat

```
open SashInterface
let start userIn userOut authlist =
    let getText inchan =
        let length = in_channel_length inchan in
        let buf = String.make length ' ' in
        let _ = input inchan buf 0 (length) in
        buf
    in
    match authlist with
          In inchan :: [] -> userOut (getText inchan)
       | In inchan :: Out outchan :: []  ->
            output_string outchan (getText inchan)
       | _ -> userOut "To use cat, an input stream is required"
```

February 14, 2006 7

- While the sash powerbox is small, the demo powerbox apps are even shorter. This is the “cat” app, doing what one would expect cat to do: given an in_stream and an out_stream, it copies the instream to the outstream.

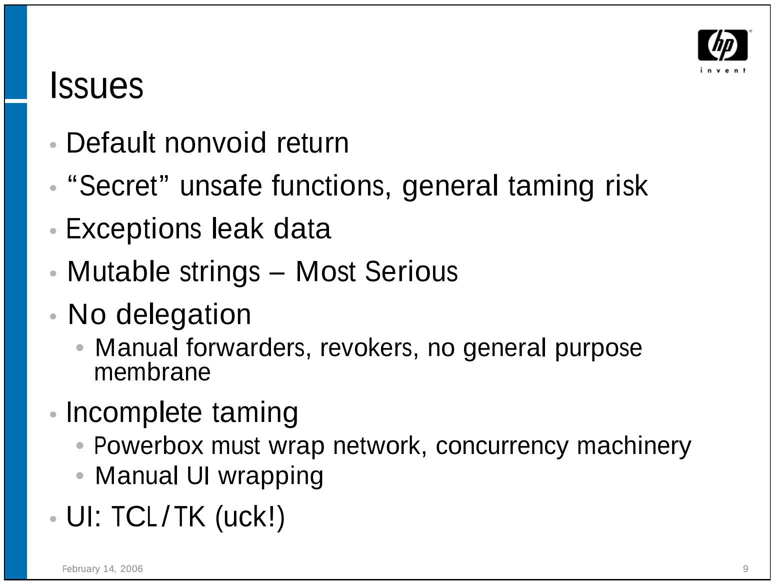


Benchmark

- Card Deck Shuffle:
  - C++ 55 seconds, 0.5MB RAM
  - Emily 57 seconds, 0.9 MB RAM
  - GCJ java 114 seconds, 5MB RAM

February 14, 2006 8

- To demonstrate that Emily loses none of the performance advantages of OCaml, we ran a very small benchmark, shuffling 4000 decks of cards 4000 times each. The code that implements the algorithm is essentially identical in Emily and OCaml – in general, computing problems that are compute-bound need little or no authority, and so Emily's verification will usually accept such code without modification.
- The results confirmed the high performance quality of the output. The Microsoft Visual Studio 2003 C++ compiler runs this benchmark in 55 seconds, Emily runs it in 57 seconds. This tends to surprise people who believe that array bounds checking and garbage collection must be expensive (both of which are parts of Emily and OCaml), but is less of a surprise to those with intimate knowledge of modern incarnations of such machinery.

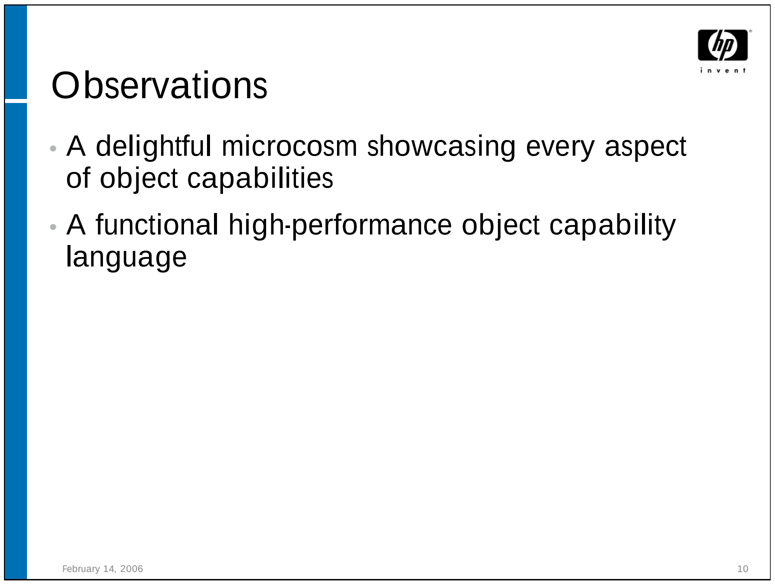


Issues

- Default nonvoid return
- “Secret” unsafe functions, general taming risk
- Exceptions leak data
- Mutable strings – Most Serious
- No delegation
  - Manual forwarders, revokers, no general purpose membrane
- Incomplete taming
  - Powerbox must wrap network, concurrency machinery
  - Manual UI wrapping
- UI: TCL/TK (uck!)

February 14, 2006 9

- While Emily is fleshed out well enough so that people who need a high-performance language for software that needs to remain robust in the presence of attacks, there are a number of limitations of which one should be aware.
- In the early days of development of the E language, we discovered that the functional style of syntax, that always returns whatever value happens to have been last produced, can be quite dangerous in an object-capability world – in E in particular, even the prime language originator had trouble accidentally leaking authority because it happened to be the result of the last evaluation of an expression. This functional style can be thought of as leaking by default, a poor choice of design when secure cooperation is a goal.
- Experience with Emily so far suggests that this is not as worrisome a hazard in Emily as it was in early E (modern versions of E require an explicit return from a function or method). There are several reasons for this, all details of the language design that cannot individually explain the difference. For one thing, the inferential type checking forces the programmer to be aware, much more explicitly under many circumstances, of the return value signature than he is in E (which has dynamic typing, not static typing). Under many circumstances, if the function is returning an unanticipate value, the compiler will deliver either a warning or an error. Also, module interfaces require explicit declaration of the return types across modules, so this also requires that the developer of a module have clear thoughts about the types he is returning. Whether Emily protects adequately from the authority leaking hazards of functional programming style will have to be investigated on an ongoing basis as Emily is used in projects larger than the sash command line example.
- The in-depth review of the security issues with the E programming language clearly showed that taming existing libraries is inherently a risky strategy. This risk also surfaces in Emily. The risk is mitigated by a couple of factors:
  - The libraries are small, and do not have deeply nested hierarchies of classes, which were two of the most serious risk factors that left the taming of the Java API for E with significant vulnerabilities.
  - Current Emily is headless, i.e., there is no graphical user interface kit that has been tamed. OCaml ships by default with TCL/TK, limited inspection suggests that it should be relatively low risk to tame (compared to Swing, the gui reviewed in the E security review). However, TK is a rather limited gui kit, and taming GTK may be more appropriate. For the moment, authors of powerboxes will have to wrap those pieces of the gui system that they want to grant to their target applications.
- Exceptions can leak data, though they cannot leak authority. This is not technically a violation of capability discipline because the programmer can in principle handle exceptions in such a fashion that leaks cannot occur, but it is a hazard that the programmer must be aware of if the leakage of data across modules is part of his threat model.
- The most serious hazard in current Emily is that, in OCaml, strings are mutable. This is a significant hazard: programmers rarely use string in a mutable fashion, which means that, in the programmer’s mental model of operations, strings are often considered immutable, with lamentable results. Limited inspection suggests that shutting off all string mutation operations should be straightforward, but with the result that Emily will be more surprising for the OCaml programmer – in particular, the way one gets data from an in_channel is typically to pass in a string to mutate. So the in_channel read operations will have to be wrapped in an object-capability fashion.
- Another issue with Emily is that the strong data typing can interfere with the construction of many of the standard secure cooperation patterns used by object-capability programmers. It is not possible to create a general-purpose maker of revocable forwarders, for example. However, for high-performance elements of an application system, this may be acceptable.



Observations

- A delightful microcosm showcasing every aspect of object capabilities
- A functional high-performance object capability language

February 14, 2006 10

- In conclusion, Emily fulfills 2 goals:
  - It is a demonstration that imposing object-cap discipline on an existing language does not have to be an excruciating process, if the underlying language is well-formed
  - It is a potentially valuable productivity tool. There are many applications that must be robust against attack, that must be written by a third party for a user who should not be required to trust totally the development team, and which must still meet strong performance goals. Web browsers are an interesting example of this: users of a web browser should not be required to give the web browser full user authority (though all current browsers expect it). Web browsers must be able to survive in a hostile environment without being patched (which no current browser can do). And yet, web browsers, particularly their rendering engines, must run at extreme speeds to avoid a sense of sluggishness in the user experience. Emily would be an excellent language in which to build such a next-generation browser.