A Survey of Mechanisms for Language Extensibility

BRYAN C. MILLS
Carnegie Mellon University
bmills@andrew.cmu.edu

We consider the problem of designing a language to allow incorporation of new features into itself from user source code, including combination with other languages. With this goal, we offer several major features that would contribute to such a language, and survey some works addressing these features. We conclude by suggesting aspects of the system that may require further research.

1. INTRODUCTION

1.1 Multi-language systems

In traditional programming language design, one starts with a particular domain and carefully crafts a stand-alone language that elegantly handles the common tasks of that domain; the purpose of a language is often central to its design. For example, Java is oriented to the single task of representing a set of objects and their interactions; SQL is highly specialized to the task of database queries, Perl is known for its straightforward text-processing capabilities, and HTML is a widespread standard for describing document markup. Other languages (such as ML and Haskell) emphasize designs based on fundamental logical structure and mathematical reasoning.

Realistic applications are, unfortunately, not so strongly confined. Many web-hosting systems (such as PHP and ASP) combine scripting, text-processing, and database queries with output in HTML and XML; mathematical languages require input, output, and access methods for data-sets, and object-oriented languages interface with functional or imperative operating system libraries. A typical approach to these combinations provides hooks from the parent language into an interpreter or library for the alternate domain; however, these hooks often do not preserve important properties of the languages involved, and may destroy the syntactic cleanliness of one or both languages.

In a combined system, performance and maintainability often suffer dramatically; programs in the combined language may require several passes through different compilers and interpreters, and may entail insertion of (otherwise unnecessary) string manipulation code or other confusing delimiters to separate "host" code from "client" code. Moreover, static type-checking nears impossibility when the interface between languages is based entirely on strings or when the compiler may only consider fragments of code in a single language at a time.

1.2 Language extensions

One approach to the multiple-language combination problem is as a special case of a more general problem: language extensibility. That is, given an existing language and a set of features, how does one incorporate those features into the language? In this formulation, we can then re-phrase the question "How may we combine language X with language Y?" as "How do we extend language X with the features of language Y?"

Recent trends in commercial programming languages have shown a strong trend toward language extension. Java 5.0, for example, adds a number of features not present in Java 1.4, such as generics, a new form of loop, and argument lists. Microsoft's C# 2.0 has similar new features; a number of other Microsoft languages have recently gained additional features and syntax for .NET compatibility. Even some of the more theoretical languages have major extensions; Objective Caml, for instance, has (as of version 3.08) at least nine features not present in the O'Caml specification. The popular New Jersey implementation of Standard ML, another ML dialect, has at least four syntactic extensions.

While extensions are often useful as programming and research tools, they are frequently added haphazardly to the original language, breaking compatibility with existing programs and future standards. In the typical case, the language has no well-defined means to determine whether any given implementation implements a particular extension; almost all languages lack any mechanism to determine which extensions a given program requires. A very small number of existing languages (in which O'Caml is included) provide integrated support for adding small syntactic extensions to the base language; however, even these systems do not provide for the kinds of sophisticated type rules that practical extensions may require.

1.3 An idealized goal: in-code extension

We would, ideally, like to be able to describe a new language feature – its syntax, type structure, and semantics – and to immediately begin using that feature in the current program, much as a lecturer makes a definition to her audience and proceeds to use the defined term throughout the rest of the lecture. Designing such a language turns out to be a tremendously difficult task with a number of yet-unresolved issues; to that end, we will consider features and patterns that support, exemplify, and otherwise relate to our desired approach to extensible programming, with the hope that better understanding of the relationship between the extensible programming problem and current materials will help further progress toward a truly extensible programming language.

2. MONADS

2.1 Monads and programming

According to Wadler [Wadler95], a monad is a triple $(M, unit, \star)$ (where M is a type constructor, $unit$ is a function of type $\tau \rightarrow \tau M$, and $\star$ (called "bind") is a function of type $\tau M \rightarrow (\tau \rightarrow \tau' M) \rightarrow \tau' M$) satisfying the following properties:

—left unit: $unit\ e_1 \star \lambda x.e_2 = [e_1/x]e_2$

—right unit: $e \star \lambda x.unit\ x = e$

—associativity: $(e_1 \star \lambda x.e_2) \star \lambda y.e_3 = e_1 \star (\lambda x.e_2 \star \lambda y.e_3)$, provided that x does not occur free in e_3

Monads are used extensively in "pure" functional programming to simulate features of "impure" languages, such as console interaction, state, and exceptions.

In [Jones93], Jones and Wadler remark on several merits of the monad-based approach. In particular, monads are highly modular (easily composed and extended), efficient (straightforward to optimize), and relatively simple (easily expressed using a Hindley-Milner type system). It is modularity and simplicity with which we are chiefly concerned, as these attributes make monads especially relevant to extensible programming.

2.2 Monads and extensibility

Wadler presents [Wadler95] a monadic evaluator for a small fragment of arithmetic (shown here in ML instead of the original Haskell):

```
fun eval (Con a) = unit a
|   eval (Div (t,u)) = eval t * (fn a =>
                       eval u * (fn b =>
                       unit (a div b)))
```

By varying the definitions of *unit* and $\star$ to carry exception status, state, or other effects, this evaluator can be easily extended with features similar to those of an impure language. The method used generalizes to a pattern for writing impure computations as sequences of pure monadic computations; [Wadler93] suggests that monads, used properly, are equivalent in power to alternate mechanisms involving CPS transformation, another abstraction commonly used for implementing explicit sequencing and stateful operations. In certain styles of implementation (namely, those which utilize a fast execution stack), the monadic implementation results in more efficient generated code.

In a multi-language setting, one might use several monads in combination to propagate effects and state from one language across manipulations in another. For example, given a monad for array manipulation, one could use a double monad ($\tau\ M\ M$) with a careful binding structure to use two arrays in tandem; similarly, a monad for exceptions could be combined with a monad for I/O to yield a program structure that supports I/O with exceptions. In this way, one can properly control sequencing between languages, allowing the intermixing of strict and lazy languages and multiple "virtual machines" while sacrificing neither soundness nor optimizability.

The fact that monads can be implemented within the Hindley-Milner type system provides an additional advantage for an extensible language: type inference for monadic programs is decideable, and can be done using the existing type inference system of an ML-like language.

2.3 I/O monads

One typical Haskell monad is used for console input and output. A simple IO monad can be written functionally as follows [Wadler95]:

```
type 'a IO = (string * 'a)
fun unit a = ("", a)
infix *
fun m * k =
  let val (s1, a) = m
      val (s2, b) = k a
  in (s1 ^ s2, b)
  end
fun runIO (m : 'a IO) : 'a =
  let val (s, a) = m
  in (print s; a)
  end
```

This implementation would rely heavily on lazy evaluation to print strings in real-time; without laziness, all program output would be delayed until the program terminates (if it does). Delayed output may be useful for some applications, but it is typically not useful for interactive programs, and may lead to unnecessarily high memory usage in intermediate stages of computation.

2.4 State monads

Monads can also be used for stateful computation, such as mutable references and the more general case of in-place array update. First we look at the pure implementation for references; due to type-checking considerations, this implementation allows only one type of state at a time per monad; one could presumably write an algebraic tag datatype to allow the state to store arbitrarily many different types of reference. We let the type `('a, 'c) ST` be a state monad in which all references have type `'c Ref`. While this seems horribly restrictive, we must bear in mind that we have only restricted the types of references available in a single monad, not the number of monads we can create. With a few clever calls to `unit` and `runST`, one can utilize multiple types of reference – each call to `unit` produces *a new monad*, and each call to `runST` converts a monad back to a value. The catch – and it is quite a snag – is that runST is not quite sound! It may only be run on code whose free references are all bound in the current monad. Carefully constructed code ought to work properly, but this additional constraint means that these references do not have the same semantics as ordinary references, and more unfortunately that the ordinary monad type system with runST is insufficient to simulate references using a pure functional implementation. Jones & Wadler remark on similar problems with asynchronous IO in the IO monad.

```
signature PUREREF =
sig
  type ('a, 'c) ST
  val unit : 'a -> ('a, 'c) ST
  val * : ('a, 'c) ST * ('a -> ('b, 'c) ST) -> ('b, 'c) ST
  val runST : ('a, 'c) ST -> 'a
  type 'a Ref
  val new : 'a -> ('a Ref, 'a) ST
  val set : 'a Ref -> 'a -> (unit, 'a) ST
  val get : 'a Ref -> ('a, 'a) ST
end
structure PureRef :> PUREREF =
struct
  type ('a, 'c) ST = 'c Vector.vector -> ('a * ('c Vector.vector))
  type 'a Ref = int
  fun unit a = fn s => (a, s)
  fun m * k = fn x =>
    let val (a, y) = m x
        val (b, z) = k a y
    in (b, z)
    end
  fun runST m = let val (a, v) = m (Vector.fromList []) in a end
  fun new a v = (Vector.length v, Vector.concat [v, Vector.fromList [a]])
  fun set r x v = ((), Vector.update (v, r, x))
  fun get r v = (Vector.sub (v, r), v)
end
(* sequencing two monads to obtain two different types of references *)
runST ((new true)
      *(fn b => ((get b)
      *(fn b' => unit (runST ((new "bag")
      *(fn s => ((if b' then (set s "foo") else (set s "bar"))
      *(fn () => (get s))
         ))
        )))
       ))
      );
```

Now we observe the mutative implementation of the reference monad, used as a means of ”purifying” an already impure language. That is, the monad makes explicit the sequencing of evaluation required for correct reference behavior; the compiler is free to optimize any unaffected code.

```
signature REF =
sig
  type 'a ST
  val unit : 'a -> 'a ST
  val * : 'a ST * ('a -> 'b ST) -> 'b ST
  val runST : 'a ST -> 'a
  type 'a Ref
  val new : 'a -> ('a Ref) ST
  val set : 'a Ref -> 'a -> unit ST
  val get : 'a Ref -> 'a ST
end
structure ImpureRef :> REF =
struct
  type 'a ST = unit -> 'a
  type 'a Ref = 'a ref
  fun unit a = fn () => a
  fun m * k = fn () =>
    let val a = m ()
        val m' = k a ()
    in m'
    end
  fun runST s = s ()
  fun new a () = ref a
  fun set r x () = r := x
  fun get r () = !r
end
```

[Wadler95] also describes a monad for updateable arrays; since large real-world details (such as bounds) are ignored, the impure version has been omitted.

```
signature ARRAY =
sig
  type 'a M
  val unit : 'a -> 'a M
  val * : 'a M * ('a -> 'b M) -> 'b M
  eqtype Ix
  type Val
  val block  : Val -> 'a M -> 'a
  val fetch  : Ix -> Val M
  val assign : Ix -> Val -> unit M
end
structure PureArray :> ARRAY =
struct
  type Ix = int
  type Val = int
  type State = Ix -> Val
  type 'a M = State -> ('a * State)
  fun unit a = fn s => (a, s)
  fun (m : 'a M) * (k : 'a -> 'b M) = fn (x : State) =>
    let val (a, y) = m x
        val (b, z) = k a y
    in (b, z)
    end
  fun block v m = let val (a, x) = m (fn _ => v) in a end
  fun fetch i = fn x => (x i, x)
  fun assign i v = fn x => ((), fn i' => if i' = i then v else x i')
end
```

2.5 Exception monads

Monads can be used to emulate exceptions in a rather straightforward manner:

```
datatype 'a M = Raise of exn | Return of 'a
fun unit a = Return a
fun bind m k =
  case m of
    Raise _ => m
  | Return a => k e
```

The implementation is quite succinct, and the exception-handling mechanism is simple to use: just return Raise e to raise exception e, and Return v to proceed as usual, returning value v. Unfortunately, exception handling using this implementation does not short-circuit as quickly as typical first-class exceptions.

2.6 A generic "impure code" monad

We have thus far seen some representations of monads that allow us to embed effectful ("impure") features into effect-free ("pure") languages; what about the dual case where we already have a language with effects (for example, operating system calls in C) and want to embed an effect-free language? We might then re-structure our monad to generate single-threaded suspensions that, when called, force effectful actions to be carried out in the correct order. We carry around a “world” to which our effects are applied, and require that the world of each previous computation be used in the next.

```
type 'a M = world -> ('a * world)
fun unit a w = (a, w)
fun bind m k w =
  let val (w', a) = m w
  in k a w'
  end
```

This pattern forces correct sequencing of operations; however, if the state of the machine keeps track of the world for us (as is the case in almost all real-world computers), we need not pass it around ourselves! Instead, we pass a place-holder value (), which conceptually represents not the world itself but rather permission to access the world. Since this value contains no data, it may be optimized away entirely – thus, the implementation of monads in an impure language can be done extremely efficiently.

2.7 Summary

Monads represent an interesting means for combining pure and impure programming languages. They are general enough to be realizable without significant damage to existing type systems, and moreover they allow systems to be designed for easy extensibility (by modifying the definition of the monad). The unfortunate drawback to monadic systems is that for monads to be useful, a number of other functions have to be defined that produce monadic values; for example, the I/O monad requires auxiliary functions for printing and inputting strings, or the Array monad requires functions to generate arrays. While many interesting properties have been demonstrated for monads themselves, the introduction of these additional functions makes proving the correctness of an implementation significantly more difficult.

In short, monads are a viable solution for controlling program sequencing and effect interaction, but they are vulnerable to problems in the auxiliary functions required to make a monad non-trivial.

3. EFFECT SYSTEMS

3.1 Formal effect systems

Since monads isolate the impurity of computations in a pure language, one might expect a similar formal system for tracking the purity of computations in an impure language. Such a system is an effect system, and may range from a very simple annotation scheme (“pure”, “reads”, “writes”) to a complex mechanism for tracking each access of individual variables.

3.2 Effect systems and monads

Wadler and Thiemann [Wadler03] describe such a system, a value-polymorphic impure language that tracks creation, reading, and writing of variables by annotationg each function type with the set of effects that application of the function may produce. They draw upon previous work with monads to demonstrate that the effect system can be represented completely by a monad-based system; the translation from the effect system to the monad system is relatively straightforward, and the resulting monadic expression is always well-typed if the effect-language expression is well-typed.

3.3 Effect systems for imperative programming

Henglein, Makholm, and Niss [Henglein05] describe a series of systems of increasing specificity, starting from a base language without effects and incrementally extending and refining the language to yield a sophisticated type system and semantics for region-based memory allocation and deallocation. The region system presented is therefore an excellent case for language extensibility.

3.4 Extension and refinement

The Henglein article derives the full effect system from several iterations and extensions of a base language BL. A few additional rules are added to the system to form TL ("tagged language"), a language in which variables may be tagged with a label. The type system of this new language guarantees that the untagging operation always succeeds, extending the soundness theorem for BL to TL. Another extension (called STL, "scoped tagged language") attempts to add a "new" operator for limiting the scope of a region; however, soundness fails to hold for STL because regions may escape unnoticed inside function closures.

Henglein et al overcome this unsoundness by moving to an effect type system RTL ("region typed language") based on the Tofte-Talpin type system, in which an operation that may read or write a region ρ has its type annotated with ρ. The system has a sub-effecting relation; that is, a program which does not manipulate ρ is still acceptable as a program which may but happens not to. Now programs which previously allowed a region to escape and be used unsafely simply fail to type-check.

Armed with the sound region-typed language, the article then presents an extension of the region-typed language that adds lists, and claims that adding other data types is straightforward. The article has been formulated from the start with a tacit assumption of extensibility; the design of the language follows naturally from simple soundness-preserving extensions and soundness-restoring refinements to a base language. This extension-based formulation of a new language is fairly typical of research-oriented language design; region types are but one example of the potential of a truly extensible programming language to expedite the production and prototyping of interesting new languages.

3.5 Intermediate languages

Region type systems are also interesting as an example of intermediate representation languages for language design. In particular, the authors propose not that RTL be used for programming, but that a region-inference algorithm may be used to map well-typed BL programs to well-typed RTL programs. Since the abstract representation of RTL takes memory management into account (while BL does not), it may be more useful for a real-world compiler to compile BL programs to RTL programs before translating to machine-code; likewise, if the machine already has an existing implementation of BL, one might write an RTL interpreter that simply type-checks the RTL code and performs erasure on it, passing the result on to the BL interpreter. The correctness of the erasure algorithm ensures correct semantics of any extensions for which erasure holds.

3.6 Summary

Effect systems allow the effects of a computation to be tracked via annotations of the types of effectful computations. They can be expressed using monads, or translated (via erasure) to less-complicated type systems.

The treatment of effect systems given in [Henglein05] utilizes language extension heavily in defining RTL, and is thus particularly well-suited to clean implementation in an extensible language. While RTL itself would be impractical to program in, it may be well-suited for use as an intermediate language to which other languages (such as BL) might be translated; likewise, small languages may be used to define and implement the semantics of larger languages (such as implementing RTL by erasure to BL).

4. MODALITY

4.1 Modal logic (and monads)

Modal logic is a logic concerning necessity and possibility across possible worlds. Many variants of modal logic exist with different properties; common formulations include one or both of the operators $\Box$ and $\diamond$, where $\Box P$ means "P is necessarily true" and $\diamond P$ means "P is possibly true". The possible-worlds formulation of modal logic projects these concepts as quantifiers over worlds; $\Box P$ then means "for all possible worlds, P is true in the world" and $\diamond P$ becomes "there exists some possible world in which P is true".

One modal system that has been explored for practical applications in programming languages is the modal logic S4. S4 has the properties that:

—$\Box P \supset P$
—$\Box(P \supset Q) \supset \Box P \supset Q$
—$\Box P \supset \Box\Box P$

This stands in contrast to monads, where

—$P \supset M\ P$
—$(P \supset M\ Q) \supset M\ P \supset M\ Q$
—$M\ P \supset M\ M\ P$

Whereas monads express single-threading of program flow the modal $\Box$ can express the independence of terms with respect to threading behavior.

4.2 Modal types for program staging

Wickline, Lee, Pfenning, and Davies [Wickline98] describe an application of S4 to run-time code generation. In particular, values of type $\Box\tau$ are run-time generators for values of type τ. $\Box$ serves as a separator between compiled code and code that remains to be compiled. One obtains a value of type τ from a generator of type

$\Box\tau$ using a let cogen block, which forces the code generator to actually compile its code down to a value, performing any optimizations it can at that time. The use of $\Box$-types allows the programmer to explicitly delay computation of an expression until the last possible time, enabling optimizations that would otherwise not occur. This leads to a more predictable means of staging programs, since the programmer knows for a fact that the code will be specialized (and isn't simply relying on the compiler's optimization system to infer when it should and should not specialize the code).

Wickline et al claim a number of benefits of their system:

—Code generation is stated explicitly in the code, and localizes staging of computations within a block of code (rather than relying on global analysis).

—Binding of dynamically-generated code is guaranteed to succeed by the type system.

—Values that are unavailable at compile-time must be explicitly added to the code generators, since use of pre-generated values may prevent important optimizations.

Importantly, their system also appears to be orthogonal to other common language features (such as mutation and exceptions); this means that modal code-generation can be added as an extension to another existing language! So it would seem that this approach to run-time code generation is well-suited for use in an extensible language.

4.3 Modal types for feature separation

The $\Box$ constructor need not just stand for staging separation; it could also be used (with some additional annotations) to separate one feature from another; in this interpretation an expression having type $e : \Box_L\tau$ might mean "e has type τ using only values from the sub-language L".

4.4 Custom typechecking and modality

One aspect of extensible programming that might possibly be solved using modality is separation of (compile-time) typechecking code from (run-time) evaluation. In particular, new features being added to a language might require some non-trivial type inference algorithm; the algorithm itself may be written in the language being extended, but only if all values used by the typechecker are already known at compile-time and do not refer to any code which has not yet been compiled.

4.5 Summary

The modal $\Box$ operator expresses the independence of a term from its surrounding world. $\Box$ can be used to extend existing languages with new features, but it can also be used in an extension system to separate independent sections of code. One such separation that we might want to enforce is the separation between compile-time and run-time, so that we can utilize extensions as part of compilation without sacrificing the run-time soundness of the language.

5. AN EXOTIC COMBINATION: BI

The logic of bunched implications (BI) presented by Pym, O'Hearn, and Yang [Pym02] describes a combination of some connectives from linear logic with propositional intuitionistic logic. Unlike most combinations of linear and propositional logic, BI does not utilize a modal barrier between the two. Instead of representing linear and non-linear propositions in separate contexts, BI combines them together into a single tree-like context in which propositions are joined either simultaneously (using ";") or separately (using ","). One common interpretation of BI's context is as a memory heap, in which sub-heaps separated by "," are distinct; this distinctness between bunches is also central to other interpretations of BI.

Pym et al take great care to point out that BI is significantly different from linear logic; this distinction is important – although the two languages share certain features, they way in which they have been combined (using bunches instead of modality) noticeably changes the meaning of the language. In extending a language, one must be careful to consider the interactions of features across languages, as these interactions determine the meaning of the language beyond being the sum of its parts.

6. REMAINING OBSTACLES

6.1 Language description

Among the number of problems yet to be resolved for a truly extensible language to exist is the problem of language specification. In particular, there exist three aspects of any feature that must be specified unambiguously in order to evaluate terms that utilize the feature: concrete syntax, a type structure, and evaluation semantics.

6.1.1 *Extensible parsing.* The first of these aspects, syntax, is generally specified for formal languages using a context-free grammar, typically in Backus-Naur Form (BNF). Well-known parsing algorithms for context-free languages can parse an expression of length l in better than $O(l^3)$ time. Extensibility challenges the basic assumption of a fixed grammar; how does this change the parsing problem? In particular, how does a changing grammar effect the complexity of parsing algorithms? Once we have an extensible parser, does that extensibility provide any further advantages?

6.1.2 *Efficient extensible unification.* The second aspect of language description, type structure, is perhaps the most formalized; current systems like ELF focus heavily on extensible type systems. However, these systems may not easily express all the constraints one might want to place on type-checking a particular language. Algorithms for type unification are often tailored to a particular type system, under the assumption that any new datatypes introduced will either be simple algebraic datatypes or aliases to other types. With the addition of new and unusual typechecking rules, how does the unification problem change?

6.1.3 *Describing evaluation.* Evaluation rules are perhaps the best-understood aspect of language description; we can often describe the correct evaluation rules for a new feature in terms of other existing features. We have examined monads as a possibility for describing interesting properties of evaluation for a language; however, monads do not easily capture, for example, the requirement that terms be evaluated lazily. Many possibilities exist for describing the semantics of languages; which representations, if any, are well-suited to describing extensions to existing languages?

6.2 Theory of language combination

Another important problem we have not addressed is safety of combinations. Many formal languages (such as BI) are interesting because they can be proven to have interesting properties (linearity, for example, or decideability). It is certainly not the case that all combinations of any two languages preserve all interesting properties of both languages; in order to add a feature to a language, we would like to be able to prove that the feature not only preserves the soundness of the system, but also that the feature's interactions with the existing system do not destroy any important properties of the feature or the system. An open challenge, then, would be to establish a good algorithm for deciding when and how two or more features can be safely combined and which aspects of the features will be preserved.

7. CONCLUSION

We have seen in these papers both the need for extensibility and mechanisms by which extensions to a language may be controlled. Often, as in the case of monads[Wadler95, Jones93] and modal logic [Wickline98], mechanisms for extension are themselves formulizable as extensions of simpler languages. In effect systems, we find that there are often several ways to represent a feature (e.g. effect languages as monads or as annotations of simpler effectful languages). In [Henglein05] in particular, we find an excellent motivation for wanting an extensible language; namely, that interesting new languages often result from extending well-understood languages with new features, and definitions of such languages are often best described by merely describing the extensions. In [Pym02], we are reminded that the form of an extension often carries with it important information; it is not sufficient simply to add new features, but one must also define their interaction with old.

We have also seen some additional problems that should be resolved for significant progress on extensible programming to be made; while much work remains to be done, we hope that these examples may shed light on methods, features, considerations for extensible programming today and their relation to the languages of tomorrow.