Ph.D. Thesis Proposal: A Next-Generation Collaborative Programming Language

Doug Orleans

August 23, 2000

1 Introduction

Computer programming is seldom a solitary effort—many projects are just too big for one person to implement, and even smaller projects can often be done faster with multiple developers. The usual approach to multi-developer programming is to use collaborative tools such as a version control system to facilitate the exchange of source code. An alternate approach, however, is to use a programming language that itself supports the concept of multiple programmers collaborating on a project. Several such languages exist, such as MOO[Cur97], ColdC[Gil], and LPC[Ree93b][Ree93a]; however, since they were written as descendants of MUD[Cow99], a multi-user text adventure game (and now referred to generically as *muds*), they were designed by hobbyists who had little background in programming language design or theory, and thus are somewhat clumsy and ill-suited for applications outside the domain of muds (and in some ways are even inadequate for the mud domain itself). Many ambitious projects have been successfully implemented using them, however, even outside the domain of muds, demonstrating that the idea of collaborative programming languages is a useful one. Little substantial progress has been made in this area over the last seven years; this indicates that some plateau had been reached, where the languages had matured to an expressiveness that was suitable enough for widespread use. I believe, however, that there are still significant improvements to be made in the design of collaborative programming languages. For my dissertation I will explore some of these possibilities, designing and implementing a prototype language that is suitable for fleshing out into a next-generation collaborative programming language. Along the way I will attempt to bring some academic rigor to the area, surveying and comparing existing languages and developing semantics for "featherweight" subsets of them (in the manner of FJ[AIW99]) as a basis for further study. The rest of this paper presents a feature overview of the most prominent collaborative programming languages, an evaluation of their advantages and shortcomings, a description of the basic prototype design for a new collaborative programming language, and finally an outline of my plan for completing the dissertation. Included as an appendix is a presentation of the formal syntax and semantics for FM (Featherweight MOO).

2 Feature overview

In order to design a next-generation collaborative programming language, I am studying current collaborative languages to see what features I need to include and what needs to be improved. What follows is an overview of the basic features of some of the more widely-used languages; my dissertation will include a more in-depth survey of the features of these and other languages.

2.1 Common features

Before getting into the details of a particular collaborative programming language, I will start with an overview of what the languages I am studying have in common. First of all, by definition, they allow multiple programmers to collaborate. In practice, this means that they act as servers, accepting input from multiple network sockets asynchronously, and allow code (i.e. behavior) to be added or modified dynamically based on this input. (A digression: the features I discuss may appear to be features of a programming language *environment*, rather than of a programming language; however, the line between the two is often blurred, even in more mainstream languages— the interactive environment is fairly integral to the Smalltalk[Lou] and Lisp[Ste90] families of languages, and the Java language specification[GJS96] covers the class loading process in detail. More importantly, however, a collaborative programming language typically *reifies* the programmer as a structure that is part of the semantics of the language, so features that usually are part of the environment really *are* part of the language.) This feature, asynchronous dynamic behavior modification, requires a number of other features: *type safety* and *garbage collection*, so that a wild pointer or memory leak caused by one programmer can't cause the whole server to crash; *multitasking* and *resource control*, so that one programmer's code can't monopolize the server by, for instance, running a tight infinite loop; *persistence*, so that the programmers don't have to re-upload all their code when the server is shutdown and restarted; and *reflection*, so that programmers can examine the code that's already on the server (their own or others').

Another feature that collaborative programming languages usually have is *security*: a programmer owns code and data that he creates, and can control access to it; however, while this can be useful for enforcing abstraction barriers (much like access control in other languages, e.g. the `private`, `protected`, and `public` keywords in C++[ES90] and Java), it is not strictly necessary for collaboration, and is often not used at all. Collaborative programming languages are often (perhaps predominantly) used in another setting than collaboration, though, which I will call *community programming*: programmers share code and data dynamically, yet they may not always be collaborating in the sense of working towards a common goal, and in fact may not trust each other at all; in this situation, access control mechanisms are essential.

The collaborative programming languages I am studying share another feature that is not essential to collaboration: they are all single-dispatch classless object-oriented languages. That is, they have objects, which have fields and methods, and which may inherit fields and methods from other objects. These languages are probably object-oriented because they were designed for the multi-user simulation environment domain, and OO languages are well suited for simulations; the fact that they are all also classless may be coincidence, but it may also be due to the dynamicity properties of the language: it's easier to modify a field or method on an object, which is dynamically inherited by the object's descendant objects, than it is to modify a field or method on a class and update all the instances of the class. Interestingly, recent versions of several of these languages have added mechanisms that can simulate the class-instance relation.

2.2 MOO

The MOO programming language is idiosyncratic in that it is a mud language, and its terminology reflects that fact: users are called *players*, super-users are called *wizards*, object methods are called *verbs* (because they are used to execute the verbs in commands, such as `look in mirror`, `get key`, or `put book on shelf`), and object fields are called *properties*. There are also a number of aspects to the language that are irrelevant to a general-purpose collaborative programming language: for example, all objects have `name`, `location`, and `contents` properties, and all user input is processed by default by a command parser that tries to match words in the command to objects in the same location as the player. In general I will ignore mud-specific aspects of the language in this paper, but occasionally they have an impact on other parts of the language.

All program code in MOO lives in the verbs; there are no global functions or static code blocks. The MOO server has a built-in command for programming a verb that accepts the code of the verb terminated by a single period on a line by itself. Typically programmers can also evaluate single expressions or sequences of statements with an `eval` command. The syntax is somewhat similar to C++ or Java, with semicolon-terminated statements and infix arithmetic; object properties are referenced with a dot, e.g. `player.name`, while verbs are invoked with a colon, e.g. `player:name()`. The server provides a set of *builtin functions* (analogous to system calls) which are invoked as if they were global functions, e.g. `notify(`*p*`, "Hello, world!")` sends the string "Hello, world!" to player *p*'s connection. Variables are untyped and do not need to be declared ahead of time; using a variable before it has been initialized results in a "Variable not found" exception. There are no formal named arguments to a verb; instead a single special variable `args` holds the list of actual arguments used in the verb invocation. Scattering assignment can be used (as in Perl) to assign the arguments to a list of variables, e.g. `{a, b, c} = args`. Other special variables include `this`, which is the receiver object of the current verb invocation, just like in C++ or Java (`this` must be explicitly used when referring to properties and verbs on the current object); `caller`, which is what the value of `this` was when the current verb was invoked; and `player`, which by default is the player who entered the command currently executing (but the value of `player` can be changed by a verb running with wizard permissions—see the paragraph on security below).

As mentioned in the overview, MOO is a single-dispatch classless object-oriented (sometimes referred to as object-based) language. Objects inherit from each other, singly; thus, each object has a parent, and when a verb or property is referenced on an object, if it's not defined on that object, the object's parent is searched. A verb can call the builtin function `pass()` to re-send the current verb call to the object's parent. Properties use copy-on-write inheritance; when a property is changed on a child object it is copied to the child and then changed. Each object has a unique fixed integer associated with it, known as its *object number*, or *objnum*; all objects can be referred to in code with a literal objnum expression `#`*objnum*. Objnums are assigned at object creation time in increasing order starting with `#0` for the system object. A shorthand exists for referring to properties and verbs on the system object: `$foo` is equivalent to `#0.foo`, and `$bar(x, y)` is equivalent to `#0:bar(x, y)`. This allows for a global namespace of sorts; a useful object can be given a name by adding a property to `#0` with the object as its value, so that, e.g., `$player` can be used to refer to the generic player object that is used as the parent of all player objects.

Objects are not garbage-collected; objects must be deleted explicitly. Objnums referring to deleted objects (as well as negative objnums) are referred to as *invalid*. Deletion of objects leaves "holes" at these invalid objnums; there is a `renumber()` builtin function that reassigns an object's number to be the lowest nonnegative invalid objnum, but this is not used in practice (except in special cases) because it does not fix existing references to the object's old number (in properties or in verb code). Instead, the custom is to never actually delete an object, but to strip all verbs and properties from it and put it onto a free list of "garbage" objects that can be reused when new objects are needed.

There are other, non-object values in MOO, such as integers, strings, and lists; list values are immutable heterogenous arrays and are garbage-collected, using a simple reference-counting scheme. Lists are often used for data structures, because objects are heavyweight, both conceptually and in time and space overhead, due to the cost of recycling and the allocation of builtin properties such as name and location that are only useful for objects representing virtual entities. A recent experimental extension to the language adds a new value called *waifs*; this extension in essence treats normal MOO objects as classes, with waifs acting as instance objects. Each waif value has a class (an objnum) and a list of values corresponding to its class's list of defined properties. Verbs are invoked on waifs just as they are on objects, by dynamically looking up the class's inheritance chain; waifs cannot define their own verbs or properties. Waifs, like lists, are garbage-collected, but are mutable, like objects, providing the best of both worlds, although the current implementation imposes the limitation that circular waif structures cannot be created, so that reference-counting garbage collection can still work.

MOO is single-threaded, but has multitasking; each command received from a player starts a task, and additional tasks may be created with a `fork...endfork` statement. Task switching is done cooperatively: a task may surrender control to other tasks for a given minimum number of seconds n (which may be 0) by calling the builtin function `suspend(`n`)`; a suspended task may be resumed before its time by calling the builtin function `resume(`*id*`)`, where *id* is the task-id of the suspended task (a unique integer). The `read(`p`)` builtin function also suspends the current task, resuming when a line of input has been read from player p's connection. In order to prevent a task from monopolizing the server by not suspending, however, the server interrupts a task if it has run for more than a certain amount of time or number of instructions (*ticks*); the default threshold amounts are 5 seconds and 30,000 ticks (for command tasks), which can be overridden by setting special properties on the system object (`#0`). When a task is interrupted by the server, an uncatchable exception is thrown and printed to the player who typed the command that created the task (this behavior can also be overridden by adding a special verb to the system object). Usually, a task that has a loop that might run a large number of times will call the builtin function `ticks_left()` each time through the loop to determine whether it needs to suspend.

In addition to time restrictions on a task, the amount of space used by a player can be restricted; if a property named `ownership_quota` exists on a player, then when that player calls the builtin function `create()` to make a new object, the value of the property is decremented by one if it's positive, or else a "Resource limit exceeded" exception is thrown. The value is likewise incremented whenever an object is destroyed with the `recycle()` builtin function. (If a "garbage" list is used as described above, this value can also be adjusted whenever objects are added to or removed from the list.) The builtin function `object_bytes()` can also be used to measure the size of an object, if a more finer-grained quota policy needs to be implemented. There are also restrictions on the number of nested verb calls in a task stack and the number of forked or suspended tasks owned by any one player.

The heap of allocated objects is known as the *database*, or *db*, although the db does not have much resemblance to a conventional database; the db is kept entirely in memory in a single process, and must be periodically checkpointed by calling the `dump_database()` builtin function, which saves the db to disk in a format that can be reloaded if the system needs to be restarted after being shut down (or crashing). A version of the MOO server called LPMOO[Lesb] is implemented on top of another mud server, DGD[Lesa] (in its language, LPC), which itself provides more continuous persistence, as well as smaller process size, by keeping all objects on disk and loading them into memory only when needed; only the working set of cached objects need be written to disk when checkpointing, which can then be done more often since it's much faster.

There are a number of builtin functions that provide reflective capabilities. For introspection, there are builtin functions to determine an object's parent, its children, its defined properties or verbs, or whether an object is a player object; there are builtin functions to inspect a property or verb definition's name, owner, and permission flags (see next paragraph), or about a builtin function, or to retrieve a verb's code; and there are builtin functions to get the list of player objects, the maximum object number, or the list of currently queued tasks. For invocation, there are builtin functions to change an object's parent, to make an object a player or not a player, to add or delete a property or verb from an object, to set a property or verb definition's name, owner, or permission flags, or to set the code of a verb; there are also builtin functions to call a builtin function given its name in a string, or to evaluate an arbitrary expression given in a string. In addition, properties can be referenced using a computed name with special syntax, e.g. `obj.(propname)` will read the property whose name is the value of `propname` (which must be a string) on the object `obj`; verbs can be invoked with a similar syntax, e.g. `obj:(verbname)()` will invoke the verb whose name is the value of `verbname` on the object `obj`. There is a limited form of intercession, as well: whenever a builtin function *fun* is called, if there is a verb on the system object whose name is bf_*fun*, it is invoked instead. There is no way to intercede on a property reference or a verb invocation, however.

Security in MOO is accomplished with a simplified Unix-style permission system. Tasks always run with the permissions of a single player, similar to the effective user id of a process in Unix. Every object, verb, and property has an owner and `r` and `w` flags that determine if tasks running with permissions other than the owner (or a wizard) can read or write them. Each object also has an `f` flag ("fertile") that determines if tasks running with permissions other than the owner may create children of the object. Property ownership and permissions are inherited with copy-on-write similar to property values: if the owner or the permission flags of a property are changed on a descendant of the object that defines the property, the values are copied to the descendant and then changed. In addition, each property has a `c` flag, which determines whether the ownership is changed on child objects to be the same as the owner of the child object; this allows a user to create a child of someone else's object, and still be able to modify the properties of the child object that are defined on the parent. If this flag is not set, then the copy of the property on the child remains owned by the owner of the parent property; this allows verbs defined on the parent object, which are usually owned by the parent object owner, to modify the properties on the object. Tasks always run with the permission of the owner of the current verb, rather than the permissions in effect when the verb is invoked, so that a verb owned by a player can always access properties owned by that player as well. This is the reverse of the normal Unix case; it is as if all programs in Unix had their set-user-ID bit on. Tasks running with wizard permissions, i.e. verbs owned by wizards, can call `set_task_perms(`*p*`)` to change the current task's permissions to player *p*; this is often used to run with the caller's permissions, by setting them to the value of the builtin function `caller_perms()`. Tasks running with wizard permissions may also change the value of the special variable `player`, in order to simulate a command being sent by another player than the one who actually sent it. A number of builtin functions are restricted to act only on objects, verbs, or properties owned by the current permissions, unless the current permissions are those of a wizard, such as `chparent()` and `recycle()`; others may only be with wizard permissions, such as `dump_database()` or `shutdown()`.

2.3 ColdC/Genesis

ColdC (originally called C−−) is a successor to MOO, and so shares many of MOO's features, improving on some of them. The main innovation was the removal of all mud-specific features, creating a true general-purpose collaborative programming language. The ColdC server (or *driver*), Genesis (originally called Coldmud), does not attempt to parse commands, nor does it maintain location and contents; even security features have been almost completely removed. The intention is that all of these features can (if desired) be implemented on top of the base functionality that the language provides. Terminology is more in line with traditional OO languages: users are called users and methods are called methods; fields are called *object variables*.

ColdC syntax is more like C++ than MOO is: both object variable and method reference is done with dot (`foo.bar` or `foo.bar()`), and object variables on the current object may be referred to by simply naming them (i.e. no explicit `this` needed); local variables must be declared at the top of a method (though there is still no static typing); method parameters are named; block structure can be provided with curly braces. Invoking a method on the current object can be written as `.foo()`; the dot is still necessary to avoid ambiguity with functions (the ColdC term for what MOO calls builtin functions). There are no special variables; instead, functions are used: `this()` returns the current receiver object, `definer()` returns the object that defines the current method (which is always an ancestor of `this()`), `sender()` returns the object that invoked the current method, and `caller()` returns the object that defines the method that invoked the current method (an ancestor of `sender()`). The function `user()` simply returns whatever was passed to the last call to `set_user()` on the current object; the intention is similar to the `player` special variable in MOO, i.e. to keep track of the originating object for the current task, but the driver does not do anything special to set or maintain it.

The ColdC object model has multiple inheritance; method inheritance uses depth-first search up the parents list, from left to right, skipping parents it has already searched due to inheriting the same parent via multiple paths. The `pass()` function re-sends to the next method that would be found in the current method invocation, i.e. it might not always be an ancestor of `definer()`. Objects have unique numbers, as in MOO, which are referred to with the syntax `#`*objnum*; they also have unique alphanumeric names, which are referred to with the syntax `$`*name*. Object names can be changed with the `set_objname()` function, but this should be done with care as methods with literal references to the old name will not be updated.[1]

Similar to MOO, objects in ColdC are not garbage collected, but there is a lightweight object data structure similar to the waifs extension to MOO, called a *frob*. Unlike waifs, however, frobs are immutable, which somewhat limits their utility.

ColdC has cooperative multitasking, similar to MOO. Atomic mode can be turned on with the function `atomic()`, which prevents any other tasks from running until atomic mode is turned off; a task running in atomic mode is also immune to the tick limit. The creation functions do not enforce any sort of quota policy, but one can be implemented in ColdC using the `bind_function()` function (see the next paragraph below). Genesis is disk-based, i.e. it keeps all objects on disk and only loads them into memory when needed; however, tasks are not persistent and must be manually saved in a restartable format if they need to stay around after the system is restarted. Reflection in ColdC is similar to MOO, but with no intercessory capabilities and no `eval()` function.

ColdC has no notion of task permissions such as MOO has; however, it allows C++/Java style access control for methods, using the keywords `public`, `protected`, and `private`, and it is a runtime error to call protected or private methods from outside the object. There are three additional access control keywords: `root`, which means a method may only be called by the `$root` object; `driver`, which means a method may only be called by the driver; and `frob`, which means a method may only be called with a frob as the receiver. All object variables are considered to be private, i.e. only the defining object may access its object variables. Functions (e.g. administrative functions like `shutdown()`) can be protected by using the `bind_function(`*fun*`,` *obj*`)` function to only allow the *fun* function to be called from methods defined on *obj* (typically `$sys` or `$root`). This can be used to force access to functions to go through methods, which can impose extra security or resource control measures. A method may also be set `nooverride`, which means that no descendant object may define a method with the same name; this can prevent spoofing of secured methods.

2.4 LPC/DGD

LPC is the language for LPMUD, a descendant of the original MUD that evolved in parallel to MOO without much cross-breeding. DGD (Dworkin's Generic Driver) is the latest LPC driver, which, like Genesis/ColdC, is a general-purpose collaborative programming language because it doesn't include any mud-specific features. Its syntax is almost identical to C, including statically typed variables but not including pointers (but objects, strings, arrays, etc. are passed by reference, similar to Java). The database consists of a Unix-style hierarchical file system; each object corresponds to a file whose name ends in `.c`, e.g. `/system/driver.c`, which contains the source code for the object variables and functions. A literal object reference is simply a string containing the source filename of the object (without the `.c`). Calls to functions in other objects are done C++-style, i.e. `obj->fun()`. Driver functions are called *kfuns* (kernel functions) and are called the same way as functions in ColdC or MOO. The kfuns `this_object()`, `previous_object()`, and `previous_program()` play the roles of ColdC's `this()`, `sender()`, and `caller()`, respec-

[1] I don't know why they aren't just stored as object numbers!

tively. An object can (multiply) inherit from other objects using `inherit` *obj*`;` declarations. A function can re-send to a function on its parent using `::`*fun*`()`. Directed re-send is accomplished by providing a tag in the `inherit` declaration and using the tag, e.g. the declaration `inherit p2 "/foo/bar";` allows re-sends to the object `"/foo/bar"` with the expression `p2::`*fun*`()`.

LPC objects are not garbage collected, but arrays and other data structures are. Something similar to waifs and frobs exists, called *clones*: instead of inheriting an object by making a new object file with an `inherit` declaration, you can call the kfun `clone_object()`, which makes a new copy of an object. A clone cannot itself be cloned or inherited from, but is otherwise like a regular object. When an object is modified and reloaded (with the kfun `compile_object()`), all of its clones are modified as well, keeping the same values for variables whose definition did not change. Clones are assigned names in the file system, based on the name of the object it was cloned from and a unique number, e.g. a clone of an object named `/foo/bar` will be named something like `/foo/bar#1234`. Since clones are always accessible by name with a literal object name expression, they are not garbage collected.

LPC has cooperative multitasking, similar to MOO and ColdC. The resources (maximum stack depth and number of ticks) available to a block of code can be specified with a `rlimits(`*depth*`,` *ticks*`) { ... }` construct. DGD is disk-based, similar to Genesis. Reflection is similar to MOO, although introspection is mainly done at the object level rather than the level of functions and object variables. (The exception is the kfun `function_object()`, which determines which inherited object provides a given function.) There is no `eval()` function. Intercession is achieved through a number of hooks that are called on the driver object (similar to the system object in MOO or ColdC); of particular note is `call_object()`, which is called whenever an object calls a function on another object—neither MOO nor ColdC has intercession on method calls. In addition, kfuns can be shadowed (and new ones added) in the root object, called the *auto* object, similar to `#0:bf_`*fun* wrappers in MOO.

LPC's approach to security is similar to ColdC's: access to functions and variables is controlled with keywords such as `private` and `static` (similar to `protected` in other languages), and all other access control must be implemented in the database (with the intercessory driver hooks, as opposed to restricting access to kfuns as is done in ColdC). The keyword `nomask` serves the same purpose as ColdC's `nooverride`.

3 Evaluation

Now that I've surveyed a few existing collaborative programming languages, I can begin identifying features that I want to keep and features that I want to improve upon. Of course, the general features I identified as essential in section 2.1 will need to go into my language: dynamic behavior modification, type safety, garbage collection, multitasking, resource control, persistence, and reflection. I will also need to at least provide a framework in which security capabilities can be implemented.

The classless object-oriented model seems to be a successful one, and my own personal bias is towards a classless object model as well; as Abadi and Cardelli observe in their book *A Theory Of Objects*[AC96], page 36:

> The main insight of object-based languages is that class-based notions need not be assumed, but instead can be emulated by more primitive notions. Moreover, these more primitive notions can be combined in more flexible ways than in a strict class discipline.

One of the key barriers to this sort of flexible combination in MOO, ColdC, and LPC is the lack of garbage collection for objects. The ability to access any object by number (in MOO) or name (in ColdC and LPC) leads to this restriction, which in turn has led to ad-hoc alternative mechanisms for lightweight objects (namely waifs, frobs, and clones) that are clumsy and limited. This can be solved by simply not allowing pointers to arbitrary objects to be created.

Dynamic behavior modification is somewhat clumsy in LPC: you can only change code by recompiling an entire object, and if the object has descendants, it and they must all be destructed and reloaded. Some of this can be automated, but the granularity of updates is still too coarse. MOO and ColdC allow users to update individual methods and fields on an object, and all descendants (and waifs and frobs) automatically inherit the changes.

Cooperative multitasking of the sort implemented in MOO, ColdC, and LPC has its benefits: you know exactly what part of your code is atomic (namely, everything between calls to `suspend()`), so you don't have to put semaphores on every public data field. However, there are two big disadvantages. One is that every method you call that you don't own might turn out to suspend when you didn't expect it; this is usually solved by the convention that every utility method that might suspend has `_suspend` in its name. However, the fatal problem with cooperative multitasking is the tick limit—it can be very tedious to always check to see how many ticks are left, and if this check is left out, the code might fail with an uncatchable exception, which could cause all sorts of havoc with unfinished data structures and the like. In practice, it becomes almost as much work to deal with the tick limit as it would be to deal with semaphores, and as soon as you start suspending you need to worry about concurrency anyway. Preemptive multitasking is the only real solution.

Most of the built-in resource control features of MOO, ColdC, and LPC deal with tick limits; with preemptive multitasking, this is unnecessary, assuming the thread scheduler avoids starvation. The quota-based approach to memory consumption seems like the best model, and powerful enough reflection capabilities allows this to be implemented in a fairly straightforward manner, i.e. by interposing a quota check in calls to object creation primitives.

The MOO server handles persistence by periodic checkpoints of the entire database; the Genesis and DGD drivers optimize the checkpoint time by only keeping a cache of active objects in memory, at the expense of the extra time needed to swap inactive objects into the cache when they are referenced. While this tradeoff may be desirable for some purposes, I believe that MOO's model is sufficient for most purposes: it's the job of the operating system to provide virtual memory and take care of swapping to and from disk, and implementing an additional swapping mechanism on top of this seems like diminishing returns. More sophisticated persistence mechanisms such as journaling or compare-and-commit might be worth investigating, but I don't consider it an important enough issue to spend time on it for my thesis. The issue of what to persist is another question; I think MOO and LPC have the right idea, where everything (including tasks) is saved to disk (ColdC doesn't save tasks). In addition, LPC and ColdC allow a user to mark fields that don't need to be saved; while this can save some time and space in some instances, I don't see it as being a crucial feature.

MOO, ColdC, and LPC all enable introspection and reflective invocation in much the same way, with an ad hoc but fairly complete set of global functions. Intercession is handled similarly in MOO and LPC, by allowing the global functions to be globally overridden; ColdC is slightly more flexible, allowing functions to be bound to particular objects which can then override the behavior. In practice, though, these schemes turn out to be virtually equivalent. LPC has the added advantage of providing kfuns that correspond to method calls, which are only available as syntactic constructs in MOO and ColdC; this gives users the important ability to intercede on every method call, which allows a much greater range of customization of the meta-behavior. While the reflective global functions idea is more or less sufficient, I think putting the reflective functions into a meta-object protocol (where every object has a meta-object as well as a parent) provides for easier and perhaps more disciplined reflection, as you then gain all the advantages of object-oriented programming when using reflection: encapsulation, inheritance, and polymorphism.

MOO's approach to security is fairly simple, but can be cumbersome to use. Programmers find themselves repeating the same code in verb after verb: inspect `caller` and/or `caller_perms()`

and raise an exception if they're not allowed to access the verb. ColdC and LPC's `private` and `protected/static` access keywords remove the need for most of this code, but conversely, neither of them allow access to an object's fields from outside of the object, so the programmer always has to write accessor methods, including code to check if the caller has the right permissions. LPC potentially allows this to be abbreviated by interceding on method invocation, however, and I think reflection (in particular, a powerful enough intercession mechanism) is the key to making security easier to implement.

4 A prototype next-generation collaborative language

One approach to implementing a better collaborative language would be to embed mechanisms for collaboration in an existing single-programmer language. Dynamically extensible languages such as Lisp or Smalltalk and their descendants would be best suited for this. This approach gives all the benefits of a full-featured general purpose language more or less for free; for example, you could write a simple program in CLOS[Ste90] that accepted connections on a socket and attached a read-eval-print loop to each connection in a separate thread, along with a small library of functions for publishing values to other connections' environments, and you would have a collaborative programming language with all the features described in section 2.1 except for resource control and security. Unfortunately, these two are very difficult to integrate with an existing language; for example, one can easily circumvent Common Lisp's module encapsulation and reference an internal symbol in another package by simply using two colons (i.e. `foo::bar`), and CLOS maintains a global class namespace—most CLOS functions allow the programmer to refer to a class by name (a symbol), rather than by value. One would have to write secure, multi-programmer-aware versions of practically every function in the CLOS library in order to be able to allow users to control access to their classes.

Java appears to be another good candidate for embedding collaborative features into, and in fact the Safe Language Kernel (SLK) project[vE] is working towards making Java into a collaborative programming language (in fact, a community programming language—one in which programmers might be sharing code and data in a single server process but not necessarily collaborating towards the same goal). Their J-Kernel[HCC+98] replaces the standard Java class loader with one that rewrites the bytecode of classes as they are loaded and interposes checks for security and (with JRes[CvE98]) resource control in every method. This approach seems to be workable, but seems awkward to implement, somewhat fragile and error-prone, and also inconvenient for the programmer, who has to reload a whole class (and replace all the instances of the old class with the new class) whenever a method or field is added or changed. And with Java's static typing, it seems like this could lead to a chain of recompilations and updating of instances. Perhaps if Java evolves to be a little more dynamic, this approach will be more attractive, but as it stand now it seems like a tough task to try to shoe-horn collaborative features into the language.

An alternate approach is, of course, to implement a new language from scratch. This has the advantage of complete control over the features; security and incremental update can be designed in from the start. It is also a good approach for pedagogical reasons: you can define a very clean, simple prototype language without needing to deal with all the issues that would be required for a full-featured practical language but are irrelevant to a collaborative programming language. However, even just implementing the features described in section 2.1 is quite a task! Multitasking and persistence, in particular, can be very tricky to implement, and while they are (practically speaking) required for a collaborative programming language, their implementation has little to do with the collaborative-ness. (More importantly, I just don't have the expertise needed to implement them in the time frame of my PhD!)

What I've decided on is a hybrid approach that gives me much of the best of both worlds. I have begun to implement a prototype language embedded in Larceny, an implementation

```
⟨definition⟩ ⟶ (define-object ⟨variable⟩ ⟨parents⟩)
    | (define-gf ⟨variable⟩)
⟨parents⟩ ⟶ ⟨object⟩*
⟨object⟩ ⟶ ⟨expression⟩

⟨expression⟩ ⟶ (add-method! ⟨gf⟩ (⟨spec formals⟩) ⟨body⟩)
    | (add-acceptor! ⟨gf⟩ (⟨spec formals⟩) ⟨rhs-variable⟩ ⟨body⟩)
    | (add-storage! ⟨gf⟩ (⟨spec formals⟩) ⟨default⟩)
    | (⟨gf⟩ ⟨actual arg⟩*)
    | (set! (⟨gf⟩ ⟨actual arg⟩*) ⟨expression⟩)
⟨gf⟩ ⟶ ⟨expression⟩
⟨spec formals⟩ ⟶ ⟨variable⟩ | (⟨variable⟩ ⟨specializer⟩)
⟨specializer⟩ ⟶ ⟨object⟩
⟨rhs-variable⟩ ⟶ ⟨variable⟩
⟨default⟩ ⟶ ⟨expression⟩
⟨actual arg⟩ ⟶ ⟨expression⟩ | (direct ⟨expression⟩ ⟨parents⟩)
```

Figure 1: The core syntax of BOB, as extensions to the Scheme syntax grammar.

of Scheme created by Lars Hansen, a fellow doctoral student at Northeastern. In Scheme, environments are truly encapsulated—values can only be accessed if they are bound to a name in the local environment, or returned by a procedure that the user already has access to—and can serve as the basis of a simple security kernel, as Jonathan Rees has outlined[Ree96]. The advantage of the Larceny implementation is that it provides multithreading and persistence (in the form of heap dumping and loading), as well as support for first-class environments.

4.1 BOB

So far, I've mainly concentrated on building an object system in Larceny suitable for a collaborative programming language; I decided to base it on the untyped subset of BeCecil[CL97], a simplified (and formalized) version of Cecil[C$^+$], hence the (working) name "BOB", which stands for "Based On BeCecil". The BOB object system is, like BeCecil, centered around generic functions (also known as multimethods) similar to those in CLOS and Dylan. It also provides a multiple-dispatch version of instance variables called storage tables, elegantly unifying fields and methods into the same lookup mechanism. Unlike BeCecil, however, everything is dynamically modifiable; methods and storage tables can be added to or deleted from generic functions, and the object inheritance relation can be mutated, all at runtime. Figure 1 summarizes the core syntax of BOB.

Objects are created with `define-object`, which makes a new object with the given name and parent(s) and binds a name to the new object in the current lexical scope. If no parents are specified, the new object inherits directly from `<object>`, the root of the inheritance hierarchy. Objects act as both classes and objects in class-based languages. For example, we can define classes for points, colored objects, and colored points with these definitions:

```
(define-object <point>)
(define-object <colored>)
(define-object <colored-point> <point> <colored>)
```

Since there is no difference between the inheritance and instance-of relationships, we can create an instance of `<point>` the same way that we created a subclass:

```
(define-object my-point <point>)
```

Generic functions are created with `define-gf`, which makes a new generic function with the given name and binds the name to the new generic function in the current scope. (Note that unlike CLOS, a generic function does not have a fixed number of arguments; methods with any number of arguments may be attached to a generic function.) Methods are attached to a generic function with `add-method!`, which creates a method and adds it to the generic function by side-effect. Each formal argument to a method has both a name and a specializer object. (A specializer object can be omitted if it is `<object>`.) The specializer objects determine which method is invoked when the generic function is applied; a method is applicable only if all of the arguments are descendants of the corresponding specializer objects. For example, assuming generic functions `x` and `y` have been defined to return the x and y coordinates of a point, we can compare two points with these methods:

```
(define-gf equal?)
(add-method! equal? ((p1 <point>) (p2 <point>))
  (and (equal? (x p1) (x p2)) (equal? (y p1) (y p2))))
(add-method! equal? ((i1 <number>) (i2 <number>))
  (= i1 i2))
```

When the generic function `equal?` is applied to two point objects, the first method is invoked, which in turn applies `equals?` to the two pairs of coordinates; each of these applications invokes the second method, which uses the Scheme primitive function `=` to compare the two numbers. Note that it would be an error to apply `equal?` to a point and a number; we could handle this case with a catch-all method that just called the Scheme primitive `eq?`, which only returns true if both arguments are the same:

```
(add-method! equal? (a b)
  (eq? a b))
```

In this case, the method's formal specializers are both `(<object>)`, so the method is applicable to all pairs of arguments, such as a number and a point. However, if `equal?` is applied to two points, there are now two applicable methods; in this case, the method that is more specific (i.e., whose specializer objects are closest to the arguments in the inheritance graph) is chosen, so the first method defined above is still chosen.

Now suppose we defined a method to compare colored objects, assuming a generic function `color` that retrieves the color value of a colored object:

```
(add-method! equal? ((c1 <colored>) (c2 <colored>))
  (equal? (color c1) (color c2)))
```

If we call `(equal?)` on two colored points, there are now three applicable methods, and two of them (the ones comparing points and comparing colored objects) are equally specific; in this case, an error occurs. The conflict can be resolved by adding a more specific method that handled colored points:

```
(add-method! equal? ((cp1 <colored-point>) (cp2 <colored-point>))
  (and (equal? (direct cp1 <point>) (direct cp2 <point>))
       (equal? (direct cp1 <colored>) (direct cp2 <colored>))))
```

The use of `direct` enables incremental inheritance, by re-applying the generic function to the arguments but constraining the applicable methods to those specific to the parent objects.

Now we show how to define the `x`, `y`, and `color` generic functions, using storage tables. A storage table associates keys to values, where a key is a tuple of objects and the value is the the value of the field attached to those objects. If the key consists of a single object, then this is equivalent to object fields (also known as instance variables or slots) in most other OO languages. A generic function can contain both storage tables and methods; reading a storage table uses the same syntax as applying a method, with the application arguments forming the key that is looked up in the table, and the value is returned. Each storage table also has a default value that is returned when a given key is not found.

```
(define-gf x)
(define-gf y)
(define-gf color)
(add-storage! x ((p <point>)) 0)
(add-storage! y ((p <point>)) 0)
(add-storage! color ((c <colored>)) 'black)
```

After the above definitions, our `my-point` object now has `x` and `y` fields, which can be accessed with `(x my-point)` and `(y my-point)`; since `my-point` has not been associated with any value in either storage table, they both return the default value, which is 0. Storage tables are modified by using an extension of Scheme's `set!` that instead of taking a variable on the left hand side, takes a generic function and a list of arguments:

```
(set! (x my-point) 3)
(set! (y my-point) 4)
```

(The syntax is modeled after Common Lisp's `setf`.) What assignment does is change the value associated with the tuple of arguments, or add an association if it doesn't already exist in the storage table.

In order to be able to override a storage table with a setter method without having to make the user switch between using the assignment syntax and using method application syntax, there is a special kind of method called an `acceptor` that is invoked with the same syntax as assignment to a storage table:

```
(define-object <pos-point> <point>)
(add-acceptor! x ((p <pos-point>)) new-x
  (if (< new-x 0)
      (error "can't set negative x")
      (set! (x (direct p <point>)) new-x)))
```

When an acceptor is assigned to, the value of the right hand side of the assignment expression is bound to the variable specified after the list of formal arguments (`new-x` in the above acceptor definition), which can then be referred to in the body of the acceptor.

To construct anonymous objects (to play the role of instances from class-based languages), we can simply define objects in a local scope that gets discarded:

```
(define-gf make)
(add-method! make ((parent <object>) . args)
  (define-object anon parent)
  (apply initialize anon args)
  anon)
```

We can then define methods on `initialize` that act like constructors:

```
(define-gf initialize)
(add-method! initialize (obj . args)
  ; default: do nothing
  )
```

```
(send ⟨username⟩ ⟨name⟩ ⟨value⟩)
(share ⟨name⟩ ⟨value⟩)
(get ⟨username⟩ ⟨name⟩)
(unshare ⟨name⟩)
(forget ⟨username⟩ ⟨name⟩)
```

Figure 2: Communicating values between users in BOB.

```
(add-method! initialize ((p <point>) (xx <number>) (yy <number>))
  (set! (x p) xx)
  (set! (y p) yy))
(add-method! initialize ((p <colored-point>) (xx <number>) (yy <number>) (c <symbol>))
  (initialize (direct p <point>) xx yy)
  (set! (color p) c))
```

4.2 Smud

The driver for BOB is called Smud (short for "Scheme mud", although it's not a mud—I'm still trying to think of a better name). Smud maintains a database of *user* objects; each user object contains a username, a password, and an environment. When a user connects to a Smud server with a valid username and password, a new thread is created and the socket is connected to a read-eval-print loop in that thread with the user object's environment. There are two big advantages to this arrangement: one is that each user has a separate environment, and thus a separate name space; the other is that a user's environment is persistent, so that you can disconnect from the server and later reconnect and still have access to all the environment definitions from the previous connection.

Initially, each user has an environment that consists of most of the standard library definitions, as well as special versions of user-specific definitions such as `current-input-port`. Users can communicate with each other by sending and publishing values; the interface is based on the "Shared Scheme" portion of MUSEME, a mud written in Scheme48[Win94], and is summarized in figure 2.

Each user has an inbox and an outbox; `(send u n v)` associates `n` with `v` from the current user in `u`'s inbox, and `(share n v)` associates `n` with `v` in the current user's outbox. `(get u n)` returns the value associated with `n` from user `u` in the current user's inbox, if it exists; otherwise it returns the value associated with `n` in user `u`'s outbox (or throws an exception if it's not there either). `unshare` and `forget` are used to remove associations from the current user's outbox and inbox, respectively.

For example, here is a simple bank account system (published by a user named `bank`:

```
(define-object <account>)
(define-gf balance)
(add-storage! balance ((acct <account>)) 0)
(define-gf transfer ((amt <number>) (from <account>) (to <account>))
  (if (> amt (balance from))
      (error "Insufficient funds")
      (begin
        (set! (balance from) (- (balance from) amt))
        (set! (balance to) (+ (balance to) amt)))))
(share <account> '<account>)
(share transfer 'transfer)
```

Since the `balance` generic function is not shared, no one can access account balances without going through the `transfer` method. A user can share his bank account object in various ways that protect access to it:

```
(define <account> (get 'bank '<account>))
(define acct (make <account>))
(send 'my-spouse 'acct acct)  ;; gives complete deposit and withdrawal access
(define-gf deposit ((amt <number>) (from <account>))
  ((get 'bank 'transfer) amt from acct))
(share 'deposit deposit)  ;; only gives deposit access
```

The name used to send or share a value is usually a symbol, but may be any Scheme value; this allows a user to share a value with multiple users (but not every user) by sharing it with a name that is a unique object, and then sending that object to the users he wants to allow access to the shared value. The value shared may also be any Scheme value, and will often be a procedure (or a generic function).

A value retrieved from another user may be assigned to a variable in the current user's environment (or otherwise made "local"), such as `<account>` in the above example, or all references to that value may be made through the inbox or outbox, such as `transfer` in the above example; the latter case allows it to be dynamically updated by the sending user, although even in the former case the value may be dynamically updated if, for example, the value is a procedure that refers to variables in the sending user's environment, which can be altered by `set!`, or if the value is a vector, which can be altered by `vector-set!`.

5 Summary and Future Work

What I've presented so far is the basic design for a new collaborative programming language, as well as descriptions and a comparison of the features of three existing languages. My thesis is that this language is a substantial improvement on these and other current languages. Before I can demonstrate that, I need to make some improvements to the language, as well as survey additional languages and related systems, and then I can compare the features of my language to the features of the best of the existing ones. In addition, my dissertation will include formal semantics for subsets of my language and some of the existing languages. (Appendix A has one such semantics, for a small subset of MOO.)

My strategy for expanding BOB and Smud is not simply to throw in a bunch of features that are useful to collaborative languages, but to add a reflection capability that is powerful enough to support implementation of all of the other features as extensions to the base language. This has two big advantages: the base language can remain very simple and elegant, and the various features can be implemented several different ways in order to experiment with different design ideas without having to change the whole language. A particular benefit of this latter feature is that different implementations can exist simultaneously, so that, for example, two groups of users can use two different security models for sharing code within each group, and the groups could use a third security model for interacting between groups. My plan is to spend more effort on designing a good reflection system, and to develop basic proof-of-concept implementations for the various collaborative features without spending too much time on any one in particular—many of them are active research areas in their own right.

I have two ideas for how to do reflection in BOB. One is to have a meta-object protocol (MOP) which reifies programming constructs into objects themselves, which lets the user extend the behavior of the language by defining new methods that override the base behavior. For example, if there were a generic `<handler>` object, then a `handler-applicable?` generic function might test a handler and a list of arguments to determine if the handler were applicable; you could then make a child of `<handler>`, `<private-handler>`, and attach a method to `handler-applicable?`

specialized on `<private-handler>` that returned false when the handler happened to be owned by another user. An alternative approach would be to use aspect-oriented programming (AOP), which was designed as a restricted but safer version of MOP; in AOP, a base program is combined with a set of aspects that augment the base behavior in a way that crosscuts multiple objects, and aspects can be selectively attached to different sets of objects. In the example above, you could define an aspect that inserted code at the beginning of all private methods that checked to see if the calling user were not the same as the owner, and did a re-send to the next applicable method if so.

It's unclear to me what the advantages and disadvantages are of the two approaches; AOP is meant to be safer, by not giving the user access to certain parts of the base functionality, but it may turn out not to be powerful enough to implement the collaborative features needed. I'm also not sure whether AOP is the best fit for a dynamic, multiple-dispatch language; the only examples I am familiar with are AspectJ[K^+] and Demeter/Java[L^+][LO97], both of which extend Java, whereas the main example of a MOP that I am familiar with is from CLOS[KdRB91], which is more akin to BOB. This will be a productive area of research, since the AOP field is currently very active, and newer versions of AOP might be able to solve the problem.

Of the collaborative features I identified in section 2.1, type safety, garbage collection, and multitasking (multithreading) come for free by embedding the language in Larceny. (Note that an object becomes garbage as soon as it has no pointers to it in anyone's environment—there is no global namespace that holds onto all objects regardless of other references, as there is in MOO, ColdC, and LPC.) A simple form of persistence also comes for free, in that Larceny has a `dump-heap` procedure that saves the entire state of the system to a file, which can be reloaded when starting Larceny by giving the heap file name as a command line argument. This allows simple periodic checkpointing, similar to MOO, and I believe this is sufficient for most needs (and far simpler to implement than any alternative I can think of). I have already addressed reflection, available through the MOP.

This leaves resource control and security. Controlling how much time a user's task uses is much less important in a multithreading system than in a cooperative multitasking system like MOO, ColdC, or LPC; assuming the thread scheduler is fair, this is not an issue, although a mechanism for adjusting thread priorities (using Larceny's ability to set the interrupt timeslice) is worth exploring. Controlling how much memory a user allocates can easily be done through the MOP by overriding the `make-object` function, as well as providing protected versions of Scheme functions that allocate memory, such as `cons` and `make-vector`. Similarly, a simple capability-based security can be implemented through the MOP using the "seal" idea from Rees's W7[Ree96]; alternative systems that keep track of a task's permissions (similar to MOO) would be interesting to explore, though I have a less clear idea of how to implement that through the MOP.

Other languages I intend to study include Muq[Pro] (a mud that actually has three different languages, based on C, Lisp, and Forth), TinyMUSH[MUS] (a mud that allows programming based on scripting of commands), POO[Str] (a Python-based mud), coolmud[Whi] (a distributed OO mud language, written by the creator of MOO), FMPL[Blo] (a general-purpose collaborative language somewhat like ColdC), and E[MS] (a scripting language for secure, persistent, distributed programming). There are also some libraries for existing general-purpose languages that are useful from a collaborative programming standpoint, such as SchMUSE[BCLZ93] (Scheme) and CL-HTTP[Mal94] (Common Lisp), as well as inter-language systems like CORBA[Obj].

A FM: Featherweight MOO

A.1 Purpose

A.2 Syntax

```
P -> "return" E ";" P -> E ";" P

E -> "#-1" | "#0" | "#1"                           // constants
E -> "this" | "args" "[" INT "]"                   // variables
E -> E ":" ID "(" [ EL ] ")"                       // verb invocation
E -> "create" "(" E ")"                            // object creation
E -> "set_verb_code" "(" E "," STR "," ESTR ")" // verb update

[Note: set_verb_code has different syntax and semantics in full MOO,
but a MOO database can be made to accept this syntax and act as
defined below with the help of a #0:bf_set_verb_code wrapper.]

EL -> E | E "," EL
ESTR -> E | STR

INT -> DIGIT+
STR -> "\"" CHAR* "\""
ID -> LETTER (LETTER|DIGIT)*
```

A.3 Semantics

```
DB = Objnum -> (Objnum x Verbs)
Verbs = Str -> P
Objnum = Nat + { T }
Nat = set of natural numbers
Str = set of strings
```

Objects are numbered with unique natural numbers; T is ”top”, representing the top of the inheritance hierarchy (the parent of the root object(s)), which is expressible as ”#−1” (object negative-one). A database (DB) is a partial function over objnums such that DB(o) = ¡p, v¿ where p is the parent of o and v is the verb map of o. A verb map is a partial function mapping verb names (strings) to programs.

```
db0 \in DB
db0(0) = <1, {}>
db0(1) = <T, {}>
```

The starting database, db0, contains two objects, #0 and #1, the latter being the parent of the former. Neither object has any verbs. Note that we could just as well start with one object (or none if general objnums were expressible in the syntax), but this reflects the standard initial MOO database where #0 is the system object and #1 is the root object.

```
K -> halt
K -> <[]; p, K>
K -> <[]:id(e1...en), K>
K -> <v:id(v1..vi-1,[],ei+1...en), K>
K -> <create([]), K>
K -> <svc([],s,es), K>
```

```
K -> <svc(v,s,[]), K>
```

K is the set of continuations. A compound continuation is a partly-evaluated expression with a hole in it, plus a next continuation; thus, a continuation represents a stack of expressions waiting to be completed.

```
Config = (P + E + Objnum) x K x DB
```

A configuration is a triple whose first element is either a program or expression to be evaluated, or a value to be passed to the continuation.

```
eval : Config -> (Objnum x DB)
```

The evaluation function maps configurations to configurations. In the set of recursive definitions below, v stand for any element of Objnum, db stands for any element of DB, e stands for any element of E, p stands for any element of P, k stands for any element of K, id stands for any element of ID, s stands for any element of STR.

Evaluations of programs and expressions:

```
eval(e; p, k, db) = eval(e, <[]; p, k>, db)
eval(return e, k, db) = eval(e, k, db)
eval(\#-1, k, db) = eval(T, k, db)
eval(\#0, k, db) = eval(0, k, db)
eval(\#1, k, db) = eval(1, k, db)
eval(e:id(e1...en), k, db) = eval(e, <[]:id(e1...en), k>, db)
eval(create(e), k, db) = eval(e, <create([]), k>, db)
eval(set_verb_code(e,s,es), k, db) = eval(e, <svc([],s,es), k>, db)
```

Applications of continuations to values:

```
eval(v, halt, db) = <v, db>
eval(v, <[]; p, k>, db) = eval(p, k, db)
eval(v, <[]:id(e1...en), k>, db) = eval(e1, <v:id([],e2...en), k>, db)
eval(v, <v0:id(v1...vi-1, [], ei+1...en), k>, db)
    = eval(ei+1, <v0:id(v1...vi-1,v, [], ei+2...en), k>, db)
eval(v, <v0:id(v1...vn-1, []), k>, db)
    = eval(p[v0/this,v1/args[1]...vn-1/args[n-1],vn/args[n]], k, db)
    where lookup(db,v0,id) = p
    [In English: p is the program code of the verb named id on the v0
    object (or one of its ancestors); execute it, after replacing all
    occurences of "this" with v0 and "args[i]" with the corresponding vi.]
eval(v, <create([]), k>, db) = eval(vnew, k, db + {vnew:<v,{}>})
    where vnew = max(dom(db))+1
eval(v, <svc([],s,e), k>, db) = eval(e, <svc(v,s,[]), k>, db)
eval(v, <svc(v0,s,[]), k>, db) = eval(v0, k, dbnew)
    where dbnew = svc(db,v0,s,return v;)
    [This simulates a property with a verb that returns the value.]
eval(v, <svc([],s1,s2), k>, db) = eval(v, k, dbnew)
    where dbnew = svc(db,v,s1,parse(s2))
```

```
lookup : (DB x Objnum x ID) -> P
```

The auxiliary function lookup takes an object and verbname and looks up the inheritance chain of the object until it finds a verb definition, returning the corresponding program.

```
lookup(db, v, id) = verbs(id) if v \in dom(db) and
                                 db(v) = <vp,verbs> and
                                 id \in dom(verbs)
lookup(db, v, id) = lookup(db, vp, id) if v \in dom(db) and
                                          db(v) = <vp,verbs> and
                                          id \not\in dom(verbs)

svc : (DB x Objnum x Str x P) -> DB
```

The auxiliary function svc takes an object, verbname, and program and adds the program as the verbname on the object, replacing the current program if there is one. Note that this does not go up the inheritance chain; it always modifies the object itself.

```
svc(db, v, s, p) = db - {v:<vp,verbs>} + {v:<vp,verbsnew>}
    where verbsnew = verbs - s:pold + s:p
    if db(v) = <vp,verbs> and (verbs(s) = pold or s \not\in dom(verbs))

parse : Str -> P
```

The auxiliary function parse converts strings to programs, using the syntax grammar given in the previous section.

Thus, starting from db0, a stream of input programs can be evaluated in sequence, producing a stream of results, modifying the database by side-effect as it runs. Note that "this" and "args[i]" are meaningless in top-level input programs; they are evaluated by substitution when a verb program is invoked.

```
eval(P1, halt, db) = <v, dbnew>
-------------------------------
run(db,P1,P2...) = v,run(dbnew,P2...)
```

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996. `http://www.luca.demon.co.uk/TheoryOfObjects.html`.

[AIW99] Benjamin Pierce Atsushi Igarashi and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Proceedings of ACM Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 34, pages 132–146. ACM SIGPLAN Notices, October 1999. `http://www.graco.c.u-tokyo.ac.jp/~igarashi/papers/oopsla99.html`.

[BCLZ93] Michael R. Blair, Natalya Cohen, David M. LaMacchia, and Brian K. Zuzga. MIT SchMUSE: Class-Based Remote Delegation in a Capricious Distributed Environment. AI Memo 1547, MIT, February 1993. `ftp://ftp.swiss.ai.mit.edu/pub/users/ziggy/Papers/Projects/SchMUSE/`.

[Blo] Jon Blow. FMPL. FTP directory. `ftp://ftp.xcf.berkeley.edu/pub/src/local/fmpl/`.

[C+] Craig Chambers et al. Cecil language. Project web page. `http://www.cs.washington.edu/research/projects/cecil/www/cecil.html`.

[CL97] Craig Chambers and Gary Leavens. BeCecil, a Core Object-Oriented Language with Block Structure and Multimethods: Semantics and Typing. In *Proceedings of the The Fourth International Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, Paris, France, January 1997.
`http://www.cs.washington.edu/research/projects/cecil/www/Papers/BeCecil.html`.

[Cow99] Andrew Cowan. MUD FAQ. Web page, August 1999.
`http://www.mudconnect.com/mudfaq/`.

[Cur97] Pavel Curtis. *LambdaMOO Programmer's Manual for version 1.8.0p6*, March 1997.
`http://www.moo.mud.org/`.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A Resource Accounting Interface for Java. In *Proceedings of 1998 ACM OOPSLA Conference*, Vancouver, BC, October 1998.
`http://www.cs.cornell.edu/slk/papers.html`.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, May 1990.
`http://www.research.att.com/~bs/arm.html`.

[Gil] Brandon Gillespie. *ColdC 1.1 Reference Manual*.
`http://www.cold.org/`.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
`http://java.sun.com/docs/books/jls/`.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.
`http://www.cs.cornell.edu/slk/papers.html`.

[K+] Gregor Kiczales et al. AspectJ. Project web page.
`http://aspectj.org`.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.
`http://www.parc.xerox.com/csl/groups/sda/publications/papers/Kiczales-AMOP/`.

[L+] Karl Lieberherr et al. Demeter. Project web page.
`http://www.ccs.neu.edu/research/demeter/`.

[Lesa] Rob Leslie. DGD: Dworkin's Generic Driver. Project web page.
`http://www.mars.org/home/rob/dgd.html`.

[Lesb] Rob Leslie. LPMOO: Simulating MOO with DGD. Project web page.
`http://www.mars.org/home/rob/proj/lpmoo/`.

[LO97] Karl J. Lieberherr and Doug Orleans. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.
`http://www.ccs.neu.edu/research/demeter/biblio/demjava.html`.

[Lou] Peter William Lount. Smalltalk.org. Web page.
`http://www.smalltalk.org/`.

[Mal94] John C. Mallery. A Common LISP Hypermedia Server. In *Proceedings of The First International Conference on The World-Wide Web*, Geneva, May 25 1994. CERN. `http://www.ai.mit.edu/projects/iiip/doc/cl-http/server-abstract.html`.

[MS] Mark S. Miller and Jonathan Shapiro. ELib and the E Language. Project web page. `http://erights.org/`.

[MUS] TinyMUSH 3.0. Web page. `http://www.godlike.com/3.0/`.

[Obj] Object Management Group (OMG). *Corba for beginners.* `http://cgi.omg.org/corba/beginners.html`.

[Pro] Jeff Prothero. Muq. Project web page. `http://www.muq.org/~cynbe/muq/`.

[Ree93a] George Reese. *Intermediate LPC*, November 1993. `http://www.mars.org/home/rob/docs/descartes-index.html`.

[Ree93b] George Reese. *LPC Basics*, April 1993. `http://www.mars.org/home/rob/docs/descartes-index.html`.

[Ree96] Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. AI Memo 1564, MIT, 1996. `http://mumble.net/jar/pubs/secureos/`.

[Ste90] Guy L. Steele. *Common Lisp The Language.* Digital Press, 2nd edition, 1990. `http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html`.

[Str] Joe Strout. *POO Programmer's Reference.* `http://www.strout.net/python/poo/`.

[vE] Thorsten von Eicken. SLK: The Safe Language Kernel Project. Project web page. `http://www.cs.cornell.edu/slk/`.

[Whi] Stephen White. coolmud. Personal web page. `http://www.csclub.uwaterloo.ca/u/sfwhite/`.

[Win94] Dan Winship. *MUSEME User's Manual*, August 1994. `http://www.ccs.neu.edu/home/dougo/museme/`.