GL - A Denotational Testbed with Continuations and Partial Continuations as First-Class Objects

Gregory F. Johnson

University of Maryland

College Park, MD 20742

1. Abstract

In this paper we describe GL, a language designed to support interactive experimentation with denotational semantics of programming languages, and the novel features of its interpreter. GL is an expressional language that might best be described as an implementation of lambda calculus augmented with several useful basic data types including l-values.

A unique aspect of the GL environment is that it presents a visible, user-accessible implementation of the continuation semantics of GL. The user is expected to understand a denotational definition of GL, and to interact with the system in terms of that definition. In particular, if a computation is temporarily halted the expression continuation extant at that point can be interactively captured and later applied to other values and stores. The implementation of this feature is via a pair of routines called *setjmpup* and *longjmpup* that provide what might be called a partial continuation facility. A partial continuation is a function over stores or store/value pairs that represents execution of a partially executed program from its current state to some later state possibly before its halt state. The semantics of partial continuations is interesting, and an extension of GL is presented that contains continuations and partial continuations as first-class objects.

The GL environment is fairly complete; it has an experimental polymorphic type inference mechanism that supports self-application and reports likely sources of user error in a robust manner, and it has a flexible breakpoint and trace facility that permits program execution to be observed and controlled at a variety of levels of granularity. Moreover, it has been used successfully to teach a graduate course in Theory of Programming Languages.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1987 ACM 0-89791-235-7/87/0006/0165.......75¢

2. Continuation Semantics as a Basis for a Program Development Environment

Denotational semantics [1] has proved to be a versatile tool for many aspects of programming language research. We explore the notion that a denotational language definition, in particular one based on continuations, is also useful as a basis for user interaction with a programming environment. The environment is for a new strongly typed, expressional language called GL (for Gedanken-like language after Reynolds's language [2]).

The environment is designed around a continuation semantics for GL, and the user of the environment interacts with the computer in terms of this denotational definition. At any point when execution of a program has been stopped, the user can interactively obtain a continuation for the computation in progress and bind it to an identifier in the programming environment. The captured continuation can then be applied to a variety of arguments in an exploratory manner. The use of continuations as first-class objects in a programming language is extremely useful; the language Scheme [3] exemplifies this principle. For a discussion of the flexibility and power offered by the presence of continuations in a language see [4]. We present here a notion of partial continuations, discuss the denotational semantics of a language that has them as first-class objects, and indicate a method for their efficient implementation in an imperative systems implementation language. Related work on partial continuations is being conducted by Friedman, et.al. at the University of Indiana [5].

The normal mode of use is for users to capture continuations in their programs at various interesting points and apply them in a flexible, interactive way to a variety of arguments and stores. Since GL permits functions to be used as fully general first-class objects, applications of captured continuations can be imbedded in arbitrary GL program fragments. A user could thus easily write a small program to apply a given continuation to all odd integers between 100 and 1000, and algorithmically determine if his program behaves as desired under those circumstances.

Efficient implementation of an interactive continuation-based environment required a novel approach to the internal design of the execution component of the system. The GL environment provides a Read-Eval-Print loop that permits recursive invocations of itself. Thus the runtime activation record stack consists of an activation of Read followed by some number of invocations of Eval, another activation of Read followed by more invocations of Eval, etc. From the point of view of implementation a continuation is simply a sequence of Eval activation records between two successive activations of Read.

As mentioned above we implementated two procedures that implement what might be called partial continuations. The routines are called *setjmpup* and *longjmpup* to suggest an analogy with the Unix system calls *setjmp* and *longjmp* . *Setjmpup* captures a continuation from after its point of call to the completion of one of its ancestors in the called-by relation. This is accomplished by capturing some number of activation records off the top of the runtime stack.

Longjmpup evaluates such a continuation by concatenating a vector of activation records onto the runtime stack. This mechanism provides an efficient way to implement dynamically obtainable continuations; if the user assigns the continuation of a function he is executing to a variable, the object assigned to the variable is the stack of Eval frames down to the previous Read frame. This technique has proved to be feasible within a relatively low-level implementation language (Pascal) and to be adequately efficient.

3. Continuations and partial continuations as first-class objects

The GL interpreter makes use of the routines *setjmpup* and *longjmpup* to achieve a degree of efficiency in implementing interactively capturable GL continuations. The semantics of *setjmpup* and *longjmpup* have themselves proved interesting; we understood them originally only in operational terms as a technique useful at the implementation level, and understanding them semantically proved challenging. We would like eventually to define an environment written in GL that implements user-accessible continuations semantics, much like the classic Lisp interpreter written in Lisp. So, it seems natural to augment GL and its denotational semantics with a feature analogous to *setjmpup* and *longjmpup* . This amounts to introducing continuations as first-class objects into the language. The result is a powerful device that introduces into an expressional language a facility that can be used to create such effects as co-routines and catch/throw error recovery.

We recently heard that a large project for an ML machine is about to be undertaken in Europe [6]; if an expressional language such as ML were to be used as a system implementation language, some form of co-routine mechanism such as the one we suggest here for GL would be desirable.

As expression continuations are perhaps a bit less intuitive than command continuations, we first give a semantics for a simple statement-oriented language that has continuations as first-class objects. The syntax follows:

```
Pgm ::= S
S ::= a := E
S ::= if b then S1 else S2
S ::= S1 ; S2
S ::= exit                        {stop the program or skip ahead to the next
                                   pending fork statement if one exists}

S ::= a := CurCont
```

The statement 'a := CurCont' means that the current command continuation should be stored in 'a', i.e. the address of the next sequential intruction should be put in 'a'.

S ::= continue a {resume execution at the address stored in a}

S ::= fork a

The statement 'fork a' causes execution to proceed to the address stored in 'a'. When the program stops because of ann 'exit' statement or because execution advances past the last statement, execution continues after the 'fork' statement.

Unusual features of this language are a statement that captures a command continuation into a variable and two ways to invoke such a command continuation. If this simple language were to be implemented, 'a := CurCont' would cause the address of the next sequential instruction to be put in 'a'; 'continue a' would cause the address in 'a' to be put into the program counter. 'Fork a' would cause the address of the next sequential instruction to be pushed on 'fork stack' and then the contents of 'a' to be put in the program counter. Under such an implementation, whenever the program stops (by executing an 'exit' or by advancing past the last statement) the fork stack is examined. If it is nonempty, the top element is popped into the program counter. Otherwise, execution of the program is complete. We now give a continuation semantics for the above language. In what follows we suppress type considerations. In particular, the domain Value is a disjoint sum, and so properly we should have all sorts of injection and projection operations in our equations, together with type consistency tests and error values. The addition of all of this is left to the interested reader. Its omission makes for a much clearer focus in the equations on what is really of interest.

Mp: Syntax → Store
Ms: Syntax → Store → Cont → Store
Me: Expr-Syntax → Store → Value { this is not elaborated }
Mn: Int-String → Integer

Cont: Store → Store
Store: Ident-String → Value

Value: Cont + Integer + (any other basic types)

Mp[[S]] = Ms[[S]] (λ id.0) (λ s.s)

Somewhat arbitrarily we initialize all identifiers to contain the value 0. We could as well initialize to bottom.

Ms[[a := e]] s c = c s[a ← Me[[e]] s]
Ms[[if b then S1 else S2]] s c = if Me[[b]] s then Ms[[S1]] s c else Ms[[S2]] s c
Ms[[S1 ; S2]] s c = Ms[[S1]] s (λ s1.Ms[[S2]] s1 c)

The above definitions are standard. The last four are of more interest:

Ms[[exit]] s c = s
Ms[[a := CurCont]] s c = c s[a ← c]

This expression involves a form of self-application; the continuation 'c' is applied to a store in which 'c' itself has been bound to an identifier in the store. If effect, 'a' becomes a label. Before 'a' can be branched to, however, the statement involving definition of the label must be executed. This is similar to *setjmpup* ; *setjmpup* captures an execution context that can later be used by *longjmpup* , but obviously one can't re-enter an execution context until *setjmpup* has been executed to capture the context.

Ms[[continue a]] s c = (s a) s

One obtains from the store the continuation bound to 'a' and then applies it to the current store. The pair of statements 'a := CurCont' and 'continue a' has a very simple semantics that contrasts with the semantics of labels and arbitrary goto's. The latter requires a complex application of the paradoxical combinator to a functional over vectors of continuations. The heart of the paradoxical combinator is of course a clever use of self-application, and in our simple semantics given above we also rely on an indirect form of self-application, in which a continuation is applied to a structure that has the same continuation imbedded in it.

Ms[[fork a]] s c = c((s a) s)

Here we simply compose the continuation of the fork statement with the continuation bound to 'a', and apply the resulting larger continuation to the current store.

We now consider an expressional language with first-class functions, and augment it with continuations as above. The resulting language is considerably more subtle than the above simple statement-oriented language, but its continuation semantics turns out to be surprisingly natural. We should mention that it took a significant amount of work to achieve such a natural semantics; we reformulated the language several times before a clean denotational definition for it could be constructed.

A syntax for the language follows:

```
Pgm ::= e
e ::= Int
e ::= ident
e ::= e + e                      { a representative binary operation }
e ::= func id.e                  { an unnamed function whose formal parameter is 'id' }
e ::= e1 ( e2 )                  { apply e1 as a function to e2 }
e ::= if e1 then e2 else e3 fi
e ::= bind id to e1 in e2 end    { statically scoped declaration }
e ::= new e                      { allocate a new location and initialize it with the value of e }
e ::= e1 := e2                   { put the value of e2 in the storage location specified by e1 }
e ::= e .                        { obtain the value in the storage location specified by e }
```

Note: a conventional constant declaration would be

bind p to 31415 in p * 3 * 3 div 1000 end whereas a variable declaration (with initialization) would be declared as

```
bind r to new 4 in p * r. * r. div 1000 end
```

Finally, the novel features of the language:

```
e ::= exit e1
```

Stop the program, returning e1 as the result of its execution. As in the previous example, this construct is meant to be executed primarily when a 'fork' is pending, and is used to implement partial continuations.

```
e ::= CurCont
```

This expression evaluates to its own expression continuation. As in the previous language its semantics entails a form of self-application. The expression 'a := CurCont' puts a marker in 'a'; at some later time execution can be resumed at that point.

```
e ::= continue e1 ( e2 )
```

This expression applies e1 to e2 (and the store resulting from evaluation of e1 and e2). Operationally speaking, control is transferred to the 'CurCont' that e1 produces, never to return. This can be used to implement a Lisp-like catch/throw mechanism, for instance. It is of course much more powerful; it can be used to jump into the middle of a function activation that has long since completed execution. This is precisely the effect of *longjmpup* . As for the value of e2, that is the value of the expression 'CurCont' in the resumed expression. So, if we execute 'a := CurCont' and later execute 'continue a. (4)', we jump back into the middle of the assignment, assign 4 to 'a', and proceed from there. This would of course prevent us from ever resuming using 'a' again, which leads to the following idiom:

```
FellInto := true;
a := CurCont;
if FellInto then
   FellInto := false;
   b := a;
else Ignore
fi;
```

Then, we execute 'continue b (5)'. 'B' is the mechanism for invoking the continuation and 'a' becomes in effect the formal parameter of the invocation.

Finally, we have fork expressions:

```
e ::= fork e1 ( e2 )
```

The continuation to which e1 evaluates is applied to the value of e2; when the program stops and produces a value, that value is returned as the value of the fork expression. When a fork is performed, it is conventional to expect the program to stop because of execution of an 'exit (e)', and so the value of e is transferred to the fork expression and returned as its value.

We now give the interesting features of the denotational semantics of the above language:

Mp: Syntax $\rightarrow$ Value
Me: Syntax $\rightarrow$ Envt $\rightarrow$ Store $\rightarrow$ Cont $\rightarrow$ $<$Value $\times$ Store$>$
Mn: Int-String $\rightarrow$ Integer
Cont: $<$Value $\times$ Store$>$ $\rightarrow$ $<$Value $\times$ Store$>$
Envt: Ident-String $\rightarrow$ Value
Store: LValue $\rightarrow$ Value
Value: Cont + LValue + Int + ... (other basic data types)

We use the notation <e1, e2> for ordered pairs of values and fst(e), snd(e) to obtain the first (resp. second) component of e, which should an ordered pair.

Me[[Int]] e s c = c < Mn[[Int]], s>

Apply the continuation to the pair consisting of the given integer and the store.

Me[[CurCont]] e s c = c <c, s>

Apply the continuation to a pair consisting of the continuation itself and the given store. Thus, 'CurCont' can be used much more generally and flexibly than as the right-hand side of an assignment operation.

Me[[func x.e1]] e s c = c < λ c1.λ <v1,s1>.Me[[e1]] e[x $\leftarrow$ v1] s1 c1, s >

'Func x.e' denotes a function from continuations and value/store pairs to value/store pairs.

Me[[e1 (e2)]] e s c =
    Me[[e1]] e s (λ <v1,s1>.Me[[e2]] e s1 (λ <v2,s2>. v1 c <v2,s2>))

(f * g represents function composition, i.e. g(f x))

As usual in continuation semantics, if there are two sub-expressions (e1 and e2 in this case), one creates a continuation using the meaning of the second expression and gives it as an argument to the meaning function applied to the first function. Here 'first' and 'second' indicate the order in which the expressions would be evaluated, which matters in the presence of an updatable store. The value to which e1 evaluates, namely v1, is applied to the application's continuation and the value/store pair produced by evaluation of e2. The semantics of application is applicative order; in particular if either e1 or e2 evaluates an 'exit' expression the application does not actually take place. Similarly, if the body of e1 executes an 'exit', the continuation of the application is ignored.

Me[[exit e]] e s c = Me[[e]] e s (λ <v,s>.<v.s>)

We evaluate e with respect to the identity continuation, discarding the remainder of the program up to any pending 'fork' expressions.

$$
\begin{aligned}
&\text{Me}[[\text{ continue e1 (e2) }]] \text{ e s c} = \\
&\qquad \text{Me}[[\text{ e1 }]] \text{ e s } (\lambda <\text{v1,s1}>.\text{Me}[[\text{ e2 }]] \text{ e s1 v1 })
\end{aligned}
$$

Here, we evaluate e1 to obtain a continuation v1, and then we evaluate e2 using v1 as its continuation.

$$
\begin{aligned}
&\text{Me}[[\text{ fork e1 (e2) }]] \text{ e s c} = \\
&\qquad \text{c (Me}[[\text{ e1 }]] \text{ e s } (\lambda <\text{v1,s1}>.\text{Me}[[\text{ e2 }]] \text{ e s1 v1 }))
\end{aligned}
$$

We evaluate e1 to obtain a continuation v1, and then evaluate e2 using the value of e1 as the continuation for the evaluation of e2. After that computation exits, we resume by applying the continuation of the fork expression to the result. Thus, the third argument 'c' to the meaning function Me is actually a partial continuation, reflecting the future of a compuation up to the next pending 'fork' expression. The 'fork' expression is in effect an expressional version of the *setjmpup* and *longjmpup* implementation routines.

4. The GL interpreter

The GL interpreter is written in Berkeley Pascal and presently runs on VAX and Sun systems running 4.3 Berkeley Unix. Since GL has first-class functions, the interpreter maintains a cactus stack of activation records, and periodically garbage collects unreachable ones. A routine named Parse reads from the keyboard and creates an abstract syntax tree. The parser then invokes an evaluator that recursively walks the abstract syntax tree to produce a value. There are two conditions that can cause evaluation to stop: evaluation-time errors and 'exit' expressions. Pascal nonlocal goto's are used to handle these situations. So, Parse actually calls a routine named CatchError which consists of not much more than a label that the recursive evaluator can branch to and a call to a routine named CatchExit. CatchExit is also just a shell that has a label for the recursive evaluator. CatchExit invokes DoEval, the routine that actually performs evaluation. As will be seen, the separation of CatchError and CatchExit is crucial for the implementation of continuations in GL.

The main novel feature of the interpreter is the implementation of continuations and partial continuations in GL. When DoEval is asked to compute the value of a component of the abstract syntax tree that turns out to be CurCont, it calls *setjmpup* to obtain the stack of AR's down to and including the first one for CatchExit. This object is the value returned by DoEval. When a 'continue e1(e2)' expression is executed, the values of e1 and e2 are determined. E1 is an AR stack (otherwise an error is indicated). Its top component is modified to return the value of e2, and then a *longjmpup* is performed using that AR stack. When control returns to the instance of DoEval that performed the *longjmpup* , that invocation of DoEval performs a nonlocal branch to its containing CatchExit activation instance to terminate execution. On the other hand, if a 'fork e1(e2)' is performed DoEval does not execute a nonlocal branch to its CatchExit; it proceeds normally with evaluation of the abstract syntax tree.

From the above discussion, it becomes clear that in our implementation a continuation corresponds to a sequence of DoEval AR's followed by a CatchExit AR. Composition of continuations is implemented as concatenation of sequences of AR's, and ignoring a continuation is accomplished by a nonlocal branch to an instance of CatchExit. Termination of execution because of an error condition is accomplished by a branch to CatchError; above the top Parse AR there is only one CatchError AR, whereas there may be many CatchExit AR's, if there are several fork expressions pending.

5. Overview of GL programming facilities

GL has several features that have proved helpful to users, some of which are absent from other implementations of similar languages such as the ML implementation from ATT Bell Labs that we have used. These include:

(1) pretty printing of closures

(2) function tracing, step-wise execution, and breakpoints

(3) symbol table tree examination and modification

(4) application of the interpreter to strings (interp 'a := 12');

(5) a facility for parsing strings into functions that represent abstract syntax. This last is particularly convenient when experimenting with denotational language definitions.

(6) the ability to choose between call-by-value and call-by-name semantics, dynamic versus static scoping, and automatic or explicit formation of closures.

Before exiting, each activation of DoEval determines if a pause in execution should occur; if so, DoEval invokes Parse, which has the effect of opening up the keyboard to the user. Before invoking the parser, however, DoEval places in a global variable named LastCont the result of a *setjmpup* operation that captures AR's down to the highest CatchError AR. a variable named LastPrefix receives the stack of AR's down to the highest CatchExit AR. So, if the user chooses to do so he can assign the value of LastCont to some other variable and obtain the continuation that is active at that point in the program. Also, the partial continuation corresponding to execution of the program ignoring any pending fork expressions is available in the variable LastPrefix.

The user can mark a function as traceable:

```
SelfAp := func * x.x x
```

If tracing is turned on, argument binding takes place, the new activation record is created, and then execution stops. The user can then use facilities in the environment to wander through the activation record tree, examining and if desired modifying values. Users of GL have indicated that the flexibility of controlled execution and examination of the environment are extremely helpful in getting programs to work. Also helpful is that fact that if a variable that the user asks the system to print contains a closure, the function is pretty-printed, together with a 'short-form' listing of the contents of any variables in the corresponding environment.

6. Implementation of *setjmpup* and *longjmpup*

As indicated previously, *setjmpup* copies a number of stack frames off the top of the runtime stack into an array that is passed to it as a *var* parameter. *Setjmpup* is given an integer argument *flag* , and it uses the flag to determine which AR is the last one it should save. It copies all AR's down to and including that AR. *Longjmpup* takes an array returned by *setjmpup* and places it on top of the runtime stack, and then branches to the return point of the corresponding *setjmpup* . So, *longjmpup* is similar to the Unix call *longjmp* in that it does not directly return to its caller; rather it causes a nonlocal branch. Unlike *longjmp* , *longjmpup* causes a change in the runtime stack that effects what amounts to an upward funarg. The last restored AR has its return address modified so that the routine returns to the code following the *longjmpup* call.

The essential difficulty in implementing *setjmpup* and *longjmpup* is that pointers to stack objects may be imbedded in the vector of stack frames that are saved and restored. The two most important examples of this are dynamic and static links and *var* parameters. We will briefly describe the implementation of *setjmpup* and *longjmpup* , focusing on the solution to this problem.

The data stored by *setjmpup* are the following:

(1) The length of the saved stack in machine words

(2) The resume address of the call to *setjmpup*

(3) The stack top address before the call to *setjmpup*

(4) The number of activation records saved

(5) The contents of non-volatile registers before the call to *setjmpup*

(6) The saved stack, modified as described below to allow for the proper treatment of pointers to stack objects

Each activation record that may be captured by *setjmpup* also must have a packet of information. Our current implementation requires that this information be contained in a record that is the first parameter of the activation; this is all information that is readily available at compile time, and could be automatically included by a compiler. Since the information is relatively easy to include at the source language level, we have not modified the compiler to produce the information automatically. The required information is the following:

(1) A tag indicating that the procedure is set up to be captured by *setjmpup*

(2) A flag used by *setjmpup* in determining whether this is the last AR it should capture

(3) A bit vector indicating which longwords in the AR contain addresses pointing to objects in the stack

(4) An indication of where the boundary is between this AR and that of its caller; this is given as an offset from the dynamic link field of the current AR and is used to determine the size of the last AR so that it can be saved properly

When *setjmpup* is called, it uses the bit vector in each AR to find and examine each pointer to a stack object in the AR. A pointer either refers to an object that is in the set of vectors being saved or to an object in the lower portion of the stack that is not being saved. In the former case no action is required. In the latter case, the address is replaced by a longword that contains two pieces of information:

(1) An identifying tag for the procedure containing the object pointed to (this is basically a numerical encoding of the name of the routine)

(2) The offset of the object pointed to in the AR that contains it

When *longjmpup* is called it restores the saved stack, modifying the last return address so that a return to the point after the *longjmpup* call will be effected when that last routine returns. It then processes each AR, examining stack pointers. If a stack pointer refers to an object in the restored stack a constant is added to it reflecting the difference between the original memory position of the stack when it was saved and its current position. Otherwise, the appropriate destination for the pointer is obtained by searching AR's starting from the top AR before the call to *longjmpup* until one is found with the desired key. The pointer is then set to have the desired offset within that AR. This amounts to a form of shallow static binding; it is desirable in our application, because the set of AR's extant when the *setjmpup* is executed may differ from the set of AR's extant when *longjmpup* gets called.

7. Conclusions

GL represents a successful experience with providing continuations as first-class objects in an expressional programming language and its environment. An efficient implementation involved system-level primitives for saving and restoring execution contexts. Our experience leads us to recommend our implementation techniques and programming environment features to implementors of similar systems.

8. Acknowledgements

Alan Demers was very helpful as a sounding board in formulating the ideas contained in this paper. Dominic Duggan has been extremely helpful in many discussions about gl and partial con-

tinuations.

References

1. Scott, D., "Data Types as Lattices," *SIAM Journal of Computing* 5(1976).

2. Reynolds, J. C., "GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept," *CACM* **13** pp. 308-319 (May 1970).

3. Sussman, Gerald Jay and Guy Lewis Steele, Jr., "Scheme: an interpreter for an extended lambda calculus," MIT Artificial Intellegence Memo 349 (December 1975).

4. Friedman, Daniel P. and Christopher T. Haynes, "Constraining Control," *Proc. 12th ACM Symp. Principles of Programming Languages*, (January 1985).

5. Felleisen, Matthias, Daniel P. Friedman, Bruce Duba, and John Murrow, "Beyond Continuations," Tech Report #216, Indiana University Computer Science Department (February 1987).

6. Joloboff, Vania. personal communication. December 1986.