Hooks and Open Points

Kurt Nørmark
Department of Mathematics and Computer Science
Institute for Electronic Systems
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
Denmark
Email: normark@iesd.auc.dk

Abstract

In this paper a programming language concept called an open point is introduced and discussed. Open points are meant to cover a collection of behavioral parameters of executable programs. Such parameters add openness to a program, which may be used to specialize the program in a variety of directions, all of which are intended to preserve the basic flavor of the original application. The paper gives a survey of different mechanisms from existing programming languages that, in some way or another, may be used as open points. In particular, the paper elaborates on the hook mechanism, as used in several Lisp environments. It is argued that open points should be supported by a set of tools. The purpose of the tools is partly to improve the programmer awareness of the presence of open points, and partly to catalyze an experimental binding of the open points in an exploratory programming process. A concrete collection of tools based on the hook mechanism in Lisp is presented together with an example of its use.

Keywords: Lisp, hooks, open points, program reusability, programming tools, Emacs.

1 Introduction

The starting point and the inspiration for this paper is the notion of hooks, as known from many Lisp environments. A hook offers the opportunity to execute one or more user-supplied program fragments at some designated place in a program. From a user's point of view, hooks may be regarded as openings of a program, which are effective when some event happens, or if some condition is fulfilled.

As an example, the Lisp `read` function may provide for the execution of some user-supplied actions when certain kinds of S-expressions are read. Such an opening may be achieved by having one or more examination hooks in the `read` function, which give control to user-defined functions whenever particular kinds of expressions are read. As another example, a hook may allow user-defined procedures to be activated as a response to interactive input events, such as mouse clicks and key strokes.

From an internal Lisp point of view it is not difficult to realize a simple hook mechanism. Given the duality between data and program in Lisp, a hook may simply be a variable, the value of which is supposed to be a valid Lisp construct. If the variable is bound, the value of the hook variable is evaluated:

`(if (boundp '`*hook*`) (eval `*hook*`))`

The hook mechanism may also be implemented as a procedure call:

`(if (boundp '`*hook*`) (funcall `*hook p1 ... pn*`)))`

In this realization, *hook* is the name of a symbol whose value must be a lambda expression with *n* parameters. As a slight variation, a conditional activation of a function may be considered as a hook too:

`(if (fboundp '`*hook*`) (`*hook p1 ... pn*`)))`

Such a hook is attached if the function is defined. The former approaches may easily be extended to deal with more than one attached function, but the latter approach, which stores attached functions in the function definition cell, does not have this flexibility.

The simple hook mechanisms shown above should not be used directly. Rather, adequate syntactic extensions [9] should be defined, which hide the underlying details. Furthermore, a whole range of semantical design options, including name binding issues, should be considered as part of the implementation of the mechanism.

The purpose of having hooks in a program is to open the program at carefully selected places in order to improve the reusability of the program. Hooks allow the user to customize and extend the program without having to know or to affect the source description of the program. The attachment of functions to hooks are intended to be done dynamically, at the time of execution of the program. Program customization may, for instance, be used to alleviate minor inconveniences of an application. Program extensions may result in an integration of an application with other applications.

The following items describe the characteristics of hooks, which we find most important:

1. *Announcement.*
   A hook in a program is only interesting if it is announced as part of the program's programmatic interface. The announcement makes the hooks known to the user of the program. Equally important, the announcement commits the designers of the program to keep the hooks relatively stable during the lifetime of the program.

2. *Late attachment.*
   A hook should remain open until execution time of the program. Elimination of hooks during macro expansion and compilation should be avoided.

3. *Selective opening.*
   Neither totally closed programs nor uniformly open program are ideal. A closed program is a bad starting point for reusability, whether customizations or extensions. A uniformly open program may be bad too, because such programs tend to contain little or no guidance on how to exploit the openings. It is the author of a program who decides which aspects of the program to open. It is a matter of good program design to place the openings appropriately.

In summary, the purpose of hooks is to provide for announced, dynamic, and selective openings of programs.

As mentioned above, the starting point of this paper is the basic hook mechanism, but the main focus is an analysis and a discussion of a broader class of mechanisms, which we call open points. In section 2 we discuss various hook mechanisms in Lisp, as described in the literature and as present in some Lisp systems and applications. In section 3 the meaning of an open point is defined, and in section 4 open point-like mechanisms from a variety of programming languages are surveyed. This discussion reveals to which degree the openness, as provided by hooks in Lisp programs, can be obtained in other languages and in other paradigms. In section 5 we return to the hook mechanism, where the variations and the design issues of hooks are considered. The discussion leads to the definition of some Lisp constructs for hooks, which abstract the underlying technical details. Finally, in section 6 we discuss a set of tools that supports the handling of hooks. Major parts of this section are based on a practical example. Among other issues, the tools address the problem of how to announce hooks for users of open programs.

2 Hooks in Lisp

The hook mechanism of Lisp is based on the simple metaphor illustrated in figure 1. The thin curve illustrates the flow of control of a program, which is symbolized by the big oval. Along this curve there are hooks to which it is possible to attach functions that affect the functionality of the program.

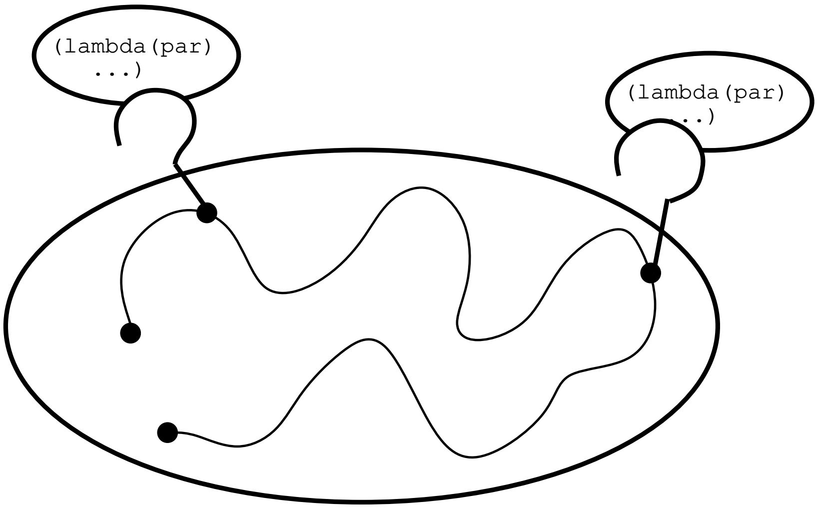

Figure 1: A program with hooks.

In Sandewall's paper on the experiences with Lisp [19], a notion of *handles* is briefly discussed (in section 5.1 of the paper). Such handles are identical to what we call hooks in this paper. Following Sandewall, a handle is

> "... a user-defined procedure that can be inserted instead of the original procedure provided by the system", [19, section 1.1].

Sandewall points out that an algorithm with handles cannot be regarded as a black box, because the handles can be substituted by arbitrary procedures. A program with handles is typically the main program—a pattern which may be augmented with user-defined contributions. Sandewall also mentions that it is convenient to use a special technique for attachment of procedures to handles, instead of using procedural arguments to higher order procedures. An incremental parameter binding technique with support of default parameters is more practical than the conventional parameter binding technique.

Hooks are also discussed by Stallman in a paper about the Emacs text editor [21]. Several Emacs systems use Lisp for extension purposes. GNU Emacs [20] is probably the most well-known of these. According to Stallman, a hook is a characteristics of an extensible system, which

> "... allows the user to provide a function to be called on certain well-defined occasions", [21, section 5.6].

The main usage of hooks in Lisp-based Emacs systems is to provide for customization and extensions of the editor's modes. Different kinds of activities in Emacs are supported by different modes. Among other things, modes determine the mapping of keys on the keyboard to functionality defined by Lisp functions. As one of the last actions during the entrance of a mode in GNU Emacs, "the hooks of the mode are run" (using Emacs terminology). Each hook may attach several functions, all of which are supposed to be without parameters. The execution of hooks makes it possible to change the behavior of the mode (such as the established bindings of keys on the keyboard) as well as to add new functionality to the mode. In addition to mode entrance, hooks are also associated with a few other interactions in Emacs, such as loading a file into a buffer and vice versa.

2.1 Common Lisp

In Common Lisp [13] expression evaluation and function application have associated hooks. The hooks are in effect when two special variables, `*evalhook*` and `*applyhook*`, are bound to user-defined functions. If, for instance, `*evalhook*` is temporarily bound to the function *f*, such as in

```
(let ((*evalhook* 'f))
   (eval expression))
```

then instead of evaluating *expression* the function *f* is called with two arguments: the construct `(eval` *expression*`)` and the lexical environment in which free names of *expression* can be meaningfully interpreted. According to *Common Lisp the Language* [13], the eval hook and apply hook mechanisms of Common Lisp are primarily designed for the implementation of Lisp development tools, such as steppers, tracers, and debuggers, within Common Lisp itself. The mechanism makes it possible to get control at critical places during evaluation of an expression, in order to report on, or otherwise utilize the available information about the state of the computation.

2.2 Interlisp

Interlisp [25, 26] supports a number of open tools. Tedit (a text editing and formatting tool [28]), Grapher (a graph drawing tool [8]), and Sketch (a more general, structured drawing program [27]) are probably the most well-known and the most important of these. Besides the interactive interfaces of these tools, a programmatic interface is supported as well. The programmatic interface consists of a set of documented functions and data structures, together with a number of hooks. Hooks are typically activated when user-initiated or user-relevant events happen such as selection with the mouse, file access, and window scrolling. In a typical pattern of use, the customizer of one of the mentioned tools attaches a number of his own functions to hooks, which in turn manipulate the state of the tool via the functions in the programmatic interface.

The author has participated in a project [15] where the openness of Tedit and the Grapher played a central role. The project focussed on tools for the support of language development. Together the tools are called the Muir Environment [24]. The most important aspect of the support came from syntax-directed editing. Tedit was used as the textual front end of the syntax-directed editor. It turned out to be possible to program the discipline of syntax-directed editing into Tedit, exclusively by using the programmatic interface of Tedit. The Interlisp source program, which implements Tedit, was never consulted or changed. The openness provided by the hooks of the programmatic interface was central to these efforts.

NoteCards [12] is a hypertext system on top of Interlisp. In a similar way to Muir, NoteCards is implemented on top of Tedit, Sketch, and the Grapher. The majority of the necessary extensions was done via the hooks and the functional interfaces of these tools. NoteCards is itself an open environment which can be adapted and tailored to the needs of its users [23]. The programmatic interface of NoteCards (more than 100 Interlisp functions) plays an important role if a user wants to add new functionality to the system. However, hooks do not play any role in the programmatic interface of NoteCards. The interface is exclusively made up of functions, which are supposed to be called from the user's extensions to NoteCards.

Despite the importance of the hook mechanism in several Interlisp applications, Interlisp does not treat the mechanism in a systematic manner. There is no standard way to attach functions to hooks. It is common to attach functions to hooks when a tool is instantiated, but the details vary from tool to tool. In all tools, and independent of each other, the hook mechanism is explained at the level of its implementation, although it would have been clearer to introduce it once and for all, as an Interlisp feature.

Advising [26] is an Interlisp mechanism which is related to the use of hooks. An advice is an addition of a piece of program to a designated function. Any function can be advised; the function does not need to be prepared for the advice, and the source expression of the function does not need to be present. It is necessary to specify how the advice is related to the existing, and perhaps already advised function (before, around, or after the original function, together with the position of the advice relative to already existing advices).

Advising is typically used to repair errors or inconveniences in functions, which are not under direct control of the user. Advising may also be the basis for a programming style, which Sandewall calls *insertive programming* [19, 18]. Following Sandewall, the essence of insertive programming is to consider a program as a skeleton and a number of amendments. An amendment may serve the purpose of attaching a procedure to a hook, but it is realized by patching the program rather than associating a function to a hook. Furthermore, Interlisp uses advising in the implementation of tracing and breakpoints.

2.3 Pan

Pan [3, 4] is a syntax-recognizing editor, which is implemented in Common Lisp. Extensibility and customizability are important objectives in the Pan design. For that purpose the implementation of Pan introduces a notion of hooks. The hook mechanism of Pan is at a higher level than that of the Interlisp tools, which we described above.

Pan supports a proper hook definition construct called `define-hook`. `Define-hook` describes and constrains the parameter signature of functions to be attached to the hook. `Define-hook` also includes hook documentation. Pan is explicit about the scope of the hook in the sense that `define-hook` specifies the scopes in which hooks may be attached. Another primitive called `perform-hook` activates functions that have been attached to the hook. Functions are attached to a hook in a specific scope, using another special form, `define-hook-function` (which either can attach a named function to the hook, or incorporate the function definition details into the construct). The list of defined hooks can be announced to users via Pan's help system.

2.4 Summary

Based on this analysis of the use of hooks in a variety of Lisp environments, we find that the following characterization of hooks covers the intended meaning of hooks in most Lisp systems:

> A hook is an *announced, program defined* activation of one or more *user-supplied* functions. The activations of the functions may substitute an existing fragment in the program. The activated functions are said to be *attached* to the hook. It is a requirement that functions can be attached at the execution time of the program. The announcement formulates opportunities of and requirements to the hook, in terms of, e.g., triggering events or conditions, expected results, and expected side effects.

There are several possible variations of the hook idea, as presented above.

1. *Hook defaults.*
   It may be possible to have some default program fragment, which is executed if no function is attached to the hook. If a default can be present, one may be able to execute the default as part of the attached function.

2. *Single or multiple attachment.*
   In the simple cases, only a single function may be attached to a hook, and this attachment prevents other functions from being attached to the same hook. It is, however, sometimes useful to be able to attach more than one function to a hook. This also ensures that the hook remains open independent of prior attachments. If multiple attachments are supported, the value of the hook may be determined in different ways from the collection of attached functions.

3. *Parametrization of hooks.*
   The hook may have no arguments, or it may pass a number of parameter expressions to the possibly attached functions.

In section 5, where we describe a concrete elaboration of a hook mechanism for Lisp, these and other variations will be discussed in more details.

3 Open Points

The hook concept, as defined in the previous section, is tightly connected to the mechanism of procedure activation. In this section we will introduce a slightly more general understanding of the concept, which we call open points.

> An *open point* is a documented, behavioral parameter of a program which is required to support default binding and execution time binding (see below for a more detailed characterization of each of these).

Being a parameter implies that an open point is a named *definition* of an entry, which must be bound (explicitly or implicitly) before execution time. The adjective "behavioral" emphasizes that parametrization of expressiveness and action is of primary concern. Open points are neither envisioned to capture simple parameters nor type parameters. An actual parameter serving as the binding of an open point will typically be a parameterizable procedural abstraction.

In addition to being a behavioral parameter of a program, an open point must obey the following requirements:

1. *Default bindings.*
   It must be possible to have default bindings of the parameters that are used as open points. It implies that a program with open points will serve as a full-fledged, executable application regardless of any explicit binding of the openings.

2. *Execution time binding.*
   It must be possible to bind the open point at execution time of the program. It implies in particular that the binding of an open point must not be prevented by any transformations carried out during the program development process.

3. *Documentation.*
   The purpose of, and the requirements to an open point must be described as an integral part of the opening. The documentation of open points contributes to the separation of internal program aspects from external aspects.

Point number one makes it possible for a program with unbound open points to serve some specific needs. If the user wants to change some of the open behavior of the program, the information called for in point number three gives the user some guidelines about how to exploit the open points. Point number two guaranties that the open points are not optimized away, e.g., as part of a program compilation. In the definition and the requirements above we have "left open" the issue of further binding of an already bound open point.

In order to bind an open point at program execution time, it must be possible to incrementally affect the program at that time. Programming the procedures that binds the open points is similar in nature, but presumably different in scale, to programming the closed parts of the program. In incremental programming environments, such as in Lisp systems, the programming of these procedures does not cause problems.

In more traditional, non-incremental environments, it is much more difficult to actually bind the open points without directly modifying the source program. There are at least two possible solutions to that problem:

1. *Increased incrementality of the environment.*
   If small changes of a program can be done without re-processing the entire program, it will be possible to use this incrementality to bind the open points of the program.

2. *Inclusion of a dynamic extension language to the environment.*
   Instead of adding incrementality to a natively non-incremental environment, an already open *extension environment*, based on for instance a Lisp dialect, can be included into the program. The extension environment must be interfaced to the definitions and the functionality of the underlying program. In a framework like this, the binding of open points is supposed to be done through the extension environment.

Incremental programming environments for languages such as Pascal and C have been subject to much research (see, e.g., [17, 16]). The idea of having a flexible extension environment is known from the GNU Emacs text editor [20]. The extensibility and the programmability of this version of Emacs are due to an integration of a Lisp environment into an efficiently implemented text editor framework. Currently, there are some efforts going on to generalize this idea to arbitrary areas of applications.[1]

Given the definition of open points from above and the definition of hooks from section 2 it should be clear that the hook mechanism is a concrete example of open points. In the following section we will analyze to which degree a number of other programming languages support the notion of open points.

4 Programming Language Support of Open Points

In this section we will review some concepts from existing programming languages that to some degree fulfill the requirements of being open points of a program.

In most object-oriented programming languages there are concepts that approximate that of open points. In the following sections we will compare open points from above with mechanisms and concepts in the object-oriented languages Simula, Beta, Smalltalk, Loops, the Common Lisp Object System, and HyperTalk.

[1] The author is aware of two general extension environments: Bob Glickstein's ELI—The Embedded Lisp Interpreter from Carnegie-Mellon University, and Oliver Laumann's ELK—The Extensible Lisp Kernel from Technical University of Berlin, which is based on Scheme. At the time of writing this paper, nothing seems to have been published about these systems.

4.1 Virtuals in Simula

In Simula [7] it is possible to define procedures as virtuals. Virtual procedures may be full procedure definitions, or they may consist only of a procedure head. A virtual procedure definition in a class C may be redefined in full detail in a subclass of C. Let us, as an example, assume that the head of a procedure P is declared as virtual in the class C and that P is defined in full details in C', which is a subclass of C. A call of the virtually declared procedure P in a C' object, say in a procedure located in the class C, may be understood as an open point, the binding of which is supplied in C'. It is possible to simulate default bindings of open points in Simula by letting the virtual procedure definition be a full procedure definition, not just a procedure head.

Simula is a compiled language in the tradition of Algol. This implies that the open points defined by the call of virtual procedures are bound definitively during the compilation process. This is against the spirit of open points, and it leads to the conclusion that Simula virtuals cannot be used to simulate open points. As discussed in section 3, it is difficult to obtain a late binding of open points in languages that emphasize compile time bindings.

4.2 Virtuals in Beta

The virtual mechanism in Beta [14] is a generalization of the virtual mechanism in Simula. There are several dimensions of generalization. First, it is possible to have virtual *patterns*, where a pattern can be used to model procedures as well as classes. Second, a virtual pattern in a pattern P may be further bound in a chain of subpatterns of P. If, e.g., P2 is a subpattern of P1 which again is a subpattern of P, a virtual in P may get contributions from additional bindings in P1 as well as from P2. In the imperative part (the statement part) of a virtual pattern it is possible to use the keyword `inner` to define how the imperatives in the virtual pattern should be combined with the imperatives in further bindings of the pattern. Third, it is possible to control whether a binding of a virtual in a pattern is a final binding in contrast to still being open.

Regarded as open points, virtuals in Beta are not much more general than those in Simula. The basic problem is still that open points simulated via virtual patterns are closed too early relatively to the execution time of the program. As for Simula, we claim that Beta does not provide mechanisms that allows simulation of open points. It is, however, worth noticing that the binding of an "open point" in Beta can be a combination of several contributions from a chain of subclasses. (This supports our notion of multiple bindings of open points, as discussed in section 1 and 3 of this paper.) In Beta the combination is programmed explicitly via the use of `inner`.

4.3 Messages in Smalltalk

Message passing in Smalltalk [11] activates a method via a dynamic method lookup process. A message intented to be handled by a method in class C may in this way be intercepted by (the most specific of the) similarly named methods in subclasses of C. Thus, in Smalltalk every act of message passing has the potential to be an open point. Instead of speaking about every method in Smalltalk as virtual (in the Simula sense), we prefer to regard Smalltalk messages as open points.

An open point is bound by defining an intercepting method along the path of classes, which is searched during the Smalltalk method lookup process. The most specific of these methods is the binding of the open point, and the most general of the methods plays the role as the default of the open point. In case a binding propagates the message to `super`, the binding may actually be a combination of a set of methods. This combination may, of course, also involve the default binding.

Due to the incrementality and the workspace-orientation of the Smalltalk environment, Smalltalk supports a late binding of the open points. Consequently, the Smalltalk environment is pretty close to supporting true open points. It might be useful to be able to distinguish normal messages from messages that are intended to serve as open points. As mentioned in section 1 this also would draw a line between internal and external aspects. Such a distinction could be made by supporting a particular announcement or documentation facility for intendedly open messages.

4.4 Messages and Active Values in Loops

Loops [5] is a multi-paradigm language on top of Interlisp. One of the supported paradigms is object-oriented programming in the style of Smalltalk. Loops supports a message passing technique which is similar to message passing in Smalltalk. Consequently, the openness of Smalltalk achieved through dynamic method lookup is also present in Loops.

One of the other paradigms supported by Loops is called access-oriented programming [22] (also known as data-oriented programming [5]). The idea of access-oriented programming is to activate one or more methods upon access of particular data objects. Objects, which in this manner trigger methods, are called *active values*. Active values are typically (but not necessarily) bound to instance variables of classes. The application of active values is dual to programming in the object-oriented paradigm, in which it is the methods that access variables.

Reading an active value causes a get-message to be sent to the underlying object. Similarly, writing into an active value triggers a put-message. The objects that receive these messages must be instances of a class called `ActiveValue`, or instances of a subclass of `ActiveValue`.

In Loops applications, active values are typically used to obtain a modular design between two or more parts of an application. Separation of the simulation model from the monitoring aspects in a simulation program is a prototypical example of the use of active values. To achieve this separation, the objects in the simulation contain active values, which trigger the executions of operations in the monitoring part of the program.

Access of an active value may be regarded as an open point, which is closely associated with a particular mechanism in the language, namely variable access. The open point is bound by specializing an access method in a subclass of `ActiveValue`. Thus the binding mechanism is the same as that used for message passing as open points. (In the version of Loops described in [5] an open point is bound by specifying the name of the get-function and the put-function directly in the active value). Default binding of the open point is accounted for, because there already exist a default get-method and put-method in the class `ActiveValue`. Similarly, execution time binding of open points is achieved because of the properties of Loops' host environment, which is Interlisp. There is no provision for any particular documentation of active values in Loops.

4.5 Generic Functions in CLOS

The Common Lisp Object System (CLOS) [6] is open in the same way as Smalltalk, but in addition and as an alternative the method combination facility (explained below) provides CLOS with yet another kind of open points.

In CLOS, a generic function can be a combination of several methods. Using standard method combination, a generic function is an aggregation of all applicable before methods, a primary method, and all applicable after methods.[2] Relative to a given method combination technique, such as standard method combination, every activation of a generic function may be considered as an open point. If one wants to fill in additional details of a given generic function, it may be done by augmenting the class hierarchy with an applicable method belonging to the generic function.

It is worth noticing that the openness of CLOS is general enough also to cover the active values in Loops, which we described in the previous section. Slots in objects are intended always to be read and written via methods (which typically are generated automatically, but on request). Such accessor methods are in the same way as all other methods subject to method combination during the call of a generic function. Thus, it is possible to achieve the effect of an active value by defining before, after, or around methods in the generic accessor functions. Consequently, both variable reading and writing are open in CLOS.

A CLOS environment is an extension of a Common Lisp environment, and as such it is incremental and workspace-oriented. Thus, CLOS provides for late binding of open points in the same way as Smalltalk does. A generic function acting as the definition behind an open point may be "multi-bound," and the resulting generic function is a declarative

[2] On purpose, we do not mention the around methods, because they rely on an imperative combination with before, primary, and after methods in standard method combination. This is analogous to the method combination in Smalltalk, which is explicitly programmed from case to case, by sending messages to `super`.

combination of the multiple bindings. This is in contrast to Smalltalk and Beta, both of which use imperative combination (using messages to the pseudo variable `super` and `inner` respectively). Furthermore, it is worth noticing that a CLOS environment distinguishes between ordinary Lisp functions and generic functions. Only generic functions have the potential to be open in the sense described above. In analogy to our discussion of open points in Smalltalk, it may be attractive only to consider a subset of the generic functions as open points.

4.6 HyperTalk

It is also relevant to discuss open points in relation to HyperTalk [2]. HyperTalk is the underlying programming language of HyperCard [1], which is a hypermedia application on the Macintosh. HyperTalk is an interpreted language. Pieces of HyperTalk program are attributes of cards and other elements, which are structured in a hypertext web. As a consequence of both the orientation towards interpretation and the flexible structuring, HyperTalk is a dynamic and incremental environment.

HyperTalk is object-oriented in the sense that messages are the basic initiators of actions. The objects manipulated by HyperCard include buttons, cards, and stacks of cards. The objects are organized according to a system-defined hierarchy of concepts. As a result of interacting with HyperCard, one or more *system messages* are sent to the collection of objects in the system. The receiver is the most specific of the objects (relative to the hierarchy) that has a handler for the event, which caused the message.

Any HyperTalk program contains a variety of open points. An open point is defined by a documented system message. It is, for instance, advertised that the `mouseEnter` message is "sent to a button as soon as the mouse pointer is moved within its rectangle"; and the `newCard` message is "sent to a card as soon as it has been created" [2]. An open point in HyperTalk is bound by defining a message handler in an object. Due to the incrementality of the HyperTalk programming environment, the system supports late binding of open points. The open points can, in the same way as in Smalltalk, be bound to defaults by defining handlers at a relatively general level in the hierarchy.

In conclusion we find that the mechanisms in HyperTalk represent a rather full-fledged realization of open points.

5 Elaboration of Hooks in a Lisp Environment

In this section we will discuss and elaborate a hook mechanism for Lisp, which fulfills the requirements of being an open point mechanism.

Hooks are treated in an ad hoc manner in many existing Lisp systems. This section gives a more systematic treatment of hooks, during which both syntactic and semantic issues are settled. In section 6, the platform established in this section will be used to build tools that support the process of learning about and attaching the hooks in Lisp programs.

The details of the hook mechanism must be adapted to the specific Lisp dialect in question. As a case study, we have implemented the whole framework in Emacs Lisp [10], which is a dynamically scoped and fairly traditional dialect of Lisp. As for this paper, we will keep the discussion at a general Lisp level rather than at the level of a particular Lisp dialect. Where appropriate, we mention the issues that depend on concrete dialects.

5.1 The Hook Construct

A hook is defined to be a special form with the following appearance:

`(hook (`*hook-id actual-parameter-list*`)`
    default-forms`)`

A hook is considered as a definition of an open point at the location where it appears in the program. If one or more procedures are attached to the hook a so-called *hook-combinator* controls the applications of attached procedures on the actual parameter list (see section 5.3). In that case the hook combinator also defines the resulting value of the hook construct. If no procedures are attached to the hook, the default forms are evaluated as if they were embedded in a `progn` form, and and the value of the `progn`-form is returned as the value of the hook construct. If no default forms are present, `nil` is returned.

It should be stressed that a hook is regarded as a *definition*, which binds the name of a hook to an open point in the program. In other words, *hook-id* of the hook construct is a defining name occurrence. Besides being a definition, a hook is also a construct which causes the attached procedures to be executed.[3] Compared with the more simple interpretation, where a hook is regarded as one or more procedure calls, this is an important difference.

Given the view that a hook is a definition, it makes sense to talk about its scope (the part of the program, in which the hook has effect). The attachment must take place in the scope of the hook. The scope of a hook may depend on the concrete Lisp dialect. One particular simple solution is to apply indefinite scope [13] for hooks (hook definitions have effect on the entire program).

5.2 Attaching and Detaching Hooks

Procedures can be attached to and detached from a hook using the following two primitives.

`(hook-attach` *hook-id procedure*`)`

[3] It is worth comparing our approach to the approach used in Pan, which we described in section 2 of this paper. In Pan, hook definition and hook execution are separated into two constructs, which are called `define-hook` and `perform-hook` respectively.

`(hook-detach` *hook-id* `&rest` *procedure*`)`.

`Hook-attach` attaches a procedure[4] to a hook. The temporal ordering of the execution of `hook-attach` forms on a hook may be important, depending on the used hook combinator (discussed in the following section). `Hook-detach` is the opposite of `hook-attach`. The form `(hook-detach hook-id)`, which is without a procedure parameter, detaches all procedures from `hook-id`.

We have found it useful to be able to activate the default forms of a hook construct in an attached procedure. This makes it easy to *extend* a pre-specialized program, rather than *changing* it. The following primitive is used to activate the default forms from an attached procedure:

`(hook-default` *hook-id*`)`.

This primitive is not defined outside procedures attached to `hook-id`. In Lisp dialects with static name bindings, free names in *default-forms* of the hook construct should be closed in the context of the hook definition.

It is assumed that the `hook-attach` and `hook-detach` forms are used in the scope of the hook definition (see section 5.1). The binding of free names in attached procedures should follow the general binding discipline of the Lisp dialect in question.

5.3 Hook Combinators

The hook combinator of a hook is defined via the form

`(hook-combinator` *hook-id combinator-key*`)`.

The hook combinator is intended to be frozen by the hook definer, because the hook definer makes assumptions about the effects and the returned value of a hook.

The combinator key controls the application process and the hook-value in the case where one or more procedures are attached to the hook. Relative to the temporal ordering of attached procedures there are combinators that return the first, the first non-nil, the last, the list, and the sum of the individual applications[5].

More precisely, assume that the procedures P_1, P_2, ..., P_n have been attached to a hook, by first attaching P_1, then P_2, and so on, until P_n. The hook-combinator receives two arguments: (1) the list (P_1 P_2 ... P_n) and (2) the argument list on which the procedures are to be applied. The combinator is in control of the order in which P_1 through P_n are applied, and of the result returned from the hook construct.

The combinator called *first* executes the procedures in attachment-order (in the order P_1 through P_n), and it returns the value of the first of these (the value of P_1 applied on the

[4] In Lisps in which procedures are not first class objects, one may prefer to use procedure names in `hook-attach` and `hook-detach`.

[5] The hook combinator mechanism is inspired by the simple built-in method combination mechanism in CLOS [6].

argument list). The *last* combinator also executes the procedure in attachment-order, but it returns the value of the last of these applications. The *fist-non-nil* combinator similarly executes the procedure in attachment order, but only until one of the applications gives a non-nil value, which becomes the returned value. Three additional combinations are defined (*first-reverse*, *last-reverse*, and *first-non-nil-reverse*) that work in ways similar to *first*, *last*, and *first-non-nil*, but on the reverse attachment order. Finally, the *sum* and the *list* combinators give the `+` and the `list` reductions respectively of the individual results.

Last is the most common hook combinator, and it is therefore the default. This reflects the philosophy that the latest attachment should have the highest degree of control—namely control of the value of the hook construct, and the ability to undo results of earlier attachments (if possible). The other combinators may, however, bee useful too. As an example, the combinator *first-non-nil-reverse*, which applies the attached procedure in reverse order until one of them returns a non-nil result, is useful in case a program is customized in several steps. In such a situation, a late customization may veto the application of earlier customizations by applying this combinator. (A veto is signaled by returning a non-`nil` result from an attached procedure). The combinations *list* and *sum* are useful if we want all attached procedures to contribute equally and accumulative to the value of the hook construct.

The hook framework itself is open in the sense that it is possible to define new combinator keys and associated combination procedures.

5.4 Hook Documentation

A hook documentation form allows a piece of informal documentation to be attached to a hook definition.[6]

`(hook-documentation` *hook-id documentation-string*`)`.

The information in the documentation string plays an important role in the tools, which support the hook mechanism. This will be discussed in section 6.

When a Lisp program is written or loaded, it is important that the Lisp programming environment gets a chance to be aware of the hooks of the program. The right way to achieve that is to modify the Lisp reader to do something special when a hook construct is

[6] It would be possible to integrate the hook documentation clause as well as the hook combinator clause into the hook construct. Following this idea, a hook construct would consist of a name, some actual parameters, a default form, a hook combinator, and a hook documentation string. However, all these pieces of information make a typical hook construct far too big. This is a pragmatic problem that should not be underestimated. We have chosen to split the construct into several dependent constructs. Another solution would be to separate the hook definition from the execution of the hook, as it is done in Pan, see section 2. A third solution would be to use a programming environment with program presentation facilities that allow for abstract presentation[15]—in this case selective presentation—of lengthy constructs.

encountered in a Lisp expression.[7] Alternatively, the environment may get its knowledge of hooks through the hook documentation clauses. Following the latter approach, it is necessary to require that all hook documentation clauses are placed at the outer expression level of a Lisp program.

5.5 Overloading and Structuring of Hooks

If more than one open point is bound to the same hook name (in the same scope), the hook is said to be overloaded. Overloaded hooks share attached procedures, combinator keys, and hook documentation.

It is important to be able to structure and classify open points, as well as all other kinds of definitions, in a Lisp program. The need for structure on the open points becomes especially urgent when making tools that support the handling of open points. In traditional Lisps, where the structuring mechanisms are weak, it is therefore tempting to invent conventions that impose some structure on the open points. In the concrete framework described above, the hook identifier may be of the form `group:name`, where `group` defines a unit of structuring to which the hook belong. This may, of course, be generalized to two or more levels of grouping if necessary. In Lisps that are stronger on overall structuring mechanisms, e.g., Common Lisp or CLOS, it is natural to use the existing structuring mechanisms (packages and classes) for open points as well.

6 Tool Support of Hooks

In this section we will describe a set of interactive tools that supports the exploitation of hooks. The tools are presented through a simple example. We are going to assume that the tools are built on top of the hook mechanism described in section 5.

6.1 An Example of an Application

A simple personal database will be used to illustrate some of the practical aspects of customizations via hooks. The database contains information about addresses, phone numbers, and electronic mail addresses of persons, institutions, and companies. The facility is implemented in Emacs Lisp, and it is used as an integrated part of Emacs. When writing letters, addresses can be extracted from the database and be inserted into the letters; similarly when sending electronic mail, the e-mail addresses can be extracted from the database; and, finally, the database can be used as an electronic phone directory.

A query on the personal database consists internally of two parts: (1) extraction of information from the database, and (2) utilization of the extracted information in some

[7] This may for instance be done via an inspection hook in the `read` function.

context. Both parts are opened via the use of hooks. When information has been extracted from the database there is an intercepting hook, which is able to modify the result before it is utilized. As the default, the information is typically displayed in a new Emacs buffer, but this may also be changed by procedures attached to the hooks.

6.2 Approaches to Customization

In order to customize an application, such as the database, it is necessary to obtain knowledge about the hooks, and probably also knowledge of other parts of the programmatic interface of the application. Such knowledge can be obtained in at least three different ways:

1. By reading the source program of the application.
2. By consulting the documentation of the programmatic interface.
3. By using special purpose tools on the program.

The solution in item number one can only be used when the source code is available. Even in that situations, though, it may be confusing to be faced with all the internal details of the program. Even worse, by reading the details of the source program, the programmer may by accident make the customization dependent on details, which may be changed without notice in future versions of the application.

The second and the third solution are similar, especially when the documentation mentioned in item two is "on line." However, there are differences too. First, the information gathering process may be more tightly integrated with the application process when using the tool approach. In the concrete example, information about the available open points of the personal database may be gathered *while using the facility.* Second, the interface of the tools, which present the hook information, may be active as opposed to the overall passive nature of online documentation. The hook supporting tools, which we shall describe in section 6.4, are browers via which hook attaching, detaching, and various kinds of querying about hooks are provided for.

6.3 Examples of Customizations

Given the personal database, as described above, we will assume that a user identifies the need for a couple of customizations:

1. When inserting the address of a person in some textual context, the user gets an error message if the database contains more than one record on the same name ("the name is ambiguous"). We want to change the behavior such that each of the possible addresses is presented for the user, and such that it is the user who decides which of them to actually insert.

```
STATIC HOOK REPORT

       database:add-database-entry                new-entry            Last
       database:insert-email-address-hook         nil                  Last
       database:insert-address-hook               nil                  Last
       database:get-email-hook                    ...list...           Last
       database:get-address-hook                  nil                  Last
       database:get-phone-hook                    nil                  Last
       database:database-access                   nil                  Last
```

Figure 2: An example of a static hook report.

```
DYNAMIC HOOK REPORT

1.     database:database-access                   nil                   Last
2.     database:get-address-hook                  nil                   Last
3.     database:database-access                   nil                   Last
4.     database:get-phone-hook                    nil                   Last
```

Figure 3: An example of a dynamic hook report.

2. The telephone company changes the area codes in a systematic manner. When asking for a phone number we would like to transform old area codes to the new ones.

6.4 Tool Support of the Customization Process

The customization process is supported by so-called *hook reports*. A hook report is a browser that displays information about hooks at some selected level of detail.

There are two ways the user of the personal database facility can approach the information about available hooks in the system: Via static or dynamic hook reports.

The static hook report relevant for our sample application is shown in figure 2. Each *entry* (each line in the presentation of the report) shows information about a single hook. The left column of the report shows all the hooks in the `database` group. The middle column shows the procedures attached to the hooks. As can be seen, the hooks `add-database-entry` and `get-email-hook` have already been attached. (The latter is attached to more than one function). The right column shows the hook combinators in effect.

Via static hook reports (as well as the dynamic hook reports), it is possible to browse into the detailed documentation of the individual hooks. The documentation presented in this way is defined by the `hook-documentation` clauses (see section 5.4). Using the static hook report in our customization example, it is possible to get knowledge about all the hooks in the `database` group, but it is up to the user to find out which of the hooks are relevant for the customizations we have in mind.

```
HOOK CLASSIFICATION REPORT

GROUP                       FILTER-CONDITION
general                     -
news                        -
database                    +
address-list                -
record-parsing              -
emacs                       +
```

Figure 4: An example of a hook classification report.

The alternative way to get information about hooks is through a dynamic hook report. A dynamic hook report displays information about hooks when they are encountered during the execution of a program. If we assume that dynamic hook reporting is active for the `database` group, then the requests of inserting the address of someone results in entry number 1 and number 2 in the report shown in figure 3. Asking for a phone number results in entry numbers 3 and 4. Thus, a dynamic hook report makes it easy to relate specific interactions to all potential open points encountered during these interactions.

Static hook reports are initiated from a more general report, which we call a hook classification report. An example of such a report is shown in figure 4. The report shows all the known groups of hooks (cf. section 5.5) together with a so-called filter-condition. If a programmer is interested in obtaining detailed, dynamic information about hooks in a specific group, he or she should turn on the filter condition for that group. In figure 4 the hooks belonging to the groups `database` and `emacs`[8] will be reported dynamically. A hook classification report is created on explicit request from the user.

Besides the browser functionality of hook reports, the reports also offer a flexible interface for experimental attaching of hooks. Instead of using the programmatic hook interface, as defined in sections 5.2 and 5.3, a terse command interface, based on direct manipulation of the entries in the reports is provided on top of the hook reports.

Let us continue our concrete example from the state of the dynamic hook report in figure 3. After reading the hook documentation, the user finds out that the hook `get-address-hook` is relevant for the first of the customizations. By just typing the single letter "`a`" (for attach) on the entry for this hook, the user will be prompted for a name of the function to be attached to the hook. If the function doesn't exist, the tool will offer to define a template of it, with the correct parameter profile.[9] Now it is straightforward to try out the effect of the attachment, simply by interacting with the

[8] The hooks in the group `emacs` are the "native hooks" of Emacs, such as the hooks associated with major modes (see section 2). Such hooks are treated as a special case in the tools which support the notion of hooks, as defined in this paper. The advantage of this is that we are able to give the same support to Emacs hooks with respect to reporting and experimental attaching as we do to our own hooks.

[9] The tools get the information about the parameter profile from the hook documentation. In order for this to work, the style of the documentation is assumed to follow a standard pattern. This is made easy by providing editing operations, which defines a template of the `hook-documentation` construct, given a hook construct.

database again. It is also easy via the command interface to get rid of an attachment.

6.5 Discussion of Tools

The tools described above support the program customizer. They help the customizer of a program to get information about existing open points in the program, and they support a smooth and flexible program customization process.

There is no tool support for the programmers who define the hooks in a program. The problems they are where to place the hooks in order to obtain a sufficiently open program. As already mentioned, this is a program design issue, and only skilled programmers are able to solve the problem. However, the following rules of thumbs may be helpful:

- *Preserve the flavor of the program.*
  It does not make much sense to open all aspects of a program, or to eliminate entire operations of it via hook attachments. If, for instance, the original program is an editor, the specialized program obtained as the result of binding open points should still be some kind of an editor.

- *Be careful to pass sufficient information in hook parameters.*
  In order to customize or extend a program via a hook, it is critical that the necessary information about the current state of the program is available in the attached procedures. It may be possible to obtain further information via the functional interface of the program, but the most direct way of getting access to the information is via the passed hook parameters.

- *Try to make the set of hooks complete.*
  Given the restrictions in the intended openness from the first rule, the set of hooks should be as complete as possible. Putting a hook into one particular operation in an interactive program, such as the personal database application, should be extended to putting hooks into all similar operations at the same level of abstraction.

The special purpose tools for open points are in some sense analogous to debuggers. A debugger lets the user examine the dynamic state of a program with the purpose of locating the cause of an error. The set of tools proposed in this paper are directed toward examination of another important aspect of a program, namely the open points. The purpose of the tools that support the hook mechanism is, as mentioned earlier, to help the programmer in the process of customizing a program. As such, the hook supporting tools are quite different from debugging tools.

7 Conclusions

The goal of this paper has been

1. to analyse and expose the hook mechanism, as present in many Lisp systems,
2. to generalize the hook mechanism to that of open points,
3. to give a survey of open point-like mechanisms in various programming languages,
4. to elaborate a concrete hook mechanism in the tradition of Lisp, and
5. to design and to describe a set of tools that supports announcement and exploratory attachment and hooks.

The support of an open point-like mechanism in a programming language is supposed to have a positive impact on the generality and the potential reusability of the programs written in the language. As we have seen, it is possible to simulate open points in many dynamic programming languages. In more statically-oriented programming languages it is, however, far more difficult to obtain the effect of open points, mainly because of the requirement of late bindings of open points.

A coherent and systematically defined hook mechanism is important for every Lisp dialect. However, when compared with open point mechanisms in object-oriented languages (such as Smalltalk, Loops, and CLOS), the object-oriented approaches should be preferred. The reason is that the open points provided by object-oriented languages are better integrated and follow the main structuring mechanism of these languages.

Finally, we will emphasize the importance of the tool support of hooks. Hook reports improve the practical value of hooks in a program. It is easy to get up-to-date and consistent information about the presence of hooks, together with their documentation. Furthermore, it is extremely flexible to explore potential attachments of the hook via the interactive command interfaces that are defined on the hook reports.

Acknowledgements. I would like to thank Bent Bruun Kristensen, Liam Peyton, Randy Trigg, Michael L. Van De Vanter, and Kasper Østerbye for help and discussions during the process of writing this paper.

References

[1] Apple Computer, Inc. *HyperCard User's Guide*, 1987.

[2] Apple Programmer's and Developers Association. *HyperCard Technical Reference Package*, August 1987.

[3] Robert A. Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, Computer Science Division, University of California Berkeley, December 1989.

[4] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system for integrated development environments. Submitted to the ACM SIGSOFT '90 Fourth Symposium on Software Development Environments, 1990.

[5] Daniel Bobrow and Mark Stefik. *The Loops Manual.* Intelligent Systems Laboratory, Xerox Corporation, December 1983.

[6] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonay E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification X3J13 document 88-002R. *Sigplan Notices*, 23(Special Issue), September 1988.

[7] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Common base language. Technical report, Norwegian Computing Center, October 1970.

[8] Envos Corporation. *Lisp Library Modules*, September 1988.

[9] Jr. Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986. Tecnical Report no. 199.

[10] The Free Software Fundation Inc. *GNU Emacs Lisp Manual*, September 1988. (Incomplete draft. Available as on-line documentation in Emacs).

[11] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

[12] Frank G. Halasz. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[13] Guy L. Steele Jr. *Common Lisp, the language.* Digital Press, 1984.

[14] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, 1987.

[15] Kurt Nørmark. *Transformations and Abstract Presentations in a Language Development Environment.* PhD thesis, The Computer Science Department, Aarhus University, Denmark, February 1987. DAIMI PB-222.

[16] D. Notkin. The gandalf project. *The Journal of Systems and Software*, 5(2):91–105, May 1985.

[17] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984. In *Software Engineering Notes*, vol. 9, no. 3. and *Sigplan Notices*, vol. 19, no. 5, May 1984.

[18] Erik Sandewall. *Some Observations on Conceptual Programming*, volume 8 of *Machine Intelligence*, chapter 14, pages 223–265. John Wiley and Sons, 1977.

[19] Erik Sandewall. Programming in an interactive environment: the "Lisp" experience. *Computing Surveys*, 10(1):35–71, March 1978. *Also in* Barstow, Shrobe, and Sandewall (editors) "Interactive Programming Environments" McGraw-Hill Book Company.

[20] Richard Stallman. *GNU Emacs manual*. The Free Software Fundation Inc, 1985.

[21] R.M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[22] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, pages 11–18, January 1986.

[23] Randall H. Trigg, Thomas P. Moran, and Frank G. Halasz. Adaptability and tailorability in NoteCards. In H.-J. Bullinger and B. Shackel, editors, *Human-Computer Interaction—Interact'87*, pages 723–728, 1987.

[24] Terry Winograd. Muir: A tool for language design. Technical Report CSLI–87–81, Center For the Study of Language and Information, Stanford University, March 1987.

[25] Xerox Artificial Intelligence Systems. *Interlisp-D Reference Manual, vol. 1: Language*, October 1985.

[26] Xerox Artificial Intelligence Systems. *Interlisp-D Reference Manual, vol. 2: Environment*, October 1985.

[27] Xerox Corporation. *Xerox Lisp Documentation Tools. A User's Guide to Sketch*, June 1987.

[28] Xerox Corporation. *Xerox Lisp Documentation Tools. A User's Guide to Tedit*, June 1987.