Control Tools and their Implementation

H. JUSTIN COVEN
Bellarmine College

An alternative direction for programming language research is proposed. Three new axioms of programming language design are proposed and combined into a new approach. The three axioms focus on: the development of reflective programming language tools; development of control abstractions; and capturing common human computational models. It is argued that a break from the mathematical model of functions is needed in order to develop more reflective language tools, and control abstractions that capture common human computational models. This approach focuses on tools that work with control structures from an operational point of view.

A number of new language tools are developed. The first tool introduced is control queues which is a general mechanism for breadth-first traversal. The next tool grab explicitly accesses the control structure. Grab can be used to create understandable recursive lambdas. Zap explicitly manipulates control structures. Zap can be used for modular exception handling. The final tool control sorted collections is a general mechanism for best-first traversal. The implementation of control queues is discussed in the final section.

A pictorial operational semantics is used to define and describe these new language features.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory - *Semantics*; D.3.2 [**Programming Languages**]: Language Classifications - *Lisp, very high-level languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features - *control structures, procedures, functions, recursion*; D.3.4 [**Programming Languages**]: Processors - *interpreters*

General Terms: Languages

Additional Key Words and Phrases: operational semantics, exception handling, breadth-first traversal, best-first traversal, reflection and metalevel architectures, language design, recursive lambdas

0. INTRODUCTION

This research started with the premise that one important way to develop new language features is to develop language tools that manipulate the different components of a language. A language is composed not only of functions; and data but also of control structures; interpreters; and scope-inheritance mechanisms. Recently there has been a lot of research in the development of reflective tools, however is has been from a functional biased view. This functional bias to confounds many of these components with each other.

Author's address: Department of Computer Science, Bellarmine College, Louisville, KY 40205.

If a break is made from the standard function definition and an operational view is taken instead then a number of useful new tools can be developed. In this article we develop a cohesive set of tools that operate upon control structures.

There is a discussion of the theoretical motivations and importance of this research in the first section. The second section describes our first control tool - control queues. A pictorial operational semantics is used to define the internal workings of control queues. We then go into a comparison of control queues and a defined function as a solution for general breadth-first traversal. In our comparison we meet two additional control tools - grab, which is a tool to explicitly access control structures; and converttoq which allows the conversion between control stacks and control queues. Section 3 describes the control tool grab in more detail. Section 4 discusses zap which allows explicit manipulation of control structures. We demonstrate that zap provides modular exception handling in comparison to the nonmodular exception handling of catch and throw of Lisp and chainto of Pop-11. Section 5 describes control sorted collections which is a general best-first traversal mechanism. Section 6 discusses the development of different kinds of control structure mechanisms, as well as a comparison with alternative control mechanisms such as coroutines. The final section describes the implementation of control queues.

1. THEORY

When introducing new language paradigms often there are underlying (and unstated) philosophical assumptions that are being challenged or proposed. Thus, we start by discussing philosophical issues underlying our work. We first discuss abstract motivations such as Reflection, Control Abstractions, and the development of alternative computational models. These motivations are crystalized into three axioms for the development of new programming language features. These combined axioms motivate the creation of a combined set of tools that operate on control structures from an operational perspective. The topic of this article is that set of control tools. We continue with discussions of semantics and software qualities and principles. Our theoretical discussion of control tools ends with a naive programmer's motivation.

1.1 Reflection

Reflection is the ability of a programming system to reason about itself. From a programming language perspective, a language needs to have features that allow a programming system to access and manipulate itself. This must include not only the data and functions that are the most commonly accessible components of a programming system, but also control structures, inheritance mechanisms and even the definition-implementation of the programming language.

Our first philosophical axiom is:

AXIOM 1. *Reflection is essential for the development of intelligent systems and can be used to new programming language features that can aid in the development of intelligent systems.*

For a system to be intelligent it must be able to reason about its own reasoning process, including the ability to create new reasoning processes. This ability to reason about reasoning is seen in many different types of human reasoning and has been simulated in many different artificial intelligence systems. The focus of this paper is on control tools, reflective control tools would have the ability to: access, manipulate, and reason about control structures. We shall delegate the development of reasoning systems to other fields of endeavor in particular Artificial Intelligence and here only focus on the tools for accessing and manipulating the control structures. These tools are a prerequisite for reasoning, for you can not reason about information that you can not access, and if you can not manipulate structures after your reasoning, then there is no practical outcome of your reasoning. In Sections 3 and 5 we describe the control tools grab and zap which access and manipulate standard control stacks respectively. In Subsection 2.5 we describe the control tool converttoq which manipulates one kind of control structure (a control stack) into a different kind of control structure (a control queue).

There has been previous work that has allowed the access and manipulation of control structures in procedural and functional languages [3, 6, 8, 17-19, 21]. However control structures were confounded with: scope-inheritance (environment), function syntax (continuations), and the philosophical axiom of the necessity of an infinite tower of reflection. Continuations are a representation of the current position of execution in the form of the functions that still have to be executed in order to finish computation. Control information is forced into a very awkwardly accessed and manipulated form: function syntax. The parameters are separated from the rest of the representations of control into another structure called the environment which is a mechanism used to implement scoping-inheritance. Parameters are tied to this scope-inheritance mechanism and confounded in this way. There is also an underlying assumption throughout this work that an infinite tower of reflection is needed to capture intelligence. In other words there needs to be the ability to create reasoning systems on top of reasoning systems infinitely with each higher system having the ability to reason about the systems below it. This has forced the creation of very complex systems where all parts can fold into each other. Thus control has had to take on a function syntax so that it would have the same form as data, functions, scope-inheritance, and interpreter-implementation.

In our work we do not assume that an infinite number of reasoning systems will be necessary to capture intelligence. It may be that only a few levels of reflection will suffice to capture intelligence. Thus we focus on developing an effective set of tools of reflection at just one level of reflection instead. Control need not be in a functional form, it need only indicate where execution is currently at in a function. We also do not investigate scope-inheritance issues, keeping parameters tied to frames that also include the code of the current frame along with the current location indicator of the current frame or function call.

Our approach to the development of Lisp like languages is different than that of much other research. We feel that the central issue of lisp is not "functions" but the ability to manipulate structures - structures such as data, symbols, lists, functions, control, scope-inheritance, and interpreter.

1.2 Control Abstractions

The development of control abstractions has been at the heart of many programming language advancements. It is very common that language developers will notice commonly occurring sets of instructions and create language features that capture these sets of instructions into new control constructs that execute the common sets of instructions implicitly. Some examples include language features such as "if" and "while" statements as well as procedure calling constructs.

AXIOM 2. *New language constructs can be created by combining common sets of instruction sequences.*

The motivation behind our second axiom is to help the programmer program faster by allowing him to execute numerous lines of code implicitly but explicitly using a few explicit lines of code that contain the new language construct.

1.3 Computational Models or "Can Language Research be Liberated from the Functional Model of Programming"

By removing the return value in the operational semantics of function calling new computational models can be created. Throughout this article we will be analyzing the operational substeps used in the implementation of function calling and seeing how to create alternative operational substeps that will capture alternative computational models.

In the title of this subsection our parody of Backus's famous article title is quite serious. Much current research in programming languages seems to be frozen into the functional model of programming. It seems that much of this work is not concerned with how their developments can be used by programmers but rather with how well their developments capture the functional model of programming. Throughout this section we focus on what types of language advancements can help programmers. We see the functional-procedural computation model as one very useful model that captures the common human conceptual construction of breaking tasks into subtasks. However current implementations have assumed a rather rigid operational semantics. An operational semantics that captures a depth-first traversal of subtask calling.

In Sections 2 and 5 we show that we can just as easily use alternative operational semantics that capture breadth-first or best-first traversal. We see depth-first, breadth-first, and best-first traversal all as common computational models that humans use to reason. Each of these being a refinement of the subtask calling computational model. This brings us to our third axiom.

AXIOM 3. *New language constructs can be created by capturing common computational models used by humans in their everyday reasoning.*

This is both a confirmation and a challenge to recent focuses of functional and logic programming. While their has been much work into trying to capture common reasoning structures into these models, there has been a limitation on forcing them too fit certain mathematical models (operational semantics) or properties including the example of the functional models assumed operational semantics described above.

1.4 Semantics

Semantics is meaning! Meaning is in the mind not in any mathematical set of symbols and manipulations! This needs to be clearly stated for too much recent work seems to have omitted any concern for the the motivations behind semantics and instead focused on the delights of mathematical machinations. Semantic systems such as operational, denotational of axiomatic are all tools. The sparse work that has mentioned the purpose behind these semantic tools have focused the following uses: 1) unambiguous definitions; 2) leaving parts undefined for implementer's decision; 3) verifiability; 4) documentation; 5) compatibility; 6) development of alternative representations; 7) as a programming aid; and 8) to help classify systems [4, 7, 9, 13, 20]. However not all semantics are suited for all of these uses or for all types of problems of language construction [1]. Care must be taken when using a semantic system so that as many uses can be made of that semantics, and the costs of the use of the semantics used does not out weigh their use. If we were to use denotational semantics to characterize the tools we present in this article we would run into some of the confounding issues discussed in Section 1.1. One most also realize that there is a nebulousness in all of the above uses (e.g. you may be able to verify that a program meets formally defined requirements, but you will never be able to verify that it meets the intended requirements of the program user, for his requirements are in his/her mind).

We choose a pictorial operational semantics to represent our language tools. This semantic form is useful for the first two and last three above uses of semantics. We use this form of semantics to 1) unambiguously define our language features yet 2) leave open numerous implementation decisions to the specific implementer. This form of semantics has 6) aided us in the development of these alternative language constructs. It is also the common representation of control structures taught in freshman level classes to 7) aid programmers in understanding function calling. We also use it later in the paper (Section 6.1) to 8) classify different control tools.

1.5 Software Qualities and Principles

It is difficult to analyze if new language features are of use or not. Some of the metrics that we do have available are desirable software qualities and principles [10]. The new tools we will describe have the following important software properties and qualities - zap allows more *modular* exception handling, control queues and control sorted collections are *general* mechanism that implicitly implement breadth-first traversal and best-first traversal algorithms respectively. Grab and zap allow explicit access and manipulation of control knowledge (*visibility*), knowledge that is usually only implicitly accessible and manipulatable. We have used grab to create *understandable* recursive lambdas.

1.6 A Naive Programmer's Motivation

Naive programmers view the procedure calling model from an operational perspective. An operational model in which procedure call instructions are abstract instructions that execute numerous assumed operational steps. These operational steps describe a standard form of control flow(depth-first). When using this view a question arises. Can we create other abstract instructions that would execute a different kind of control flow (breadth-first or best-first)?

The standard function control flow model is very useful when we attack problems that use a stack (such as depth-first traversal), however it is not so useful when working with problems that are easily represented using a queue (such as breadth-first traversal). Can we create an abstract instruction that would execute a control flow useful for handling problems that are easily represented using queues? I have heard this same question asked a couple of times before. However it was always understood from the functional - mathematical perspective with the standard assumed operational semantics standing in the way, and thus difficult to identify as a valid question. Here is a vnews example:

>>Is there any way of doing a breadth-first search recursively ...?

>... I'm not quite sure why you want to do this recursively. It seems to me that the most
>straight-forward approach is to make a queue, ... I guess my point is that this doesn't
>seem like a fundamentally recursive process, the way a depth-first search would be. ...
>perhaps I am misunderstanding your question? [12]

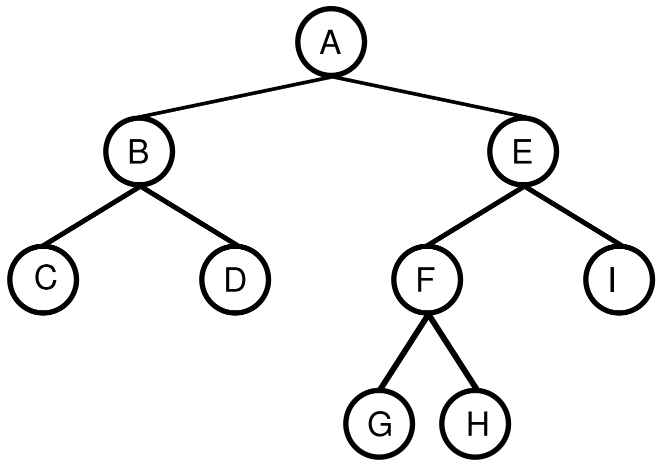
The confusion is that the questioner is viewing the function calling model as a control abstraction, but is unable to express himself from that viewpoint. The responder views the functional model as mathematical assuming the standard operational semantics attached to function calling, and the question can not match this model.

The real answer is yes we can do breadth-first search recursively. An example of depth-first traversal follows in Figure 1. We are using a new variant of Lisp called qlisp or standard common Lisp in all of the the examples throughout this article. Figure 2 describes the data structure used to represent the tree that is to be traversed. Note that in figure 1 the second "I" is the return value whereas all other letters are the values printed out as the tree is being traversed.

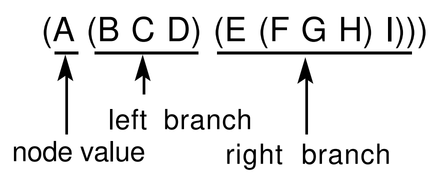
```
qlisp> (defun pre-ord (tree)
  (if (atom tree) (prin1 tree)
    (progn
      (prin1 (first tree))
      (pre-ord (second tree))
      (pre-ord (third tree)))))
PRE-ORD

qlisp> (pre-ord (quote (a (b c d) (e (f g h) i))))
ABCDEFGHII
```

Fig. 1. Recursive depth-first traversal.



A. A tree to be searched breadth-first.



B. A list representation of the same tree.

Fig. 2. A list representation of a tree.

We introduce a new construct "defq" which is identical to "defun" with the exception of the operational semantics attached to its control flow. Here instead of creating a control stack, defq will create a control queue. Using defq we can do recursive breadth-first traversal, as is seen in Figure 3. Again note that the last value printed is the return value NIL.

```
qlisp> (defq q-ord (tree)
  (if (atom tree) (prin1 tree)
    (progn
      (prin1 (first tree))
      (q-ord (second tree))
      (q-ord (third tree)))))
Q-ORD
qlisp> (q-ord (quote (a (b c d) (e (f g h) i))))
ABECDFIGHNIL
```

Fig. 3. Recursive breadth-first traversal.

In the next four sections we describe a number of control tools that we developed by focusing on Axioms 1,2, and 3. The first tool is control queues which fits the naive programmers motivation described above.

2 CONTROL QUEUES

In this section we introduce the language construct control queues. In the first subsection we give a brief outline of its operational semantics. In the second subsection we give detailed operational semantics for both control queues and control stacks. In Section 2.3 we show an operational trace of one example. The fourth subsection includes a comparison of control queues and a defined function as tools for general breadth-first traversal. The final Subsection 2.5 discusses the control tool converttoq which modifies the control structure as the program is running from a control stack into a control queue.

2.1 Basic Idea

We develop a control queue mechanism that will allow us to do breadth-first traversal without worrying about explicit housekeeping of control flow using an explicit queue. The queue will be maintained automatically by the same mechanism that does the calls to code. We have to reorganize how control structures are maintained when code is called and when code is returned to. This mechanism is not just an algorithm but rather it is a new control flow abstraction. This control queue mechanism can be added to any function or procedure calling language.

In order to put control queues into perspective we first look at the operational semantics for control stacks in Figure 4. There is a stack of frames one each for a nested defined function call. The most recent frame is current and not connected to the stack. Each frame contains enough information to run the function it controls. This includes some description or link to the functions code, a position indicator, and parameter values (refereed to alternatively as locals, local variables and environment throughout this article). The last function called from is the first function returned to. Thus the frames are organized into a FILO stack.

Standard control structure model given to introductory Computer Science students. Each frame contains code, position, and locals. The specifics of how these are implemented are left up to individual implementers.

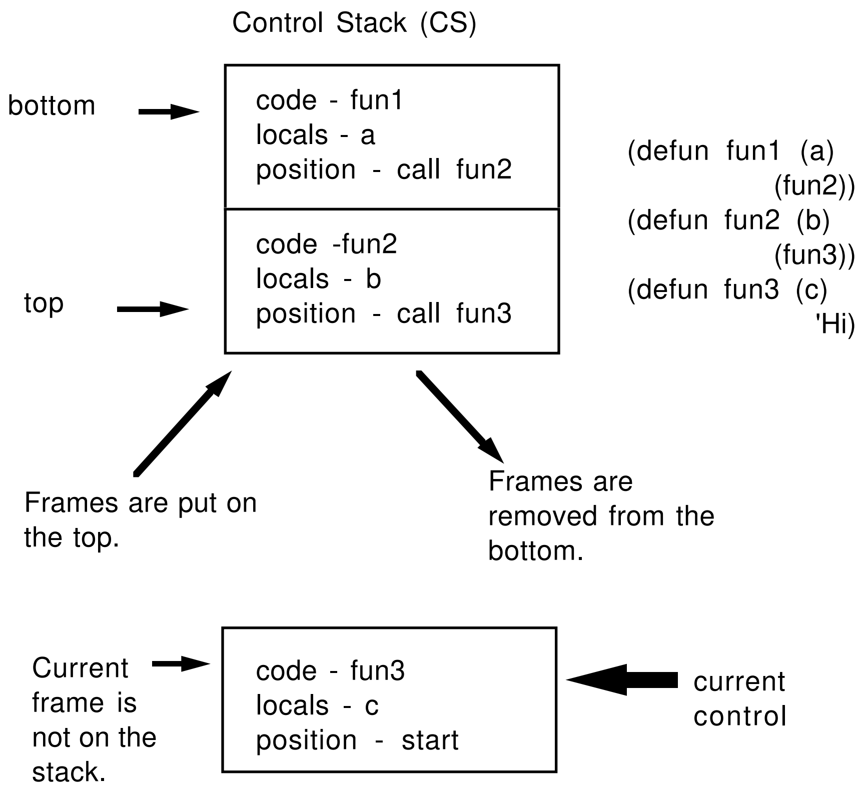


Fig. 4. Basic control stack operational semantics.

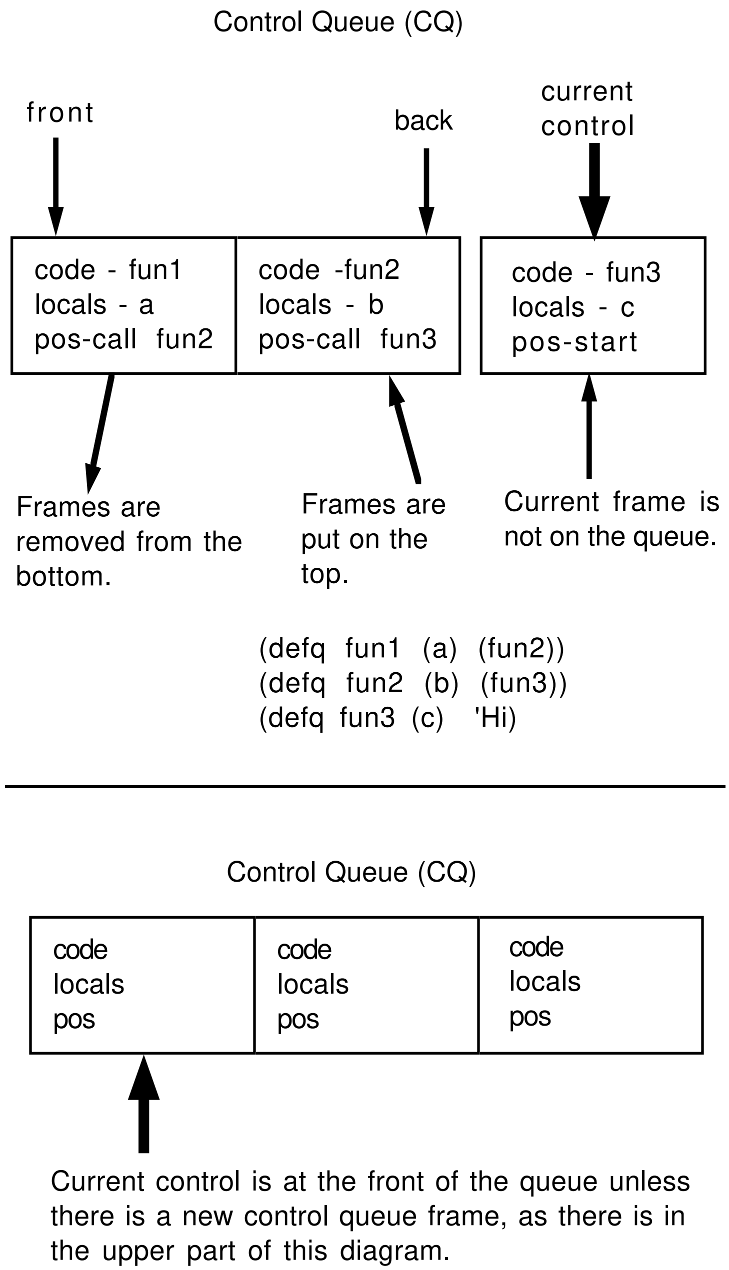


Fig. 5. Basic control queue operational semantics.

Operational semantics for control queues are identical to that for control stacks except that the first frame called from is the first frame returned to. This is described in Figure 5 which has a FIFO queue organization. Unlike control stacks current control is not always in an unattached frame at the top of the stack. In order to capture breadth-first traversal with this mechanism we have two distinct situations. One situation is analogous to control stacks in that current control can be in an unattached frame at the rear of the queue. Alternatively current control can be at the front of the queue, in which case there is no unattached frame.

Normal control functions and defq's can be freely intermixed. If a control queue is called from a control stack frame then a new frame is created that contains a control queue inside of it. In Figure 6 we show a control queue in side of the top control stack frame. Because the control queue is so big we have exploded a view of it with an arrow from the actual frame to its exploded view. We will call each internal frame to the control queue a *queue frame*. Normal functions can be called from within a defq function. In this way a queue frame can build up its own control stack. Figure 7 is a simplified view of Figure 6, in most cases we will leave out some or all of the details included in Figure 7. This is for two reasons. One, only important features need to be identified, details will only confound the reader. Two, many of the details are implementation dependent and thus should not be specified in general semantics.

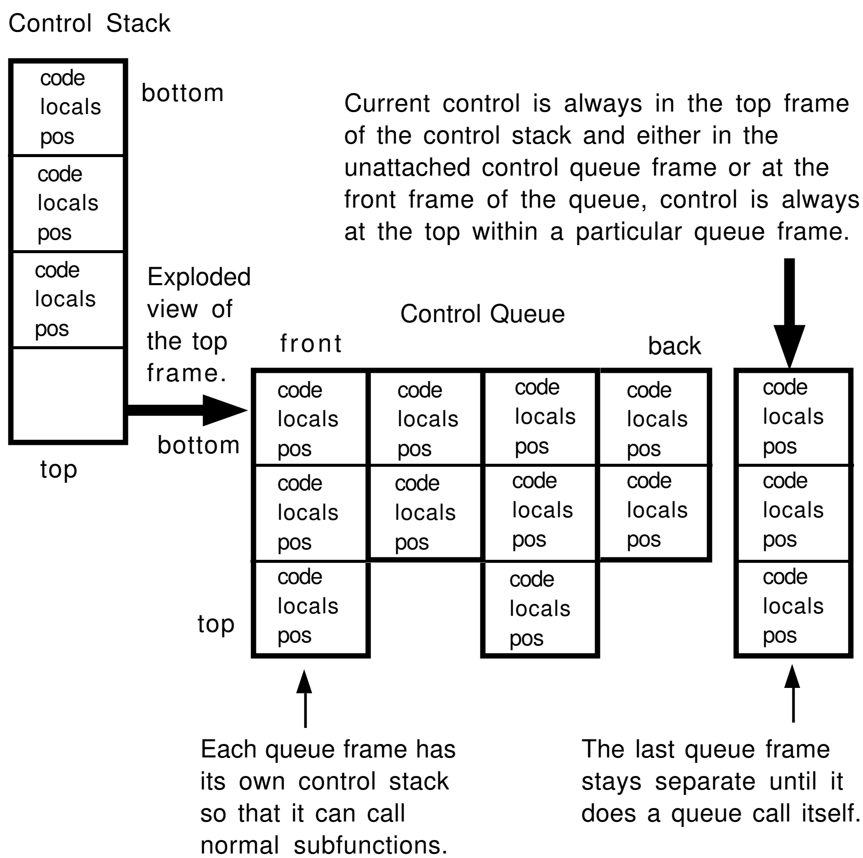


Fig. 6. Control queue operational semantics.

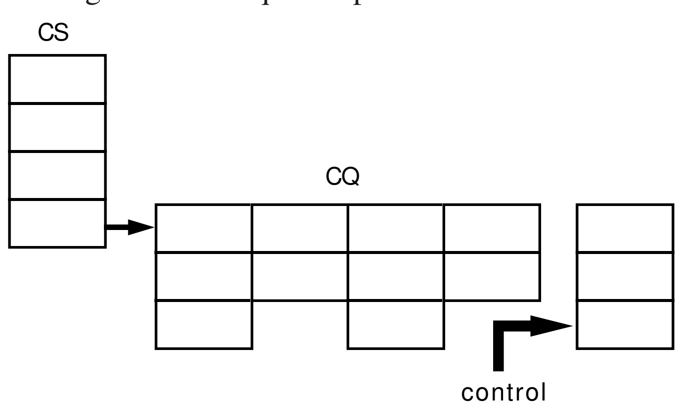


Fig. 7. Brief control queue operational semantics.

Because there are different elements that are essential to understanding different language constructs, we will be simplifying operational diagrams throughout the paper. In some instances we will also need to go into more detail describing the code, current position, and variables in some detail. We will be using some conventions in order to make the diagrams more understandable. Some of the simplifications will include leaving out position, code and/or variables when they are not necessary. Code, variables and position will be separated by dashed lines, and solid lines will separate frames. Only the immediate values around a position will be included in describing a position. This may include stacked arguments for the next instruction to use, the stack will always have its top just above the instruction to execute and will go upwards from there. Only important variables will be indicated, and sometimes with only enough detail to get the idea across. Code will typically only be indicated by the name of the function. If we wish to indicate generic code, variables or position without any detail we will use - code, vars or locals, and pos or position.

2.2 Operational Semantics of an Example

There are two assembly level instructions associated with every defq function call. This is essential for capturing breadth-first traversal. Before looking at the detailed operational semantics for control let us remember that in implementing a control stack there are numerous steps that are executed. When a normal function is called there are a number of steps that are taken: the current function is stored away on a control stack; parameters are bound with arguments; and control is transferred to the code of the function called. When a normal function returns; the top element of the control stack is reactivated with its code position being the place where control returns; a value is also returned from the the returning function to be used by the top element of the control stack. In order to evaluate the complexity of this new mechanism we need to compare its complexity with the complexity of similar operational semantics for control stacks, which we include at the end of this subsection.

With control queues we will not concern ourselves with the binding or returning of values. Control queues will always return NIL. There are two different types of queue function calls depending upon which of the two assembly level instructions the program is situated at. We name the first assembly level instruction call1 and the second call2. We need to describe what happens at each of the call1, call2 and return instructions. All of the rules governing control queues appear in Figure 8.

Control Queue Algorithm:

"Call1"
1) Increment calling frame pointer.
2) **If** qcall is from within a queue frame **and** an unattached queue frame **then** attach that frame to the queue **else** do nothing.
3) Transfer control to the front of the control queue or top of the control stack if no control queue exists yet.

"Call2"
1) Increment calling frame pointer.
2) If not in a control queue then attach the unattached control stack frame.
3) Create a new unattached queue frame for the called function and transfer control to it.

"Return"
1) Remove queue frame from control queue, transfer control to front of control queue, **if** no more control queue frames **then** remove control control queue and transfer control to top of control stack.

Fig. 8. Control queue implementation algorithm.

What follows in Figures 9 though 15 are a description of the above steps in pictorial form.

All the diagrams for control queues do not include notation for locals and code, they only contain sufficient notation to indicate position. Bold open arrows indicate the transformation from one operational structure to another, while bold lettering indicates current position.

Call1

1) Increment calling frame pointer.

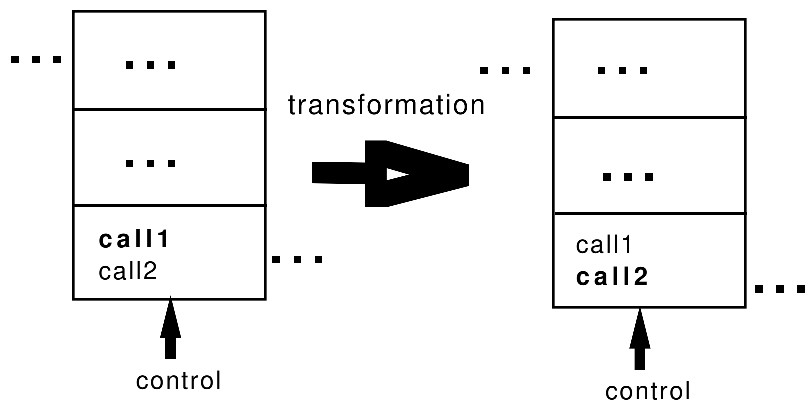


Fig. 9. Control queue operational semantics call1 step 1).

2) **if** qcall is from within a queue frame **and** an unattached queue frame **then** attach that frame to the queue **else** do nothing.

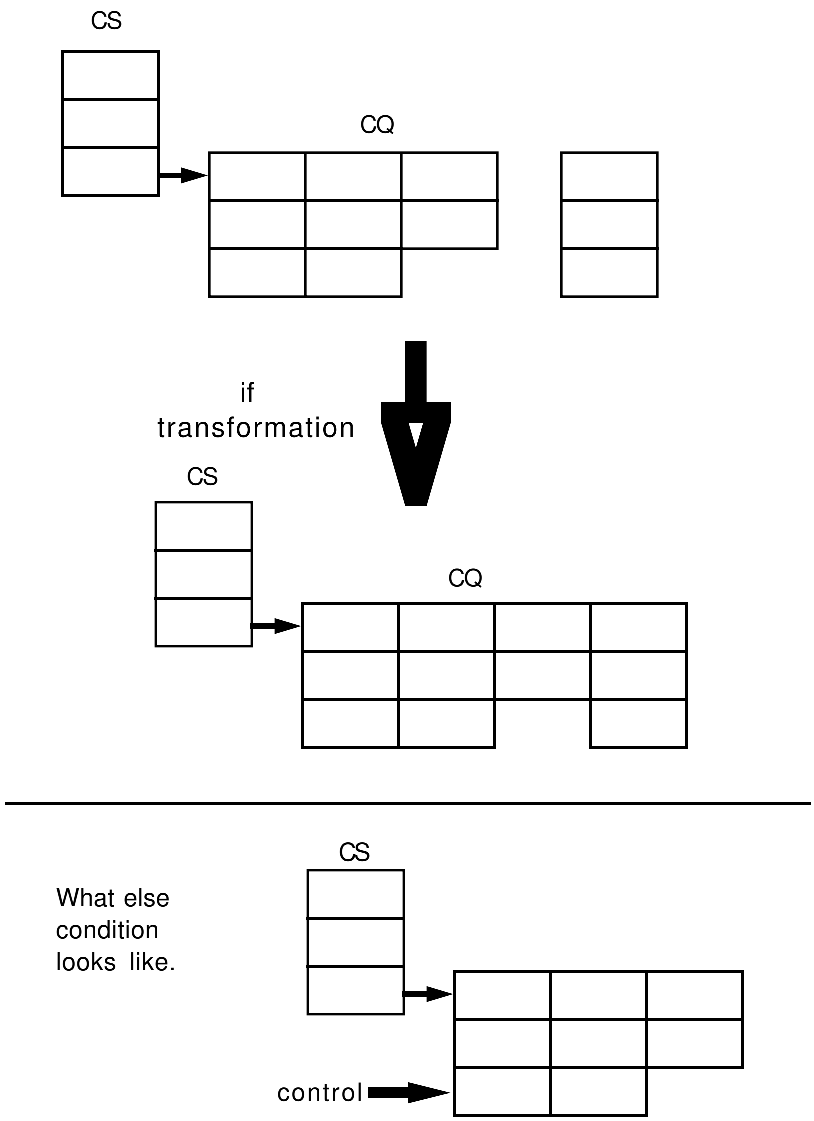
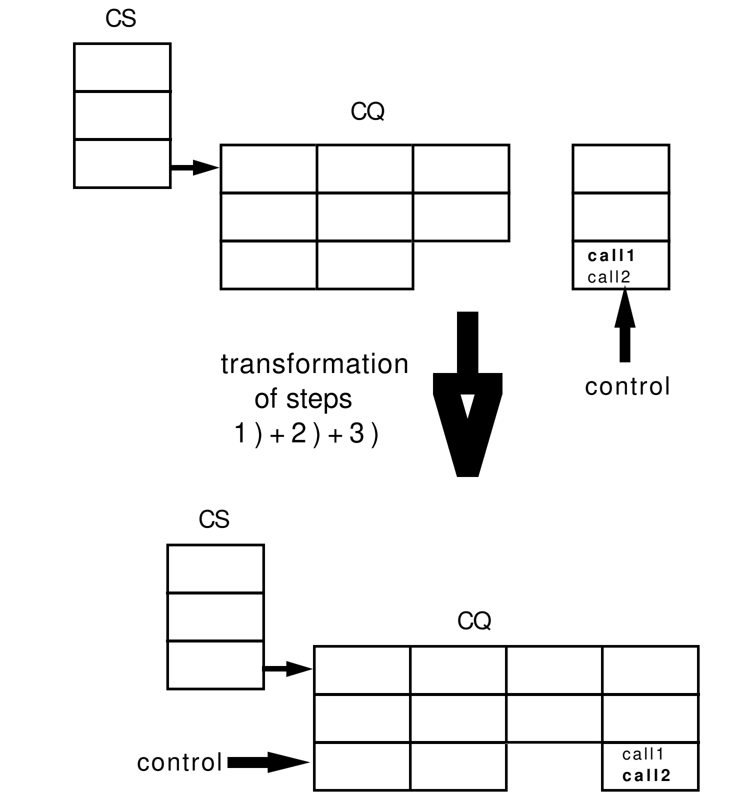


Fig. 10. Control queue operational semantics call1 step 2).

3) Transfer control to the front of the control queue or top of control stack if no control queue exists yet.



If no control queue exists yet.

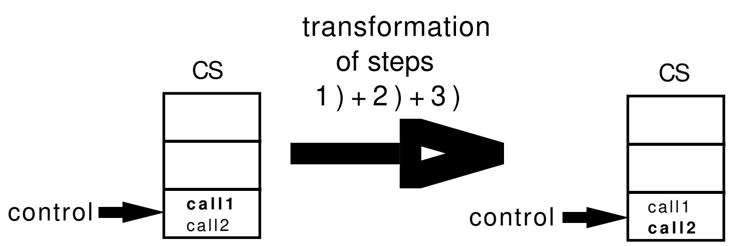


Fig. 11. Control queue operational semantics call1 step 3).

Call2

1) Increment calling frame pointer.

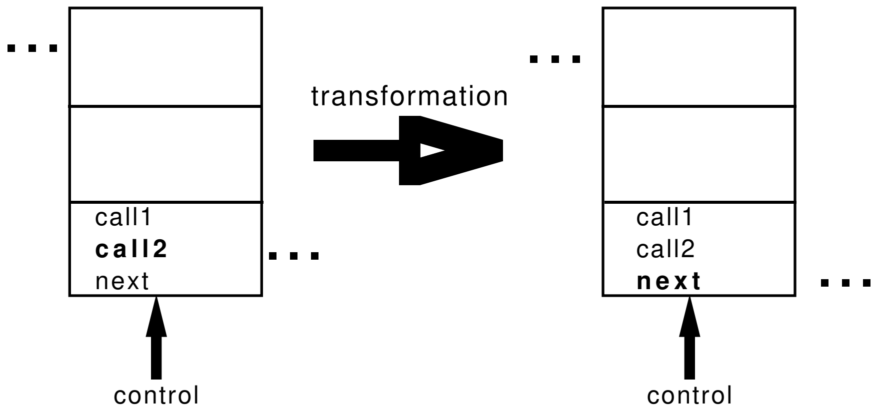
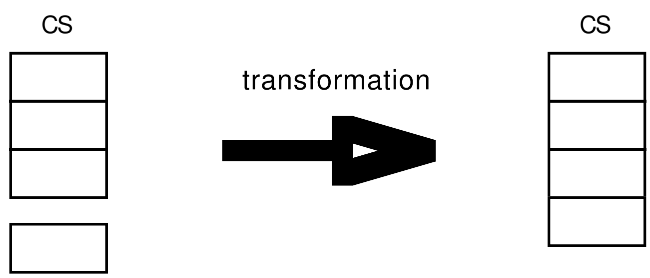


Fig. 12. Control queue operational semantics call2 step 1).

2) If not in a control queue then attach the unattached control stack frame.



Example of being in a a control queue.

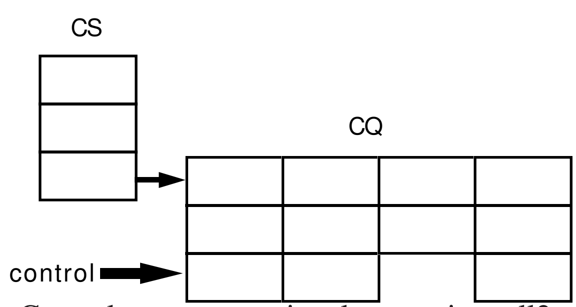


Fig. 13. Control queue operational semantics call2 step 2).

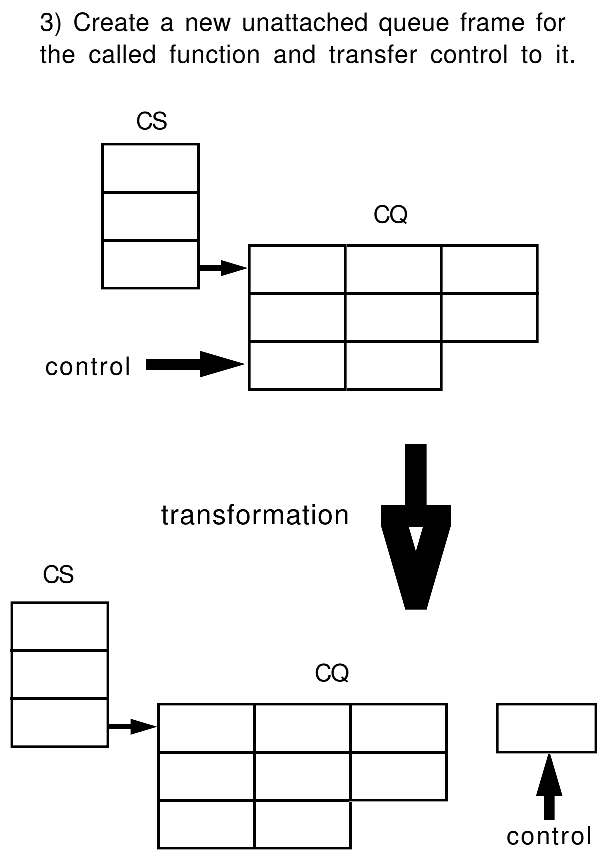


Fig. 14. Control queue operational semantics call2 step 3).

Return

1) Remove queue frame from control queue, transfer control to front of control queue, **if** no more control queue frames **then** remove control queue and transfer control to top of control stack.

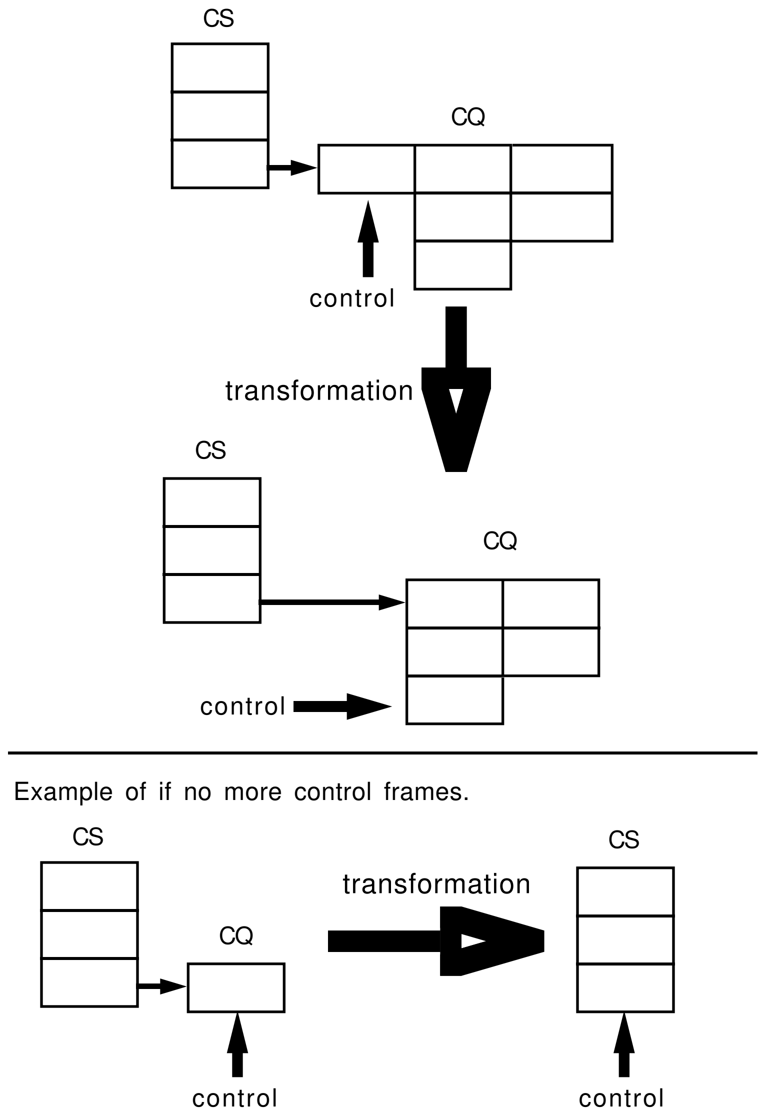


Fig. 15. Control queue operational semantics return step 1).

For comparison purposes we now give a pictorial operational semantics for normal function calling. The call for normal function calling is more complicated than either call1 or call2, because it is the whole instruction not broken in two and because a return value must been handled. The return instruction is likewise more complicated because of the return value. However we need only worry about two instruction - call and return not three assembly level instructions as with control queues. Figures 16 through 21 describe control stack operational semantics.

Operational Semantics for ordinary function calling.

CALL

1) Advance location.

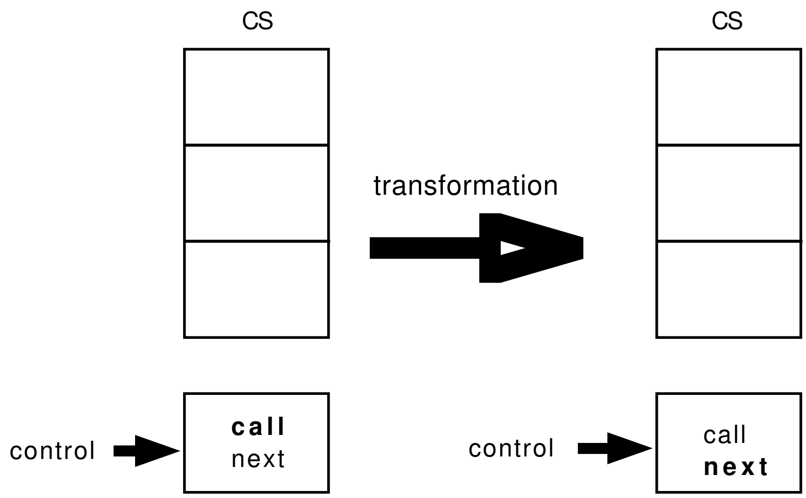


Fig. 16 Control stack operational semantics - call step 1).

2) Append frame to control stack.

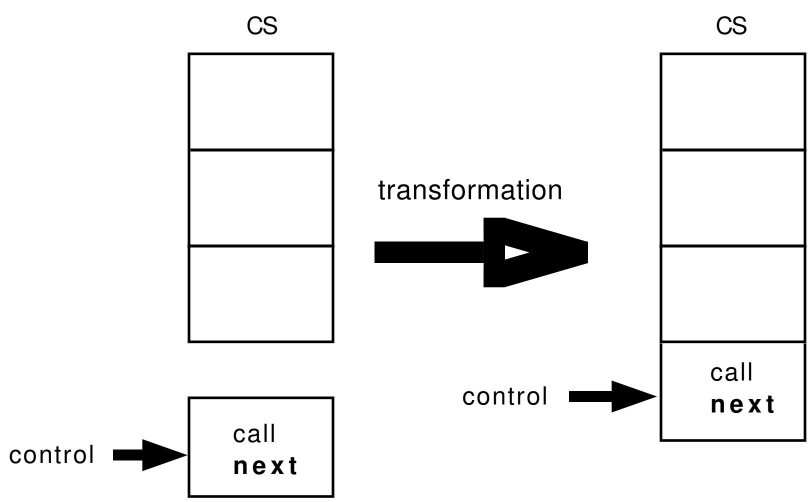


Fig. 17. Control stack operational semantics - call step 2).