Building a Backtracking Facility in Smalltalk Without Kernel Support

Wilf R. LaLonde and Mark Van Gulik

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract Languages like Snobol, Prolog, and Icon were designed with backtracking facilities from the outset and these facilities are deeply intertwined with the implementation. Retrofitting a backtracking facility in a language that wasn't designed for it has never been achieved. We report on an experiment to retrofit Smalltalk with a backtracking facility. The facility is provided through a small number of primitives written in the language (no modifications to the kernel were made). The ability to do this is a direct result of the power provided by the objectification of contexts.

1 Introduction

Backtracking is difficult to retrofit into a language that was not designed to support it.

Backtracking is widely acknowledged to be a powerful computational facility. It has existed for some time in languages like Snobol [4], Icon [3], and Prolog [1]. In each of theses cases, the facility is integral to the language and as a consequence deeply intertwined with the implementation. There has never been a successful retrofit of backtracking in a language that was not designed for it. There are two reasons for this: (1) the facility interacts with the existing language making it difficult if not impossible to define clear semantics for the combined language or (2) the extension affects the entire implementation causing a totally different implementation to have to be developed.

This research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada).

We report on an experiment to add backtracking to Smalltalk [2]. The addition is achieved by the introduction of 3 methods to the existing library. More important, this extension is written in the language and does not require kernel support. That it could be done is a testimonial to the power of the object-oriented paradigm. More specifically, the extension was possible because contexts could be manipulated as objects and used for purposes not originally conceived by the originators of the system.

Although we used Smalltalk as the implementation vehicle, the ideas are much more general. In particular, the same could be done in any language that provides access to the underlying contexts (stack frames); e.g., Lisp. It can also be done in a language with reflective capabilities [7] (see [6] for a recent exposition on reflection) or one in which the processor source code is available.

2 The Core Backtracking Facility

Backtracking provides alternative solutions when the initial solutions are unacceptable.

Backtracking is a facility that permits computations with multiple solutions and multiple solution techniques. Coupled with unification in Prolog, it leads to a powerful symbolic processing capability. In Smalltalk, the facility can be integrated into a stream-like facility that can be used to compute any number of solutions. It can also be used for problems with an infinite number of solutions.

To add the core backtracking facility to Smalltalk, we introduce two basic operations: "self **succeed:** aBlock" where aBlock computes a value, say anObject, that is meant to be returned as a solution and "self **fail**" which indicates that another solution is required. The succeed operation is similar to "↑anObject", which returns anObject from the current method but expects that a subsequent failure will cause the computation to come back to return further solutions. Writing "↑anObject" explicitly is interpreted as a special case of the succeed dictating that no further solutions are available; hence computation will not "come back". The fail operation forces the computation to resume after the most recent succeed statement; i.e., intuitively, it is as if the return

had not been made and computation were simply continuing after the succeed.

Both primitives cause major changes to the semantics of methods that use them since backtracking is not a standard facility in Smalltalk. Consequently, using methods that backtrack is not the same as using methods that don't. The converse, however, is more important. Arbitrary Smalltalk code can be integrated into backtracking methods without impact as long as it is side-effect free. As we will see, backtracking cannot undo global side-effects although it does undo all local side-effects.

We begin with a simple example. Suppose the following method were added as a class method, say in Object.

```
examples

return123
    self succeed: [1].
    self succeed: [2].
    ↑3
    "Transcript print: self return123; show: ' '.
    self fail"
```

When the contents of the comment at the end of the method is executed, 1, then 2, and finally 3 is printed on the transcript followed by a notifier indicating that no more backtracking is possible (we will come back to the notifier later). What is happening is that "self **succeed:** [1]" is causing 1 to be returned to the sender (as the print parameter) which in turn is sent to the transcript for printing. The **show:** is then executed followed by the **fail**. At that point, computation is "backed up" to the last succeed point where a second answer is computed; i.e., "self **succeed:** [2]" and the process leading to the second **fail** repeated. This second **fail** causes "↑3" to be executed (indicating no more answers are forthcoming); when the third **fail** is reached, a notifier indicates that there are no more answers.

The backtracking facility is more flexible than we might expect. To illustrate this, we could execute "self **return123**" in a workspace or browser. The scenario might proceed as shown in Listing 1. To eliminate the error notifier at the end, a third block primitive called **capture** is introduced. It serves to limit the range of the backtracking and always returns **nil**. For example, the first example above can be redone in the following way using **capture** to ensure that 1, 2, and 3 are printed on the transcript followed by a normal return.

```
[Transcript
    print: self return123; show: ' '.
    self fail] capture
```

A captured block with a fail facility is effectively a loop. From the point of view of semantics, however, it is quite different. In particular, backtracking causes the state of the computation to be undone so that it is in the same state it was prior to returning a solution (assuming additional solutions are also provided). The backtracking facility undoes all the effects of local computation; i.e., local variables are restored to their former state. However, global effects are not undone.

Consequently, the standard technique for summing elements in a loop (for example) cannot be used naively with the backtracking facility. For example, the following code does not work. It returns 3 and not the expected 6 which is 1+2+3. Can you deduce why?

```
| sum |
sum ← 0.
[sum ← sum + self return123. self fail] capture.
↑sum
```

The reason is evident from the backtracking semantics. When method **return123** is about to return 1, sum has value 0. It is true that the 1 returned is added to 0 and sum updated to 1. However, when the **fail** is executed, backtracking causes the computation to be restored to the state it was in when method **return123**

```
"Type self return123 in a workspace, highlight it, and select printIt from the menu."
    self return123
"The result of execution is the following:"
    self return123 1
"Type self fail to get an alternative solution, highlight it, and select printIt."
    self return123 1 self fail
"The result of execution is the following:"
    self return123 1 self fail 2
"Execute another self fail again to get the final solution."
    self return123 1 self fail 2 self fail 3
"If we execute self fail one more time, an error notifier appears."
```

Listing 1: An example that uses the backtracking primitives

was about to return 1. Hence sum is restored to its original state; i.e., to 0. When the 2 is returned, this is added to sum (now 0 again) so that sum is updated to 2. The same thing happens again for the next **fail**. However, even though sum is restored to 0, "↑3" is executed instead of a succeed. Hence no further backtracking will be possible. After 3 is added to 0 again to update sum to 3, the fail determines that no more backtracking is possible and the capture causes a normal return. Sum ends up being 3.

If we understand the backtracking semantics, it is possible to produce the desired effect. The idea is to make global modifications. This does not necessarily mean modifying global variables although that would work too. For instance, we could do the following instead.

```
| sumHolder |
sumHolder ← Array with: 1.
sumHolder at: 1 put: 0.
[sumHolder
    at: 1
    put: (sumHolder at: 1) + self return123.
    self fail] capture.
↑sumHolder at: 1
```

The notion that backtracking restores the state of the computation to its former state is crucial. It is what makes Prolog such a powerful language. The user, for example, doesn't have to worry about restoring the bindings to the logic variables (it's automatic). What we see above is an *interface interaction* between two systems: the standard Smalltalk system and the backtracking system. In the next section, we introduce the notion of a backtracking stream that provides a more intuitive interface between the two systems. It provides less knowledgeable users with better control over the computation. We don't expect users to program in the manner indicated above.

For the moment, we continue with further examples that illustrate the backtracking primitives **succeed:**, **fail**, and **capture**. Consider the following:

```
returnAValue
    self succeed: [self return123]. "first value"
    self succeed: [self return123]. "second value"
    ↑self return123 "third value"
    "[Transcript
        print: self returnAValue; show: ' '.
        self fail] capture"
```

In this case, "1 2 3 1 2 3 1 2 3" is printed on the transcript. Why? Method **returnAValue** is designed to return three values. However, each value has itself three possibilities: 1, 2, and then 3. The initial **returnAValue** message causes the first **succeed:** to be executed which means that the **return123** result is returned. The **return123** result is obtained by executing "self **succeed:** [1]" which causes 1 to be be obtained and subsequently returned to be printed. When the **fail** is encountered, the system backtracks to the last **succeed:** (the "self **succeed:** [1]) which results in the execution of "self **succeed:** [2]". Hence the computation is repeated with 2 (just as it did with 1). The next **fail** repeats this again with 3 instead of 2. With a subsequent **fail** , however, no more values from **return123** are available so backtracking backs up even further to the point immediately after the first **succeed:** in method **returnAValue**. Hence the process starts all over again with 1, 2, and 3 successively returned. Finally, backtracking will occur to compute the third value and another 1, 2, and 3 will be successively returned.

The next example illustrate a similar interaction. In this case, it should be clear that "(1 1)" is printed on the transcript the first time. What value is printed next?

```
returnArray
    ↑Array
        with: self return123
        with: self return123
    "[Transcript
        print: self returnArray; show: ' '.
        self fail] capture"
```

Since the last **succeed:** executed was "self **succeed:** [1]" in the second call to **return123**, we should expect the **fail** to cause "(1 2)" to be printed followed next by "(1 3)" for the next **fail**. Another **fail** should cause backtracking to the first call to **return123** which causes it to return 2 and then invoke the second "self **return123**" anew. Hence the next value output is "(2 1)", then "(2 2)", and finally "(2 3)". Further backtracking results in "(3 1)", then "(3 2)", and finally "(3 3)".

A Minor Extension

Because it is relatively inconvenient to write code like

```
self succeed: [1].
self succeed: [2].
self succeed: [3].
    ...
```

we extended method **succeed:** to accept a non-block as a parameter. This value is returned unchanged. Now we can write the following instead of the above.

```
self succeed: 1.
self succeed: 2.
self succeed: 3.
    ...
```

The modified version has the same semantics as the original as long as the succeed parameter is computed

without using backtracking methods. In the latter case, an unnatural backtracking order results; more specifically, backtracking to the **succeed:** occurs before backtracking to the succeed parameter. For example, if method **returnAValue** were re-written as follows, the output would be 1 1 1 2 3 2 3 2 3.

```
returnAValue
    self succeed: self return123. "first value"
    self succeed: self return123. "second value"
    ↑self return123 "third value"
    "[Transcript
        print: self returnAValue; show: ' '.
        self fail] capture"
```

For simplicity, we explain only the first two results. The first call causes "self **return123**" to be evaluated and its result (namely 1) to be returned from the first succeed, call it S, in **returnAValue**. When a subsequent **fail** occurs, the last **succeed:** is S; hence execution proceeds with the second **succeed:** in **returnAValue**. In this case, the result of the second "self **return123**" (namely the second 1) is computed and returned. In essence, the difference between the two methods is the execution order; e.g.,

self $\mathbf{succeed}_1$: [self $\mathbf{succeed}_2$]
    $\Rightarrow$ order $\mathbf{succeed}_1$ $\mathbf{succeed}_2$.
self $\mathbf{succeed}_1$: self $\mathbf{succeed}_2$
    $\Rightarrow$ order $\mathbf{succeed}_2$ $\mathbf{succeed}_1$.

3 Backtracking Streams

> *Backtracking streams can be designed to provide a more flexible and more natural facility.*

The backtracking facility provided above is difficult to control and use because it affects the normal execution of well-understood objects. In particular, providing a solution in any method that uses a backtracking method causes the current execution state to be retained for later resumption when a failure occurs. This failure can occur arbitrarily far in the future. If a suitable solution is obtained and alternative solutions are no longer desired, it is impossible to deactivate the backtracking facility. After all, backtracking could occur in any method at any point in time. Even if we could deactivate the facility, it is impossible to selectively deactivate specific parts.

To provide better control of the backtracking facility, we introduce the notion of a **backtracking stream**, a read stream that provides objects on demand via the standard stream operations **atEnd**, **peek** , **next**, **contents**, and **do:**. The latter two operations only apply when the stream is finite.

Except for the fact that stream elements are only obtained when needed, the above operations have the usual stream semantics. At the moment, we don't permit the stream to be reset although it is a simple task to extend it. To use a backtracking stream, it is sufficient to understand how to create one. We create a backtracking stream by executing

```
BacktrackingStream on: aBlock
```

where aBlock computes and returns the successive stream elements via "self **succeed:** anObject" or "↑anObject", the latter indicating that no more answers are forthcoming. The stream itself takes care of executing "self **fail**" eliminating the need for it in the block. The block result itself is discarded. Thus a block that contains neither a **succeed:** nor an ↑ will be guaranteed to be an empty stream. For example, the following stream will return the successive elements 1 through 10.

```
BacktrackingStream on:
    [1 to: 10 do: [:count | self succeed: count]]
```

In addition, to prevent backtracking streams from affecting the contexts in which the block parameters are defined, the stream executes the blocks in a copy of their defining context. Thus a method fragment such as the following is prevented from interacting.

```
...
count ← 5.
stream1 ← BacktrackingStream on:
    [1 to: 10 do: [:count | self succeed: count]].
count ← count + 5. "count is 10"
stream2 ← BacktrackingStream on:
    [100 to: 104 do: [:count | self succeed: count]].
count ← count + 5. "count is 15"
result ← stream1 contents, stream2 contents.
count ← count + 5. "count is 20"
...
```

Each stream has its own copy of the method's local variables. Thus executing "stream1 **next**" twice and "stream2 **next**" once at the point where count is indicated to be 15 (for example, using the debugger) would have the effect of changing the count variable for stream1 to 2 and the count variable for stream2 to 100; they don't affect each other since they are distinct variables. Neither would the count in the method fragment be modified. If the method fragment were executed as is, result would end up with "(1 2 3 4 5 6 7 8 9 10 100 101 102 103 104)".

Recall the example of the previous section where the successive values computed by method **return123** were to be summed. With backtracking streams, the sum could be computed as follows:

```
| aStream |
aStream ← BacktrackingStream on:
    [self succeed: [self return123]].
sum ← 0.
aStream do: [:element | sum ← sum + element].
↑sum
```

The following solution would also work just as well. Why? The answer is simple. After the first **succeed:** above, there is neither a second **succeed:** nor an ↑ to indicate another solution. Hence it is equivalent to a standard ↑. Of course, the solution returned itself backtracks so that the return occurs 3 times.

```
| aStream |
aStream ← BacktrackingStream on:
    [↑self return123].
sum ← 0.
aStream do: [:element | sum ← sum + element].
↑sum
```

Backtracking streams are not restricted to a finite number of solutions. For example, an infinite stream of primes is generated in the example of Listing 2. A more interesting example that uses backtracking (and failure) to a greater degree is shown in Listing 3. The example comes from LaLonde [5]. The task is to assign unique digit values to variables b,i,g,o,y,m,a, and n so that the equation **big+boy=man** is solved. An obvious restriction is for b and m to be non-zero. Two support methods are used: method **digits** which successively returns the digits 0 through 9 (as characters) and method **"digitsDifferentFrom:** aString" which successively returns only those digits not in aString.

A Prime Number Generator

Number class methods

```
allPrimes
    "Construct an infinite stream that generates primes."

    | sieve candidate limit subcollection |
    ↑BacktrackingStream on: [
        sieve ← OrderedCollection with: 2.
        self succeed: 2. "The first prime is a special case."
        candidate ← 1.
        [true] whileTrue: [
            candidate ← candidate + 2. "3, then 5, 7, 9, etc."
            "Is it prime?"
            limit ← candidate sqrt truncated.
            subcollection ← sieve select: [:element | element <= limit].
            subcollection
                detect: [:previousPrime | (candidate \\ previousPrime) = 0]
                ifNone: [
                    self succeed: candidate. "found one" sieve add: candidate]]].
```

BacktrackingStream class methods

```
primeExample
    | stream first50Primes |
    first50Primes ← OrderedCollection new. stream ← Number allPrimes.
    50 timesRepeat: [first50Primes add: stream next].
    ↑first50Primes
    "BacktrackingStream primeExample"
```

Listing 2: An example that uses backtracking streams

A Puzzle Solver

BacktrackingStream class methods

```
digits
    '012345678' do: [:aCharacter | self succeed: aCharacter]. ↑$9

digitsDifferentFrom: aString
    | aDigit |
    aDigit ← self digits. (aString includes: aDigit) ifTrue: [self fail]. ↑aDigit
```

The actual method for computing solutions is shown below. Of interest is the fact that there are 64 solutions.

```
puzzleExample
    "Solve the BIG + BOY = MAN puzzle."

    | stream b m bString first i bmString o bmiString a bmioString g bmioaString y bmioagString n
    bmioagyString big boy man bigInteger boyInteger manInteger answer |
    stream ← BacktrackingStream on: [
        b ← self digitsDifferentFrom: ''.
            b = $0 ifTrue: [self fail].
        m ← self digitsDifferentFrom: (bString ← (String with: b)).
            m = $0 ifTrue: [self fail].
            (m digitValue between: (first ← 2*b digitValue) and: first+1)
                ifFalse: [self fail].

        i ← self digitsDifferentFrom: (bmString ← bString, (String with: m)).
        o ← self digitsDifferentFrom: (bmiString ← bmString, (String with: i)).
        a ← self digitsDifferentFrom: (bmioString ← bmiString, (String with: o)).
            (a digitValue between: (first ← i digitValue + o digitValue) and: first+1)
                ifFalse: [self fail].

        g ← self digitsDifferentFrom: (bmioaString ← bmioString, (String with: a)).
        y ← self digitsDifferentFrom: (bmioagString ← bmioaString, (String with: g)).
        n ← self digitsDifferentFrom: (bmioagyString ← bmioagString, (String with: y)).
            (n digitValue between: (first ← g digitValue + y digitValue) and: first+1)
                ifFalse: [self fail].

        big ← String with: b with: i with: g.
        boy ← String with: b with: o with: y.
        man ← String with: m with: a with: n.
        bigInteger ← big inject: 0 into: [:sum :character | 10*sum + character digitValue].
        boyInteger ← boy inject: 0 into: [:sum :character | 10*sum + character digitValue].
        manInteger ← man inject: 0 into: [:sum :character | 10*sum + character digitValue].

        answer ← big, '+', boy, '=', man. Transcript cr; show: answer. "For debugging only."
        bigInteger + boyInteger = manInteger ifFalse: [self fail]. Transcript show: ' ***'. "For debugging only."
        self succeed: answer].

    ↑stream contents
"BacktrackingStream example5"
"The 64 solutions begin with '213+265=478' '215+263=478' '213+276=489' ..."
```

Listing 3: Another example that uses backtracking streams

4 Implementing The Backtracking Primitives

Backtracking can be implemented easily in Smalltalk because contexts are objects. Well!!! if you can figure it out.

The backtracking facility is based on a suitable implementation of the **succeed:** primitive. Intuitively, the implementation of the primitive is easy to describe. When **succeed:** is encountered, a normal return is prohibited because some later backtracking might need to resume at a point after the **succeed:**. Instead of returning to the sender, the primitive must instead cause a copy of the sender context to be "called". This is equivalent to the notion of a **success continuation** as used in typical implementations of Prolog. The copied context can be executed as normal except that returns from it must be modified to repeat the above process; i.e., to continue in this success continuation mode. New message sends are unaffected. Unless interrupted by a **fail** operation, this success continuation mode will eventually lead to executing the last message in the process that contains it. The **fail** operation deactivates the success continuation mode and causes a normal return to be executed; this is done by "backing up" execution to the point immediately after the most recent **succeed:** if there is one, just as though the succeed had simply returned its evaluated parameter. Thus "(self **succeed:** 12) + 5" results in the value 17 being computed after the fail returns execution. A series of snapshots illustrating this notion is shown later.

The implementation relies on a small number of operations that manipulate **contexts**, the Smalltalk terminology for stack frames. Two kinds of contexts are available: **method contexts** (the usual kind) and **block contexts** (contexts for blocks). When a method such as **example1** below executes, the **to:do:** message is sent to 1 which in turn sends a **value:** message to the block as indicated in Figure 1. Three contexts are constructed: a method context for message **example1**, a method context for message **to:do:**, and a block context for executing the block containing the code "sum ← sum + index. sum > 1000 **ifTrue:** [↑index]".

```
Number class method

example1
    "How many consecutive integers must be
    added to get a value larger than 1000?"
    | sum |
    sum ← 0.
    1 to: 1000 do: [:index |
        sum ← sum + index.
        sum > 1000 ifTrue: [↑index]].
    "We never get to this point."

Number instance method

to: aLimit do: aBlock
    "A while loop is used to successively invoke
    aBlock with a changing index. Only the
    block value: message is shown."

    ... aBlock value: aLoopIndexValue ...
```

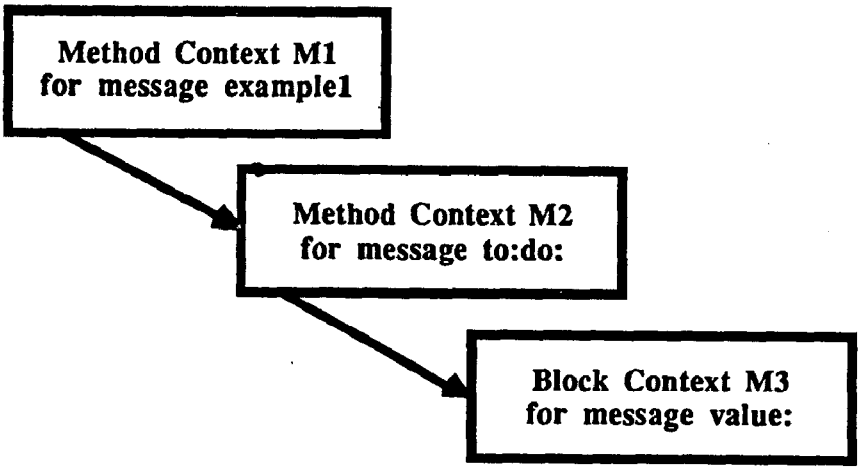

Figure 1: An example calling sequence

A context M's **home** is defined to be M if M is a method context and the defining method context if it is a block context. In the above example, M_3's home is M_1. This notion is particularly important because the return statement ↑index in Figure 1 returns not from the context it is in but from the home context; i.e., it returns from the method context for **example1**. It is just a convenience that the home context for M_2 is M_2 and the home context for M_1 is M_1. Note that M_3 would return to M_2 only if no explicit return statement were encountered; i.e., if the end of the block were reached (in that case, the last value computed by the block is returned).

A summary of the operations used for implementing backtracking is given below with a short explanation. Although these operations are Smalltalk specific, corresponding operations in other suitably powerful languages could be used.

thisContext — A pseudo variable containing the context that is currently executing.

aContext **home** — Returns the home as defined above; e.g., M_3's home is M_1, M_1's home is M_1.

aContext **sender** — Returns the sending context; e.g., M_3's sender is M_2, M_2's sender is M_1.

aContext **method** — Returns the compiled method for the context; e.g., M_1 and M_3's method is the compiled version of **example1**.

aContext **copy** — Returns a copy of the context.

aContext **swapSender**: anotherContext

> Changes the receiver's sender to anotherContext; also, returns the old sender (which we ignore most of the time); anotherContext could be nil instead of a context object.

aContext **tempAt**: i — Returns the value of the i^{th} temporary variable defined in the context; counting starts with the parameters and then proceeds to the local variables; e.g., if a method has parameters p_1 and p_2 and local variables l_1, l_2, ..., the third temporary is l_1.

aContext **releaseTo**: oldContext

> Release (by setting the temporaries in the context stack to nil) all contexts from the receiver, its sender, its sender's sender, etc. up to but excluding oldContext. It is a superfluous method that helps the garbage collector.

aBlockContext **fixTemps**

> Changes the receiver's home context to a copy and sets the copy's sender to **nil**.

A final crucial point is that contexts keep track of where they should resume executing when control returns to them. For example, if we can (somehow) return from M_3 above to M_1, execution would resume at the point where message **to:do:** was initially sent. This is of course what happens when ↑index is executed in M_3.

For tutorial purposes, consider how we might simulate ↑index from within the block without using a return statement. More specifically, consider executing

```
thisContext
    swapSender:
        thisContext sender sender
```

in M_3. Observe that thisContext is M_3. Since M_3's sender is M_2 and M_2's sender is M_1, it should be clear that M_3's sender is changed to M_1. Hence when control reaches the end of the block, it automatically returns to M_1 bypassing M_2. We could have alternatively had it return to M_1's sender by having executed the following instead:

```
thisContext
    swapSender:
        thisContext sender sender sender
```

We now present our three primitives. Method **capture** is trivial but methods **succeed:** and **fail** are quite complex. Without comments, the three methods easily fit on a page (as shown in Listing 4). However, the comments are crucial. Without them, only extremely knowledgeable Smalltalk experts would successfully decipher them; perhaps some of you might consider it a challenge to work from Listing 4. For the others (myself included), the detailed comments and the additional explanation provided by Listing 5 will help substantially.

The Uncommented Code

BlockContext instance methods

backtracking

capture
"If you change this method you must run the following code to make it work:
CaptureMethod ← BlockContext **compiledMethodAt:** #capture"
self **value** "The real work is done by **fail.**"

Object instance methods

backtracking

succeed: aBlockOrPreviouslyEvaluatedExpression
"If you change this method you must run the following code to make it work:
SsucceedMethod ← Object **compiledMethodAt:** #succeed:"

| caller succeedResult callerCopy result |
caller ← thisContext **sender home sender.**

succeedResult ← aBlockOrPreviouslyEvaluatedExpression.
(aBlockOrPreviouslyEvaluatedExpression **isKindOf:** BlockContext)
ifTrue: [succeedResult ← aBlockOrPreviouslyEvaluatedExpression **value**].
result ← succeedResult.

[true] **whileTrue:** [
[caller **method** == SucceedMethod] **whileTrue:** [caller ← caller **tempAt:** 2].
callerCopy ← caller **copy.**
callerCopy **swapSender:** thisContext.
caller ← caller **sender.** "Remember who should get the result."
result ← [thisContext **swapSender:** callerCopy. result] **value**]

fail

| oldContext method succeedOrCaptureContext |
oldContext ← thisContext **sender.**

[((method ← oldContext **method**) == SucceedMethod) | (method == CaptureMethod)]
whileFalse: [
oldContext ← oldContext **sender.**
oldContext **isNil ifTrue:** [self **notify:** 'backtracking failure (off end of process)']].

succeedOrCaptureContext ← oldContext **home.**
oldContext ← succeedOrCaptureContext **sender.**
thisContext sender **releaseTo:** oldContext.
thisContext **swapSender:** oldContext.

method == SucceedMethod
ifTrue: [↑succeedOrCaptureContext **tempAt:** 3 "succeedResult"]
ifFalse: [↑**nil**]

Listing 4: Uncommented primitives

The Commented Code

Two globals are used: CaptureMethod and SucceedMethod (see methods **capture** and **succeed:** for their initialization).

BlockContext instance methods

backtracking

capture
"[expr1] capture evaluates expression1 and returns nil. If backtracking occurs, it is prevented from backtracking past this capture point."

"If you change this method you must run the following code to make it work:
CaptureMethod ← BlockContext **compiledMethodAt: #capture**"

self **value** "The real work is done by **fail.**"

Object instance methods

backtracking

succeed: aBlockOrPreviouslyEvaluatedExpression
"If you change this method you must run the following code to make it work:
SucceedMethod ← Object **compiledMethodAt: #succeed:**"

"Pretend to return the evaluated parameter to the caller, the sender of the sender's home context, exactly like the '↑' instruction does in the sender. This is achieved by copying the caller, making the copy return to this context, and faking a return into the copy. If a fail later occurs, back up to the actual sender of this message with the evaluated parameter as the result. If the copy returns (to here), fake the return to that copy's original sender, skipping over sections currently being traced by other succeed: contexts (a brief discussion explains why below; a slightly more detailed discussion is in the paper)."

| caller succeedResult callerCopy result |
caller ← thisContext **sender home sender.**
"Note: (1) the sender is the method containing 'self succeed: something', (2) the home of the sender is the context that '↑something' would return **from** had it been used instead of 'self succeed: something', (3) the sender of the home of the sender is where '↑something' would return **to.** "

succeedResult ← aBlockOrPreviouslyEvaluatedExpression.
(aBlockOrPreviouslyEvaluatedExpression **isKindOf:** BlockContext)
ifTrue: [succeedResult ← aBlockOrPreviouslyEvaluatedExpression **value**].
result ← succeedResult.

"This method acts as a gate; i.e., when control returns to this method after the block parameter is evaluated, it copies existing methods from the call stack and invokes them after modifying them to return to the gate. More specifically, if the calling code is of the form c1, c2, c3, ..., cn, gate when control returns, it is replaced by (n) c1, c2, c3, ..., gate, cn' (notation ci' denotes a copy of ci); once the gate method is reached again, the code is replaced by (...) ..., then (3) c1, c2, c3, ..., gate, c3', then (2) c1, c2, c3, ..., gate, c2', and finally by (1) c1, c2, c3, ..., gate, c1'. Only a *fail* message will deactivate and pop the gate. Note that gates interior to the call stack; i.e., those between c1 and ..., must be carefully skipped over (they don't get used until the current gate is deactivated with a fail). See the more detailed explanation later in this section."

```
[true] whileTrue: [
    "Skip interior gates by skipping to the context immediately prior to the copied context. Fortunately, this
    context is still in instance variable 'caller' which happens to be the second temporary variable (local
    variables come after the parameters which must also be counted). Note that the following works even if the
    context is a block context instead of a method context because both have access to the home context's
    temporary variables."
    [caller method == SucceedMethod] whileTrue: [caller ← caller tempAt: 2].

    "The following is tricky and works as follows: the copy is changed to return to this context; since contexts
    keep track of their own resume point, it will in this case return from the last call, the message value
    below. When message value is executed, a new block context is created to execute it (this block context, for
    example, would be listed as [] Object >> succeed: in the debugger). This block context's sender is changed
    to the callerCopy by the swapSender: message. When the end of block is reached, result is returned to the
    callerCopy. It in turn will use this result, compute some new result, and return it to this method's last call
    point, the value message, which gets stored in result."

    callerCopy ← caller copy.
    callerCopy swapSender: thisContext.
    caller ← caller sender. "Remember who should get the result."
    result ← [thisContext swapSender: callerCopy. result] value.

    "When the new result comes back from the copy, we go back around the loop and repeat the process with
    the copy's sender, caller."]

fail
    "Causes a return to the context containing the last 'succeed:' message send."

    | oldContext method succeedOrCaptureContext |

    "Find the previous succeed: gate and cancel it (it is an error if none exists)."
    oldContext ← thisContext sender.
    [((method ← oldContext method) == SucceedMethod) | (method == CaptureMethod)]
        whileFalse: [
            oldContext ← oldContext sender.
            oldContext isNil ifTrue: [self notify: 'backtracking failure (off end of process)']].

    "Since the succeed: parameter is sometimes evaluated with a value message, oldContext might not be the home
    context for the succeed: method's context; if it is, asking for the home gets us the original context; otherwise, we
    get the same context back. The same applies to the capture method's context since it sends a value message."
    oldContext ← oldContext home.
    succeedOrCaptureContext ← oldContext.

    "Back up one more step to the sender of the succeed: or capture message."
    oldContext ← succeedOrCaptureContext sender.

    "Release intermediate contexts and fudge this context so that we return to the old context."
    thisContext sender releaseTo: oldContext.
    thisContext swapSender: oldContext.

    "For a succeed: message, return the succeedResult; for the capture, return nil."
    method == SucceedMethod
        ifTrue: [↑succeedOrCaptureContext tempAt: 3 "succeedResult"]
        ifFalse: [↑nil]
```

Listing 5: Commented primitives

Although several points in Listing 5 are tricky, they are accompanied with a hopefully sufficient explanation to make it understandable. However, there is one aspect that has not been fully explained: why we need to access temporary variable "caller" in interior instantiations of method **succeed:**. This is most easily understood by tracing the execution of a sequence of message sends that involves two succeeds at different places.

Consider Figure 2. A_4 and B_3 contain **succeed:** messages. Before following the sequence of contexts that would be created with backtracking, it is worth considering the calling sequence without it; e.g., as it would be if each pair of **succeed:** messages were replaced by a standard return statement. Using $\Rightarrow$ to denote a message send and $\Leftarrow$ to denote a return, it should be clear that the following sequence is observed.

$$
\begin{array}{llllll}
A_1 & \Rightarrow A_2 & \Rightarrow A_3 & \Rightarrow A_4 & & \\
 & & \Leftarrow & & & \\
 & \Leftarrow & & & & \\
 & \Rightarrow B_1 & \Rightarrow B_2 & \Rightarrow B_3 & & \\
 & & & \Leftarrow & & \\
 & & \Leftarrow & & & \\
 & \Leftarrow & & & & \\
\Leftarrow & & & & & \\
 & \Rightarrow C_1 & & & & \\
 & \Leftarrow & & & & \\
\Leftarrow & & & & &
\end{array}
$$

With backtracking, A_4 can't return to A_3, for example. It must make a copy and return into the copy. What does the copy of A_3 return to? To a copy of A_2. This process is illustrated in more detail in Figure 3.

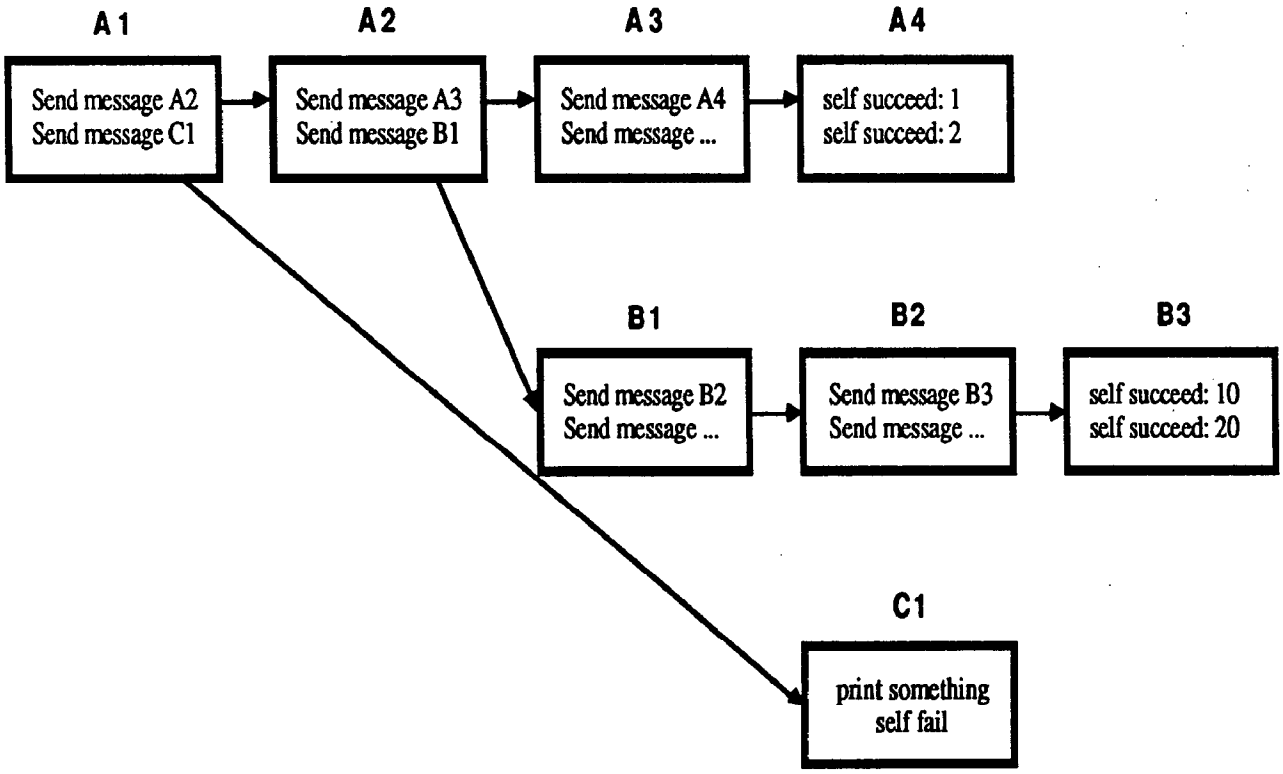

Figure 2: An example calling sequence

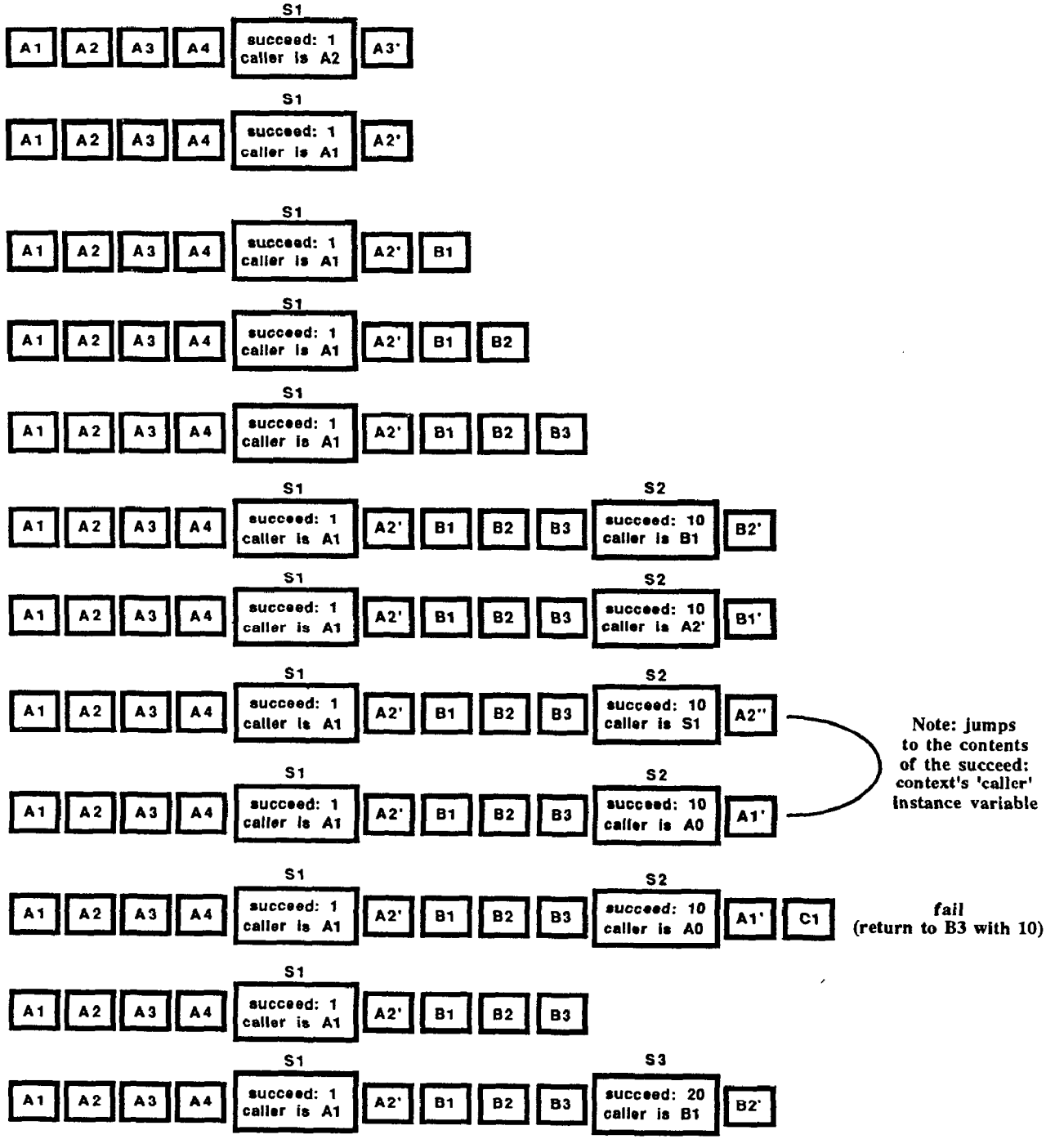

Figure 3: Tracing the example calling sequence
(A3' denotes a copy of context A3; A0 is the calling context that is not shown)

When A4 sends a **succeed:** 1 message, it is clear that it is returning to A3. So the **succeed:** context S1 makes a copy of A3 (denoted A3'), modifies it to return to S1, and remembers A2 in local variable 'caller', the return context for A3'. Then a return into A3' is faked. A3' in turn comes back to S1 which "backs up" the return even further; i.e., the context A2 in caller is duplicated as A2', modified to return to S1, and the new return point A1 saved in 'caller'. As above, a return into A2' is faked. A2' calls B1, which calls B2, which calls B3. Note that "backtrack simulation" occurs only on returns, not on calls.

When B3 sends a **succeed:** 10 message, it is clear that it is returning to B2. The same process as described above is at work. We'll now use a more abbreviated description. S2 fakes a return to B2'. When it returns, S2 fakes a return into B1'. When it returns, S2 fakes yet

another return to A2" (a copy of A2' which is already a copy of A2). Finally, when it returns, we might expect S2 to fake a return into S1', a **succeed:** method context. ***This is not what happens.*** Why not! Intuitively, this is easy to see if we pretend succeeds were normal returns. In that case, returning from A2 (see Figure 2), should lead us back to A1. As you can see, A1 is quite some distance in the stack; i.e., it is separated from S1 by A2, A3, and A4. But note that A1 is the value in S1's instance variable 'caller'. ***This is no coincidence.*** It is the **succeed:** method contexts like S1 and S2 that keep track of the real return points. This is why S_2 must skip over interior **succeed:** method contexts like S_1.

Note that a **succeed:** method context cannot be removed until a **fail** is executed; only the **fail** causes an alternative answer to be requested. The fail is very simple. It simply searches for the top **succeed:** method context in the stack and pops it. The context below is exactly where it must return. Additionally, the value to be returned is in the **succeed:** method context (in temporary variable succeedResult); this can easily be extracted before returning to the context below. Note that other **succeed:** method contexts deeper in the stack are never modified nor is anything else in the stack modified. If they were, we could never restore them to their initial state upon backtracking. That's why the top **succeed:** method context jumps over interior ones rather than attempt to update them in some way.

5 Implementing Backtracking Streams

Backtracking streams need only store a suitable backtracking context that is reactivated when an element is needed.

To ensure that elements are computed only on demand, we keep a cache that is filled only when a query like **atEnd**, **next**, or **peek** forces it. To permit **nil** as a valid stream element, we use a boolean cacheFull to indicate whether or not something is in the cache. Finally, the stream maintains a context that can be used for computing the next stream element; it is called suspendedContext. A new element is obtained (if there is one) by faking a return into the suspended context (this is done by method **switchToSuspendedContext**). This context in turn either computes a new element and sets cacheFull to **true** or determines that there are no more elements, in which case it sets cacheFull to **false**. It then returns by switching back the context once again (the return point was temporarily stored in the suspended context). The standard stream methods are relatively obvious. They are shown below in Listing 6.

Definition of Backtracking Streams

class	**BacktrackingStream**
superclass	**Stream**
instance variable names	cache cacheFull suspendedContext
comment	I represent a finite or infinite set of solutions to some backtracking-based expression. I am created with 'on: aBlock' where aBlock computes values that are returned via 'self **succeed:** aValue' if more values are pending or '↑aValue' is no more are pending. A block with neither represents an empty stream. I support the usual stream messages **atEnd**, **next**, **peek**, **contents**, and **do:** (the last two only if it's finite, of course).

class methods

instance creation

on: aBlock
"Create a new BacktrackingStream with the given block. The block is allowed to backtrack to yield multiple solutions (each of which must be returned one at a time from the block) until the block tries to fail completely. At this point the stream is said to be at the end."

"(BacktrackingStream on: [1 to: 10 do: [:count | self succeed: count]]) contents"

↑self **basicNew on:** aBlock

instance methods

private instance initialization

on: aBlock
"Discussed later."

accessing

atEnd
"Check if there are any alternative solutions left. Note: if necessary, this will force the next solution to be computed and stored for the subsequent 'next' operation."

cacheFull **ifTrue**: [↑**false**].
self **switchToSuspendedContext**. "To obtain the next stream element"
↑cacheFull **not** "Check again, it will be true if one was obtained; false, otherwise."

next
"Force the cache to have the next value, by using 'atEnd'. If there are no values available, generate an error."

| result |
self **atEnd ifTrue**: [self **error**: 'Attempted to read past end of stream'].
"If necessary, the atEnd operation forces a stream element to be computed and cached."
result ← cache. cache ← **nil**. cacheFull ← **false**. "Invalidate cache to force a subsequent element to be obtained."
↑result

peek
"Answer what would be returned with a self next, without moving past the element. If the receiver is at the end, answer nil."

self **atEnd ifTrue**: [↑**nil**] **ifFalse**: [↑**cache**]

contents
"Return the rest of the solutions of the receiver."

| collection |
collection ← OrderedCollection **new**.
self **do**: [:each | collection **add**: each].
↑collection

private

switchToSuspendedContext
"I am responsible for installing and running the suspended context. The sender is saved so that re-executing this method again from within the suspended context will restore the previous state; i.e., to begin executing the suspended context, we perform one switch; to get back, we perform a second switch."

"The following changes the sender to the suspended context so that we return to it at the end of this method. The original sender of this context is returned and stored into the suspended context. A subsequent switch will undo this."
suspendedContext ← thisContext **swapSender**: suspendedContext

Listing 6: Definition of backtracking streams

Note that method **switchToSuspendedContext** in Listing 6 is not really needed because the switch could have been done inline. However, it is more understandable this way. For example, we could have replaced the code 'self **switchToSuspendedContext**' in **atEnd** by

[suspendedContext ← (thisContext
swapSender: suspendedContext)] **value**

Can you see why? The answer is simple. Sending a value message creates a block context instead of a method context. In this block context, thisContext is the block context. The assignment however does not care whether the executing context is a block context or a method context. A return from the context always returns to the sender which the code changed to the suspended context.

The only complicated operation is the private **on:** operation that sets up the suspended context and controls the computation that obtains the successive values through backtracking. We have attempted to document how it works with extensive comments.

The Missing Operation

instance methods

private

on: aBlock
"This code never returns explicitly. Instead, it makes a copy of the block's home context and sets it to return to an appropriate point in the block context initiated by the capture below. This same context is saved as a suspended context and a return to the on: sender is faked. See comments below for more details."

[
cacheFull ← **false.** "Initialize."

"Make a copy of the block's home context and modify it to return here. Note: 'here' is the block context initiated by the **value** message sent by the capture that surrounds this code"
aBlock **fixTemps.** aBlock **home swapSender:** thisContext.

suspendedContext ← thisContext. "Save for subsequent suspensions after each stream element is obtained."

"Fake a return to the sender (a real return will release the context and destroy its contents)."
[thisContext **swapSender:** thisContext "here" **home** "on: context" **sender** "on: context sender". self] **value.**

"The first time the suspended context is resumed, execution will return to this point because the value message above is the last thing executed in this method."

"The next part is tricky. To get things going, we need to execute 'aBlock value'. Since succeed: and ↑ causes 'returns' to the sender of the home context (to here since we modified it above) and since 'aBlock value' will also ultimately return when no more of these answers are available, we need some way to differentiate between the two. Why? Because the former implies that a new stream element is available and the latter implies that there are no more. The solution? Execute 'aBlock value' inside another block (see the assignment to cache below). Since the value message to this containing block is the last message sent in this context, this is where values computed by succeed: and ↑ in the block will be returned. Actually, it returns here for ↑ and to a copy of here for succeed:. Thus cache will receive the stream element. If a fail is executed some time later, backtracking will occur to cause subsequent elements to be computed. Backtracking in the case of ↑ jumps right out of the capture. It is only when no more elements are available that a return from 'aBlock value' will occur. Note that 'aBlock value' is not executed in this context but in the context reached by executing the outer value message. Since no further backtracking is impossible, another fail will cause execution to backup past the capture that surrounds the whole thing. A more pictorial explanation is provided in Figure 4."
cache ← [aBlock **value.** self **fail** "Cause backup out of the capture."] **value.**
cacheFull ← **true.** "Got a element; indicate that we did."
self **switchToSuspendedContext.** "Return to atEnd"
self **fail** "Backtrack to get another value."
] **capture.**

```
"Control returns here only when there are no more stream elements left."
[true] whileTrue: [
    "atEnd was written to switch context even when there are no more answers; so from now on, we indicate
    that there are no more solutions."
    cacheFull ← false. "Indicate there are no more elements."
    self switchToSuspendedContext] "Return to atEnd."
```

Listing 7: The missing backtracking streams initialization operation

Figure 4 is intended to clarify the tricky code initiated by "cache ← [aBlock **value**. self **fail**] **value**". Context C4 executes the code in aBlock. The three possibilities; namely, executing "self succeed: anObject", "↑anObject", and falling off the end of the block are discussed as three separate cases.

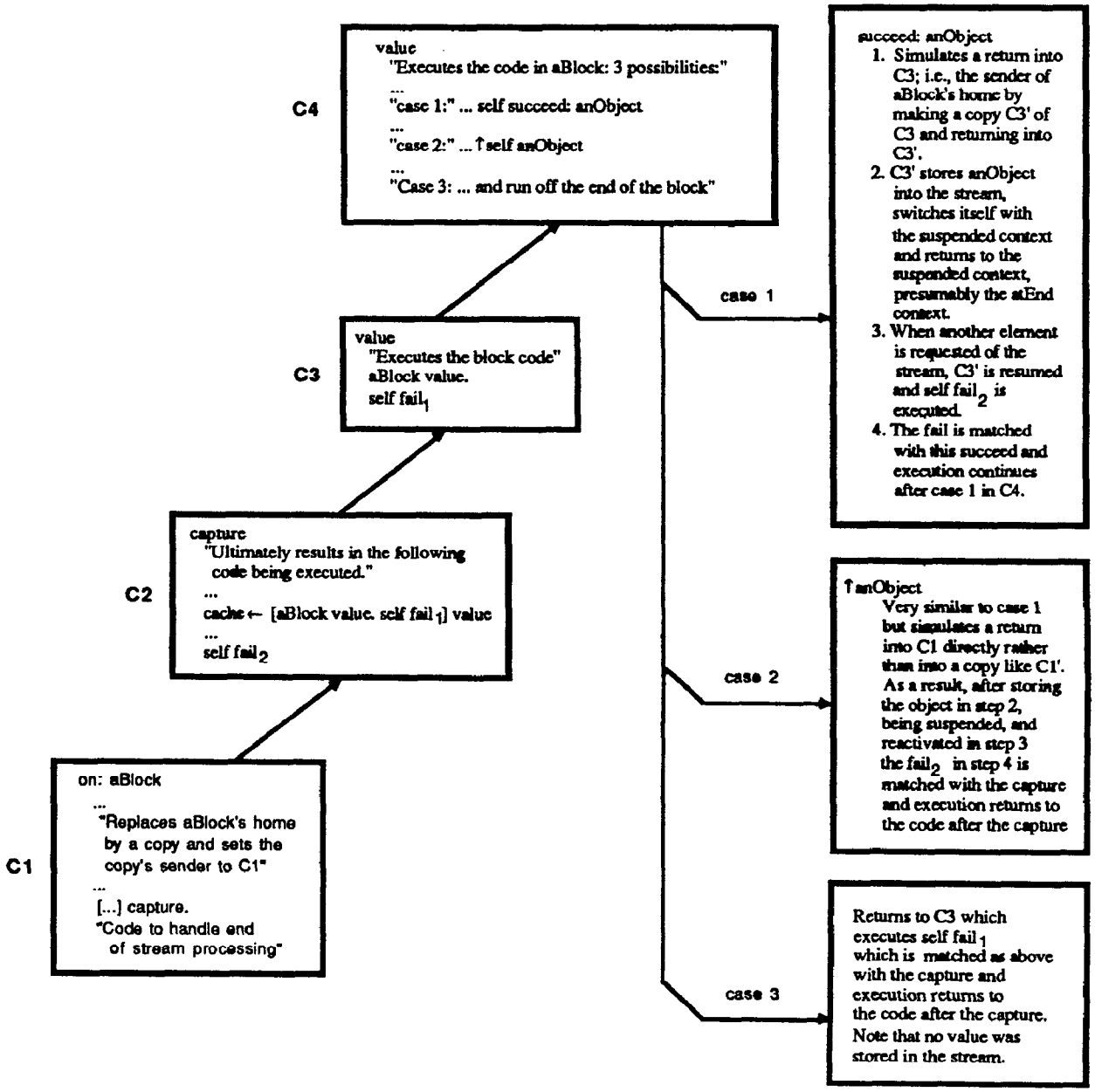

Figure 4: A partial explanation of executing private instance method "on: aBlock"

6 Conclusions

Providing backtracking at the kernel level would not increase the power of the facility although it would make it slightly more efficient.

We have shown how backtracking can be retrofitted into an existing object-oriented language like Smalltalk. This was possible only because Smalltalk permits contexts to be manipulated as first-class objects. The facility could be used for experimentation with knowledge-based programming. It could also be used for designing Prolog-like engines.

Note that the basic approach is not programming language dependent. Translating our implementation to some other suitably powerful language should be easy if the Smalltalk details are well-understood.

References

1. Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*. Springer-Verlag. 1981.
2. Goldberg, A. and Robson, D. *Smalltalk-80: The language and its implementation*. Addison-Wesley. 1983.
3. Griswold, R.E. and Griswold, M.T. *The Icon programming language*. Prentice-Hall. 1983.
4. Griswold, R.E., Poage, J.F., and Polonsky, I.P. *The Snobol4 programming language*. Prentice-Hall. 1971.
5. LaLonde, W.R. A novel rule-based facility for Smalltalk. ECOOP '87, Paris, France, June, 1987, pp. 193-198.
6. Maes, P. *Concepts and Experiments in Computational Reflection*. Proceedings of OOPSLA '87, Orlando, Florida, October 1987, pp. 147-155.
7. Smith, B. *Reflections and semantics in a procedural language*. M.I.T. Laboratory for Computer Science Report MIT-TR-272, 1982.