```
def Drive(j, s, x_0)
 ⟨s', μ⟩ ← s(x_0)  % first message
 % drive the agent until quiesced
 while μ ≠ ∅ do
  % respond to obligation releases first
  while pop @obligate⟨σ⟩ from μ do
    ObligateSet(j,σ)

  % dispatch any lookups in a DFS manner
  while pop @lookup⟨i⟩ from μ do
    ⟨v, m⟩ ← LookupFromBelow(i, j)
    ObligateAdd(j,i)
    ⟨s', μ'⟩ ← s'(@value⟨i, v, m⟩)
    μ ← μ ∪ μ'

  % terminal messages cause us to stop driving
  if    pop @valueIs⟨_, _⟩ as c from μ
     or pop @notify⟨⟩ as c from μ then
    assert μ = ∅ ; return ⟨s', c⟩

% construct an agent only for @lookup⟨⟩
def ConstAgent(v)
 def ret(m)
  assert m = @lookup⟨⟩
  return ⟨ret, @valueIs⟨v, false⟩⟩
 return ret
```

Listing 1: Agent-handling code

```
% Lookup and Compute have merged
def LookupCompute(j ∈ 𝓘) ∈ ⟨χ, bool⟩
 s ← 𝓜(j)  % find or make agent state
 if s = UNK then s ← initState(j)

 active ← active ∪ {j}
 ⟨s', @valueIs⟨v, m⟩⟩ ← Drive(j,s,@lookup⟨⟩)
 active ← active ∖ {j}

 if j ∈ 𝓘_der then 𝓜(j) ← UNK
 maybe
  𝓜(j) ← s'
  % Preserve Invariant 5
  if m then 𝓐_notify(i) ← ∅ ⇐; was UNK
 return ⟨v, m⟩

% Interaction with forward–chaining
% and cycle detection and breaking
def LookupFromBelow(i ∈ 𝓘, j) ∈ ⟨χ, bool⟩
 % try a value temporarily cached in notification
 if σ ⇐; was w = 𝓐_notify(i) and j ∉ σ and w ≠ UNK then
  return ⟨w, false⟩

 if i ∈ active then % cycle? break by guessing NULL
  𝓐_update(i) ← ⇐ UNK
  𝓐_notify(i) ← ∅ ⇐; was UNK
  𝓜(i) ← ConstAgent(NULL)
  return ⟨NULL, false⟩

 % else go see what the agent can tell us
 ⟨v, m'⟩ ← LookupCompute(i)
 m ← (𝓐_notify(i) = σ ⇐; was _) and j ∉ σ
 return ⟨v, m ∨ m'⟩
```

Listing 2: Backward-chaining

Figure 1: Single-threaded EarthBound, with collected changes. These listings introduce the single-threaded adaptor code for agents (section 2.3) and give the replacement backward-chaining machinery, including changes for both notifications with old values (section 2.4.1) and partial propagation (section 2.4.2) as well as strict cycle detection (section 2.5). The memo table and updates now carry *agent states*; notifications carry past *values*, rather than agent states, in their "`was`" components. `active` is assumed to be ∅ at the start of execution.

```
1  def Apply(j, v, s′)
2    M(j) ← s′ % register new agent state
3    A_notify(j) ← ∅ ⇐; was v % queue notification
4
5  def Update(j ∈ I_der, i : ⇐; was w)
6   s ← UNK
7
8   % already an update? use that agent state
9   if j ∈ dom(A_update) then
10    % update with no agent state? have to keep it
11    if A_update(j) = ⇐ UNK then return
12    ⇐ s ← A_update(j) % otherwise, update's state
13
14  % no agent state yet, but have a memo?
15  if s = UNK and M(j) ≠ UNK then
16    s ← M(j) % grab agent from memo
17
18    % try to extract its old value and cache it
19    M(j) ← UNK
20    case s(@lookup⟨⟩) of
21      ⟨_, {@valueIs⟨o, _⟩, _}⟩ →
22        M(j) ← ConstAgent(o)
23
24  % if memo, can queue update
25  maybe if M(j) ≠ UNK then
26    assert s ≠ UNK % must have found one by now
27
28    s′ ← UNK
29    maybe
30      % inform agent, queue new state in update
31      ⟨s′, @notify⟨⟩⟩ ← Drive(j, s, @notifyFrom⟨i, w⟩)
32
33    A_update(j) ← ⇐ s′
34    return
35
36  % no memo or do not want one
37  Apply(j, UNK, UNK)
38
39 def propagate(i)
40   let σ ⇐; was v = A_notify(i)
41   finished ← true
42
43   % visit each obligated child
44   foreach j ∈ (C_i ∖ σ) where obl(i, j) do
45     % optionally, skip this child
46     maybe { finished ← false ; continue }
47     Update(j, i : ⇐; was v)
48
49   % re-queue or remove notification
50   if finished then delete A_notify(i)
51   else A_notify(i) ← σ ⇐; was v
```

Listing 3: Forward-chaining internals

```
1  def Flush(j)
2    M(j) ← UNK
3    % Preserve Invariant 4
4    if j ∈ dom(A_update) then
5      delete A_update(j)
6      A_notify(j) ← ∅ ⇐; was UNK
7
8  def handleUpdate(i)
9    % extract old value; ensured by line 20 of listing 3
10   ⟨_, @valueIs⟨v, false⟩⟩ ←
11     Drive(j, M(j), @lookup⟨⟩)
12
13   ⇐ s′ ← A_update(i)
14   % could compute new state now
15   maybe s′ ← UNK
16   maybe if s′ ≠ UNK then
17     ⟨s′, _⟩ ← Drive(j, s′, @compute⟨⟩)
18
19   % make new state hold
20   Apply(i, v, s′)
21   delete A_update(i)
22
23 def DoneAgenda(j)
24  foreach m ∈ A_update ∪ A_notify do
25   case m of
26    i : ⇐ _ → % updates in ancestry
27      if i ∈ ancestry(j) then return false
28    i : σ ⇐; was _ → % notifications targeting ancestry
29      if (C_i ∖ σ) ∩ ancestry(j) ≠ ∅ then return false
30  return true % only if neither of the above hold
31
32 def RunAgenda(j)
33  until DoneAgenda(j)
34   FreelyManipulateM()
35   peek i : m from A_update ∪ A_notify
36   case m of
37     ⇐ _ → handleUpdate(i : m)
38     σ ⇐; was w → propagate(i)
39
40 def Query(i ∈ I)
41   do % until result unmarked
42     RunAgenda(i)
43     ⟨v, m⟩ ← LookupCompute(i)
44    while m
45   return v
46
47 def UpdateInp(i ∈ I_inp, v)
48   A_update(i) ← ConstAgent(v)
```

Listing 4: More forward-chaining

Figure 2: Single-threaded EarthBound, with collected changes. These listings give the replacement forward-chaining internals and user methods. The agenda is split into two components, $\mathcal{A}_{\text{update}}$ and $\mathcal{A}_{\text{notify}}$ so that items may have both updates and notifications pending. The obligation management and query functions, ObligateAdd, ObligateSet, and obl are opaque. The @compute⟨⟩ message invented for line 17 of listing 4 is much like @lookup⟨⟩ but indicates a request for a potentially newer value.