A Model for Concurrency using Single-Writer Single-Assignment Variables

CPA 2011, Limerick, Ireland

Matthew Huntbach
School of Electronic Engineering and Computer Science
Queen Mary, University of London
mmh@eecs.qmul.ac.uk

An Abstract Model for Computation with Agents acting Concurrently

- The universe consists of Agents which have shared access to Variables
- Each variable must have exactly one agent which can write to it (single writer)
- Each variable may have any number of agents which can read it (but see linearity later)
- Once a variable has been written to, it cannot be written to again (single assignment)

Reduction to Assignments

- The simplest form of agent is an Assignment
- An assignment represents a variable assigned to a tuple containing a tag and further variables
- An assignment counts as the writer of the variable it assigns to
- An application of a computation rule is an agent reducing to a network of agents
- An agent which consists only of assignments cannot reduce further
- When an agent reduces, every variable to which it is a writer must have exactly one agent which is its new writer, the new agent may be an assignment

Process

- A Process is an agent which consists of a set of Rules
- A Rule consists of a left-hand-side (lhs) which contains Matches, and a right-hand-side (rhs) which is an agent (so could be a network of agents)
- A match consists of a variable which is matched, and a Tuple it matches, consisting of a tag and any number of new variables
- A lhs may have 0 or 1 matches for each variable to which the process has read access
- A lhs may have 0 or 1 matches for each variable which is in the tuple of another match in the lhs
- A rhs must put every variable to which the process has write access in a write position in its network

Reduction

- A Reduction occurs when a tuple assigned to a variable to which a process has read access has the same tag and same number of variables as a match on the lhs of a rule
- The variables in the tuple are linked to the variables in the match, and the match is removed from the rule
- If a rule has no matches, the process reduces to the network of agents which is its rhs. The other rules are discarded, and the rhs is incorporated directly into the universe of agents
- Assignments on the rhs to variables which the process had write access may then match matches on the lhs of other processes which have read access to those variables

Futures

- A variable may be assigned a tuple which contains variables which have not yet been assigned
- Such a variable may then be used like any other variable
- The result is to build structures with “holes” which are later written to
- A process which needs to know the value of a variable for a match to reduce cannot reduce until it is assigned
- This is like a “future” but requires no special syntax

Linear Variables

- Variables may be designated either linear or non-linear
- Linear variables must have exactly one reader as well as exactly one writer
- A linear variable may be assigned a tuple containing non-linear and linear variables
- A non-linear variable may only be assigned a tuple containing non-linear variables

Back Communication

- An assignment to a non-linear variable can only have read access to variables in its tuple
- An assignment to a linear variable may have write access rather than read access to variables in its tuple
- On reduction, the reader of a linear variable which is assigned a tuple which has write access to further variables must put each of those variables in exactly one write access position

Long-lived Agents

- When an agent reduces to a network of agents, we may consider one of them its successor
- When a variable is assigned a tuple, we may consider one of the variables in the tuple as its successor
- Agents may have rules in which one of the agents in the network of agents on the rhs is a recursive process
- This enables us to model long-lived agents communicating over channels,

Non-determinacy

- The lhs of rules in a process need not be mutually exclusive
- This enables us to have non-determinate agents: a situation may exist where more than one rule may be applied
- There is no backtracking to alternative once a rule is selected
- There is an indeterminate time delay between a rule becoming applicable due to all matches being matches, and the rule being applied, so we cannot say that if another rule becomes applicable, the first will still be used

The World

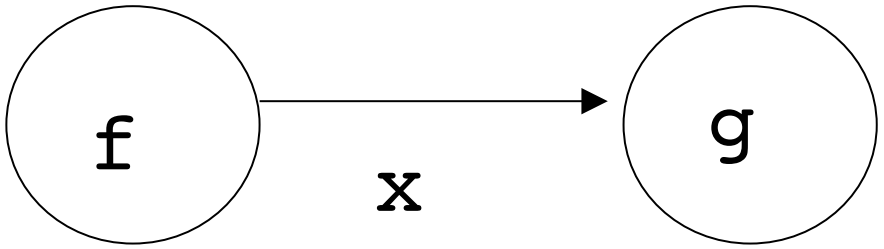
- The world can be considered just another agent within the agents network, which communicates through an interface of shared variables.
- So all situations can be considered in terms of a closed network
- The rhs of a rule can be considered a universe in which the world is an additional agent whihjc reads the variables the rhs has wrote access to and writes to the variables the rhs has write access to
- Any subset of a network of agents can have a boundary drawn round it and the whole considered “an agent”

A graphical representation

- Agents can be written as nodes in a directed graph
- A linear variable can be shown as an arc from one agent (its writer) to another (its reader)
- A non-linear variable with more than one readers can be shown as an arc leading to a Duplicator, with two arcs leading from the duplicator to other agents or further duplicators
- A non-linear variable with no readers can be shown as an arc leading from an agent to an Eraser

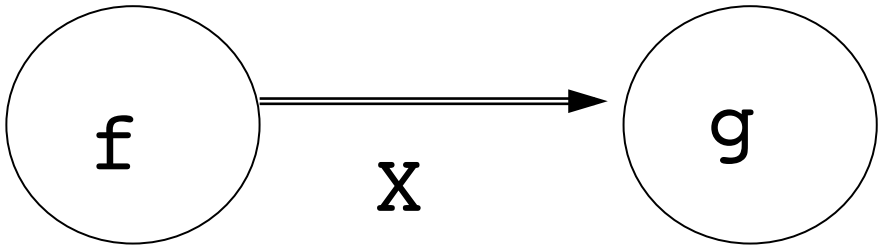
A variable with one reader and one writer

```
f(…)->(…,x,…), g(…,x,…)->…
```



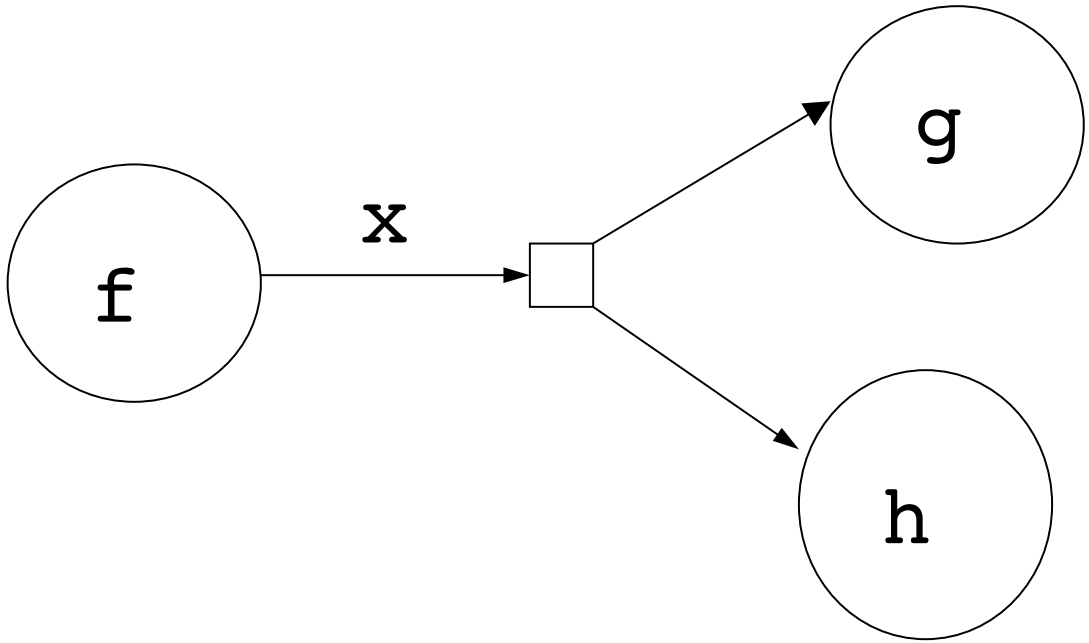
A linear variable with one reader and one writer

```
f(…)->(…,X,…), g(…,X,…)->…
```



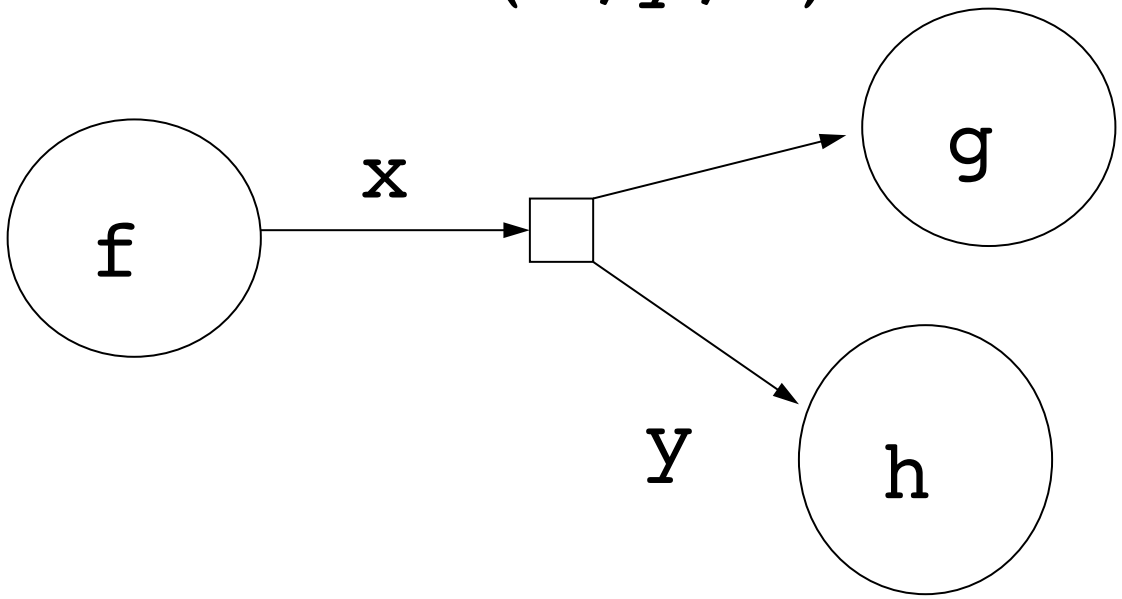
A non-linear variable with two readers and one writer

`f(…)->x, g(…,x,…), h(…,x,…)`



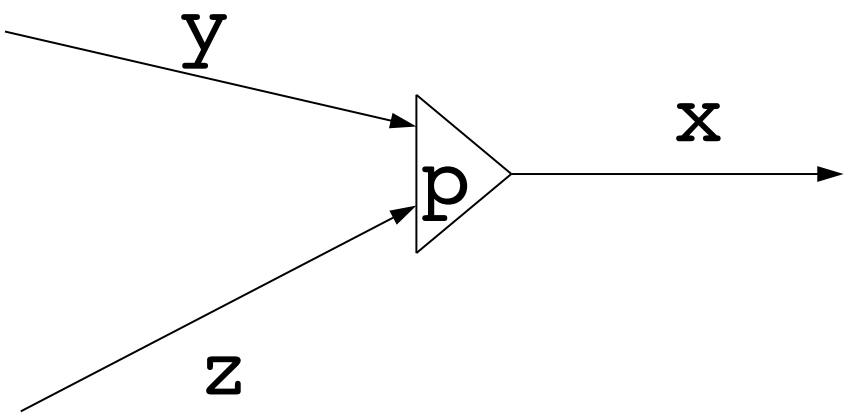
Non-linear variable with two readers and one writer showing variable to variable assignment

```
f(…)—>x, g(…,x,…), y<—x,  h
                  (…,y,…)
```



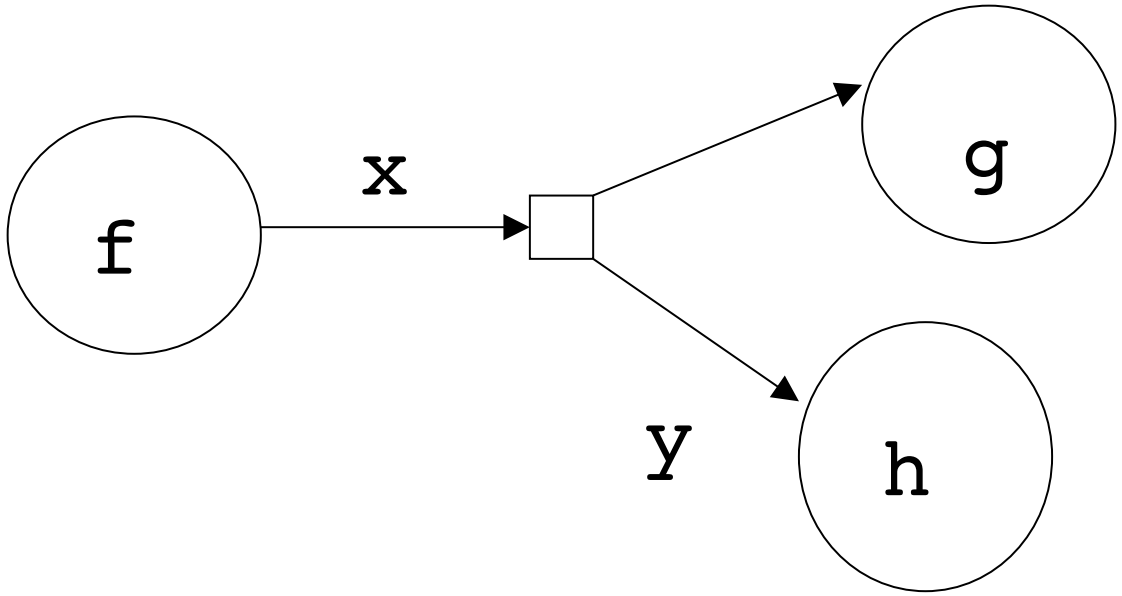
Assignment

```
x=p(y,z)
```



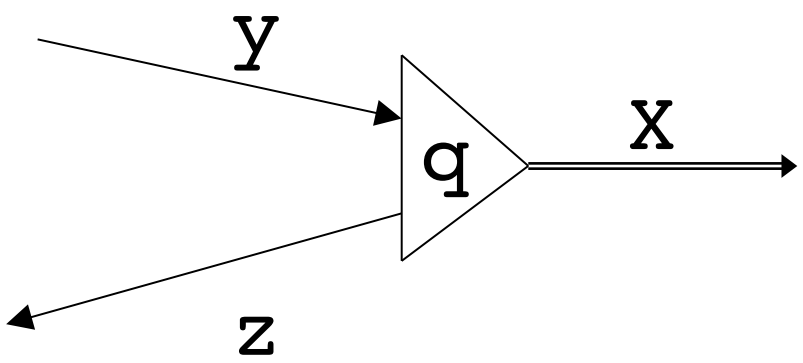
A non-linear variable with two readers and one writer showing variable to variable assignment

```
f(…)—>x, g(…,x,…), y<—x,  h(…,y,…)
```



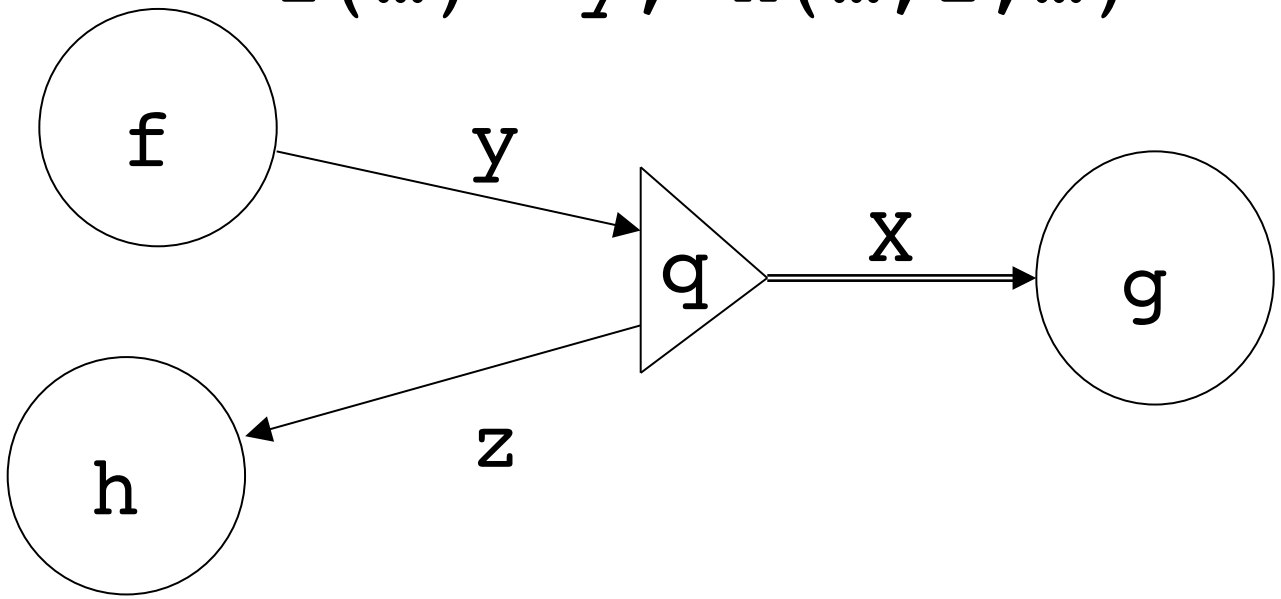
Assignment with back communication

```
X=q(y)->z
```

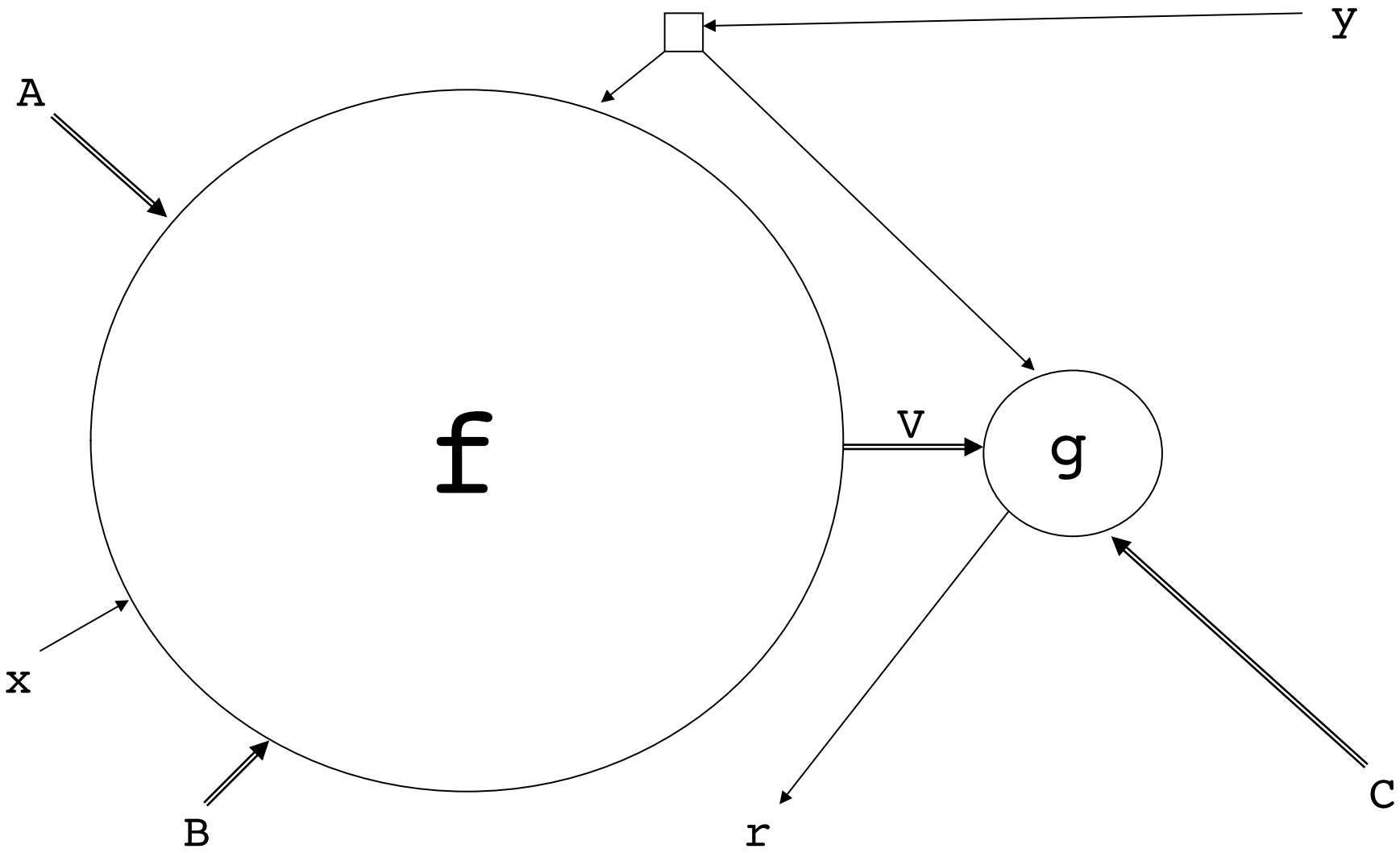


Three-way communication

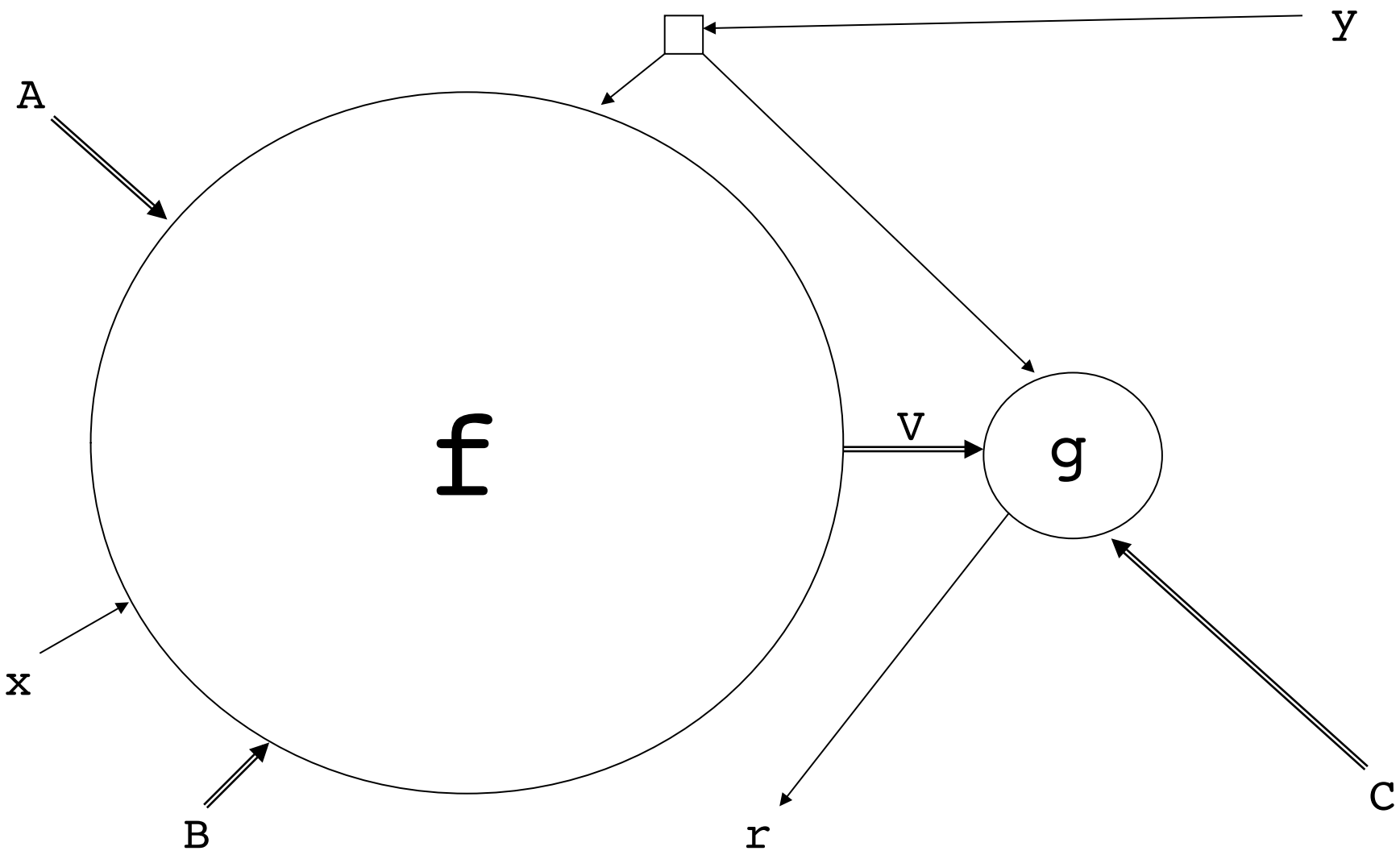
```
X=q(y)->z, g(…,X,…),
f(…)->y, h(…,z,…)
```



Communication through assignment extrusion and absorption

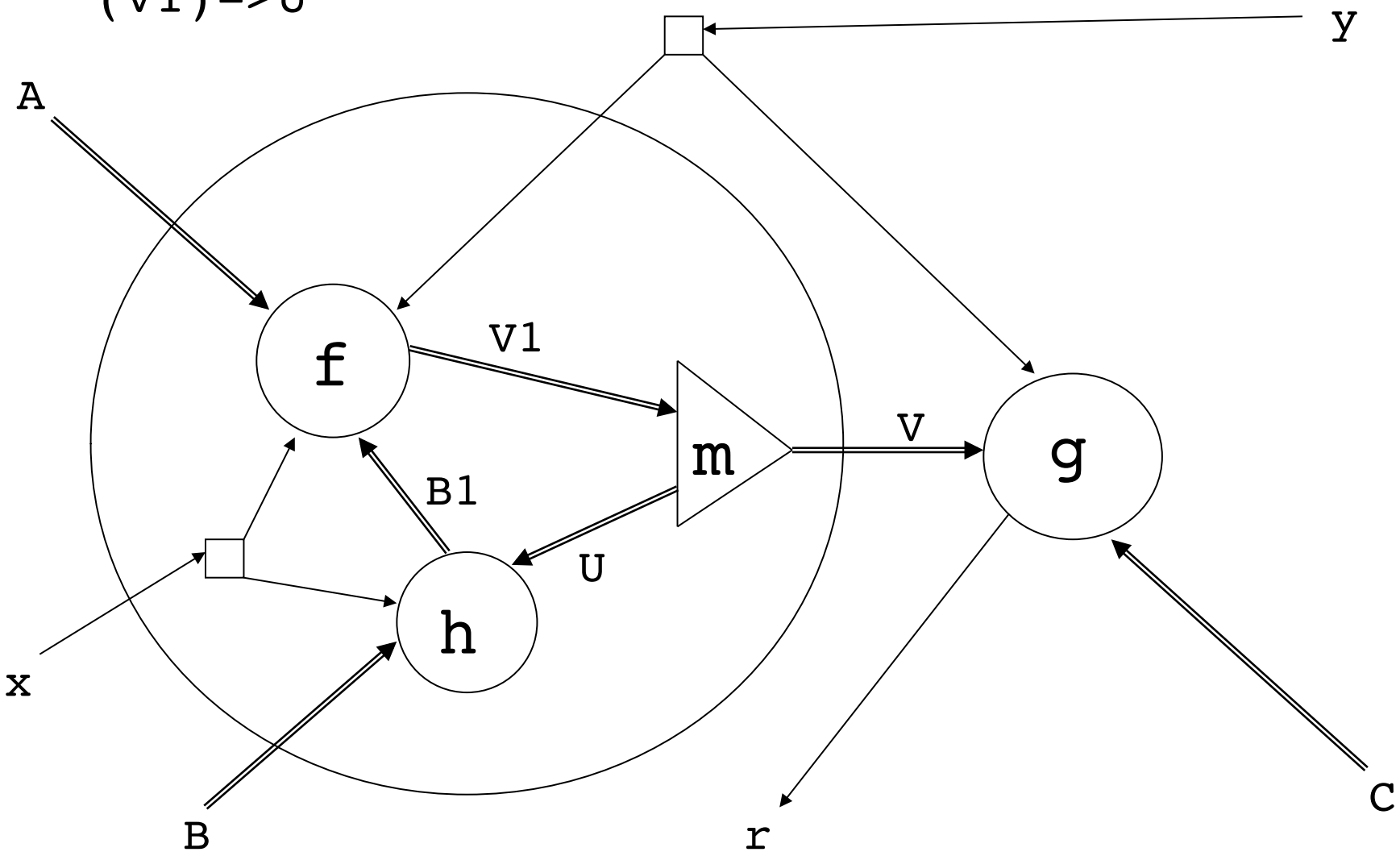


f(A,x,B,y)—>V, g(y,V,C)—>r

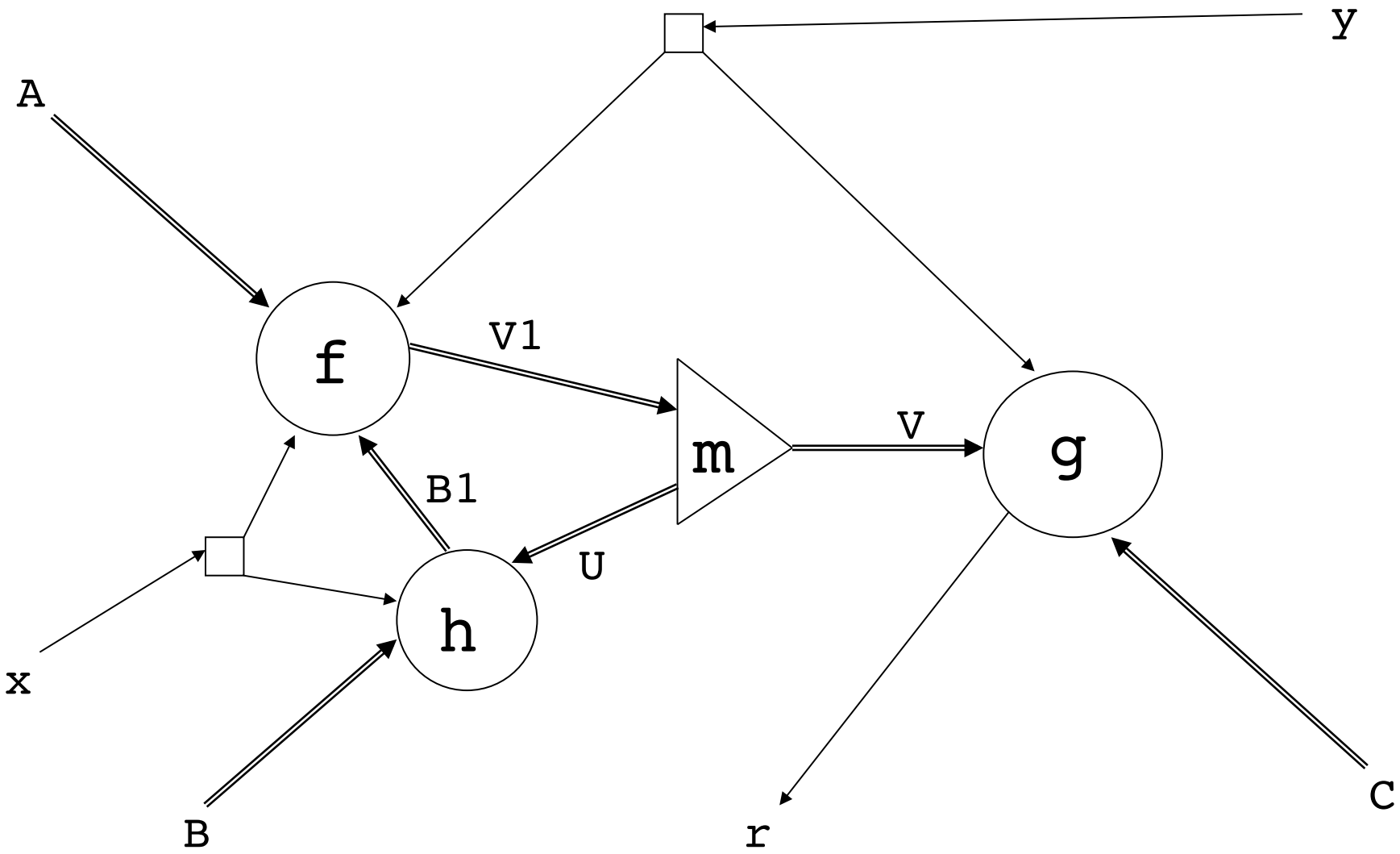


f(A,x,B,y)–>V ⟹

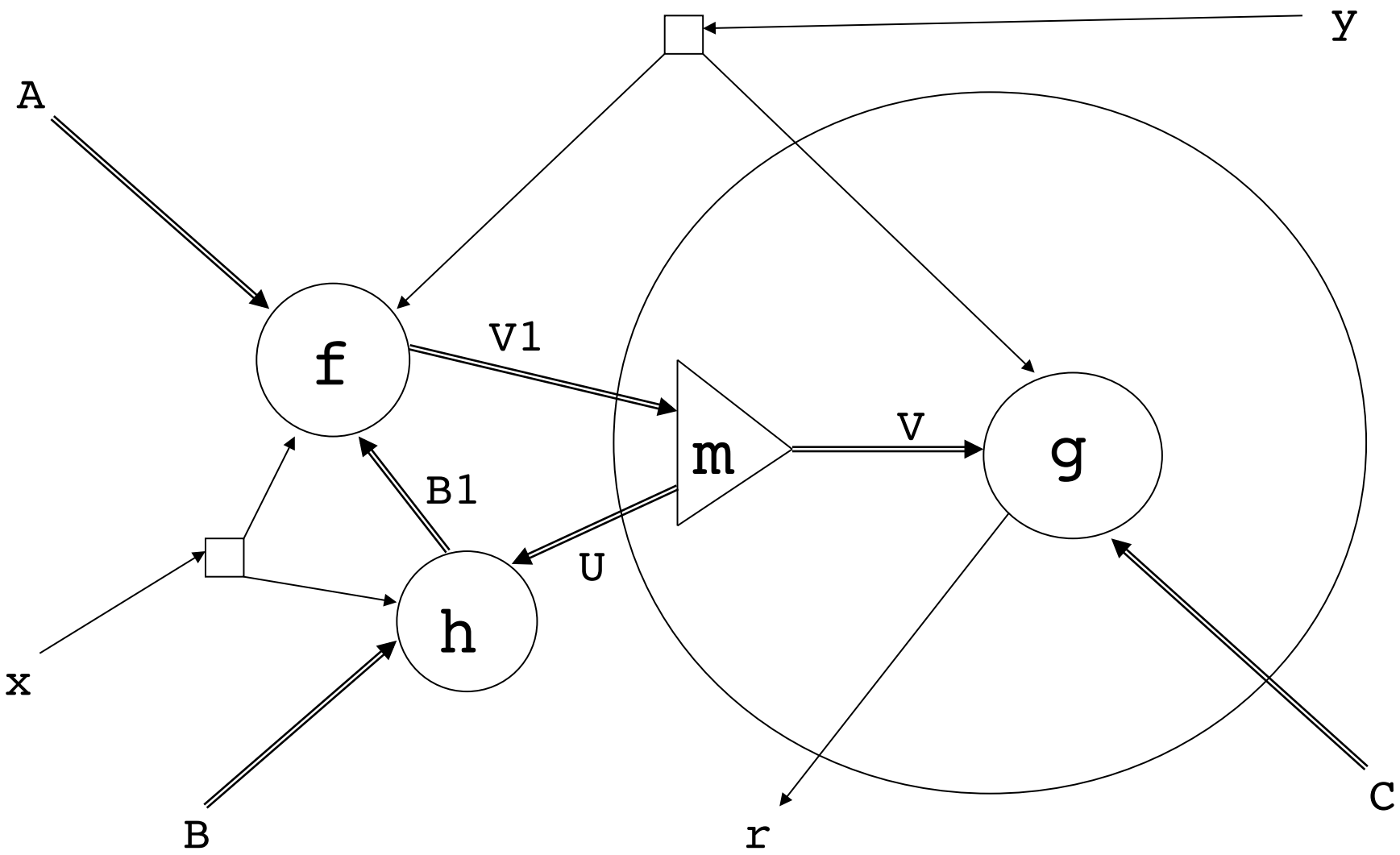
f(A,x,B1,y)–>V1, h(B,x,U)->B1, V=m(V1)->U



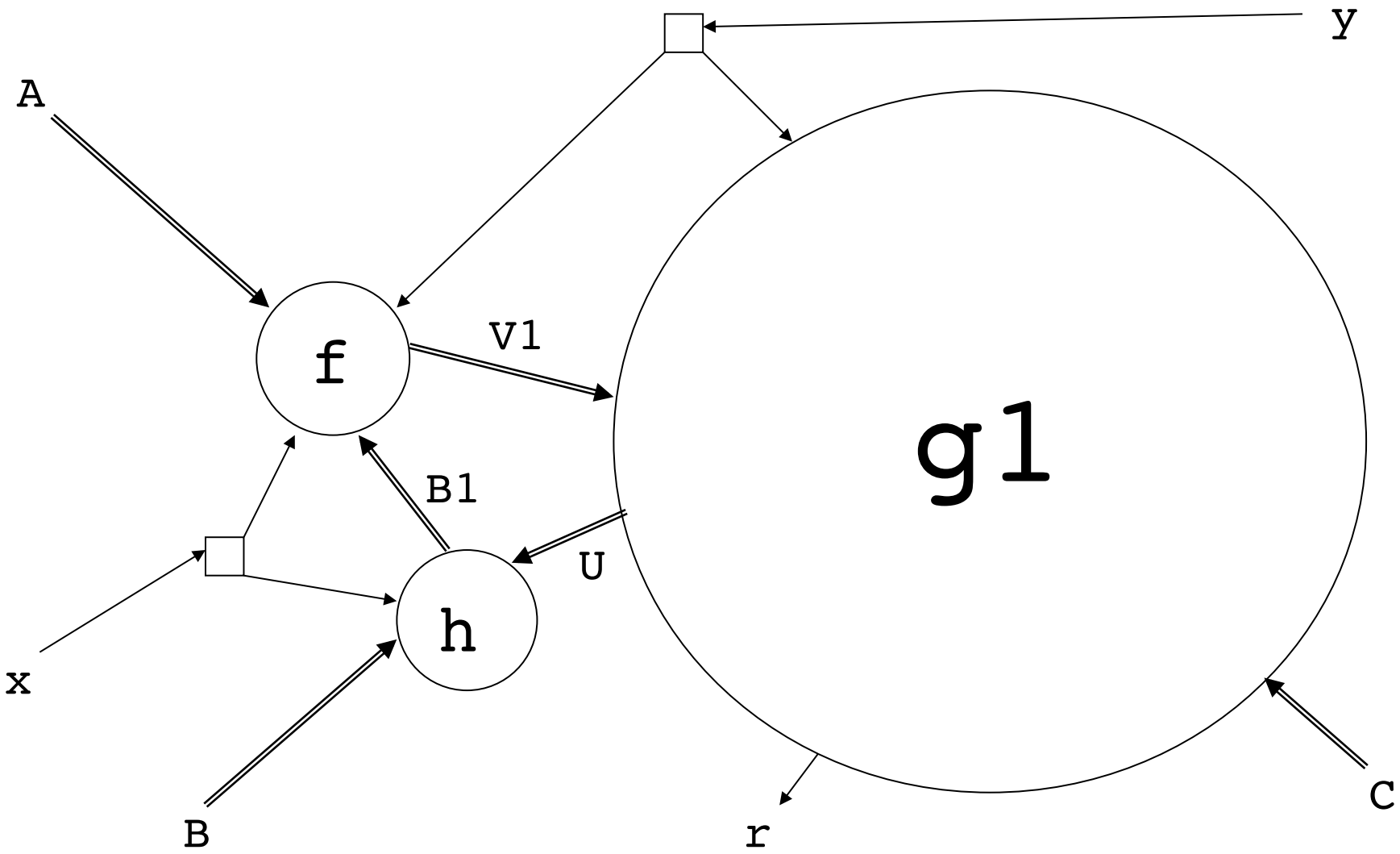
f(A,x,B1,y)->V1, g(y,V,C)->r h (B,x,U)->B1, V=m(V1)->U



g(y,V,C)->r @ V=m(V1)->U ⟹ g1(y,V1,C)->(r,U)

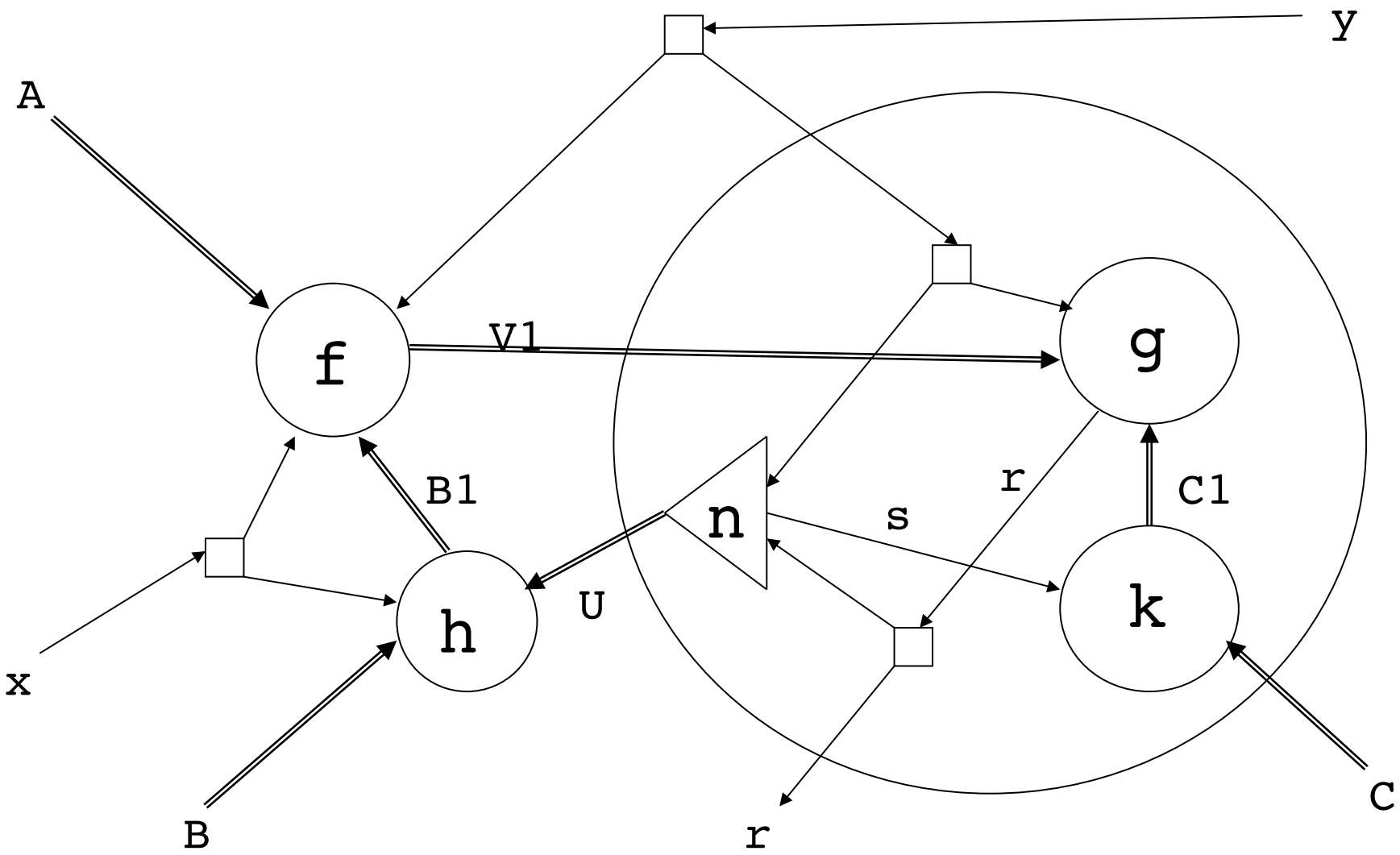


f(A,x,B1,y)—>V1, g1(y,V1,C)—>
(r,U) h(B,x,U)—>B1

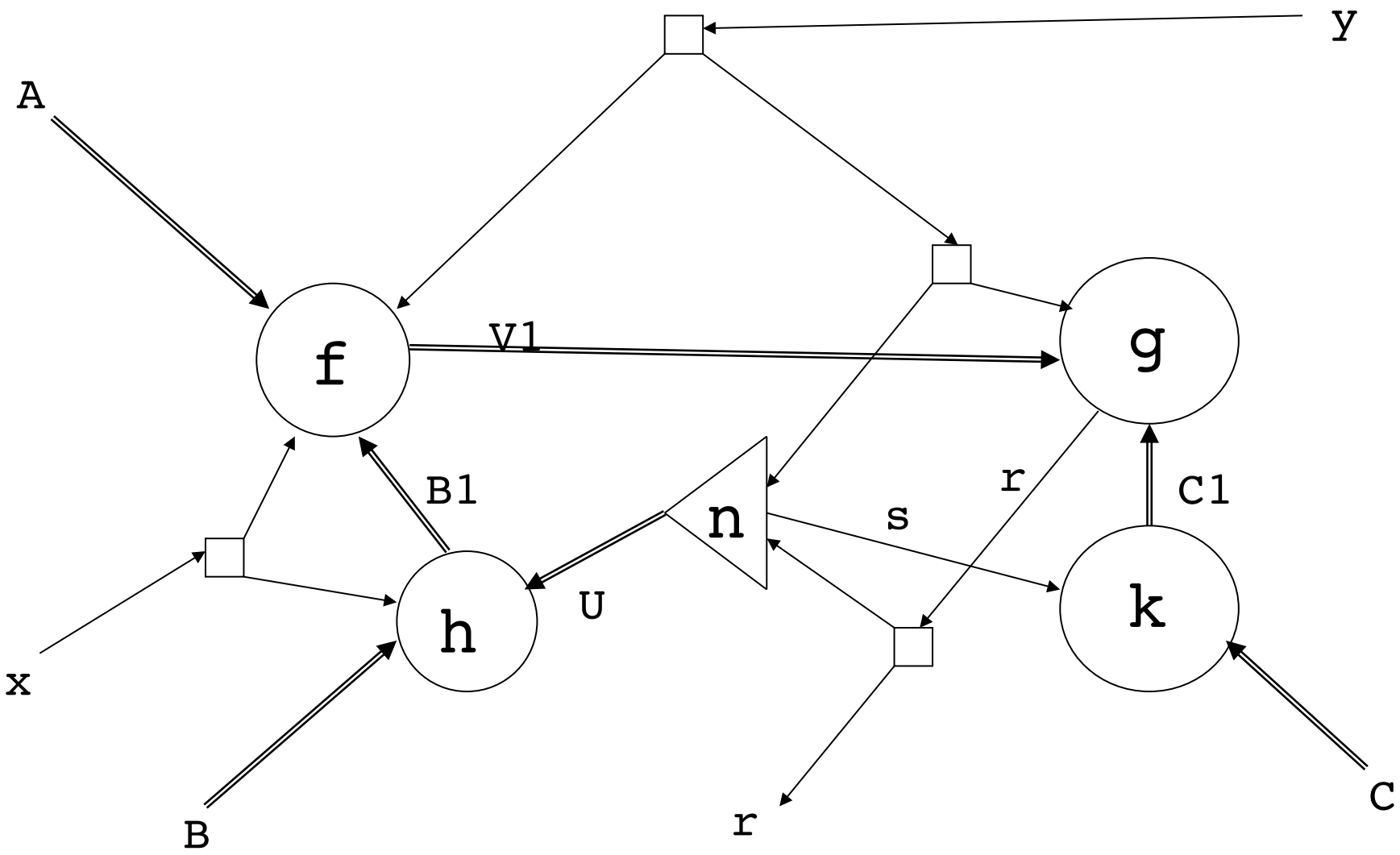


g1(y,V1,C)->(r,U) ⟹

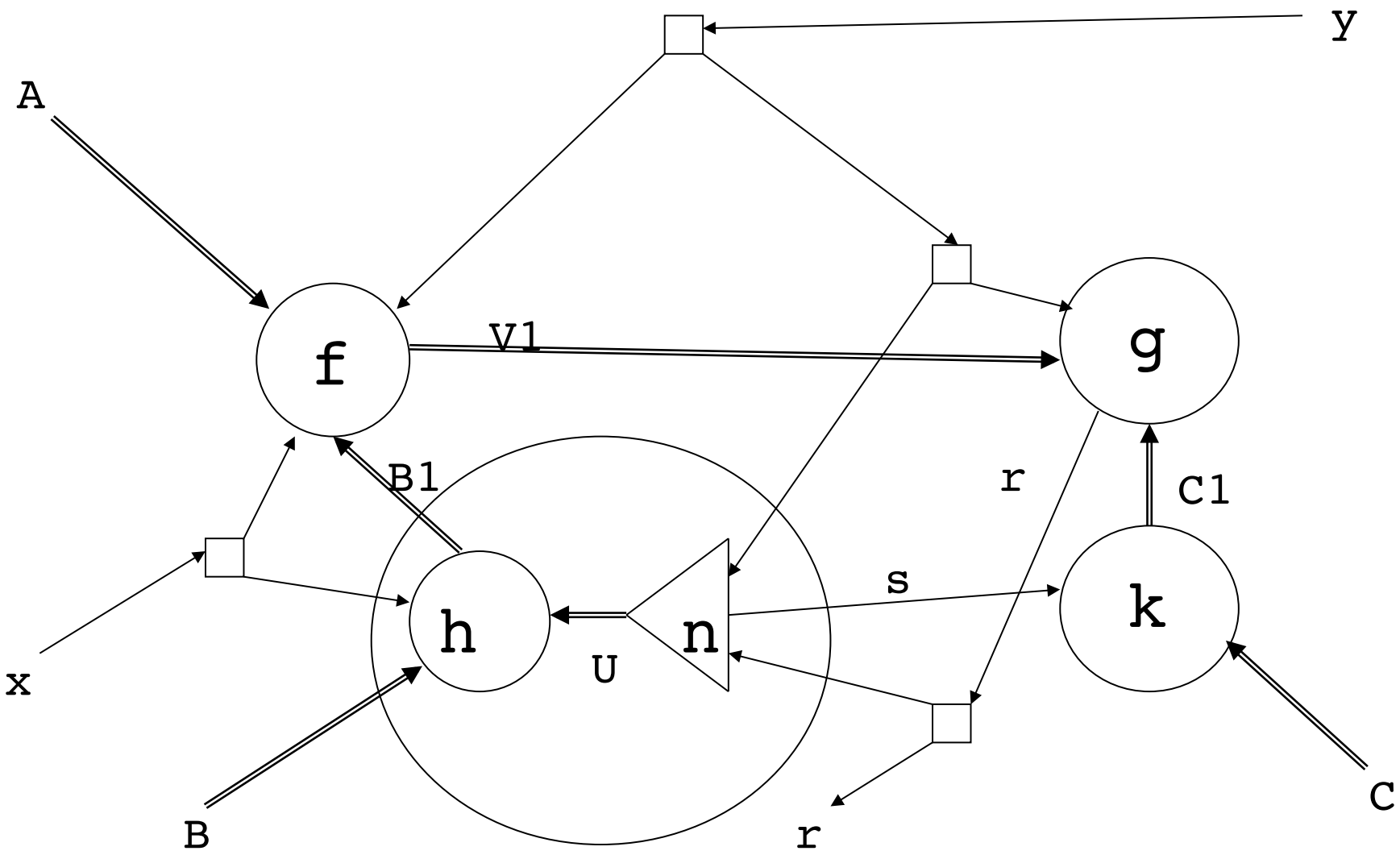
g(y,V1,C1)->r, U=n(y,r)->s, k(s,C)->C1



f(A,x,B1,y)—>V1, g(y,V1,C1)—>r h (B,x,U)—>B1, U=n(y,r)—>s

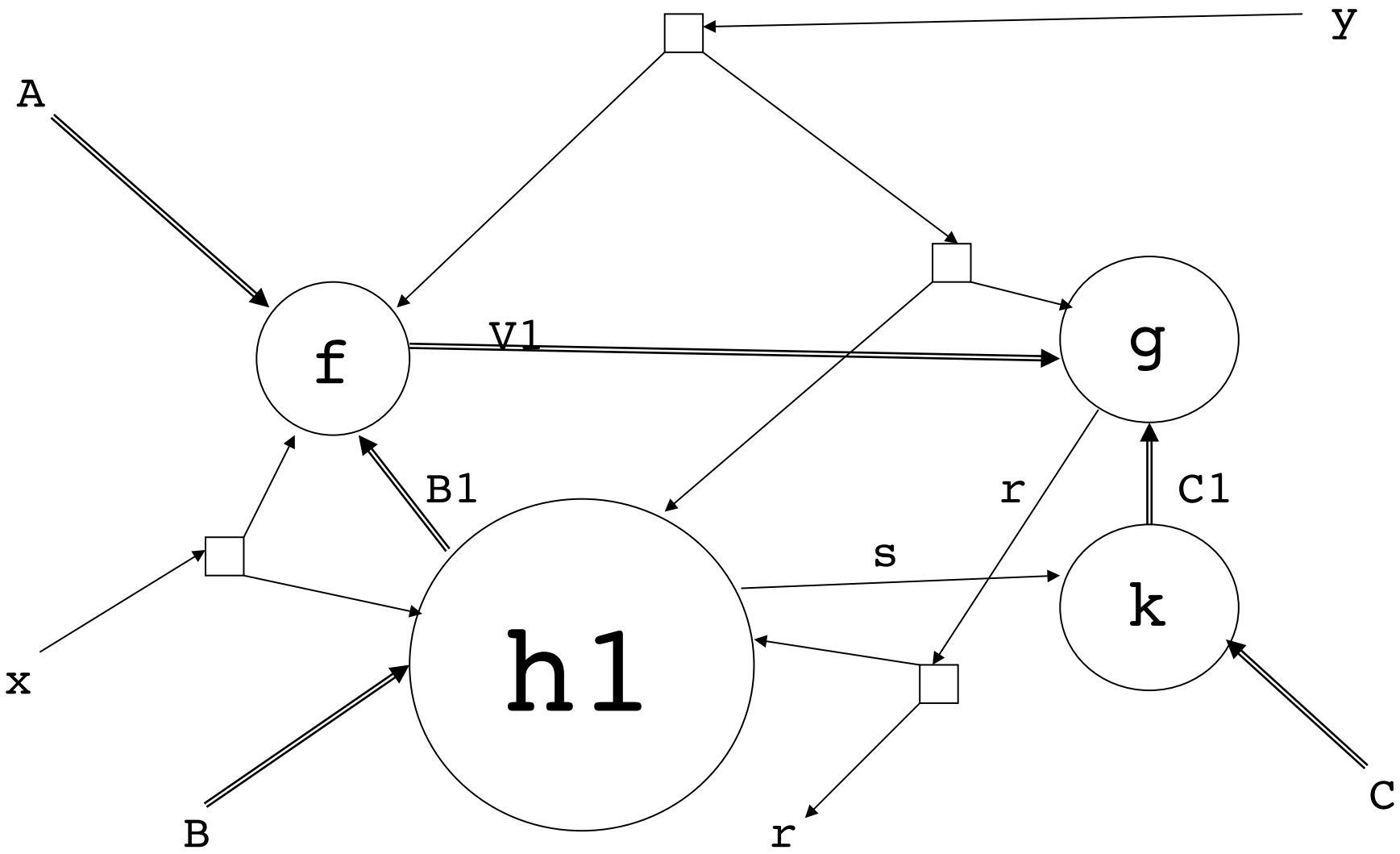


f(A,x,B1,y)—>V1, g(y,V1,C1)—>r h (B,x,U)—>B1, U=n(y,r)—>s

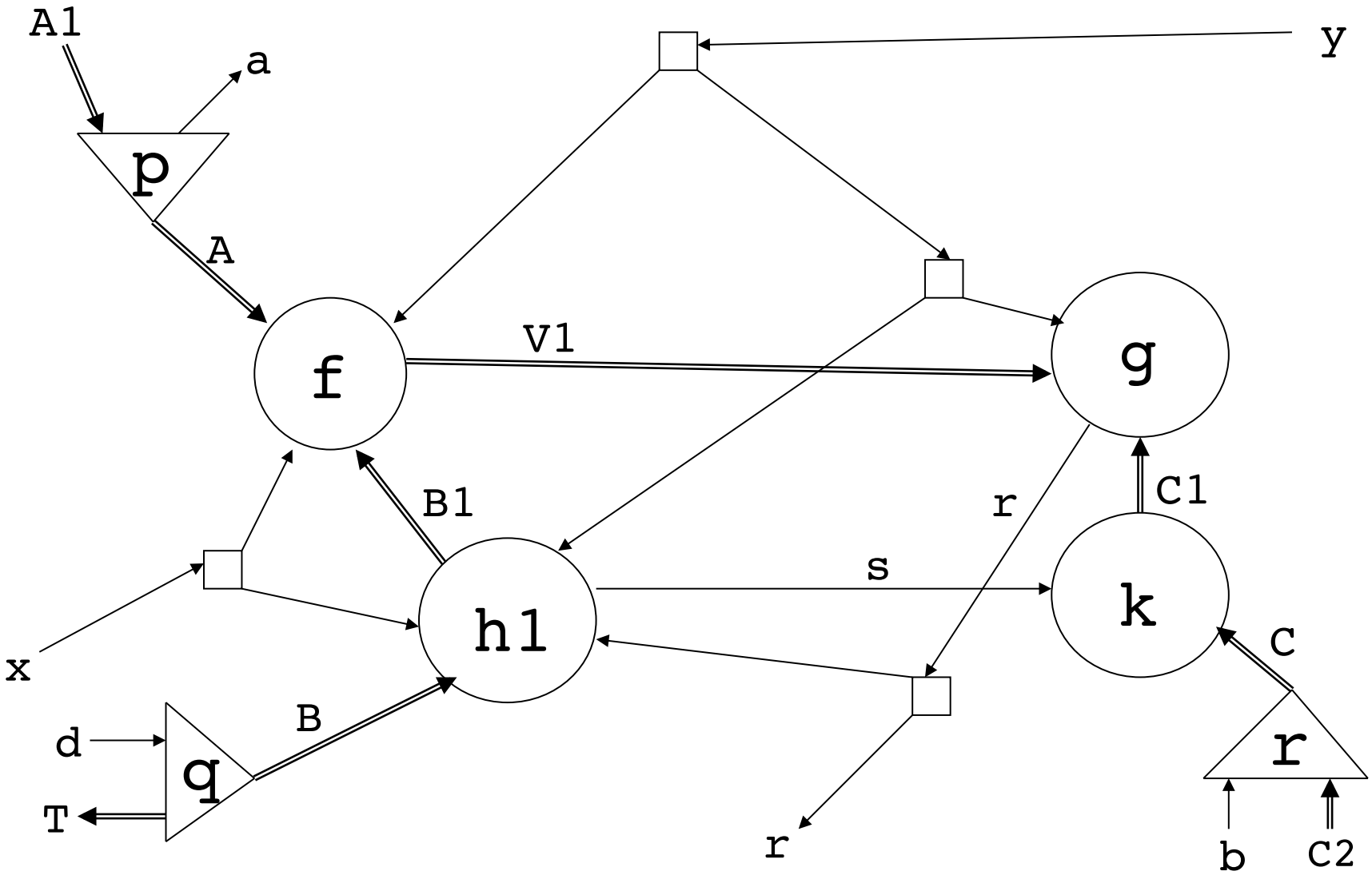


h(B,x,U)—>B1 @ U=n(y,r)—>s ⟹

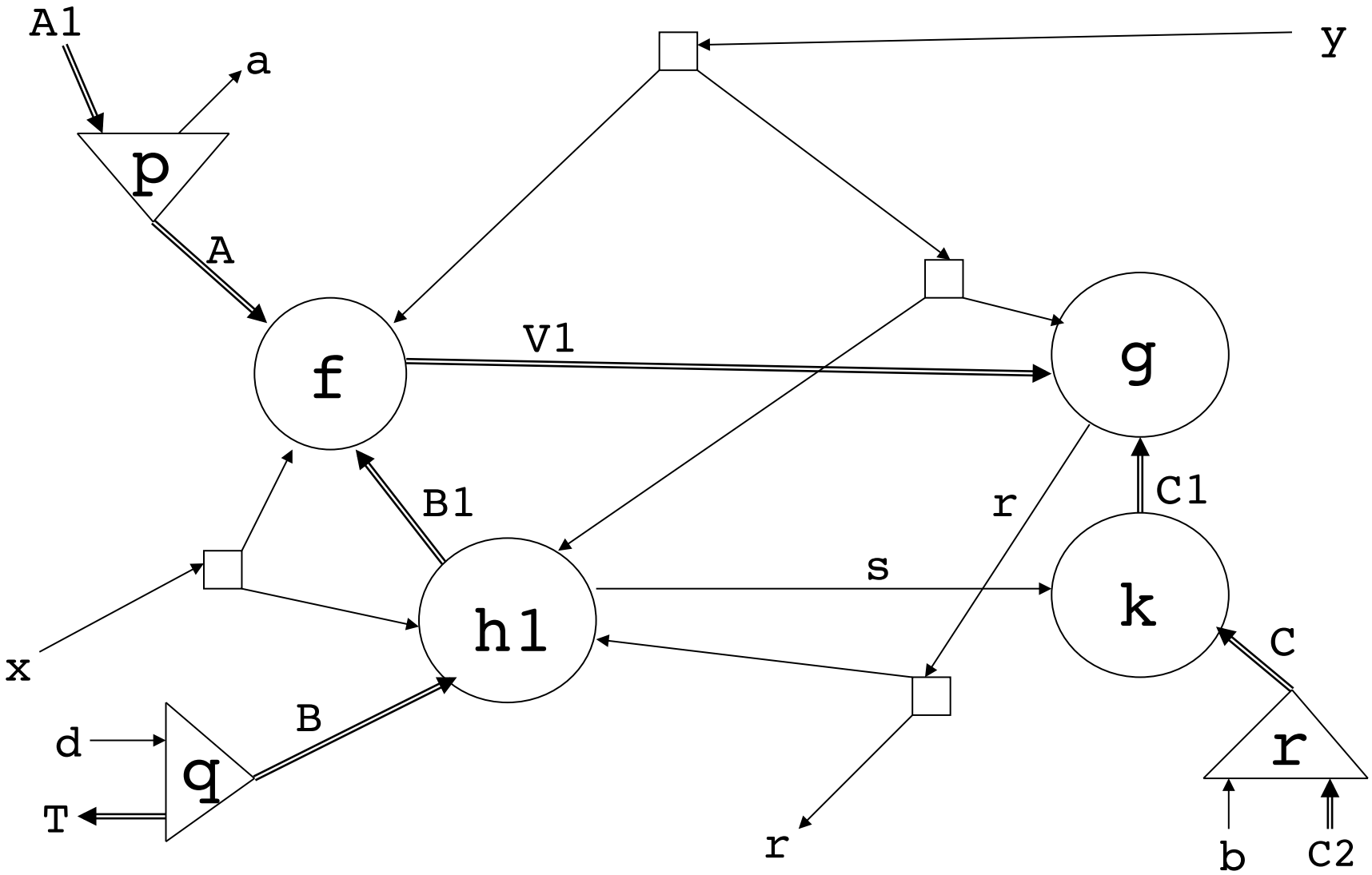
h1(B,x,y,r)—>(B1,s)



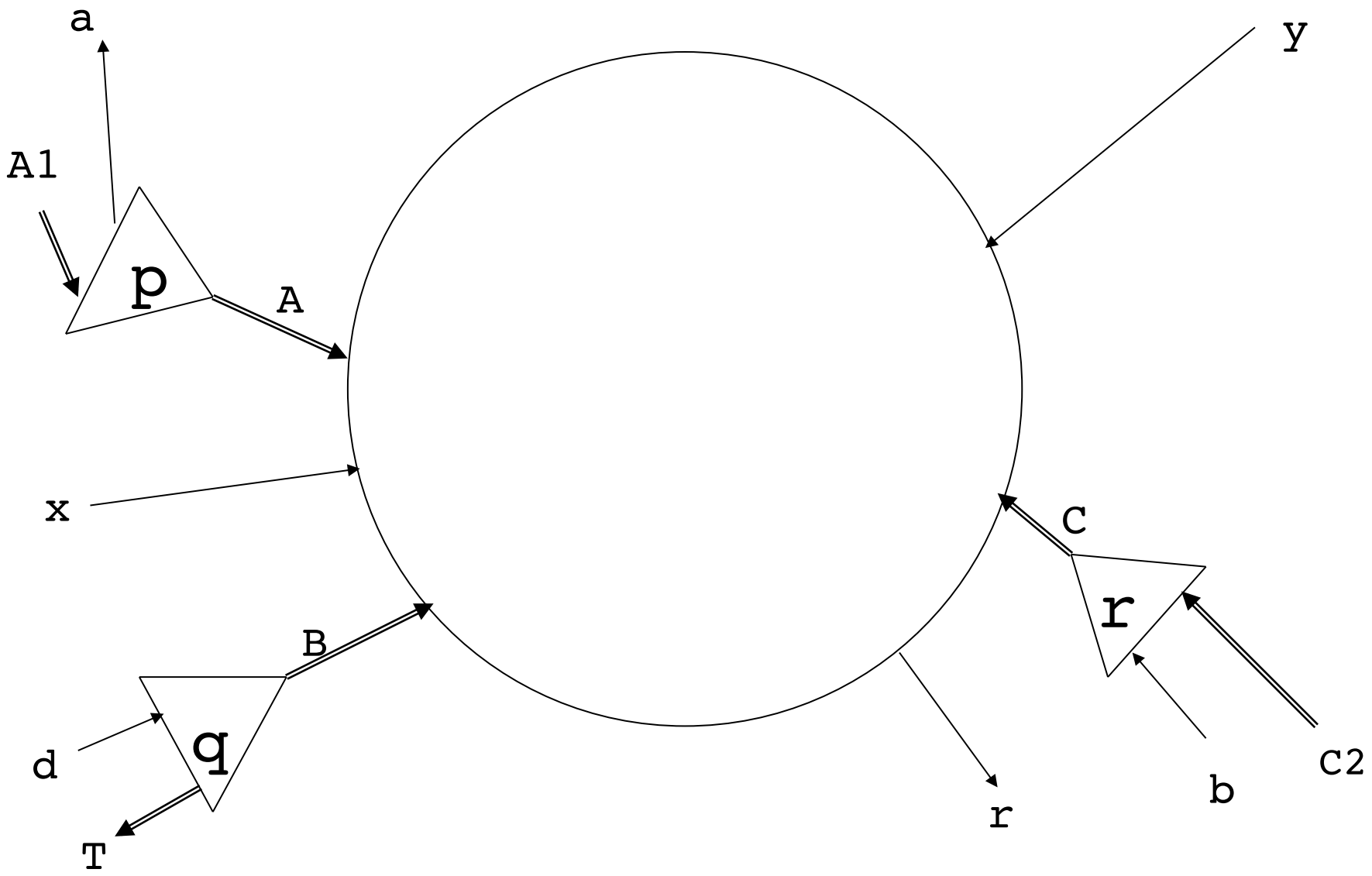
A=p(A1)->a, B=q(d)->T, C=r(b,C2)



Nondeterminacy

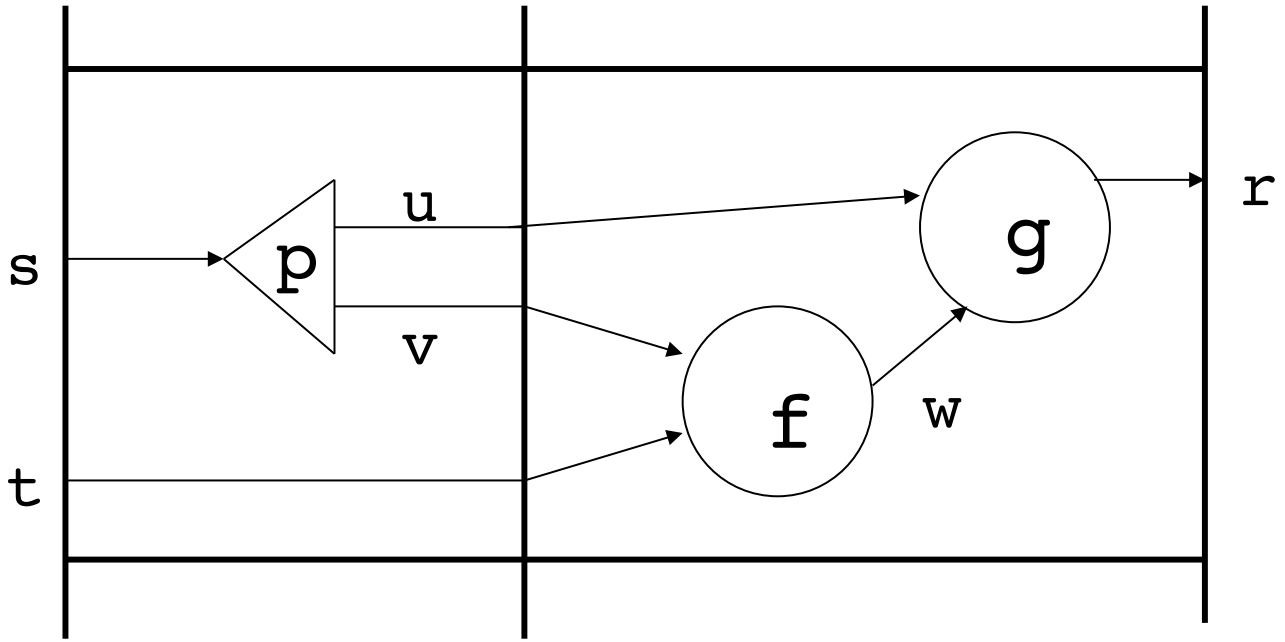


Nondeterminacy



Rule with one match

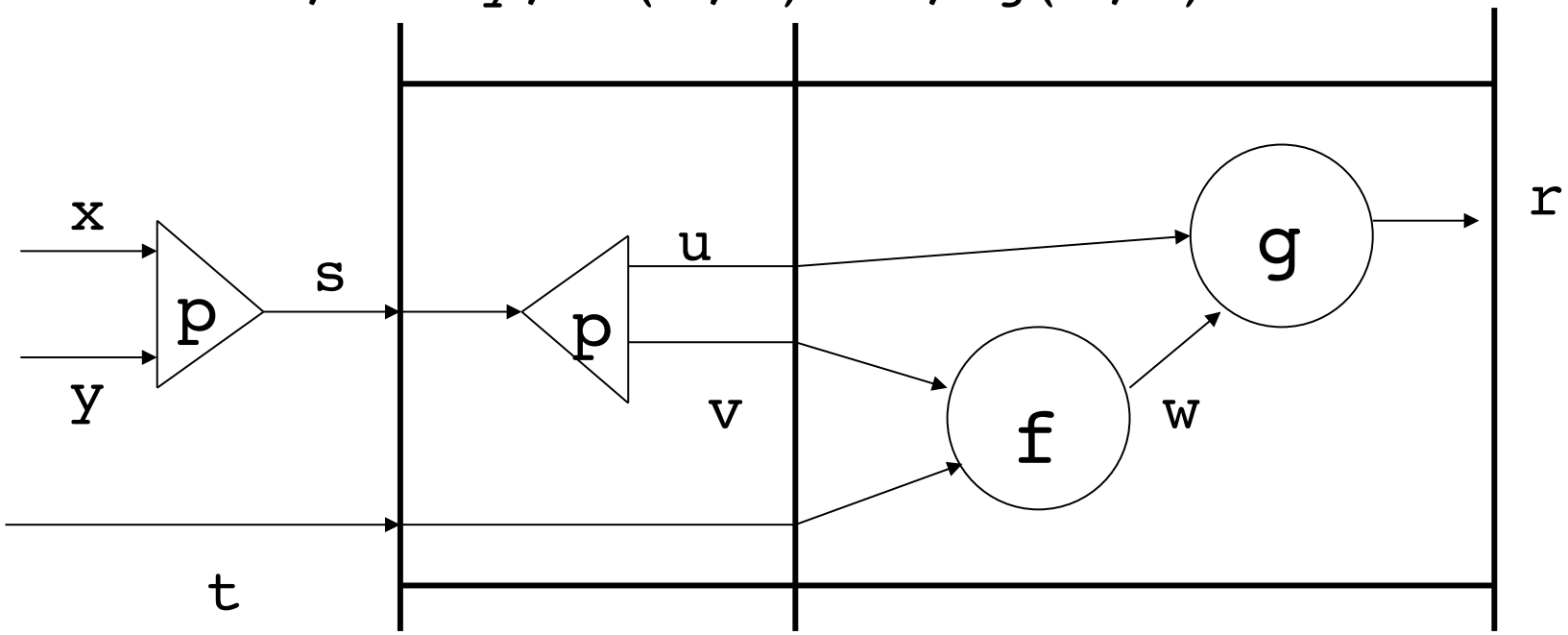
```
h(s,t)->r
{
…
   s=p(u,v) || f(v,t)->w, g(u,w)->r;
…
}
```



Successful match and rule reduction

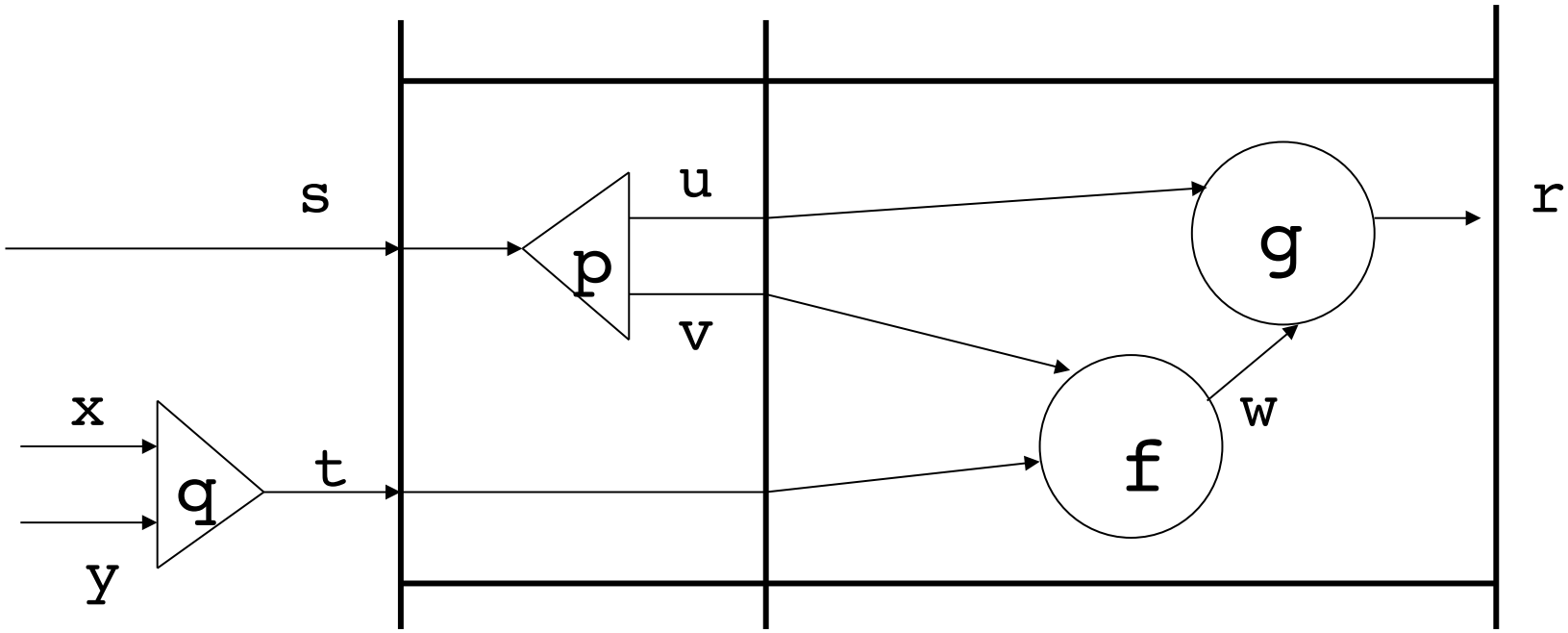
`h(s,t)->r, s=p(x,y)` ⇒

`u<-x, v<-y, f(v,t)->w, g(u,w)->r`



Assignment Absorption (1)

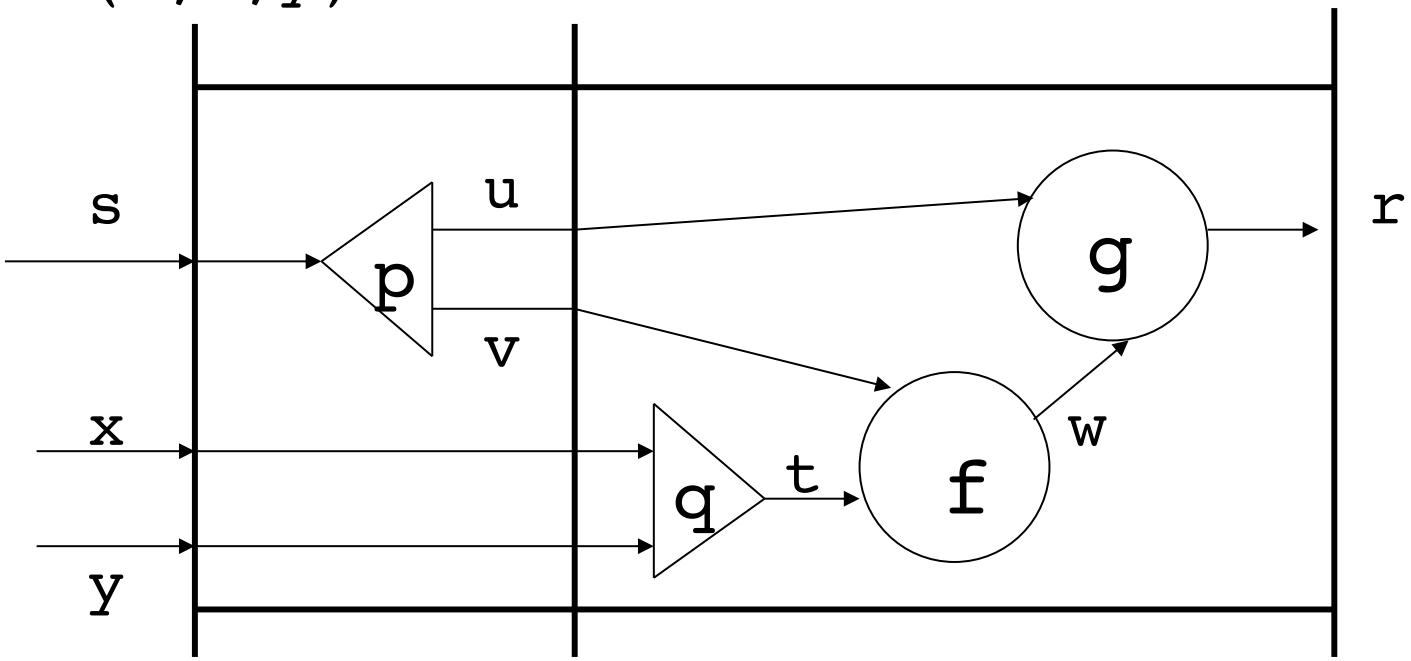
```
h(s,t)->r, t=q(x,y)
```



Assignment Absorption (1)

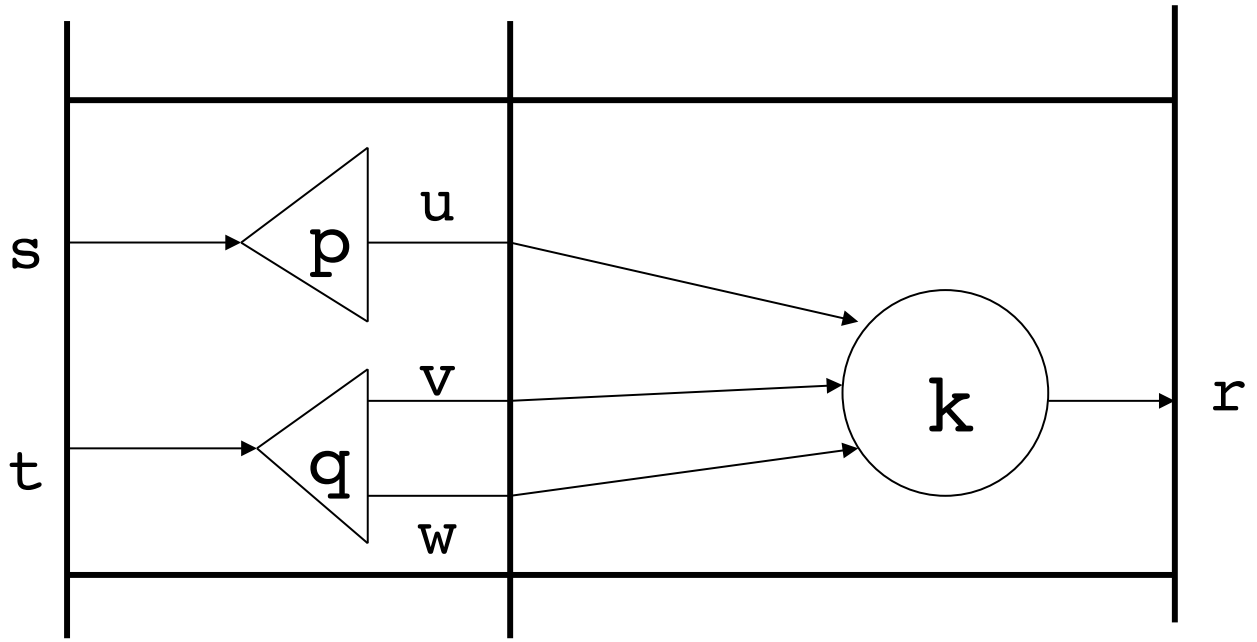
`h(s,t)->r, t=q(x,y)` ⇒

`h1(s,x,y)->r`



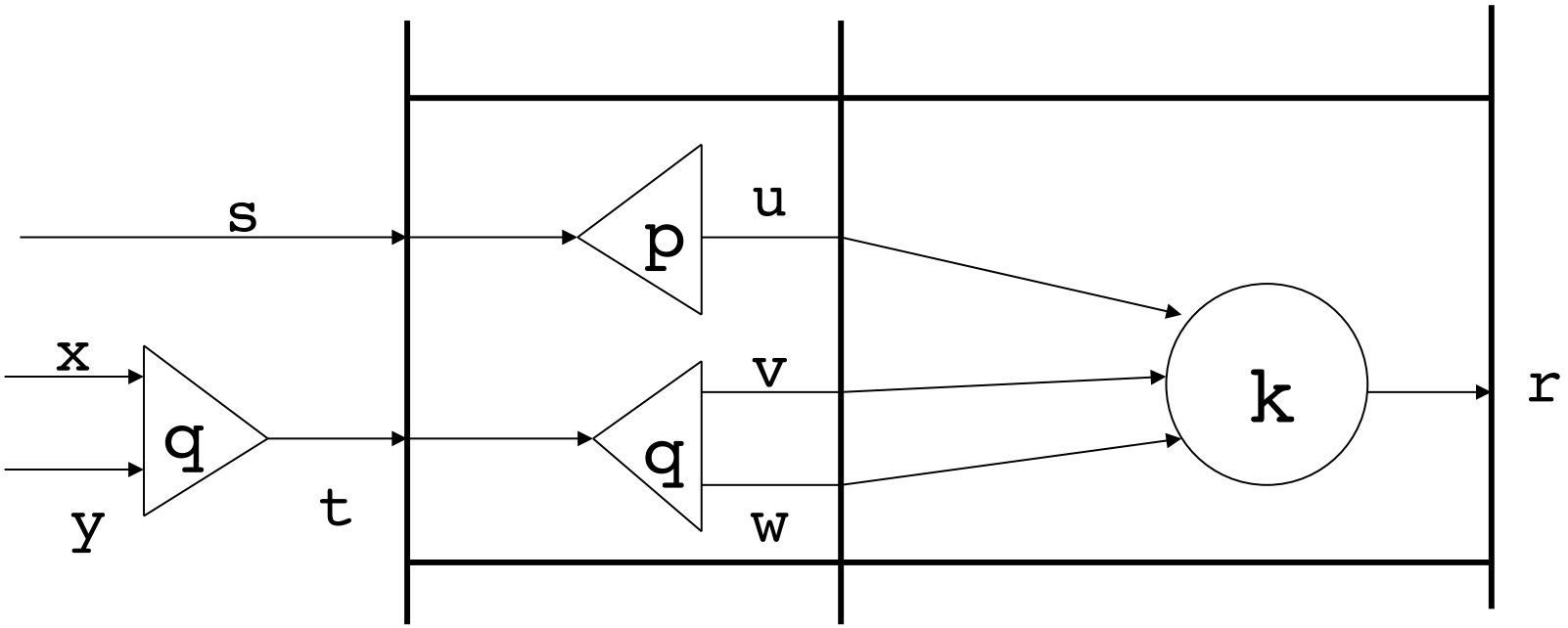
Rule with two independent matches

`s=p(u), t=q(v,w) ‖ k(u,v,w)—>r`



Assignment absorption (2)

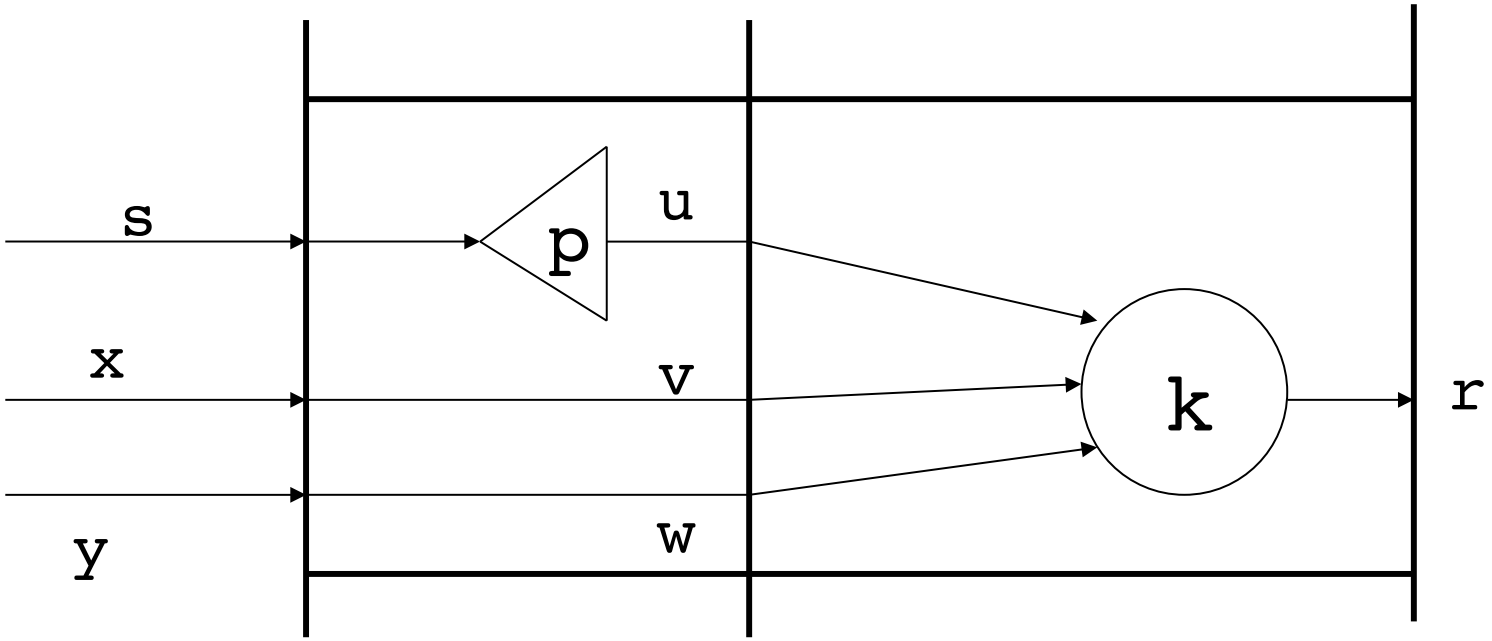
```
t=q(x,y),
h1(s,x,y)->r
{ … s=p(u), t=q(v,w) || k(u,v,w)->r … }
```



Assignment absorption (2)

```
h(s,t)—>r @ t=q(x,y) ⟹ h1(s,x,y)—>r

h2(s,x,y)—>r
{ … s=p(u) ‖ v<—x, w<—y, k(u,v,w)—>r … }
```

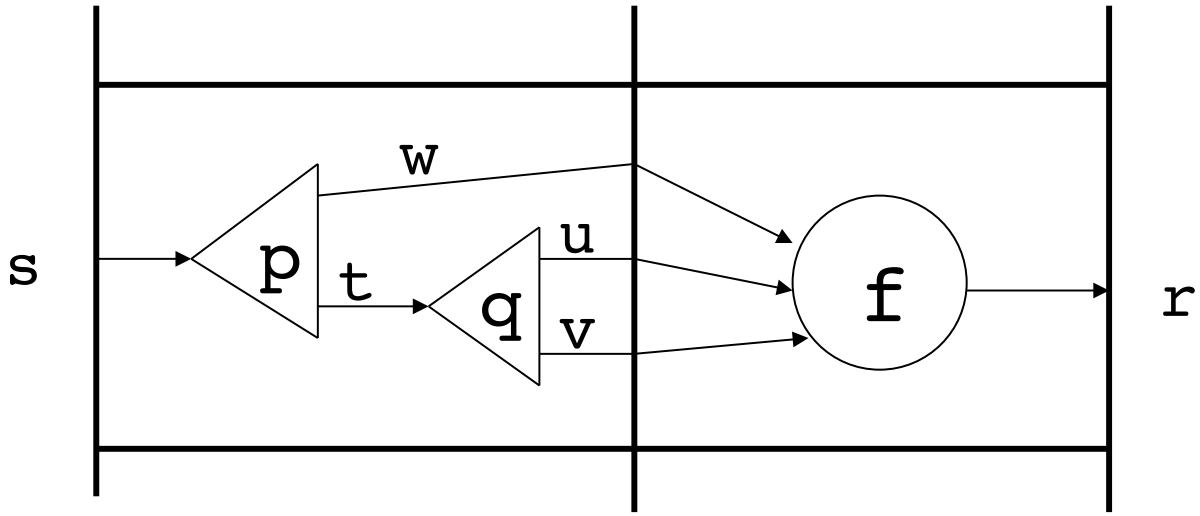


Assignment absorption (3)

- Rule eliminated from set if the tag or number of variables or mode of variables does not match

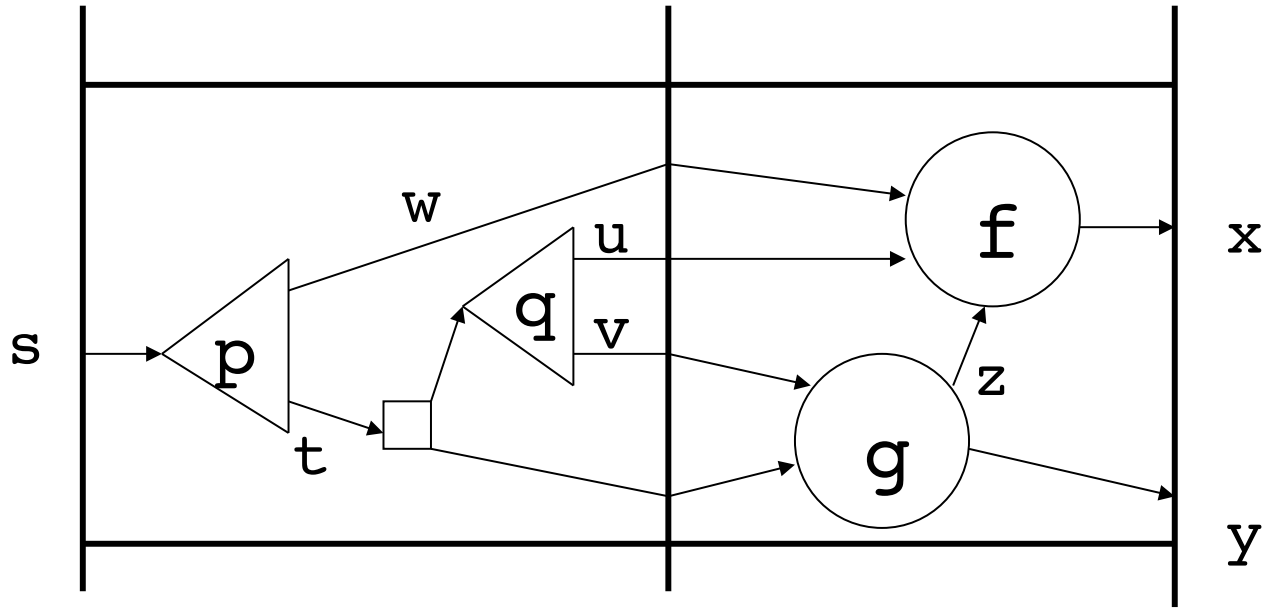
Rule with a match and a dependent match

```
s=p(w,t), t=q(u,v) || f(w,u,v)->r
```



Rule with a left hand side duplicator and a dependent match

`s=p(w,t), t=q(u,v) ‖ f(w,u,z)—>x, g(v,t)—>(z,y)`



Representing a mutable variable

```
mutvar(S)->V
{
 S=empty || V=empty;
 S=cons(M,S1), M=get(V1) ||
    merge(V1,V2)->V, mutvar(S1)->V2;
 S=cons(M,S1), M=set()->V1 ||
    V=empty, mutvar(S1)->V1
}
```

Syntactic Sugar

```
mutvar(S)->V
{
 S=[] || V=[];
 S=[get(V1)|S1] ||
    merge(V1,V2)->V, mutvar(S1)->V2;
 S=[set->V1|S1] ||
    V=[], mutvar(S1)->V1
}
```

More Syntactic Sugar

```
mutvar(S)->V
{
 S=[] || V=[];
 S.get(V1) |
    merge(V1,V)->V;
 S.set->V1 |
    V=[], V1<-V
}
```

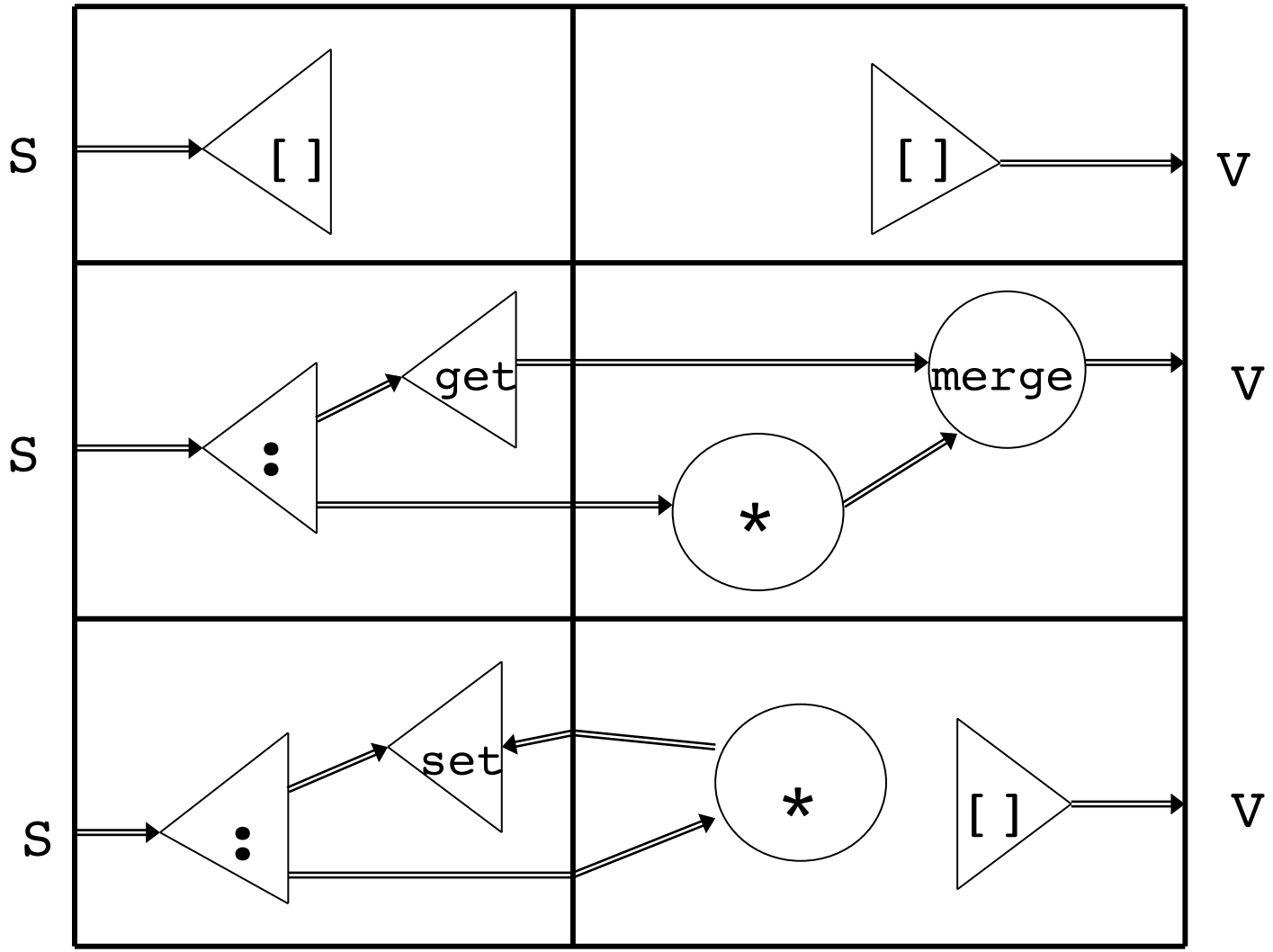
Representing a value

```
value(n,S)
{
 S=empty ||;
 S=cons(M,S1), M=get()—>v ||
      v<—n, value(S1,n);
}
```

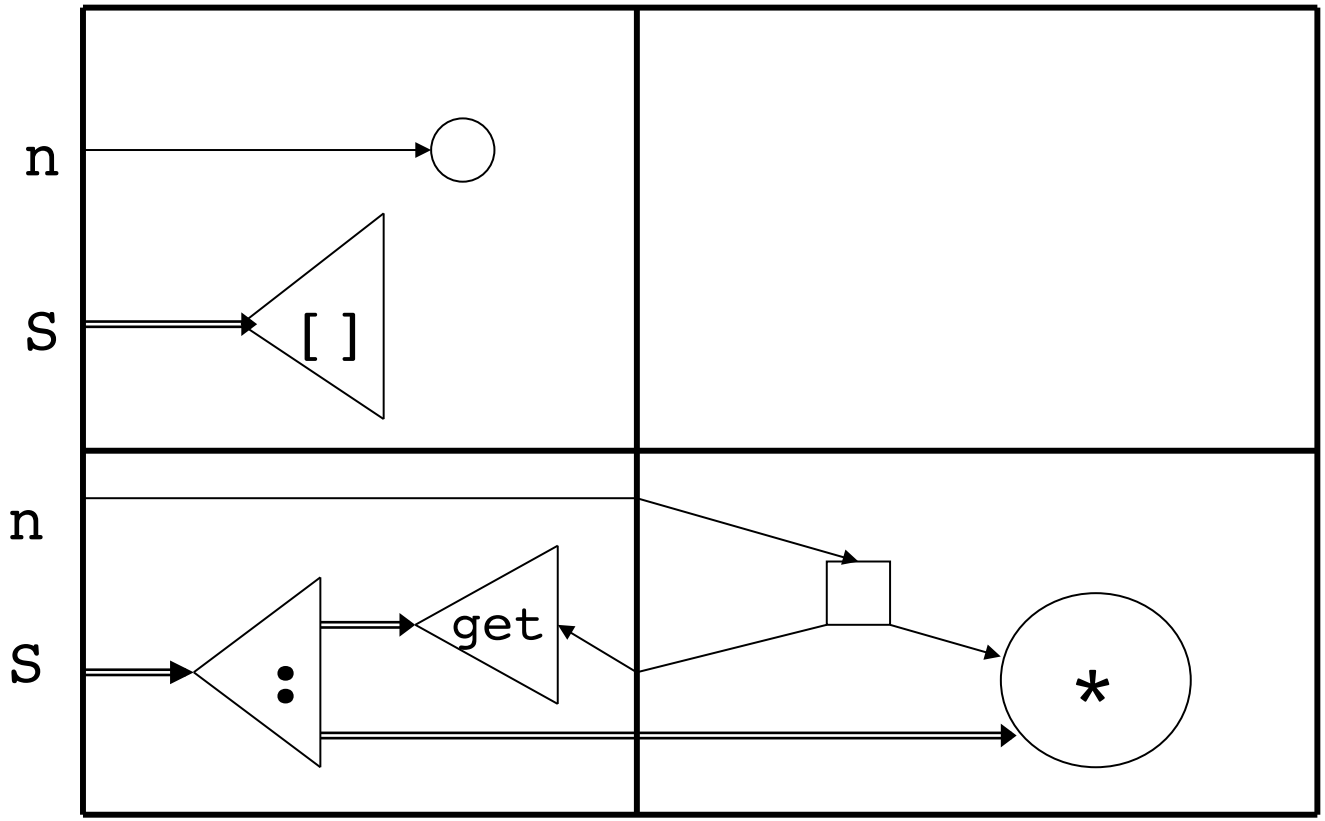
```
mutvar(S)—>V, value(n,V)
```

sets a new mutable variable to store the value `n`, it can proceed without `n` being assigned.

mutvar(S)->V



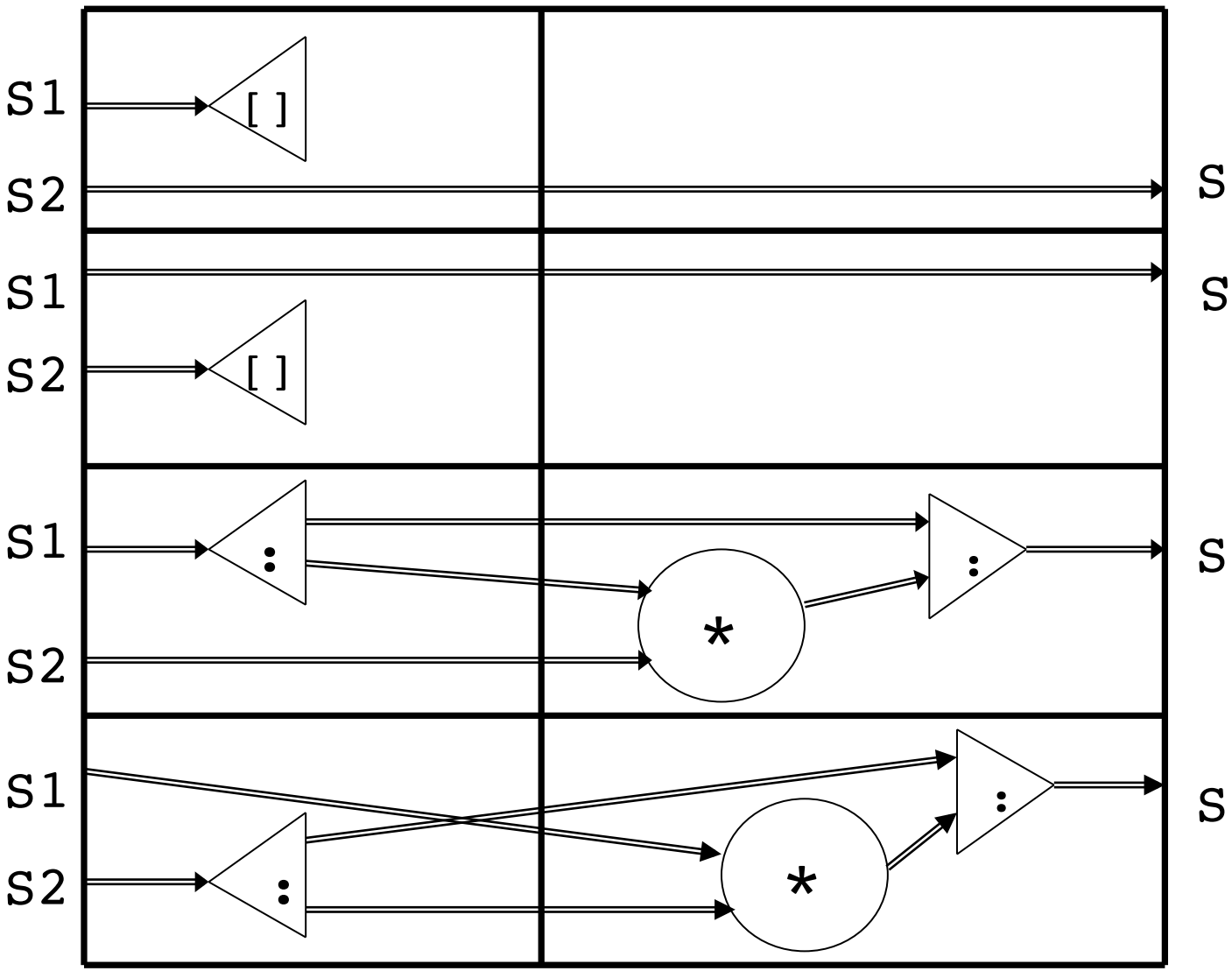
value(n,S)



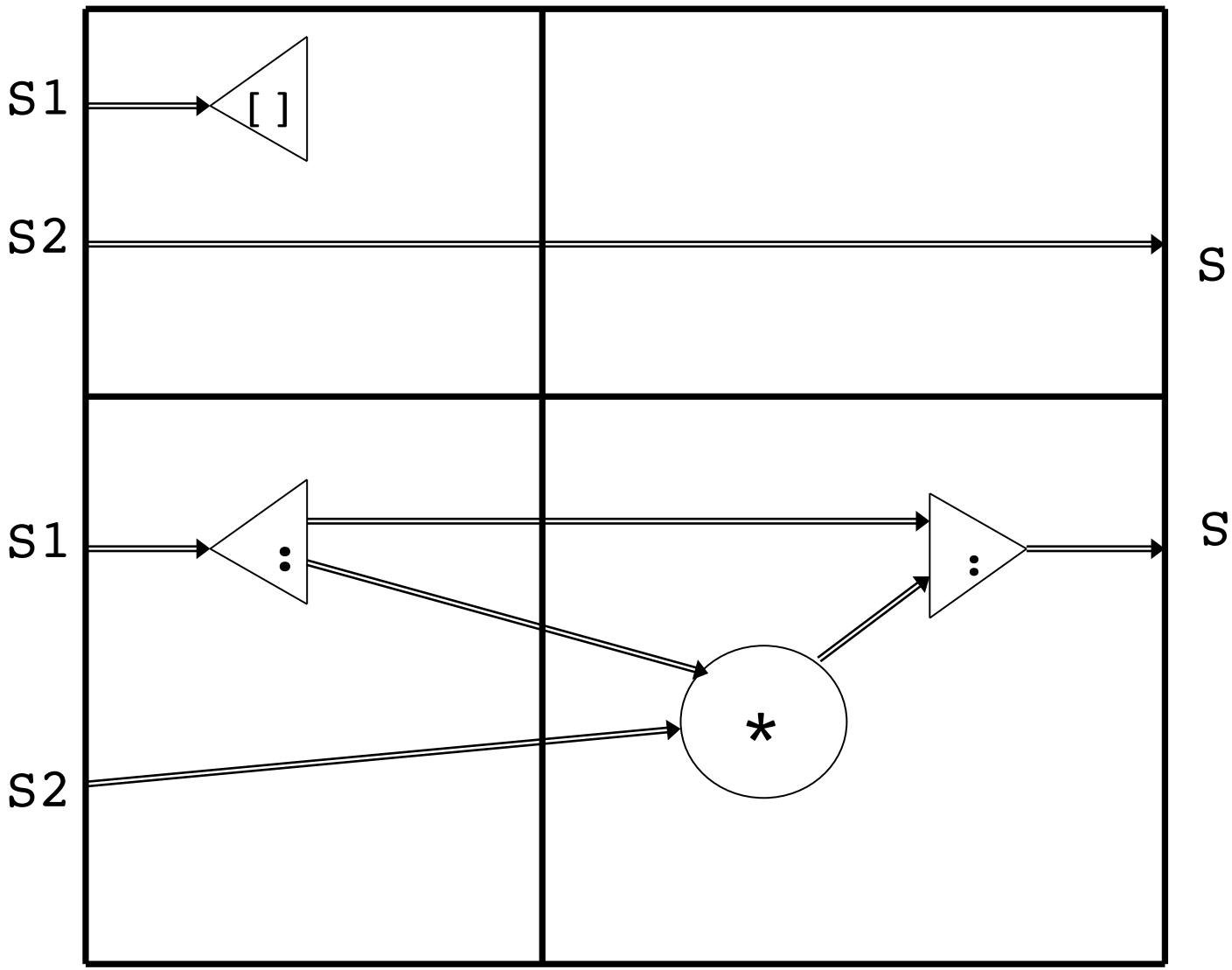
Nondeterminate Stream Merger

```
merge(S1,S2)->S
{
 S1=empty || S<-S2;
 S2=empty || S<-S1;
 S1=cons(M,S1t) ||
     S=cons(M,St), merge(S1t,S2)->St;
 S2=cons(M,S2t) ||
     S=cons(M,St), merge(S1,S2t)->St;
}
```

merge(S1,S2)->S



append(S1,S2)—>S



Representing a lockable variable

```
lockvar(S)->V
{
 S=empty || V=empty;
 S=cons(M,S1), M=get(V1) ||
    merge(V1,V2)->V, lockvar(S1)->V2;
 S=cons(M,S1), M=set()->V1 ||
    V=empty, lockvar(S1)->V1;
 S=cons(M,S1), M=lock(S2) ||
    append(S2,S1)->S3, lockvar(S3)->V;
}
```

Representing an Asynchronous Channel

```
achan(Ch)->(Puts,Gets)
{
 Ch=[] || Puts=[], Gets=[];
 Ch=[P|Ch1], P=()->V ||
     P1=()->V, Puts=[P1|Puts1], achan(Ch1)—>(Puts1,Gets);
 Ch=[G|Ch1], G=(V) ||
     G1=(V), Gets=[G1|Gets1], achan(Ch)—>(Puts,Gets1);
}

achan1(Puts,Gets)
{
 Puts=[P|Puts1], P=()—>V1, Gets=[G|Gets1], G=(V2) ||
   V1<—V2, achan1(Puts1,Gets1);
}
```

Using a mutable variable or asynchronous channel

`S=[()->V|S1]`

`S=[(V)|S1]`

use `S1` for variable/channel subsequently

`append(S1,S2)->S`

use `S1` and `S2` for variable/channel (sequential access)

`merge(S1,S2)->S`

use `S1` and `S2` for variable/channel (concurrent access)

Representing a λ-expression

With `exp(E)->(X,V)` where `E` represents the expression, `X` is the argument and `V` represents a free variable, λx.exp is represented by `F` where

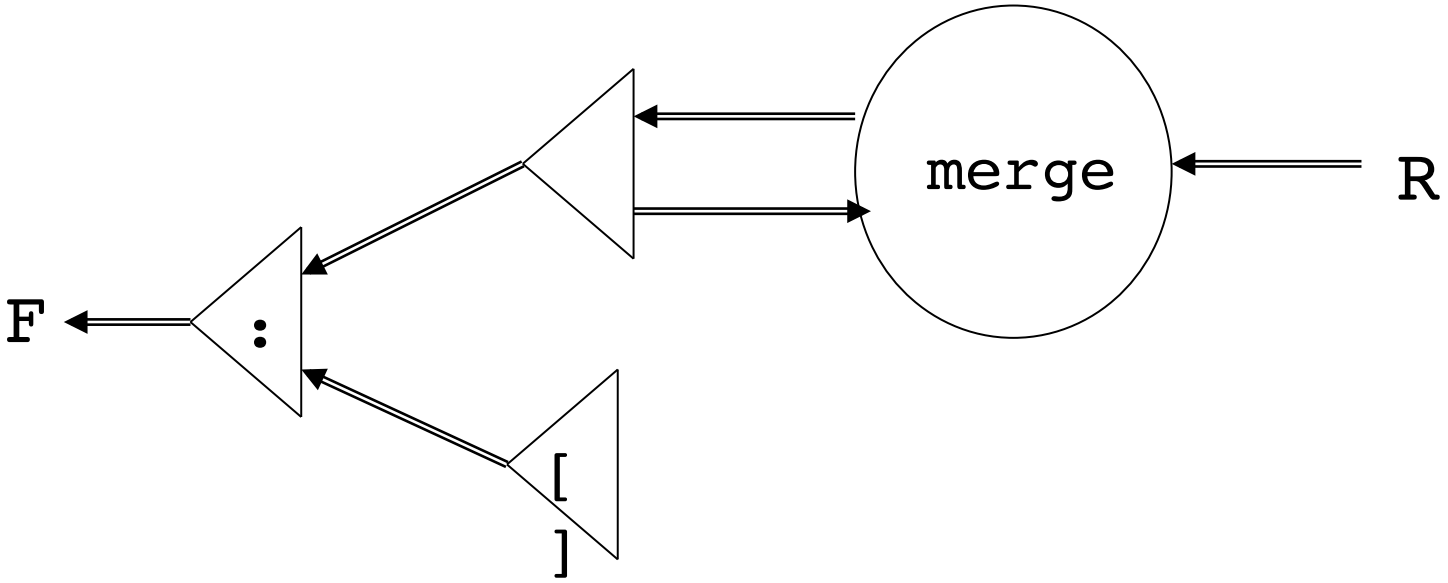
```
lambda(F)->V
{
 F=[] || V=[];
 F=[C|F1], C=(R)->X || exp(R)->(X,V1),
    lambda(F1)->V2, merge(V1,V2)->V
}
```

then (λx.exp) n is given by `R` in `F=[(R)->N|F1]` where `N` represents n, with `F1` used for λx.exp subsequently.

Y Combinator

If `F` represents a λ-expression f, then the expression Y f can be represented by `R` with:

```
F=[(R1)->F1], merge(F1,R)->R1
```



History: “Aldwych Core”

- “Aldwych” general purpose concurrent language, originally syntactic sugar for concurrent logic programming
- Identification of minimal subset of concurrent logic language needed for Aldwych
- Establishment of variable moding and linearity throughout
- Improved operational model making use of knowledge of modes and linearity

Current work

- Aldwych syntactic sugar incorporated features of major programming language paradigms
- Aldwych enabled a fine control of concurrent variations of language features
- So Aldwych Core can be used as an abstract model of concurrency giving an operational semantics
- Aldwych computes naturally with “holes”: variables not currently assigned, as they are being written to concurrently

Future work

- Computing with “holes” is a form of proof: finding properties of programs before data is supplied
- Further use of partial evaluation techniques provides deeper proofs
- Aldwych computation is naturally in terms of an agent interacting with “the world”
- This has similarities with the “game semantics” insight
- Can partial evaluation of Aldwych be taken to the point of providing full denotational semantics?