Now you C the point!: Making sense of pointers in C

Background: Why are C pointers hard to understand?

Many of us have struggled to understand pointers in C. This can be attributed to at least two different reasons. First, the syntax of C, specially when declaring pointers is awkward and unintuitive. Second, most of us lack a robust mental model of C's semantics that represents our understanding of how C manipulates pointers.

The unintuitive syntax of C

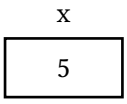
C insists that we write `int x` instead of the more intuitive `x int`. It gets worse when we have pointers. So, `int *p` looks strange when compared to `p *int`. One way to think of the type declaration for `p` is to consider navigating from `p` on a `*` and end up at `int`. This idea of navigation is central to the model we present but C's syntax doesn't quite align with this model of thinking if we are used to reading things from left to right (which is the case with many Indo-European languages, but not scripts for Arabic, Persian, Urdu and Hebrew).

Traditional Box and pointer models of C

Before we introduce the new model, let us consider one of the most common mental models employed by students of C programming. It is called the *box and pointer* model. In this model, boxes are memory locations and the boxes contain values.

Example 1: Conflating a variable with its address

Consider the the C statement `int x = 5` is represented as the box diagram



Notice that the box (an address) is itself labeled `x`. This results in conflating `x` with its address, so, there is no way to distinguish `x` from `&x`, namely the address of `x`. So, `printf("%p", &x);` will print a value that is neither `x` nor `5`.

Example 2: Reasoning with equality violated

Here is another example: Now consider the C fragment

```
int x = 5;
int y = 5;
```

. Clearly, `x` and `y` both now denote the value 5. We write this as $x \underset{c}{=} y$. Now think of `&` as an operator, that takes a value and returns its address. So `&x` returns the address of `x`. So we have $x \underset{c}{=} y$ but $\&x \underset{c}{\neq} \&y$. This is counter-intuitive to logical reasoning because it violates the priniciple of substitution: when $e_1 = e_2$, then if we any expression containing e_1, replacing e_1 with e_2 should make no difference.

Path Model

The alternative mental model we propose is motivated by the need to be able to do simple mathematical reasoning involving function application and mathematical equality. Our mental models are now represented as labelled directed graphs. A graph is simply a collection of vertices and an edge relation between vertices. In addition, each edge is labelled. Reasoning corresponds to traversing paths in the graph.

For the sake of simplicity, let us assume we only have `int` the primitive type.

1. There are three kinds of vertices: variable vertices, addresses vertices and value vertices.
2. Values vertices are integer value vertices or addresses value vertices. There could be multiple vertices with the same value.
3. Uninitialized values are denoted by $\perp$.
4. There is an arrow labelled & between a variable node and its address.
5. There is an arrow labelled * between the address and its value.

The *-labelled arrow captures the relation that an address *stores* a value.

6. There is a back arrow labelled & between the value and its address. A &-labelled arrow captures the relation that a value is stored in an address.
7. A value may occur multiple times stored at different addresses.
8. There is a derived edge labelled `r` which is the composition of the & and the * edges.
9. The labels on arrows may be thought of as maps that take the (tail) of the arrow to its head.

The memory graph evolves as the C program statements execute.

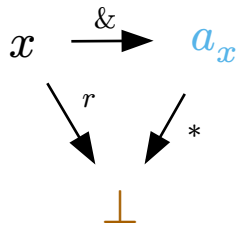
The best way to understand the path model is through examples

Example 3: Executing a simple fragment of code

Consider the program fragment.

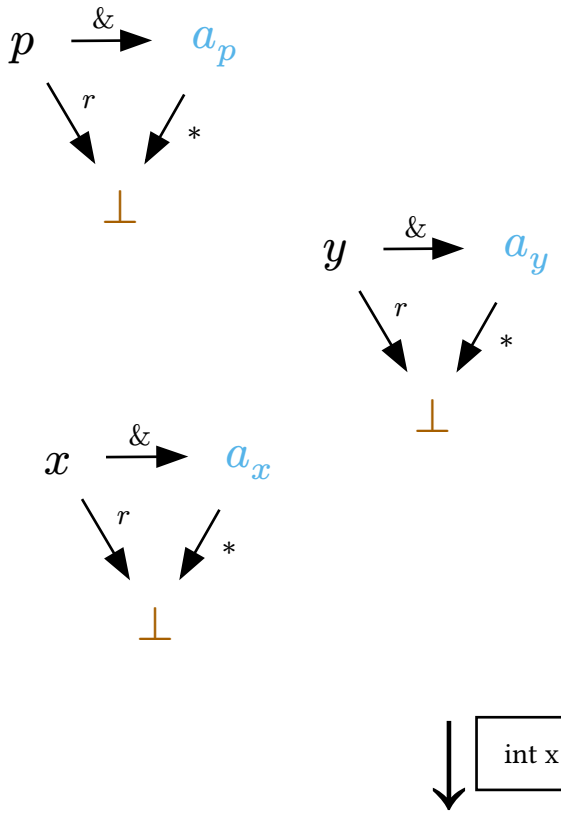
```
int x; x=5
```

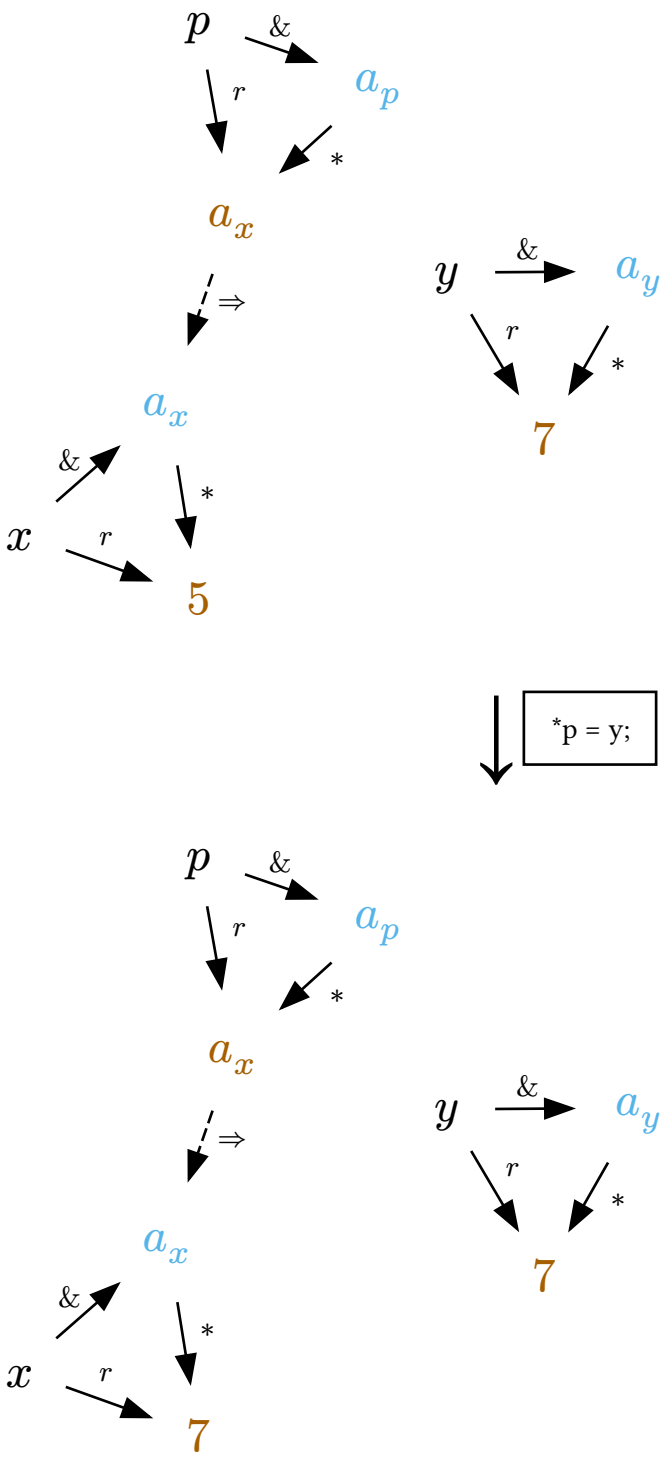
The initial graph has three nodes: the variable x, its address a_x and the (initial) value bottom.



Example 4

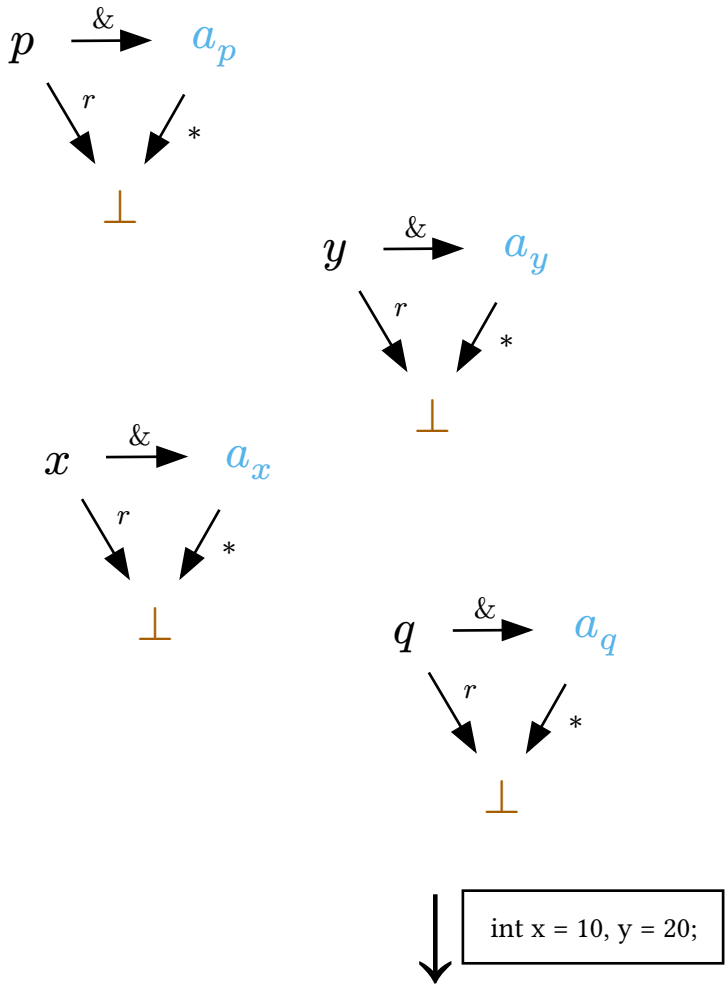
```
int x = 5, y = 7;
int *p = &x;
*p = y;
```

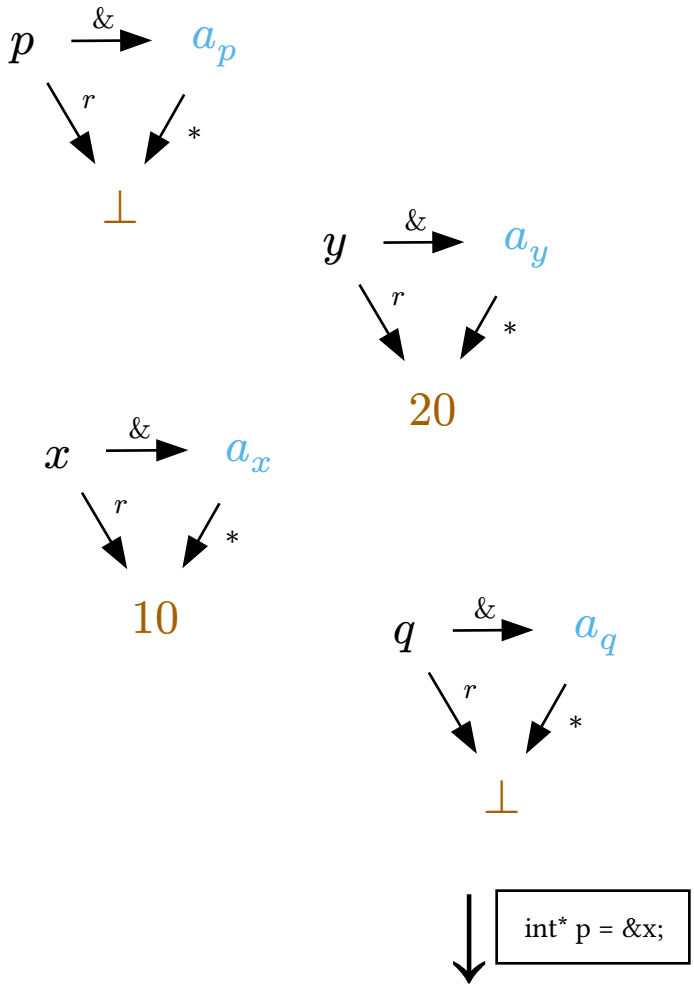

p
&
a_p
r
*
⊥
y
&
a_y
r
*
⊥
x
&
a_x
r
*
⊥
int x = 5, y = 7;
p
&
a_p
r
*
⊥
y
&
a_y
r
*
7
x
&
a_x
r
*
5
int *p = &x;

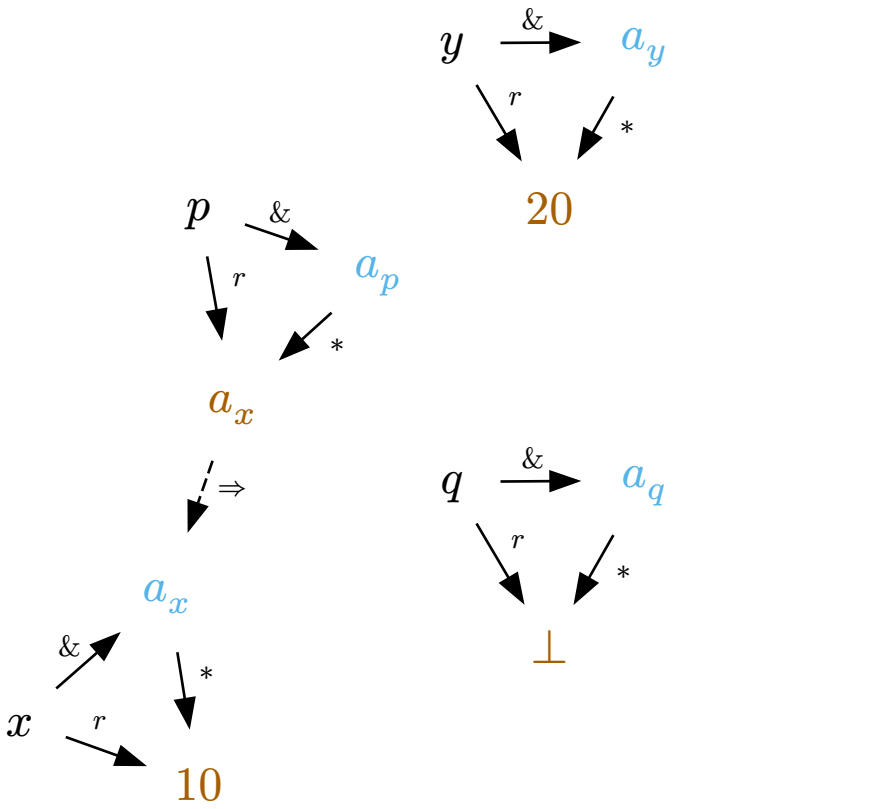


Example 5

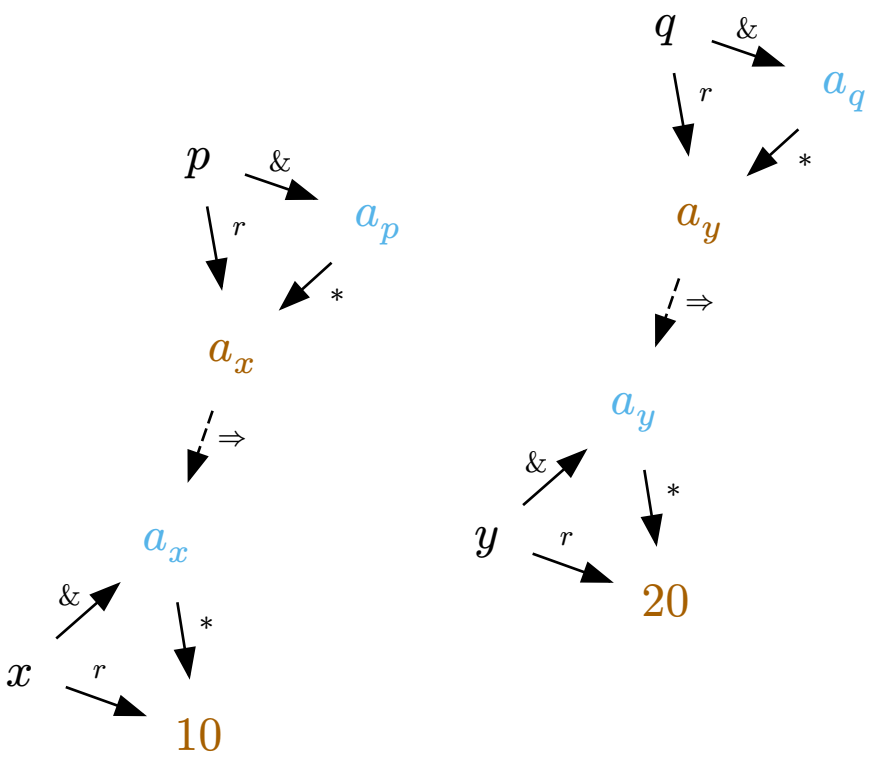
```
int x = 10, y = 20;
int* p = &x;
int* q = &y;
*p = *q;
q = p;
*q = 50;
```

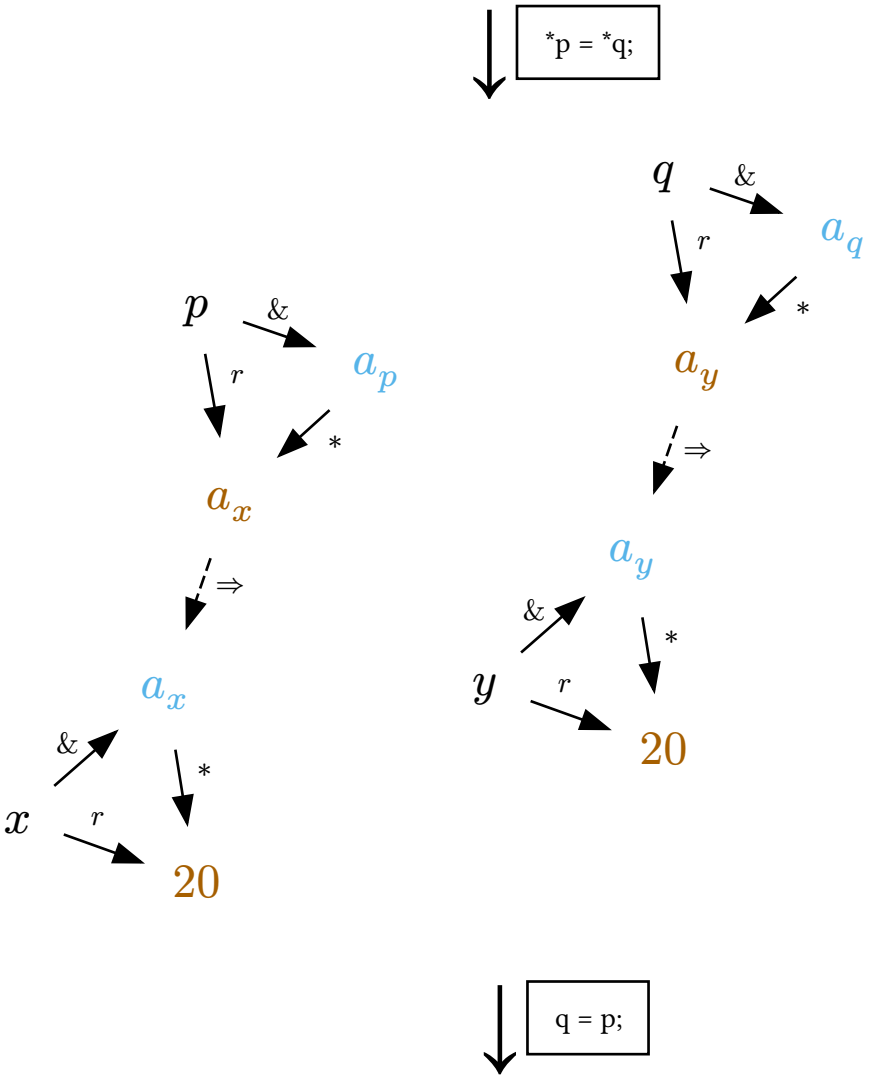

p
&
a_p
r
*
⊥
y
&
a_y
r
*
⊥
x
&
a_x
r
*
⊥
q
&
a_q
r
*
⊥
int x = 10, y = 20;

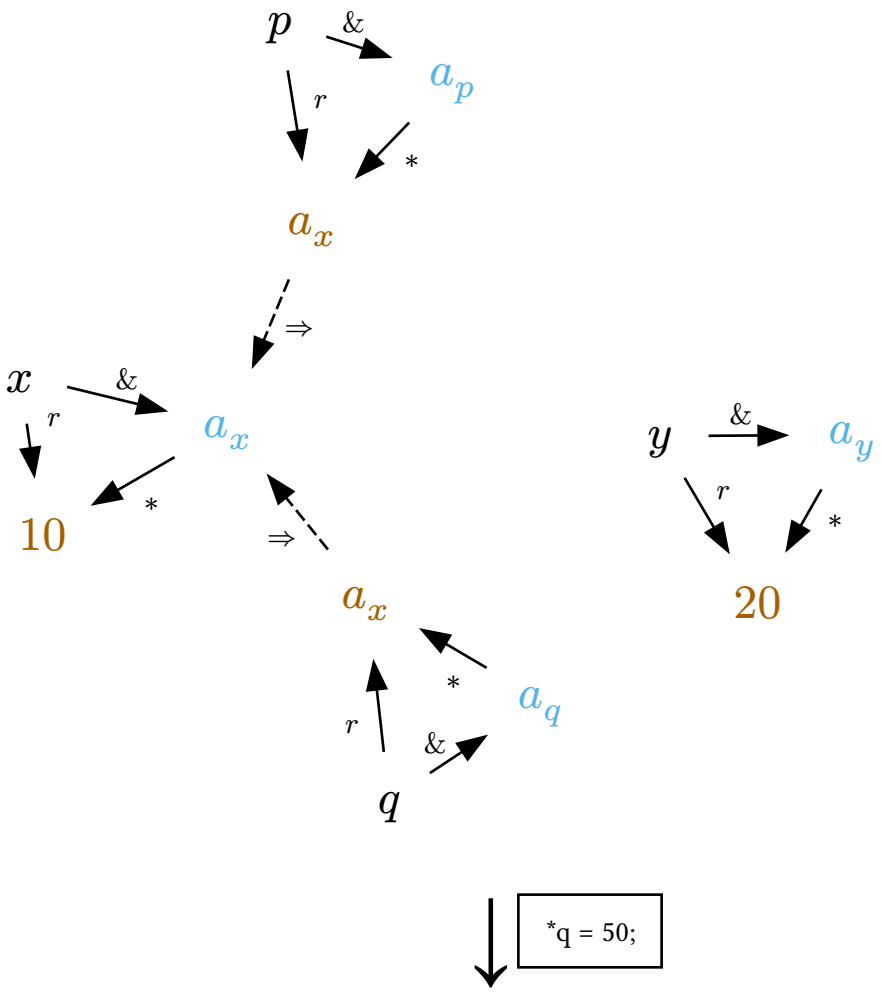

p
&
a_p
r
*
⊥
y
&
a_y
r
*
20
x
&
a_x
r
*
10
q
&
a_q
r
*
⊥
int* p = &x;

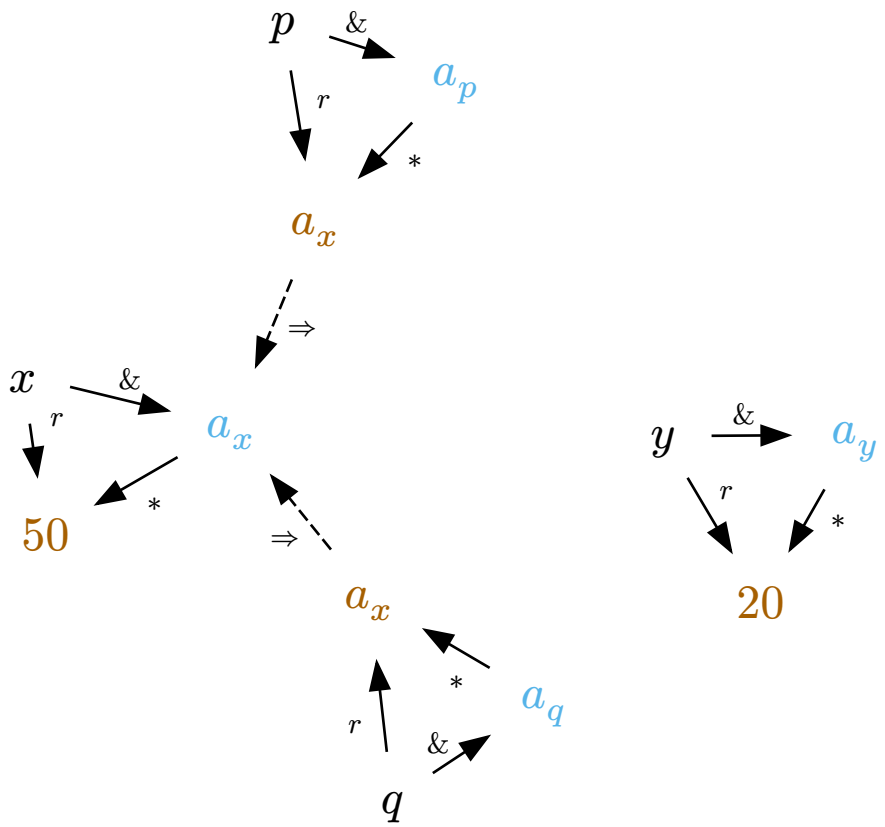


int* q = &y;



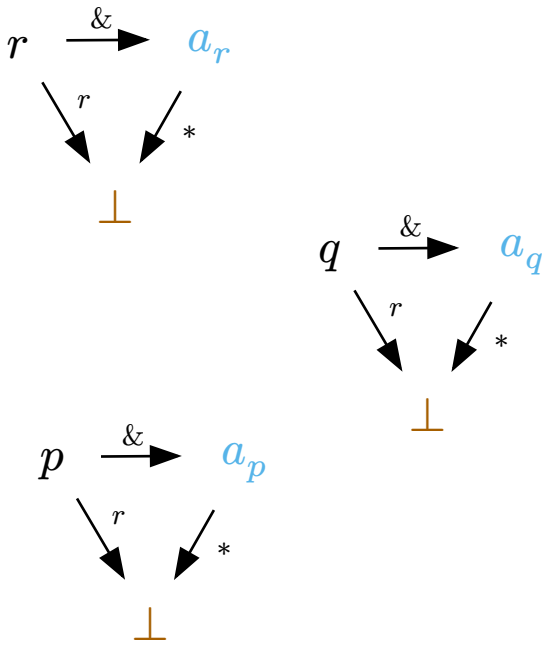

*p = *q;
p
&
r
a_p
*
a_x
⇒
a_x
&
*
x
r
20
q
&
r
a_q
*
a_y
⇒
a_y
&
*
y
r
20
q = p;

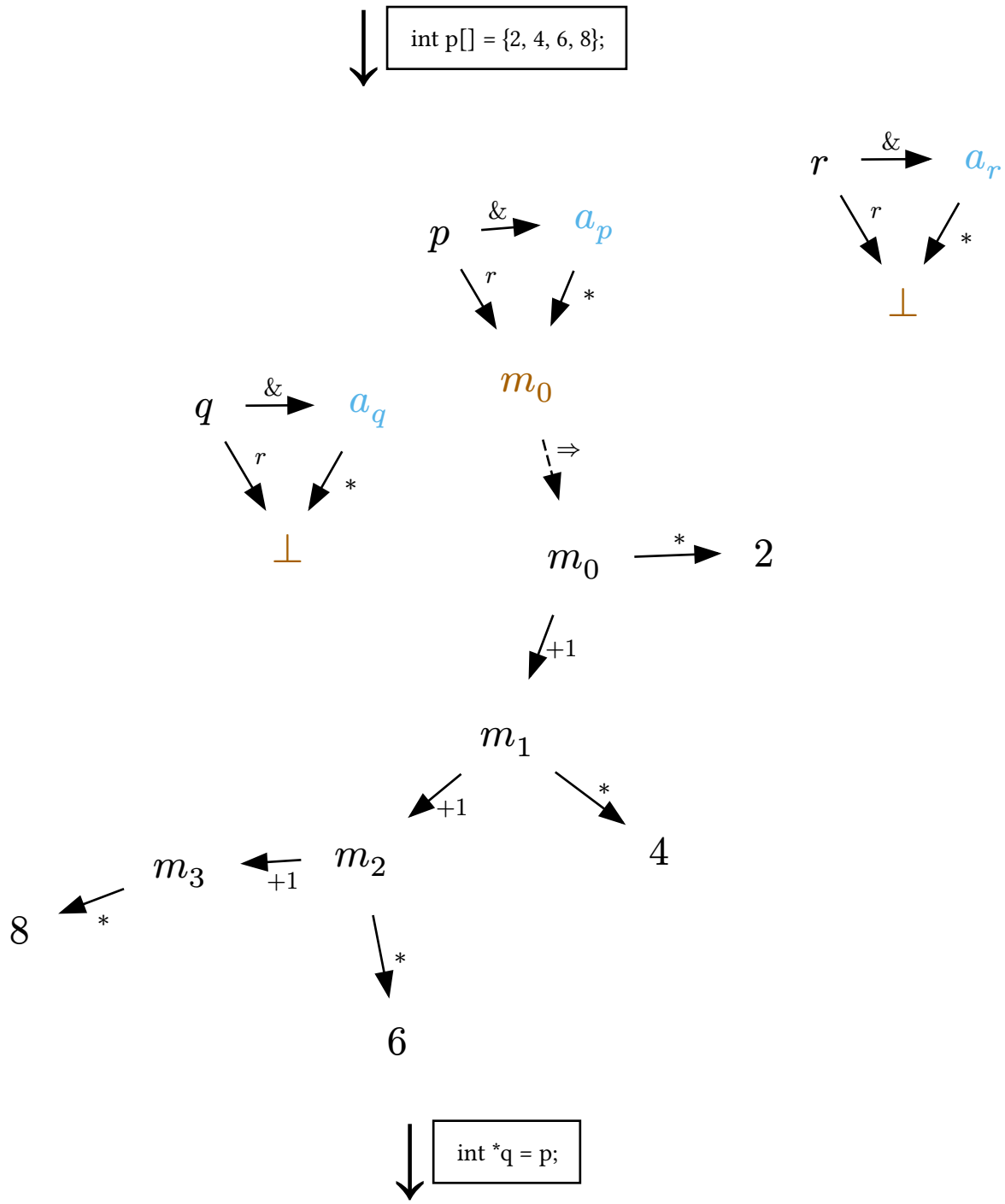

p
&
a_p
r
*
a_x
⇒
x
&
r
a_x
*
10
⇒
y
&
a_y
r
*
20
a_x
*
a_q
r
&
q
*q = 50;

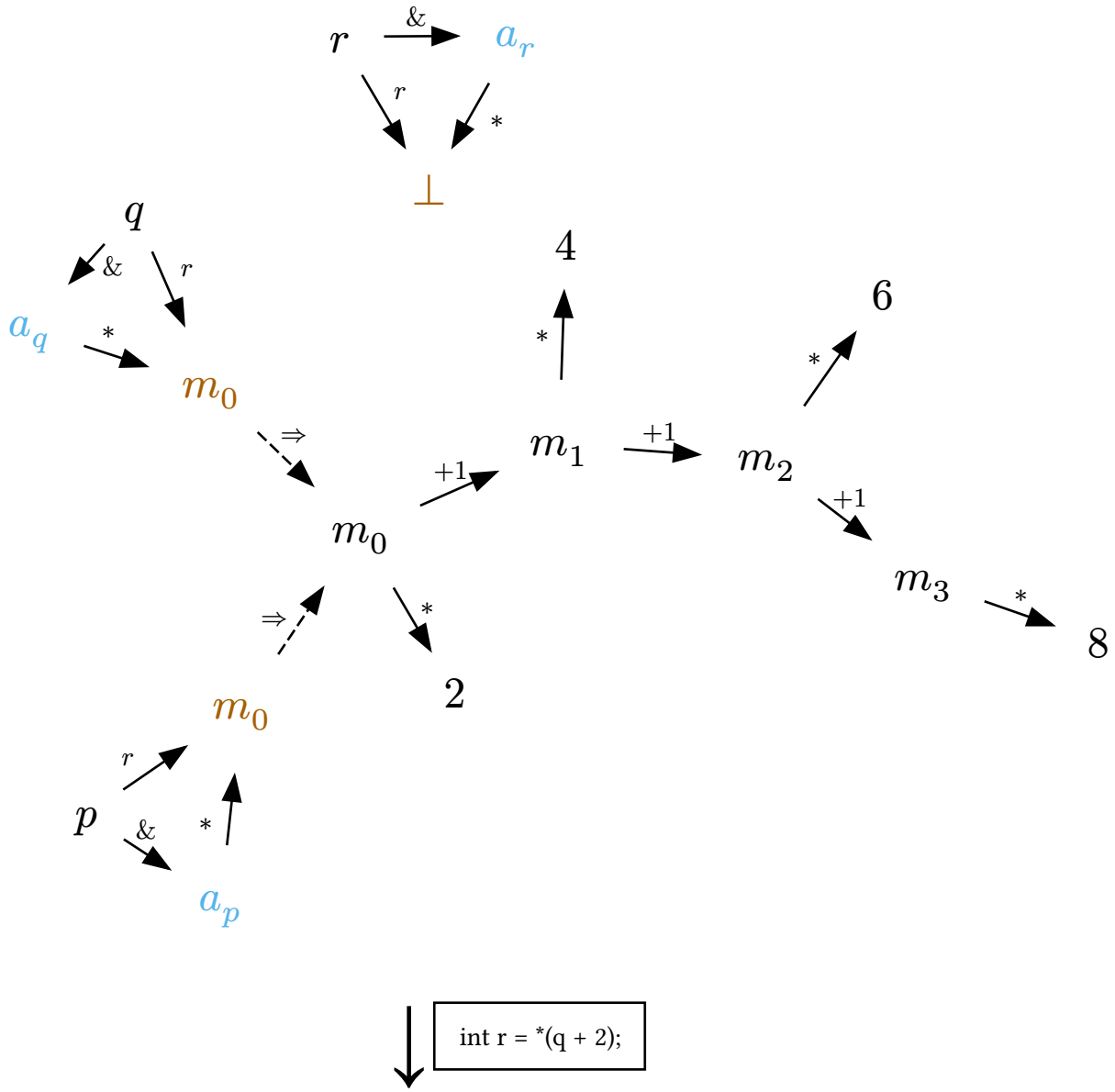


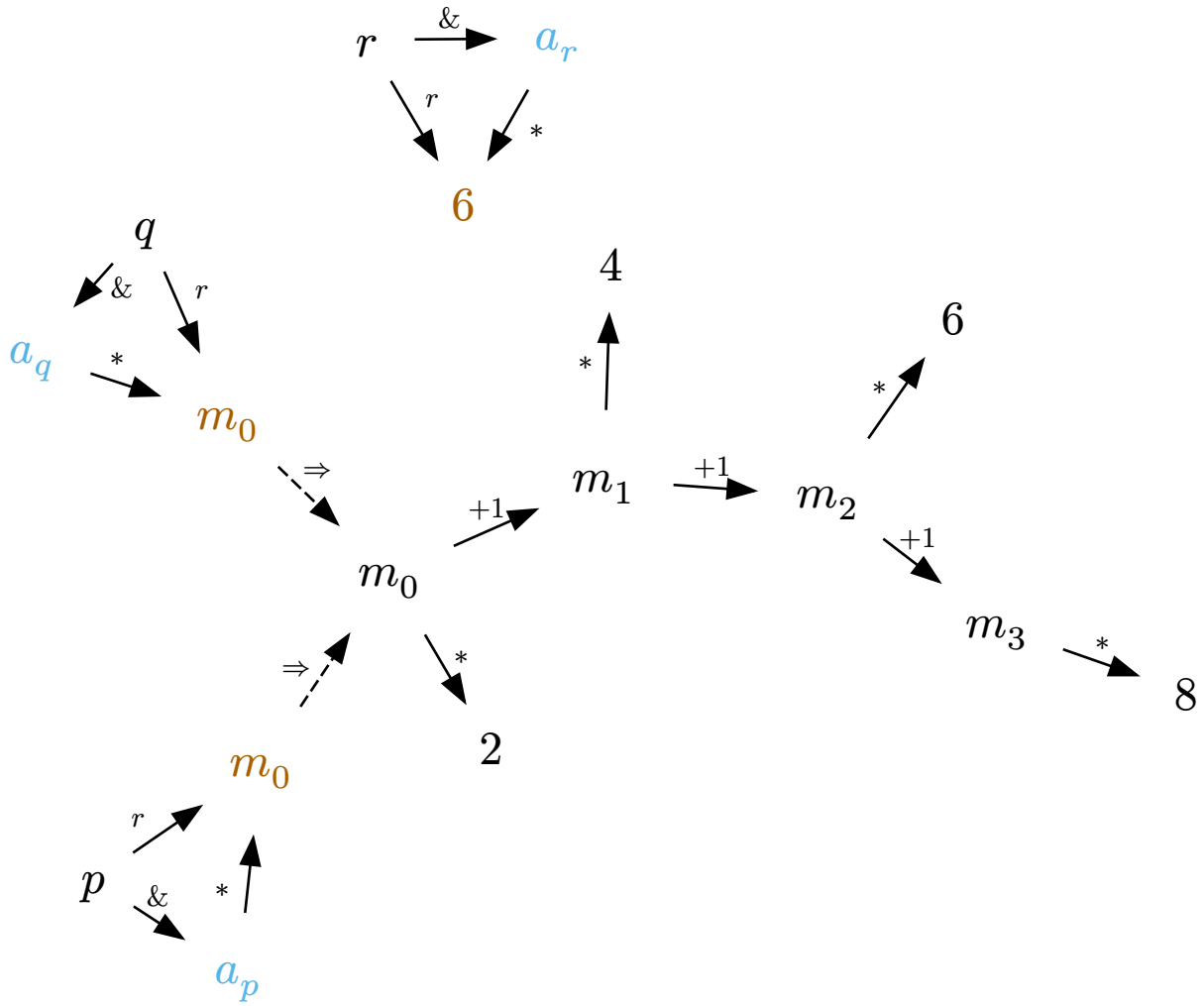
Example 6

```
int p[] = {2, 4, 6, 8};
int *q = p;
int r = *(q + 2);
```



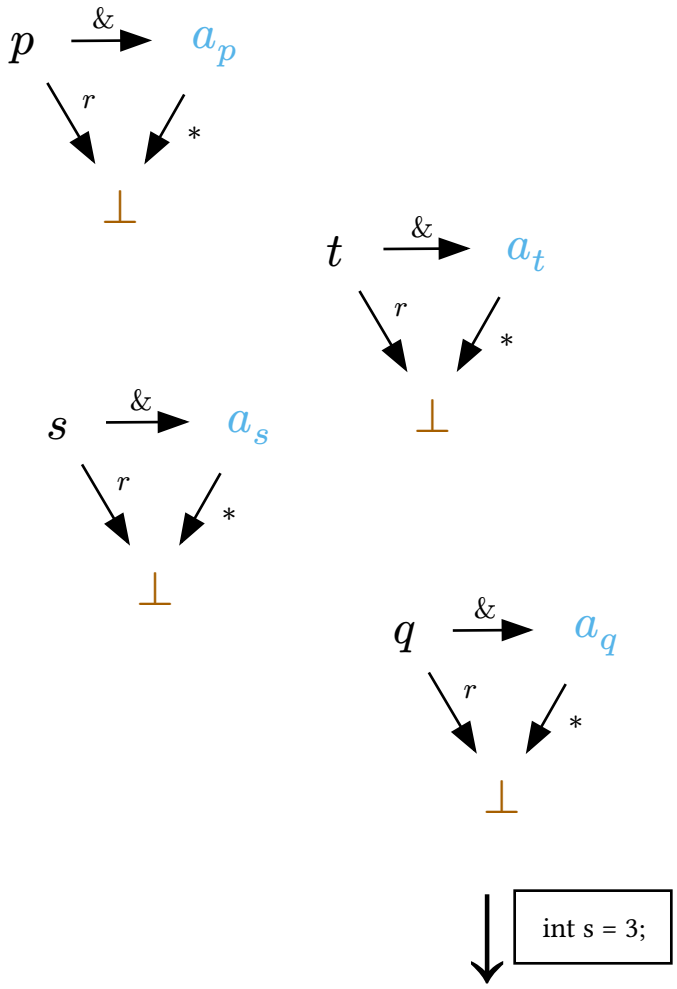

int p[] = {2, 4, 6, 8};
r
&
a_r
r
*
⊥
p
&
a_p
r
*
m_0
q
&
a_q
r
*
⊥
⇒
m_0
*
2
+1
m_1
+1
*
4
m_3
+1
m_2
8
*
*
6
int *q = p;

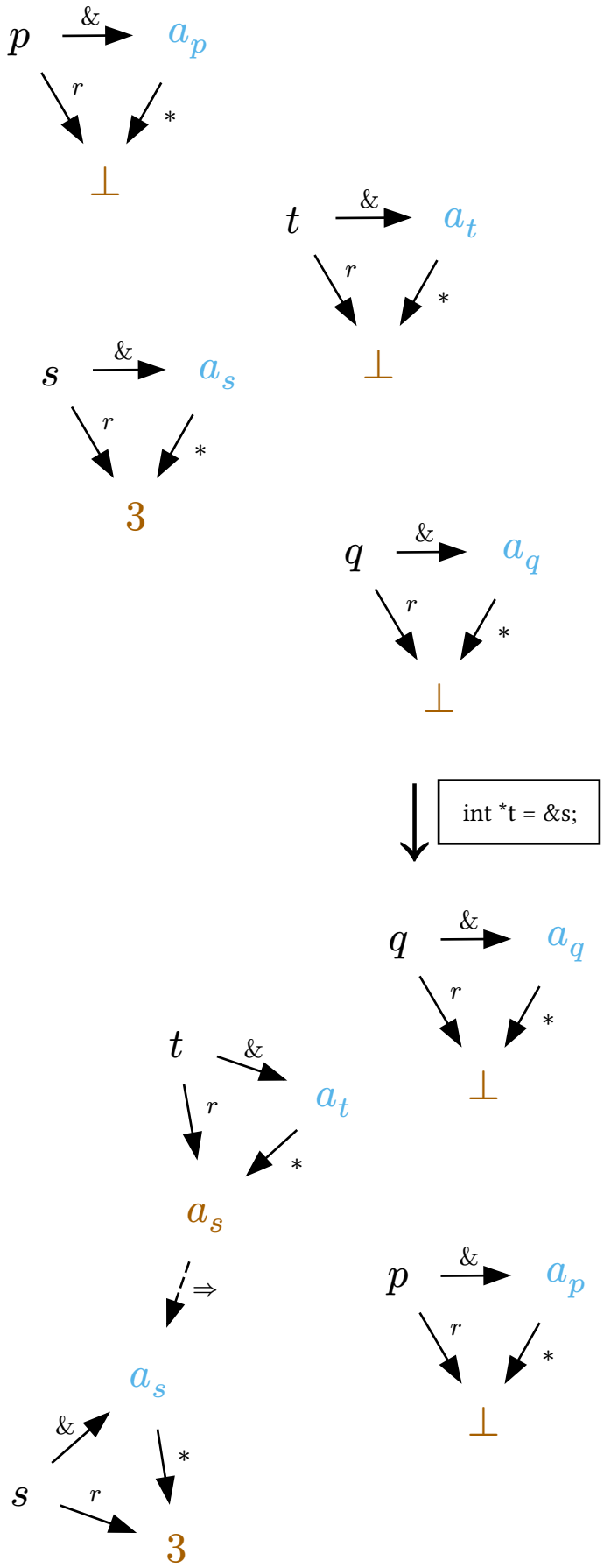

r
&
a_r
r
*
⊥
q
&
r
a_q
*
m_0
⇒
+1
m_0
⇒
*
2
m_0
r
p
&
*
a_p
4
*
m_1
+1
m_2
6
*
+1
m_3
*
8
int r = *(q + 2);

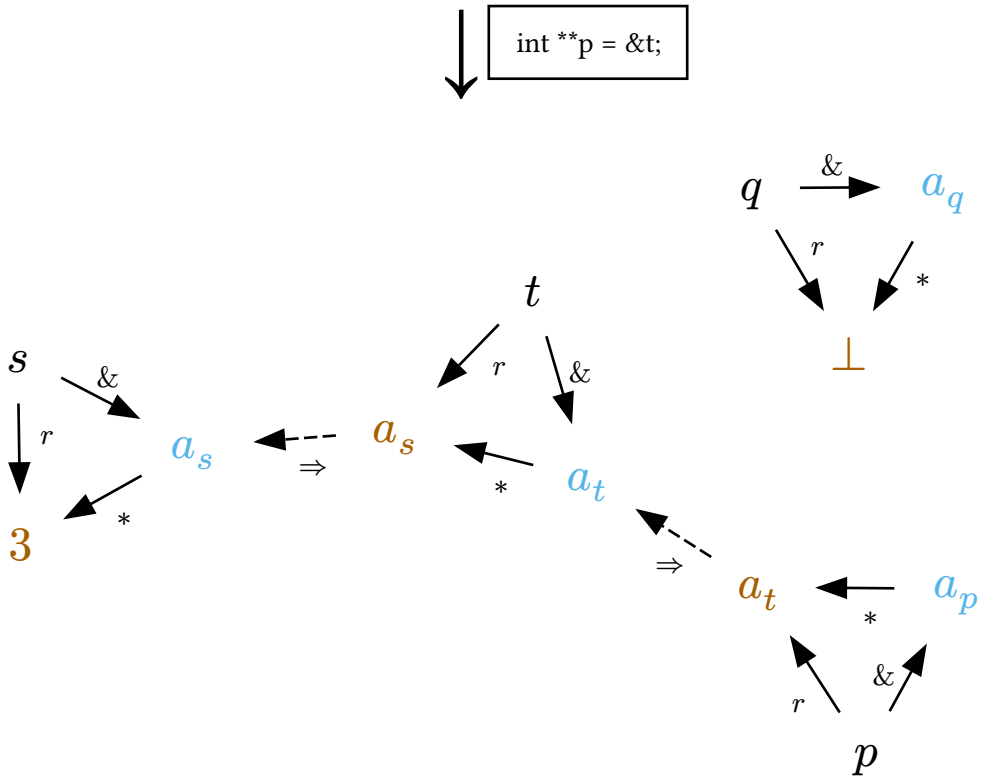


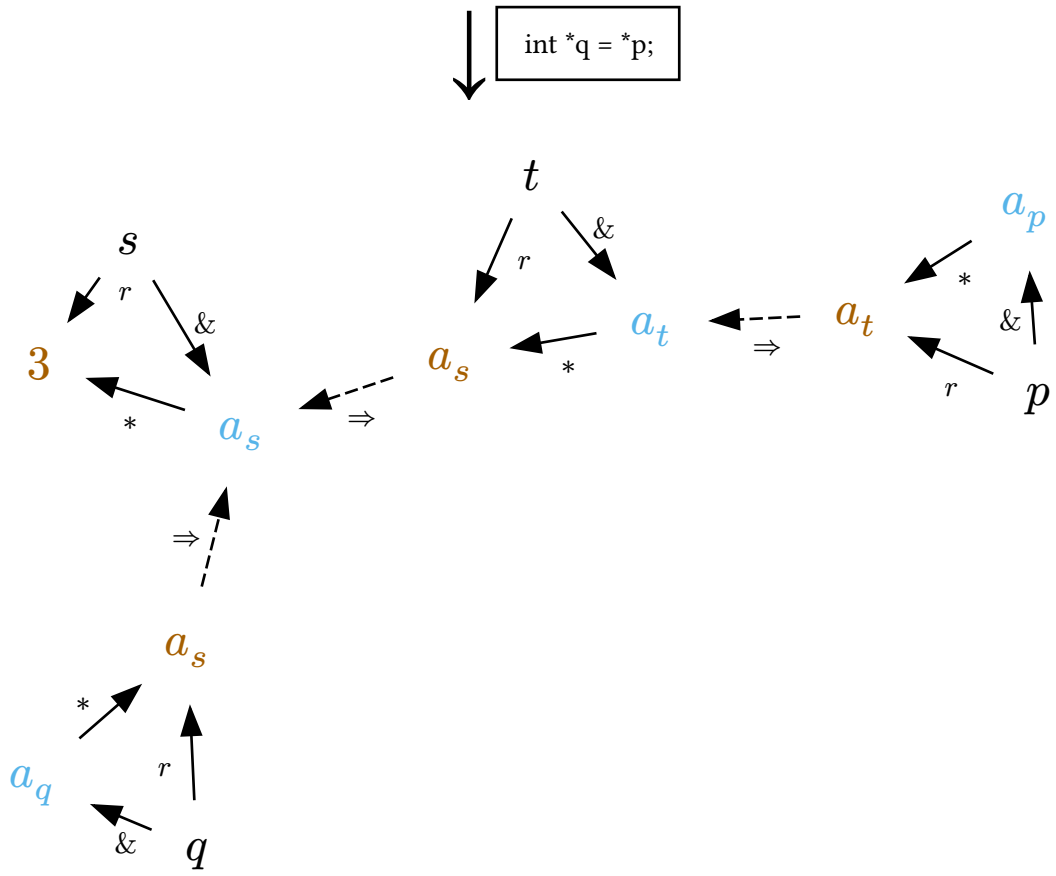
Example 7

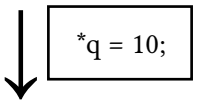
```
int s = 3;
int *t = &s;
int **p = &t;
int *q = *p;
*q = 10;
```

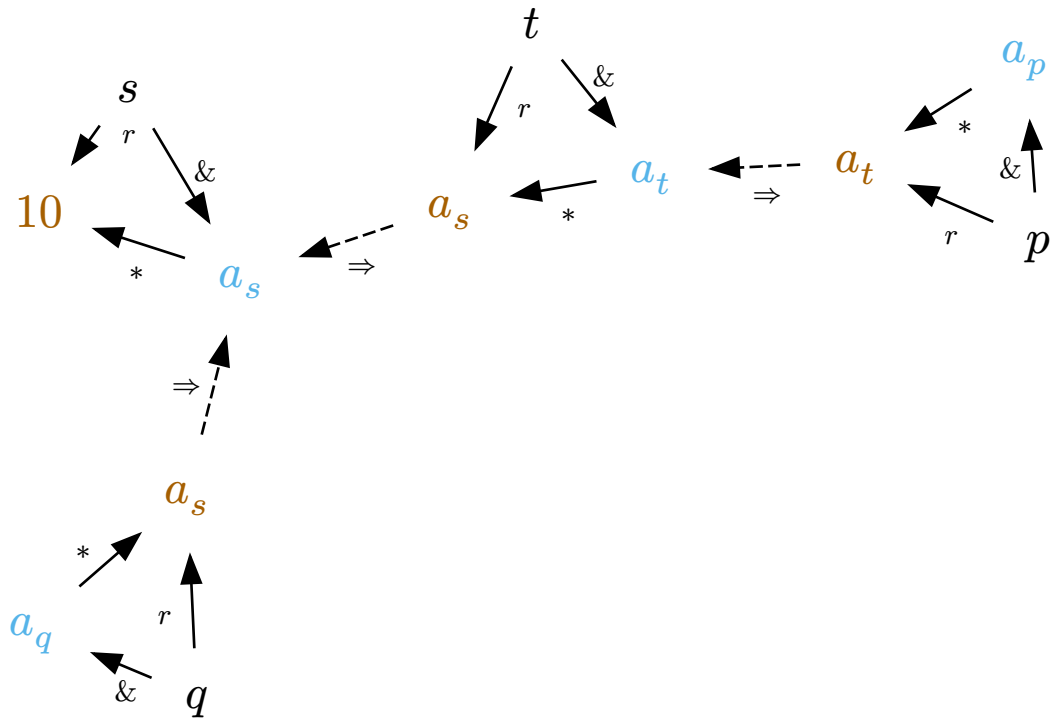

p
&
a_p
r
*
⊥
t
&
a_t
r
*
⊥
s
&
a_s
r
*
⊥
q
&
a_q
r
*
⊥
int s = 3;


p
&
a_p
r
*
⊥
t
&
a_t
r
*
⊥
s
&
a_s
r
*
3
q
&
a_q
r
*
⊥
int *t = &s;
q
&
a_q
r
*
⊥
t
&
r
a_t
*
a_s
⇒
a_s
&
*
s
r
3
p
&
a_p
r
*
⊥


int **p = &t;
q
&
a_q
r
*
⊥
t
r
&
s
&
r
a_s
a_s
⇒
*
a_t
3
*
⇒
a_t
a_p
*
r
&
p

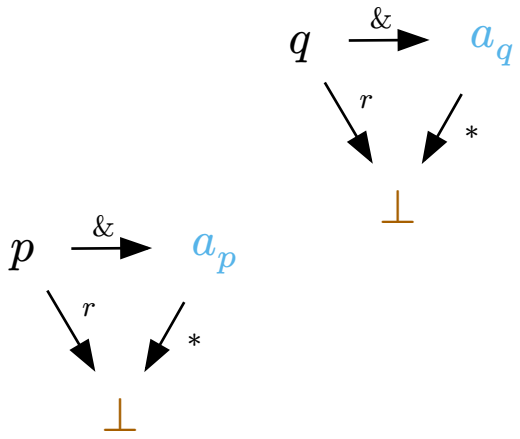

int *q = *p;
t
&
r
a_p
s
r
&
a_t
*
a_t
*
&
3
*
a_s
⇒
r
p
⇒
a_s
*
r
a_q
&
q

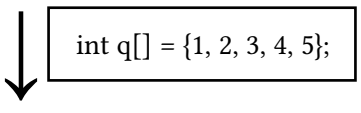

*q = 10;

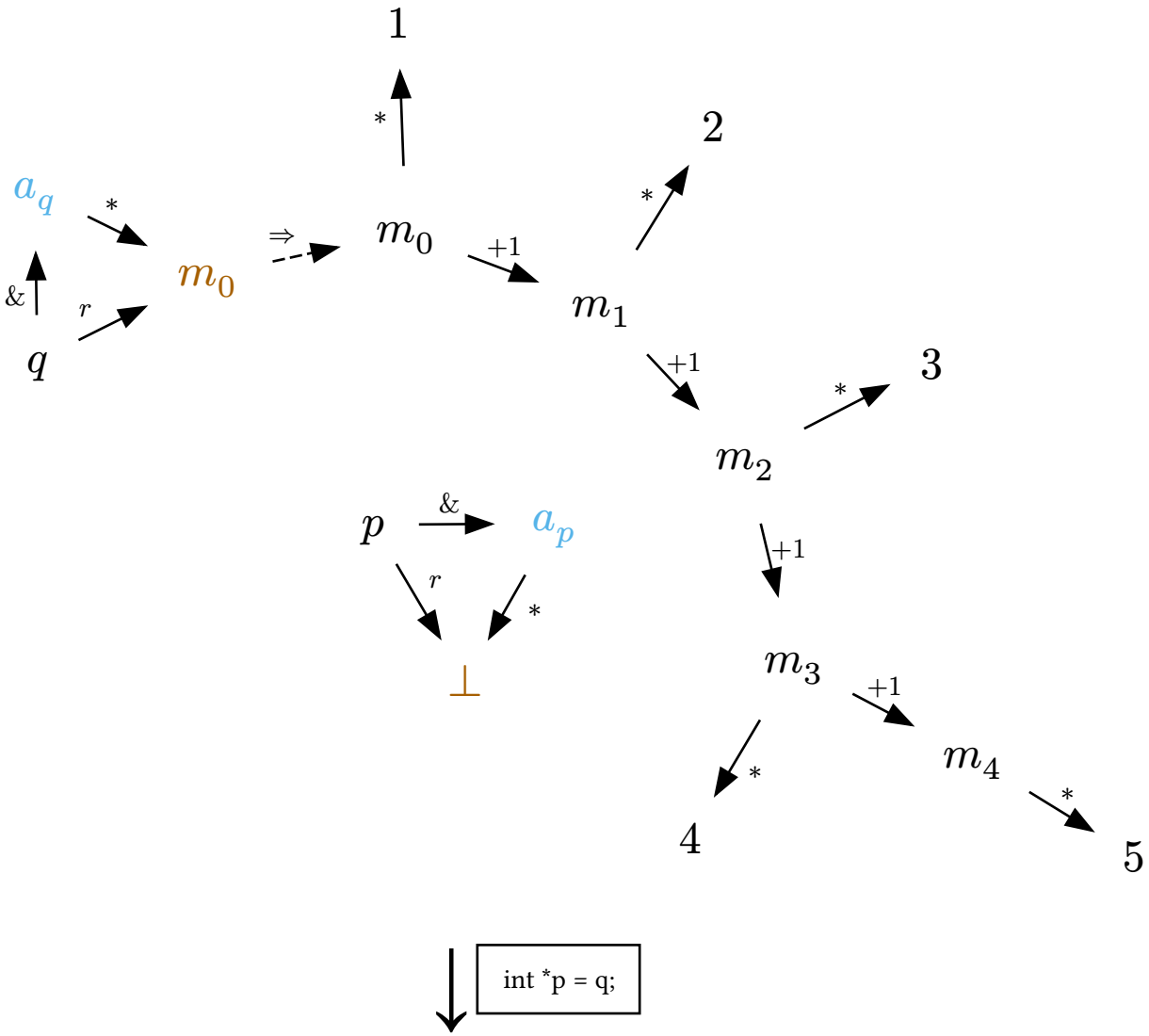


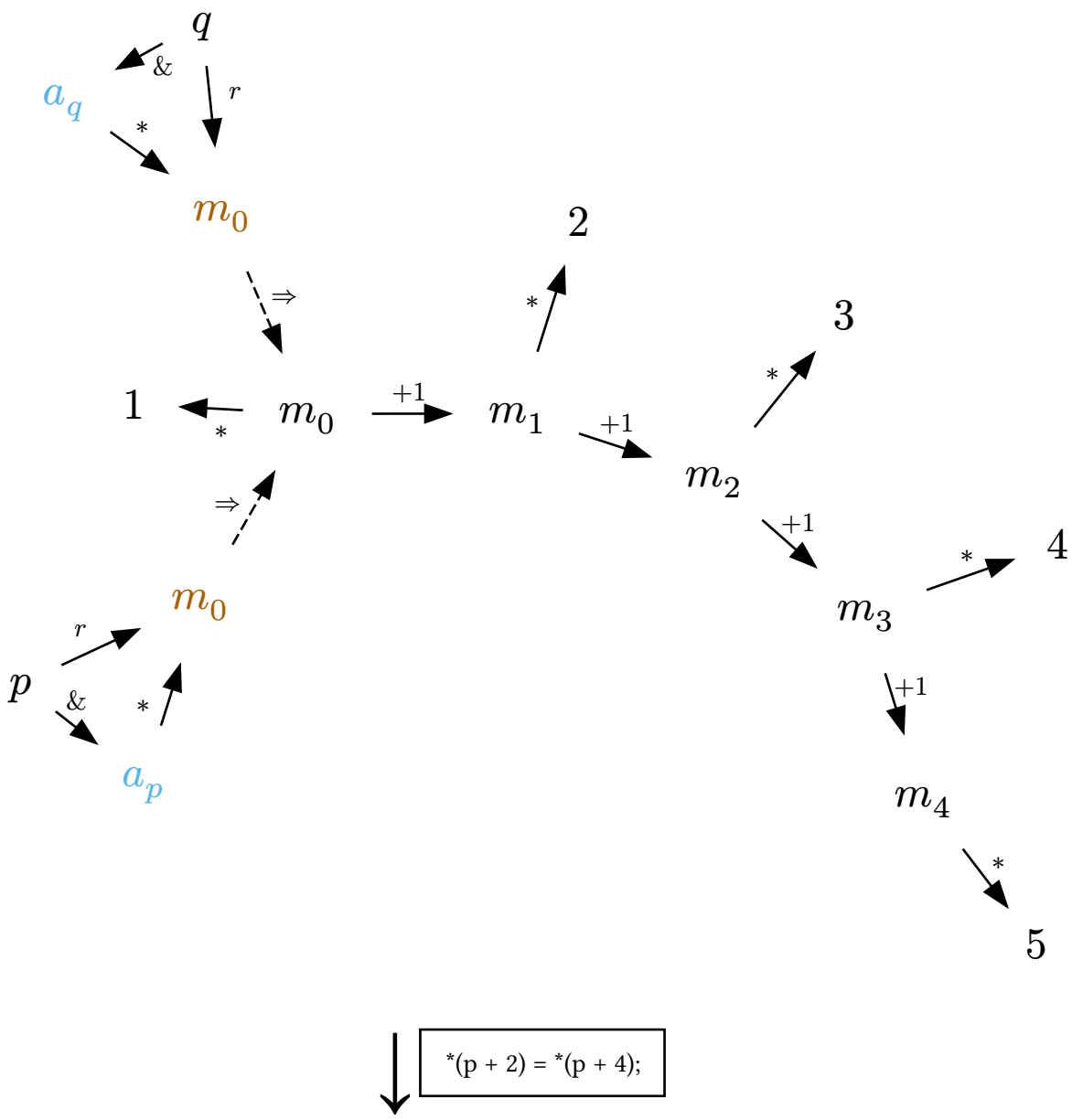
Example 8

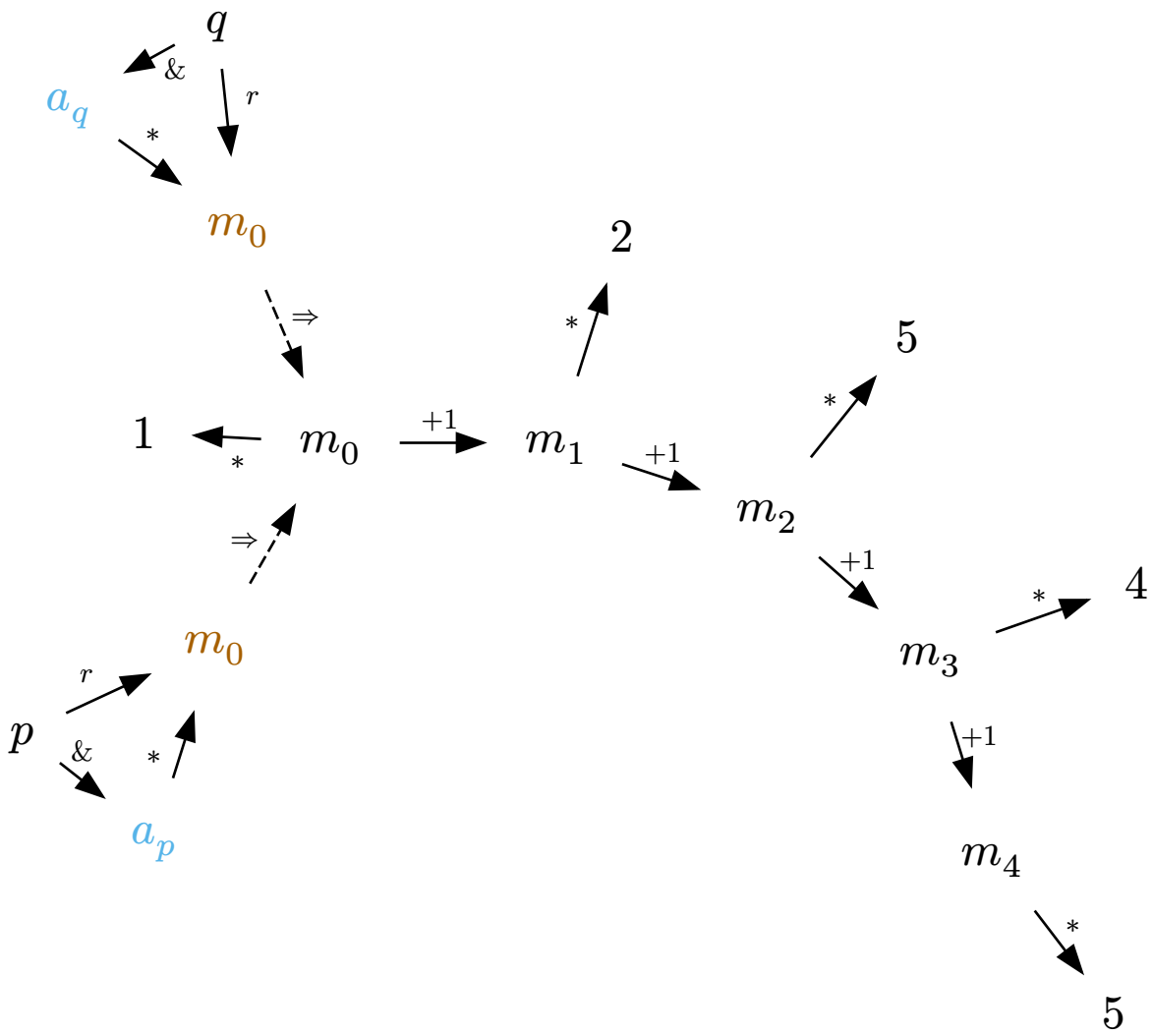
```
int q[] = {1, 2, 3, 4, 5};
int *p = q;
*(p + 2) = *(p + 4);
```





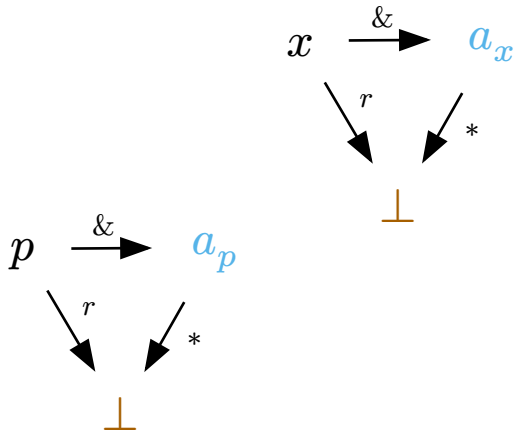

1
2
3
4
5
a_q
*
&
r
q
m_0
⇒
m_0
*
+1
m_1
*
+1
m_2
*
+1
m_3
+1
m_4
*
*
p
&
a_p
r
*
⊥
int *p = q;

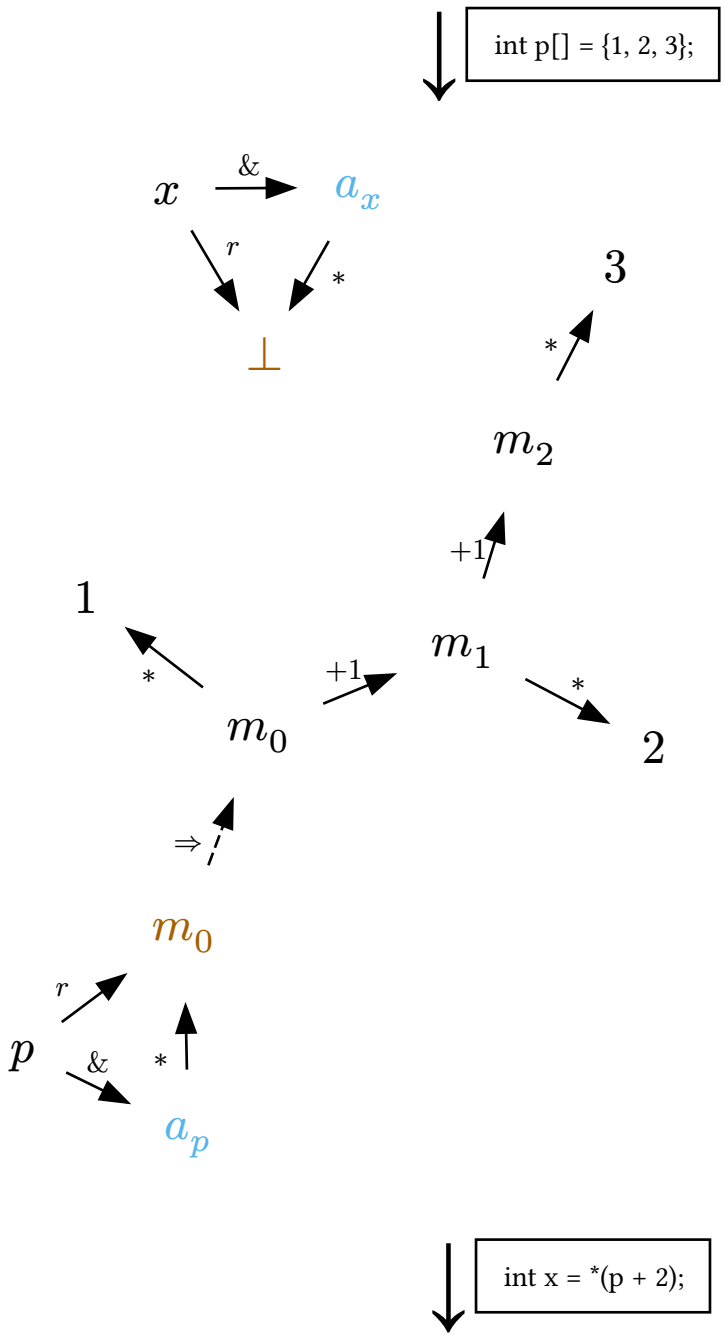

q
&
r
a_q
*
m_0
⇒
2
*
3
1
m_0
*
+1
m_1
+1
m_2
*
⇒
+1
4
*
m_0
m_3
r
p
&
*
+1
a_p
m_4
*
5
*(p + 2) = *(p + 4);

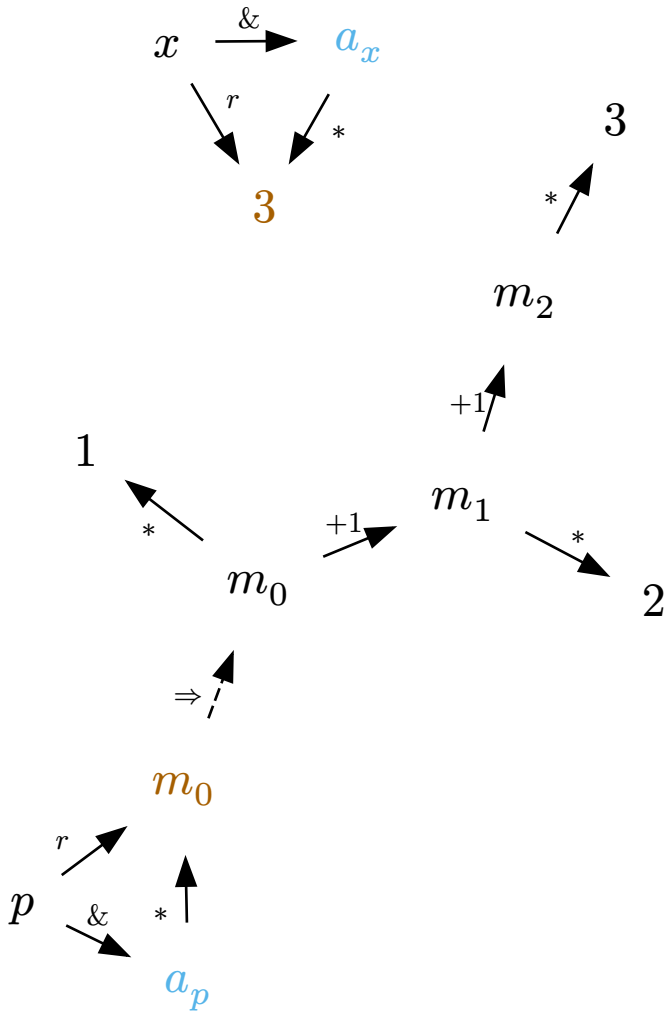


Example 9

```
int p[] = {1, 2, 3};
int x = *(p + 2);
```




int p[] = {1, 2, 3};
x
&
a_x
r
*
⊥
3
*
m_2
+1
1
*
m_1
+1
*
m_0
2
⇒
m_0
r
p
&
*
a_p
int x = *(p + 2);


x
&
a_x
r
*
3
3
*
m_2
+1
1
*
m_1
+1
*
m_0
2
⇒
m_0
r
p
&
*
a_p