Type inference (tiny) example

Consider this program fragment ...

x = y + 32;

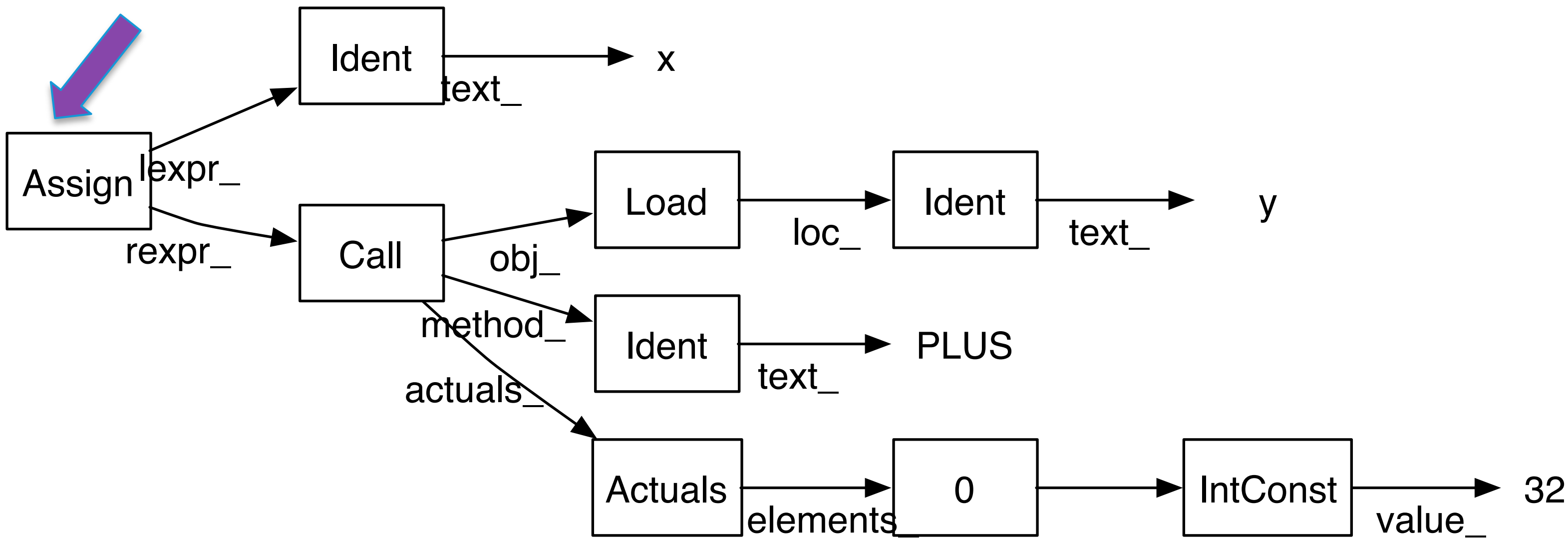
We will evaluate it in a program state with bindings

x	**?**
y	15 (Int)

Result will be a new program state, i.e.,

new values in the table

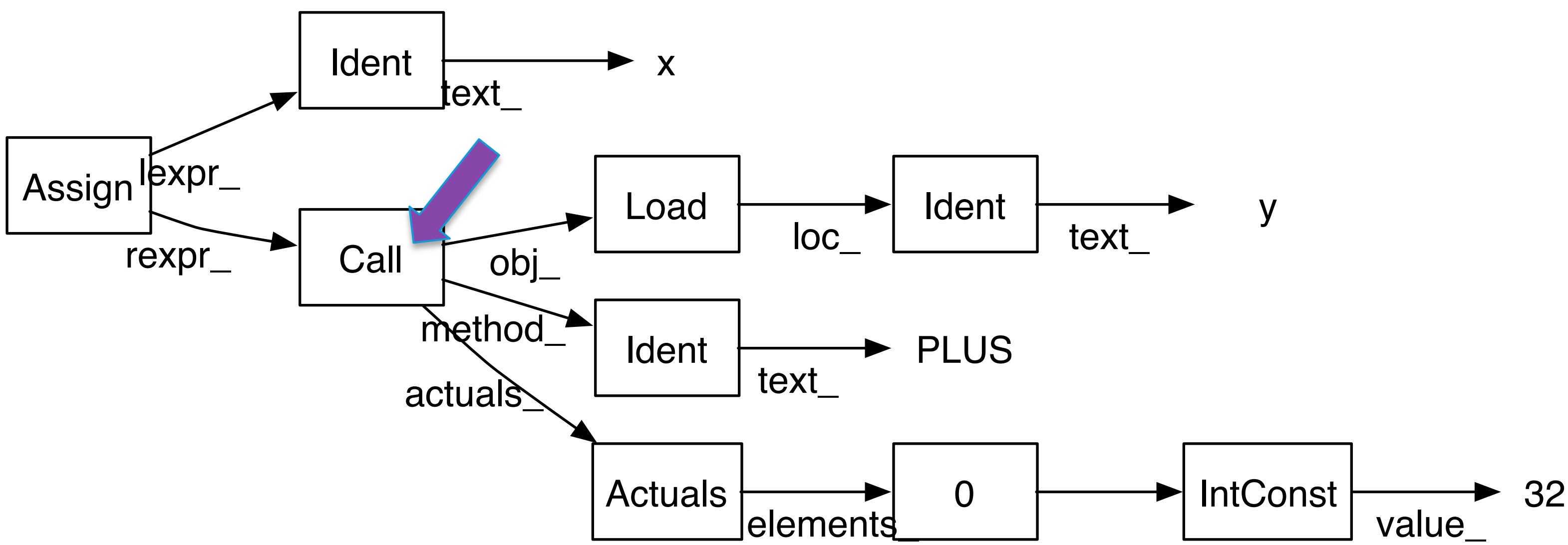
The program statement



x	**?**
y	15 (Int)

To evaluate an assignment,
Evaluate the right hand side for a value,
Evaluate the left hand side for a location,
Store the value in the location

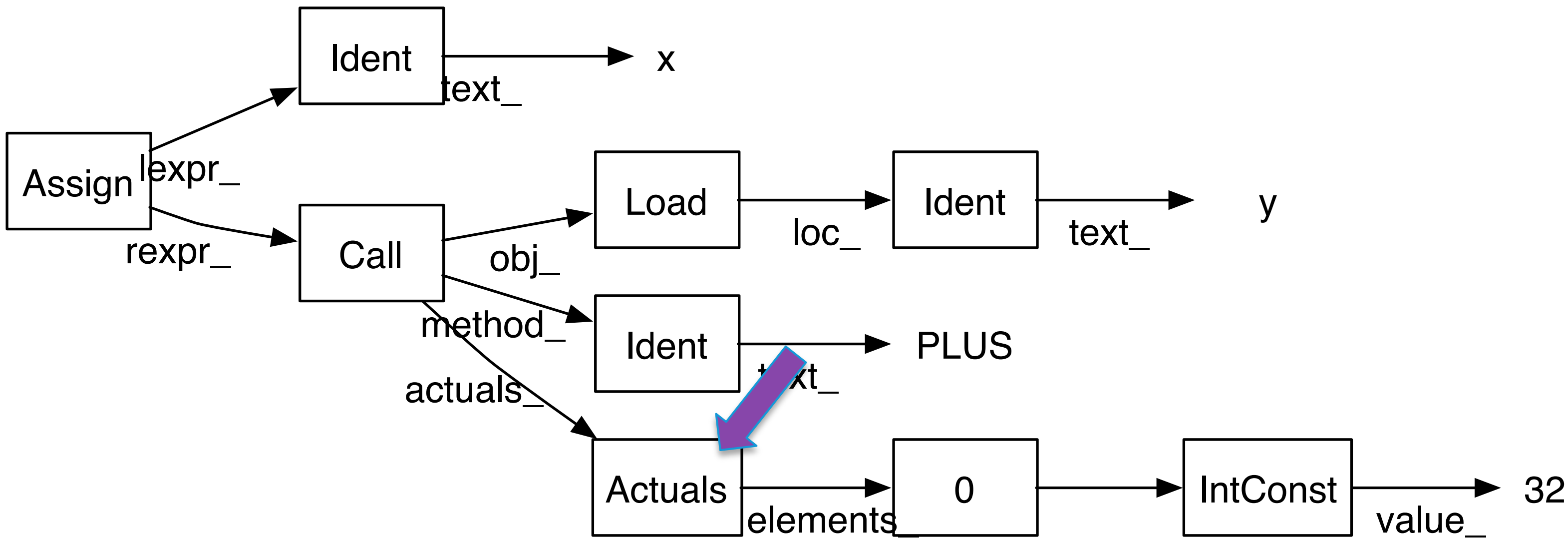
The program statement



x	**?**
y	15 (Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Pass the arguments to the method

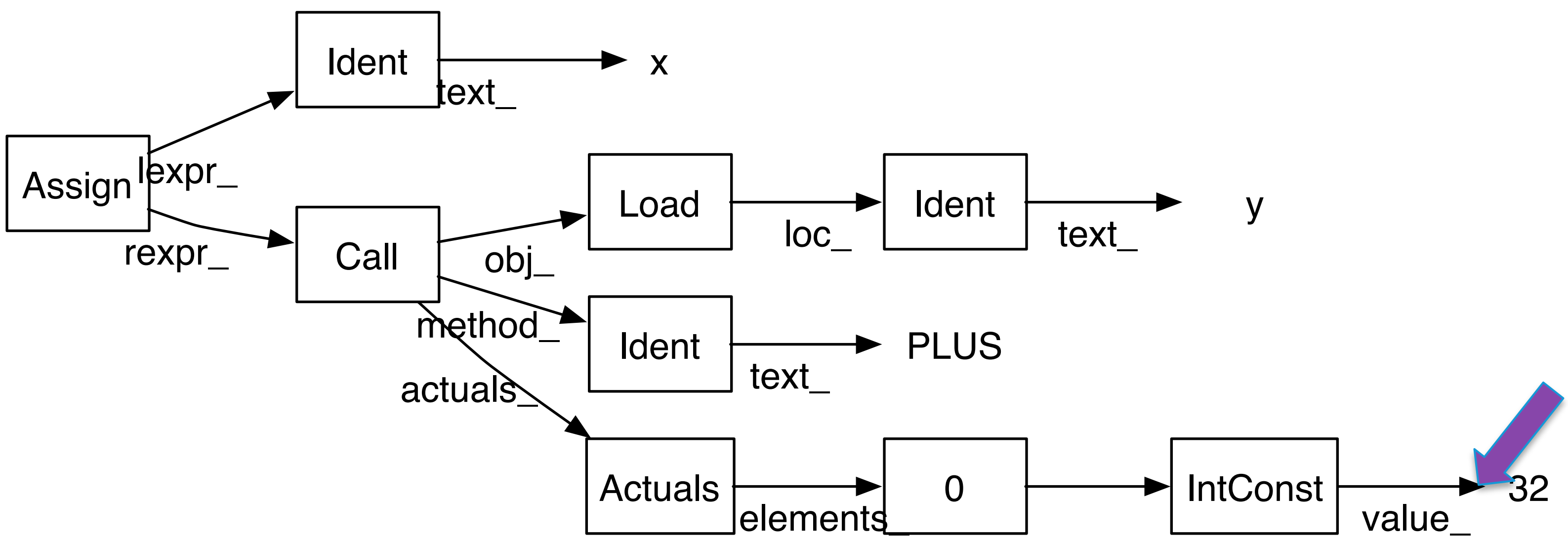
The program statement



x	**?**
y	15 (Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Pass the arguments to the method

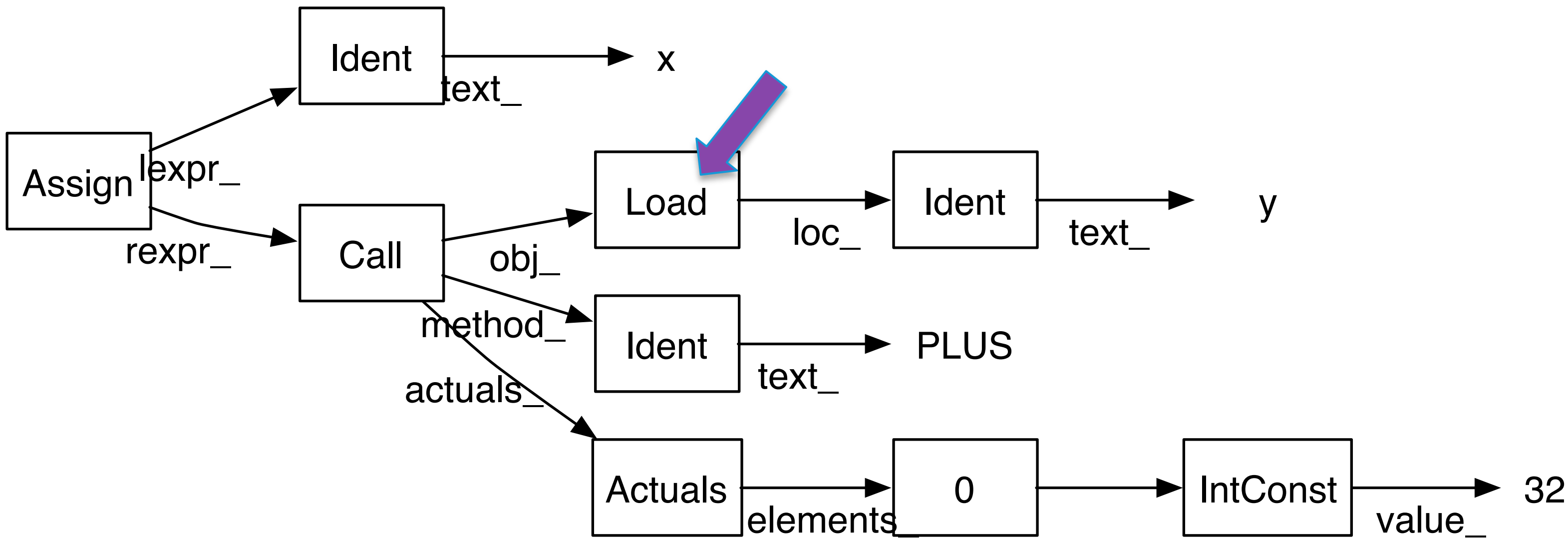
The program statement



x	**?**
y	15 (Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Pass the arguments to the method

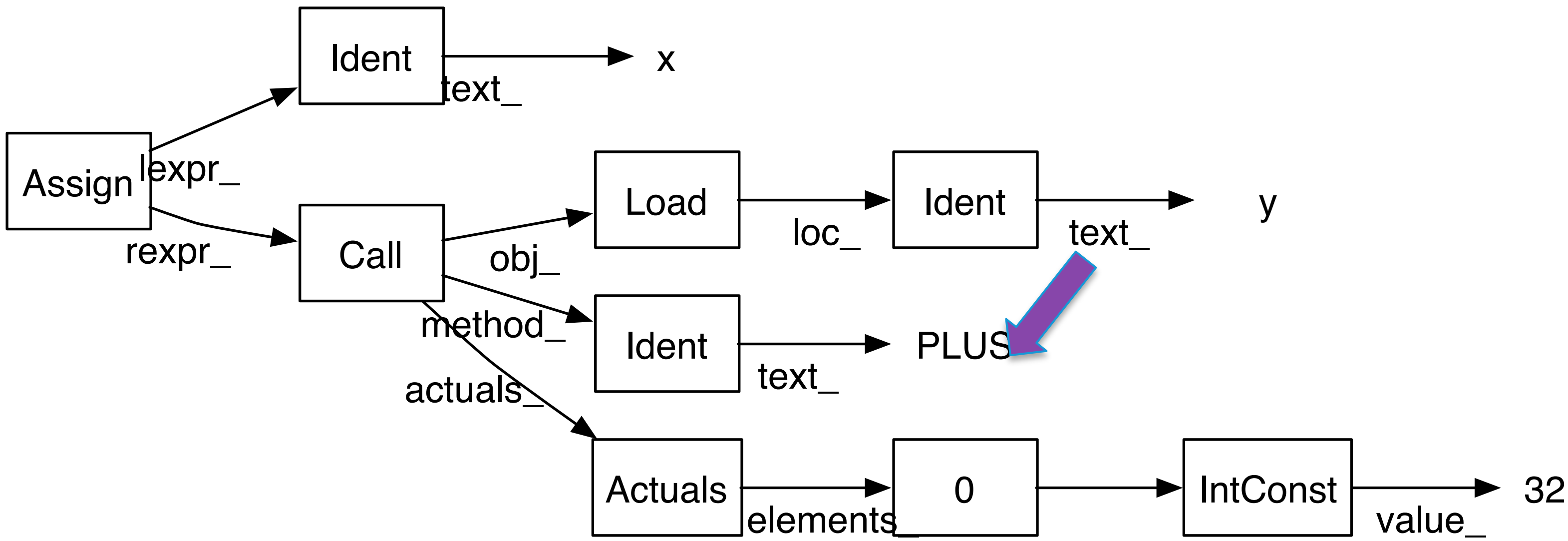
The program statement



x	**?**
y	15 (Int)

To find the method to call,
Evaluate the object expression,
Look up the method

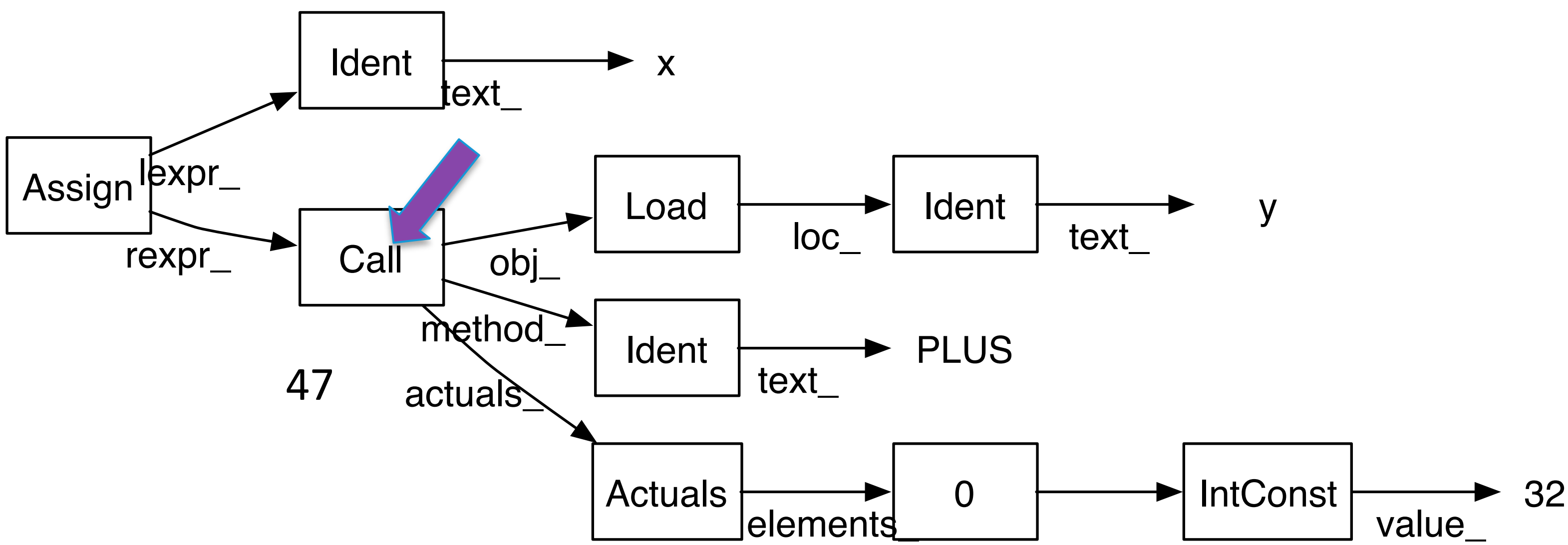
The program statement



x	**?**
y	15 (Int)

To find the method to call,
Evaluate the object expression,
Look up the method

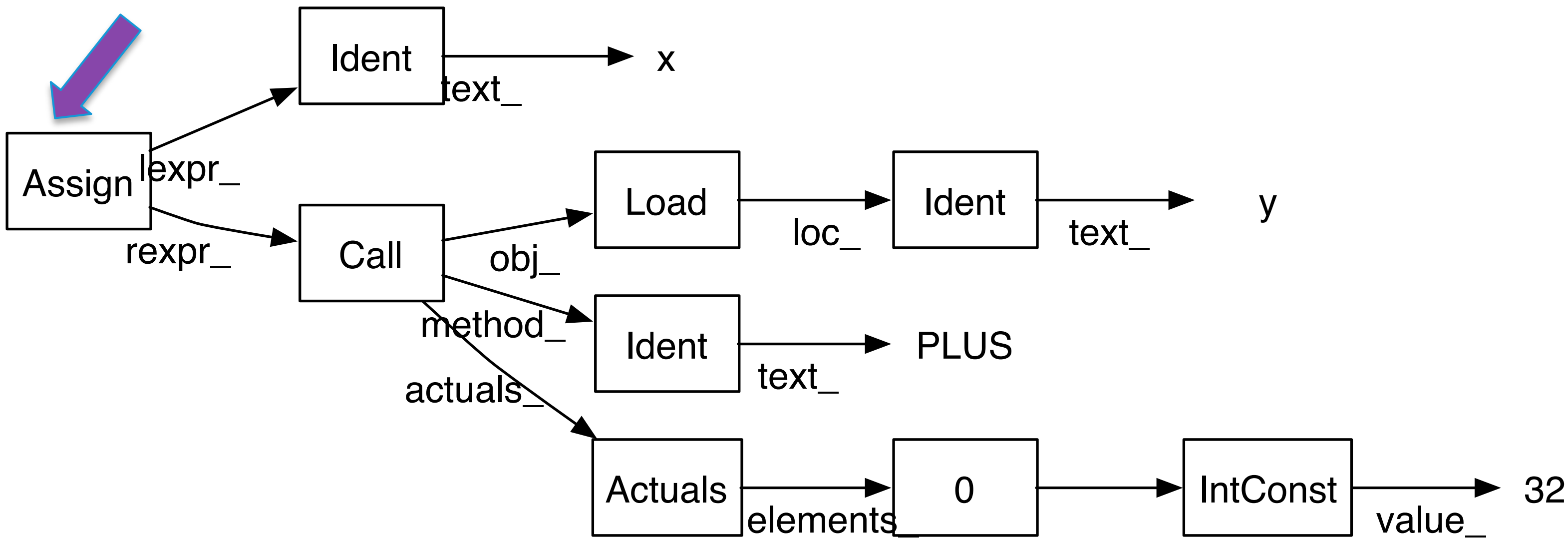
The program statement



x	?
y	15 (Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Pass the arguments to the method

The program statement



x	**47 (int)**
y	15 (Int)

To evaluate an assignment,
Evaluate the right hand side for a value,
Evaluate the left hand side for a location,
Store the value in the location

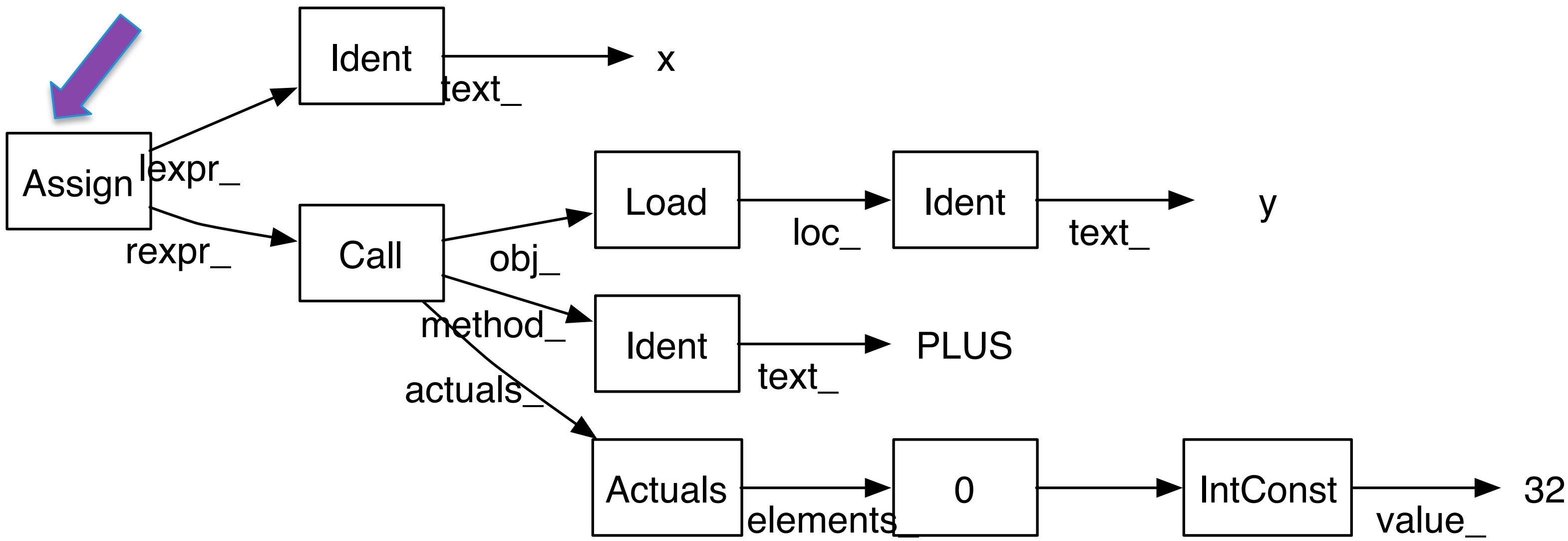
Once again, with types ...

The basic type inference step is just evaluation in which we track only value types, rather than actual values

Instead of Int value 32, just Int

Instead of PLUS(15,32) => 15, just PLUS(Int, Int) => Int

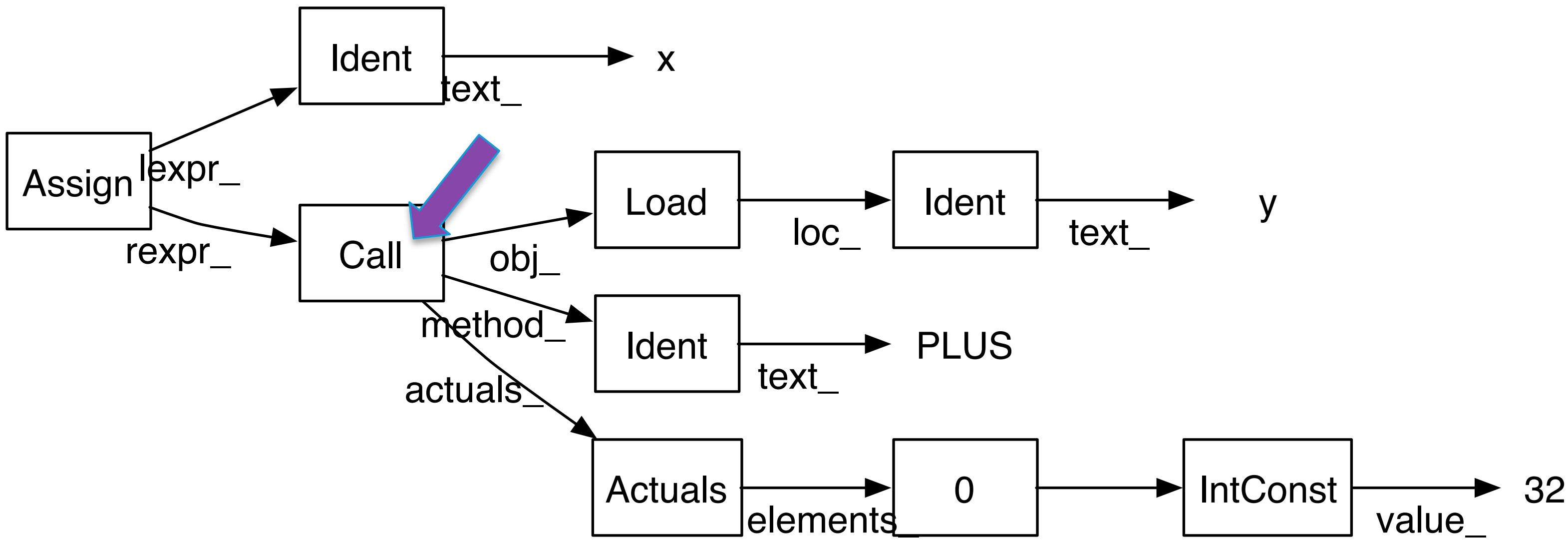
The program statement



x	**?**
y	(Int)

To evaluate an assignment,
Evaluate the right hand side for a ~~value~~ type,
Evaluate the left hand side for a location,
Store the ~~value~~ type in the location

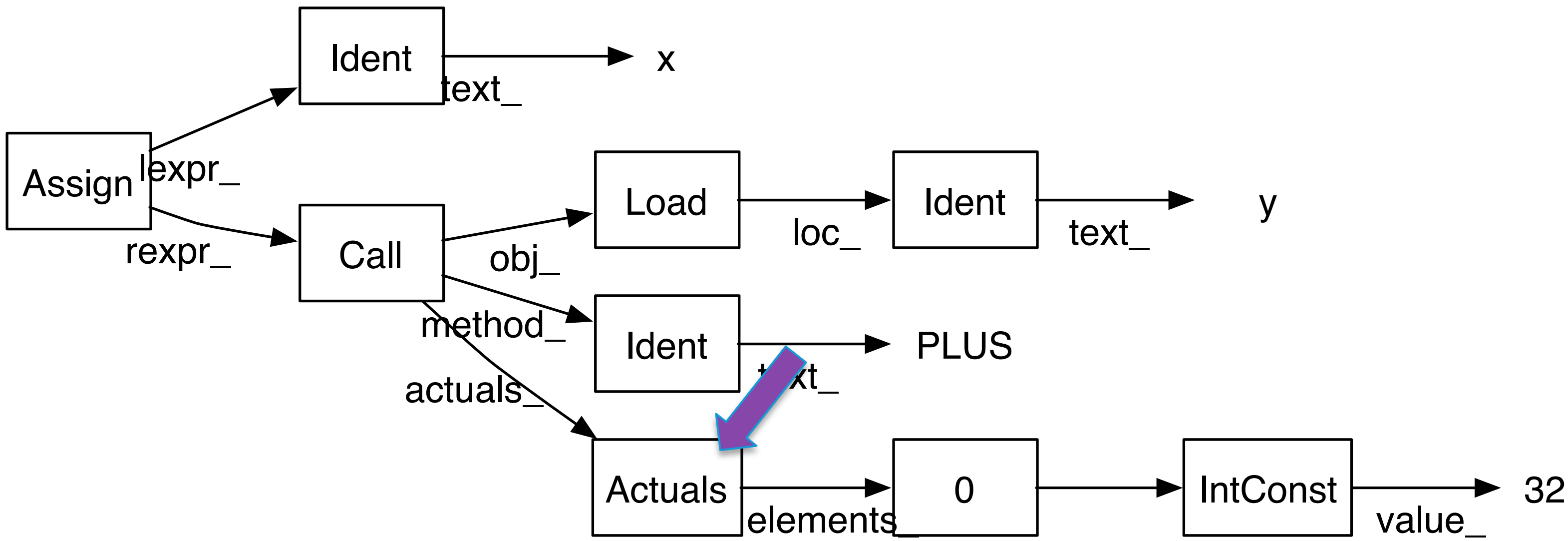
The program statement



x	**?**
y	(Int)

To evaluate a call,
Evaluate each actual argument for a type,
Find the method to call,
~~Pass the arguments to the method~~
Determine the return type of the method

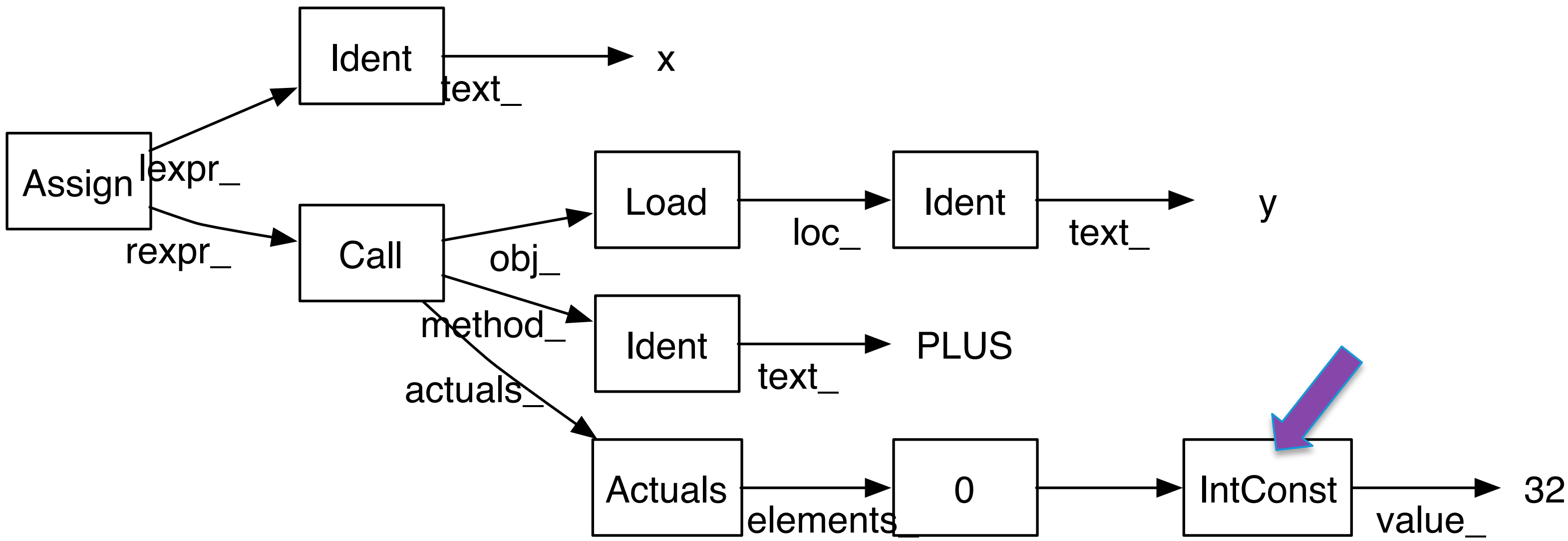
The program statement



x	**?**
y	(Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Determine the return type

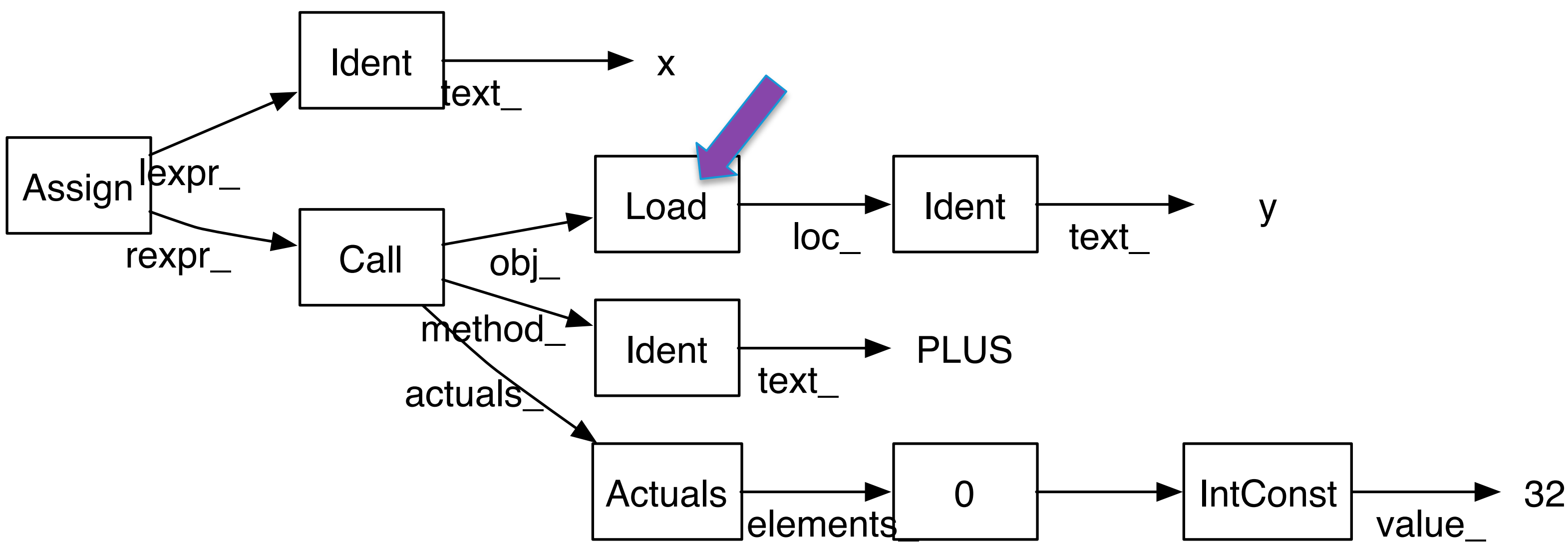
The program statement



x	**?**
y	(Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Pass the arguments to the method

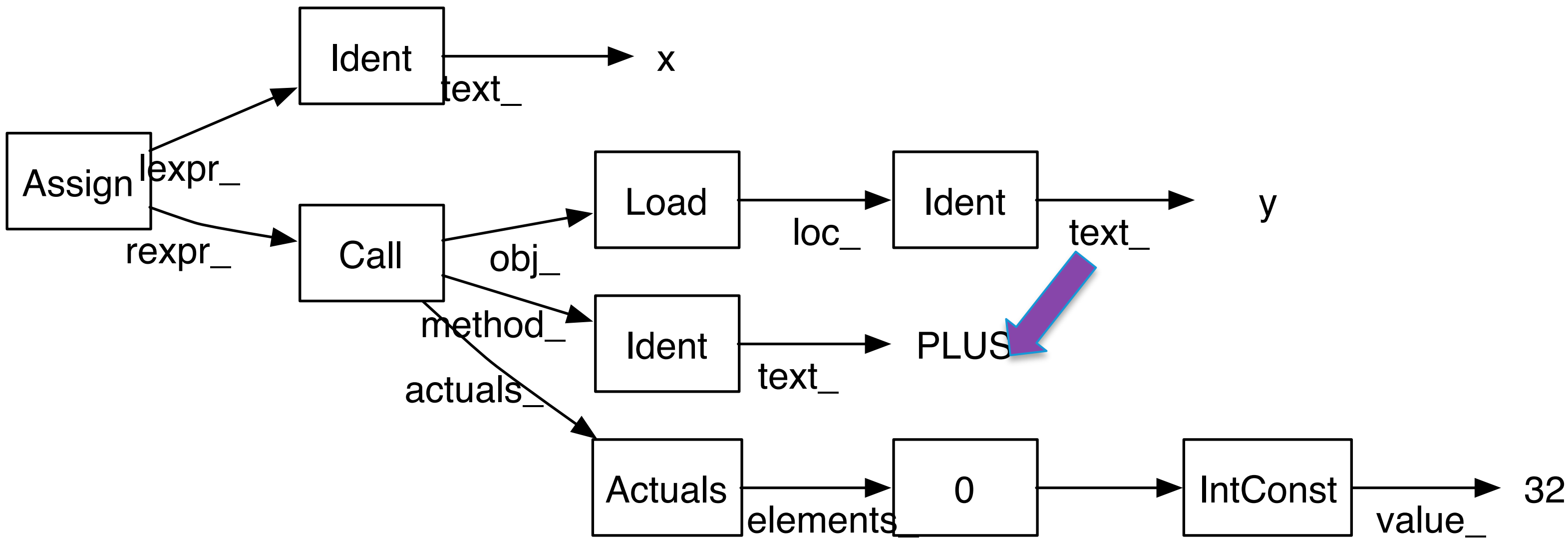
The program statement



x	**?**
y	(Int)

To find the method to call,
Evaluate the object expression,
Look up the method

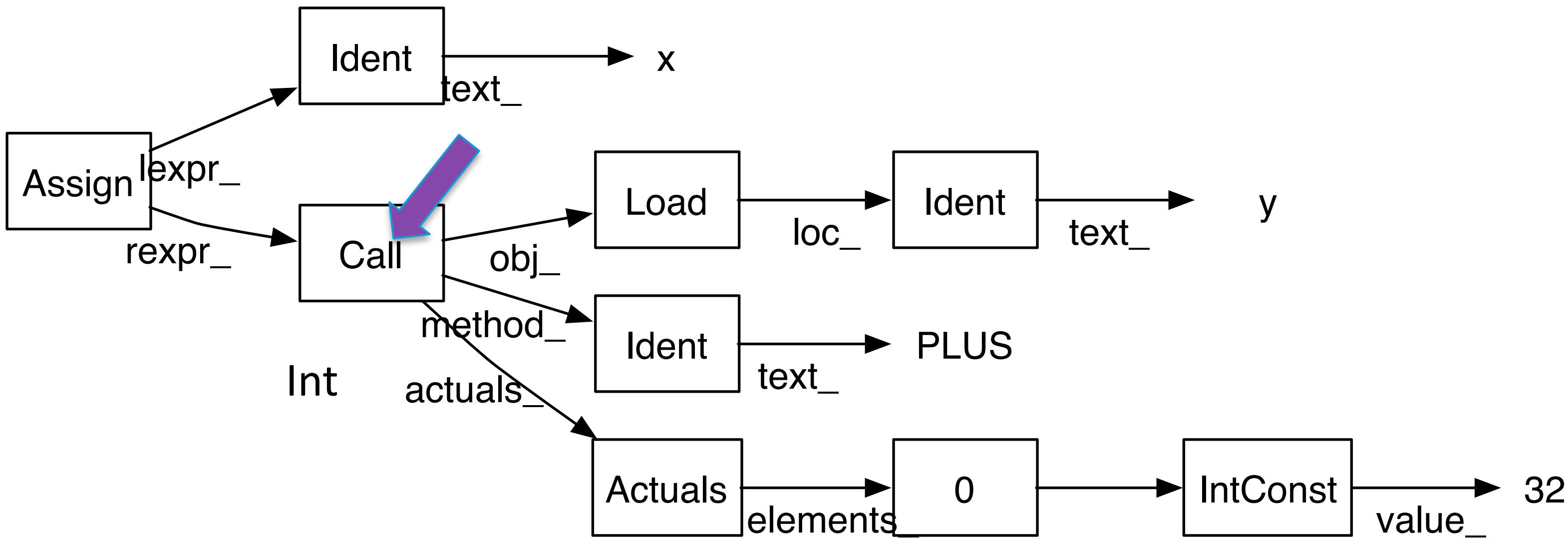
The program statement



x	**?**
y	(Int)

To find the method to call,
Evaluate the object expression,
Look up the method

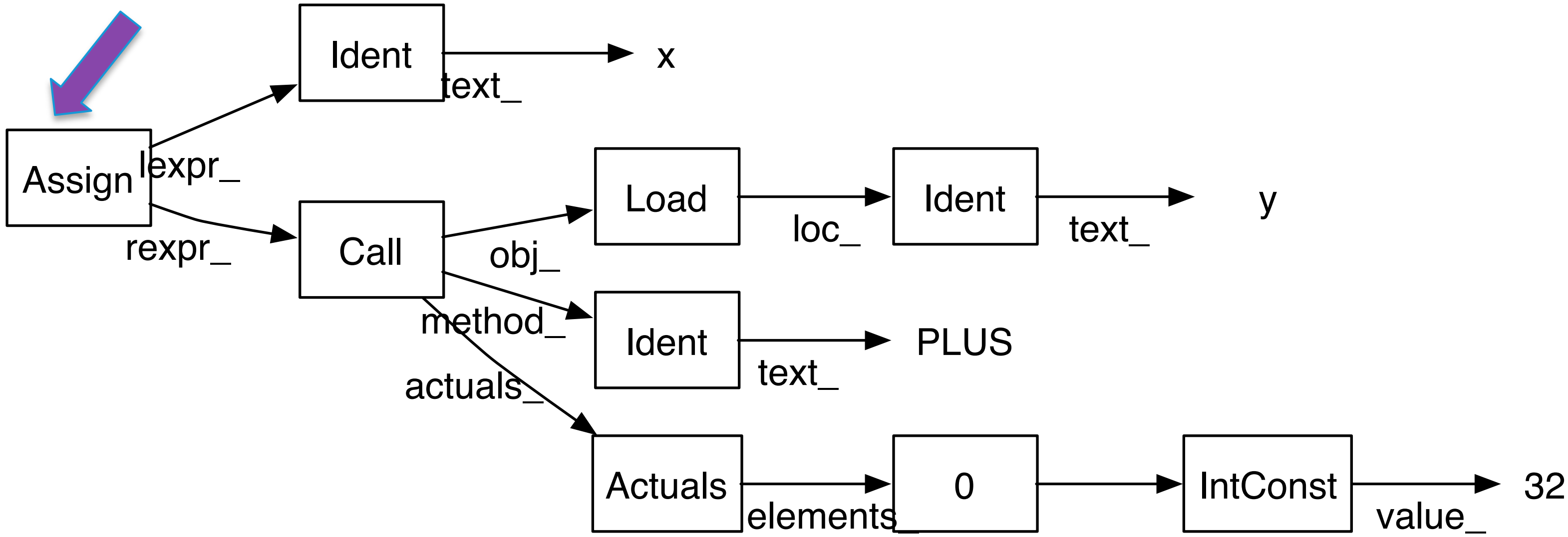
The program statement



x	**?**
y	(Int)

To evaluate a call,
Evaluate each actual argument,
Find the method to call,
Look up the return type

The program statement



x	**(int)**
y	(Int)

To evaluate an assignment,
Evaluate the right hand side for a type,
Evaluate the left hand side for a location,
Store the type in the location

But ... but ... what if the dynamic type changes?

We may have several different assignments to x,

or just one but it may get a different type on different loop iterations

Rule: All the dynamic types of values held in x must conform to the static type of x

Collecting semantics: Gather all possible executions of all statements that assign to x. The static type is the closest common ancestor type of all values stored in x.

All we need is infinite space and time ... should be easy?

Types as abstraction of values

Instead of values, we store types as "abstract values"

If x has abstract value Int, and we assign it a String, instead of remembering both values { Int, String },

take the least common ancestor immediately.

x = 17;

x	**(Int)**

x = "Ostrich";

x	**(Obj)**

LCA(Int,String) = Obj

x = x + "Feathers";

We will look for method PLUS in class Obj. No such method. This is a type error.

What we will need

The class hierarchy is an actual data structure

We need an operation LCA(T_1, T_2) that returns the least common ancestor of T_1 and T_2

LCA(X, $\top$) = $\top$

LCA(X, $\perp$) = X

We can check one method at a time.

Initial type of input arguments is declared; all other variables start at $\perp$

At method call, we check for method in current estimate of receiver object type to get result type

Why does this work?

The subtype hierarchy is a lattice.

The type inference is monotone:

- Given one set of type estimates,
- a re-calculation can only move estimates up the lattice,
- never across or down

Guaranteed to converge!

(And to get the right answer.)

Underlying theory: Abstract interpretation.

Flow sensitive vs. flow insensitive

Abstract interpretation can calculate an abstract value at each program point. That would be a *flow sensitive* analysis. Used in verification of properties (not just types).

Our type inference (like most type systems) is *flow insensitive*. We lump together all values a variable takes, without respect to where.

If our analysis were flow sensitive this would be ok:

x = 32;

x = x + 7;

x = "Ostrich";

x = x + "Feathers";