Concrete Types – Introduction

- Provide specific and detailed information (in contrast to abstract and general)
- Reveal the implementation and internal structure of objects:
    - how the they are laid out in memory,
    - how many instance variables they have,
    - what code they execute to perform operations, etc
- Have to statically solve the late-binding that exists between messages and methods – the real problem

Concrete Type – Definition

The concrete type of an object is:

- the *exact* class of the object, plus
- the concrete type of each instance variable, plus
- the concrete type union of all the indexed variables

For example:

3 concreteType ⇒ <SmallInteger>

(1.0@1.0) concreteType ⇒ <Point x: <Float> y: <Float>>

#(1.0 'two') concreteType ⇒ <Array (<Float> <String>)>

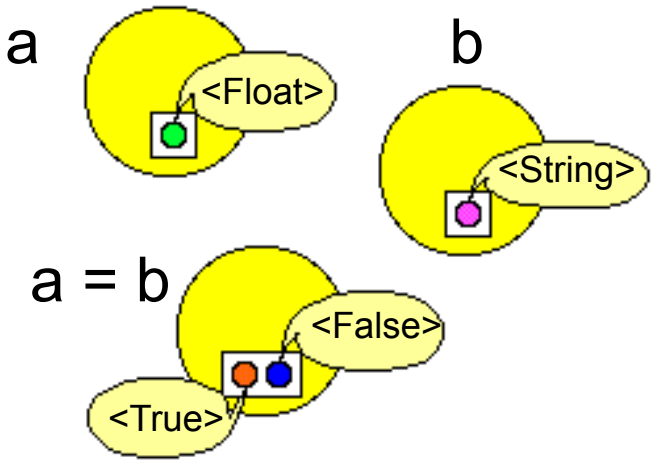
Concrete Types of Expressions

- The concrete type of an expression is a set

  a = b ⇒ {<True> <False>}

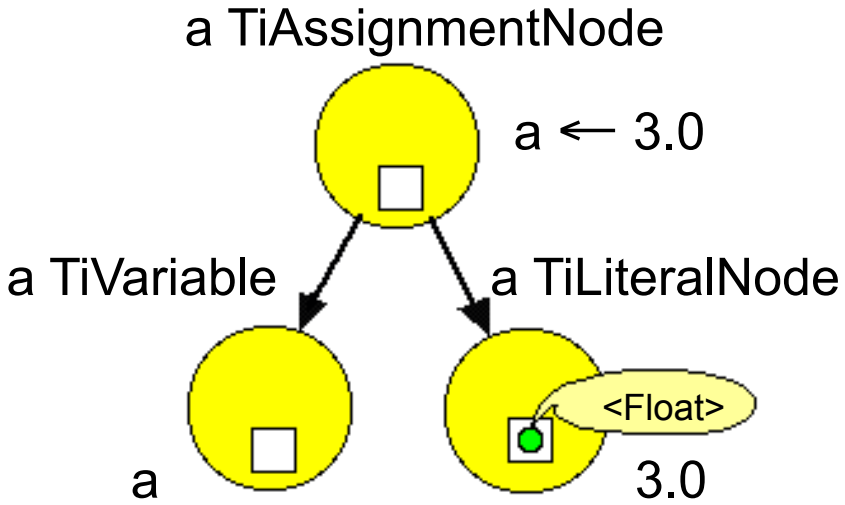
- We assign a slot to every expression
- Each slot holds the concrete type set of its expression

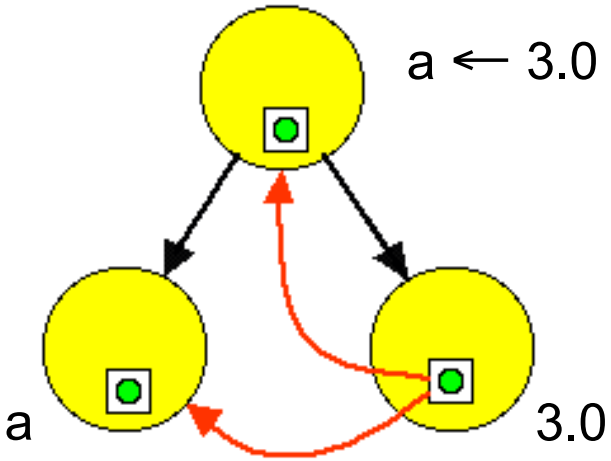
a ← 3.0.
b ← ‘hello’.
↑ a = b



Slots and Connections

- Slots are connected reflecting the runtime data-flow
- Types flow across connections

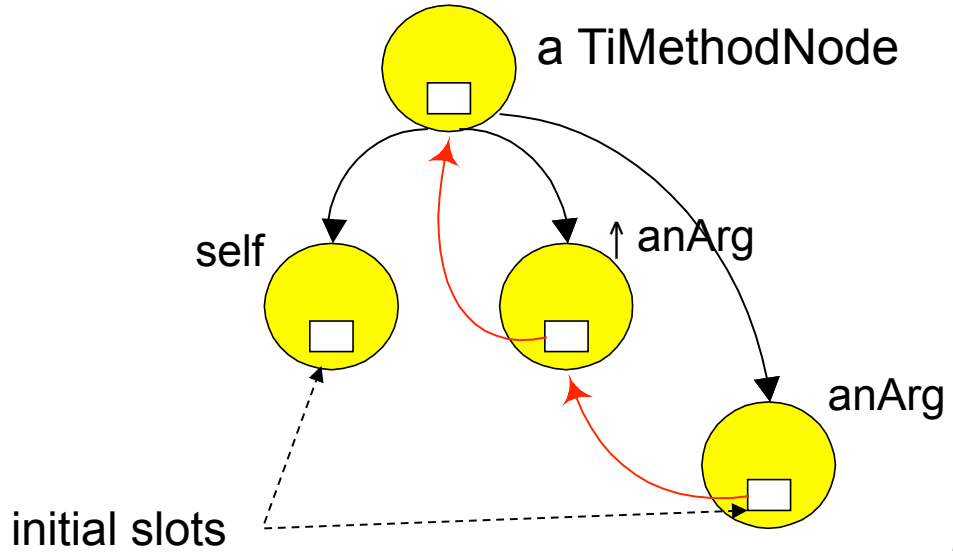




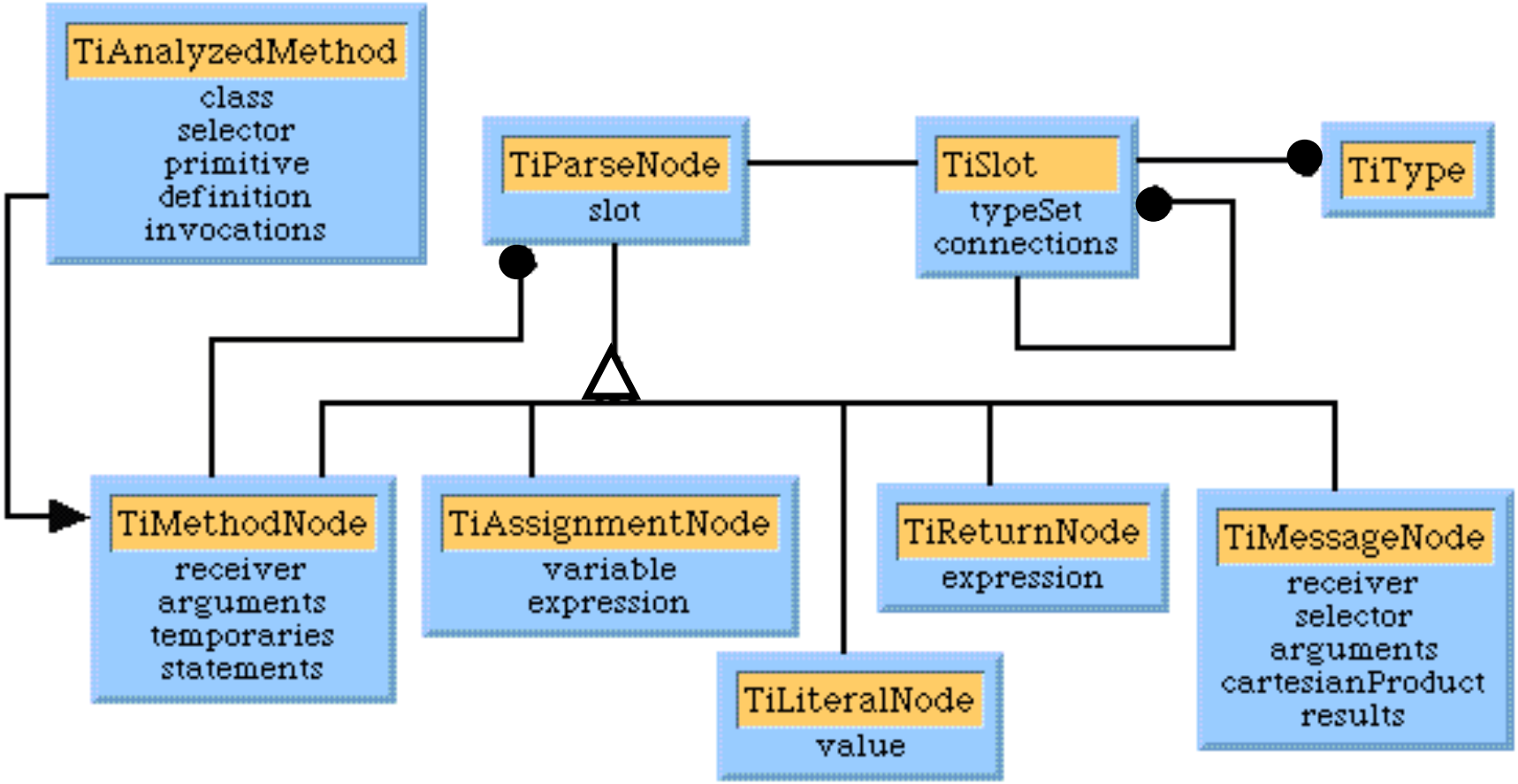
Method Connections

- A type flow network is built for every analyzed method
- The network is built by connecting the method parse tree
- Method analysis begins with the monomorphic seeding of initial slots

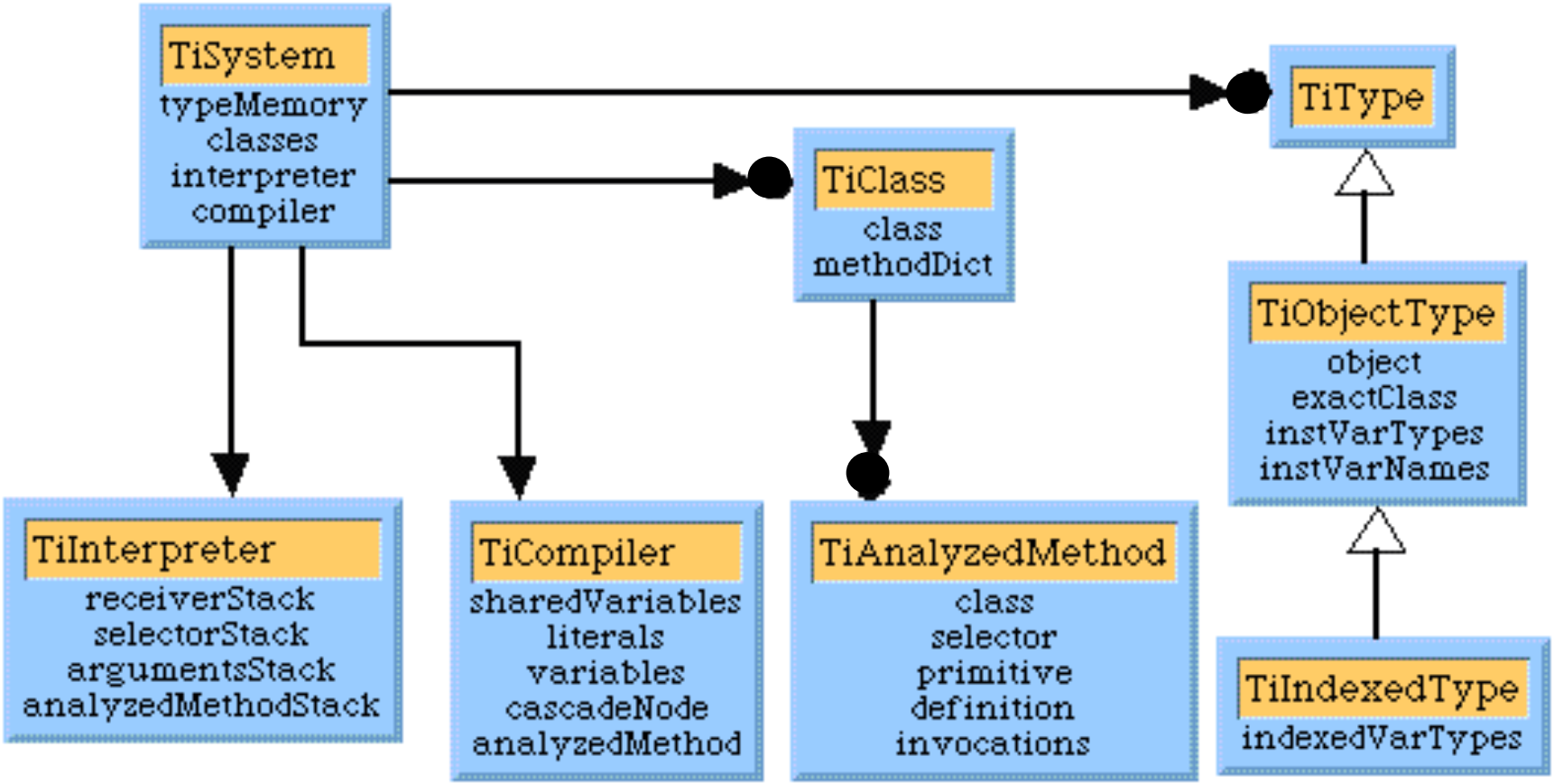
Object | returnArg: anArg
↑ anArg



Analyzed Methods

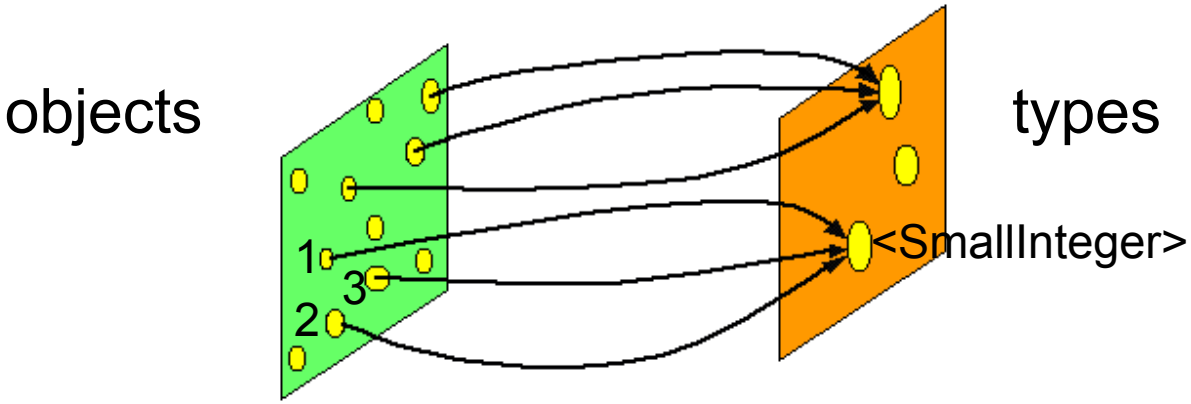


The Type Inference System



System Correspondance

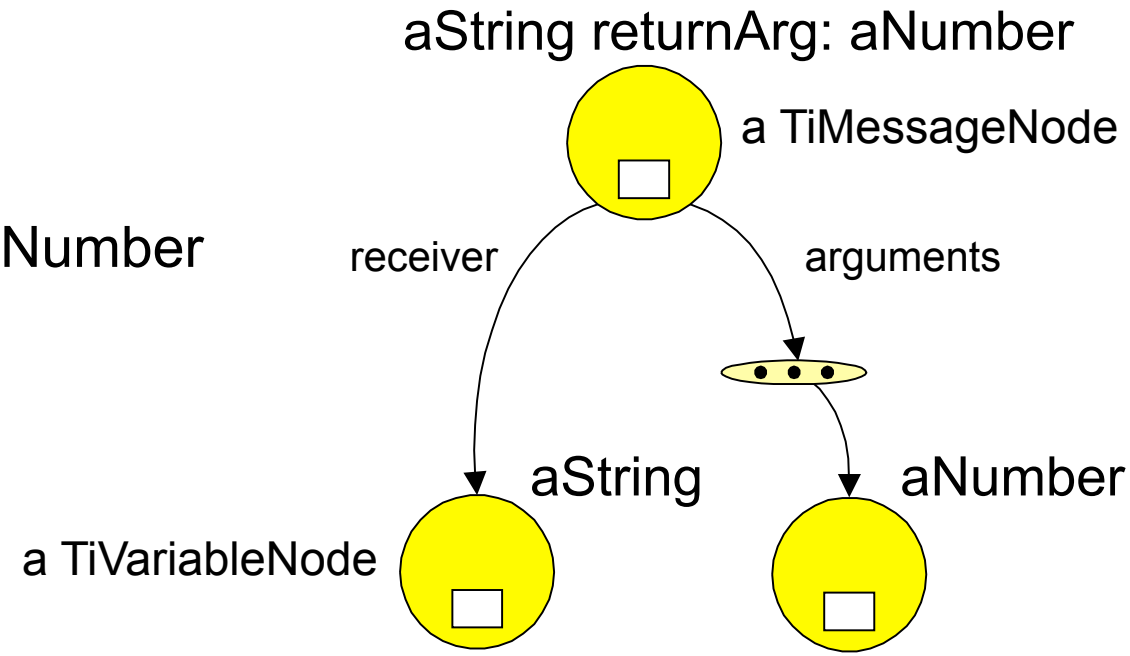
Smalltalk System	Type Inference System
compiled method	analyzed method
bytecodes	type flow network
compile time	connection time
run time	flow time



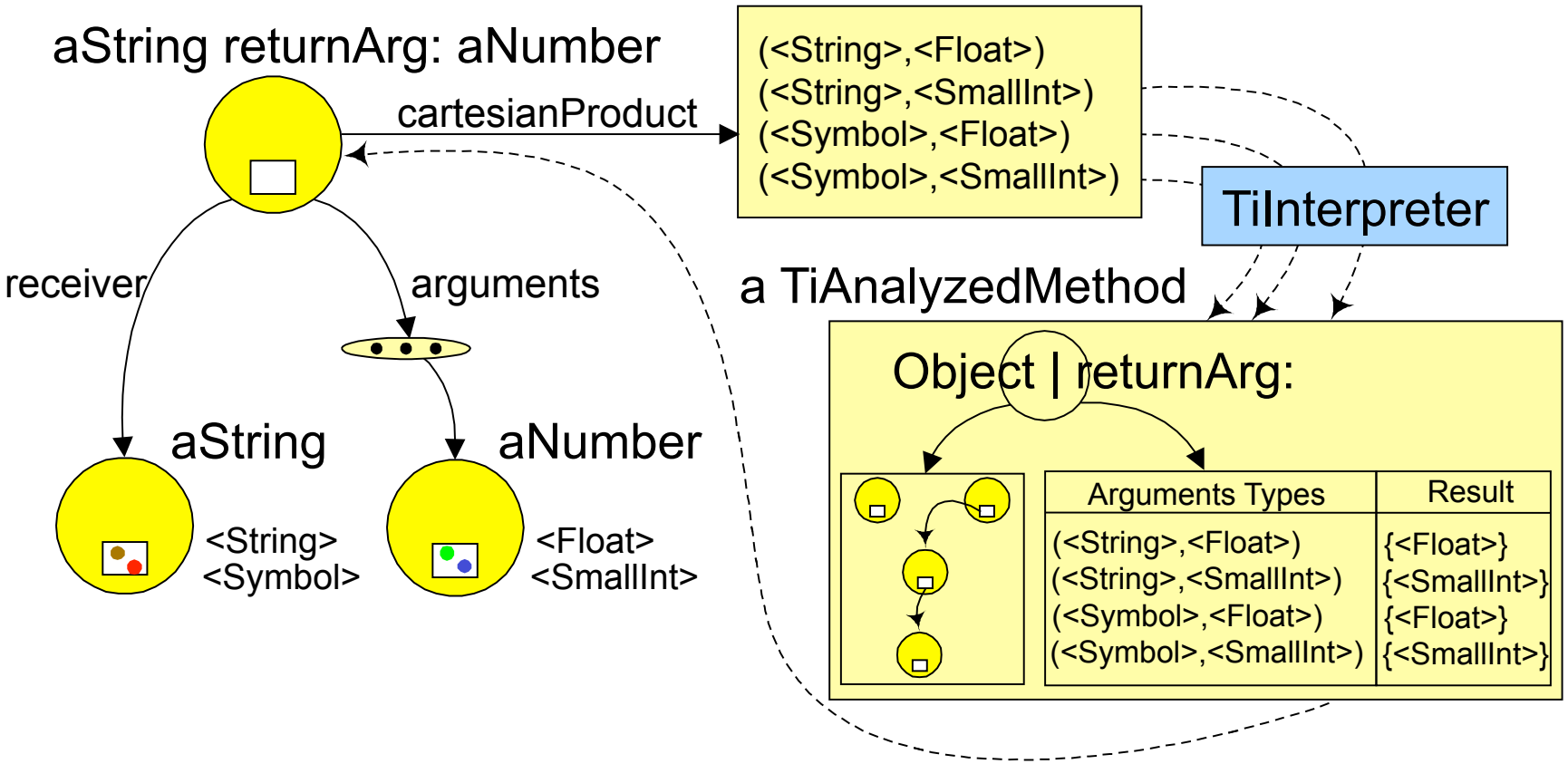
Connecting Message Expressions

It is impossible to meaningfully connect the slots of a message send expression

```
| aString aNumber |
.
.
aString returnArg: aNumber
.
```



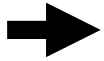
Late Binding in action



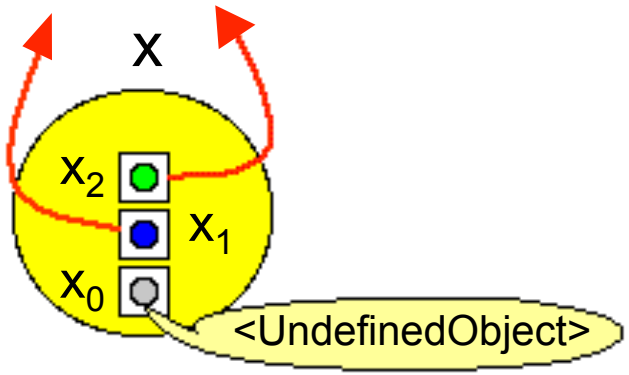
Local Variable States

- An assignment generates a new state in the assigned variable
- Each variable state is represented with a unique slot (and subindex)
- Only variables have states (and thus, multiple slots)

```
| x |
x ← 1.
x ← x@1.
↑ x
```

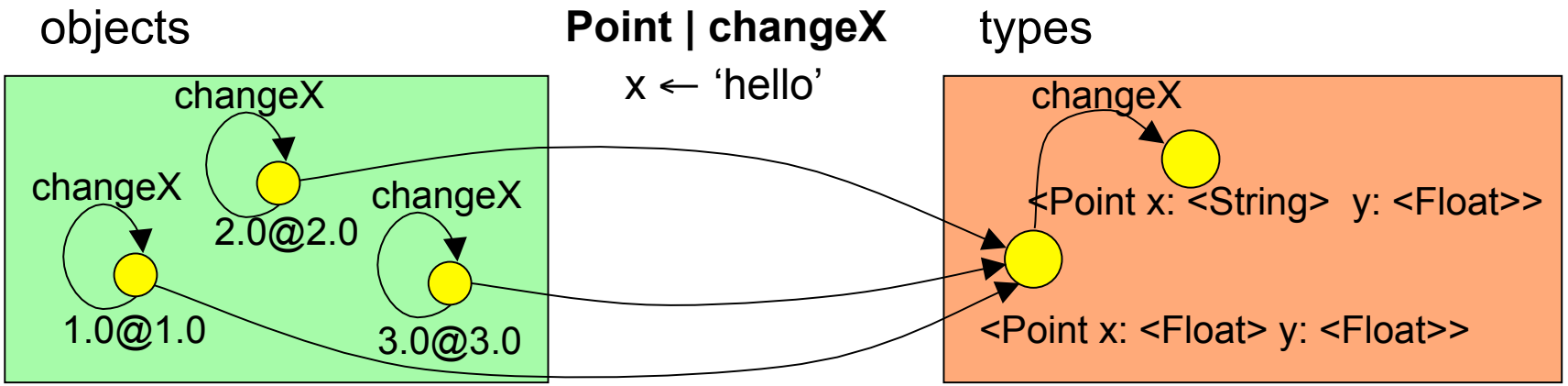


```
| x0 |
x1 ← 1.
x2 ← x1@1.
↑ x2
```



Side Effects on Types

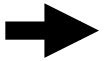
- Messages could have side effects on objects
- A side effect is a change to the internal structure of an object
- A side effect is reflected with a new type



Side Effects on Variables

- When a variable is an argument, it could suffer side effects
- A variable that suffers side effects, changes its type
- A new state is added when a variable plays the role of a message argument

```
| a |
a ← 1.0@1.0.
a changeX.
↑ a
```



```
| a0 |
a1 ← <Point x: <Float> y: <Float>>.
a1 changeX.
a2 ← messageResult( receiver ).
↑ a2
```

† **Point | changeX**
x ← 'hello'

Instance Variable States

- An assignment to an instance variable changes the receiver
- Instance variable slots should be synchronized with receiver slots

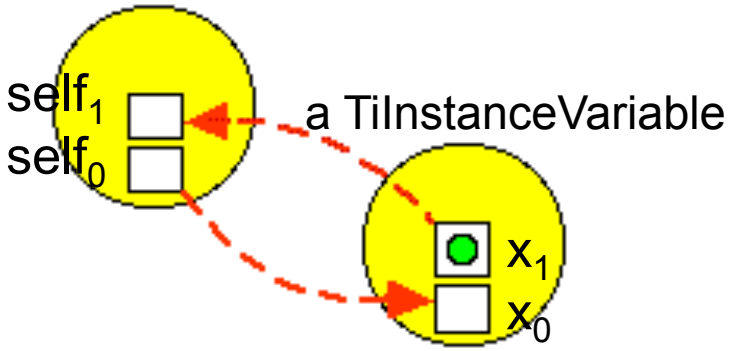
Point | changeX

x ← 'hello'

x_0 ← $self_0$.x

x_1 ← <String>

$self_1$ ← *merge($self_0$, x_1)*



$self_0$	<Point x: <Float> y: <Float>>
x_0	<Float>
x_1	<String>
$self_1$	<Point x: <String> y: <Float>>

Primitives

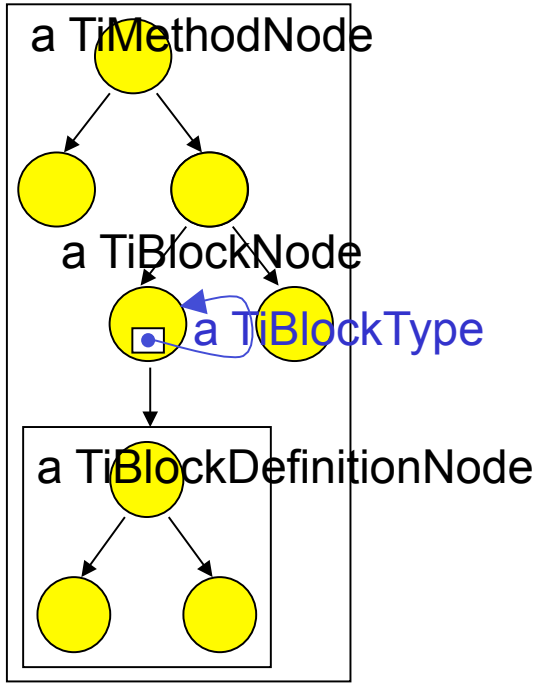
- Primitive responses are simulated with primitive analysis
- Each primitive analysis is implemented inside a TiPrimitive method
- Each primitive analysis is a different challenge


```
TiPrimitive | primitiveFloatAdd
    ↑ (self matchReceiver: Float arguments: Float)
        ifTrue: [self createResult addReturn: Float createType]
        ifFalse: [TiPrimitiveResult failed]
```

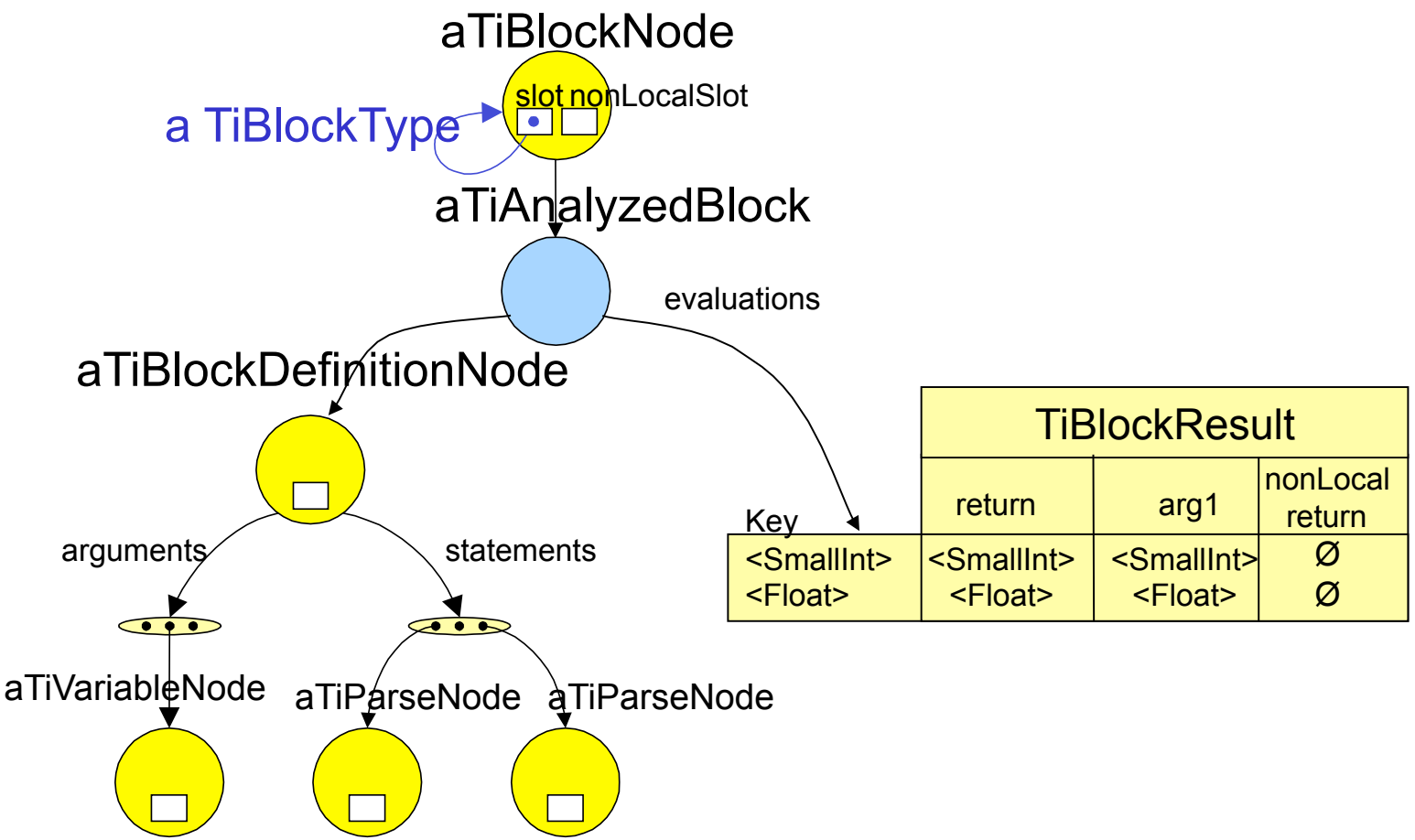
```
TiPrimitive | primitiveNew
    ↑ self createResult addReturn: self receiver exactClass
    soleInstance createType
```

Block Types

- Each block has a different type
- The type of a block is special
- The type of a block has a reference to the block definition
- Block evaluation is requested in a different method where it was defined

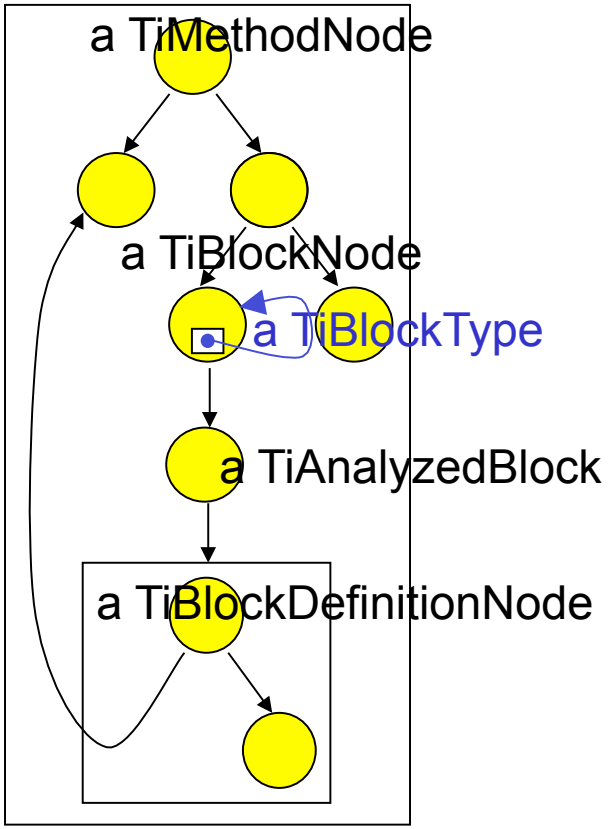


Analyzed Blocks



References to outer scopes

- We can correctly analyze blocks that only refer to their own arguments
- But blocks could have references to variables defined in outer scopes
- Those references are our major source of inaccuracy
- Solution: convert those references into block pseudo-arguments



Type Inference in Action

- Challenger is a simple class (38 lines of code) to test arithmetic proficiency on children
- The initial expression is ‘Challenger new run’
- It infers 17 concrete types and analyzes 23 methods

- ‘Display boundingBox’ returns:

{<Rectangle origin: <Point x: <SmallInteger> y: <SmallInteger>>
corner: <Point x: <SmallInteger> y: <SmallInteger>>>}

Further References

- Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Standford University, 1996.
- Francisco Garau. *Concrete Type Inference in Squeak*. Licenciate thesis, University of Buenos Aires, 2001.

http://typeinference.swiki.net