Static Analysis of Dynamically Typed Languages made Easy

Yin Wang

School of Informatics and Computing

Indiana University

Overview

- Work done as two internships at Google (2009 summer and 2010 summer)

- Motivation:
  - The *Grok* Project: static analysis of all code at Google (C++, Java, JavaScript, Python, Sawzall, Protobuf …)
  - Initial goal was not ambitious:
    - Implement "IDE-like" code-browsing
    - Turns out to be hard for Python

Achieved Goals

- Build high-accuracy semantic indexes
- Detect and report semantic bugs
  - type errors
  - missing return statement
  - unreachable code
  - …

Demo Time

Go

Problems Faced by Static Analysis of Dynamically Typed Languages

1. Problems with Dynamic Typing

- Dynamic typing makes it hard to resolve some names

- Mostly happen in *polymorphic functions*

```
def h(x):
  return x.z
```

1. Problems with Dynamic Typing

- Dynamic typing makes it hard to resolve some names

- Mostly happen in *polymorphic functions*

```
def h(x):
  return x.z
```

Q: Where is ‘z’ defined?
A: ... wherever we defined ‘x’

1. Problems with Dynamic Typing

- Dynamic typing makes it hard to resolve some names

- Mostly happen in *polymorphic functions*

Q: What is ‘x’ ?
A: Uhh…

```
def h(x):
  return x.z
```

Q: Where is ‘z’ defined?
A: … wherever we defined ‘x’

1. Problems with Dynamic Typing

- Dynamic typing makes it hard to resolve some names

- Mostly happen in *polymorphic functions*

Solution:

- use a static type system
- use inter-procedural analysis to infer types

Q: What is 'x' ?
A: Uhh…

```
def h(x):
  return x.z
```

Q: Where is 'z' defined?
A: … wherever we defined 'x'

Static Type System for Python

Mostly a usual type system, with two extras: *union* and *dict*

- primitive types
  - int, str, float, bool
- tuple types
  - (int,float), (A, B, C)
- list types
  - [int], [bool], [(int,bool)]
- **dict types**
  - {int => str}, {A => B}
- class types
  - ClassA, ClassB
- **union types**
  - {int | str}, {A | B | C}
- recursive types
  - #1(int, 1), #2(int -> 2)
- function types
  - int -> bool, A -> B

2. Problems with Control-Flow Graph

- CFGs are tricky to build for high-order programs
- Attempts to build CFGs have led to complications and limitations in *control-flow analysis*
  - Shivers 1988, 1991
    - build CFG after CPS
  - Might & Shivers 2006,2007
    - solve problems introduced by CFG
  - Vardoulakis & Shivers 2010,2011
    - solve problems introduced by CPS

```
def g(f,x):
   return f(x)

def h1(x):
   return x+1

def h2(x):
   return x+2
```

2. Problems with Control-Flow Graph

- CFGs are tricky to build for high-order programs

- Attempts to build CFGs have led to complications and limitations in *control-flow analysis*
  - Shivers 1988, 1991
    - build CFG after CPS
  - Might & Shivers 2006,2007
    - solve problems introduced by CFG
  - Vardoulakis & Shivers 2010,2011
    - solve problems introduced by CPS

Where is the CFG target?

```
def g(f,x):
   return f(x)

def h1(x):
   return x+1

def h2(x):
   return x+2
```

2. Problems with Control-Flow Graph

- CFGs are tricky to build for high-order programs
- Attempts to
  complicatio
  control-flow
  - Shivers 1988, 1991
    - build CFG after CPS
  - Might & Shivers 2006,2007
    - solve problems introduced by CFG
  - Vardoulakis & Shivers 2010,2011
    - solve problems introduced by CPS

Solution:

- Don’t CPS the input program
- Don’t try constructing the CFG
- Use direct-style, recursive *abstract interpreter*

```
def g(f,x):
  return f(x)

def h1(x):
  return x+1

def h2(x):
  return x+2
```

Where is the CFG target?

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
   x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
    x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
    x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
    x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

- create “abstract objects” at constructor calls

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
    x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

- create “abstract objects” at constructor calls

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
    x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

- create “abstract objects” at constructor calls
- Actually change the abstract objects when fields are created

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
   x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

- create “abstract objects” at constructor calls
- Actually change the abstract objects when fields are created

3. Problems with Dynamic Field Creation/ Deletion

```
class A:
   x = 1
obj = A()
obj.y = 3
print obj.x, obj.y
```

Solution:

- create “abstract objects” at constructor calls
- Actually change the abstract objects when fields are created
- Classes are not affect by the change

4. Problems with More Powerful Dynamic Features

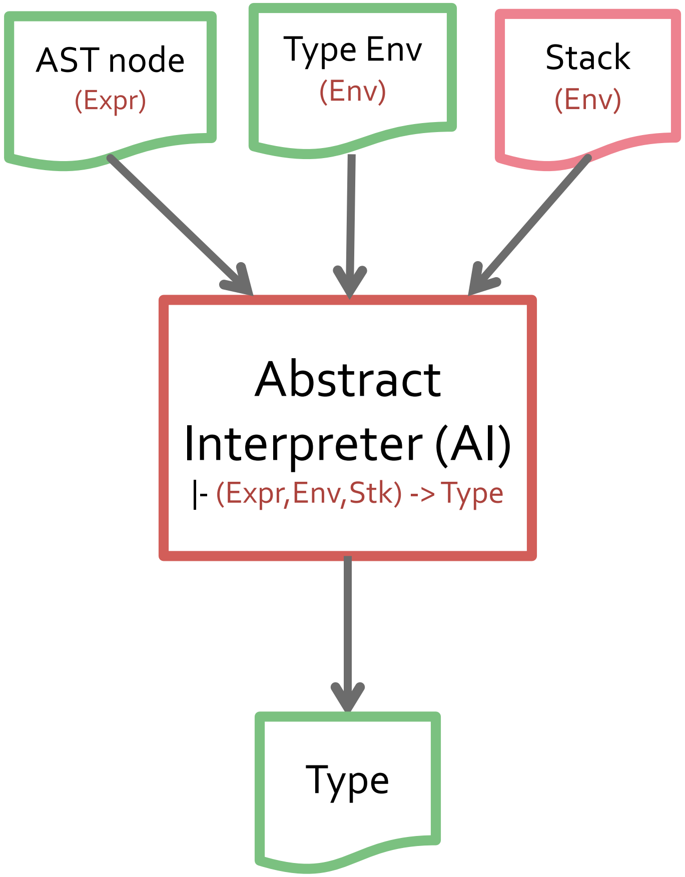
- direct operations on __dict__ (e.g. setattr, delattr, ...)
- dynamic object reparenting
- import hacks
- eval
- ...

4. Problems with More Powerful Dynamic Features

- direct operations on __dict__ (e.g. setattr, delattr, …)
- dynamic object reparenting
- import hacks
- eval
- …

Solution:
"Python Style Guide"

Overall Structure of Analysis



AST node (Expr)

Type Env (Env)

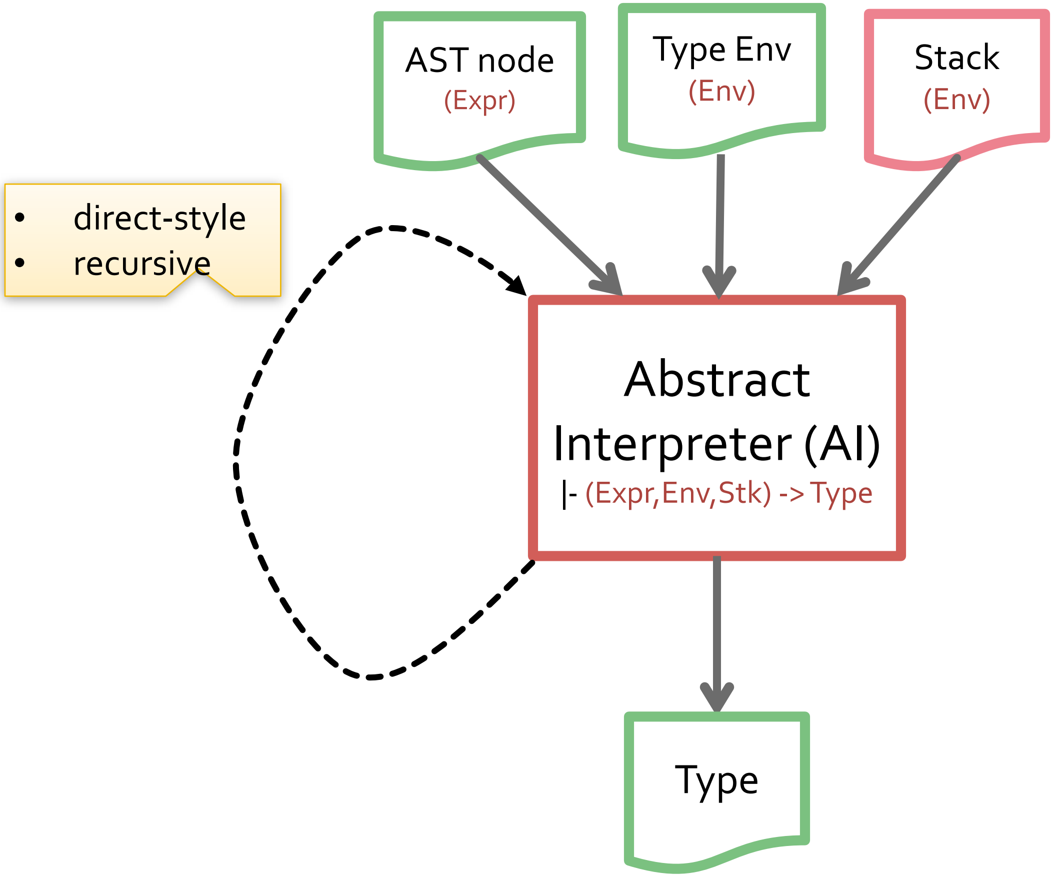
Stack (Env)

Abstract Interpreter (AI)

|- (Expr,Env,Stk) -> Type

Type

Overall Structure of Analysis



- direct-style
- recursive

AST node
(Expr)

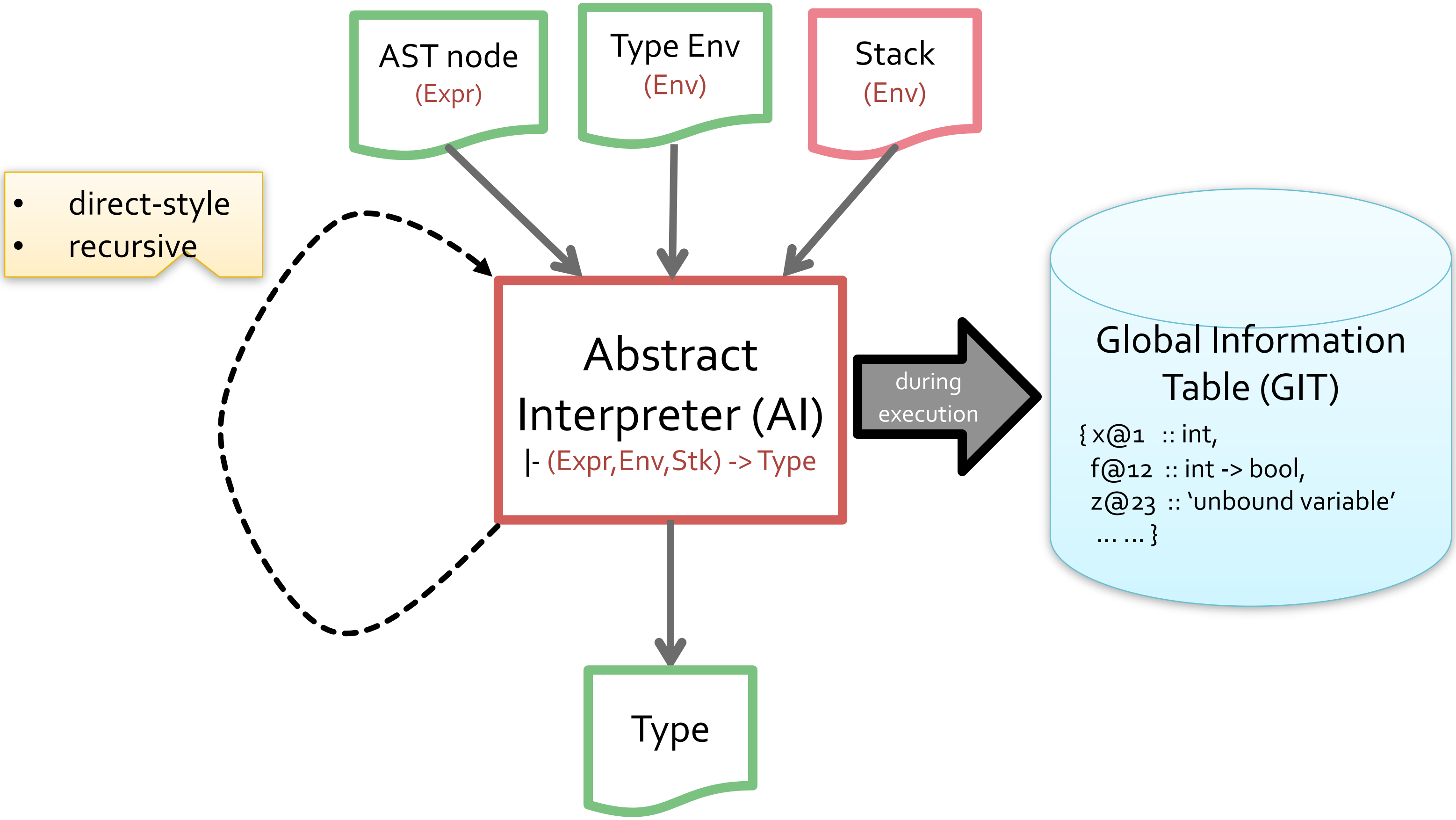
Type Env
(Env)

Stack
(Env)

Abstract Interpreter (AI)
|- (Expr,Env,Stk) -> Type

Type

Overall Structure of Analysis


AST node
(Expr)
Type Env
(Env)
Stack
(Env)
Abstract Interpreter (AI)
|- (Expr,Env,Stk) -> Type
during execution
Global Information Table (GIT)
Type

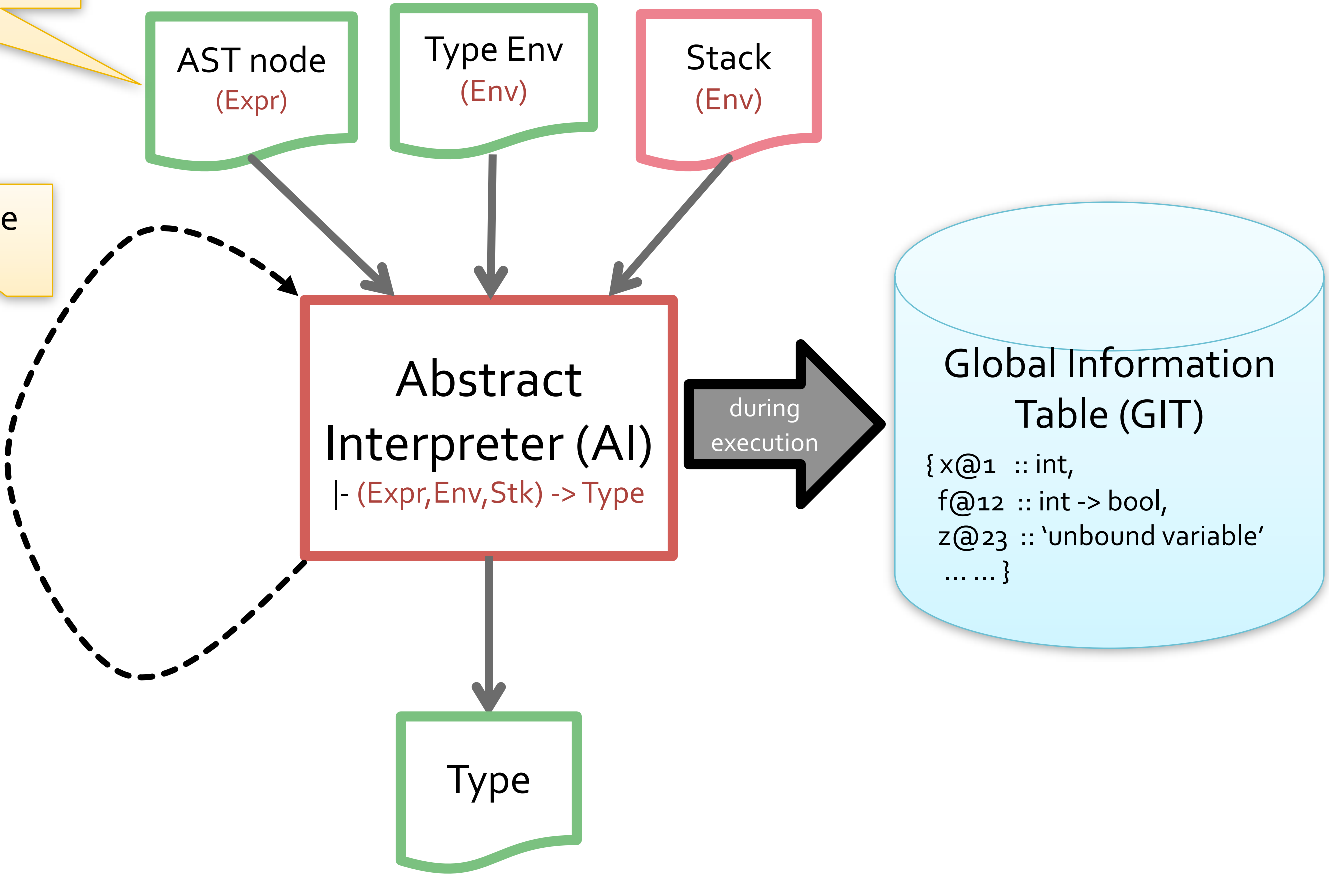
- direct-style
- recursive

```
{ x@1  :: int,
  f@12  :: int -> bool,
  z@23  :: 'unbound variable'
  … … }
```

Overall Structure of Analysis

- direct-style (not CPSed)
- no CFG generated
- keep it "clean" (don't record anything on it)

- direct-style
- recursive



AST node
(Expr)

Type Env
(Env)

Stack
(Env)

Abstract Interpreter (AI)
|- (Expr,Env,Stk) -> Type

during execution

Global Information Table (GIT)

{ x@1 :: int,
f@12 :: int -> bool,
z@23 :: 'unbound variable'
… … }

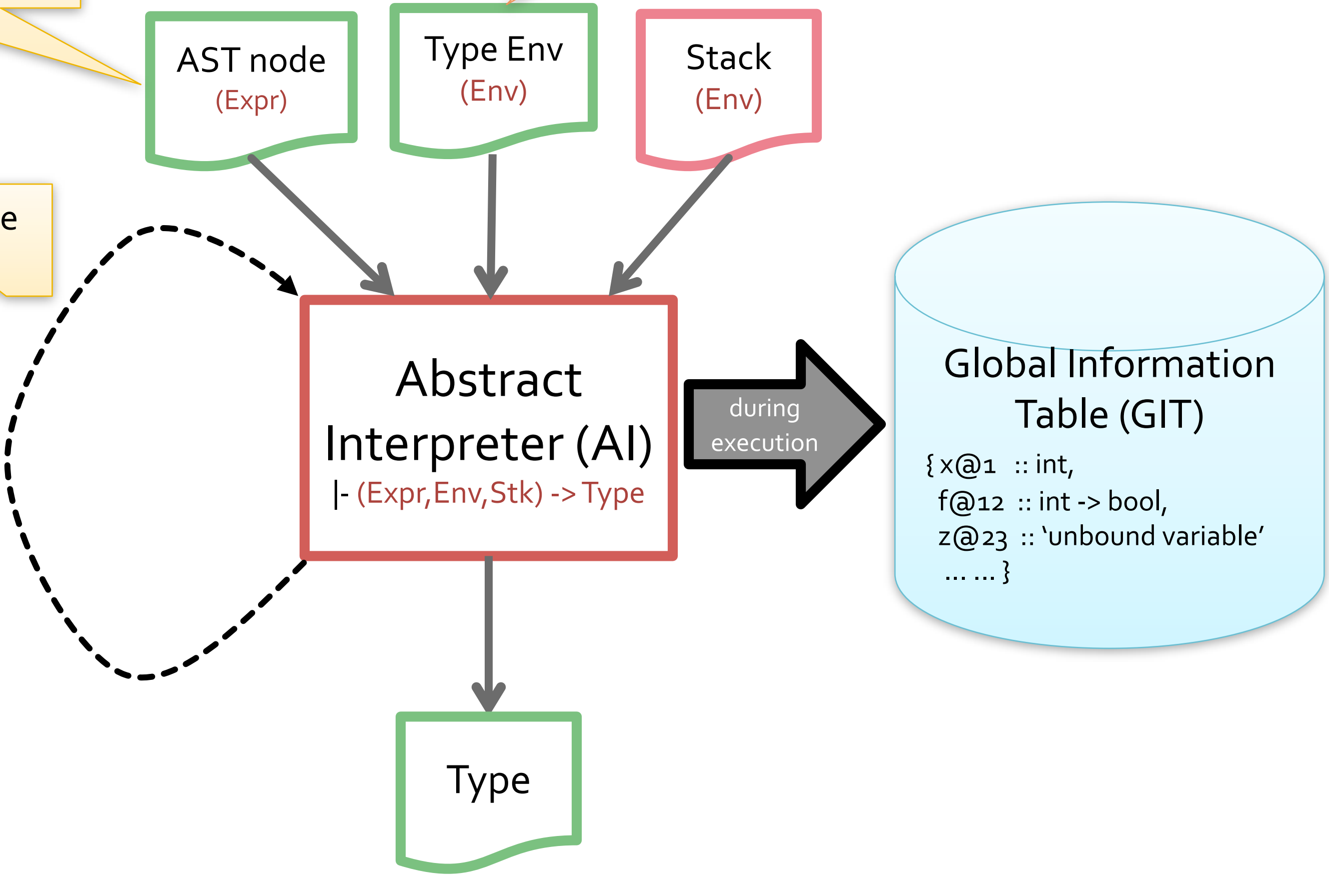
Type

Overall Structure of Analysis

- direct-style (not CPSed)
- no CFG generated
- keep it "clean" (don't record anything on it)

purely functional data structures

- direct-style
- recursive


AST node
(Expr)
Type Env
(Env)
Stack
(Env)
Abstract Interpreter (AI)
|- (Expr,Env,Stk) -> Type
during execution
Global Information Table (GIT)
{ x@1 :: int,
f@12 :: int -> bool,
z@23 :: 'unbound variable'
… … }
Type

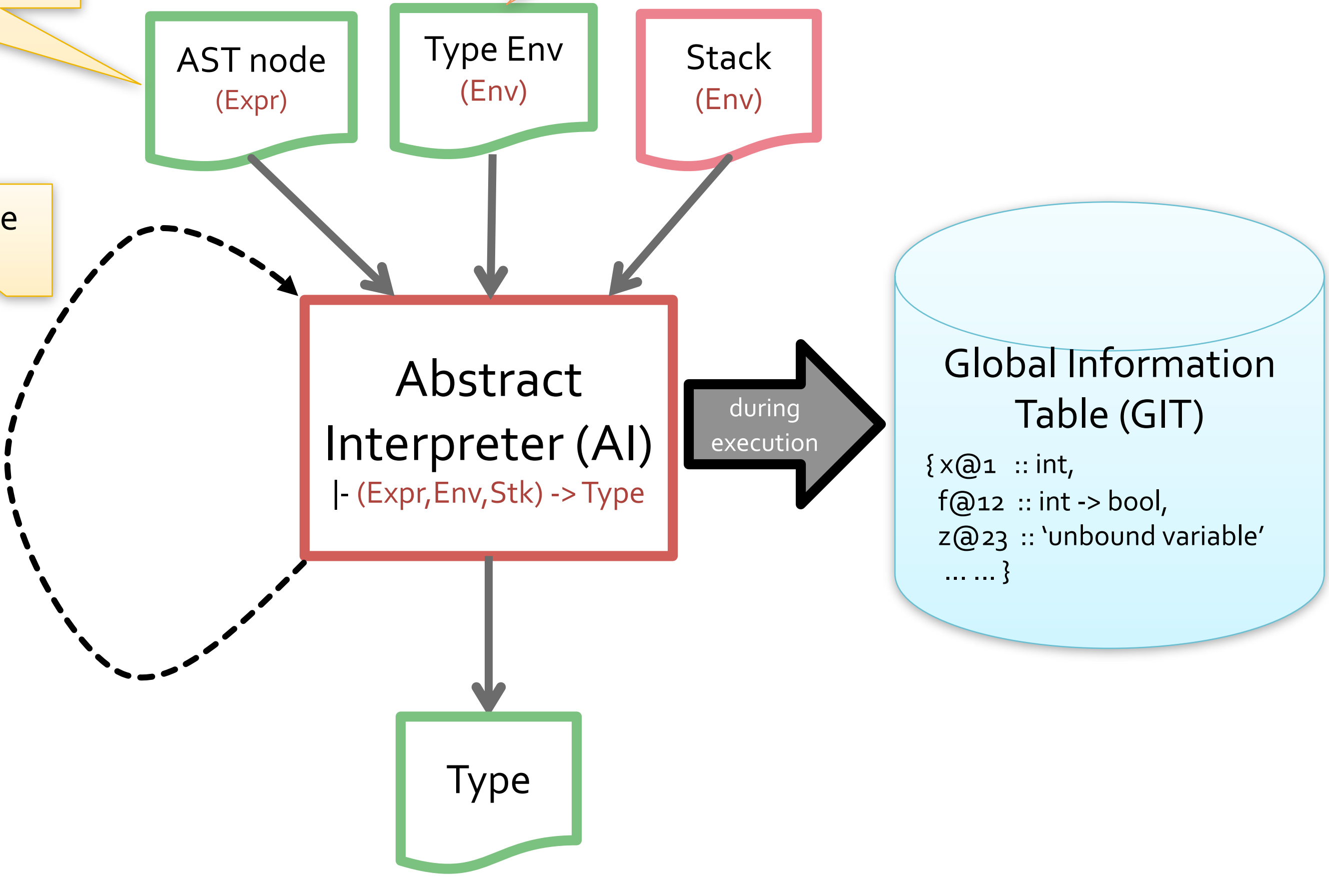
Overall Structure o alysis

- direct-style (not CPSed)
- no CFG generated
- keep it “clean” (don’t record anything on it)

purely functional data structures

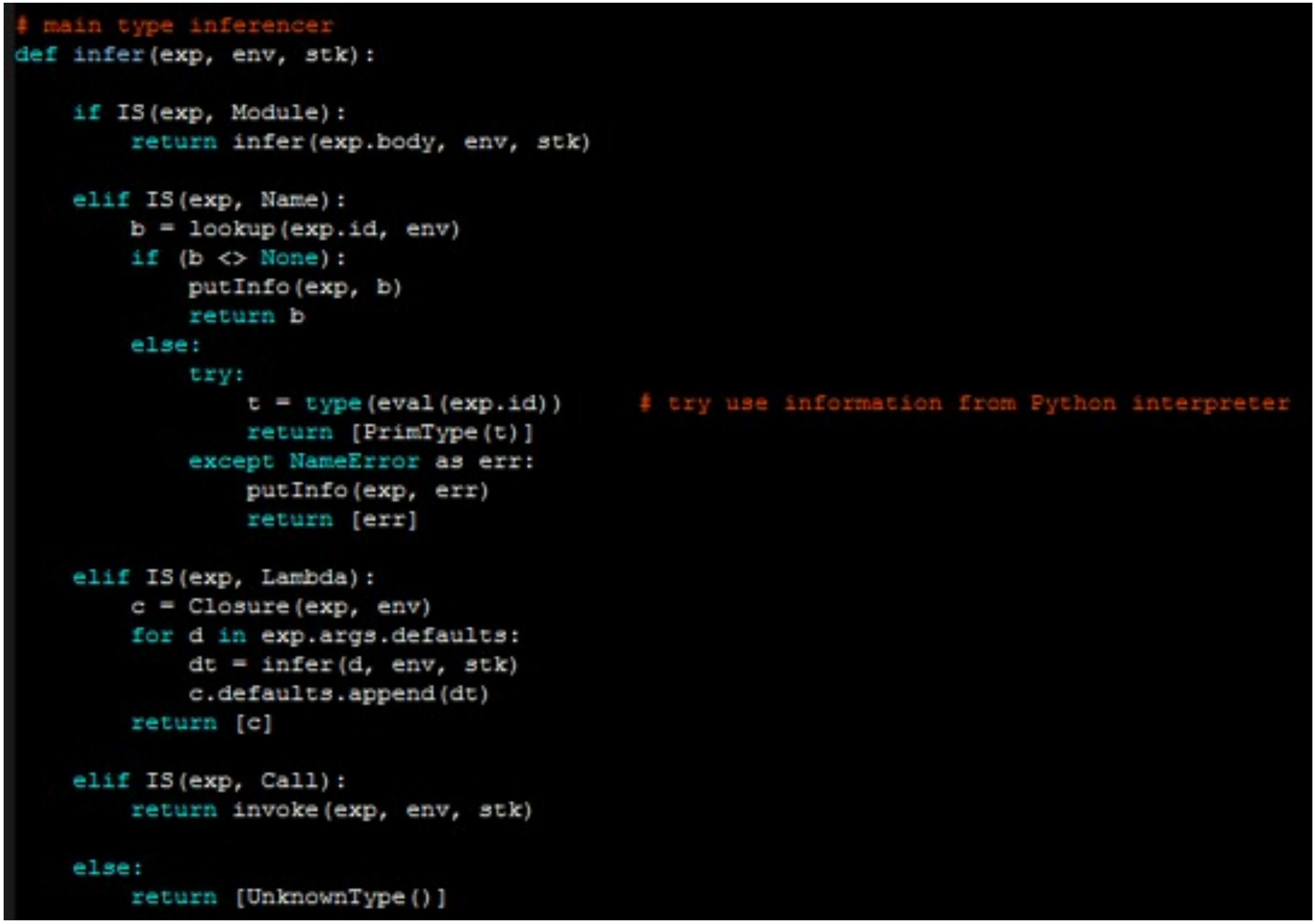
- direct-style
- recursive

loop detection (using stack)


AST node
(Expr)
Type Env
(Env)
Stack
(Env)
Abstract Interpreter (AI)
|- (Expr,Env,Stk) -> Type
during execution
Global Information Table (GIT)
Type

```
{ x@1  :: int,
  f@12  :: int -> bool,
  z@23  :: ‘unbound variable’
  … … }
```

Actual Code of Main Interpreter



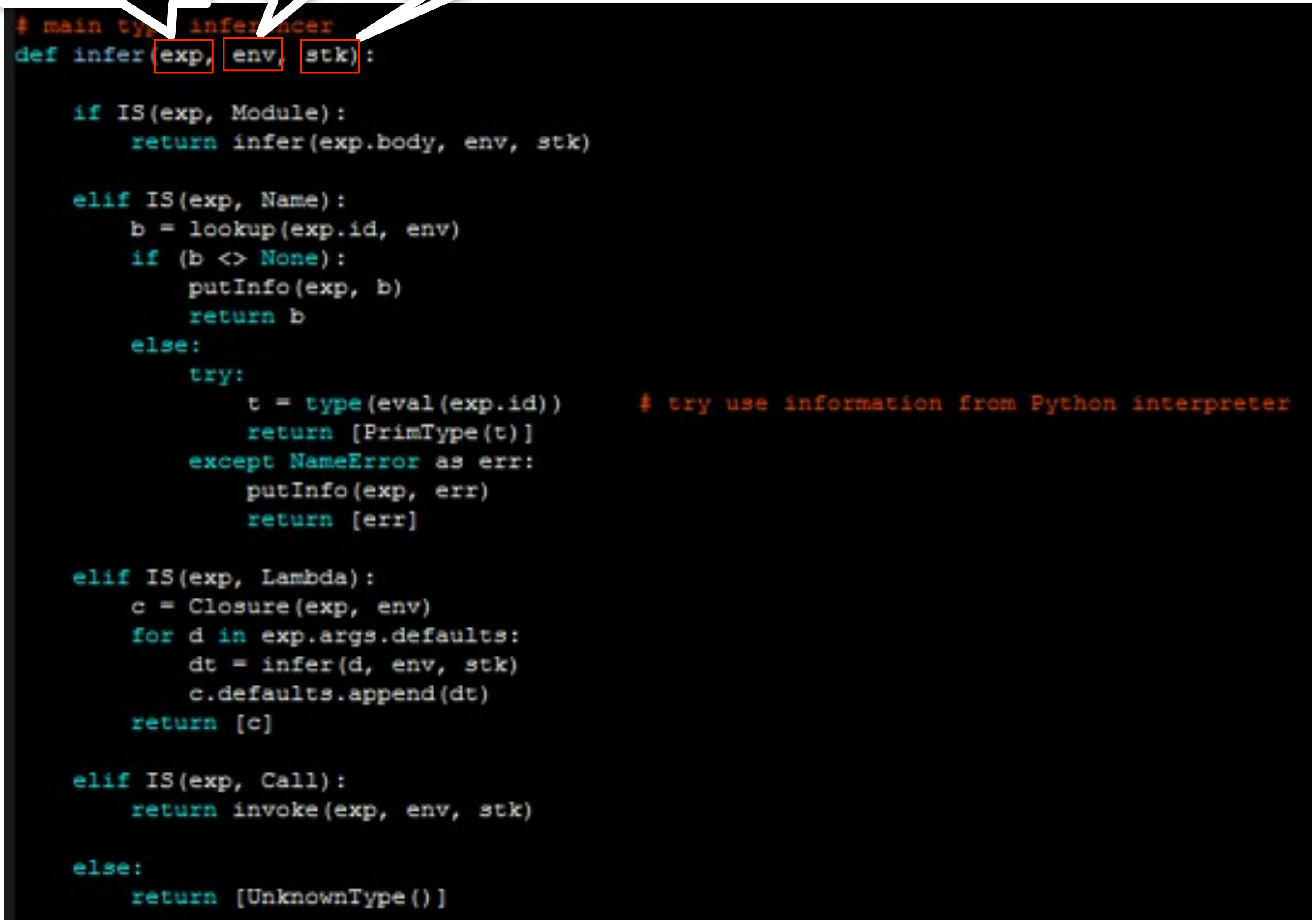
```
# main type inferencer
def infer(exp, env, stk):

    if IS(exp, Module):
        return infer(exp.body, env, stk)

    elif IS(exp, Name):
        b = lookup(exp.id, env)
        if (b <> None):
            putInfo(exp, b)
            return b
        else:
            try:
                t = type(eval(exp.id))      # try use information from Python interpreter
                return [PrimType(t)]
            except NameError as err:
                putInfo(exp, err)
                return [err]

    elif IS(exp, Lambda):
        c = Closure(exp, env)
        for d in exp.args.defaults:
            dt = infer(d, env, stk)
            c.defaults.append(dt)
        return [c]

    elif IS(exp, Call):
        return invoke(exp, env, stk)

    else:
        return [UnknownType()]
```

Actual Code of Main I

input expression

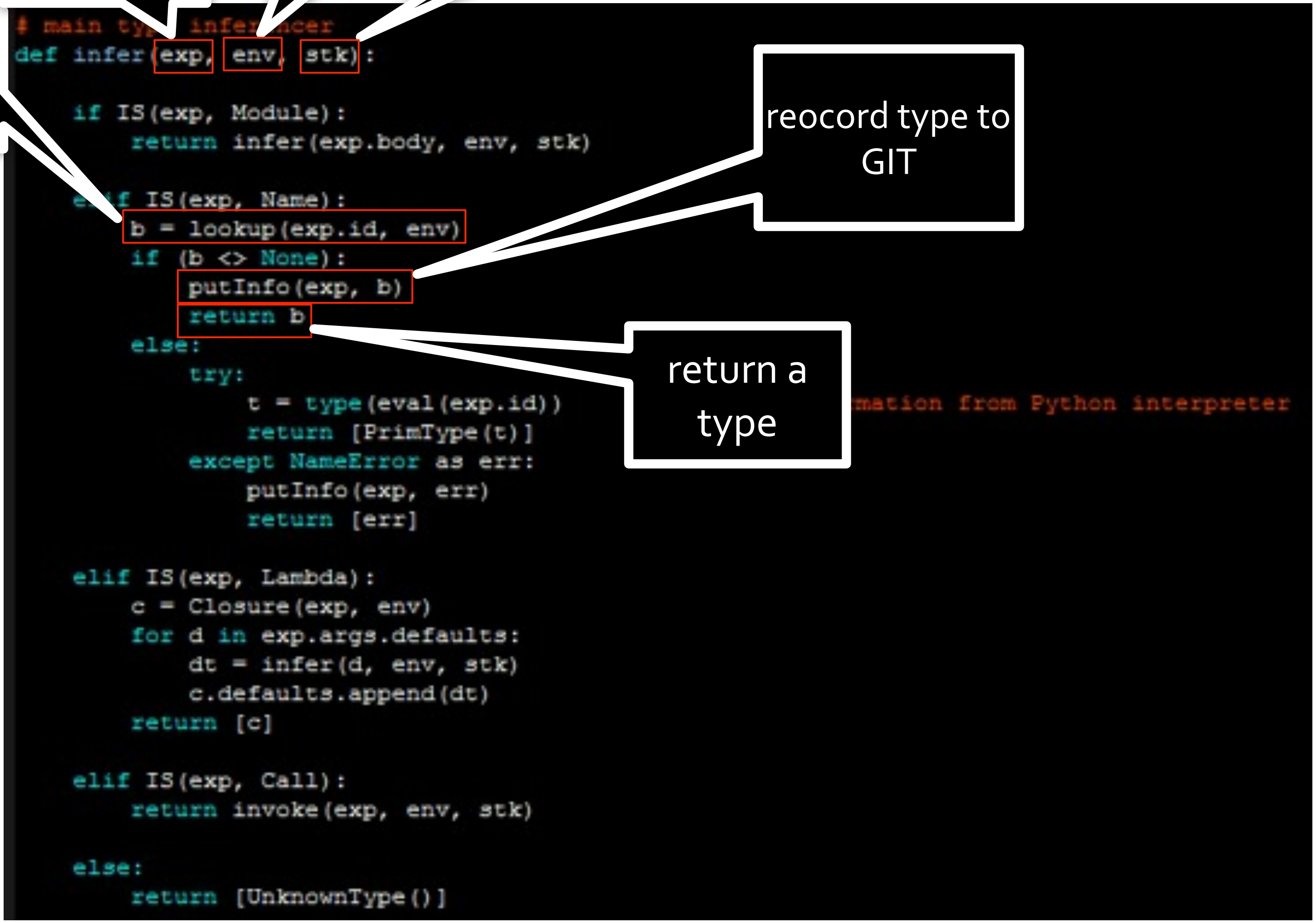
type environment

stack of nodes on path (recursion detection)



```
# main type inferencer
def infer(exp, env, stk):

    if IS(exp, Module):
        return infer(exp.body, env, stk)

    elif IS(exp, Name):
        b = lookup(exp.id, env)
        if (b <> None):
            putInfo(exp, b)
            return b
        else:
            try:
                t = type(eval(exp.id))      # try use information from Python interpreter
                return [PrimType(t)]
            except NameError as err:
                putInfo(exp, err)
                return [err]

    elif IS(exp, Lambda):
        c = Closure(exp, env)
        for d in exp.args.defaults:
            dt = infer(d, env, stk)
            c.defaults.append(dt)
        return [c]

    elif IS(exp, Call):
        return invoke(exp, env, stk)

    else:
        return [UnknownType()]
```

Actual Code of Main I



```
# main type inferencer
def infer(exp, env, stk):

    if IS(exp, Module):
        return infer(exp.body, env, stk)

    elif IS(exp, Name):
        b = lookup(exp.id, env)
        if (b <> None):
            putInfo(exp, b)
            return b
        else:
            try:
                t = type(eval(exp.id))          mation from Python interpreter
                return [PrimType(t)]
            except NameError as err:
                putInfo(exp, err)
                return [err]

    elif IS(exp, Lambda):
        c = Closure(exp, env)
        for d in exp.args.defaults:
            dt = infer(d, env, stk)
            c.defaults.append(dt)
        return [c]

    elif IS(exp, Call):
        return invoke(exp, env, stk)

    else:
        return [UnknownType()]
```

input expression

type environment

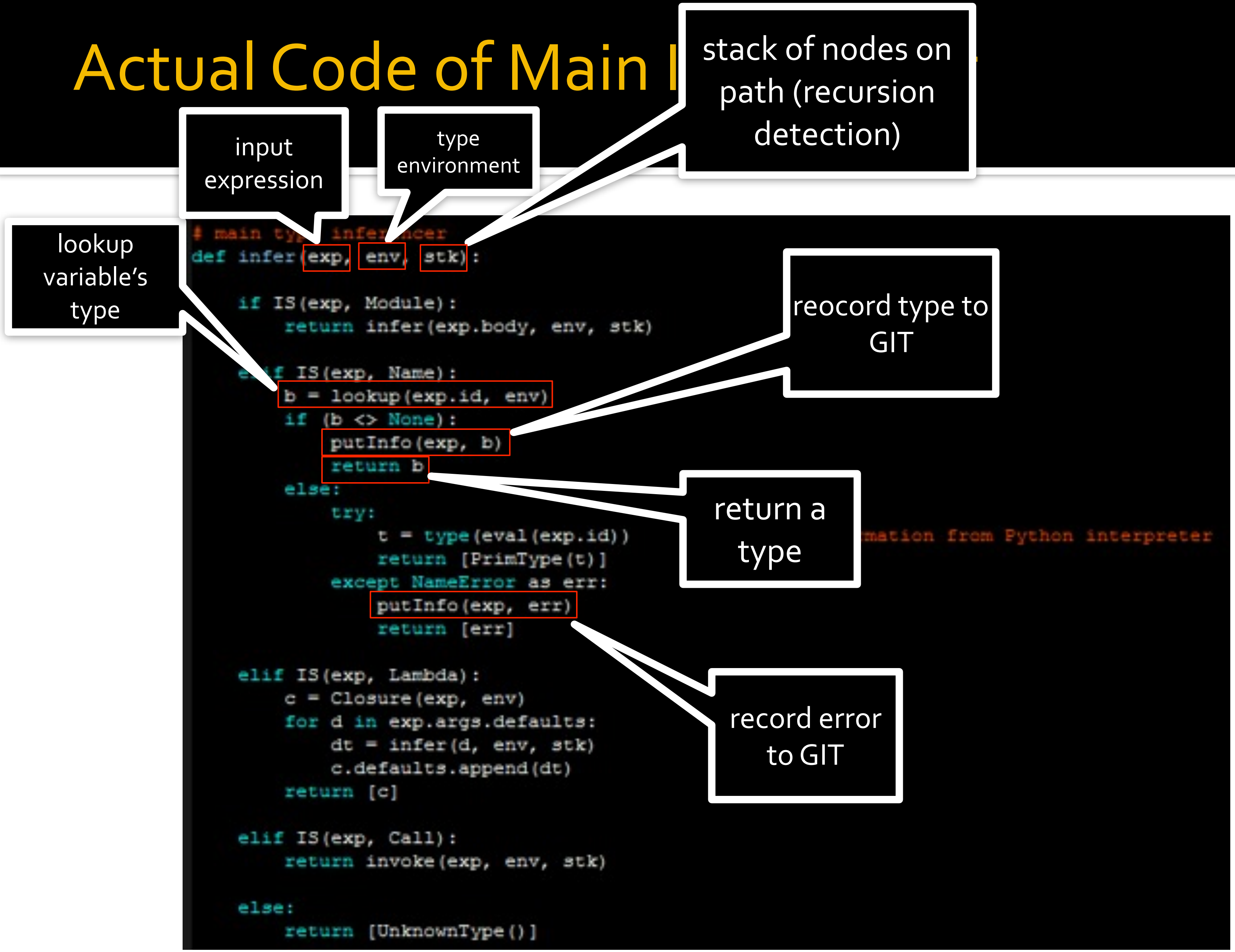
stack of nodes on path (recursion detection)

lookup variable's type

reocord type to GIT

return a type

Actual Code of Main I



```
# main ty    infer   ncer
def infer(exp, env, stk):

    if IS(exp, Module):
        return infer(exp.body, env, stk)

    elif IS(exp, Name):
        b = lookup(exp.id, env)
        if (b <> None):
            putInfo(exp, b)
            return b
        else:
            try:
                t = type(eval(exp.id))                 mation from Python interpreter
                return [PrimType(t)]
            except NameError as err:
                putInfo(exp, err)
                return [err]

    elif IS(exp, Lambda):
        c = Closure(exp, env)
        for d in exp.args.defaults:
            dt = infer(d, env, stk)
            c.defaults.append(dt)
        return [c]

    elif IS(exp, Call):
        return invoke(exp, env, stk)

    else:
        return [UnknownType()]
```

input expression

type environment

stack of nodes on path (recursion detection)

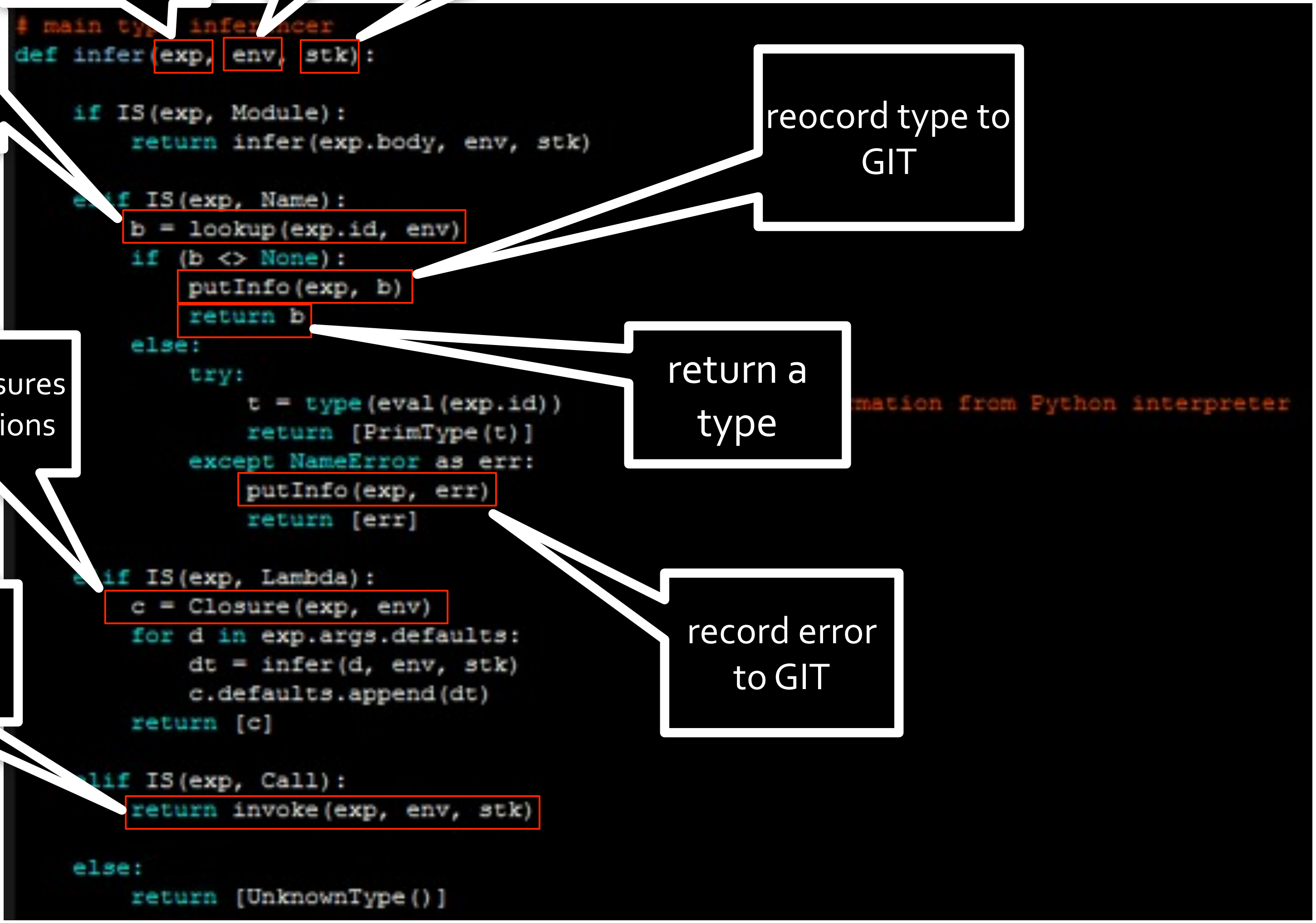
lookup variable's type

reocord type to GIT

return a type

record error to GIT

Actual Code of Main I



```
# main ty    infer  ncer
def infer(exp, env, stk):

    if IS(exp, Module):
        return infer(exp.body, env, stk)

    elif IS(exp, Name):
        b = lookup(exp.id, env)
        if (b <> None):
            putInfo(exp, b)
            return b
        else:
            try:
                t = type(eval(exp.id))                    mation from Python interpreter
                return [PrimType(t)]
            except NameError as err:
                putInfo(exp, err)
                return [err]

    elif IS(exp, Lambda):
        c = Closure(exp, env)
        for d in exp.args.defaults:
            dt = infer(d, env, stk)
            c.defaults.append(dt)
        return [c]

    elif IS(exp, Call):
        return invoke(exp, env, stk)

    else:
        return [UnknownType()]
```

stack of nodes on path (recursion detection)

input expression

type environment

lookup variable's type

reocord type to GIT

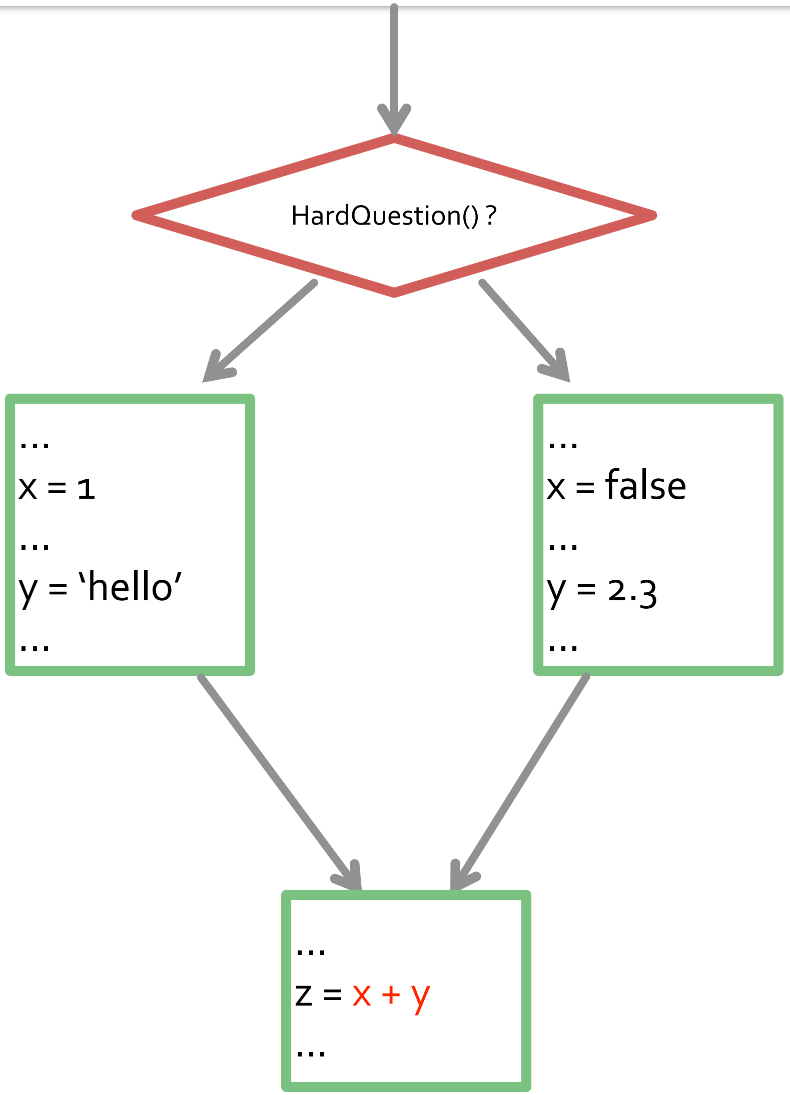
return a type

make closures for functions

record error to GIT

invoke function (closure)

“Multiple-Worlds Model”



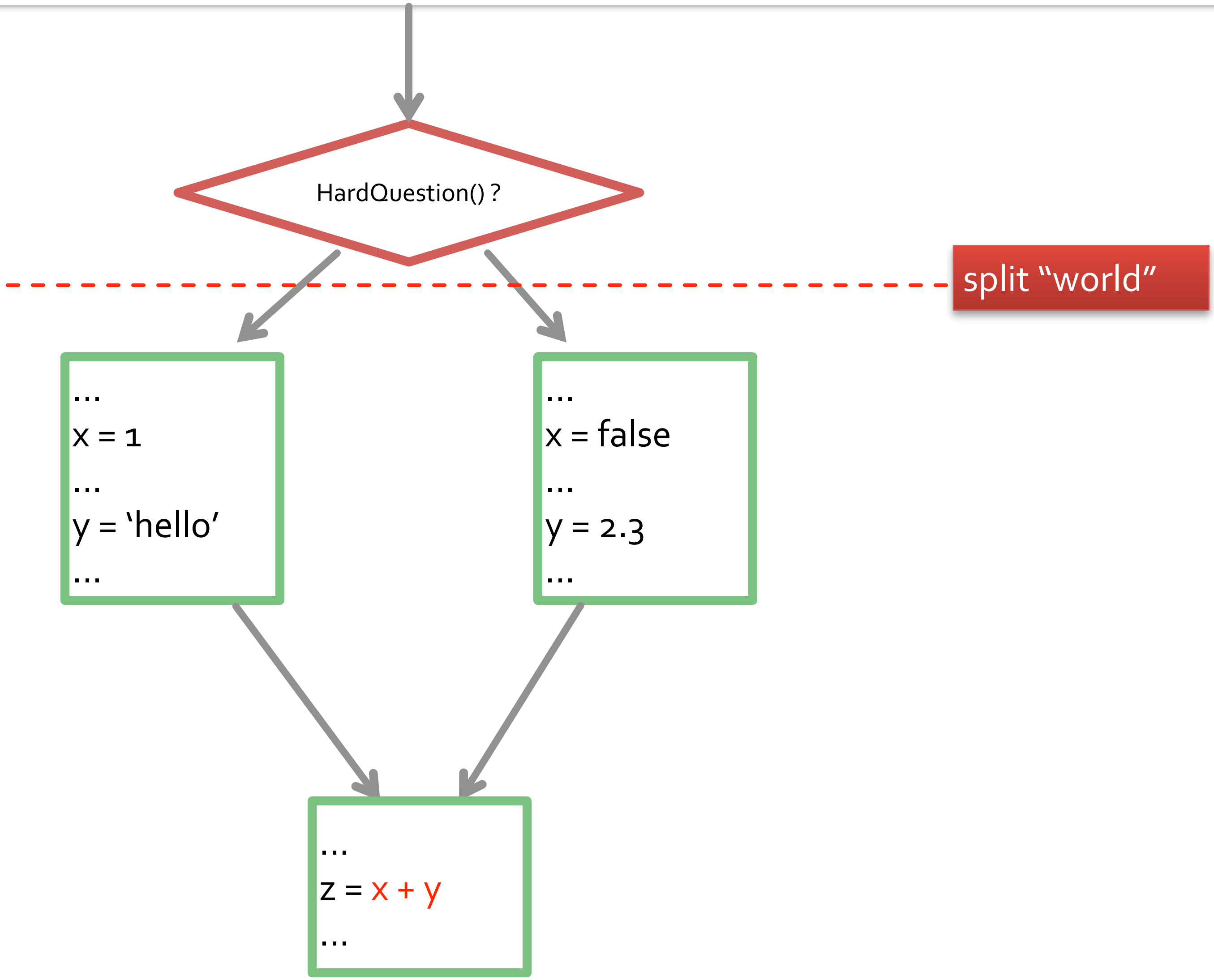
HardQuestion() ?

```
…
x = 1
…
y = ‘hello’
…
```

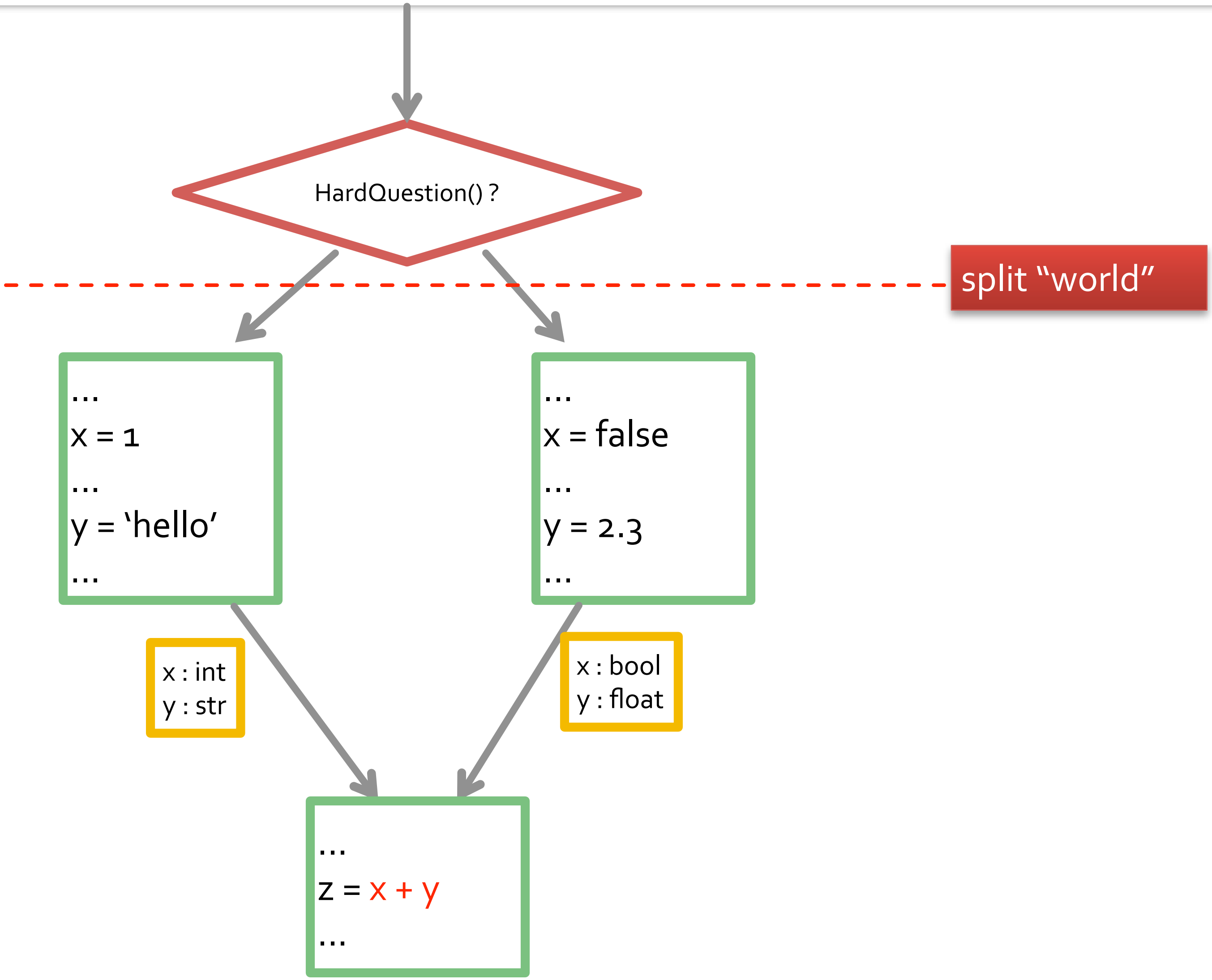
```
…
x = false
…
y = 2.3
…
```

```
…
z = x + y
…
```

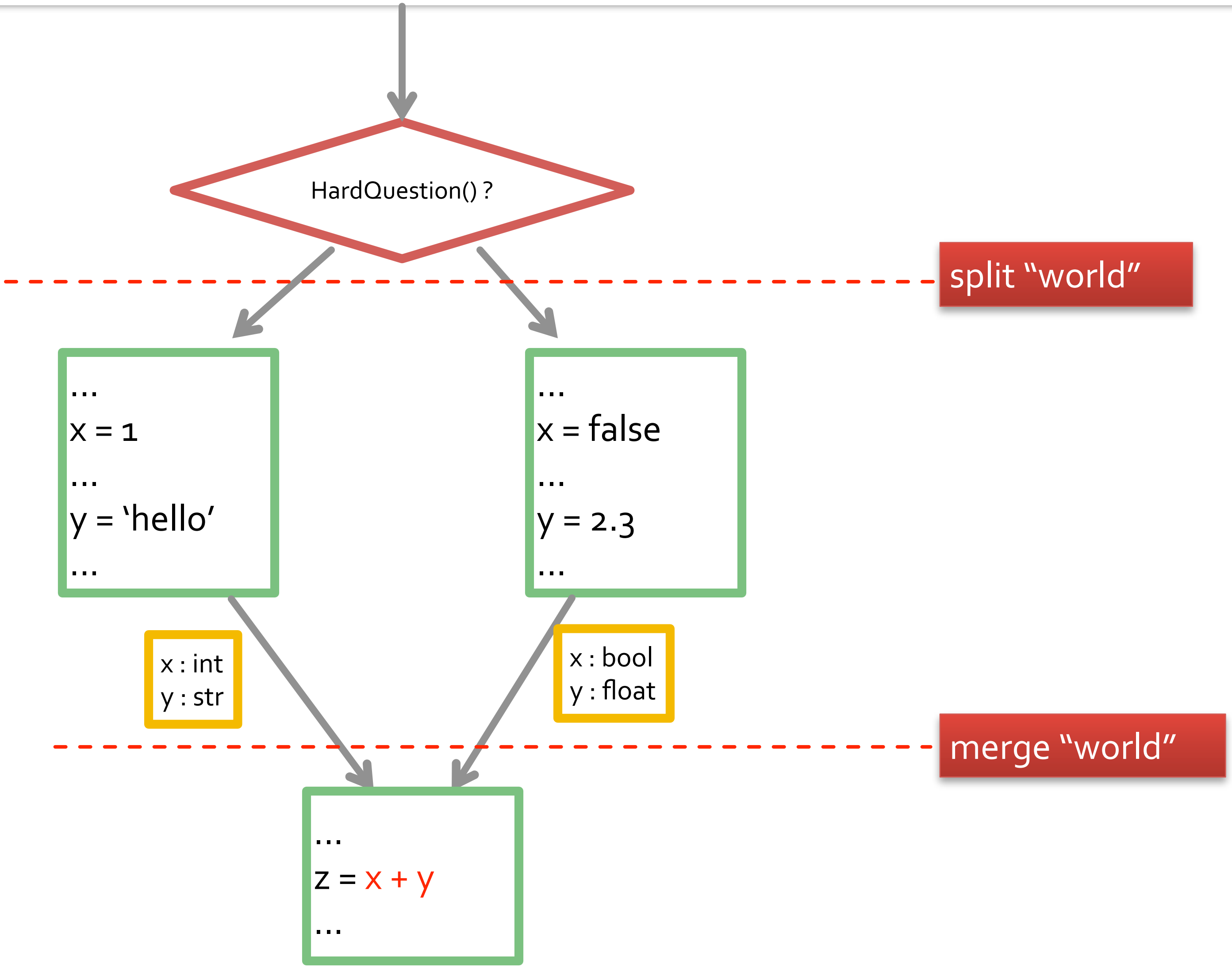
“Multiple-Worlds Model”



“Multiple-Worlds Model”



“Multiple-Worlds Model”



HardQuestion() ?

split “world”

```
…
x = 1
…
y = ‘hello’
…
```

x : int
y : str

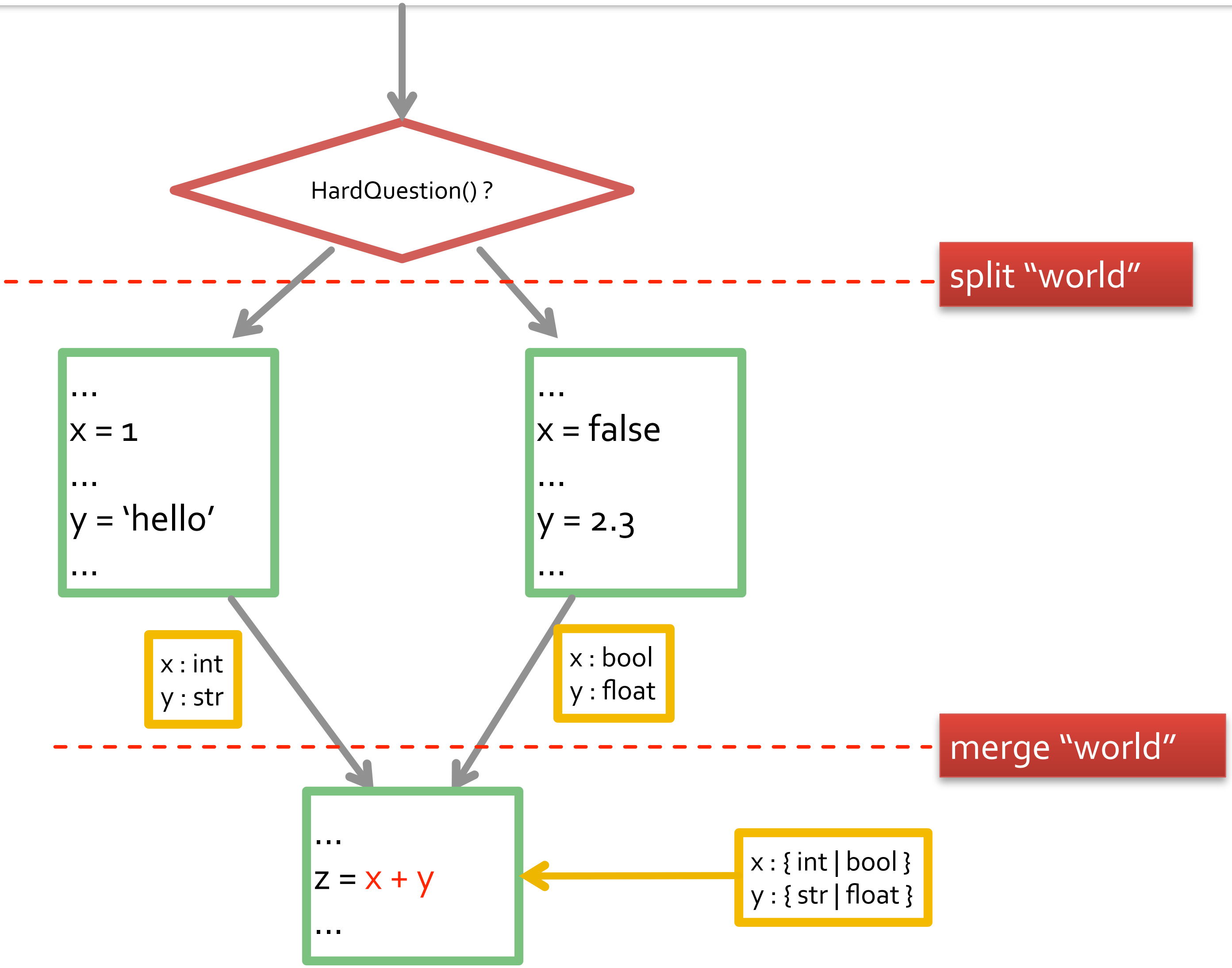
```
…
x = false
…
y = 2.3
…
```

x : bool
y : float

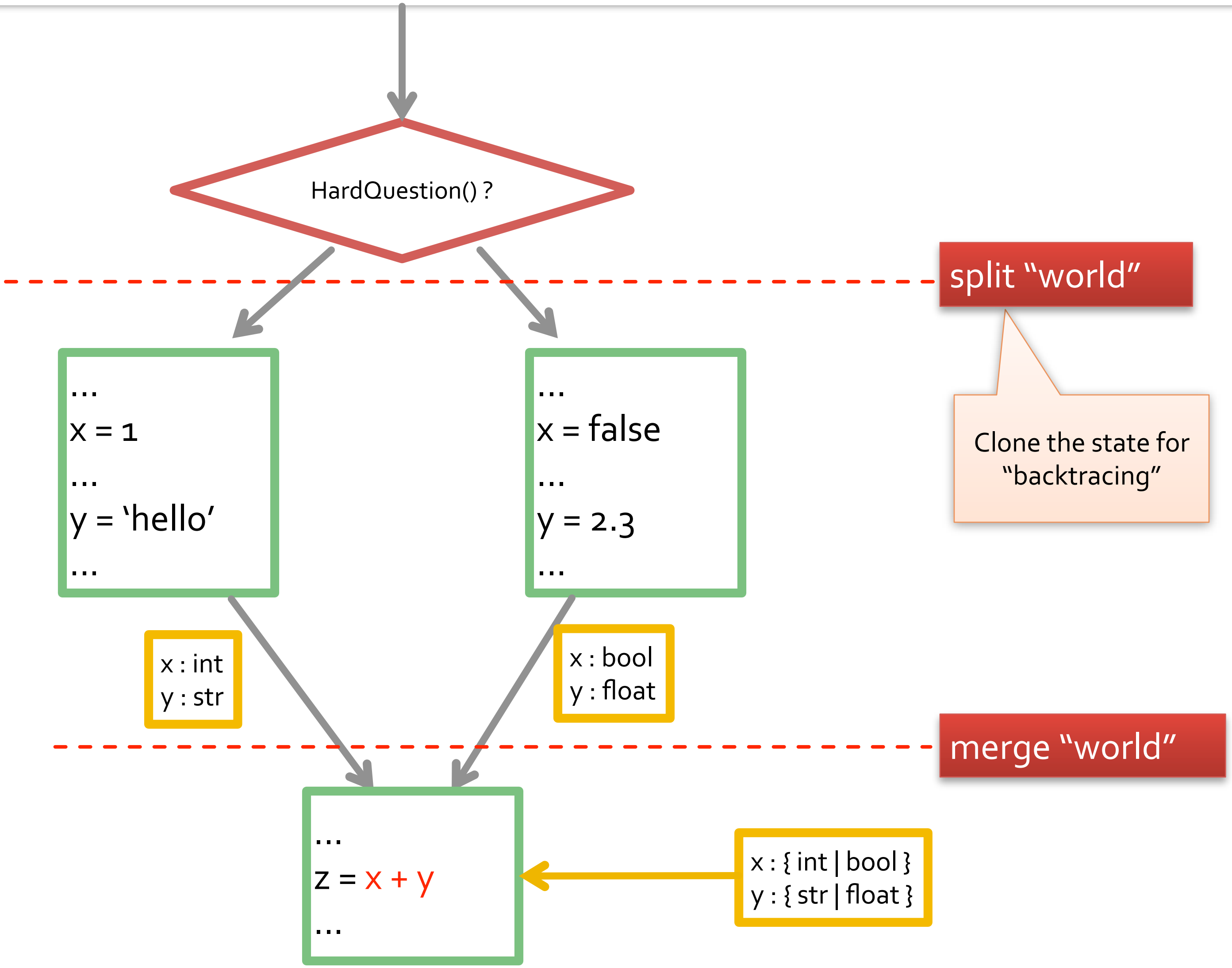
merge “world”

```
…
z = x + y
…
```

“Multiple-Worlds Model”



“Multiple-Worlds Model”



Recursion Detection (1)

```
def fact(n):
   if (n == 0):
      return 1
   else:
      return n * fact(n-1)

fact(5)
```

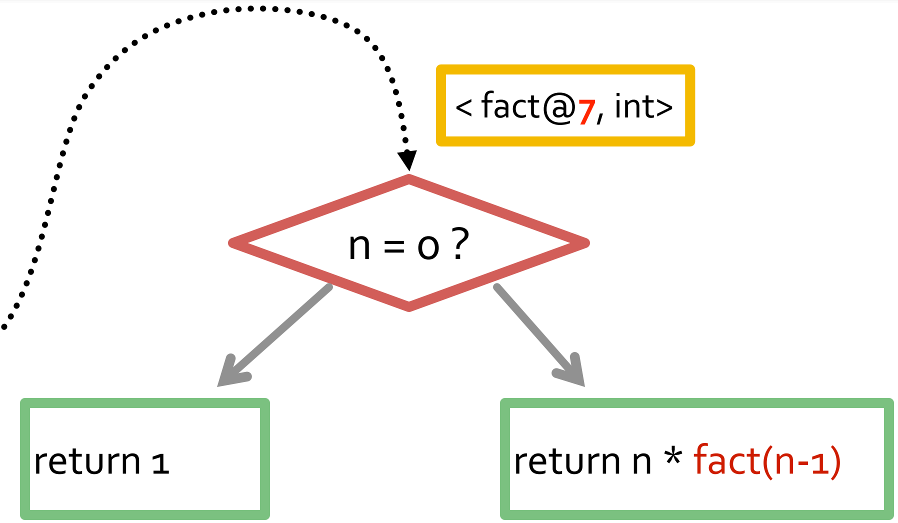
Recursion Detection (1)

```
def fact(n):
   if (n == 0):
      return 1
   else:
      return n * fact(n-1)

fact(5)
```

Assumption: the same *call site* with the same *argument types* always produces the same *output type (or nontermination)*

Recursion Detection (1)

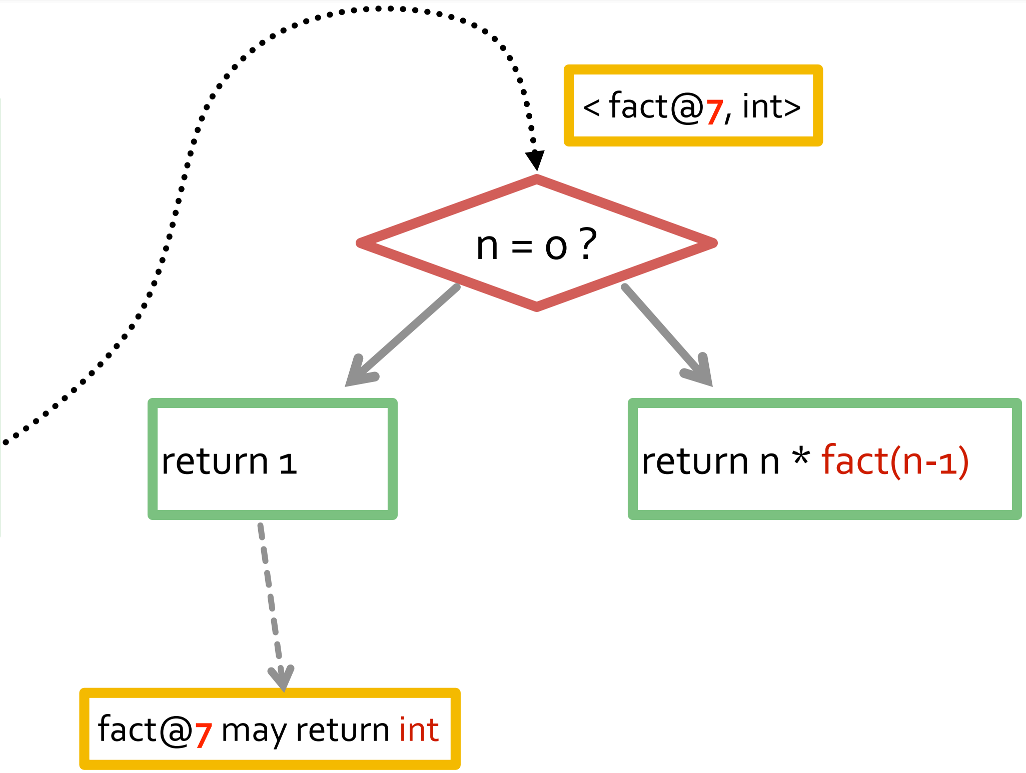
```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



Assumption: the same *call site* with the same *argument types* always produces the same *output type (or nontermination)*

Recursion Detection (1)

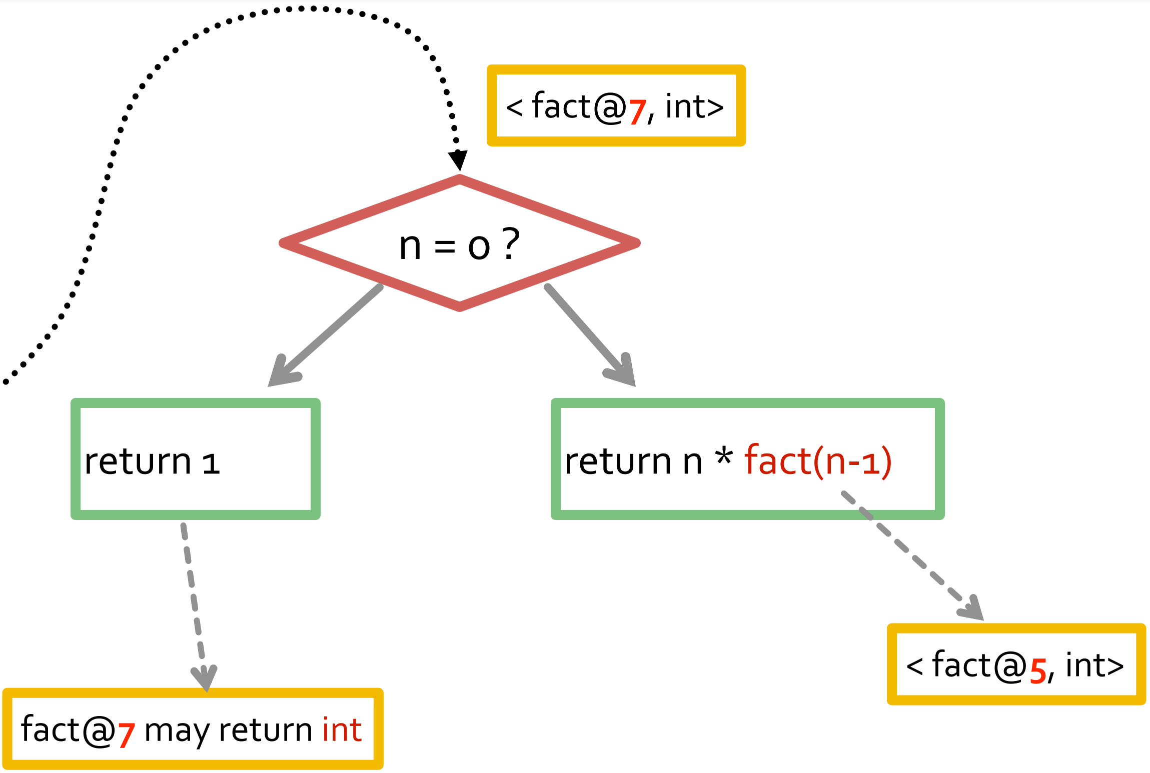
```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



Assumption: the same *call site* with the same *argument types* always produces the same *output type (or nontermination)*

Recursion Detection (1)

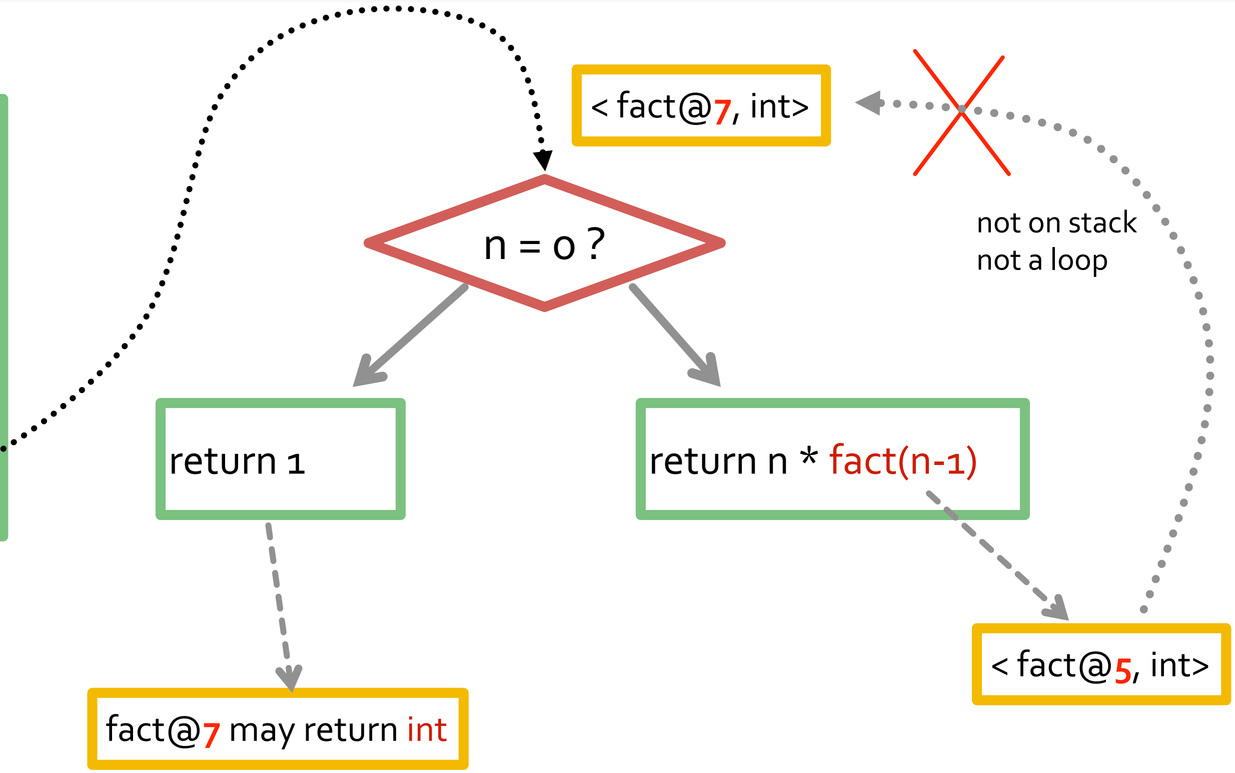
```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



Assumption: the same *call site* with the same *argument types* always produces the same *output type (or nontermination)*

Recursion Detection (1)

```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



< fact@7, int>

n = 0 ?

return 1

return n * fact(n-1)

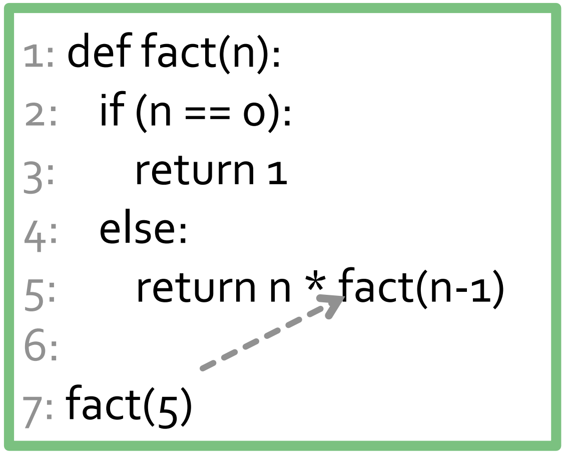
fact@7 may return int

< fact@5, int>

not on stack
not a loop

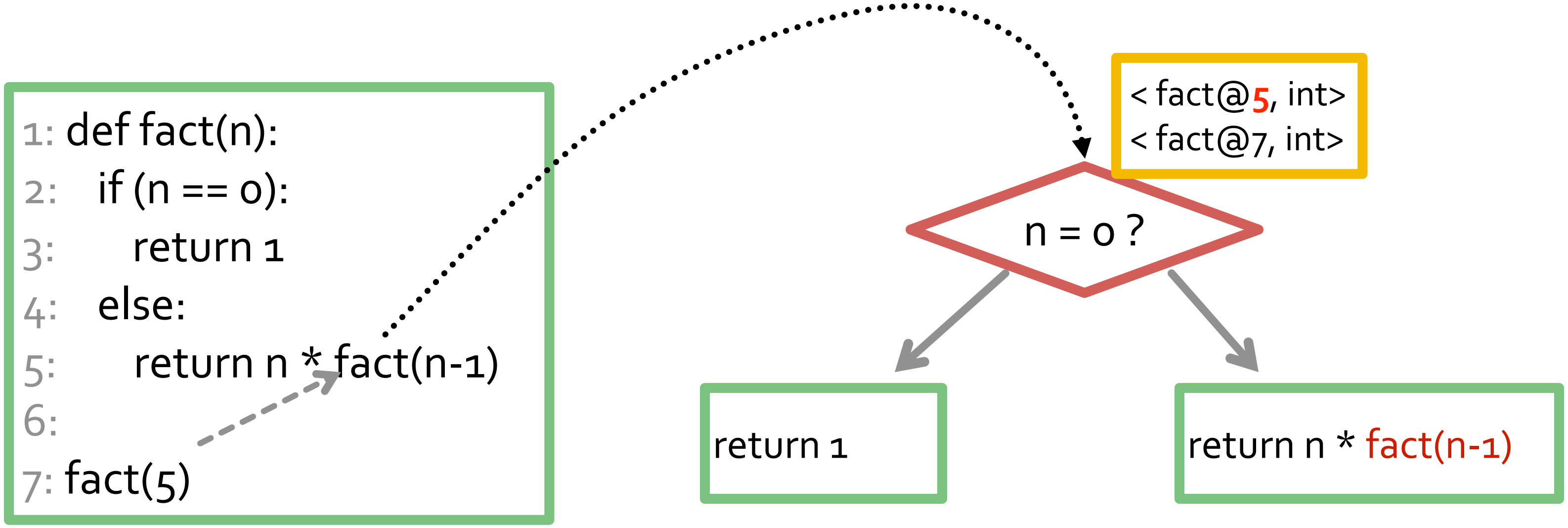
Assumption: the same *call site* with the same *argument types* always produces the same *output type (or nontermination)*

Recursion Detection (2)



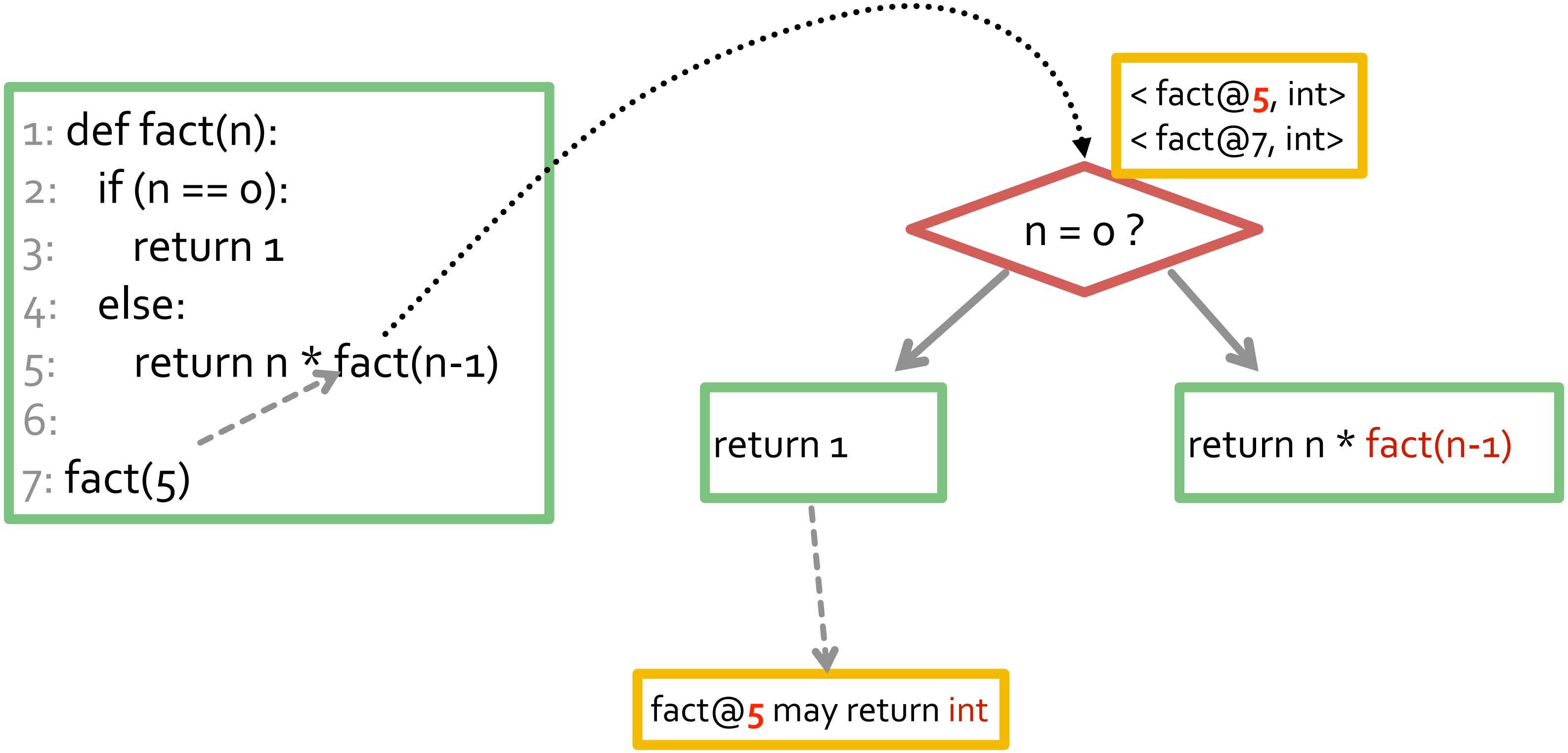
```
def fact(n):
   if (n == 0):
      return 1
   else:
      return n * fact(n-1)

fact(5)
```

Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:        return 1
4:    else:
5:        return n * fact(n-1)
6:
7: fact(5)
```

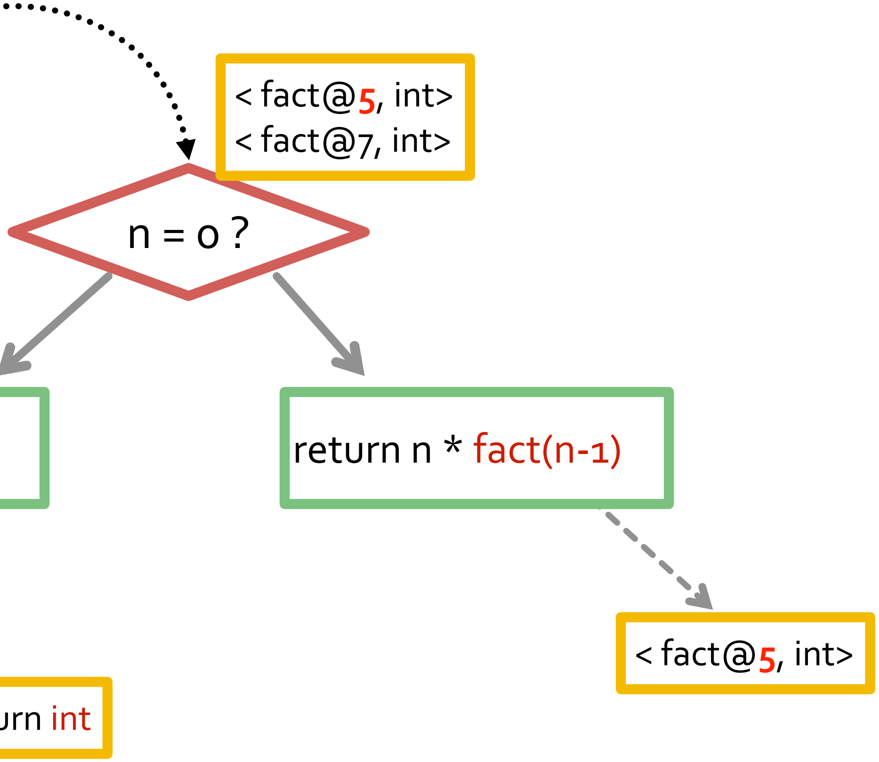
Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

Recursion Detection (2)

```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



< fact@5, int>
< fact@7, int>

n = 0 ?

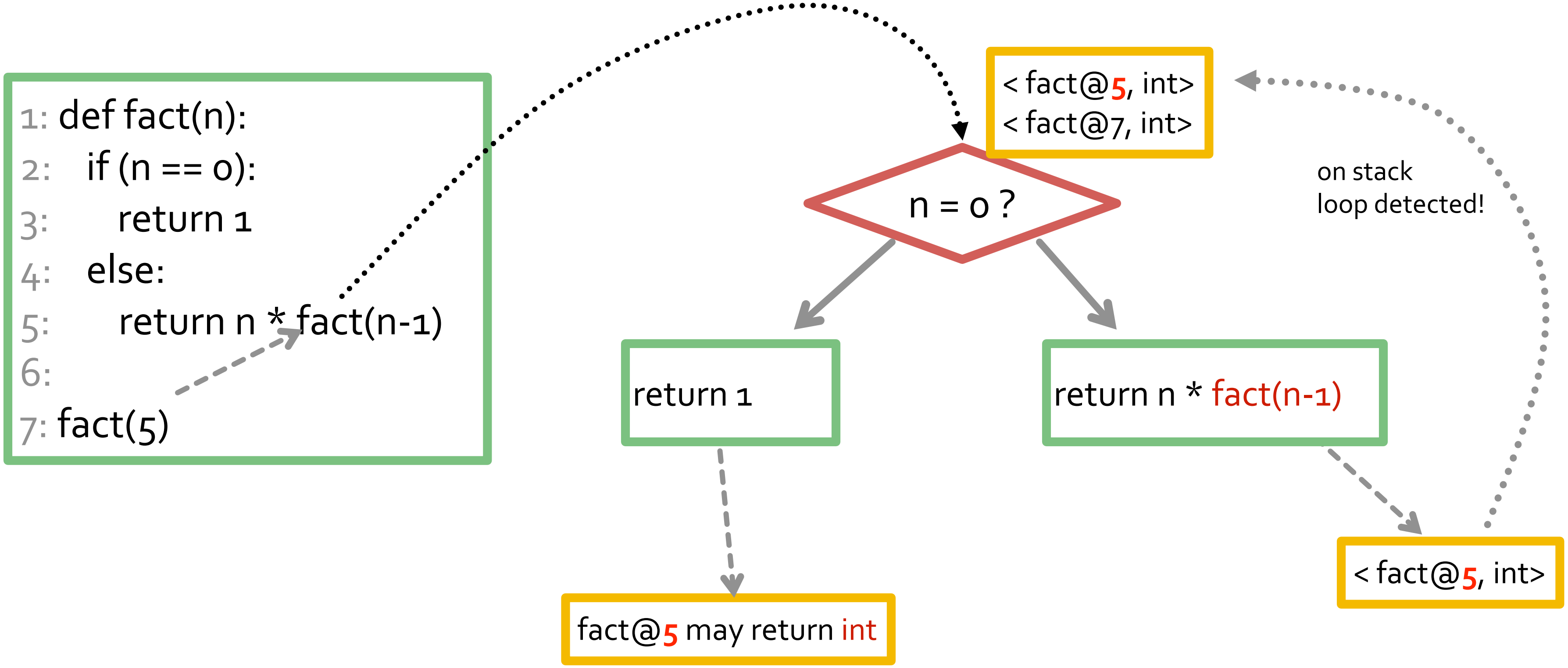
return 1

return n * fact(n-1)

fact@5 may return int

< fact@5, int>

Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

< fact@5, int>
< fact@7, int>

n = 0 ?

return 1

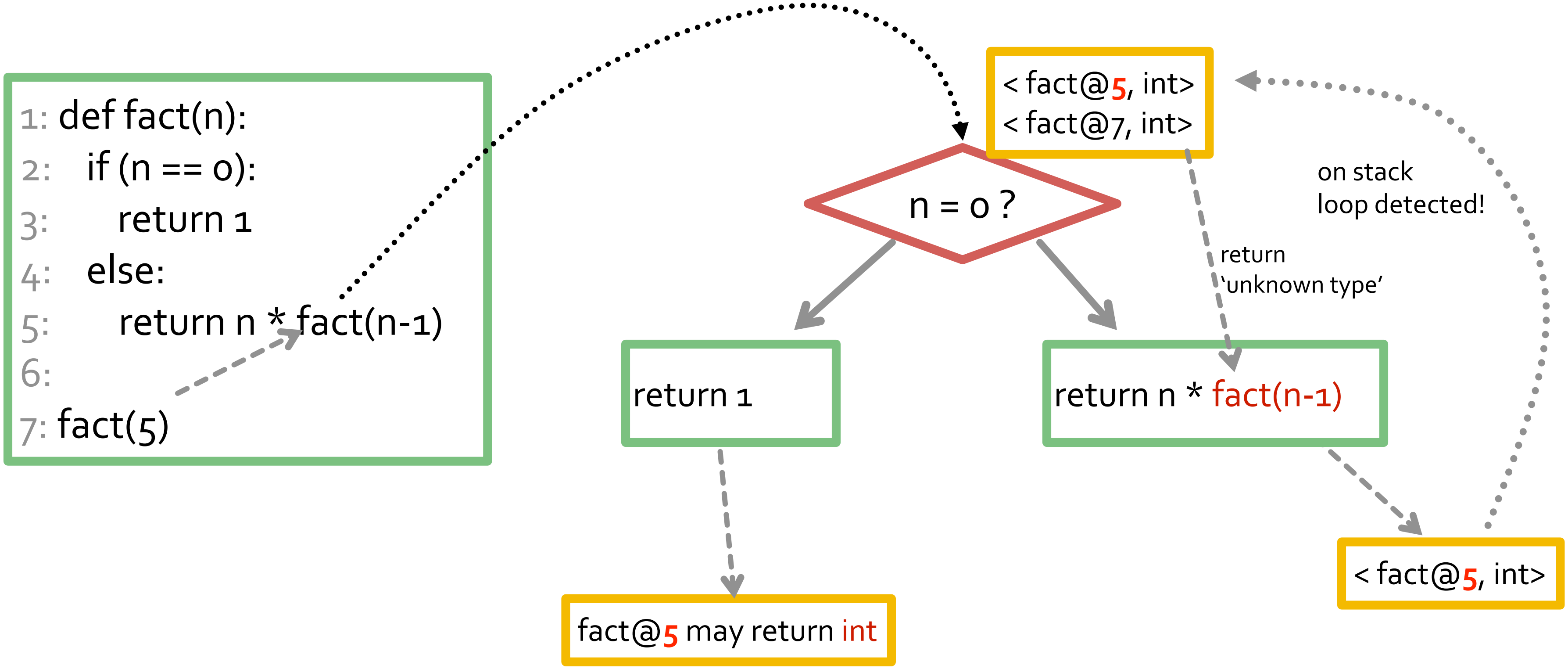
return n * fact(n-1)

fact@5 may return int

< fact@5, int>

on stack
loop detected!

Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

< fact@5, int>
< fact@7, int>

n = 0 ?

return 1

return n * fact(n-1)

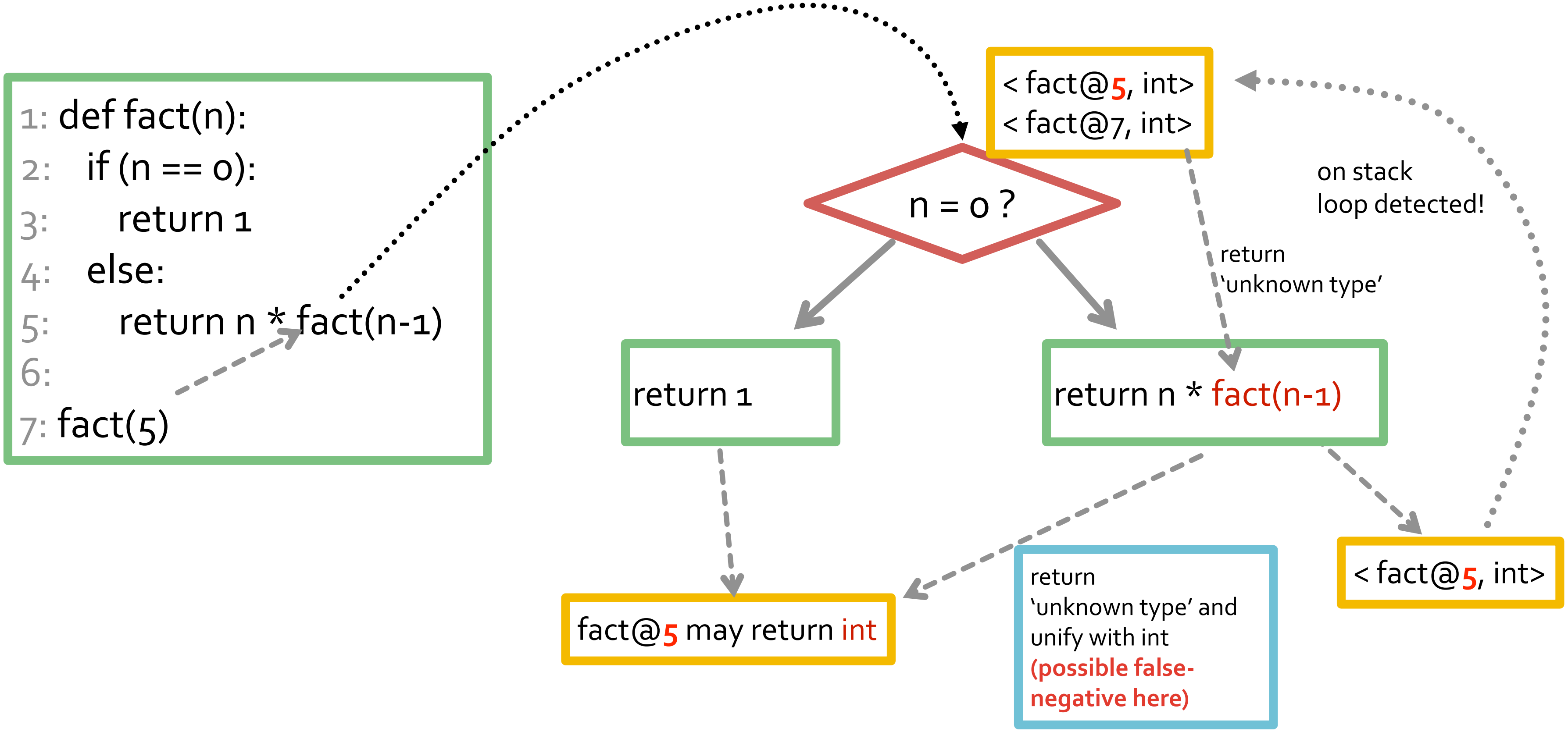
return
'unknown type'

on stack
loop detected!

fact@5 may return int

< fact@5, int>

Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

< fact@5, int>
< fact@7, int>

n = 0 ?

return 1

return n * fact(n-1)

return
'unknown type'

on stack
loop detected!

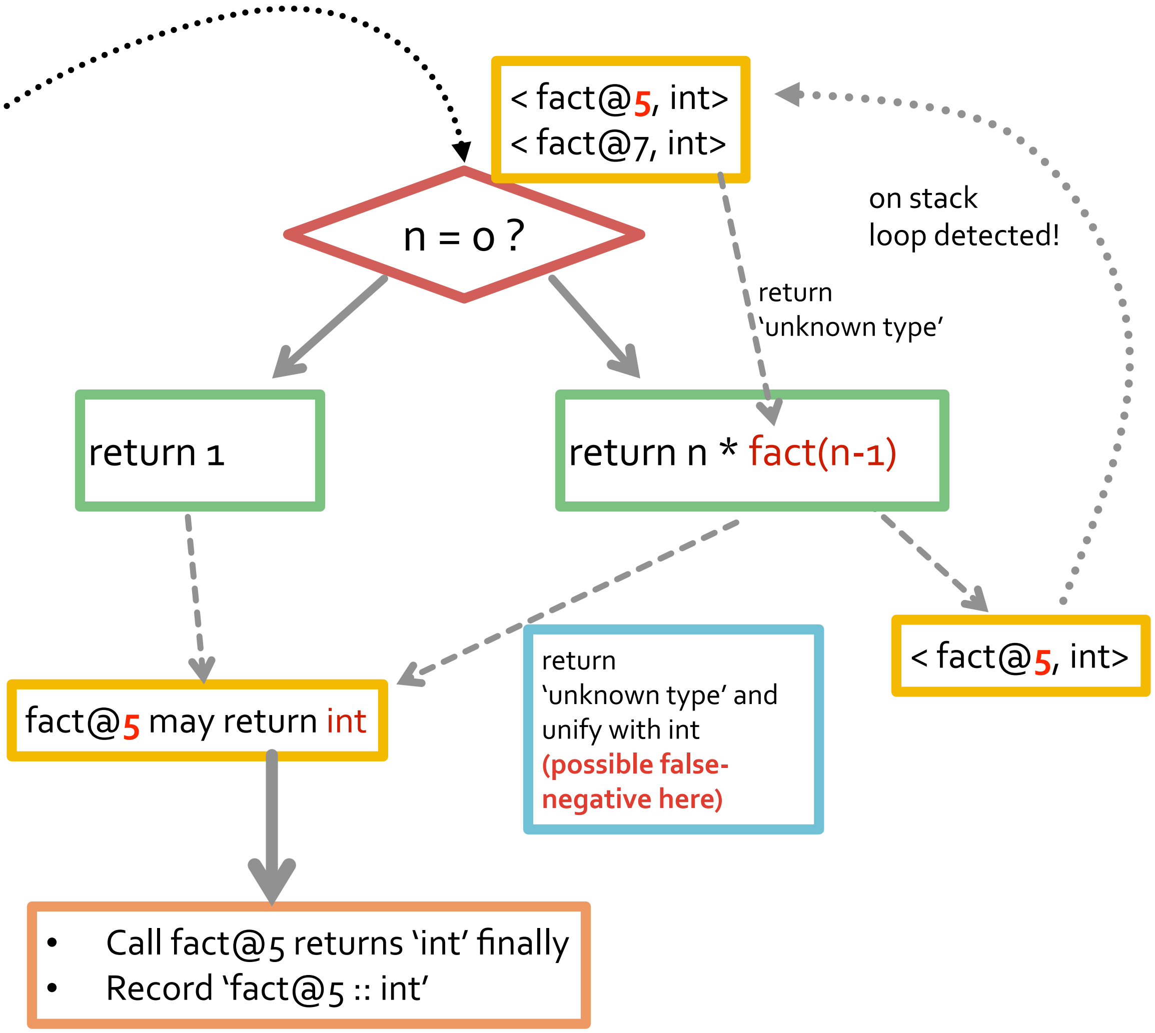
fact@5 may return int

return
'unknown type' and
unify with int
(possible false-negative here)

< fact@5, int>

Recursion Detection (2)

```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```



< fact@5, int>
< fact@7, int>

n = 0 ?

return 1

return n * fact(n-1)

return 'unknown type'

on stack
loop detected!

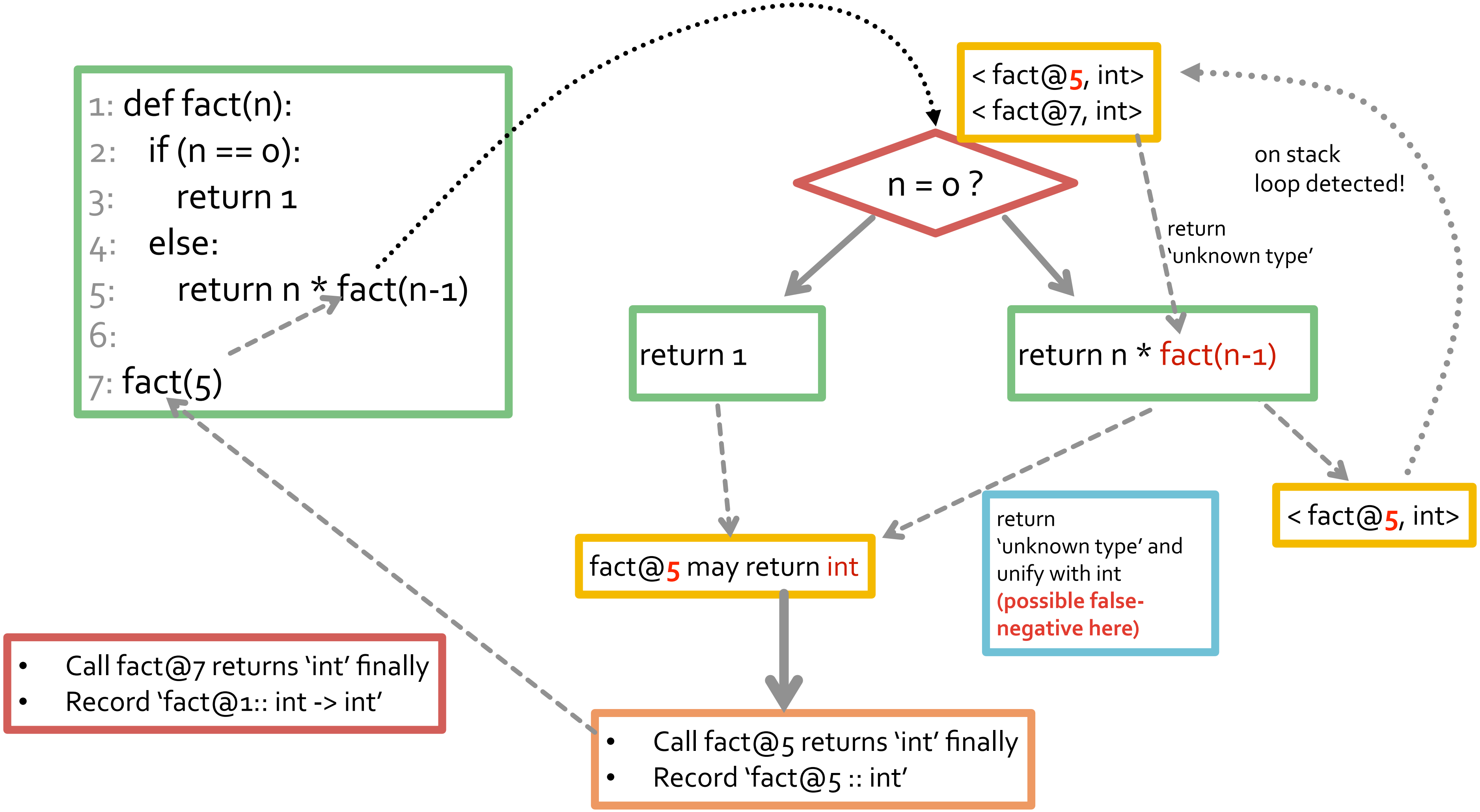
< fact@5, int>

fact@5 may return int

return 'unknown type' and unify with int **(possible false-negative here)**

- Call fact@5 returns 'int' finally
- Record 'fact@5 :: int'

Recursion Detection (2)



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

< fact@5, int>
< fact@7, int>

n = 0 ?

return 1

return n * fact(n-1)

return 'unknown type'

on stack loop detected!

< fact@5, int>

fact@5 may return int

return 'unknown type' and unify with int **(possible false-negative here)**

- Call fact@5 returns 'int' finally
- Record 'fact@5 :: int'

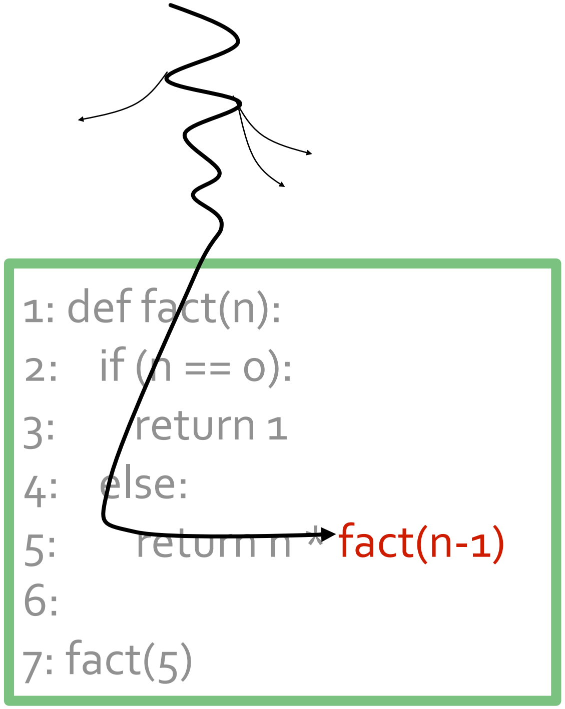
- Call fact@7 returns 'int' finally
- Record 'fact@1:: int -> int'

Correctness of Recursion Detection

```
def fact(n):
    if (n == 0):
        return 1
    else:
        return n * fact(n-1)

fact(5)
```

- Every program is a dynamic circuit
- Every call site is a ‘*conjuction point*’ in the dynamic circuit, because it connects to an *instance of a function body*
- The same call site with the same arguments is a unique joint point in the process graph, with a deterministic ‘future’
- If the same <*call site, argument type*> combination has appear before in the path, there must be a loop

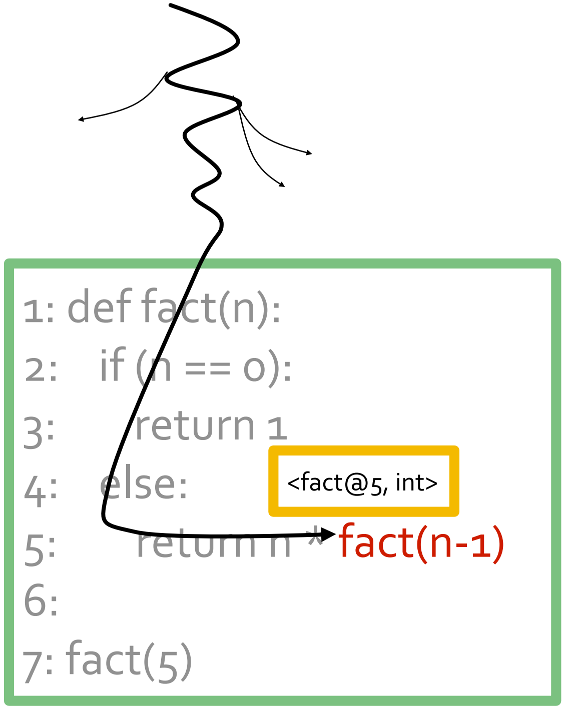
Correctness of Recursion Detection



```
def fact(n):
    if (n == 0):
        return 1
    else:
        return n * fact(n-1)

fact(5)
```

- Every program is a dynamic circuit
- Every call site is a '*conjuction point*' in the dynamic circuit, because it connects to an *instance of a function body*
- The same call site with the same arguments is a unique joint point in the process graph, with a deterministic 'future'
- If the same <*call site, argument type*> combination has appear before in the path, there must be a loop

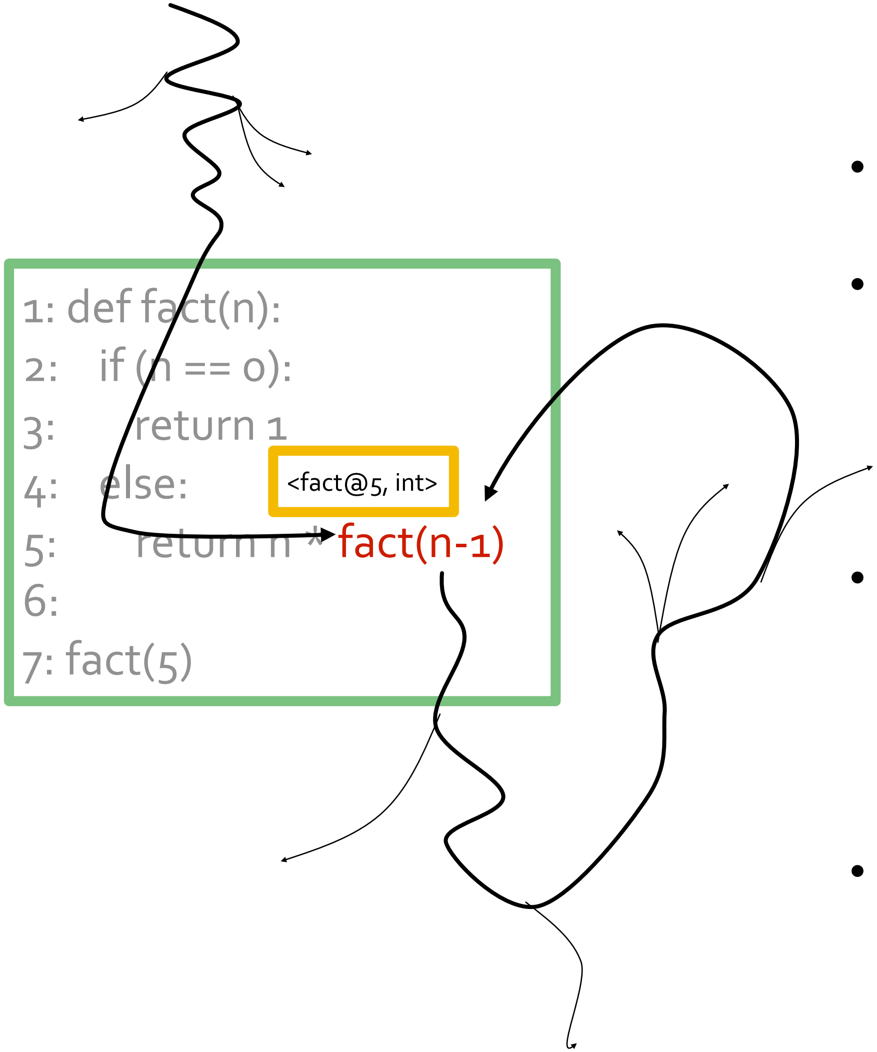
Correctness of Recursion Detection



```
1: def fact(n):
2:    if (n == 0):
3:       return 1
4:    else:
5:       return n * fact(n-1)
6:
7: fact(5)
```

- Every program is a dynamic circuit
- Every call site is a '*conjuction point*' in the dynamic circuit, because it connects to an *instance of a function body*
- The same call site with the same arguments is a unique joint point in the process graph, with a deterministic 'future'
- If the same <*call site, argument type*> combination has appear before in the path, there must be a loop

Correctness of Recursion Detection



```
1: def fact(n):
2:    if (n == 0):
3:        return 1
4:    else:
5:        return n * fact(n-1)
6:
7: fact(5)
```

- Every program is a dynamic circuit

- Every call site is a ‘*conjuction point*’ in the dynamic circuit, because it connects to an *instance of a function body*

- The same call site with the same arguments is a unique joint point in the process graph, with a deterministic ‘future’

- If the same <*call site, argument type*> combination has appear before in the path, there must be a loop

Related Work

- Similar to "control-flow analyses", but much simpler
  - No need to build CFG (as in original CFAs)
  - No need to maintain stack manually (as in CFA2)
  - "CFG" here is dynamic and implicit (maybe impossible to build statically)
  - Doesn't record any information on the AST
  - Recursive style leads to *full utilization of host language*

- Much simpler than type inferencer of JSCompiler (Google's type inference and static checker for JavaScript)
  - JSCompiler also needs type annotations, iirc

- Very similar to NCI (Near Concrete Interpretation)
  - But using another way to detect recursion

Connections to “Deeper” Theories

- In essence, the analysis is doing a simple version of “supercompilation”
- Similar to technique used by automatic theorem provers such as ACL2
- Does not track as much information (only type information is tracked)
- Termination technique is more efficient (no expensive “*homeomorphic embedding*” checks)
  - .. but may not be as accurate
  - may cause false-negatives!

Limitations

- Doesn’t process bytecode. Needs all source code to be available (except for built-ins which was hard-coded or mocked)
- Does not track value/range of numbers
- Does not track heap storage (assume side-effects on heap won’t affect typing)
- May produce false-negatives at recursions
- Worst-case complexity is high
  - More approximations can be used to improve efficiency (at the cost of reducing accuracy)
- Error reports are not user-friendly for deep bugs

Applicability

- A general way of type inference/static analysis
- Can be applied to any programming language
- More useful for *dynamic languages* because type annotations of static languages make it a lot easier and more *modular*
- There are always trade-offs though

Availability

- 2009 version “Jython Indexer” (in Java, open-source)
  - *modular analysis* with unification (similar to HM system)
  - can’t resolve some names
  - fast
  - currently used by Google for building code index
  - open-sourced to Jython

- 2010 version “PySonar” (in Java, not open-source)
  - *inter-procedural analysis*
  - *can resolve most names*
  - *can detect deeper semantic bugs*
  - *slow*

- *2011 version “mini-pysonar” (in Python, open-source)*
  - *available from GitHub*
  - *contains only the essential parts for illustrating the idea*

Possible Future Work

- Apply the technique to other (hopefully simpler) languages
- Publish a paper about the general method
- Derive other ideas from the same intuition

Thank you!