Preemptive type checking in dynamically typed programs

Neville Grech, Julian Rathke, Bernd Fischer
n.grech@ecs.soton.ac.uk

Dynamically-typed languages

Principal type of a variable is mutated through assignments:

```
def plus2(a):
    if isinstance(a,str):
        add='  '
    elif isinstance(a,int):
        add=2
    else:
        return 0
    return a+add
```

Dynamically-typed languages

Principal type of a variable is mutated through assignments:

```
def plus2(a):
    if isinstance(a,str):
        add='  '
    elif isinstance(a,int):
        add=2
    else:
        return 0
    return a+add
```

- Variable type depends on the path through the control flow graph (CFG).

Dynamically-typed languages

Principal type of a variable is mutated through assignments:

```
def plus2(a):
    if isinstance(a,str):
        add='  '
    elif isinstance(a,int):
        add=2
    else:
        return 0
    return a+add
```

- Variable type depends on the path through the control flow graph (CFG).
- *Static typing is, in general, uncomputable.*

Disadvantages of dynamically-typed languages

- *Slower* - they're usually interpreted.
- No static type safety guarantee.
- Lack of type annotations – lack of documentation.
- Need more testing to find basic type errors.

Advantages of dynamically-typed languages

- Lack of type annotations simplifies the syntax – easier to learn.
- Implementations and programmer tools are easier to write.
- Support higher-level language constructs such as metaprogramming and reflection.
- Increased developer productivity.

Advantages of dynamically-typed languages

- Lack of type annotations simplifies the syntax – easier to learn.
- Implementations and programmer tools are easier to write.
- Support higher-level language constructs such as metaprogramming and reflection.
- Increased developer productivity.

- Popularity has increased (JavaScript, Python, Ruby, etc...)
- *We want an easy way to find type errors in these languages*

A mini-language with dynamic typing

The language only supports basic control flow structures, function calls, assignments and function definitions.
Built-in functions: `useint` and `usestr`.

A mini-language with dynamic typing

The language only supports basic control flow structures, function calls, assignments and function definitions.
Built-in functions: `useint` and `usestr`.

- These raise an error if applied to a non-int or a non-str, respectively.

A mini-language with dynamic typing

The language only supports basic control flow structures, function calls, assignments and function definitions.
Built-in functions: `useint` and `usestr`.

- These raise an error if applied to a non-int or a non-str, respectively.
- `true` and `false` are built-ins, not constants.

A mini-language with dynamic typing

The language only supports basic control flow structures, function calls, assignments and function definitions.
Built-in functions: `useint` and `usestr`.

- These raise an error if applied to a non-int or a non-str, respectively.
- `true` and `false` are built-ins, not constants.

Functions, built-ins can be redefined - constants cannot.
All state is global.

A mini-language with dynamic typing

The language only supports basic control flow structures, function calls, assignments and function definitions.
Built-in functions: `useint` and `usestr`.

- These raise an error if applied to a non-int or a non-str, respectively.
- `true` and `false` are built-ins, not constants.

Functions, built-ins can be redefined - constants cannot.
All state is global.

```
data Const = I Int
           | S String
           | P Program
           | None
```

Abstact Syntax

A program is simply a list of statements:

```
data Stmt = Def Id Id [Stmt]        -- Function definitions
          | Id := Expr              -- Assignment
          | Return Expr             -- Return from function
          | If Expr [Stmt] [Stmt]   -- if..then..else
          | While Expr [Stmt]       -- While Loop
          | Only Expr               -- An expression is also
                                    -- a valid statement

data Expr = Id Id             -- An Identifier is an Expr
          | Co Const          -- So is a constant
          | Of Id Expr        -- And a function call

compile :: [Stmt] -> Program
```

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
```

Pseudocode: LC c $\Longrightarrow$ TOS:=c

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
```

Pseudocode: LG x $\Longrightarrow$ TOS:=x

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
```

Pseudocode: SG x $\Longrightarrow$ x:=TOS

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
```

Pseudocode: CF f $\Longrightarrow$ f()

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
```

Pseudocode: $\text{MF p} \implies \text{TOS:=p}$

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
```

Pseudocode: JP n $\Longrightarrow$ PC:=n

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
          | JIF Loc -- Jump if false
```

Pseudocode: JIF n $\Longrightarrow$ if TOS then PC+1 else n

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
          | JIF Loc -- Jump if false
          | RET -- Return
```

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
          | JIF Loc -- Jump if false
          | RET -- Return
          | HLT -- Halt
```

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
          | JIF Loc -- Jump if false
          | RET -- Return
          | HLT -- Halt
          | INIT -- Initialise virtual machine
```

Bytecode instructions

Runtime is based on low-level dynamically-typed bytecode.

```
data Inst = LC Const -- Load Constant
          | LG Id -- Load Global
          | SG Id -- Store Global
          | CF Id -- Call Function
          | MF Program -- Make Function
          | JP Loc -- Jump
          | JIF Loc -- Jump if false
          | RET -- Return
          | HLT -- Halt
          | INIT -- Initialise virtual machine
          | INITF -- Initialisation in function
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Compilation example

AST

```
[x := Id true,
 While randchoice
   [If randchoice
   -- THEN
     [x := Co (I 2)]
   -- ELSE
     [y := Id x]
   ]
]
```

Bytecode

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Virtual Machine

- Easily implemented using a set of reduction rules:

```
redi :: Inst -> (Map Id Const, Loc)
             -> (Map Id Const, Loc)
```

Virtual Machine

- Easily implemented using a set of reduction rules:

```
redi :: Inst -> (Map Id Const, Loc)
             -> (Map Id Const, Loc)
```

- For example:

```
redi (LC c) (env, pc) = (env <+> (tos,c), pc+1)
redi (LG x) (env, pc) = (env <+> (tos, env!x), pc+1)
redi (SG x) (env, pc) = (env <+> (x,env!tos), pc+1)
redi (JP t) (env, _) = (env, t)
..
..
```

Interpretation example

```
[INIT,
  LC (I 3),
  LC (S "h"),
  SG g,
  MF [INITF,
    LC (I 4),
    SG g,
    RET],
  SG f,
  CF f,
  LG f,
  HLT]
```

$env = []$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos : 3]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos : "h"]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos : "h", g : "h"]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos :< function... >, g : "h"]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$env = [tos :< function... >, f :< function... >, g : "h"]$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$env = [tos : 4, f :< function... >, g : "h"]$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos : 4, f :< function... >, g : 4]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$$env = [tos : 4, f :< function... >, g : 4]$$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$env = [tos :< function... >, f :< function... >, g : 4]$

Interpretation example

```
[INIT,
 LC (I 3),
 LC (S "h"),
 SG g,
 MF [INITF,
   LC (I 4),
   SG g,
   RET],
 SG f,
 CF f,
 LG f,
 HLT]
```

$env = [tos :< function... >, f :< function... >, g : 4]$

A flow-sensitive type inference algorithm

Types of variables are dependent on location.

- Therefore, type mappings (`Map Id Type`) are associated with program locations (`Loc`).

A flow-sensitive type inference algorithm

Types of variables are dependent on location.

- Therefore, type mappings (`Map Id Type`) are associated with program locations (`Loc`).

Our type inferencer infers two mappings for every location:

```
infer :: Program -> (Map Loc Mapping, Map Loc Mapping)
```

We refer to the left mapping as `p` (present):

- "The possible types of variables *after* executing the instruction at location `Loc`"

A flow-sensitive type inference algorithm

Types of variables are dependent on location.

- Therefore, type mappings (`Map Id Type`) are associated with program locations (`Loc`).

Our type inferencer infers two mappings for every location:

```
infer :: Program -> (Map Loc Mapping, Map Loc Mapping)
```

We refer to the left mapping as `p` (present):

- "The possible types of variables *after* executing the instruction at location `Loc`"

We refer to the right mapping as `f` (future):

- "The possible types that variables will be used as, at locations accessible from `Loc`, `Loc` inclusive"

A flow-sensitive type inference algorithm

The $\mathtt{p}$ mapping is formed mainly by a forward analysis:

- Control flow joins introduce union types – since the execution could have come from either way.

A flow-sensitive type inference algorithm

The `p` mapping is formed mainly by a forward analysis:

- Control flow joins introduce union types – since the execution could have come from either way.

The `f` mapping is formed mainly by a backwards analysis:

- Control-flow splits introduce union types – since we cannot statically say where the execution would proceed.

A flow-sensitive type inference algorithm

The `p` mapping is formed mainly by a forward analysis:

- Control flow joins introduce union types – since the execution could have come from either way.

The `f` mapping is formed mainly by a backwards analysis:

- Control-flow splits introduce union types – since we cannot statically say where the execution would proceed.

Function *types* are made up of two mappings:

- *side-effects* - types of all variables after invoking the function. Corresponds to `p` mapping.
- *constraints* - types of all variables for the function to succeed. Corresponds to `f` mapping.

A flow-sensitive type inference algorithm

The `p` mapping is formed mainly by a forward analysis:

- Control flow joins introduce union types – since the execution could have come from either way.

The `f` mapping is formed mainly by a backwards analysis:

- Control-flow splits introduce union types – since we cannot statically say where the execution would proceed.

Function *types* are made up of two mappings:

- *side-effects* - types of all variables after invoking the function. Corresponds to `p` mapping.
- *constraints* - types of all variables for the function to succeed. Corresponds to `f` mapping.

Algorithm is based on low-level dynamically-typed bytecode.
At runtime, the source is no longer available.

(Part of) our type definitions

```
data Type = Int | Str | Pr | Bool | NoneType
          | Undef
          | Uncons
          | Err
          | Fn Mapping Mapping
..
..
```

Concrete types: Runtime values can only have a concrete type.

(Part of) our type definitions

```
data Type = Int | Str | Pr | Bool | NoneType
          | Undef
          | Uncons
          | Err
          | Fn Mapping Mapping
..
..
```

The type of a variable that has not been defined and initialised is Undef: can only appear in the P environment.

(Part of) our type definitions

```
data Type = Int | Str | Pr | Bool | NoneType
          | Undef
          | Uncons
          | Err
          | Fn Mapping Mapping
..
..
```

The type of a variable in the F environment is Uncons if this variable is not read

(Part of) our type definitions

```
data Type = Int | Str | Pr | Bool | NoneType
          | Undef
          | Uncons
          | Err
          | Fn Mapping Mapping
..
..
```

Represents a type error

(Part of) our type definitions

```
data Type = Int | Str | Pr | Bool | NoneType
          | Undef
          | Uncons
          | Err
          | Fn Mapping Mapping
..
..
```

a function, constraints on existing variables are expressed in the first mapping while side-effects on types are represented in the second mapping.

Typing rules

Reduction rules match on instructions, transforming an environment into another:

```
type Mapping = Map Id Type
type Env = (Mapping, Mapping)
red :: Inst -> Env -> Env
```

Typing rules

Reduction rules match on instructions, transforming an environment into another:

```
type Mapping = Map Id Type
type Env = (Mapping, Mapping)
red :: Inst -> Env -> Env
```

- The p mapping given to `red` is the *p mapping from the previous location* in the program.

Typing rules

Reduction rules match on instructions, transforming an environment into another:

```
type Mapping = Map Id Type
type Env = (Mapping, Mapping)
red :: Inst -> Env -> Env
```

- The p mapping given to `red` is the *p mapping from the previous location* in the program.
- The f mapping given to `red` is the *f mapping from the next location* in the program.

Typing rules

Reduction rules match on instructions, transforming an environment into another:

```
type Mapping = Map Id Type
type Env = (Mapping, Mapping)
red :: Inst -> Env -> Env
```

- The p mapping given to `red` is the *p mapping from the previous location* in the program.
- The f mapping given to `red` is the *f mapping from the next location* in the program.
- If next or previous location is more than one location, the mappings are joined into one mapping, introducing union types.

Rules

```
red (LC c) (p,f) =
```

`typeof` returns the type of a constant

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c),
```

`typeof` returns the type of a constant

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
```

typeof returns the type of a constant

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
```

gT gets the type of an identifier from a type mapping

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
```

gT gets the type of an identifier from a type mapping

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
```

gT gets the type of an identifier from a type mapping

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
        f <+> (tos, gT f x) <+> (x,Uncons))
```

gT gets the type of an identifier from a type mapping

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
        f <+> (tos, gT f x) <+> (x,Uncons))
red (INIT) (_,f) =
   (toTypeMap initBindings <+> (ALL,Undef),f)
```

toTypeMap transforms a Id-Value map to Id-Type map

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
        f <+> (tos, gT f x) <+> (x,Uncons))
red (INIT) (_,f) =
   (toTypeMap initBindings <+> (ALL,Undef),f)
red (INITF) (_,f) = (defaultFnMap, f)
```

defaultFnMap containts the default types for all variables

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
        f <+> (tos, gT f x) <+> (x,Uncons))
red (INIT) (_,f) =
   (toTypeMap initBindings <+> (ALL,Undef),f)
red (INITF) (_,f) = (defaultFnMap, f)
red (RET) (p,_) = (p, defaultFnMap)
```

defaultFnMap containts the default types for all variables

Rules

```
red (LC c) (p,f) =
   (p <+> (tos, typeof c), f <+> (tos, Uncons))
red (LG x) (p,f) =
   (p <+> (tos, gT p x),
        f <+> (x,gT f tos) <+> (tos,Uncons))
red (SG x) (p,f) =
   (p <+> (x, gT p tos),
        f <+> (tos, gT f x) <+> (x,Uncons))
red (INIT) (_,f) =
   (toTypeMap initBindings <+> (ALL,Undef),f)
red (INITF) (_,f) = (defaultFnMap, f)
red (RET) (p,_) = (p, defaultFnMap)
red (HLT) (p,_) = (p, Map.fromList [(ALL,Uncons)])
```

`defaultFnMap` containts the default types for all variables

More rules

```
red (JP n) env = env
```

More rules

```
red (JP n) env = env
red (JIF n) (p,f) = (p, f <+> (tos, Bool))
```

More rules

```
red (JP n) env = env
red (JIF n) (p,f) = (p, f <+> (tos, Bool))
red (MF pr) (p,f) =


  where typs=infer pr
```

`infer` returns all present and future types for all variables for all locations for a particular program

More rules

```
red (JP n) env = env
red (JIF n) (p,f) = (p, f <+> (tos, Bool))
red (MF pr) (p,f) =
   (p <+> (tos, Fn (lst typs !  0) (fst typs !  (length pr -1))),
       f)
   where typs=infer pr
```

`infer` returns all present and future types for all variables for all locations for a particular program

More rules

```
red (JP n) env = env
red (JIF n) (p,f) = (p, f <+> (tos, Bool))
red (MF pr) (p,f) =
   (p <+> (tos, Fn (lst typs !  0) (fst typs !  (length pr -1))),
       f)
   where typs=infer pr
red (CF fn) (p,f) =
   (pp,Map.fromList [(k,meetType (gT f k) (gT ff k))
                              | k <- allids f ff])
   where (pp,ff)=apply (p,f) (gT p fn)
```

meetType performs an intersection of two types

apply introduces contstraints and side effects of a given function to a given environment

Type inference example with loops

```
[INIT,
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,       -- true:Bool, y:Uninit
 LG true,    -- TOS:Bool
 SG x,
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,   -- TOS:Bool
 SG x,      -- x:Bool
 LC None,
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,       -- true:Bool, y:Uninit
 LG true,    -- TOS:Bool
 SG x,       -- x:Bool
 LC None,    -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool,
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,   -- TOS:Bool
 SG x,      -- x:Bool
 LC None,   -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool
 JIF 15,
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,
 SG y,
 JP 3,
 HLT]
```

Type inference example with loops

```
[INIT,       -- true:Bool, y:Uninit
 LG true,    -- TOS:Bool
 SG x,       -- x:Bool
 LC None,    -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool
 JIF 15,      -- TOS:Bool, x:Bool
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),
 SG x,
 JP 14,
 LG x,       -- x:Bool
 SG y,       -- y:Bool
 JP 3,
 HLT]        -- TOS:Bool, x:Bool    , y:Uninit
```

Type inference example with loops

```
[INIT,        -- true:Bool, y:Uninit
 LG true,     -- TOS:Bool
 SG x,        -- x:Bool
 LC None,     -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool
 JIF 15,      -- TOS:Bool, x:Bool
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),    -- TOS:Int, x:Bool
 SG x,
 JP 14,
 LG x,        -- x:Bool
 SG y,        -- y:Bool
 JP 3,
 HLT]         -- TOS:Bool, x:Bool    , y:Uninit
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,   -- TOS:Bool
 SG x,      -- x:Bool
 LC None,   -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool
 JIF 15,      -- TOS:Bool, x:Bool
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),  -- TOS:Int, x:Bool
 SG x,      -- x:Int
 JP 14,
 LG x,      -- x:Bool
 SG y,      -- y:Bool
 JP 3,
 HLT]       -- TOS:Bool, x:Bool    , y:Uninit
```

Type inference example with loops

```
[INIT,       -- true:Bool, y:Uninit
 LG true,    -- TOS:Bool
 SG x,       -- x:Bool
 LC None,    -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool/Int
 JIF 15,      -- TOS:Bool, x:Bool/Int
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),   -- TOS:Int, x:Bool
 SG x,       -- x:Int
 JP 14,
 LG x,       -- x:Bool
 SG y,       -- y:Bool
 JP 3,
 HLT]        -- TOS:Bool, x:Bool    , y:Uninit
```

Type inference example with loops

```
[INIT,       -- true:Bool, y:Uninit
 LG true,    -- TOS:Bool
 SG x,       -- x:Bool
 LC None,    -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool/Int
 JIF 15,      -- TOS:Bool, x:Bool/Int
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),   -- TOS:Int, x:Bool
 SG x,       -- x:Int
 JP 14,
 LG x,       -- x:Bool/Int
 SG y,       -- y:Bool/Int
 JP 3,
 HLT]        -- TOS:Bool, x:Bool    , y:Uninit
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,   -- TOS:Bool
 SG x,      -- x:Bool
 LC None,   -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool/Int
 JIF 15,      -- TOS:Bool, x:Bool/Int
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2), -- TOS:Int, x:Bool
 SG x,      -- x:Int
 JP 14,
 LG x,      -- x:Bool/Int
 SG y,      -- y:Bool/Int
 JP 3,
 HLT]       -- TOS:Bool, x:Bool/Int, y:Uninit/Int/Bool
```

Type inference example with loops

```
[INIT,      -- true:Bool, y:Uninit
 LG true,   -- TOS:Bool
 SG x,      -- x:Bool
 LC None,   -- TOS:NoneType, randbool:  Fn (TOS:NoneType->Bool)
 CF randbool, -- TOS:Bool, x:Bool/Int
 JIF 15,      -- TOS:Bool, x:Bool/Int
 LC None,
 CF randbool,
 JIF 12,
 LC (I 2),  -- TOS:Int, x:Bool
 SG x,      -- x:Int
 JP 14,
 LG x,      -- x:Bool/Int
 SG y,      -- y:Bool/Int
 JP 3,
 HLT]       -- TOS:Bool, x:Bool/Int, y:Uninit/Int/Bool
```

F-environment is done in a similar way, but predominantly using a backwards analysis.

More types

The types described here do not really appear at runtime:

More types

The types described here do not really appear at runtime:

```
type SetOfType = Set Type
data Type = ...
```

The types we described so far

More types

The types described here do not really appear at runtime:

```
type SetOfType = Set Type
data Type = ...
          | Union SetOfType
```

Union types, introduced in control flow joins/splits

More types

The types described here do not really appear at runtime:

```
type SetOfType = Set Type
data Type = ...
          | Union SetOfType
          | Inter SetOfType
```

Intersection types, introduced in the F environment by successive function applications that introduce different constraints to the same variable

More types

The types described here do not really appear at runtime:

```
type SetOfType = Set Type
data Type = ...
          | Union SetOfType
          | Inter SetOfType
          | T Id -- variable types
```

A placeholder for types of variables that cannot be determined at this stage

More types

The types described here do not really appear at runtime:

```
type SetOfType = Set Type
data Type = ...
          | Union SetOfType
          | Inter SetOfType
          | T Id -- variable types
          | Aff Type Env Id -- affected types
```

An effect or constraint introduced by a function of a particular type on a variable with identifier Id, under environment Env

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f,
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g,

 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f, --f:  Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g,

 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL)),
     g: Fn (x: Uncons -> Int)
 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL)),
    g: Fn (x: Uncons -> Int)
 CF f, -- environment e0
 HLT]
```

We evaluate `f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))` under environment 0.

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL)),
    g: Fn (x: Uncons -> Int)
 CF f, -- environment e0
 HLT]
```

We evaluate `f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))` under environment 0.

`ALL:(Aff (T g) _ ALL)` in the p env. evaluates to `x: Int`

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     CF g,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     LC (I 3),
     SG x,
     RET],
 SG g, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL)),
    g: Fn (x: Uncons -> Int)
 CF f, -- environment e0
 HLT]
```

We evaluate `f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))` under environment 0.

`ALL:(Aff (T g) _ ALL)` in the p env. evaluates to `x: Int`

`ALL:(Aff (T g) _ ALL)` in the f env. evaluates to `x: Uncons`

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     MF [INITF,
         LC (I 4),
         SG x,
         RET],
     SG g,
     CF h,
     RET],
 SG f,
 MF [INITF,
     CF g,
     RET],
 SG h,
 CF f,
 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
      MF [INITF,
           LC (I 4),
           SG x,
           RET],
      SG g, --g: Fn (x: Uncons -> Int)
      CF h,
      RET],
 SG f,
 MF [INITF,
      CF g,
      RET],
 SG h,
 CF f,
 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     MF [INITF,
         LC (I 4),
         SG x,
         RET],
     SG g, --g: Fn (x: Uncons -> Int)
     CF h,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     CF g,
     RET],
 SG h,
 CF f,
 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     MF [INITF,
         LC (I 4),
         SG x,
         RET],
     SG g, --g: Fn (x: Uncons -> Int)
     CF h,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     CF g,
     RET],
 SG h, --h: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 CF f,
 CF f,
 HLT]
```

Variable/affected types and type evaluation

```
[INIT,
 MF [INITF,
     MF [INITF,
         LC (I 4),
         SG x,
         RET],
     SG g, --g: Fn (x: Uncons -> Int)
     CF h,
     RET],
 SG f, --f: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 MF [INITF,
     CF g,
     RET],
 SG h, --h: Fn (ALL: (Aff (T g) _ ALL) -> (Aff (T g) _ ALL))
 CF f, --x: Int
 CF f, --x: Int
 HLT]
```

Conclusion and Future work

It is very difficult (though possible) to infer *anything* in a dynamically-typed langauge where everything can be re-defined at runtime.

Conclusion and Future work

It is very difficult (though possible) to infer *anything* in a dynamically-typed langauge where everything can be re-defined at runtime.

Once we formalise assertion insertion, we shall prove:

- Transformed program p' never raises unexpected type errors.
- If $p = p'$ then original program never raises unexpected type errors.
- If p doesn't raise any type errors, the evaluation of p and p' are the same.

Conclusion and Future work

It is very difficult (though possible) to infer *anything* in a dynamically-typed langauge where everything can be re-defined at runtime.

Once we formalise assertion insertion, we shall prove:

- Transformed program p' never raises unexpected type errors.
- If $p = p'$ then original program never raises unexpected type errors.
- If p doesn't raise any type errors, the evaluation of p and p' are the same.

We shall also explore the use of SMT to find type errors in dynamically-typed programs.