Polymorphic Type Inference for Machine Code

Matt Noonan, Alexey Loginov, David Cok

{mnoonan,alexey,dcok}@grammatech.com

June 15, 2016

Introduction

The challenge

Given an optimized, real-world binary with no debug information, recover the original source-level types of program variables in the binary.

What is it good for?

- Reverse-engineering and program understanding.
- Better decompilation [Schwartz et al., 2013]
- Narrowing the focus of further analysis passes.

Why is type reconstruction difficult?

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

```
;; disassemble...
call foo
;; ...now keep going?
```

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

- Bit-twiddling
- Untagged unions and type-punning
- Quake III inverse sqrt
- Stealing unused pointer bits
- Downcasts (from generic to specific)
- xor-combined doubly-linked lists

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

- `typedef` aliases (`HANDLE`, `size_t`)
- Basic types with specific purposes (file descriptors, IP addresses)
- Entire un-enforced type hierarchies! (`HBRUSH` *extends* `HGDI`)

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

- Nominal subtype polymorphism
- Physical subtype polymorphism
  `struct { FILE* x; char* y; }`
  extends `FILE*`
- Emulated polytypes (allocators, `memcpy`)

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

```c
T * get_T(void)
{
  S * s = get_S();
  if (s == NULL) {
    return NULL;
  }
  T * t = S2T(s);
  return t;
}
```

```
get_T:
    call get_S
    test eax, eax
    jz   local_exit
    push eax
    call S2T
    add  esp, 4
local_exit:
    ret
```

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

```c
T * get_T(void)
{
  S * s = get_S();
  if (s == NULL) {
    return NULL;
  }
  T * t = S2T(s);
  return t;
}
```

```asm
get_T:
    call get_S
    test eax, eax
    jz   local_exit
    push eax
    call S2T
    add  esp, 4
local_exit:
    ret
```

Returns (T*) NULL

Why is type reconstruction difficult?

Challenges

- Accurate disassembly is hard!
- C and C++ have unsound type systems.
- Programmers introduce ad-hoc type disciplines.
- Polymorphic functions exist!
- Compiler optimizations can be performed on type-erased code.

Examples

```c
T * get_T(void)
{
  S * s = get_S();
  if (s == NULL) {
    return NULL;
  }
  T * t = S2T(s);
  return t;
}
```

```asm
get_T:
    call get_S
    test eax, eax
    jz   local_exit
    push eax
    call S2T
    add  esp, 4
local_exit:
    ret
```

Returns `(T*) NULL`

Returns `(S*) NULL`!

Type system desiderata

Based on a survey of idioms found in optimized C and C++ binaries, we need

- Subtypes (nominal and physical)
- Polymorphic functions
- Recursive types

and a reconstruction algorithm that is

- Scalable
- Not unification-driven
- Does not require much sound points-to data

Static approaches to type reconstruction

	TIE	SecondWrite	Theory Prop.	Retypd
Appeared in	NDSS '11	PLDI '13	PPDP '13	PLDI '16
Allows subtypes?	Limited	No	No	Yes
Points-to oracle?	Yes (DVSA)	Best-effort pts-to	via SMT	Stack only
Recursive types?	Indirect	Indirect	Yes	Yes
Polymorphic types?	No	No	No	Yes
Type system	C-like lattice	C-like lattice	Rational trees	Decorated trees
Concept	Intervals	Unsound pts-to	SMT	Pushdown automata

- Lee et al. [2011]: infer types by tracking lattice constraints.
- ElWazeer et al. [2013]: sound points-to analysis may not be required.
- Robbins et al. [2013]: use of rational trees for machine-code types.
- This work: implemented as Retypd, a type-reconstruction module for CodeSurfer for Binaries.

Static approaches to type reconstruction

	TIE	SecondWrite	Theory Prop.	Retypd
Appeared in	NDSS '11	PLDI '13	PPDP '13	PLDI '16
Allows subtypes?	Limited	No	No	Yes
Points-to oracle?	Yes (DVSA)	Best-effort pts-to	via SMT	Stack only
Recursive types?	Indirect	Indirect	Yes	Yes
Polymorphic types?	No	No	No	Yes
Type system	C-like lattice	C-like lattice	Rational trees	Decorated trees
Concept	Intervals	Unsound pts-to	SMT	Pushdown automata

- Lee et al. [2011]: infer types by tracking lattice constraints.
- ElWazeer et al. [2013]: sound points-to analysis may not be required.
- Robbins et al. [2013]: use of rational trees for machine-code types.
- This work: implemented as Retypd, a type-reconstruction module for CodeSurfer for Binaries.

Static approaches to type reconstruction

	TIE	SecondWrite	Theory Prop.	Retypd
Appeared in	NDSS '11	PLDI '13	PPDP '13	PLDI '16
Allows subtypes?	Limited	No	No	Yes
Points-to oracle?	Yes (DVSA)	Best-effort pts-to	via SMT	Stack only
Recursive types?	Indirect	Indirect	Yes	Yes
Polymorphic types?	No	No	No	Yes
Type system	C-like lattice	C-like lattice	Rational trees	Decorated trees
Concept	Intervals	Unsound pts-to	SMT	Pushdown automata

- Lee et al. [2011]: infer types by tracking lattice constraints.
- ElWazeer et al. [2013]: sound points-to analysis may not be required.
- Robbins et al. [2013]: use of rational trees for machine-code types.
- This work: implemented as Retypd, a type-reconstruction module for CodeSurfer for Binaries.

Static approaches to type reconstruction

	TIE	SecondWrite	Theory Prop.	Retypd
Appeared in	NDSS '11	PLDI '13	PPDP '13	PLDI '16
Allows subtypes?	Limited	No	No	Yes
Points-to oracle?	Yes (DVSA)	Best-effort pts-to	via SMT	Stack only
Recursive types?	Indirect	Indirect	Yes	Yes
Polymorphic types?	No	No	No	Yes
Type system	C-like lattice	C-like lattice	Rational trees	Decorated trees
Concept	Intervals	Unsound pts-to	SMT	Pushdown automata

- Lee et al. [2011]: infer types by tracking lattice constraints.
- ElWazeer et al. [2013]: sound points-to analysis may not be required.
- Robbins et al. [2013]: use of rational trees for machine-code types.
- This work: implemented as Retypd, a type-reconstruction module for CodeSurfer for Binaries.

Static approaches to type reconstruction

	TIE	SecondWrite	Theory Prop.	Retypd
Appeared in	NDSS '11	PLDI '13	PPDP '13	PLDI '16
Allows subtypes?	Limited	No	No	Yes
Points-to oracle?	Yes (DVSA)	Best-effort pts-to	via SMT	Stack only
Recursive types?	Indirect	Indirect	Yes	Yes
Polymorphic types?	No	No	No	Yes
Type system	C-like lattice	C-like lattice	Rational trees	Decorated trees
Concept	Intervals	Unsound pts-to	SMT	Pushdown automata

- Lee et al. [2011]: infer types by tracking lattice constraints.
- ElWazeer et al. [2013]: sound points-to analysis may not be required.
- Robbins et al. [2013]: use of rational trees for machine-code types.
- This work: implemented as Retypd, a type-reconstruction module for CodeSurfer for Binaries.

The constraint system

From a program to a constraint set

- Abstract location X $\rightsquigarrow$ basic type variable x.
- Dataflow between abstract locations $\rightsquigarrow$ subtype constraints:

`mov eax,ebx` $\rightsquigarrow$ $ebx_p \sqsubseteq eax_q$

- Observed capabilities of a type variable are encoded with field labels:

`mov eax,[ebx]` $\rightsquigarrow$ $ebx_p.\mathsf{load}.@0 \sqsubseteq eax_q$

- ...even for function types!

`X = F(Y)` $\rightsquigarrow$ $f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq x, \quad y \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$

From a program to a constraint set

- Abstract location X $\rightsquigarrow$ basic type variable x.
- Dataflow between abstract locations $\rightsquigarrow$ subtype constraints:

$$\texttt{mov eax,ebx} \quad \rightsquigarrow \quad ebx_p \sqsubseteq eax_q$$

- Observed capabilities of a type variable are encoded with field labels:

$$\texttt{mov eax,[ebx]} \quad \rightsquigarrow \quad ebx_p.\mathsf{load}.@0 \sqsubseteq eax_q$$

- ...even for function types!

$$\texttt{X = F(Y)} \quad \rightsquigarrow \quad f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq x, \quad y \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

From a program to a constraint set

- Abstract location X $\rightsquigarrow$ basic type variable x.
- Dataflow between abstract locations $\rightsquigarrow$ subtype constraints:

$$\texttt{mov eax,ebx} \quad \rightsquigarrow \quad ebx_p \sqsubseteq eax_q$$

- Observed capabilities of a type variable are encoded with field labels:

$$\texttt{mov eax,[ebx]} \quad \rightsquigarrow \quad ebx_p.\mathsf{load}.\mathsf{@0} \sqsubseteq eax_q$$

- ...even for function types!

$$\texttt{X = F(Y)} \quad \rightsquigarrow \quad f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq x, \quad y \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

From a program to a constraint set

- Abstract location $X \quad \rightsquigarrow \quad$ basic type variable x.
- Dataflow between abstract locations $\quad \rightsquigarrow \quad$ subtype constraints:

$$\texttt{mov eax,ebx} \quad \rightsquigarrow \quad ebx_p \sqsubseteq eax_q$$

- Observed capabilities of a type variable are encoded with field labels:

$$\texttt{mov eax,[ebx]} \quad \rightsquigarrow \quad ebx_p.\mathsf{load}.\mathsf{@0} \sqsubseteq eax_q$$

- ...even for function types!

$$\texttt{X = F(Y)} \quad \rightsquigarrow \quad f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq x, \quad y \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.\text{@4 exists}, \quad y.\text{@4 exists} \quad \vdash \quad x.\text{@4} \sqsubseteq y.\text{@4}$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\text{load} \sqsubseteq y.\text{load}$$
$$y.\text{store} \sqsubseteq x.\text{store}$$

Functions, $f \sqsubseteq g$:

$$f.\text{out}_{\text{eax}} \sqsubseteq g.\text{out}_{\text{eax}}$$
$$g.\text{in}_{\text{stack0}} \sqsubseteq f.\text{in}_{\text{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.\text{@4 exists}, \quad y.\text{@4 exists} \quad \vdash \quad x.\text{@4} \sqsubseteq y.\text{@4}$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\mathsf{load} \sqsubseteq y.\mathsf{load}$$
$$y.\mathsf{store} \sqsubseteq x.\mathsf{store}$$

Functions, $f \sqsubseteq g$:

$$f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq g.\mathsf{out}_{\mathsf{eax}}$$
$$g.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.@4 \text{ exists}, \quad y.@4 \text{ exists} \quad \vdash \quad x.@4 \sqsubseteq y.@4$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\mathrm{load} \sqsubseteq y.\mathrm{load}$$
$$y.\mathrm{store} \sqsubseteq x.\mathrm{store}$$

Functions, $f \sqsubseteq g$:

$$f.\mathrm{out}_{\mathrm{eax}} \sqsubseteq g.\mathrm{out}_{\mathrm{eax}}$$
$$g.\mathrm{in}_{\mathrm{stack0}} \sqsubseteq f.\mathrm{in}_{\mathrm{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.\text{@4 exists}, \quad y.\text{@4 exists} \quad \vdash \quad x.\text{@4} \sqsubseteq y.\text{@4}$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\mathsf{load} \sqsubseteq y.\mathsf{load}$$
$$y.\mathsf{store} \sqsubseteq x.\mathsf{store}$$

Functions, $f \sqsubseteq g$:

$$f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq g.\mathsf{out}_{\mathsf{eax}}$$
$$g.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.\text{@4 exists}, \quad y.\text{@4 exists} \quad \vdash \quad x.\text{@4} \sqsubseteq y.\text{@4}$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\text{load} \sqsubseteq y.\text{load}$$
$$y.\text{store} \sqsubseteq x.\text{store}$$

Functions, $f \sqsubseteq g$:

$$f.\text{out}_{\text{eax}} \sqsubseteq g.\text{out}_{\text{eax}}$$
$$g.\text{in}_{\text{stack0}} \sqsubseteq f.\text{in}_{\text{stack0}}$$

Derived typing judgements

In addition to standard judgements for subtype relations, we would like to infer constraints between derived type variables.

$$x \sqsubseteq y, \quad x.\text{@4 exists}, \quad y.\text{@4 exists} \quad \vdash \quad x.\text{@4} \sqsubseteq y.\text{@4}$$

Caveat: Some of the capabilities must act contravariantly!

Mutable references, $x \sqsubseteq y$:

$$x.\mathsf{load} \sqsubseteq y.\mathsf{load}$$
$$y.\mathsf{store} \sqsubseteq x.\mathsf{store}$$

Functions, $f \sqsubseteq g$:

$$f.\mathsf{out}_{\mathsf{eax}} \sqsubseteq g.\mathsf{out}_{\mathsf{eax}}$$
$$g.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq f.\mathsf{in}_{\mathsf{stack0}}$$

Constraint entailment rules

$$\frac{\alpha \sqsubseteq \beta}{\alpha \text{ exists}} \text{(T-Left)} \qquad \frac{\alpha \sqsubseteq \beta}{\beta \text{ exists}} \text{(T-Right)} \qquad \frac{\alpha \text{ exists}}{\alpha \sqsubseteq \alpha} \text{(S-Refl)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma} \text{(S-Trans)}$$

$$\frac{\alpha \sqsubseteq \beta, \quad \alpha.\ell \text{ exists}}{\beta.\ell \text{ exists}} \text{(T-InheritL)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}}{\alpha.\ell \text{ exists}} \text{(T-InheritR)} \qquad \frac{\alpha.\ell \text{ exists}}{\alpha \text{ exists}} \text{(T-Prefix)}$$

$$\frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ covariant}}{\alpha.\ell \sqsubseteq \beta.\ell} \text{(S-Field}_{\oplus}\text{)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ contravariant}}{\beta.\ell \sqsubseteq \alpha.\ell} \text{(S-Field}_{\ominus}\text{)}$$

$$\frac{\alpha.\text{load exists}, \quad \alpha.\text{store exists}}{\alpha.\text{store} \sqsubseteq \alpha.\text{load}} \text{(S-Pointer)}$$

Constraint entailment rules

$$\frac{\alpha \sqsubseteq \beta}{\alpha \text{ exists}}\ (\text{T-Left}) \qquad \frac{\alpha \sqsubseteq \beta}{\beta \text{ exists}}\ (\text{T-Right}) \qquad \frac{\alpha \text{ exists}}{\alpha \sqsubseteq \alpha}\ (\text{S-Refl}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma}\ (\text{S-Trans})$$

$$\frac{\alpha \sqsubseteq \beta, \quad \alpha.\ell \text{ exists}}{\beta.\ell \text{ exists}}\ (\text{T-InheritL}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}}{\alpha.\ell \text{ exists}}\ (\text{T-InheritR}) \qquad \frac{\alpha.\ell \text{ exists}}{\alpha \text{ exists}}\ (\text{T-Prefix})$$

$$\frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ covariant}}{\alpha.\ell \sqsubseteq \beta.\ell}\ (\text{S-Field}_{\oplus}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ contravariant}}{\beta.\ell \sqsubseteq \alpha.\ell}\ (\text{S-Field}_{\ominus})$$

$$\frac{\alpha.\text{load exists}, \quad \alpha.\text{store exists}}{\alpha.\text{store} \sqsubseteq \alpha.\text{load}}\ (\text{S-Pointer})$$

Constraint entailment rules

$$\frac{\alpha \sqsubseteq \beta}{\alpha \text{ exists}}\ \text{(T-Left)} \qquad \frac{\alpha \sqsubseteq \beta}{\beta \text{ exists}}\ \text{(T-Right)} \qquad \frac{\alpha \text{ exists}}{\alpha \sqsubseteq \alpha}\ \text{(S-Refl)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma}\ \text{(S-Trans)}$$

$$\frac{\alpha \sqsubseteq \beta, \quad \alpha.\ell \text{ exists}}{\beta.\ell \text{ exists}}\ \text{(T-InheritL)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}}{\alpha.\ell \text{ exists}}\ \text{(T-InheritR)} \qquad \frac{\alpha.\ell \text{ exists}}{\alpha \text{ exists}}\ \text{(T-Prefix)}$$

$$\frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ covariant}}{\alpha.\ell \sqsubseteq \beta.\ell}\ (\text{S-Field}_{\oplus}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ contravariant}}{\beta.\ell \sqsubseteq \alpha.\ell}\ (\text{S-Field}_{\ominus})$$

$$\frac{\alpha.\text{load exists}, \quad \alpha.\text{store exists}}{\alpha.\text{store} \sqsubseteq \alpha.\text{load}}\ \text{(S-Pointer)}$$

Constraint entailment rules

$$\frac{\alpha \sqsubseteq \beta}{\alpha \text{ exists}}\ (\text{T-Left}) \qquad \frac{\alpha \sqsubseteq \beta}{\beta \text{ exists}}\ (\text{T-Right}) \qquad \frac{\alpha \text{ exists}}{\alpha \sqsubseteq \alpha}\ (\text{S-Refl}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma}\ (\text{S-Trans})$$

$$\frac{\alpha \sqsubseteq \beta, \quad \alpha.\ell \text{ exists}}{\beta.\ell \text{ exists}}\ (\text{T-InheritL}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}}{\alpha.\ell \text{ exists}}\ (\text{T-InheritR}) \qquad \frac{\alpha.\ell \text{ exists}}{\alpha \text{ exists}}\ (\text{T-Prefix})$$

$$\frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ covariant}}{\alpha.\ell \sqsubseteq \beta.\ell}\ (\text{S-Field}_{\oplus}) \qquad \frac{\alpha \sqsubseteq \beta, \quad \beta.\ell \text{ exists}, \quad \ell \text{ contravariant}}{\beta.\ell \sqsubseteq \alpha.\ell}\ (\text{S-Field}_{\ominus})$$

$$\frac{\alpha.\mathsf{load} \text{ exists}, \quad \alpha.\mathsf{store} \text{ exists}}{\alpha.\mathsf{store} \sqsubseteq \alpha.\mathsf{load}}\ (\text{S-Pointer})$$

Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store.@0} \\ P.\mathsf{load.@0} \sqsubseteq Y \end{array} \right\}$

Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?

Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q  :=  p;
*q  :=  x;
 y  := *p;
```

: $\left\{ \begin{array}{c} \boxed{P \sqsubseteq Q} \\ X \sqsubseteq Q\text{.store.@0} \\ P\text{.load.@0} \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?

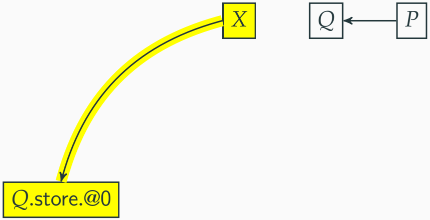
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



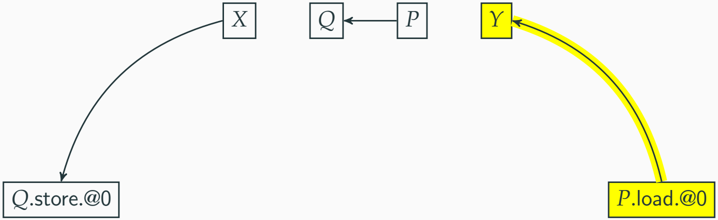
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathrm{store}.@0 \\ P.\mathrm{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



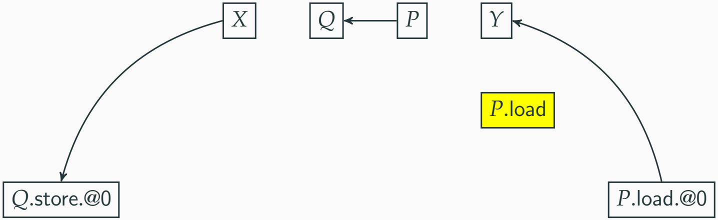
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{P.\mathsf{store}.@0 \text{ exists}}{P.\mathsf{store} \text{ exists}} \text{(T-Prefix)}$$

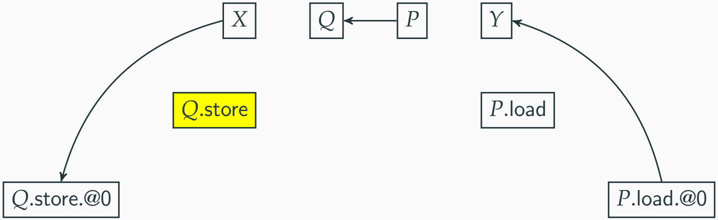
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\text{store.@0} \\ P.\text{load.@0} \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{Q.\text{store.@0 exists}}{Q.\text{store exists}} \ \text{(T-Prefix)}$$

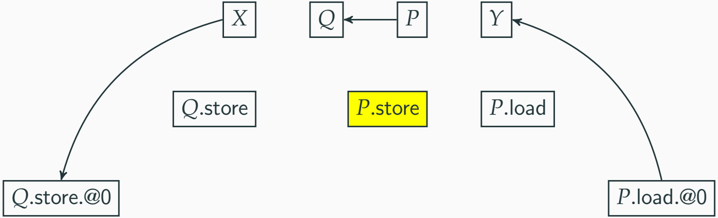
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q  :=  p;
*q  :=  x;
 y  := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{P \sqsubseteq Q, \qquad Q.\mathsf{store}\text{ exists}}{P.\mathsf{store}\text{ exists}}\ \text{(T-InheritR)}$$

Who needs points-to analysis?

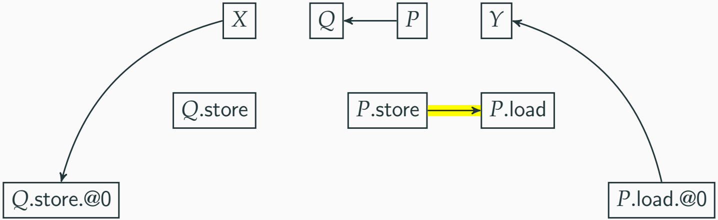
Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q  :=  p;
*q  :=  x;
 y  := *p;
```

:

$$\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$$

Idea: Model entailment with graph reachability?



$$\frac{P.\mathsf{load}\ \text{exists}, \qquad P.\mathsf{store}\ \text{exists}}{P.\mathsf{store} \sqsubseteq P.\mathsf{load}}\ \text{(S-Pointer)}$$

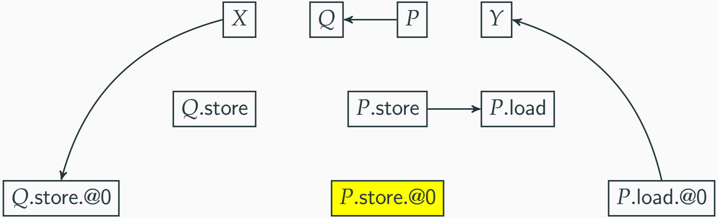
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{P.\mathsf{store} \sqsubseteq P.\mathsf{load}, \quad P.\mathsf{load}.@0 \text{ exists}}{P.\mathsf{load}.@0 \text{ exists}} \text{(T-InheritR)}$$

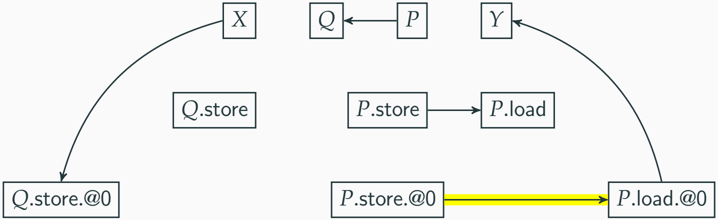
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{P.\mathsf{store} \sqsubseteq P.\mathsf{load},\quad P.\mathsf{load}.@0 \text{ exists},\quad @0 \text{ covariant}}{P.\mathsf{store}.@0 \sqsubseteq P.\mathsf{load}.@0}\ (\text{S-Field}_{\oplus})$$

Who needs points-to analysis?

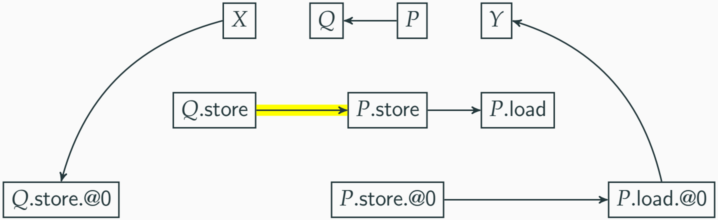
Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

:

$$\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$$

Idea: Model entailment with graph reachability?



$$\frac{P \sqsubseteq Q, \qquad Q.\mathsf{store}\ \text{exists}, \qquad \mathsf{store}\ \text{contravariant}}{Q.\mathsf{store} \sqsubseteq P.\mathsf{store}}\ (\text{S-Field}_{\ominus})$$

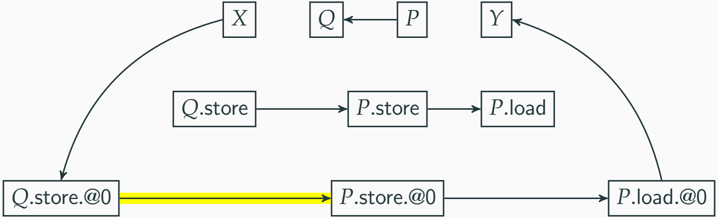
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



$$\frac{Q.\mathsf{store} \sqsubseteq P.\mathsf{store},\quad P.\mathsf{load}.@0 \text{ exists},\quad @0 \text{ covariant}}{Q.\mathsf{store}.@0 \sqsubseteq P.\mathsf{store}.@0}\ (\text{S-Field}_{\oplus})$$

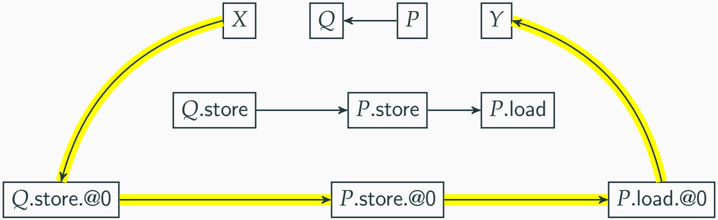
Who needs points-to analysis?

Challenge: Prove $X \sqsubseteq Y$ for the program

```
 q := p;
*q := x;
 y := *p;
```

: $\left\{ \begin{array}{c} P \sqsubseteq Q \\ X \sqsubseteq Q.\mathsf{store}.@0 \\ P.\mathsf{load}.@0 \sqsubseteq Y \end{array} \right\}$

Idea: Model entailment with graph reachability?



What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathsf{load}.@0\}$$

$$\boxed{X} \longrightarrow \boxed{X.\mathsf{load}.@0}$$

What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathsf{load}.@0\}$$

$$\begin{tikzcd} \boxed{X} \arrow[r] & \boxed{X.\mathsf{load}.@0} \\ \boxed{X.\mathsf{load}} & \end{tikzcd}$$

What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathrm{load}.@0\}$$

$$\boxed{X} \longrightarrow \boxed{X.\mathrm{load}.@0}$$

$$\boxed{X.\mathrm{load}} \longrightarrow \boxed{X.\mathrm{load}.@0.\mathrm{load}}$$

What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathsf{load}.@0\}$$

$$\boxed{X} \longrightarrow \boxed{X.\mathsf{load}.@0} \longrightarrow \boxed{X.\mathsf{load}.@0.\mathsf{load}.@0}$$

$$\boxed{X.\mathsf{load}} \longrightarrow \boxed{X.\mathsf{load}.@0.\mathsf{load}}$$

What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathrm{load}.@0\}$$

$$\boxed{X} \longrightarrow \boxed{X.\mathrm{load}.@0} \longrightarrow \boxed{X.\mathrm{load}.@0.\mathrm{load}.@0}$$

$$\boxed{X.\mathrm{load}} \longrightarrow \boxed{X.\mathrm{load}.@0.\mathrm{load}} \longrightarrow \boxed{X.\mathrm{load}.@0.\mathrm{load}.@0.\mathrm{load}}$$

What about recursive constraints?

The problem with recursive constraints

One constraint entailed infinitely many; no finite graph suffices.

$$\{X \sqsubseteq X.\mathsf{load}.@0\}$$

$$\boxed{X} \longrightarrow \boxed{X.\mathsf{load}.@0} \longrightarrow \boxed{X.\mathsf{load}.@0.\mathsf{load}.@0} \longrightarrow \cdots$$

$$\boxed{X.\mathsf{load}} \longrightarrow \boxed{X.\mathsf{load}.@0.\mathsf{load}} \longrightarrow \boxed{X.\mathsf{load}.@0.\mathsf{load}.@0.\mathsf{load}} \longrightarrow \cdots$$

Taming the infinite

Abstracting the tails

What went wrong in the recursive example?

One constraint, infinitely many edges:

$$\boxed{X.w} \longrightarrow \boxed{Y.w} \quad \text{or} \quad \boxed{X.w} \longleftarrow \boxed{Y.w}$$

Abstracting the tails

What went wrong in the recursive example?

One constraint, infinitely many edges:

$$\boxed{X.w} \longrightarrow \boxed{Y.w} \quad \text{or} \quad \boxed{X.w} \longleftarrow \boxed{Y.w}$$

Solution: Abstract away the tails

Collapse all of these edges into just two!

$$\boxed{X.\oplus} \longrightarrow \boxed{Y.\oplus} \qquad \boxed{X.\ominus} \longleftarrow \boxed{Y.\ominus}$$

Abstracting the tails

What went wrong in the recursive example?

One constraint, infinitely many edges:

$$\boxed{X.w} \longrightarrow \boxed{Y.w} \quad \text{or} \quad \boxed{X.w} \longleftarrow \boxed{Y.w}$$

Solution: Abstract away the tails

Collapse all of these edges into just two!

$$\boxed{X.\oplus} \longrightarrow \boxed{Y.\oplus} \qquad \boxed{X.\ominus} \longleftarrow \boxed{Y.\ominus}$$

$$\{X.w \mid \mathcal{C} \vdash X.w \text{ exists} \wedge w \text{ covariant seq.}\}$$

Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.\mathsf{@0}\}$$

$$\boxed{X.\oplus} \longrightarrow \boxed{X.\mathsf{load}.\mathsf{@0}.\oplus} \qquad \boxed{X.\ominus} \longleftarrow \boxed{X.\mathsf{load}.\mathsf{@0}.\ominus}$$

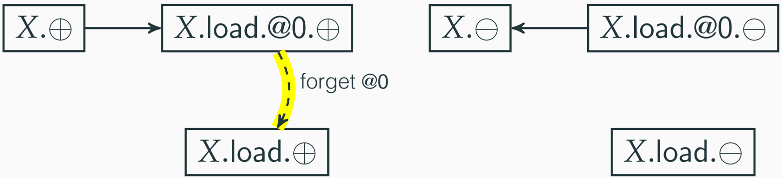
Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.@0\}$$

$$\begin{tikzcd}
\boxed{X.\oplus} \arrow[r] & \boxed{X.\mathsf{load}.@0.\oplus} & \boxed{X.\ominus} & \boxed{X.\mathsf{load}.@0.\ominus} \arrow[l] \\
& \boxed{X.\mathsf{load}.\oplus} & & \boxed{X.\mathsf{load}.\ominus}
\end{tikzcd}$$

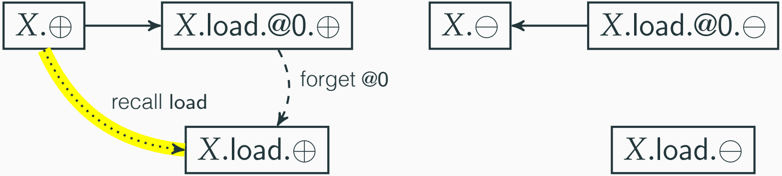
Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.@0\}$$



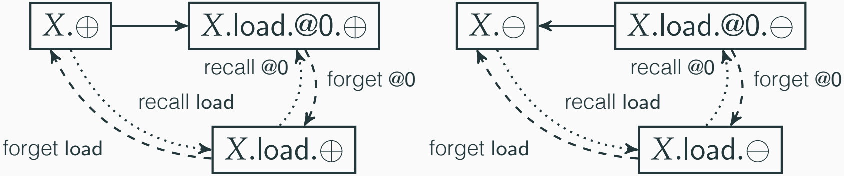
Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.@0\}$$



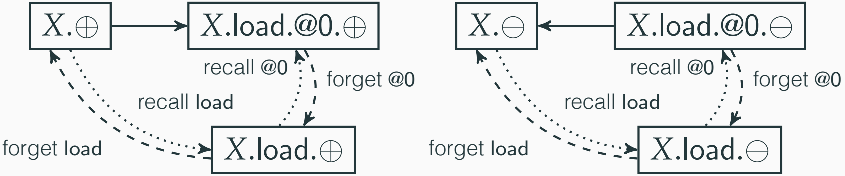
Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.@0\}$$



Labeled edges in the finite graph

$$\mathcal{C} = \{X \sqsubseteq X.\mathsf{load}.@0\}$$



$$\mathsf{forget}\ \alpha \cdot \mathsf{recall}\ \alpha = 1$$

$$\mathsf{forget}\ \alpha \cdot \mathsf{recall}\ \beta = 0$$

What can you get from manipulating constraints as graphs?

- Automata that recognize the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$
- Simplified / minimized constraint sets for type schemes.
- Satisfiability checks.
- Construction of sketches (marked regular trees) satisfiying the constraints.

What can you get from manipulating constraints as graphs?

- Automata that recognize the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$
- Simplified / minimized constraint sets for type schemes.
- Satisfiability checks.
- Construction of sketches (marked regular trees) satisfiying the constraints.

What can you get from manipulating constraints as graphs?

- Automata that recognize the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$
- Simplified / minimized constraint sets for type schemes.
- Satisfiability checks.
- Construction of sketches (marked regular trees) satisfiying the constraints.

What can you get from manipulating constraints as graphs?

- Automata that recognize the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$
- Simplified / minimized constraint sets for type schemes.
- Satisfiability checks.
- Construction of sketches (marked regular trees) satisfiying the constraints.

What can you get from manipulating constraints as graphs?

- Automata that recognize the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$
- Simplified / minimized constraint sets for type schemes.
- Satisfiability checks.
- Construction of sketches (marked regular trees) satisfiying the constraints.

Example: A recursive type in the wild

```c
#include <stdlib.h>

struct LL
{
    struct LL * next;
    int handle;
};

int close_last(struct LL * list)
{
    while (list->next != NULL)
    {
        list = list->next;
    }
    return close(list->handle);
}
```

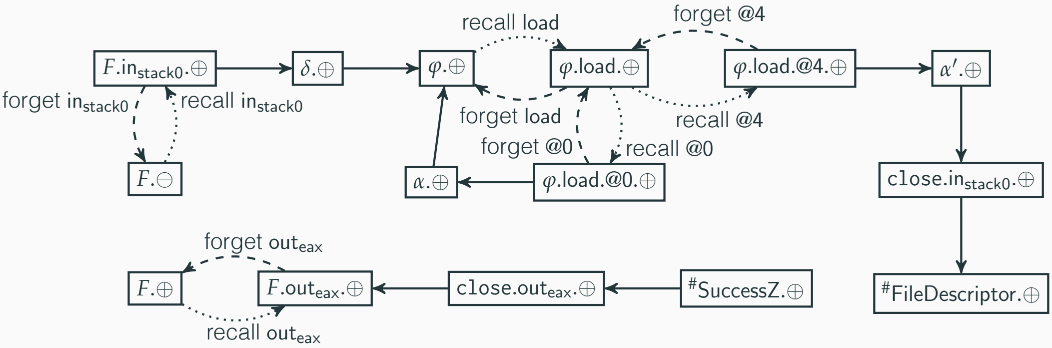
```asm
close_last:
    push    ebp
    mov     ebp,esp
    sub     esp,8
    mov     edx,dword [ebp+arg_0]
    jmp     loc_8048402
loc_8048400:
    mov     edx,eax
loc_8048402:
    mov     eax,dword [edx]
    test    eax,eax
    jnz     loc_8048400
    mov     eax,dword [edx+4]
    mov     dword [ebp+arg_0],eax
    leave
    jmp     __thunk_.close
```

Example: A recursive type in the wild

$$F.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq \delta$$
$$\alpha \sqsubseteq \varphi$$
$$\delta \sqsubseteq \varphi$$
$$\varphi.\mathsf{load}.@0 \sqsubseteq \alpha$$
$$\varphi.\mathsf{load}.@4 \sqsubseteq \alpha'$$
$$\alpha' \sqsubseteq \mathsf{close}.\mathsf{in}_{\mathsf{stack0}}$$
$$\mathsf{close}.\mathsf{out}_{\mathsf{eax}} \sqsubseteq F.\mathsf{out}_{\mathsf{eax}}$$
$$\mathsf{close}.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq {}^{\#}\mathsf{FileDescriptor}$$
$${}^{\#}\mathsf{SuccessZ} \sqsubseteq \mathsf{close}.\mathsf{out}_{\mathsf{eax}}$$

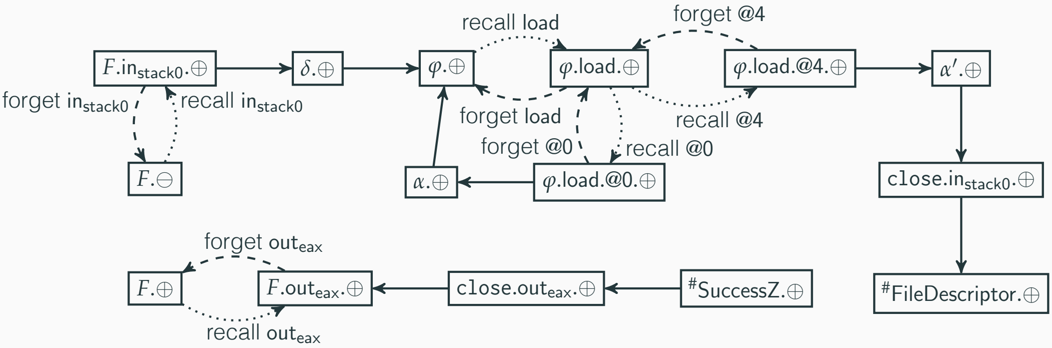
```
close_last:
    push    ebp
    mov     ebp,esp
    sub     esp,8
    mov     edx,dword [ebp+arg_0]
    jmp     loc_8048402
loc_8048400:
    mov     edx,eax
loc_8048402:
    mov     eax,dword [edx]
    test    eax,eax
    jnz     loc_8048400
    mov     eax,dword [edx+4]
    mov     dword [ebp+arg_0],eax
    leave
    jmp     __thunk_.close
```

Example, cont'd: Simplifying constraint sets



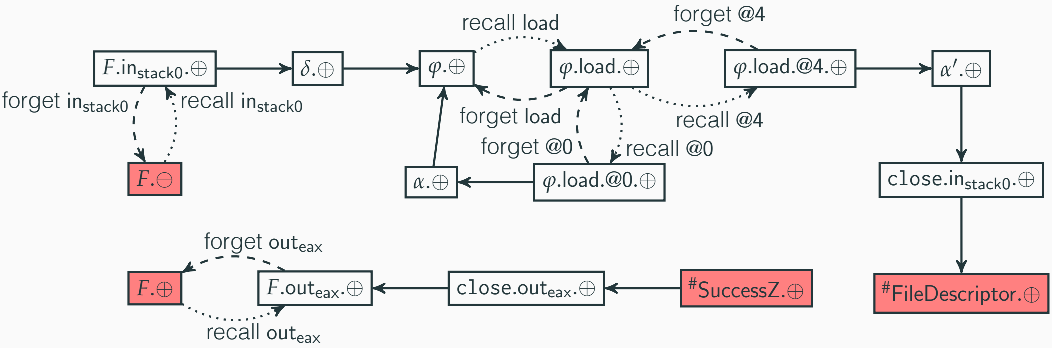
Step 1: From the constraint set, build a graph with an edge for each subtype constraint, and forget/recall-labeled edges describing the derived type variables.

Example, cont'd: Simplifying constraint sets



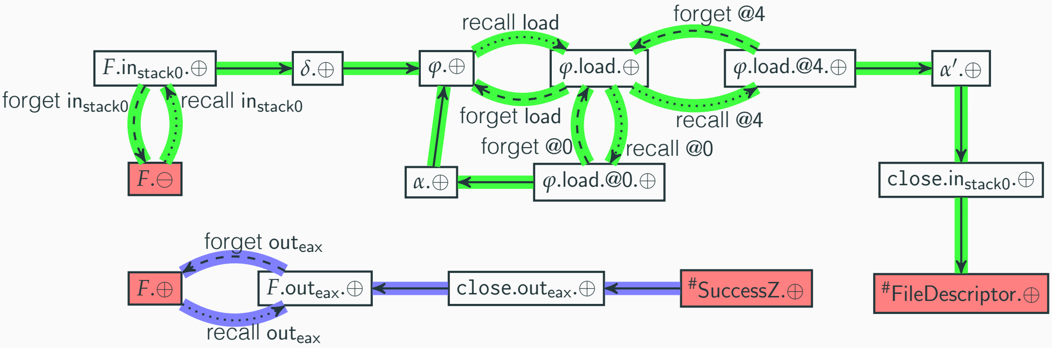
Step 2: Add edges to reify the relation forget $\alpha \cdot$ recall $\alpha = 1$. Lazily instantiate applications of S-Pointer as needed.

Example, cont'd: Simplifying constraint sets



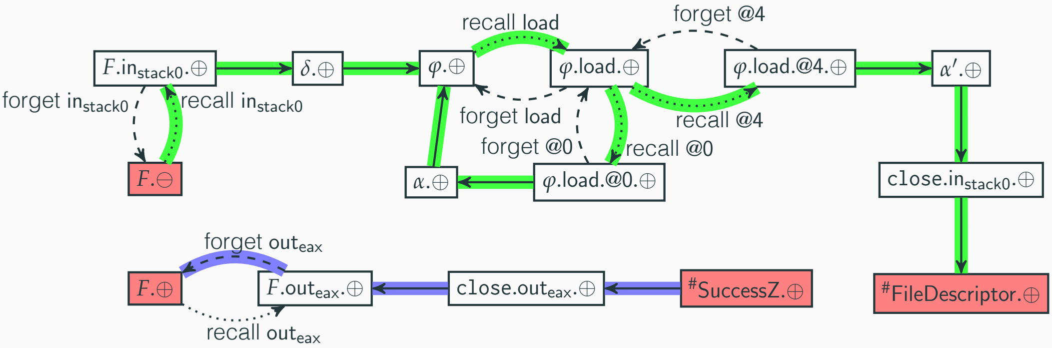
Step 3: Identify the "externally-visible" type variables and constants; call that set $\mathcal{E}$.

Example, cont'd: Simplifying constraint sets



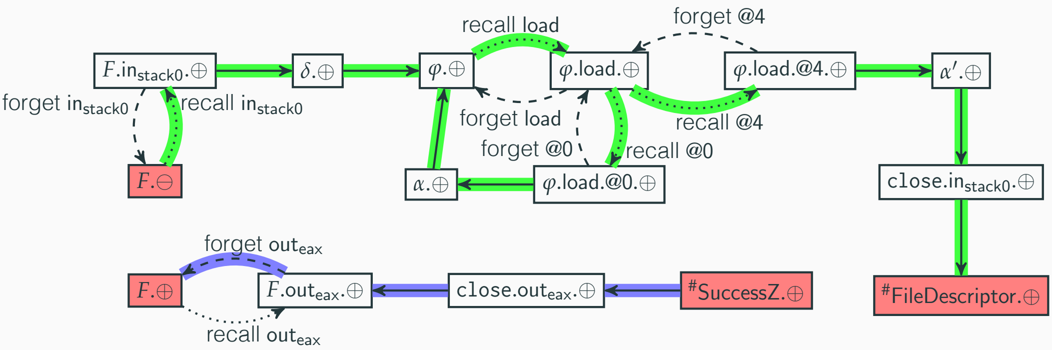
Step 4: Use Tarjan's path-expression algorithm to describe all paths that start and end in $\mathcal{E}$ but only travel through $\mathcal{E}^c$.

Example, cont'd: Simplifying constraint sets



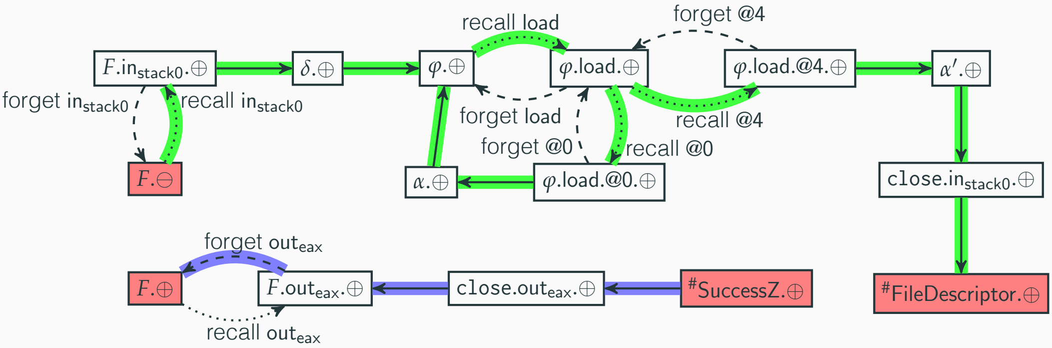
Step 5: Intersect the path expressions with the language $(\text{recall }\_)^*(\text{forget }\_)^*$.

Example, cont'd: Simplifying constraint sets



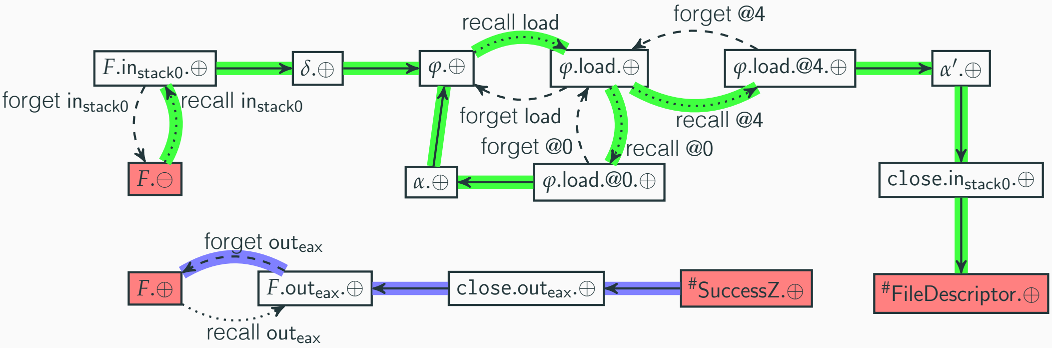
Step 6: Interpret the resulting language as a regular set of subtype constraints. ("forgets" on the right, "recalls" on the left)

Example, cont'd: Simplifying constraint sets



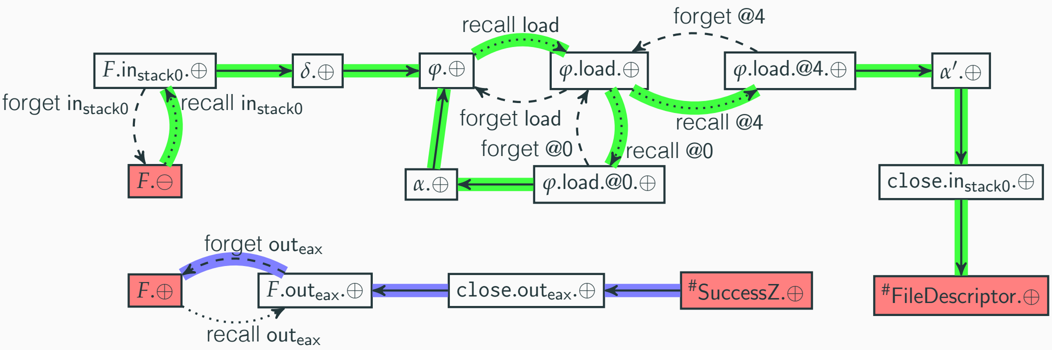
$$
\begin{aligned}
{}^{\#}\text{SuccessZ} &\rightsquigarrow F : && \text{forget out}_{\text{eax}} \\
F &\rightsquigarrow {}^{\#}\text{FileDescriptor} : && \text{recall } \left(\text{in}_{\text{stack0}} \cdot (\text{load} \cdot @0)^* \cdot \text{load} \cdot @4\right)
\end{aligned}
$$

Example, cont'd: Simplifying constraint sets



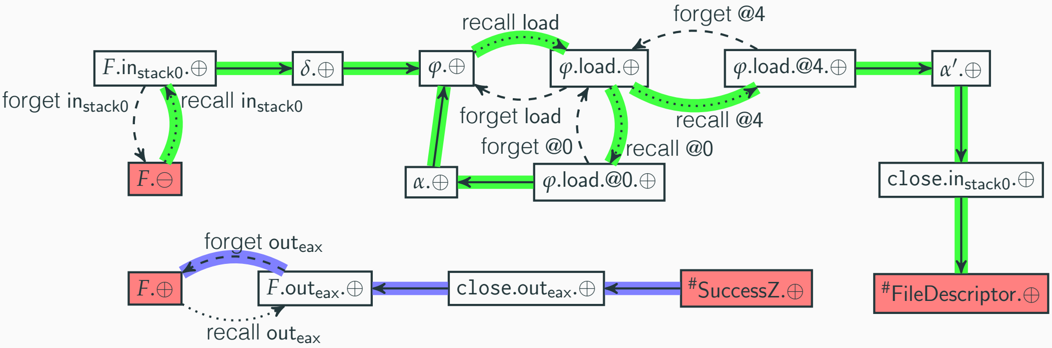
$$\mathcal{C} = \left\{ \begin{array}{l} {}^{\#}\mathsf{SuccessZ} \sqsubseteq F.\mathsf{out}_{\mathsf{eax}} \\ F.\mathsf{in}_{\mathsf{stack0}}\ (.\mathsf{load}.@0)^{*}\ .\mathsf{load}.@4 \sqsubseteq {}^{\#}\mathsf{FileDescriptor} \end{array} \right\}$$

Example, cont'd: Simplifying constraint sets



$$\mathcal{C} = \left\{ \begin{array}{l} ^{\#}\mathsf{SuccessZ} \sqsubseteq F.\mathsf{out}_{\mathsf{eax}} \\ F.\mathsf{in}_{\mathsf{stack0}}\,(.\mathsf{load}.@0)^*.\mathsf{load}.@4 \sqsubseteq {}^{\#}\mathsf{FileDescriptor} \end{array} \right\}$$

Example, cont'd: Simplifying constraint sets



$$\mathcal{C} = \exists\tau.\ \left\{ \begin{array}{l} ^{\#}\mathsf{SuccessZ} \sqsubseteq F.\mathsf{out}_{\mathsf{eax}} \\ F.\mathsf{in}_{\mathsf{stack0}} \sqsubseteq \tau, \quad \tau.\mathsf{load}.@0 \sqsubseteq \tau, \quad \tau.\mathsf{load}.@4 \sqsubseteq {}^{\#}\mathsf{FileDescriptor} \end{array} \right\}$$

Example, cont'd: Getting back to C types

```c
#include <stdlib.h>

struct LL
{
    struct LL * next;
    int handle;
};

int close_last(struct LL * list)
{
    while (list->next != NULL)
    {
        list = list->next;
    }
    return close(list->handle);
}
```

```asm
close_last:
    push    ebp
    mov     ebp,esp
    sub     esp,8
    mov     edx,dword [ebp+arg_0]
    jmp     loc_8048402
loc_8048400:
    mov     edx,eax
loc_8048402:
    mov     eax,dword [edx]
    test    eax,eax
    jnz     loc_8048400
    mov     eax,dword [edx+4]
    mov     dword [ebp+arg_0],eax
    leave
    jmp     __thunk_.close
```

Example, cont'd: Getting back to C types

$$\texttt{close\_last} : \forall F.\ (\exists \tau.\, \mathcal{C}) \Rightarrow F$$

where

$$\mathcal{C} = \left\{ \begin{array}{ll} {}^{\#}\mathsf{SuccessZ} & \sqsubseteq F.\mathsf{out}_{\mathsf{eax}} \\ F.\mathsf{in}_{\mathsf{stack0}} & \sqsubseteq \tau \\ \tau.\mathsf{load}.\mathsf{@0} & \sqsubseteq \tau \\ \tau.\mathsf{load}.\mathsf{@4} & \sqsubseteq {}^{\#}\mathsf{FileDesc}. \end{array} \right\}$$

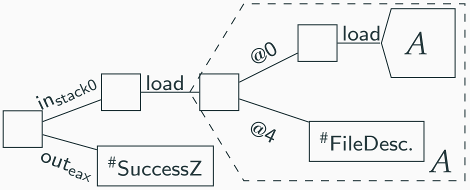
```
close_last:
    push    ebp
    mov     ebp,esp
    sub     esp,8
    mov     edx,dword [ebp+arg_0]
    jmp     loc_8048402
loc_8048400:
    mov     edx,eax
loc_8048402:
    mov     eax,dword [edx]
    test    eax,eax
    jnz     loc_8048400
    mov     eax,dword [edx+4]
    mov     dword [ebp+arg_0],eax
    leave
    jmp     __thunk_.close
```

Example, cont'd: Getting back to C types

$$\texttt{close\_last} : \forall F.\ (\exists \tau.\ \mathcal{C}) \Rightarrow F$$

where

$$\mathcal{C} = \left\{ \begin{array}{ll} {}^{\#}\text{SuccessZ} & \sqsubseteq F.\text{out}_{\text{eax}} \\ F.\text{in}_{\text{stack0}} & \sqsubseteq \tau \\ \tau.\text{load.@0} & \sqsubseteq \tau \\ \tau.\text{load.@4} & \sqsubseteq {}^{\#}\text{FileDesc.} \end{array} \right\}$$

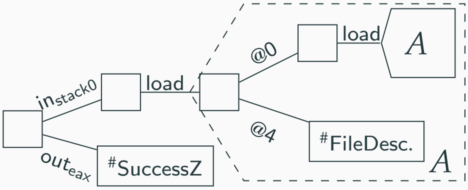


Example, cont'd: Getting back to C types

$$\texttt{close\_last} : \forall F.\ (\exists \tau.\ \mathcal{C}) \Rightarrow F$$

where

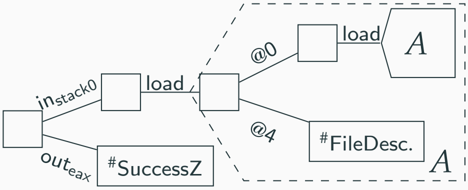
$$\mathcal{C} = \left\{ \begin{array}{ll} {}^{\#}\text{SuccessZ} & \sqsubseteq F.\text{out}_{\text{eax}} \\ F.\text{in}_{\text{stack0}} & \sqsubseteq \tau \\ \tau.\text{load}.@0 & \sqsubseteq \tau \\ \tau.\text{load}.@4 & \sqsubseteq {}^{\#}\text{FileDesc.} \end{array} \right\}$$




```c
typedef struct Struct_0 {
    struct Struct_0 * field_0;
    int // #FileDescriptor
      field_4;
} Struct_0;

int // #SuccessZ
close_last(const Struct_0 *);
```

Example, cont'd: Getting back to C types

```c
#include <stdlib.h>

struct LL
{
    struct LL * next;
    int handle;
};

int close_last(struct LL * list)
{
    while (list->next != NULL)
    {
        list = list->next;
    }
    return close(list->handle);
}
```




```c
typedef struct Struct_0 {
    struct Struct_0 * field_0;
    int // #FileDescriptor
      field_4;
} Struct_0;

int // #SuccessZ
close_last(const Struct_0 *);
```

Evaluation of the technique

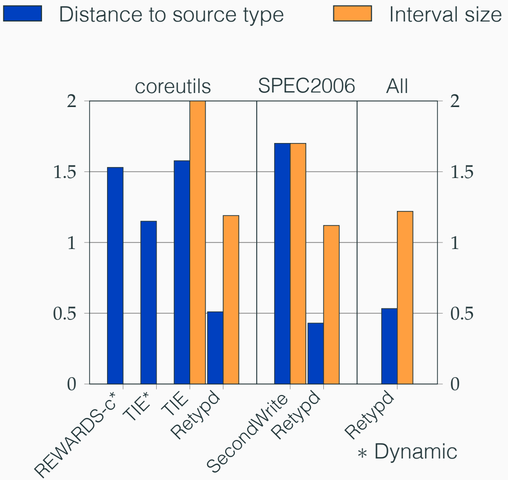
Benchmark suite

Retypd was evaluated on a suite of 160 optimized 32-bit x86 binaries with debug information removed. Separate optimized debug builds were used to establish ground truth.

The benchmark suite included:

- A suite of real-world Windows binaries compiled with Visual Studio.
- The SPEC2006 binaries used to evaluate type inference for SecondWrite.
- The coreutils binaries used to evaluate TIE.

Does it work?

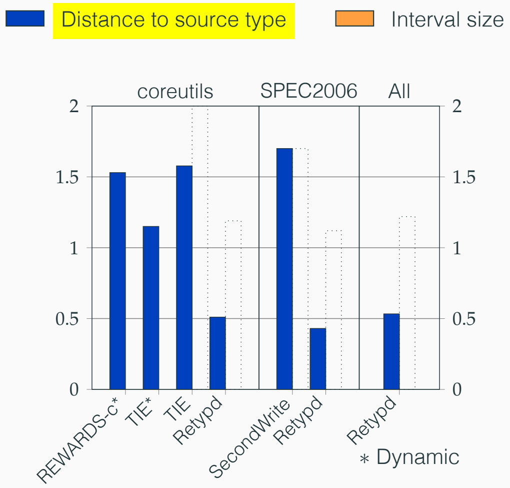


Retypd generates types that are:

- More accurate
- More tightly constrained

than existing approaches.

Does it work?

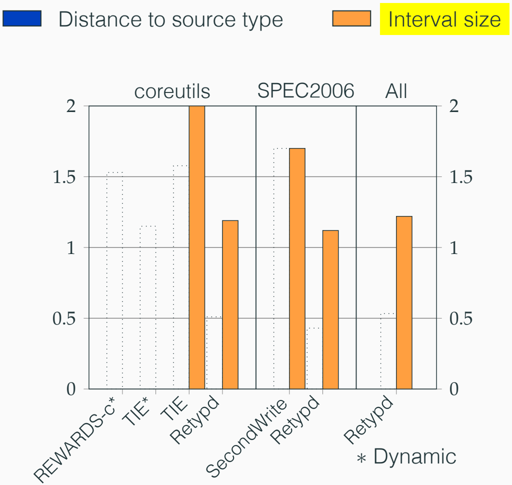


Retypd generates types that are:

- More accurate
- More tightly constrained

than existing approaches.

Does it work?

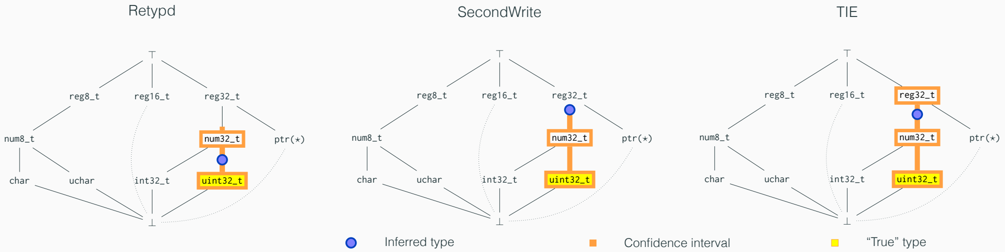


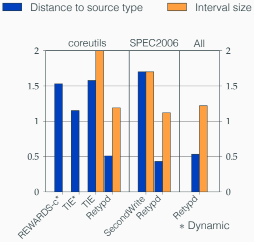
Retypd generates types that are:

- More accurate
- More tightly constrained

than existing approaches.

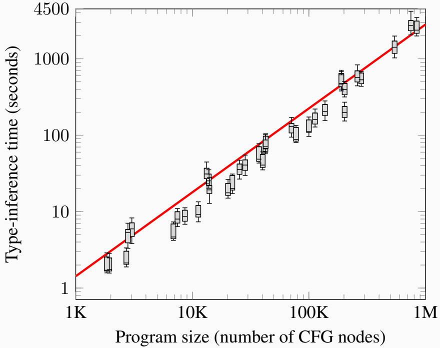
Visualizing the precision improvement

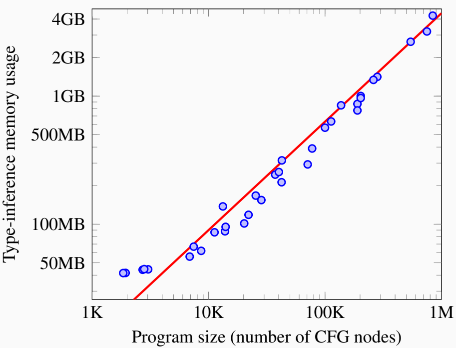




Performance measurements

Real-world performance scales like $N^{1.078}$ in time and $N^{0.882}$ in memory.





Summary

- Type reconstruction is impossible!
    - But we can do pretty well anyway.
- Common C and C++ idioms force us into an interesting corner of the type system design space, with
    - Subtypes
    - Recursive types
    - Polymorphic functions
- Reachability on certain kinds of infinite graphs gives us effective algorithms for manipulating constraint sets.

Thank you for your time!

Summary

- Type reconstruction is impossible!
  - But we can do pretty well anyway.
- Common C and C++ idioms force us into an interesting corner of the type system design space, with
  - Subtypes
  - Recursive types
  - Polymorphic functions
- Reachability on certain kinds of infinite graphs gives us effective algorithms for manipulating constraint sets.

Thank you for your time!

Summary

- Type reconstruction is impossible!
  - But we can do pretty well anyway.
- Common C and C++ idioms force us into an interesting corner of the type system design space, with
  - Subtypes
  - Recursive types
  - Polymorphic functions
- Reachability on certain kinds of infinite graphs gives us effective algorithms for manipulating constraint sets.

Thank you for your time!

Summary

- Type reconstruction is impossible!
  - But we can do pretty well anyway.
- Common C and C++ idioms force us into an interesting corner of the type system design space, with
  - Subtypes
  - Recursive types
  - Polymorphic functions
- Reachability on certain kinds of infinite graphs gives us effective algorithms for manipulating constraint sets.

Thank you for your time!

Summary

- Type reconstruction is impossible!
    - But we can do pretty well anyway.
- Common C and C++ idioms force us into an interesting corner of the type system design space, with
    - Subtypes
    - Recursive types
    - Polymorphic functions
- Reachability on certain kinds of infinite graphs gives us effective algorithms for manipulating constraint sets.

Thank you for your time!

References

K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In Proceedings of the 34th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), volume 48, pages 51–60. ACM, 2013.

J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS '11), 2011.

References II

E. Robbins, J. M. Howe, and A. King. Theory propagation and rational-trees. In Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming, pages 193–204. ACM, 2013.

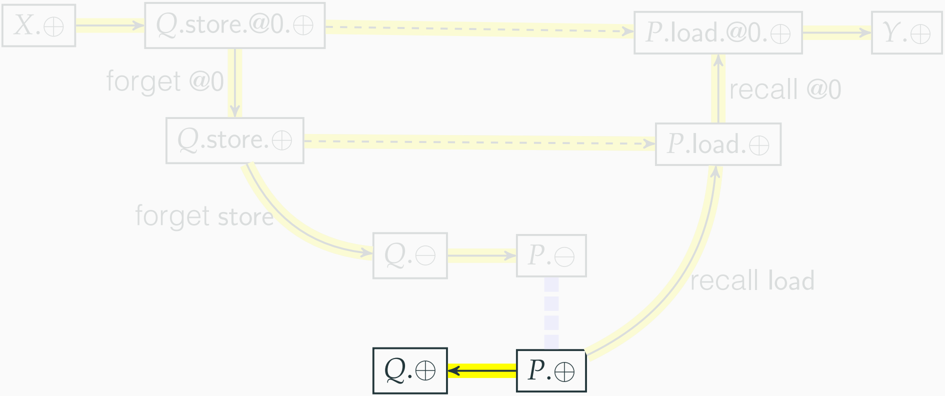
E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In Proceedings of the USENIX Security Symposium, page 16, 2013.

Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$

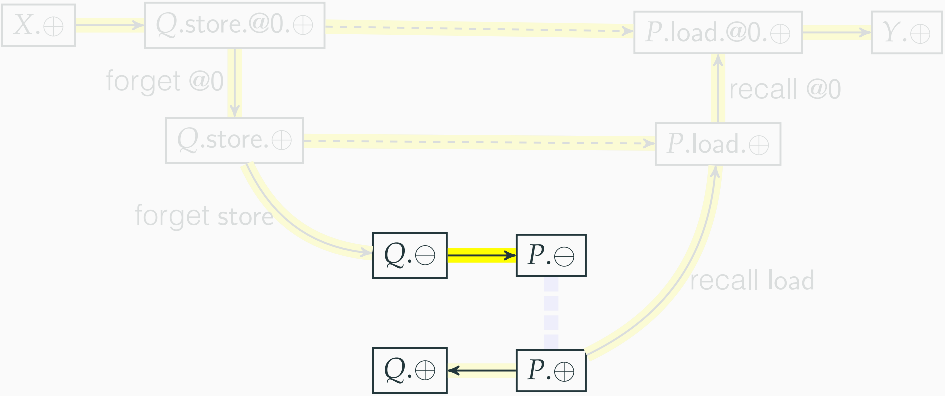
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\text{store}.@0, \quad P.\text{load}.@0 \sqsubseteq Y\}$$



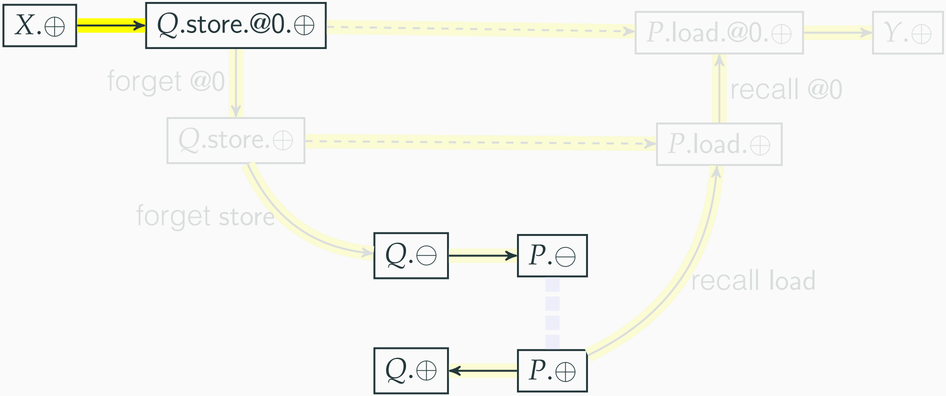
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$



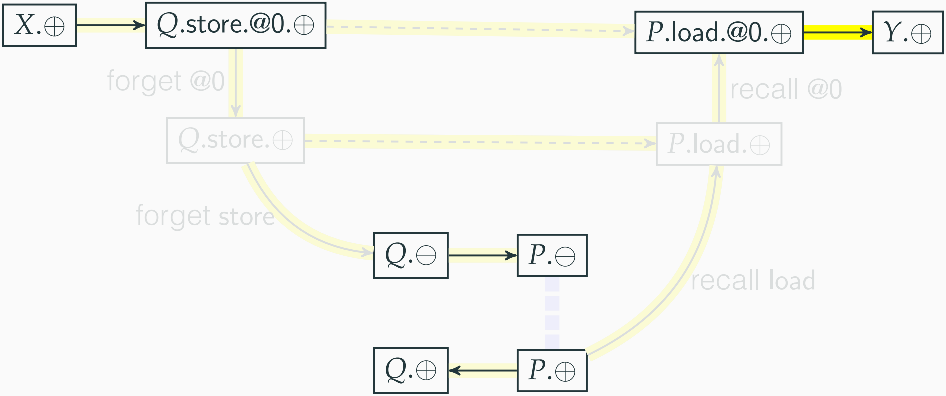
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\text{store}.@0, \quad P.\text{load}.@0 \sqsubseteq Y\}$$



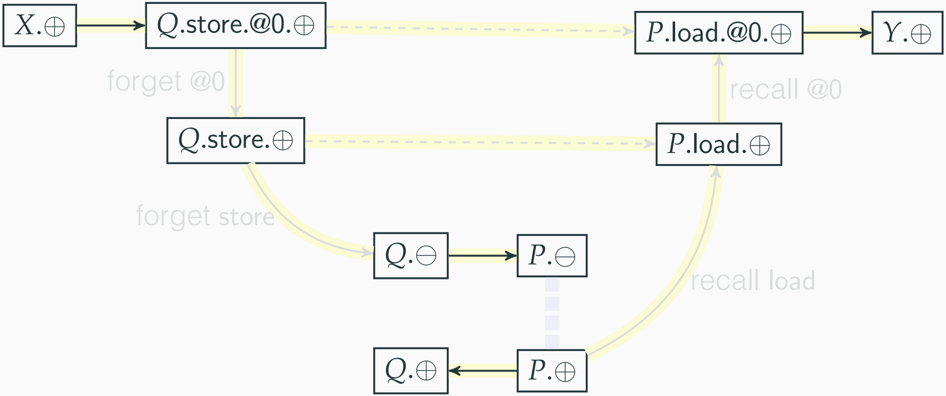
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathrm{store}.@0, \quad P.\mathrm{load}.@0 \sqsubseteq Y\}$$



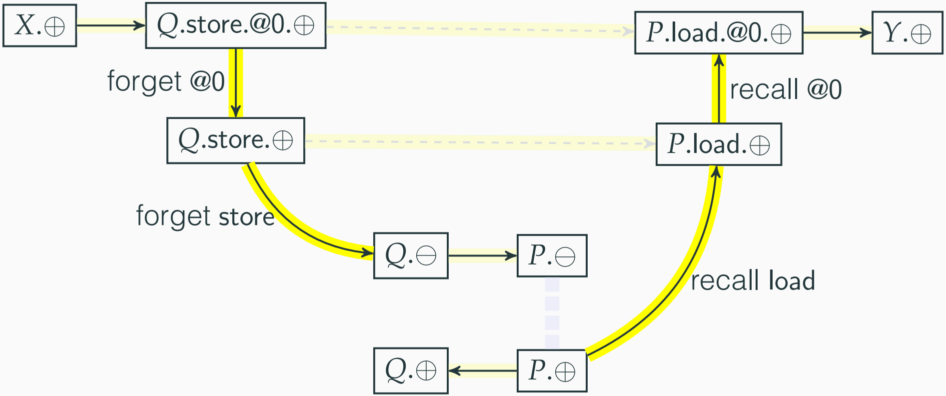
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathrm{store}.@0, \quad P.\mathrm{load}.@0 \sqsubseteq Y\}$$



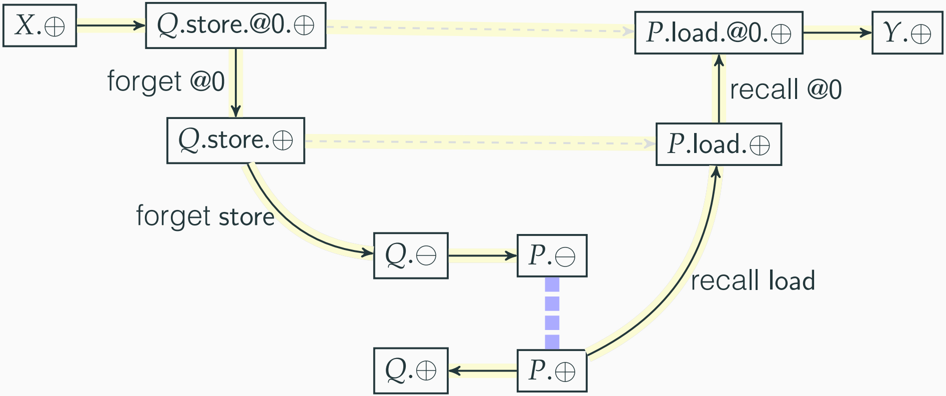
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$



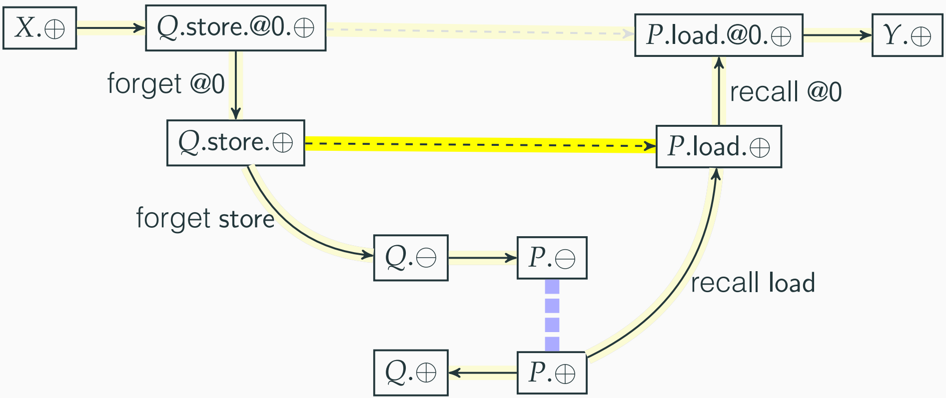
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$



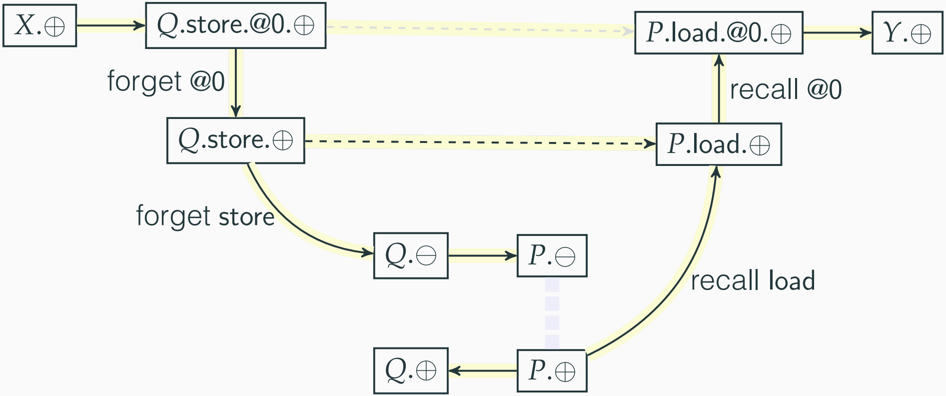
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$



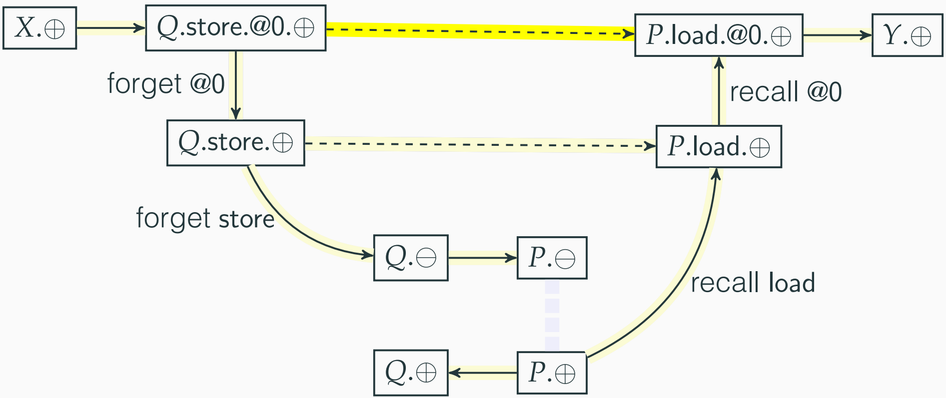
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathsf{store}.@0, \quad P.\mathsf{load}.@0 \sqsubseteq Y\}$$



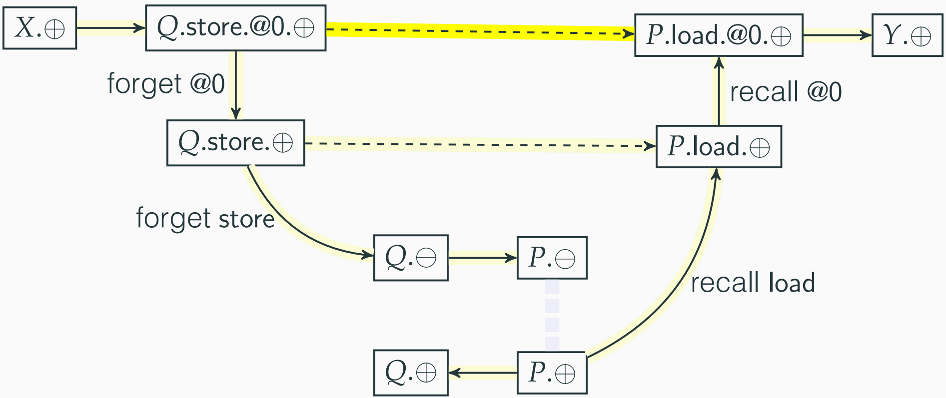
Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathrm{store}.@0, \quad P.\mathrm{load}.@0 \sqsubseteq Y\}$$

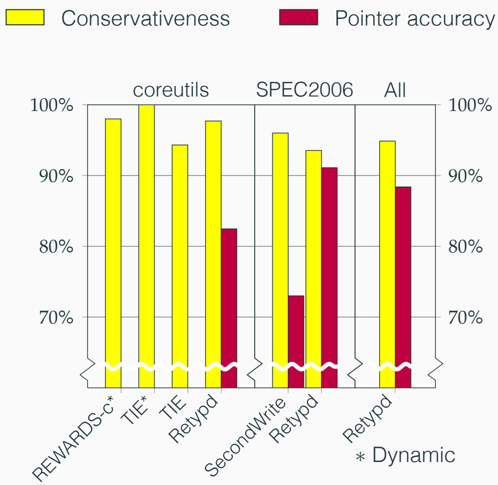


Adding the missing edges

$$\{P \sqsubseteq Q, \quad X \sqsubseteq Q.\mathrm{store}.@0, \quad P.\mathrm{load}.@0 \sqsubseteq Y\}$$



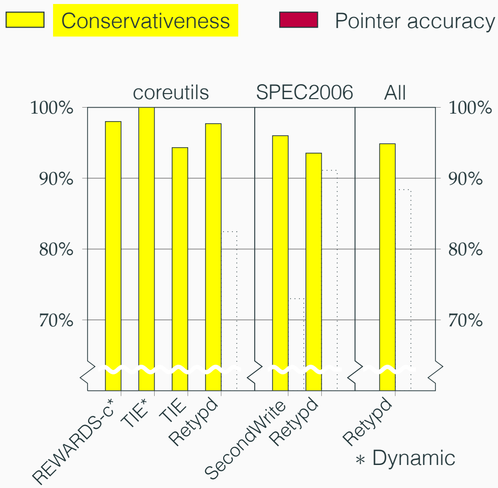
Does it work?



Retypd generates types that:

- Are comparably conservative to existing approaches
- Remain accurate at more levels of indirection [ElWazeer et al., 2013]

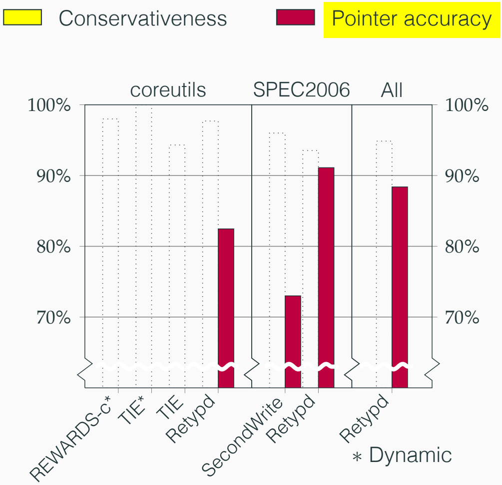
Does it work?



Retypd generates types that:

- Are comparably conservative to existing approaches
- Remain accurate at more levels of indirection [ElWazeer et al., 2013]

Does it work?



Retypd generates types that:

- Are comparably conservative to existing approaches
- Remain accurate at more levels of indirection [ElWazeer et al., 2013]