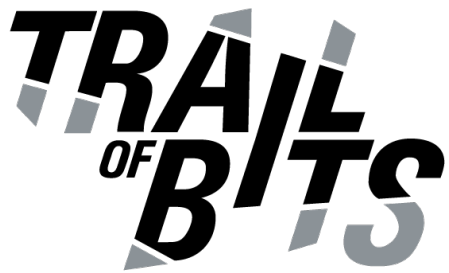
# BinSub: The Simple Essence of Polymorphic Type Inference for Machine Code

Ian Smith




April 3rd, 2025

# Introduction

# Decompilation without Types

```
void ** lookup(void *param_1,void *param_2)
{
  uint uVar1;
  int iVar2;
  void **local_18;

  uVar1 = key_hash(param_1,param_2);
  local_18 = *(void ***)(&DAT_00104040 + (ulong)(uVar1 & 0xfff) * 8);
  while( true ) {
    if (local_18 == (void **)0x0) {
      return (void **)0x0;
    }
    if ((((void *)(ulong)uVar1 == local_18[4]) && (param_2 == local_18[2])) &&
       (iVar2 = memcmp(param_1,*local_18,(size_t)param_2), iVar2 == 0)) break;
    local_18 = (void **)local_18[5];
  }
  return local_18;
}
```

# Decompilation with Types

```
struct_for_node_0_15 * lookup(void *param_1,size_t param_2)

{
  int iVar1;
  ulong uVar2;
  struct_for_node_0_15 *local_18;

  uVar2 = key_hash(param_1,param_2);
  local_18 = (&PTR_00104040)[(uint)(uVar2 & 0xffffffff) & 0xfff];
  while( true ) {
    if (local_18 == (struct_for_node_0_15 *)0x0) {
      return (struct_for_node_0_15 *)0x0;
    }
    if ((((uVar2 & 0xffffffff) == local_18->field_at_32)
           && (param_2 == local_18->field_at_16)) &&
       (iVar1 = memcmp(param_1,(void *)local_18->field_at_0,param_2),
       iVar1 == 0)) break;
    local_18 = local_18->field_at_40;
  }
  return local_18;
}
```

# Type Recovery

## Usecases

- Decompilation
- Dynamic Instrumentation
- Vulnerability Discovery

## Difficulties

- Lossy Compilation

Collection of difficulties leads to complex type systems and type inference algorithms

# Prior Work

# Hurdle 1: Subtyping

Where `f(void*, int)` and there is a call `f(NULL, 0)` [5]

```
xor eax, eax
push eax
push eax
call f
```

$[eax] = [p1]$
$[eax] = [p2]$
$p1 = void*$
$p2 = int$

$[eax] \leq [p1]$
$[eax] \leq [p2]$
$p1 \leq void*$
$p2 \leq int$

# Hurdle 2: Variance

We have created a new problem by introducing subtyping in a language with mutability [2].

## Covariant Treatment of Pointer Parameters

$\texttt{ptr}(\alpha) \leq \texttt{ptr}(\beta), x \leq \beta, \alpha \leq y$ covariant rule implies $\alpha \leq \beta$, cannot prove $x \leq y$

## Contravariant Treatment of Pointer Parameters

$\texttt{ptr}(\beta) \leq \texttt{ptr}(\alpha), x \leq \beta, \alpha \leq y$ contravariant rule implies $\alpha \leq \beta$, cannot prove $x \leq y$

## Split Capabilities

$$\frac{\text{Var } \alpha.\text{load} \qquad \text{Var } \alpha.\text{store}}{\alpha.\text{store} \leq \alpha.\text{load}}$$

Figure 1: S-Pointer rule in Retypd [5]

# Achieving these Features in Retypd [5]

$$\text{S-Field}_{\oplus}\ \frac{\alpha \leq_B \beta \qquad \text{Var}(\beta.l) \qquad < l >= \oplus}{\alpha.l \leq \beta.l}$$

$$\text{S-Field}_{\ominus}\ \frac{\alpha \leq_B \beta \qquad \text{Var}(\beta.l) \qquad < l >= \ominus}{\beta.l \leq \alpha.l}$$

Figure 2: Label rules for Retypd

- Field rules and transitivity derivations in a weighted pushdown automata
- Automata is saturated to a finite state transducer ($O(N^3)$)
- Path exploration in the transducer collects the set of reduced constraints between *interesting variables*

# Algebraic Subtyping: Expressivity and Efficiency

Are the features required by binary type inference vastly different from those explored in high-level languages?

## Algebraic Subtyping Features

- Parametric polymorphism
- Subtyping
- Principal type inference
- Recursive types

## Principal Hindley Milner Style Type Inference with Subtyping [3, 6]

- Defines a distributive lattice of types via union and intersection types
- Bifurcates types by polarity (negative/positive, input/output)
- When performing substitution performs substitution separately for negative and positive occurences of variables

# BinSub: Applying Algebraic Subtyping to the Binary Type Inference Problem

# Overview

- Consumes a lifted intermediate representation to produce type constraints (requires stack analysis etc [1])
- Generates type constraints (*In the Angr implementation generates Retypd constraints then translates*)
- Performs bi-unification to aggregate constraints and produce a canonical finite type for a given type variable
- Simplifies the type using a type automata representation, then lowers the simplified type to a C type
- Achieve compositional polymorphic type inference by cloning callsite type variables and then joining types

# Example Program

```
struct lst* x; // free var 1
void* y; // free var 2
struct lst* curr = x;
struct lst* cpy = y;
while(curr != NULL) {
    cpy = curr;
    curr = cpy->next;
    cpy->value++;
}
... // use cpy here
```

Figure 3: Example C program incrementing a linked list

# Example Intermediate Representation

```
block_1:
    1: stack_slot_1 = x;
    2: stack_slot_2 = y;

block_2:
    3: stack_slot_2 = stack_slot_1;
    4: t1 = load [stack_slot_2 + 4], 4;
    5: stack_slot_1 = t1;
    6: t2 = load [stack_slot_2], 4;
    7: t3 = t2 + 1
    8: store [stack_slot_2], t3
```

Figure 4: Example IR program incrementing a linked list

# Type Syntax

$$\tau ::= \{(n_0, n_1) : \tau, ...(n_{m-1}, n_m) : \tau\} | (n_0 : \tau, ...n_p : \tau) \to (n_{p+1} : \tau, ...n_r : \tau) | \texttt{ptr}(\tau, \tau) | \alpha | \top | \bot | \tau \sqcup \tau | \tau \sqcap \tau | \mu\alpha.\tau | \rho$$
$$\rho = \texttt{int64} | \texttt{int} | \texttt{float} | ...$$

Figure 5: BinSub Types

## Polarity Restrictions

$$\tau^+ = \{(n_0, n_1) : \tau^+, ...(n_{m-1}, n_m) : \tau^+\} | (n_0 : \tau^-, ...n_p : \tau^-) \to (n_{p+1} : \tau^+, ...n_r : \tau^+) | \texttt{ptr}(\tau^-, \tau^+) | \tau^+ \sqcup \tau^+ | \mu\alpha.t^+ | \bot$$

$$\tau^- = \{(n_0, n_1) : \tau^-, ...(n_{m-1}, n_m) : \tau^-\} | (n_0 : \tau^+, ...n_p : \tau^+) \to (n_{p+1} : \tau^-, ...n_r : \tau^-) | \texttt{ptr}(\tau^+, \tau^-) | \tau^- \sqcap \tau^- | \mu\alpha.t^- | \top$$

# Constraint Generation

```
1: x≤stack_slot_1
2: y≤stack_slot_2
3: stack_slot_1≤stack_slot_2
4₀: stack_slot_2≤ptr(a, {(4,4): b}) //ptr load
    ∧a≤{(4,4): b} //mantain store/load
4₁: b≤t1
5: t1≤stack_slot_1
6₀: stack_slot_2≤ptr(c, {(0,4): d})
    ∧c≤{(0,4): d}
6₁: d≤t2
7₀: t2≤int32
7₁: int32≤t3
8₀: stack_slot_2≤ptr({(0,4): e}, f)
    ∧{(0,4): e}≤f
8₁: t3≤e
```

Figure 6: Constraints generated from Figure 4

# Simple Bi-Unification

- Need to assign types to variables
- Variables have lower and upper bounds implied by the constraints and transitivity
- Polarity allows a constraint set to be represented finitely as $\bigsqcup \tau^+$ or $\bigsqcap \tau^-$

Original MLSub [3] work performed bi-unification over type automata directly performing substitutions, SimpleSub [6] takes a simplified approach:

- Decompose constraints implied by rules onto variables
- Coalesce constraints for a given variable, aggregating constraints depending on the polarity

# Decomposition Algorithm

Recursively decompose a constraint $\tau_0 \leq \tau_1$ into constraint set $C$:

**function** constrain($C, \tau_0, \tau_1$)
  **if** $\texttt{ptr}(a, b) = \tau_0$ & $\texttt{ptr}(c, d) = \tau_1$ **then**
    constrain($C, c, a$)
    constrain($C, b, d$)
  **else if** $\alpha = \tau_0$ **then**
    concat($C[\alpha]_{ub}, \tau_1$)
    **for all** $x \in C[\alpha]_{lb}$ **do**
      constrain($C, x, \tau_1$)
    **end for**
  **else if** $\alpha = \tau_1$ **then**
    concat($C[\alpha]_{lb}, \tau_0$)
    **for all** $x \in C[\alpha]_{ub}$ **do**
      constrain($C, \tau_0, x$)
    **end for**
  **end if**
**end function**

# Decomposition Example

```
constrain(t3, e) when int32 < t3, d > e, t2 > d
    t3: {upper: [e], lower: [int32]}
    constrain(int32, e)
        e: {upper: [d], lower: [int32]}
        constrain(int32, d)
            d: {upper: [t2], lower: [int32]}
            constrain(int32, t2)
                t2: {upper: [int32], lower: [int32]}
; Results
stack_slot_1: {upper: [stack_slot_2})]
                    lower: []}
stack_slot_2: {upper: [ ptr(a, {(4,4): b}),
                            ptr(c, {(0,4): d}),
                            ptr({(0,4): e}, f)]})]
                    lower: []}
t1: {upper: [stack_slot_1], lower: []}
b: {upper: [t1], lower: []}
```

# Coalescing Algorithm

**function** coalesce($C, \tau, p, R$)
  **if** $\texttt{ptr}(a, b) = \tau$ **then**
    **return** $\texttt{ptr}(\text{coalesce}(C, a, \neg p, R), \text{coalesce}(C, b, p, R))$
  **else if** $\alpha = \tau$ **then**
    **if** $\alpha^p \in R$ **then**
      **return** $\alpha$
    **end if**
    $S \leftarrow \alpha$
    $bounds = C[\alpha]_{\textbf{if } p \textbf{ then } lb \textbf{ else } ub}$
    $rec = \text{occurs}(\alpha^p, bounds)$
    $R' = R \cup \{\alpha^p\}$
    **for all** $x \in bounds$ **do**
      $y = \text{coalesce}(C, x, p, R')$
      **if** p **then**
        $S \leftarrow S \sqcup y$
      **else**
        $S \leftarrow S \sqcap y$
      **end if**
    **end for**
    **if** *rec* **then**
      **return** $\mu\alpha.S$
    **else**
      **return** $S$
    **end if**
  **end if**

# Simplification

$$\mu\alpha.\alpha \sqcap \texttt{stack\_slot\_2} \sqcap \texttt{ptr}(a, \{(4,4) : b \sqcap (t1 \sqcap \alpha)\}) \sqcap$$
$$\texttt{ptr}(c, \{(0,4) : d \sqcap (t2 \sqcap \texttt{int32})\}) \sqcap \texttt{ptr}(\{(0,4) : e \sqcup \texttt{int32}, f)$$

$$\mu\alpha.\{(0,4) : \texttt{int32}, (4,4) : \texttt{ptr}(\alpha)\}$$

## Simplification Procedure

- Construct an automata representing the type, associating the relevant constructors for each node
- Apply determinization and Hopcroft minimization, merging constructors when nodes are merged (via lattice operations)
- Decompile to a type by applying each constructor for a node recursively

# Type Automata

$H =$ constructor node labeling function
$Q =$ the sub-expressions of the type
$\Sigma = \{\epsilon, \texttt{FnIn}(n), \texttt{FnOut}(n)\ , \texttt{RecLabel}(n, m), \texttt{StoreLabel}, \texttt{LoadLabel}\}$ where $n, m \in \mathbb{N}$

$$q(\tau_0(\sqcap/\sqcup)\tau_1) \xrightarrow{\epsilon} q(\tau_0)$$
$$q(\tau_0(\sqcap/\sqcup)\tau_1) \xrightarrow{\epsilon} q(\tau_1)$$
$$q(\texttt{ptr}(\tau_0, \tau_1)) \xrightarrow{\texttt{StoreLabel}} q(\tau_0)$$
$$q(\texttt{ptr}(\tau_0, \tau_1)) \xrightarrow{\texttt{LoadLabel}} q(\tau_1)$$

$$q(\{...(n, m) : \tau...\}) \xrightarrow{\texttt{RecLabel}(n,m)} q(\tau)$$
$$q((...n : \tau...) \to (...)) \xrightarrow{\texttt{FnIn}(n)} q(\tau)$$
$$q((...) \to (...n : \tau...)) \xrightarrow{\texttt{FnOut}(n)} q(\tau)$$
$$q(\mu\alpha.\tau) \xrightarrow{\epsilon} q(\tau)$$

# Constructor Lattice

$$H(\tau) = (\texttt{polarity}(\tau), E(\tau))$$
$$E : \tau \to L$$
$$L : (T : \mathcal{P}(V) \times I : \mathcal{P}(\mathbb{N}) \times O : \mathcal{P}(\mathbb{N}) \times R : \mathcal{P}(\mathbb{N} \times \mathbb{N}) \times P : \mathcal{P}(\{p\}) \times A)$$
$$E(\alpha) = (\{\alpha\}, \emptyset, \emptyset, \emptyset, \emptyset, \bot_A)$$
$$E(\texttt{ptr}(\tau_0, \tau_1)) = (\emptyset, \emptyset, \emptyset, \emptyset, \{p\}, \bot_A)$$
$$E(\{...(n, m) : \tau...\}) = (\emptyset, \emptyset, \emptyset, \texttt{dom}(\{...(n, m) : \tau...\}), \emptyset, \bot_A)$$
$$E((...n : \tau, ...) \to (...m : \tau, ...)) = (\emptyset, \texttt{dom}((...n : \tau, ...)), \texttt{dom}((...m : \tau, ...)), \emptyset, \emptyset, \bot_A)$$
$$E(\rho) = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \rho)$$

$$\alpha \sqcup \beta = (.., \alpha_i \cap \beta_i, ..., \alpha_\rho \sqcup_A \beta_\rho)$$
$$\alpha \sqcap \beta = (.., \alpha_i \cup \beta_i, ..., \alpha_\rho \sqcap_A \beta_\rho)$$

$$M((\texttt{tt}, l_0), (\texttt{tt}, l_1)) = (\texttt{tt}, l_0 \sqcup l_1)$$
$$M((\texttt{ff}, l_0), (\texttt{ff}, l_1)) = (\texttt{ff}, l_0 \sqcap l_1)$$

# Example Simplified Automata

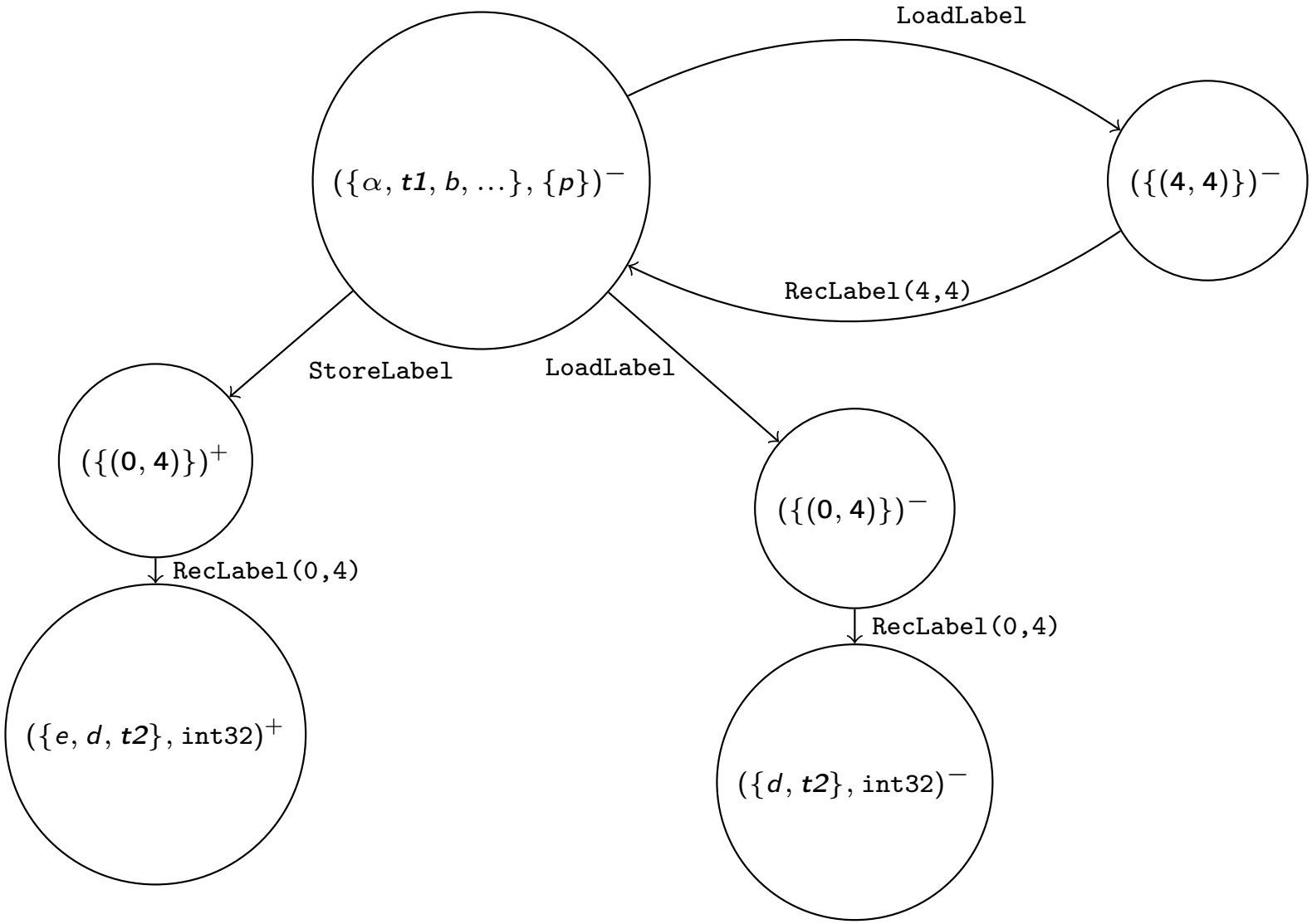



Figure 7: Simplified automata for `stack_slot_1`

# Lowering Automata

Normally type decompilation would lower automata back to BinSub types but we need to produce C types for decompilation.

## Lowering Procedure

- Break loops by inserting recursive variables that consume SCC edges
- Pick types for each node, starting at leaves
- Apply identities to merge multiple constructor instantiations as needed

# Lowering Identities

$$\{(a,b):c\} \sqcap \{(d,e):f\} = \begin{cases} \{(a,b):c \sqcap f\} & a = d \wedge b = e \\ \{(a,b):c,(d,e):f\} & \text{otherwise} \end{cases}$$

$$\texttt{ptr}(a,b) \sqcap \texttt{ptr}(c,d) = \texttt{ptr}((a \sqcup c) \sqcap (b \sqcap d), b \sqcap d)$$

$$(a:b) \to (c:d) \sqcap (e:f) \to (g:h) = \begin{cases} (a:b \sqcup f) \to (c:d \sqcap h) & a = e \wedge c = g \\ (a:b \sqcup f) \to (c:d, g:h) & a = e \wedge c \neq g \\ (a:b, e:f) \to (c:d \sqcap h) & a \neq e \wedge c = g \\ (a:b, e:f) \to (c:d, g:h) & \text{otherwise} \end{cases}$$

Collecting the recursive edges for Figure 7: $\{(0,4):\texttt{int32},(4,4):\texttt{ptr}(\beta)\}$

# Formal Comparison to Retypd

We want to compare the expressiveness of BinSub to Retypd given that they seem to preserve the same properties.

## Typing Retypd types as BinSub types:

$$\text{Pointer-Load}\ \frac{x : \texttt{ptr}(a, b) \qquad a \leq_B b}{x.\text{load} : b}$$

$$\text{Pointer-Store}\ \frac{x : \texttt{ptr}(a, b) \qquad a \leq_B b}{x.\text{store} : a}$$

# Comparing Typing Rules

$$\text{Pointer-}\oplus\text{-}\ominus \quad \frac{\triangleleft\Gamma \vdash \gamma \leq_B \alpha \qquad \triangleleft\Gamma \vdash \beta \leq_B \delta}{\Gamma \vdash \texttt{ptr}(\alpha, \beta) \leq_B \texttt{ptr}(\gamma, \delta)}$$

$$\text{S-Field}_{\oplus} \quad \frac{\alpha \leq_R \beta \qquad \mathsf{Var}(\beta.l) \qquad < l >= \oplus}{\alpha.l \leq_R \beta.l}$$

$$\text{S-Field}_{\ominus} \quad \frac{\alpha \leq_R \beta \qquad \mathsf{Var}(\beta.l) \qquad < l >= \ominus}{\beta.l \leq_R \alpha.l}$$

Figure 8: Pointer Rule for BinSub and Field Rules for Retypd

$$C \vdash x_R \leq_R y_R \implies \forall x_B, y_B.(x_R : x_B \wedge y_R : y_B \implies C' \vdash x_b \leq y_b)$$
where $C' = \{x_B \leq_B y_B | x_R : x_B \wedge y_R : y_B \wedge x_R \leq_R y_R \in C\}$

# Results

# Evaluation

## Implementation

- Angr implements Retypd type inference in Typehoon as part of Clinic
- Add an additional type solver that translates constraints to BinSub constraints and performs BinSub type inference
- Emit lowered C types and allow Clinic to apply these types

## Comparison

- Use ALLSTAR dataset and compare inferred type to ground truth signature
- Use type distance metric from prior work [4]
- ALLSTAR_1568 is the collection of functions that were assigned a type signature for each of the 7 microbenchmarks
- Microbenchmark at a 1 minute timeout, 43 function timeouts in BinSub and 161 in Typehoon with 19 overlapping timeouts

# Results I

Table 1: Complexity of Comparable Functions from ALLSTAR_1568

| Metric | Value (Constraints) |
|---|---|
| Max | 704 |
| Median | 18 |
| Mean | 35 |

| Metric | Value (Bytes) |
|---|---|
| Max | 3260 |
| Median | 96 |
| Mean | 175 |

Table 2: Evaluation Results on Comparable ALLSTAR_1568 Functions

| Algorithm | Average Type Distance | Average Runtime (seconds/function) |
|---|---|---|
| BinSub | 1.67 | .024 |
| Typehoon | 1.68 | 1.51 |

# Results II

95% CI pairwise mean percentage improvement: (87%, 88%)

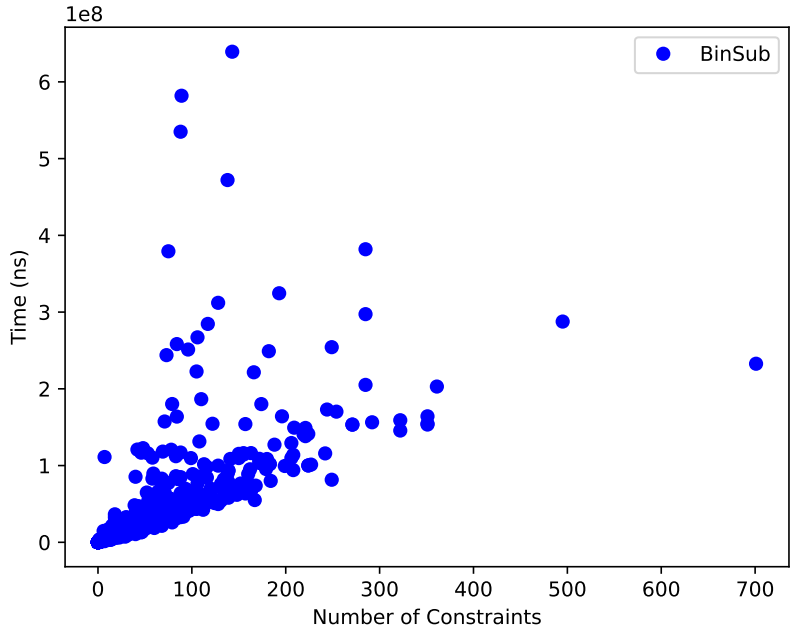



Figure 9: BinSub Time (1e8 ns) vs Size

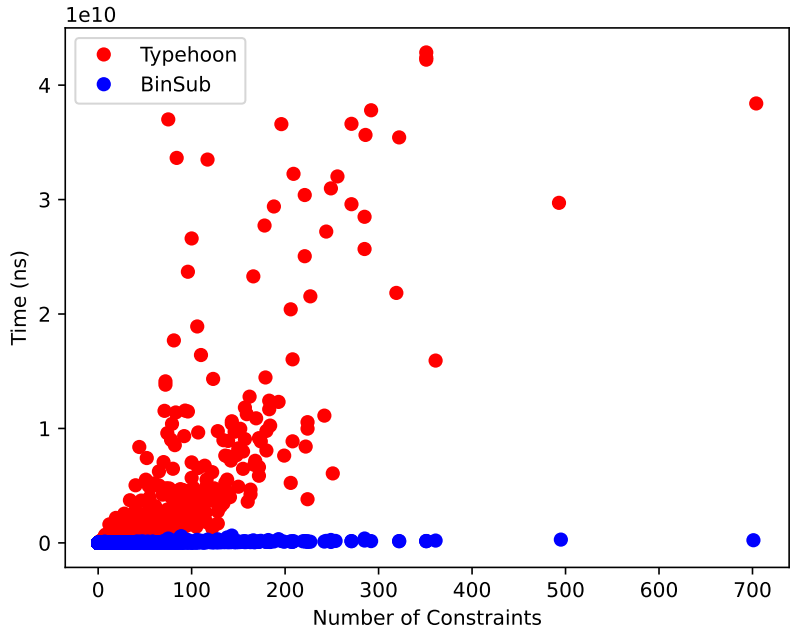



Figure 10: Time (1e10ns) vs Size

# Conclusions

- Subtyping, pointer variance, polymorphism, and recursive types can be captured in algebraic subtyping
- These features enable precise and efficient binary type inference: BinSub
- BinSubs type rules can be compared with Retypd showing a preservation of derivations
- Algebraic subtyping enables binary type inference that is efficient and connected with type inference in high-level languages

# References

# References I


[1] Gogul Balakrishnan and Thomas Reps. "DIVINE: DIscovering Variables IN Executables". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Byron Cook and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–28. isbn: 978-3-540-69738-1.

[2] Rowan Davies and Frank Pfenning. "Intersection types and computational effects". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 198–208. isbn: 1581132026. doi: 10.1145/351240.351259. url: https://doi.org/10.1145/351240.351259.

# References II


[3] Stephen Dolan and Alan Mycroft. "Polymorphism, subtyping, and type inference in MLsub". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 60–72. isbn: 9781450346603. doi: 10.1145/3009837.3009882. url: https://doi.org/10.1145/3009837.3009882.

[4] JongHyup Lee, Thanassis Avgerinos, and David Brumley. "TIE: Principled Reverse Engineering of Types in Binary Programs". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. url: https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs.

# References III


[5] Matt Noonan, Alexey Loginov, and David Cok. "Polymorphic type inference for machine code". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. Santa Barbara, CA, USA: Association for Computing Machinery, 2016, pp. 27–41. isbn: 9781450342612. doi: 10.1145/2908080.2908119. url: https://doi.org/10.1145/2908080.2908119.

[6] Lionel Parreaux. "The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). doi: 10.1145/3409006. url: https://doi.org/10.1145/3409006.