Polymorphic type inference for machine code *

Matthew Noonan Alexey Loginov David Cok
GrammaTech, Inc.
Ithaca NY, USA
{mnoonan,alexey,dcok}@grammatech.com

Abstract

For many compiled languages, source-level types are erased very early in the compilation process. As a result, further compiler passes may convert type-safe source into type-unsafe machine code. Type-unsafe idioms in the original source and type-unsafe optimizations mean that type information in a stripped binary is essentially nonexistent. The problem of recovering high-level types by performing type inference over stripped machine code is called *type reconstruction*, and offers a useful capability in support of reverse engineering and decompilation.

In this paper, we motivate and develop a novel type system and algorithm for machine-code type inference. The features of this type system were developed by surveying a wide collection of common source- and machine-code idioms, building a catalog of challenging cases for type reconstruction. We found that these idioms place a sophisticated set of requirements on the type system, inducing features such as recursively-constrained polymorphic types. Many of the features we identify are often seen only in expressive and powerful type systems used by high-level functional languages.

Using these type-system features as a guideline, we have developed Retypd: a novel static type-inference algorithm for machine code that supports recursive types, polymorphism, and subtyping. Retypd yields more accurate inferred types than existing algorithms, while also enabling new capabilities such as reconstruction of pointer `const` annotations with 98% recall. Retypd can operate on weaker program representations than the current state of the art, removing the need for high-quality points-to information that may be impractical to compute.

* This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
DISTRIBUTION A. Approved for public release; distribution unlimited.

DOI: http://dx.doi.org/10.1145/(to come)

1. Introduction

In this paper we introduce Retypd, a machine-code type inference tool that finds **re**gular **ty**pes using **p**ush**d**own systems. Retypd includes several novel features targeted at improved types for reverse engineering, decompilation, and high-level program analyses. These features include:

- Inference of most-general type schemes (§ 5)
- Inference of recursive structure types (figure 2)
- Sound analysis of pointer subtyping (§ 3.3)
- Tracking of customizable, high-level information such as purposes and typedef names (§ 3.5)
- Inference of type qualifiers such as `const` (§ 6.4)
- No dependence on high-quality points-to data (§ 6)
- More accurate recovery of source-level types (§ 6)

Retypd continues in the tradition of SecondWrite [8] and TIE [15] by introducing a principled static type inference algorithm applicable to stripped binaries. Diverging from previous work, we follow the principled type inference phase by a second phase that uses heuristics to "downgrade" the inferred types to human-readable C types before display. By factoring type inference into two phases, we can sequester unsound heuristics and quirks of the C type systems from the sound core of the type inference engine. This adds a degree of freedom to the design space so that we may leverage a relatively complex type system during type analysis, yet still emit familiar C types for the benefit of the reverse engineer.

Retypd operates on an intermediate representation (IR) recovered by automatically disassembling a binary using GrammaTech's static analysis tool CodeSurfer® for Binaries [4]. By generating type constraints from a TSL-based abstract interpreter [16], Retypd can operate uniformly on binaries for any platform supported by CodeSurfer, including x86, x86-64, and ARM.

During the development of Retypd, we carried out an extensive investigation of common machine-code idioms that

create challenges for existing type-inference methods. For each challenging case, we identified requirements for any type system that could correctly type the idiomatic code. The results of this investigation appear in § 2. The type system used by Retypd was specifically designed to satisfy these requirements. These common idioms pushed us into a far richer type system than we had first expected, including features like recursively constrained type schemes that have not previously been applied to machine code type inference.

2. Challenges

There are many challenges to carrying out type inference on machine code, and many common idioms that lead to sophisticated demands on the feature set of a type system. In this section, we describe several of the challenges seen during the development of Retypd that led to our particular combination of type system features.

2.1 Optimizations after type erasure

Since type erasure typically happens early in the compilation process, many compiler optimizations may take well-typed machine code and produce functionally equivalent but ill-typed results. We found that there were three common optimization techniques that required special care: the re-use of stack slots, the use of a variable as a syntactic constant, and early returns along error paths.

Re-use of stack slots: If a function uses two variables of the same size in disjoint scopes, there is no need to allocate two separate stack slots for those variables. Often the optimizer will reuse a stack slot from a variable that has dropped out of scope. This is true even if the new variable has a different type. This optimization even applies to the stack slots used to store formal-in parameters, as in Figure 2; once the function's argument is no longer needed, the optimizer can overwrite it with a local variable of an incompatible type.

More generally, we cannot assume that the map from program variables to physical locations is one-to-one. We cannot even make the weaker assumption that the program variables inhabiting a single physical location at different times will all belong to a single type.

Semi-syntactic constants: Suppose a function with signature `void f(int x, char* y)` is invoked as `f(0, NULL)`. This will usually be compiled to x86 machine code similar to

```
xor  eax, eax
push eax        ; y := NULL
push eax        ; x := 0
call f
```

This represents a code-size optimization, since `push eax` can be encoded in one byte instead of the five bytes needed to push an immediate value (0). We must be careful that the type variables for x and y are not unified; here, `eax` is being used more like a syntactic constant than a dynamic value that should be typed.

```
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
```

Figure 1. A common fortuitous reuse of a known value.

Fortuitous reuse of values: A related situation appears in the common control flow pattern represented by the snippet of C and the corresponding machine code in figure 1.
Note that on procedure exit, the return value in `eax` may have come from either the return value of `S2T` or from the return value of `get_S` (if `NULL`). If this situation is not detected, we will see a false relationship between the incompatible return types of `get_T` and `get_S`.

We handle these issues through a combination of type-system features (subtyping instead of unification) and program analyses (reaching definitions for stack variables and trace partitioning [19]).

2.2 Polymorphic functions

We were surprised to discover that, although not directly supported by the C type system, most programs define or make use of functions that are effectively polymorphic. The most well-known example is `malloc`: the return value is expected to be immediately cast to some other type `T*`. Each call to `malloc` may be thought of as returning some pointer of a different type. The type of `malloc` is effectively *not* $\texttt{size\_t} \to \texttt{void}*$, but rather $\forall\tau.\texttt{size\_t} \to \tau*$.

The problem of a polymorphic `malloc` could be mitigated by treating each call site p as a call to a distinct function $\texttt{malloc}_p$, each of which may have a distinct return type $\texttt{T}_p*$. Unfortunately it is not sufficient to treat a handful of special functions like `malloc` this way: it is common to see binaries that utilize user-defined allocators and wrappers to `malloc`. All of these functions would also need to be accurately identified and duplicated for each callsite.

A similar problem exists for functions like `free`, which is polymorphic in its lone parameter. Even more complex are functions like `memcpy`, which is polymorphic in its first two parameters and its return type, though the three types are not independent of each other. Furthermore, the polymorphic type signatures

$$\texttt{malloc} : \forall\tau.\texttt{size\_t} \to \tau*$$

$$\texttt{free} : \forall\tau.\tau* \to \texttt{void}$$

$$\texttt{memcpy} : \forall\alpha,\beta.(\beta \sqsubseteq \alpha) \Rightarrow (\alpha* \times \beta* \times \texttt{size\_t}) \to \alpha*$$

are all strictly more informative to the reverse engineer than the standard C signatures. How else could one know that the `void*` returned by `malloc` is not meant to be an opaque handle, but rather should be cast to some other pointer type?

In compiled C++ binaries, polymorphic functions are even more common. For example, a class member function must potentially accept both `base_t*` and `derived_t*` as types for `this`.

Foster, Johnson, Kodumal, and Aiken [9] noted that using bounded polymorphic type schemes for `libc` functions increased the precision of type-qualifier inference, at the level of source code. To advance the state of the art in machine-code type recovery, we believe it is important to also embrace polymorphic functions as a natural and common feature of machine code. Significant improvements to static type reconstruction — even for monomorphic types — will require the capability to infer polymorphic types of some nontrivial complexity.

2.3 Recursive types

The relevance of recursive types for decompilation was recently discussed by Schwartz et al. [27], where lack of a recursive type system for machine code was cited as an important source of imprecision. Since recursive data structures are relatively common, it is desirable that a type inference scheme for machine code be able to represent and infer recursive types natively.

2.4 Offset and reinterpreted pointers

Unlike in source code, there is no syntactic distinction in machine code between a pointer-to-`struct` and a pointer-to-first-member-of-`struct`. For example, if X has type `struct { char*, FILE*, size_t }*` on a 32-bit platform, then it should be possible to infer that $X + 4$ can be safely passed to `fclose`; conversely, if $X + 4$ is passed to `fclose` we may need to infer that X points to a structure that, at offset 4, contains a `FILE*`. This affects the typing of local structures, as well: a structure on the stack may be manipulated using a pointer to its starting address or by manipulating the members directly, e.g., through the frame pointer.

These idioms, along with casts from `derived*` to `base*`, fall under the general class of *physical* [28] or *non-structural* [21] subtyping. In Retypd, we model these forms of subtyping using type scheme specialization (§ 3.5). Additional hints about the extent of local variables are found using parameter-offset analysis [11].

2.5 Disassembly failures

The problem of producing correct disassembly for stripped binaries is equivalent to the halting problem. As a result, we can never assume that our reconstructed program representation will be perfectly correct. Even sound analyses built on top of an unsound program representation may exhibit inconsistencies and quirks.

Thus, we must be careful that incorrect disassembly or analysis results from one part of the binary will not influence the correct type results we may have gathered for the rest of the binary. Type systems that model value assignments as type unifications are vulnerable to over-unification issues caused by bad IR. Since unification is non-local, bad constraints in one part of the binary can degrade *all* type results.

Another instance of this problem arises from the use of register parameters. Although the x86 `cdecl` calling convention uses the stack for parameter passing, most optimized binaries will include many functions that pass parameters in registers for speed. Often, these functions do not conform to any standard calling convention. Although we work hard to ensure that only true register parameters are reported, conservativeness demands the occasional false positive.

Type reconstruction methods that are based on unification are generally sensitive to precision loss due to false positive register parameters. A common case is the "`push ecx`" idiom that reserves space for a single local variable in the stack frame of a function f. If `ecx` is incorrectly viewed as a register parameter of f in a unification-based scheme, whatever type variables are bound to `ecx` at each callsite to f will be falsely unified. In our early experiments, we found these overunifications to be a persistent and hard-to-diagnose source of imprecision.

In our early unification-based experiments, mitigation heuristics against overunification quickly ballooned into a disproportionately large and unprincipled component of type analysis. We designed Retypd's subtype-based constraint system to avoid the need for such ad-hoc prophylactics against overunification.

2.6 Cross-casting and bit twiddling

Even at the level of source code, there are already many type-unsafe idioms in common use. Most of these idioms operate by directly manipulating the bit representation of a value, either to encode additional information or to perform computations that are not possible using the type's usual interface. Some common examples include:

- hashing values by treating them as untyped bit blocks [1],
- stealing unused bits of a pointer for tag information, such as whether a thunk has been evaluated [18],
- reducing the storage requirements of a doubly-linked list by XOR-combining the `next` and `prev` pointers, and
- directly manipulating the bit representation of another type, as in the `quake3` inverse square root trick [26].

Because of these type-unsafe idioms, it is important that a type-inference scheme continues to produce useful results even in the presence of apparently contradictory constraints. We handle this situation in three ways:

1. separating the phases of constraint entailment, solving, and consistency checking,
2. modeling types with sketches (§ 3.5), which carry more information than C types, and
3. using unions to combine types with otherwise incompatible capabilities (e.g., τ is both `int`-like and pointer-like).

2.7 Incomplete points-to information

Degradation of points-to accuracy on large programs has been identified as a source of type-precision loss in other systems [8]. Our algorithm can provide high-quality types even in the presence of relatively weak points-to information. Precision can be further improved by increasing points-to knowledge via machine-code analyses such as VSA, but good results are already attained with no points-to analysis beyond the simpler problem of tracking the stack pointer.

2.8 Ad-hoc subtyping

Programs may define an ad-hoc type hierarchy via typedefs. This idiom appears in the Windows API, where a variety of handle types are all defined as typedefs of `void*`. Some of the handle types are to be used as subtypes of other handles; for example, a GDI handle (`HGDI`) is a generic handle used to represent any one of the more specific `HBRUSH`, `HPEN`, *etc*. In other cases, a typedef may indicate a *supertype*, as in `LPARAM` or `DWORD`; although these are typedefs of `int`, they have the intended semantics of a generic 32-bit type, which in different contexts may be used as a pointer, an integer, a flag set, and so on.

To accurately track ad-hoc hierarchies requires a type system based around subtyping rather than unification. Models for common API type hierarchies are useful; still better is the ability for the end user to define or adjust the initial type hierarchy at run time. We support this feature by parameterizing the main type representation by an uninterpreted lattice Λ, as described in § 3.5.

3. The type system

The type system used by Retypd is based around the inference of *recursively constrained type schemes* (§ 3.1). Solutions to constraint sets are modeled by *sketches* (§ 3.5); the sketch associated to a value consists of a record of capabilities which that value holds, such as whether it can be stored to, called, or accessed at a certain offset. Sketches also include markings drawn from a customizable lattice $(\Lambda, \vee, \wedge, <:)$, used to propagate high-level information such as typedef names and domain-specific purposes during type inference.

Retypd also supports recursively constrained type schemes that abstract over the set of types subject to a constraint set $\mathcal{C}$. The language of type constraints used by Retypd is weak enough that for any constraint set $\mathcal{C}$, satisfiability of $\mathcal{C}$ can be reduced (in cubic time) to checking a set of scalar constraints $\kappa_1 <: \kappa_2$, where κ_i are constants belonging to Λ.

Thanks to the reduction of constraint satisfiability to scalar constraint checking, we can omit expensive satisfiability checks during type inference. Instead, we delay the check until the final stage when internal types are converted to C types for display, providing a natural place to instantiate union types that resolve any inconsistencies. Since compiler optimizations and type-unsafe idioms in the original source frequently lead to program fragments with unsatisfiable type constraints (§ 2.5, § 2.6), this trait is particularly desirable.

Label	Variance	Capability
$.\mathsf{in}_L$	$\ominus$	Function with input in location L.
$.\mathsf{out}_L$	$\oplus$	Function with output in location L.
.load	$\oplus$	Readable pointer.
.store	$\ominus$	Writable pointer.
$.\sigma\mathsf{N@k}$	$\oplus$	Has N-bit field at offset k.

Table 1. Example field labels (type capabilities) in Σ.

3.1 Syntax: the constraint type system

Throughout this section, we fix a set $\mathcal{V}$ of *type variables*, an alphabet Σ of *field labels*, and a function $\langle\cdot\rangle : \Sigma \to \{\oplus, \ominus\}$ denoting the *variance* (Def. 3.2) of each label. We do not require the set Σ to be finite. Retypd makes use of a large set of labels; for simplicity, we will focus on those in table 1.

Within $\mathcal{V}$, we assume there is a distinguished set of *type constants*. These type constants are symbolic representations $\overline{\kappa}$ of elements κ belonging to some lattice, but are otherwise uninterpreted. It is usually sufficient to think of the type constants as type names or semantic tags.

Definition 3.1. A *derived type variable* is an expression of the form αw with $\alpha \in \mathcal{V}$ and $w \in \Sigma^*$.

Definition 3.2. The variance of a label ℓ encodes the subtype relationship between $\alpha.\ell$ and $\beta.\ell$ when α is a subtype of β, formalized in rules S-FIELD$_\oplus$ and S-FIELD$_\ominus$ of Figure 3. The variance function $\langle\cdot\rangle$ can be extended to Σ^* by defining $\langle\varepsilon\rangle = \oplus$ and $\langle xw\rangle = \langle x\rangle \cdot \langle w\rangle$, where $\{\oplus, \ominus\}$ is the sign monoid with $\oplus \cdot \oplus = \ominus \cdot \ominus = \oplus$ and $\oplus \cdot \ominus = \ominus \cdot \oplus = \ominus$. A word $w \in \Sigma^*$ is called *covariant* if $\langle w\rangle = \oplus$, or *contravariant* if $\langle w\rangle = \ominus$.

Definition 3.3. Let $\mathcal{V} = \{\alpha_i\}$ be a set of base type variables. A *constraint* is an expression of the form VAR X ("existence of the derived type variable X") or $X \sqsubseteq Y$ ("X is a subtype of Y"), where X and Y are derived type variables. A *constraint set over* $\mathcal{V}$ is a finite collection $\mathcal{C}$ of constraints, where the type variables in each constraint are either type constants or members of $\mathcal{V}$. We will say that $\mathcal{C}$ *entails* c, denoted $\mathcal{C} \vdash c$, if c can be derived from the constraints in $\mathcal{C}$ using the deduction rules of figure 3. We also allow projections: given a constraint set $\mathcal{C}$ with free variable τ, the projection $\exists\tau.\mathcal{C}$ binds τ as an "internal" variable in the constraint set. See τ in figure 2 for an example or, for a more in-depth treatment of constraint projection, see Su et al. [31].

The field labels used to form derived type variables are meant to represent *capabilities* of a type. For example, the constraint VAR $\alpha.\mathsf{load}$ means α is a readable pointer, and the derived type variable $\alpha.\mathsf{load}$ represents the type of the memory region obtained by loading from α.

Let us briefly see how operations in the original program translate to type constraints, using C syntax for clarity. The

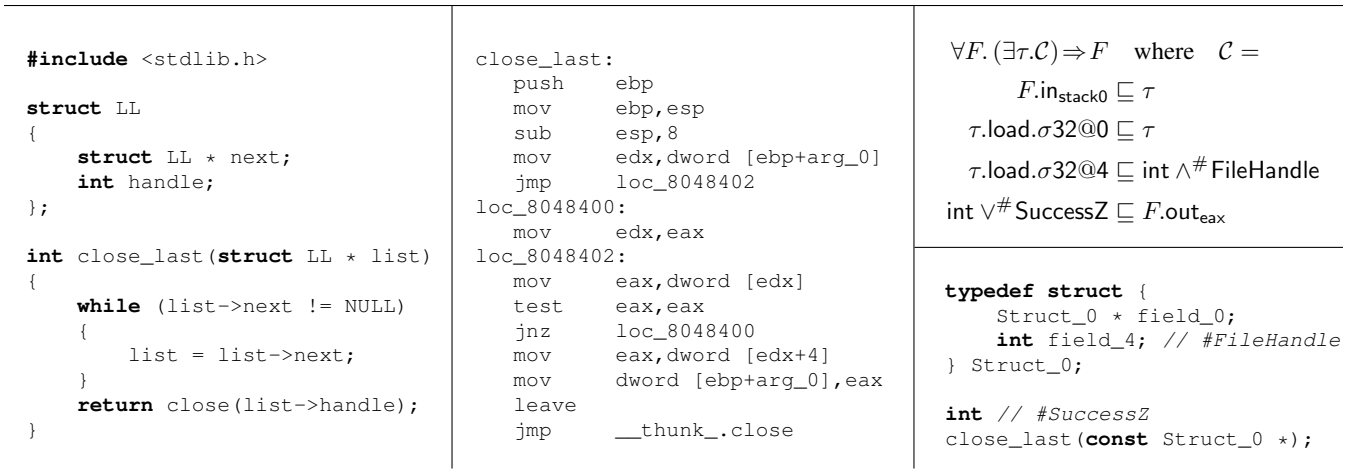


Figure 2. Example C code, disassembly, inferred type scheme, and reconstructed C type. The code was compiled with `gcc` 4.5.4 on Linux with flags `-m32 -O2`. The tags #FileHandle and #SuccessZ encode inferred higher-level purposes.

full conversion from disassembly to type constraints is described in appendix A.

Value copies: When a value is moved between program variables in an assignment like `x := y`, we make the conservative assumption that the type of `x` may be upcast to a supertype of `y`. We will generate a constraint of the form $Y \sqsubseteq X$.

Loads and stores: Suppose that `p` is a pointer to a 32-bit type, and a value is loaded into `x` by the assignment `x := *p`. Then we will generate a constraint of the form $P.\mathsf{load}.\sigma 32@0 \sqsubseteq X$. Similarly, a store `*q := y` results in the constraint $Y \sqsubseteq Q.\mathsf{store}.\sigma 32@0$.

In some of the pointer-based examples in this paper we omit the final $.\sigma\mathsf{N}@\mathsf{k}$ access after a .load or .store to simplify the presentation.

Function calls: Suppose the function f is invoked by `y := f(x)`. We will generate the constraints $X \sqsubseteq F.\mathsf{in}$ and $F.\mathsf{out} \sqsubseteq Y$, reflecting the flow of actuals to and from formals. Note that if we define $A.\mathsf{in} = X$ and $A.\mathsf{out} = Y$ then the two constraints are equivalent to $F \sqsubseteq A$ by the rules of figure 3. This encodes the fact that the called function's type must be at least as specific as the type used at the callsite.

One of the primary goals of our type inference engine is to associate to each procedure a type scheme.

Definition 3.4. A *type scheme* is an expression of the form $\forall\overline{\alpha}.\mathcal{C} \Rightarrow \alpha_1$ where $\forall\overline{\alpha} = \forall\alpha_1 \ldots \forall\alpha_n$ is quantification over a set of type variables, and $\mathcal{C}$ is a constraint set over $\{\alpha_i\}_{i=1..n}$.

Type schemes provide a way of encoding the pre- and post-conditions that a function places on the types in its calling context. Without the constraint sets, we would only be able to represent conditions of the form "the input must be a subtype of X" and "the output must be a supertype of Y". The constraint set $\mathcal{C}$ can be used to encode more interesting type relations between inputs and outputs, as in the case of `memcpy` (§ 2.2). For example, a function that returns the second 4-byte element from a `struct*` may have the type scheme $\forall\tau.\,(\tau.\mathsf{in}.\mathsf{load}.\sigma 32@4 \sqsubseteq \tau.\mathsf{out}) \Rightarrow \tau$.

3.2 Deduction rules

The deduction rules for our type system appear in Figure 3. Most of the rules are self-evident under the interpretation in Def. 3.3, but a few require some additional motivation.

S-FIELD$_\oplus$ and S-FIELD$_\ominus$: These rules ensure that field labels act as co- or contra-variant type operators, generating subtype relations between derived type variables from subtype relations between the original variables.

T-INHERITL and T-INHERITR: The rule T-INHERITL should be uncontroversial, since a subtype should have all capabilities of its supertype. The rule T-INHERITR is more unusual since it moves capabilities in the other direction; taken together, these rules require that two types in a subtype relation must have exactly the same set of capabilities. This is a form of structural typing, ensuring that comparable types have the same shape.

Structural typing appears to be at odds with the need to cast more capable objects to less capable ones, as described in § 2.4. Indeed, T-INHERITR eliminates the possibility of forgetting capabilities during value assignments. But we still maintain this capability at procedure invocations due to our use of polymorphic type schemes. An explanation of how type-scheme instantiation enables us to forget fields of an object appears in § 3.4, with more details in § E.1.2.

These rules ensure that Retypd can perform "iterative variable recovery"; lack of iterative variable recovery was cited by the authors of the Phoenix decompiler [27] as a common cause of incorrect decompilation when using TIE [15] for type recovery.

S-POINTER: This rule is a consistency condition ensuring that the type that can be loaded from a pointer is a supertype of the type that can be stored to a pointer. Without this rule, pointers would provide a channel for subverting the type

Derived Type Variable Formation

$$\frac{\alpha \sqsubseteq \beta}{\text{VAR } \alpha} \text{ (T-LEFT)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \text{VAR } \alpha.\ell}{\text{VAR } \beta.\ell} \text{ (T-INHERITL)}$$

$$\frac{\alpha \sqsubseteq \beta}{\text{VAR } \beta} \text{ (T-RIGHT)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \text{VAR } \beta.\ell}{\text{VAR } \alpha.\ell} \text{ (T-INHERITR)}$$

$$\frac{\text{VAR } \alpha.\ell}{\text{VAR } \alpha} \text{ (T-PREFIX)}$$

Subtyping

$$\frac{\text{VAR } \alpha}{\alpha \sqsubseteq \alpha} \text{ (S-REFL)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \text{VAR } \beta.\ell, \quad \langle \ell \rangle = \oplus}{\alpha.\ell \sqsubseteq \beta.\ell} \text{ (S-FIELD}_\oplus\text{)}$$

$$\frac{\alpha \sqsubseteq \beta, \quad \beta \sqsubseteq \gamma}{\alpha \sqsubseteq \gamma} \text{ (S-TRANS)} \qquad \frac{\alpha \sqsubseteq \beta, \quad \text{VAR } \beta.\ell, \quad \langle \ell \rangle = \ominus}{\beta.\ell \sqsubseteq \alpha.\ell} \text{ (S-FIELD}_\ominus\text{)}$$

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

Figure 3. Deduction rules for the type system. α, β, γ represent derived type variables; ℓ represents a label in Σ.

system. An example of how this rule is used in practice appears in § 3.3 below.

The deduction rules of figure 3 are simple enough that each proof may be reduced to a normal form (see B.1). An encoding of the normal forms as transition sequences in a modified pushdown system is used to provide a compact representation of the entailment closure $\overline{\mathcal{C}} = \{c \mid \mathcal{C} \vdash c\}$. The pushdown system modeling $\overline{\mathcal{C}}$ is queried and manipulated to provide most of the interesting type inference functionality. An outline of this functionality may be found in § 5.2.

3.3 Modeling pointers

To model pointers soundly in the presence of subtyping, we found that our initial naïve approach suffered from unexpected difficulties when combined with subtyping. Following the C type system, it seemed natural to model pointers by introducing an injective unary type constructor Ptr, so that $\mathsf{Ptr}(T)$ is the type of pointers to T. In a unification-based type system, this approach works as expected.

In the presence of subtyping, the issue becomes trickier. Consider the two programs in figure 4. Since the type variables P and Q associated to `p`, `q` can be seen to be pointers, we can begin by writing $P = \mathsf{Ptr}(\alpha), Q = \mathsf{Ptr}(\beta)$. The first program will generate the constraint set $\mathcal{C}_1 = \{\mathsf{Ptr}(\beta) \sqsubseteq \mathsf{Ptr}(\alpha), X \sqsubseteq \alpha, \beta \sqsubseteq Y\}$ while the second generates $\mathcal{C}_2 = \{\mathsf{Ptr}(\beta) \sqsubseteq \mathsf{Ptr}(\alpha), X \sqsubseteq \beta, \alpha \sqsubseteq Y\}$. Since each program has the effect of copying the value in x to y, both constraint sets should satisfy $\mathcal{C}_i \vdash X \sqsubseteq Y$. To do this, the pointer subtype constraint must entail some constraint on α and β, but which one?

If we assume that Ptr is covariant, then $\mathsf{Ptr}(\beta) \sqsubseteq \mathsf{Ptr}(\alpha)$ entails $\beta \sqsubseteq \alpha$ and so $\mathcal{C}_2 \vdash X \sqsubseteq Y$, but $\mathcal{C}_1 \nvdash X \sqsubseteq Y$. On the other hand, if we make Ptr contravariant then $\mathcal{C}_1 \vdash X \sqsubseteq Y$ but $\mathcal{C}_2 \nvdash X \sqsubseteq Y$.

It seems that our only recourse is to make subtyping degenerate to type equality under Ptr: we are forced to declare that $\mathsf{Ptr}(\beta) \sqsubseteq \mathsf{Ptr}(\alpha) \vdash \alpha = \beta$, which of course means that $\mathsf{Ptr}(\beta) = \mathsf{Ptr}(\alpha)$ already. This is a catastrophe for subtyping as used in machine code, since many natural subtype relations are mediated through pointers. For example, the unary Ptr constructor cannot handle the simplest kind of C++ class subtyping, where a derived class physically extends a base class by appending new member variables.

```
f() {
   p =  q;
  *p =  x;
   y = *q;
}
```

```
g() {
   p =  q;
  *q =  x;
   y = *p;
}
```

Figure 4. Two programs, each mediating a copy from `x` to `y` through a pair of aliased pointers.

The root cause of the difficulty seems to be in conflating two capabilities that (most) pointers have: the ability to be written through, and the ability to be read through. In Retypd, these two capabilities are modeled using different field labels .store and .load. The .store label is contravariant, while the .load label is covariant.

To see how the separation of pointer capabilities avoids the loss of precision suffered by Ptr, we revisit the two example programs. The first generates the constraint set

$$\mathcal{C}_1' = \{Q \sqsubseteq P, \quad X \sqsubseteq P.\mathsf{store}, \quad Q.\mathsf{load} \sqsubseteq Y\}$$

By rule T-INHERITR we may conclude that Q also has a field of type .store. By S-POINTER we can infer that $Q.\mathsf{store} \sqsubseteq Q.\mathsf{load}$. Finally, since .store is contravariant and $Q \sqsubseteq P$, S-FIELD$_\ominus$ says we also have $P.\mathsf{store} \sqsubseteq Q.\mathsf{store}$. Putting these parts together gives the subtype chain

$$X \sqsubseteq P.\mathsf{store} \sqsubseteq Q.\mathsf{store} \sqsubseteq Q.\mathsf{load} \sqsubseteq Y$$

The second program generates the constraint set

$$\mathcal{C}_2' = \{Q \sqsubseteq P, \quad X \sqsubseteq Q.\mathsf{store}, \quad P.\mathsf{load} \sqsubseteq Y\}$$

Since $Q \sqsubseteq P$ and P has a field .load, we conclude that Q has a .load field as well. Next, S-POINTER requires that $Q.\mathsf{store} \sqsubseteq Q.\mathsf{load}$. Since .load is covariant, $Q \sqsubseteq P$ implies that $Q.\mathsf{load} \sqsubseteq P.\mathsf{load}$. This gives the subtype chain

$$X \sqsubseteq Q.\mathsf{store} \sqsubseteq Q.\mathsf{load} \sqsubseteq P.\mathsf{load} \sqsubseteq Y$$

By splitting out the read- and write-capabilities of a pointer, we can achieve a sound account of pointer subtyping that does not degenerate to type equality. Note the importance of the consistency condition S-POINTER: this

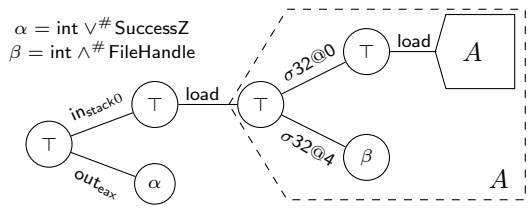


Figure 5. A sketch instantiating the type scheme in Fig. 2.

rule ensures that writing to a pointer and reading a result out cannot subvert the type system.

The need for separate handling of read- and write-capabilities in a mutable reference has been rediscovered multiple times. A well-known instance is the covariance of the array type constructor in Java and C#, which can cause runtime type errors if the array is mutated; in these languages, the read capabilities are soundly modeled only by sacrificing soundness for the write capabilities.

3.4 Non-structural subtyping and T-InheritR

It was noted in § 3.2 that the rule T-InheritR leads to a system with a form of structural typing: any two types in a subtype relation must have the same capabilities. Superficially, this seems problematic for modeling typecasts that forget about fields, such as a cast from `derived*` to `base*` when `derived*` has additional fields (§ 2.4).

The missing piece that allows us to effectively forget capabilities is instantiation of callee type schemes at a call-site. To demonstrate how polymorphism enables forgetfulness, consider the example type scheme $\forall F.\,(\exists \tau.\mathcal{C}) \Rightarrow F$ from figure 2. The function `close_last` can be invoked by providing any actual-in type α such that $\alpha \sqsubseteq F.\mathsf{in}_{\mathsf{stack0}}$; in particular, α can have *more* fields than those required by $\mathcal{C}$. We simply select a more capable type for τ in $\mathcal{C}$. In effect, we have used *specialization* of polymorphic types to model *non-structural* subtyping idioms, while *subtyping* is used only to model *structural* subtyping idioms. This restricts our introduction of non-structural subtypes to points where a type scheme is instantiated, such as at a call site.

3.5 Semantics: the poset of sketches

The simple type system defined by the deduction rules of figure 3 defines the *syntax* of legal derivations in our type system. The constraint solver of § 5.2 is designed to find a simple representation for all conclusions that can be derived from a set of type constraints. Yet there is no notion of what a type *is* inherent to the deduction rules of figure 3. We have defined the rules of the game, but not the equipment with which it should be played.

We found that introducing C-like entities at the level of constraints or types resulted in too much loss of precision when working with the challenging examples described in § 2. Consequently we developed the notion of a *sketch*, a certain kind of regular tree labeled with elements of an auxiliary lattice Λ. Sketches are related to the recursive types studied in Amadio and Cardelli [3] and Kozen et al. [14], but do not depend on *a priori* knowledge of the ranked alphabet of type constructors.

Definition 3.5. A *sketch* is a (possibly infinite) tree T with edges labeled by elements of Σ and nodes marked with elements of a lattice Λ, such that T only has finitely many subtrees up to labeled isomorphism. By collapsing isomorphic subtrees, we can represent sketches as deterministic finite state automata with each state labeled by an element of Λ. The set of sketches admits a lattice structure, with operations described in appendix E.

The lattice of sketches serve as the model in which we interpret type constraints. The interpretation of the constraint VAR $\alpha.u$ is "the sketch S_α admits a path from the root with label sequence u", and $\alpha.u \sqsubseteq \beta.v$ is interpreted as "the sketch obtained from S_α by traversing the label sequence u is a subsketch (in the lattice order) of the sketch obtained from S_β by traversing the sequence v."

The main utility of sketches is that they are nearly a free tree model [22] of the constraint language. Any constraint set $\mathcal{C}$ is satisfiable over the lattice of sketches, as long as $\mathcal{C}$ cannot prove an impossible subtype relation in the auxiliary lattice Λ. In particular, we can always solve the fragment of $\mathcal{C}$ that does not reference constants in Λ. Stated operationally, we can always recover the tree structure of sketches that potentially solve $\mathcal{C}$. This observation is formalized by the following theorem:

Theorem 3.1. Suppose that $\mathcal{C}$ is a constraint set over the variables $\{\tau_i\}_{i\in I}$. Then there exist sketches $\{S_i\}_{i\in I}$ such that $w \in S_i$ if and only if $\mathcal{C} \vdash \text{VAR } \tau_i.w$.

Proof. The idea is to symmetrize $\sqsubseteq$ using an algorithm that is similar in spirit to Steensgaard's method of almost-linear-time pointer analysis [30]. Begin by forming a graph with one node $n(\alpha)$ for each derived type variable appearing in $\mathcal{C}$, along with each of its prefixes. Add a labeled edge $n(\alpha) \xrightarrow{\ell} n(\alpha.\ell)$ for each derived type variable $\alpha.\ell$ to form a graph G. Now quotient G by the equivalence relation $\sim$ defined by $n(\alpha) \sim n(\beta)$ if $\alpha \sqsubseteq \beta \in \mathcal{C}$, and $n(\alpha') \sim n(\beta')$ whenever there are edges $n(\alpha) \xrightarrow{\ell} n(\alpha')$ and $n(\beta) \xrightarrow{\ell'} n(\beta')$ in G with $n(\alpha) \sim n(\beta)$ where either $\ell = \ell'$ or $\ell = .\mathsf{load}$ and $\ell' = .\mathsf{store}$.

By construction, there exists a path through $G/\sim$ with label sequence u starting at the equivalence class of τ_i if and only if $\mathcal{C} \vdash \text{VAR } \tau_i.u$; the (regular) set of all such paths yields the tree structure of S_i. □

Working out the lattice elements that should label S_i is a trickier problem; the basic idea is to use the same automaton Q constructed during constraint simplification (Theorem 5.1) to answer queries about which type constants are upper and lower bounds on a given derived type variable. The full algorithm is listed in appendix D.4.

In Retypd, we use a large auxiliary lattice Λ containing hundreds of elements that includes a collection of standard C type names, common typedefs for popular APIs, and user-specified semantic classes such as #FileHandle in figure 2. This lattice helps model ad-hoc subtyping and preserve high-level semantic type names, as discussed in § 2.8.

Note. Sketches are just one of many possible models for the deduction rules that could be proposed. A general approach is to fix a poset $(\mathcal{T}, <:)$ of types, interpret $\sqsubseteq$ as $<:$, and interpret co- and contra-variant field labels as monotone (resp. antimonotone) functions $\mathcal{T} \to \mathcal{T}$.

The separation of syntax from semantics allows for a simple way to parameterize the type inference engine by a model of types. By choosing a model $(\mathcal{T}, \equiv)$ with a symmetric relation $\equiv \subseteq \mathcal{T} \times \mathcal{T}$, a unification-based type system similar to SecondWrite [8] is generated. On the other hand, by forming a lattice of type intervals and interval inclusion, we may construct a type system similar to TIE [15] that outputs upper and lower bounds on each type variable.

4. Analysis framework

Retypd is built on top of GrammaTech's machine-code analysis tool CodeSurfer for Binaries. CodeSurfer carries out common program analyses on binaries for multiple CPU architectures, including x86, x86-64, and ARM. CodeSurfer is used to recover a high-level IR from the raw machine code; type constraints are generated directly from this IR, and resolved types are applied back to the IR and become visible to the GUI and later analysis phases.

CodeSurfer achieves platform independence through TSL [16], a language for defining a processor's concrete semantics in terms of concrete numeric types and mapping types that model register- and memory banks. Interpreters for a given abstract domain are automatically created from the concrete semantics simply by specifying the abstract domain $\mathcal{A}$ and an interpretation of the concrete numeric and mapping types. Retypd uses CodeSurfer's recovered IR to determine the number and location of inputs and outputs to each procedure, as well as the program's call graph and per-procedure control-flow graphs. An abstract interpreter then generates sets of type constraints from the concrete TSL instruction semantics. A detailed account of the abstract semantics for constraint generation appears in Appendix A.

After the initial IR is recovered, type inference proceeds in two stages: first, type constraint sets are generated in a bottom-up fashion over the strongly-connected components of the callgraph. Pre-computed type schemes for externally linked functions may be inserted at this stage. Each constraint set is simplified by eliminating type variables that do not belong to the SCC's interface; the simplification algorithm is outlined in section § 5 (see algorithm F.1 for full details). Once type schemes are available, the callgraph is traversed bottom-up, assigning sketches to type variables as outlined in § 3.5. During this stage, type schemes are specialized based on the calling contexts of each function. Appendix F lists the full algorithms for constraint simplification (F.1) and solving (F.2).

The final phase of type resolution converts the inferred sketches to C types for presentation to the user. Since C types and sketches are not directly comparable, this resolution phase necessarily involves the application of heuristic conversion policies.

Restricting the heuristic policies to a single post-inference phase provides us with substantial flexibility to generate high-quality, human-readable C types while maintaining soundness and generality during type reconstruction.

5. The simplification algorithm

In this section, we sketch an outline of the simplification algorithm at the core of the constraint solver. The complete algorithm can be found in Appendix D.

5.1 Inferring a type scheme

The goal of the simplification algorithm is to take an inferred type scheme $\forall \overline{\alpha}.\, C \Rightarrow \tau$ for a procedure and create a smaller constraint set $\mathcal{C}'$ such that any constraint on τ implied by $\mathcal{C}$ is also implied by $\mathcal{C}'$.

Let $\mathcal{C}$ denote the constraint set generated by abstract interpretation of the procedure being analyzed, and let $\overline{\alpha}$ be the set of free type variables in $\mathcal{C}$. We could already use $\forall \overline{\alpha}.\, \mathcal{C} \Rightarrow \tau$ as the constraint set in the procedure's type scheme, since the input and output types used in a valid invocation of `f` are tautologically those that satisfy $\mathcal{C}$. Yet as a practical matter we cannot use the constraint set directly, since this would result in constraint sets with many useless free variables and a high growth rate over nested procedures.

Instead, we seek to generate a *simplified constraint set* $\mathcal{C}'$ such that if c is an "interesting" constraint and $\mathcal{C} \vdash c$ then $\mathcal{C}' \vdash c$ as well. But what makes a constraint interesting?

Definition 5.1. For a type variable τ, a constraint is called *interesting* if it has one of the following forms:

- A capability constraint of the form VAR $\tau.u$.
- A recursive subtype constraint of the form $\tau.u \sqsubseteq \tau.v$.
- A subtype constraint of the form $\tau.u \sqsubseteq \overline{\kappa}$ or $\overline{\kappa} \sqsubseteq \tau.u$ where $\overline{\kappa}$ is a type constant.

We will call a constraint set $\mathcal{C}'$ a *simplification* of $\mathcal{C}$ if $\mathcal{C}' \vdash c$ for every interesting constraint c such that $\mathcal{C} \vdash c$. Since both $\mathcal{C}$ and $\mathcal{C}'$ entail the same set of constraints on τ, it is valid to replace $\mathcal{C}$ with $\mathcal{C}'$ in any valid type scheme for τ.

Simplification heuristics for set-constraint systems were studied in Fhndrich and Aiken [10]; our simplification algorithm encompasses all of these heuristics.

5.2 Unconstrained pushdown systems

The constraint-simplification algorithm works on a constraint set $\mathcal{C}$ by building a pushdown system $\mathcal{P}_{\mathcal{C}}$ whose transition sequences represent valid derivations of subtyping judgements. We briefly review pushdown systems and some necessary generalizations here.

Definition 5.2. An *unconstrained pushdown system* is a triple $\mathcal{P} = (\mathcal{V}, \Sigma, \Delta)$ where $\mathcal{V}$ is the set of *control locations*, Σ is the set of *stack symbols*, and $\Delta \subseteq (\mathcal{V} \times \Sigma^*)^2$ is a (possibly infinite) set of *rewrite rules*. We will denote a rewrite rule by $\langle X; u\rangle \hookrightarrow \langle Y; v\rangle$ where $X, Y \in \mathcal{V}$ and $u, v \in \Sigma^*$. We define the set of *configurations* to be $\mathcal{V} \times \Sigma^*$. In a configuration (p, w), p is called the *control state* and w the *stack state*.

Note that we require neither the set of stack symbols nor the set of rewrite rules to be finite. This freedom is required to model the derivation S-POINTER of figure 3, which corresponds to an infinite set of rewrite rules.

Definition 5.3. An unconstrained pushdown system $\mathcal{P}$ determines a *rewrite relation* $\longrightarrow$ on the set of configurations: $(X, w) \longrightarrow (Y, w')$ if there is a suffix s and a rule $\langle X; u\rangle \hookrightarrow \langle Y; v\rangle$ such that $w = us$ and $w' = vs$. The transitive closure of $\longrightarrow$ is denoted $\xrightarrow{*}$.

With this definition, we can state the primary theorem behind our simplification algorithm.

Theorem 5.1. Let $\mathcal{C}$ be a constraint set and $\mathcal{V}$ a set of base type variables. Define a subset $S_{\mathcal{C}}$ of $(\mathcal{V} \cup \Sigma)^* \times (\mathcal{V} \cup \Sigma)^*$ by $(Xu, Yv) \in S_{\mathcal{C}}$ if and only if $\mathcal{C} \vdash X.u \sqsubseteq Y.v$. Then $S_{\mathcal{C}}$ is a regular set, and an automaton Q to recognize $S_{\mathcal{C}}$ can be constructed in $O(|\mathcal{C}|^3)$ time.

Proof. The basic idea is to treat each $X.u \sqsubseteq Y.v \in \mathcal{C}$ as a rewrite rule $\langle X; u\rangle \hookrightarrow \langle Y; v\rangle$ in the pushdown system $\mathcal{P}$. In addition, we add a control states ${}^{\#}\textsc{Start}, {}^{\#}\textsc{End}$ with transitions $\langle {}^{\#}\textsc{Start}; X\rangle \hookrightarrow \langle X; \varepsilon\rangle$ and $\langle X; \varepsilon\rangle \hookrightarrow \langle {}^{\#}\textsc{End}; X\rangle$ for each $X \in \mathcal{V}$. For the moment, assume that (1) all labels are covariant, and (2) the rule S-POINTER is ignored. By construction, $({}^{\#}\textsc{Start}, Xu) \xrightarrow{*} ({}^{\#}\textsc{End}, Yv)$ in $\mathcal{P}$ if and only if $\mathcal{C} \vdash X.u \sqsubseteq Y.v$. A theorem of Büchi [24] ensures that for any two control states A and B in a standard (not unconstrained) pushdown system, the set of all pairs (u, v) with $(A, u) \xrightarrow{*} (B, v)$ is a regular language; Caucal [6] gives a saturation algorithm that constructs an automaton to recognize this language.

In the full proof, we add two novelties: first, we support contravariant stack symbols by encoding variance data into the control states and rewrite rules. The second novelty involves the rule S-POINTER; this rule is problematic since the natural encoding would result in infintely many rewrite rules. We extend Caucal's construction to lazily instantiate all necessary applications of S-POINTER during saturation. The details may be found in appendix D. □

Since $\mathcal{C}$ will usually entail an infinite number of constraints, this theorem is particularly useful: it tells us that the full set of constraints entailed by $\mathcal{C}$ has a finite encoding by an automaton Q. Further manipulations on the constraint closure can be carried out on Q, such as efficient minimization. By restricting the transitions to and from ${}^{\#}\textsc{Start}$ and ${}^{\#}\textsc{End}$, the same algorithm is used to eliminate type variables, producing the desired constraint simplifications.

6. Evaluation

6.1 Implementation

Retypd is implemented as a module within CodeSurfer for Binaries. By leveraging the multi-platform disassembly capabilities of CodeSurfer, it can operate on x86, x86-64, and ARM code. We performed the evaluation using minimal analysis settings, disabling value-set analysis (VSA) and nearly all points-to analysis. Enabling additional CodeSurfer phases such as VSA can greatly improve the reconstructed IR, at the expense of increased analysis time.

Existing type inference algorithms such as TIE [15] and SecondWrite [8] require some modified form of VSA to resolve points-to data. Our approach shows that high-quality types can be recovered even from relatively weak points-to information, allowing type inference to proceed even when computing points-to data is too unreliable or expensive.

6.2 Evaluation Setup

Our benchmark suite consists of 136 32-bit x86 binaries for both Linux and Windows, compiled with a variety of `gcc` and Microsoft Visual C/C++ versions. The benchmark suite includes a mix of executables, static libraries, and DLLs. The suite includes the same coreutils and SPEC2006 benchmarks used to evaluate REWARDS, TIE, and SecondWrite [8, 15, 17]; additional benchmarks came from a standard suite of real-world programs used for precision and performance testing of CodeSurfer. All binaries were built with optimizations enabled and debug information disabled.

Ground truth is provided by separate copies of the binaries that have been built with debug information (DWARF on Linux, PDB on Windows). We used IdaPro to read the debug information, which allowed us to use the same scripts for collecting ground truth from both DWARF and PDB data.

All benchmarks were evaluated on a 2.6 GHz Intel Xeon E5-2670 CPU, running on a single logical core.

6.3 Sources of imprecision

Although Retypd is built around a sound core of constraint simplification and solving, there are several ways that imprecision can occur. As described in § 2.5, disassembly failures can lead to unsound constraint generation. Second, the heuristics for converting from sketches to C types are lossy by necessity. Finally, we treat the source types as ground truth, leading to "failures" whenever Retypd recovers an accurate type that does not match the original program — a common situation with type-unsafe source code.

6.4 `const` correctness

As a side-effect of separately modeling .load and .store capabilities, Retypd is easily able to recover information about how pointer parameters are used for input and/or output. We take this into account when converting sketches to C types;

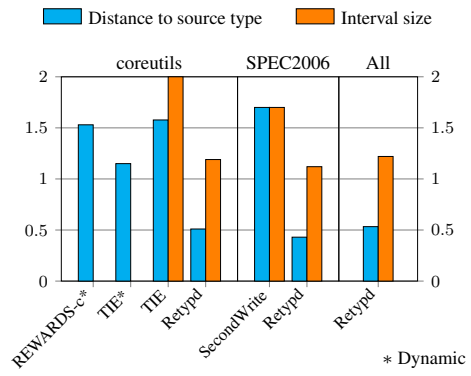


Figure 6. Distance to true type and size of the interval between inferred upper and lower bounds. Smaller distances represent more accurate types; smaller interval sizes represent increased confidence.

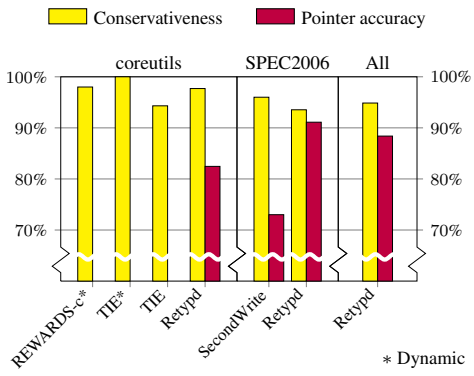


Figure 7. Conservativeness and pointer accuracy metric. Perfect type reconstruction would be 100% conservative and match on 100% of pointer levels. Note that the y axis begins at 70%.

Benchmark	Description
distributor	UltraVNC repeater
freeglut	The `freeglut.dll` library
freeglut-Fractals	A freeglut demo program
freeglut-Lorenz	A freeglut demo program
freeglut-Shapes	A freeglut demo program
glut	The `glut32.dll` library
putty	SSH utilities
sphinx2	Speech recognition
vpx-e	VPx encoders
vpx-d	VPx decoders
libbz2	BZIP library, as a DLL
libidn	Domain name translator
miranda	IRC client
ogg	Multimedia library
pngtest	A test of libpng
python21	Python 2.1
quake3	Quake 3
sharaza	Peer-to-peer file sharing
TinyCad	Computed-aided design
Tutorial00	Direct3D tutorial
XMail	Email server
yasm	Modular assembler
zlib	Compression library

Table 2. Additional benchmarks, not including coreutils or SPEC2006.

if a function's sketch includes .in$_L$.load but not .in$_L$.store then we annotate the parameter at L with `const`, as in figures 5 and 2. Retypd appears to be the first machine code type inference system to directly infer `const` annotations.

On our benchmark suite, we found that 98% of parameter `const` annotations in the original source code were recovered by Retypd. Furthermore, Retypd inferred `const` annotations on many other parameters; unfortunately, since most C and C++ code does not use `const` in every possible situation, we do not have a straightforward way to detect how many of Retypd's additional `const` annotations are correct.

Manual inspection of the missed `const` annotations shows that most instances are due to imprecision when analyzing one or two common library functions. This imprecision then propagates outward to callers, leading to decreased `const` correctness overall. Still, we believe that a 98% recovery rate shows that Retypd offers a very promising approach to `const` inference.

6.5 Comparisons to other tools

We chose to gather results over several metrics that have been used to evaluate SecondWrite, TIE, and REWARDS. These metrics were defined in Lee et al. [15] and are briefly reviewed here.

TIE infers upper and lower bounds on each type variable, with the bounds belonging to a lattice of C-like types. The lattice is naturally stratified into levels, with the distance between two comparable types roughly being the difference between their levels in the lattice, with a maximum distance of 4. A recursive formula for computing distances between pointer and structural types is also used. TIE also determines a policy such that either the upper- or lower-bound on a type variable is chosen as the final type to display.

TIE considers three metrics based on this lattice: the conservativeness rate, the interval size, and the distance. A type interval is *conservative* if the interval bounds overapproximate the declared type of a variable. The *interval size* is the lattice distance from the upper to the lower bound on a type variable. The *distance* measures the lattice distance from the final displayed type to the true type. REWARDS and SecondWrite both use unification-based algorithms, and have been evaluated using the same TIE metrics. The evaluation of REWARDS using TIE metrics appears in Lee et al. [15].

Distance and interval size: Retypd shows substantial improvements over other approaches in the distance and interval size metrics, indicating that it generates more accurate types with less uncertainty. The mean distance to the true type was 0.54 for Retypd, compared to 1.15 for dynamic TIE, 1.53 for REWARDS, 1.58 for static TIE, and 1.70 for SecondWrite. The mean interval size shrunk to 1.2 with Retypd, compared to 1.7 for SecondWrite and 2.0 for TIE.

Multi-level pointer accuracy: ElWazeer et al. [8] also introduced a multi-level pointer accuracy rate, which attempts to quantify how many "levels" of pointers were correctly inferred. On SecondWrite's benchmark suite Retypd attained a mean multi-level pointer accuracy of 91%, compared with SecondWrite's reported 73%. Across all benchmarks, Retypd averages 88% pointer accuracy.

Conservativeness: The best type system would have a high conservativeness rate (few unsound decisions) coupled with a low interval size (tightly specified results) and low distance (inferred types are close to true types). In each of these metrics, Retypd performs about as well or better than existing approaches. Retypd's mean conservativeness rate is 95%, compared to 94% for TIE. But note that TIE was evaluated only on `coreutils`; on that cluster, Retypd's conservativeness was 98%. SecondWrite's overall conservativeness is 96%, measured on a subset of the SPEC2006 benchmarks; Retypd attained a slightly lower 94% on this subset.

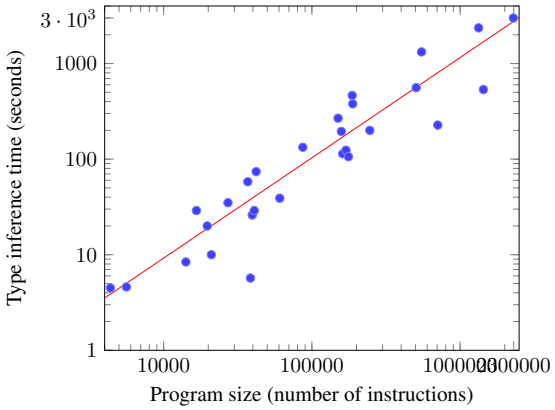


Figure 8. Type inference time on benchmarks. The red line is the best-fit exponential $t = 0.000587 \cdot N^{1.049}$, demonstrating slightly superlinear real-world scaling behavior. The coefficient of determination is $R^2 = 0.78$.

It is interesting to note that Retypd's conservativeness rate on `coreutils` is comparable to that of REWARDS, even though REWARDS' use of dynamic execution traces suggests it would be more conservative than a static analysis by virtue of only generating feasible type constraints.

6.6 Performance

Although the core simplification algorithm of Retypd has cubic worst-case complexity, it only needs to be applied on a per-procedure basis. This means that worst-case scaling behavior depends on maximum procedure size, not on the whole-program size. Figure 8 shows that on our benchmark suite Retypd gives nearly linear performance, with execution time scaling like $N^{1.049}$. The largest executable tested was `gcc`, at about 2.5 million instructions; on this benchmark, Retypd took just under 50 minutes to perform type inference.

The constraint-simplification workload of Retypd would be straightforward to parallelize over the directed acyclic graph of strongly-connected components in the callgraph, further reducing the scaling constant.

7. Related work

Machine code type recovery: Hex-Ray's reverse engineering tool IdaPro [13] is an early example of type reconstruction via static analysis. The exact algorithm is proprietary, but it appears that IdaPro propagates types through unification from library functions of known signature, halting the propagation when a type conflict appears. IdaPro's reconstructed IR is relatively sparse, so the type propagation fails to produce useful information in many common cases, falling back to the default `int` type. On the other hand, the analysis is very fast.

SecondWrite [8] is an interesting approach to static IR reconstruction with a particular eye to scalability. The authors combine a best-effort VSA variant for points-to analysis with a unification-based type inference engine. Accurate types in SecondWrite depend on high-quality points-to data; the authors note that this can cause type accuracy to suffer on larger programs. In contrast, Retypd is less dependent on points-to data for type recovery and makes use of subtyping rather than unification for increased precision.

TIE [15] is a static type-reconstruction tool used as part of Carnegie Mellon University's binary analysis platform (BAP). TIE was the first machine code type inference system to track subtype constraints and explicitly maintain upper- and lower-bounds on each type variable. As an abstraction of the C type system, TIE's type lattice is relatively simple; missing features such as recursive types were later identified by the authors as an important target for future research [27].

HOWARD [29] and REWARDS [17] both take a dynamic approach, generating type constraints from execution traces. Through a comparison with HOWARD, the authors of TIE showed that static type analysis can produce higher-precision types than dynamic type analysis, though a small penalty must be paid in conservativeness of constraint set generation. TIE also showed that type systems designed for static analysis can be easily modified to work on dynamic traces; we expect the same is true for Retypd, though we have not yet performed these experiments.

Most previous work on machine code type recovery, including TIE and SecondWrite, either disallow recursive types or only support recursive types by combining type inference results with a points-to oracle. For example, to infer that x has a the type `struct S { struct S *, ...}*` in a unification-based approach like SecondWrite, we must first have resolved that x points to some memory region M, that M admits a 4-byte abstract location α at offset 0, and that the type of α should be unified with the type of x. If pointer analysis has failed to compute an explicit memory region pointed to by x, it will not be possible to correctly determine the type of x. The complex interplay between type inference, points-to analysis, and abstract location delineation leads to a relatively fragile method for inferring recursive types. In contrast, our type system can infer recursive types even when points-to facts are completely absent.

Robbins et al. [25] developed an SMT solver equipped with a theory of rational trees and applied it to type reconstruction. Although this allows for recursive types, the lack of subtyping and the performance of the SMT solver make it difficult to scale this approach to real-world binaries. Except for test cases on the order of 500 instructions, precision of the recovered types was not assessed.

Related type systems: The type system used by Retypd is related to the recursively constrained types (rc types) of Eifrig, Smith, and Trifonov [7]. Retypd generalizes the rc type system by building up all types using flexible records; even the function-type constructor $\rightarrow$, taken as fundamental in the rc type system, is decomposed into a record with in and out fields. This allows Retypd to operate without the knowledge of a fixed signature from which type construc-

tors are drawn, which is essential for analysis of stripped machine code.

The use of CFL-reachability to perform polymorphic subtyping first appeared in Rehof and Fähndrich [23], extending previous work relating simpler type systems to graph reachability [2, 20]. Retypd continues by adding type-safe handling of pointers and a simplification algorithm that allows us to compactly represent the type scheme for each function.

CFL-reachability has also been used to extend the type system of Java [12] and C++ [9] with support for additional type qualifiers. Our reconstructed `const` annotations can be seen as an instance of this idea, although our qualifier inference is not separated from type inference as a whole.

To the best of our knowledge, no authors have previously investigated the application of polymorphic type systems with subtyping to machine code.

8. Future work

One interesting avenue for future research could come from the application of dependent and higher-rank type systems to machine-code type inference, although we rapidly approach the frontier where type inference is undecidable. A natural example of dependent types appearing in machine code is `malloc`, which could be typed as `malloc` : $(n$: `size_t`$) \to \top_n$ where $\top_n$ denotes the common supertype of all n-byte types. The key feature is that the *value* of a parameter determines the *type* of the result.

Higher-rank types are needed to properly model functions that accept pointers to polymorphic functions as parameters. Such functions are not entirely uncommon; for example, any function that is parameterized by a custom polymorphic allocator will have rank ≥ 2.

9. Conclusion

By examining a diverse corpus of optimized binaries, we have identified a number of common idioms that are stumbling blocks for machine code type inference. For each of these idioms, we identified a type system feature that could enable the difficult code to be properly typed. We gathered these features into a type system and implemented the inference algorithm in the tool Retypd. Despite weakening the requirement for points-to data compared to similar tools, Retypd is able to accurately and conservatively type a wide variety of real-world binaries. Our results with Retypd suggest that inference of polymorphic types from machine code is a promising direction for further study. We hope Retypd demonstrates the utility of high-level type systems for reverse engineering and machine-code static analysis.

References

[1] ISO/IEC TR 19768:2007: Technical report on C++ library extensions, 2007.

[2] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis*, pages 78–100. Springer, 1994.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[4] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – a platform for analyzing x86 executables. In *Compiler Construction*, pages 250–254. Springer, 2005.

[5] A. Carayol and M. Hague. Saturation algorithms for model-checking pushdown systems. In *Proceedings of the 14th International Conference on Automata and Formal Languages*, volume 151 of *AFL 2014*, pages 1–24, 2014.

[6] D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61–86, 1992.

[7] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 30, pages 169–184. ACM, 1995.

[8] 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.

[9] J. S. Foster, R. Johnson, J. Kodumal, and A. Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):1035–1087, 2006.

[10] M. Fhndrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints*, 1996.

[11] D. Gopan, E. Driscoll, D. Nguyen, D. Naydich, A. Loginov, and D. Melski. Data-delineation in software binaries and its application to buffer-overrun discovery. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 145–155, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818775.

[12] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 42, pages 321–336. ACM, 2007.

[13] Hex-Rays. Hex-Rays IdaPro. http://www.hex-rays.com/products/ida/, 2015. Accessed 2015-08-15.

[14] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(01):113–125, 1995.

[15] 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.

[16] J. Lim and T. Reps. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(1):4, 2013.

[17] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS '10)*, 2010.

[18] S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 07)*, volume 42, pages 277–288. ACM, 2007.

[19] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Programming Languages and Systems*, pages 5–20. Springer, 2005.

[20] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(4):576–599, 1995.

[21] J. Palsberg, M. Wand, and P. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9(1): 49–67, 1997.

[22] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10. MIT Press, 2005.

[23] J. Rehof and M. Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 54–66, New York, NY, USA, 2001. ACM. doi: 10.1145/360204.360208. URL http://doi.acm.org/10.1145/360204.360208.

[24] J. Richard Büchi. Regular canonical systems. *Archive for Mathematical Logic*, 6(3):91–111, 1964.

[25] 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.

[26] M. Robertson. *A Brief History of InvSqrt*. PhD thesis, University of New Brunswick, 2012.

[27] 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.

[28] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Software EngineeringESEC/FSE99*, pages 180–198. Springer, 1999.

[29] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS '11)*, 2011.

[30] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 32–41. ACM, 1996.

[31] Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 203–216. ACM, 2002.

A. Constraint generation

Type constraint generation is performed by a parameterized abstract interpretation $\textsc{Type}_\mathsf{A}$; the parameter A is itself an abstract interpreter that is used to transmit additional analysis information such as reaching definitions, propagated constants, and value-sets (when available).

Let $\mathcal{V}$ denote the set of type variables and $\mathcal{C}$ the set of type constraints. Then the primitive TSL value- and map-types for $\textsc{Type}_\mathsf{A}$ are given by

$$\mathsf{BASETYPE}_{\textsc{Type}_\mathsf{A}} = \mathsf{BASETYPE}_\mathsf{A} \times 2^{\mathcal{V}} \times 2^{\mathcal{C}}$$
$$\mathsf{MAP}[\alpha,\beta]_{\textsc{Type}_\mathsf{A}} = \mathsf{MAP}[\alpha,\beta]_\mathsf{A} \times 2^{\mathcal{C}}$$

Since type constraint generation is a syntactic, flow-insensitive process, we can regain flow sensitivity by pairing with an abstract semantics that carries a summary of flow-sensitive information. Parameterizing the type abstract interpretation by A allows us to factor out the particular way in which program variables should be abstracted to types (e.g. SSA form, reaching definitions, and so on).

A.1 Register loads and stores

The basic reinterpretations proceed by pairing with the abstract interpreter A. For example,

```
regUpdate(s, reg, v) =
  let (v', t, c) = v
      (s', m)    = s
      s'' = regUpdate(s', reg, v')
      (u, c') = A(reg, s'')
  in
      ( s'', m ∪ c ∪ { t ⊑ u } )
```

where $\mathtt{A}(\mathtt{reg}, \mathtt{s})$ produces a type variable from the register `reg` and the A-abstracted register map $\mathtt{s}''$.

Register loads are handled similarly:

```
regAccess(reg, s) =
  let (s', c) = s
      (t, c') = A(reg, s')
  in
      ( regAccess(reg, s'), t, c ∪ c' )
```

Example A.1. Suppose that A represents the concrete semantics for x86 and $\mathtt{A}(\mathtt{reg}, \cdot)$ yields a type variable $(\mathsf{reg}, \{\})$ and no additional constraints. Then the x86 expression `mov ebx, eax` is represented by the TSL expression `regUpdate(S, EBX(), regAccess(EAX(), S))`, where S is the initial state $(S_{\text{conc}}, \mathcal{C})$. After abstract interpretation, $\mathcal{C}$ will become $\mathcal{C} \cup \{\mathsf{eax} \sqsubseteq \mathsf{ebx}\}$.

By changing the parametric interpreter A, the generated type constraints may be made more precise.

Example A.2. We continue with the example of `mov ebx, eax` above. Suppose that A represents an abstract semantics that is aware of register reaching definitions, and define $\mathtt{A}(\mathtt{reg}, \mathtt{s})$ by

```
A(reg, s) =
 case reaching-defs(reg, s) of
   { p } → (reg_p, {})
   defs →
        let t = fresh
            c = { reg_p ⊑ t | p ∈ defs }
        in (t, c)
```

where `reaching-defs` yields the set of definitions of `reg` that are visible from state `s`. Then TYPE_A at program point q will update the constraint set $\mathcal{C}$ to

$$\mathcal{C} \cup \{\mathsf{eax}_p \sqsubseteq \mathsf{ebx}_q\}$$

if p is the lone reaching definition of `EAX`. If there are multiple reaching definitions P, then the constraint set will become

$$\mathcal{C} \cup \{\mathsf{eax}_p \sqsubseteq t \mid p \in P\} \cup \{t \sqsubseteq \mathsf{ebx}_q\}$$

A.2 Addition and subtraction

It is useful to track translations of a value through additions or subtraction of a constant. To that end, we overload the `add(x,y)` and `sub(x,y)` operations in the cases where `x` or `y` have statically-determined constant values. For example, if `INT32(n)` is a concrete numeric value then

```
add(v, INT32(n)) =
  let (v', t, c) = v in
    ( add(v', INT32(n)), t.+n, c )
```

In the case where neither operand is a statically-determined constant, we generate a fresh type variable representing the result and a 3-place constraint on the type variables:

```
add(x, y) =
  let (x', t1, c1) = x
      (y', t2, c2) = y
      t = fresh
  in
    ( add(x', y'),
      t,
      c1 ∪ c2 ∪ { ADD(t1, t2, t) } )
```

Similar interpretations are used for `sub(x,y)`.

A.3 Memory loads and stores

Memory accesses are treated similarly to register accesses, except for the use of dereference accesses and the handling of points-to sets. For any abstract `A`-value `a` and `A`-state `s`, let `A(a,s)` denote a set of type variables representing the address `A` in the context `s`. Furthermore, define $\texttt{PtsTo}_\text{A}(a, s)$ to be a set of type variables representing the values pointed to by `a` in the context `s`.

The semantics of the `N`-bit load and store functions $\texttt{memAccess}_\text{N}$ and $\texttt{memUpdate}_\text{N}$ are given by

```
memAccess_N(s, a) =
  let (s0, cs)   = s
      (a0, t, ct) = a
      cpt = { x ⊑ t.load.σN@0
            | x ∈ PtsTo(a0,s0) }
  in
      ( memAccess_N(s0, a0),
        t.load.σN@0,
        cs ∪ ct ∪ cpt )

memUpdate_N(s, a, v) =
  let (s0, cs)   = s
      (a0, t, ct) = a
      (v0, v, cv) = v
      cpt = { t.store.σN@0 ⊑ x
            | x ∈ PtsTo(a0,s0) }
  in
      ( memUpdate_N(s0, a0, v0),
        cs ∪ ct ∪ cv ∪ cpt
          ∪ { v ⊑ t.store.σN@0 } )
```

We have found acceptable results by using a bare minimum points-to analysis that only tracks constant pointers to the local activation record or the data section. The use of the .load / .store accessors allows us to track multi-level pointer information without the need for explicit points-to data. The minimal approach tracks just enough points-to information to resolve references to local and global variables.

A.4 Procedure invocation

Earlier analysis phases are responsible for delineating procedures and gathering data about each procedure's formal-in and formal-out variables, including information about how parameters are stored on the stack or in registers. This data is transformed into a collection of *locators* associated to each function. Each locator is bound to a type variable representing the formal; the locator is responsible for finding an appropriate set of type variables representing the actual at a callsite, or the corresponding local within the procedure itself.

Example A.3. Consider this simple program that invokes a 32-bit identity function.

```
 p:    push ebx  ; writes to local ext4
 q:    call id
        ...
id:    ; begin procedure id()
 r:    mov eax, [esp+arg0]
       ret
```

The procedure `id` will have two locators:

- A locator L_i for the single parameter, bound to a type variable id_i.
- A locator L_o for the single return value, bound to a type variable id_o.

At the procedure call site, the locator L_i will return the type variable $\mathsf{ext4}_p$ representing the stack location `ext4` tagged by its reaching definition. Likewise, L_o will return the type variable eax_q to indicate that the actual-out is held in the version of `eax` that is defined at point q. The locator results

are combined with the locator's type variables, resulting in the constraint set

$$\{\mathsf{ext4}_p \sqsubseteq \mathsf{id}_i, \quad \mathsf{id}_o \sqsubseteq \mathsf{eax}_q\}$$

Within procedure `id`, the locator L_i returns the type variable $\mathsf{arg0}_{\mathtt{id}}$ and L_o returns eax_r, resulting in the constraint set

$$\{\mathsf{id}_i \sqsubseteq \mathsf{arg0}_{\mathtt{id}}, \quad \mathsf{eax}_r \sqsubseteq \mathsf{id}_o\}$$

A procedure may also be associated with a set of type constraints between the locator type variables, called the *procedure summary*; these type constraints may be inserted at function calls to model the known behavior of a function. For example, invocation of `fopen` will result in the constraints

$$\{\mathsf{fopen}_{i_0} \sqsubseteq \underline{\mathsf{char}*}, \ \mathsf{fopen}_{i_1} \sqsubseteq \underline{\mathsf{char}*}, \ \underline{\mathsf{FILE}*} \sqsubseteq \mathsf{fopen}_o\}$$

To support polymorphic function invocation, we instantiate fresh versions of the locator type variables that are tagged with the current callsite; this prevents type variables from multiple invocations of the same procedure from being linked.

Example A.4 (cont'd)**.** When using callsite tagging, the callsite constraints generated by the locators would be

$$\{\mathsf{ext4}_p \sqsubseteq \mathsf{id}_i^q, \quad \mathsf{id}_o^q \sqsubseteq \mathsf{eax}_q\}$$

The callsite tagging must also be applied to any procedure summary. For example, a call to `malloc` will result in the constraints

$$\{\mathsf{malloc}_i^p \sqsubseteq \underline{\mathsf{size\_t}}, \quad \underline{\mathsf{void}*} \sqsubseteq \mathsf{malloc}_o^p\}$$

If `malloc` is used twice within a single procedure, we see an effect like `let`-polymorphism: each use will be typed independently.

A.5 Other operations

A.5.1 Floating-point

Floating point types are produced by calls to known library functions and through an abstract interpretation of reads to and writes from the floating point register bank. We do not track register-to-register moves between floating point registers, though it would be straightforward to add this ability. In theory, this causes us to lose precision when attempting to distinguish between typedefs of floating point values; in practice, such typedefs appear to be extremely rare.

A.5.2 Bit manipulation

We assume that the operands and results of most bit-manipulation operations are integral, with some special exceptions:

- Common idioms like `xor reg,reg` and `or reg,-1` are used to initialize registers to certain constants. On x86 these instructions can be encoded with 8-bit immediates, saving space relative to the equivalent versions `mov reg,0` and `mov reg,-1`. We do not assume that the results of these operations are of integral type.
- We discard any constraints generated while computing a value that is only used to update a flag status. In particular, on x86 the operation `test reg1,reg2` is implemented like a bitwise-AND that discards its result, only retaining the effect on the flags.
- For specific operations such as $y := x$ AND `0xfffffffc` and $y := x$ OR 1, we act as if they were equivalent to $y := x$. This is because these specific operations are often used for *bit-stealing*; for example, requiring pointers to be aligned on 4-byte boundaries frees the lower two bits of a pointer for other purposes such as marking for garbage collection.

A.6 Additive constraints

The special constraints ADD and SUB are used to conditionally propagate information about which type variables represent pointers and which represent integers when the variables are related through addition or subtraction. The deduction rules for additive constraints are summarized in figure 9. We obtained good results by inspecting the unification graph used for computing $\mathcal{L}(S_i)$; the graph can be used to quickly determine whether a variable has pointer- or integer-like capabilities. In practice, the constraint set also should be updated with new subtype constraints as the additive constraints are applied, and a fully applied constraint can be dropped from $\mathcal{C}$. We omit these details for simplicity.

B. Normal forms of proofs

A finite constraint set with recursive constraints can have an infinite entailment closure. In order to manipulate entailment closures efficiently, we need finite (and small) representations of infinite constraint sets. The first step towards achieving a finite representation of entailment is to find a normal form for every derivation. In appendix D, finite models of entailment closure will be constructed that manipulate representations of these normal forms.

Lemma B.1. For any statement P provable from $\mathcal{C}$, there exists a derivation of $\mathcal{C} \vdash P$ that does not use rules S-REFL or T-INHERIT.

Proof. The redundancy of S-REFL is immediate. As for T-INHERIT, any use of that rule in a proof can be replaced by S-FIELD$_\oplus$ followed by T-LEFT, or S-FIELD$_\ominus$ followed by T-RIGHT. □

In the following, we make the simplifying assumption that $\mathcal{C}$ is closed under T-LEFT, T-RIGHT, and T-PREFIX. Concretely, we are requiring that

1. if $\mathcal{C}$ contains a subtype constraint $\alpha \sqsubseteq \beta$ as an axiom, it also contains the axioms VAR α and VAR β.
2. if $\mathcal{C}$ contains a term declaration VAR α, it also contains term declarations for all prefixes of α.

	ADD						SUB						
X	i	I	p	P	I	i	i	I	P	P	p	p	p
Y	i	I	I	i	p	P	I	i	i	p	P	i	I
Z	I	i	P	p	P	p	I	i	p	I	i	P	p

Figure 9. Inference rules for $\text{ADD}(X, Y; Z)$ and $\text{SUB}(X, Y; Z)$. Lower case letters denote known integer or pointer types. Upper case letters denote inferred types. For example, the first column says that if X and Y are integral types in an $\text{ADD}(X, Y; Z)$ constraint, then Z is integral as well.

Lemma B.2. If $\mathcal{C}$ is closed under T-LEFT, T-RIGHT, and T-PREFIX then any statement provable from $\mathcal{C}$ can be proven without use of T-PREFIX.

Proof. By the previous lemma, we may assume that S-REFL and T-INHERIT are not used in the proof. We will prove the lemma by transforming all subproofs that end in a use of T-PREFIX to remove that use. To that end, assume we have a proof ending with the derivation of VAR α from VAR $\alpha.\ell$ using T-PREFIX. We then enumerate the ways that VAR $\alpha.\ell$ may have been proven.

- If VAR $\alpha.\ell \in \mathcal{C}$ then VAR $\alpha \in \mathcal{C}$ already, so the entire proof tree leading to VAR α may be replaced with an axiom from $\mathcal{C}$.
- The only other way a VAR $\alpha.\ell$ could be introduced is through T-LEFT or T-RIGHT. For simplicity, let us only consider the T-LEFT case; T-RIGHT is similar. At this point, we have a derivation tree that looks like

$$\dfrac{\dfrac{\alpha.\ell \sqsubseteq \varphi}{\text{VAR } \alpha.\ell}}{\text{VAR } \alpha}$$

  How was $\alpha.\ell \sqsubseteq \varphi$ introduced? If it were an axiom of $\mathcal{C}$ then our assumed closure property would imply already that VAR $\alpha \in \mathcal{C}$. The other cases are:
  - $\varphi = \beta.\ell$ and $\alpha.\ell \sqsubseteq \varphi$ was introduced through one of the S-FIELD axioms. In either case, α is the left- or right-hand side of one of the antecedents to S-FIELD, so the whole subderivation including the T-PREFIX axiom can be replaced with a single use of T-LEFT or T-RIGHT.
  - $\ell = .\mathsf{store}$ and $\varphi = \alpha.\mathsf{load}$, with $\alpha.\ell \sqsubseteq \varphi$ introduced via S-POINTER. In this case, VAR $\alpha.\ell$ is already an antecedent to S-POINTER, so we may elide the subproof up to T-PREFIX to inductively reduce the problem to a simpler proof tree.
  - Finally, we must have $\alpha.\ell \sqsubseteq \varphi$ due to S-TRANS. But in this case, the left antecedent is of the form $\alpha.\ell \sqsubseteq \beta$; so once again, we may elide a subproof to get a simpler proof tree.

  Each of these cases either removes an instance of T-PREFIX or results in a strictly smaller proof tree. It follows that iterated application of these simplification rules results in a proof of VAR α with no remaining instances of T-PREFIX.

□

Corollary B.1. If $\mathcal{C} \vdash$ VAR α, then either VAR $\alpha \in \mathcal{C}$, or $\mathcal{C} \vdash \alpha \sqsubseteq \beta, \beta \sqsubseteq \alpha$ for some β.

To simplify the statement of our normal-form theorem, recall that we defined the variadic rule S-TRANS' to remove the degrees of freedom from regrouping repeated application of S-TRANS:

$$\dfrac{\alpha_1 \sqsubseteq \alpha_2,\ \alpha_2 \sqsubseteq \alpha_3,\ \ldots\ \alpha_{n-1} \sqsubseteq \alpha_n}{\alpha_1 \sqsubseteq \alpha_n}\ \text{(S-TRANS')}$$

Theorem B.1 (Normal form of proof trees)**.** Let $\mathcal{C}$ be a constraint set that is closed under T-LEFT, T-RIGHT, and T-PREFIX. Then any statement provable from $\mathcal{C}$ has a derivation such that

- There are no uses of the rules T-PREFIX, T-INHERIT, or S-REFL.
- For every instance of S-TRANS' and every pair of adjacent antecedents, at least one is *not* the result of a S-FIELD application.

Proof. The previous lemmas already handled the first point. As to the second, suppose that we had two adjacent antecedents that were the result of S-FIELD$_\oplus$:

$$\dfrac{\cdots\ \dfrac{\dfrac{P}{\alpha \sqsubseteq \beta} \quad \dfrac{Q}{\text{VAR } \beta.\ell}}{\alpha.\ell \sqsubseteq \beta.\ell} \quad \dfrac{\dfrac{R}{\beta \sqsubseteq \gamma} \quad \dfrac{S}{\text{VAR } \gamma.\ell}}{\beta.\ell \sqsubseteq \gamma.\ell}\ \cdots}{T}$$

First, note that the neighboring applications of S-FIELD$_\oplus$ must both use the same field label ℓ, or else S-TRANS would not be applicable. But now observe that we may move a S-FIELD application upwards, combining the two S-FIELD applications into one:

$$\dfrac{\cdots\ \dfrac{\dfrac{\dfrac{P}{\alpha \sqsubseteq \beta} \quad \dfrac{R}{\beta \sqsubseteq \gamma}}{\alpha \sqsubseteq \gamma} \quad \dfrac{S}{\text{VAR } \gamma.\ell}}{\alpha.\ell \sqsubseteq \gamma.\ell}\ \cdots}{T}$$

For completeness, we also compute the simplifying transformation for adjacent uses of S-FIELD$_\ominus$: the derivation

$$\dfrac{\cdots\dfrac{\dfrac{P}{\beta\sqsubseteq\alpha}\quad\dfrac{Q}{\textsc{Var}\ \alpha.\ell}}{\alpha.\ell\sqsubseteq\beta.\ell}\quad\dfrac{\dfrac{R}{\gamma\sqsubseteq\beta}\quad\dfrac{S}{\textsc{Var}\ \beta.\ell}}{\beta.\ell\sqsubseteq\gamma.\ell}\cdots}{T}$$

can be simplified to

$$\dfrac{\cdots\dfrac{\dfrac{Q}{\textsc{Var}\ \alpha.\ell}\quad\dfrac{\dfrac{R}{\gamma\sqsubseteq\beta}\quad\dfrac{P}{\beta\sqsubseteq\alpha}}{\gamma\sqsubseteq\alpha}}{\alpha.\ell\sqsubseteq\gamma.\ell}\cdots}{T}$$

We also can eliminate the need for the subderivations Q and S except for the very first and very last antecedents of S-TRANS' by pulling the term out of the neighboring subtype relation; schematically, we can always replace

$$\dfrac{\cdots\dfrac{\dfrac{P}{\alpha\sqsubseteq\beta}\quad\dfrac{S}{\textsc{Var}\ \beta.\ell}}{\alpha.\ell\sqsubseteq\beta.\ell}\quad\dfrac{U}{\beta.\ell\sqsubseteq\gamma}\cdots}{T}$$

by

$$\dfrac{\cdots\dfrac{\dfrac{P}{\alpha\sqsubseteq\beta}\quad\dfrac{\dfrac{U}{\beta.\ell\sqsubseteq\gamma}}{\textsc{Var}\ \beta.\ell}}{\alpha.\ell\sqsubseteq\beta.\ell}\quad\dfrac{U}{\beta.\ell\sqsubseteq\gamma}\cdots}{T}$$

This transformation even works if there are several S-FIELD applications in a row, as in

$$\dfrac{\cdots\dfrac{\dfrac{\dfrac{P}{\alpha\sqsubseteq\beta}\quad\dfrac{Q}{\textsc{Var}\ \beta.\ell}}{\alpha.\ell\sqsubseteq\beta.\ell}\quad\dfrac{R}{\textsc{Var}\ \beta.\ell.\ell'}}{\alpha.\ell.\ell'\sqsubseteq\beta.\ell.\ell'}\quad\dfrac{U}{\beta.\ell.\ell'\sqsubseteq\gamma}\cdots}{T}$$

which can be simplified to

$$\dfrac{\cdots\dfrac{\dfrac{P}{\alpha\sqsubseteq\beta}\quad\dfrac{\dfrac{\dfrac{U}{\beta.\ell.\ell'\sqsubseteq\gamma}}{\textsc{Var}\ \beta.\ell.\ell'}}{\textsc{Var}\ \beta.\ell}}{\alpha.\ell\sqsubseteq\beta.\ell}\quad\dfrac{U}{\beta.\ell.\ell'\sqsubseteq\gamma}}{\alpha.\ell.\ell'\sqsubseteq\beta.\ell.\ell'}\quad\dfrac{U}{\beta.\ell.\ell'\sqsubseteq\gamma}\cdots}{T}$$

Taken together, this demonstrates that the VAR $\beta.\ell$ antecedents to S-FIELD are automatically satisfied if the consequent is eventually used in an S-TRANS application where $\beta.\ell$ is a "middle" variable. This remains true even if there are several S-FIELD applications in sequence. □

Since the S-FIELD VAR antecedents are automatically satisfiable, we can use a simplified schematic depiction of proof trees that omits the intermediate VAR subtrees. In this simplified depiction, the previous proof tree fragment would be written as

$$\dfrac{\cdots\dfrac{\dfrac{\dfrac{P}{\alpha\sqsubseteq\beta}}{\alpha.\ell\sqsubseteq\beta.\ell}}{\alpha.\ell.\ell'\sqsubseteq\beta.\ell.\ell'}\quad\dfrac{U}{\beta.\ell.\ell'\sqsubseteq\gamma}\cdots}{T}$$

Any such fragment can be automatically converted to a full proof by re-generating VAR subderivations from the results of the input derivations P and/or U.

B.1 Algebraic representation

Finally, consider the form of the simplified proof tree that elides the VAR subderivations. Each leaf of the tree is a subtype constraint $c_i \in \mathcal{C}$, and there is a unique way to fill in necessary VAR antecedents and S-FIELD applications to glue the constraints $\{\rfloor_i\}$ into a proof tree in normal form. In other words, the normal form proof tree is completely determined by the sequence of leaf constraints $\{c_i\}$ and the sequence of S-FIELD applications applied to the final tree. In effect, we have a term algebra with constants C_i representing the constraints $c_i \in \mathcal{C}$, an associative binary operator $\odot$ that combines two compatible constraints via S-TRANS, inserting appropriate S-FIELD applications, and a unary operator S_ℓ for each $\ell \in \Sigma$ that represents an application of S-FIELD. The proof tree manipulations provide additional relations on this term algebra, such as

$$S_\ell(R_1) \odot S_\ell(R_2) = \begin{cases} S\ell(R_1 \odot R_2) & \text{when } \langle\ell\rangle = \oplus \\ S\ell(R_2 \odot R_1) & \text{when } \langle\ell\rangle = \ominus \end{cases}$$

The normalization rules described in this section demonstrate that every proof of a subtype constraint entailed by $\mathcal{C}$ can be represented by a term of the form

$$S_{\ell_1}\left(S_{\ell_2}\left(\cdots S_{\ell_n}(R_1 \odot \cdots \odot R_k)\cdots\right)\right)$$

C. The StackOp weight domain

In this section, we develop a weight domain $\mathsf{StackOp}_\Sigma$ that will be useful for modeling constraint sets by pushdown systems. This weight domain is generated by symbolic constants representing actions on a stack.

Definition C.1. The weight domain $\mathsf{StackOp}_\Sigma$ is the idempotent $*$-semiring generated by the symbols $\mathsf{pop}\ x$, $\mathsf{push}\ x$ for all $x \in \Sigma$, subject only to the relation

$$\mathsf{push}\ x \otimes \mathsf{pop}\ y = \delta(x,y) = \begin{cases} \underline{1} & \text{if } x = y \\ \underline{0} & \text{if } x \neq y \end{cases}$$

To simplify the presentation, when $u = u_1 \cdots u_n$ we will sometimes write

$$\mathsf{pop}\ u = \mathsf{pop}\ u_1 \otimes \cdots \otimes \mathsf{pop}\ u_n$$

and

$$\mathsf{push}\ u = \mathsf{push}\ u_n \otimes \cdots \otimes \mathsf{push}\ u_1$$

Definition C.2. A monomial in $\mathsf{StackOp}_\Sigma$ is called *reduced* if its length cannot be shortened through applications of the $\mathsf{push}\ x \otimes \mathsf{pop}\ y = \delta(x, y)$ rule. Every reduced monomial has the form

$$\mathsf{pop}\ u_1 \cdots \mathsf{pop}\ u_n \otimes \mathsf{push}\ v_m \cdots \mathsf{push}\ v_1$$

for some $u_i, v_j \in \Sigma$.

Elements of $\mathsf{StackOp}_\Sigma$ can be understood to denote possibly-failing functions operating on a stack of symbols from Σ:

$$\begin{aligned}
[\![\underline{0}]\!] &= \lambda s\ .\ \mathsf{fail} \\
[\![\underline{1}]\!] &= \lambda s\ .\ s \\
[\![\mathsf{push}\ x]\!] &= \lambda s\ .\ \mathsf{cons}(x, s) \\
[\![\mathsf{pop}\ x]\!] &= \lambda s\ .\ \mathsf{case}\ s\ \mathsf{of} \\
&\qquad \mathsf{nil} \to \mathsf{fail} \\
&\qquad \mathsf{cons}(x', s') \to \mathsf{if}\ x = x'\ \mathsf{then}\ s' \\
&\qquad\qquad\qquad\qquad \mathsf{else\ fail}
\end{aligned}$$

Under this interpretation, the semiring operations are interpreted as

$$\begin{aligned}
[\![X \oplus Y]\!] &= \text{nondeterministic choice of } [\![X]\!] \text{ or } [\![Y]\!] \\
[\![X \otimes Y]\!] &= [\![Y]\!] \circ [\![X]\!] \\
[\![X^*]\!] &= \text{nondeterministic iteration of } [\![X]\!]
\end{aligned}$$

Note that the action of a pushdown rule $R = \langle P; u\rangle \hookrightarrow \langle Q; v\rangle$ on a stack configuration $c = (X, w)$ is given by the application

$$[\![\mathsf{pop}\ P \otimes \mathsf{pop}\ u \otimes \mathsf{push}\ v \otimes \mathsf{push}\ Q]\!]\,(\mathsf{cons}(X, w))$$

The result will either be $\mathsf{cons}(X', w')$ where (X', w') is the configuration obtained by applying rule R to configuration c, or fail if the rule cannot be applied to c.

Lemma C.1. There is a one-to-one correspondence between elements of $\mathsf{StackOp}_\Sigma$ and regular sets of pushdown system rules over Σ.

Definition C.3. The variance operator $\langle\cdot\rangle$ can be extended to $\mathsf{StackOp}_\Sigma$ by defining

$$\langle \mathsf{pop}\ x\rangle = \langle \mathsf{push}\ x\rangle = \langle x\rangle$$

D. Constructing the transducer for a constraint set

D.1 Building the pushdown system

Let $\mathcal{V}$ be a set of type variables and Σ a set of field labels, equipped with a variance operator $\langle\cdot\rangle$. Furthermore, suppose that we have fixed a set $\mathcal{C}$ of constraints over $\mathcal{V}$. Finally, suppose that we can partition $\mathcal{V}$ into a set of *interesting* and *uninteresting* variables:

$$\mathcal{V} = \mathcal{V}_i \amalg \mathcal{V}_u$$

Definition D.1. Suppose that $X, Y \in \mathcal{V}_i$ are interesting type variables, and there is a proof $\mathcal{C} \vdash X.u \sqsubseteq Y.v$. We will call the proof *elementary* if the normal form of its proof tree only involves uninteresting variables on internal leaves. We write

$$\mathcal{C} \vdash {}^{\mathcal{V}_i}_{\text{elem}} X.u \sqsubseteq Y.v$$

when $\mathcal{C}$ has an elementary proof of $X.u \sqsubseteq Y.v$.

Our goal is to construct a finite state transducer that recognizes the relation $\sqsubseteq^{\mathcal{V}_i}_{\text{elem}} \subseteq \mathcal{V}_i' \times \mathcal{V}_i'$ between derived type variables defined by

$$\sqsubseteq^{\mathcal{V}_i}_{\text{elem}} = \left\{ (X.u, Y.v) \mid \mathcal{C} \vdash {}^{\mathcal{V}_i}_{\text{elem}} X.u \sqsubseteq Y.v \right\}$$

We proceed by constructing an unconstrained pushdown system $\mathcal{P}_\mathcal{C}$ whose derivations model proofs in $\mathcal{C}$; a modification of Caucal's saturation algorithm [6] is then used to build the transducer representing $\mathsf{Deriv}_{\mathcal{P}_\mathcal{C}}$.

Definition D.2. Define the left- and right-hand tagging rules lhs, rhs by

$$\mathsf{lhs}(x) = \begin{cases} x_{\mathsf{L}} & \text{if } x \in \mathcal{V}_i \\ x & \text{if } x \in \mathcal{V}_o \end{cases}$$

$$\mathsf{rhs}(x) = \begin{cases} x_{\mathsf{R}} & \text{if } x \in \mathcal{V}_i \\ x & \text{if } x \in \mathcal{V}_o \end{cases}$$

Definition D.3. The unconstrained pushdown system $\mathcal{P}_\mathcal{C}$ associated to a constraint set $\mathcal{C}$ is given by the triple $(\widetilde{\mathcal{V}}, \widetilde{\Sigma}, \Delta)$ where

$$\begin{aligned}
\widetilde{\mathcal{V}} &= (\mathsf{lhs}(\mathcal{V}_i) \amalg \mathsf{rhs}(\mathcal{V}_i) \amalg \mathcal{V}_o) \times \{\oplus, \ominus\} \\
&\quad \cup \{^{\#}\textsc{Start}, {}^{\#}\textsc{End}\}
\end{aligned}$$

We will write an element $(v, \odot) \in \widetilde{\mathcal{V}}$ as $v^\odot$.

The stack alphabet $\widetilde{\Sigma}$ is essentially the same as Σ, with a few extra tokens added to represent interesting variables:

$$\widetilde{\Sigma} = \Sigma \cup \{v^\oplus \mid v \in \mathcal{V}_i\} \cup \{v^\ominus \mid v \in \mathcal{V}_i\}$$

To define the rewrite rules, we first introduce the helper functions:

$$\begin{aligned}
\mathsf{rule}^\oplus(p.u \sqsubseteq q.v) &= \langle \mathsf{lhs}(p)^{\langle u\rangle}; u\rangle \hookrightarrow \langle \mathsf{rhs}(q)^{\langle v\rangle}; v\rangle \\
\mathsf{rule}^\ominus(p.u \sqsubseteq q.v) &= \langle \mathsf{lhs}(q)^{\ominus\cdot\langle v\rangle}; v\rangle \hookrightarrow \langle \mathsf{rhs}(p)^{\ominus\cdot\langle u\rangle}; u\rangle \\
\mathsf{rules}(c) &= \{\mathsf{rule}^\oplus(c),\ \mathsf{rule}^\ominus(c)\}
\end{aligned}$$

The rewrite rules Δ are partitioned into four parts $\Delta = \Delta_{\mathcal{C}} \amalg \Delta_{\mathsf{ptr}} \amalg \Delta_{\mathsf{start}} \amalg \Delta_{\mathsf{end}}$ where

$$\Delta_{\mathcal{C}} = \bigcup_{c \in \mathcal{C}} \mathsf{rules}(c)$$
$$\Delta_{\mathsf{ptr}} = \bigcup_{v \in \mathcal{V}'} \mathsf{rules}(v.\mathsf{store} \sqsubseteq v.\mathsf{load})$$
$$\Delta_{\mathsf{start}} = \left\{ \langle {}^{\#}\textsc{Start}; v^{\oplus} \rangle \hookrightarrow \langle v_{\mathsf{L}}^{\oplus}; \varepsilon \rangle \mid v \in \mathcal{V}_i \right\} \cup \left\{ \langle {}^{\#}\textsc{Start}; v^{\ominus} \rangle \hookrightarrow \langle v_{\mathsf{L}}^{\ominus}; \varepsilon \rangle \mid v \in \mathcal{V}_i \right\}$$
$$\Delta_{\mathsf{end}} = \left\{ \langle v_{\mathsf{R}}^{\oplus}; \varepsilon \rangle \hookrightarrow \langle {}^{\#}\textsc{End}; v^{\oplus} \rangle \mid v \in \mathcal{V}_i \right\} \cup \left\{ \langle v_{\mathsf{R}}^{\ominus}; \varepsilon \rangle \hookrightarrow \langle {}^{\#}\textsc{End}; v^{\ominus} \rangle \mid v \in \mathcal{V}_i \right\}$$

Note 1. The $\{\oplus, \ominus\}$ superscripts on the control states are used to track the current variance of the stack state, allowing us to distinguish between uses of an axiom in co- and contravariant position. The tagging operations lhs, rhs are used to prevent derivations from making use of variables from $\mathcal{V}_i$, preventing $\mathcal{P}_{\mathcal{C}}$ from admitting derivations that represent non-elementary proofs.

Note 2. Note that although $\Delta_{\mathcal{C}}$ is finite, Δ_{ptr} contains rules for *every derived type variable* and is therefore infinite. We carefully adjust for this in the saturation rules below so that rules from Δ_{ptr} are only considered lazily as $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$ is constructed.

Lemma D.1. For any pair $(X^a u, Y^b v)$ in $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$, we also have $(Y^{\ominus \cdot b} v, X^{\ominus \cdot a} u) \in \mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$.

Proof. Immediate due to the symmetries in the construction of Δ. □

Lemma D.2. For any pair $(X^a u, Y^b v)$ in $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$, the relation

$$a \cdot \langle u \rangle = b \cdot \langle v \rangle$$

must hold.

Proof. This is an inductive consequence of how Δ is defined: the relation holds for every rule in $\Delta_{\mathcal{C}} \cup \Delta_{\mathsf{ptr}}$ due to the use of the rules function, and the sign in the exponent is propagated from the top of the stack during an application of a rule from Δ_{start} and back to the stack during an application of a rule from Δ_{end}. Since every derivation will begin with a rule from Δ_{start}, proceed by applying rules from $\Delta_{\mathcal{C}} \cup \Delta_{\mathsf{ptr}}$, and conclude with a rule from Δ_{end}, this completes the proof. □

Definition D.4. Let $\mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}} \subseteq \mathcal{V}'_i \times \mathcal{V}'_i$ be the relation on derived type variables induced by $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$ as follows:

$$\mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}} = \left\{ (X.u, Y.v) \mid (X^{\langle u \rangle} u, Y^{\langle v \rangle} v) \in \mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}} \right\}$$

Lemma D.3. If $(p, q) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}}$ and $\sigma \in \Sigma$ has $\langle \sigma \rangle = \oplus$, then $(p\sigma, q\sigma) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}}$ as well. If $\langle \sigma \rangle = \ominus$, then $(q\sigma, p\sigma) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}}$ instead.

Proof. Immediate, from symmetry considerations. □

Lemma D.4. $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$ can be partitioned into $\mathsf{Deriv}^{\oplus}_{\mathcal{P}_{\mathcal{C}}} \amalg \mathsf{Deriv}^{\ominus}_{\mathcal{P}_{\mathcal{C}}}$ where

$$\mathsf{Deriv}^{\oplus}_{\mathcal{P}_{\mathcal{C}}} = \left\{ (X^{\langle u \rangle} u, Y^{\langle v \rangle} v) \mid (X.u, Y.v) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}} \right\}$$
$$\mathsf{Deriv}^{\ominus}_{\mathcal{P}_{\mathcal{C}}} = \left\{ (Y^{\ominus \cdot \langle v \rangle} v, X^{\ominus \cdot \langle u \rangle} u) \mid (X.u, Y.v) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}} \right\}$$

In particular, $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$ can be entirely reconstructed from the simpler set $\mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}}$.

Theorem D.1. $\mathcal{C}$ has an elementary proof of the constraint $X.u \sqsubseteq Y.v$ if and only if $(X.u, Y.v) \in \mathsf{Deriv}'_{\mathcal{P}_{\mathcal{C}}}$.

Proof. We will prove the theorem by constructing a bijection between elementary proof trees and derivations in $\mathcal{P}_{\mathcal{C}}$.

Call a configuration (p^a, u) *positive* if $a = \langle u \rangle$. By lemma D.2, if (p^a, u) is positive and $(p^a, u) \xrightarrow{*} (q^b, v)$ then (q^b, v) is positive as well. We will also call a rewrite rule positive when the configurations appearing in the rule are positive. Note that by construction, every positive rewrite rule from $\Delta_{\mathcal{C}} \cup \Delta_{\mathsf{ptr}}$ is of the form $\mathsf{rule}^{\oplus}(c)$ for some axiom $c \in \mathcal{C}$ or load/store constraint $c = (p.\mathsf{store} \sqsubseteq p.\mathsf{load})$.

For any positive rewrite rule $R = \mathsf{rule}^{\oplus}(p.u \sqsubseteq q.v)$ and $\ell \in \Sigma$, let $S_\ell(R)$ denote the positive rule

$$S_\ell(R) = \begin{cases} \langle p^{\langle u\ell \rangle}; u\ell \rangle \hookrightarrow \langle q^{\langle v\ell \rangle}; v\ell \rangle & \text{if } \langle \ell \rangle = \oplus \\ \langle q^{\langle v\ell \rangle}; v\ell \rangle \hookrightarrow \langle p^{\langle u\ell \rangle}; u\ell \rangle & \text{if } \langle \ell \rangle = \ominus \end{cases}$$

Note that by the construction of Δ, each $S_\ell(R)$ is redundant, being a restriction of one of the existing rewrite rules $\mathsf{rule}^{\oplus}(p.u \sqsubseteq q.v)$ or $\mathsf{rule}^{\ominus}(p.u \sqsubseteq q.v)$ to a more specific stack configuration. In particular, adding $S_\ell(R)$ to the set of rules does not affect $\mathsf{Deriv}_{\mathcal{P}_{\mathcal{C}}}$.

Suppose we are in the positive state $(p^{\langle w \rangle}, w)$ and are about to apply a rewrite rule $R = \langle p^{\langle u \rangle}; u \rangle \hookrightarrow \langle q^{\langle v \rangle}; v \rangle$. Then we must have $w = u\ell_1\ell_2 \cdots \ell_n$, so that the left-hand side of the rule $S_{\ell_n}(\cdots(S_{\ell_1}(R))\cdots)$ is exactly $\langle p^{\langle w \rangle}; w \rangle$.

Let $R_{\mathsf{start}}, R_1, \cdots, R_n, R_{\mathsf{end}}$ be the sequence of rule applications used in an arbitrary derivation $({}^{\#}\textsc{Start}, p^{\langle u \rangle} u) \xrightarrow{*} ({}^{\#}\textsc{End}, q^{\langle v \rangle} v)$ and let $R \odot R'$ denote the application of rule R followed by rule R', with the right-hand side of R *exactly* matching the left-hand side of R'. Given the initial stack state $p^{\langle u \rangle} u$ and the sequence of rule applications, for each R_k there is a unique rule $R'_k = S_{\ell_n^k}(\cdots(S_{\ell_1^k}(R_k))\cdots)$ such that $\partial = R'_{\mathsf{start}} \odot R'_1 \odot \cdots \odot R'_n \odot R'_{\mathsf{end}}$ describes the derivation exactly. Now normalize ∂ using the rule $S_\ell(R_i) \odot S_\ell(R_j) \mapsto S_\ell(R_i \odot R_j)$; this process is clearly reversible, producing a bijection between these normalized expressions and derivations in $\mathsf{Deriv}^{\oplus}_{\mathcal{P}_{\mathcal{C}}}$.

To complete the proof, we obtain a bijection between normalized expressions and the normal forms of elementary proof trees. The bijection is straightforward: R_i represents the introduction of an axiom from $\mathcal{C}$, $\odot$ represents an application of the rule S-$\textsc{Trans}$, and S_ℓ represents an application of the rule S-$\textsc{Field}_\ell$. This results in an algebraic expression

describing the normalized proof tree as in section B.1, which completes the proof. □

D.2 Constructing the initial graph

In this section, we will construct a finite state automaton that accepts strings that encode some of the behavior of $\mathcal{P}_\mathcal{C}$; the next section describes a saturation algorithm that modifies the initial automaton to create a finite state transducer for $\mathsf{Deriv}'_{\mathcal{P}_\mathcal{C}}$.

The automaton $\mathcal{A}_\mathcal{C}$ is constructed as follows:

1. Add an initial state $^{\#}\textsc{Start}$ and an accepting state $^{\#}\textsc{End}$.
2. For each left-hand side $\langle p^a; u_1 \cdots u_n \rangle$ of a rule in $\Delta_\mathcal{C}$, add the transitions

$$^{\#}\textsc{Start} \xrightarrow{\mathsf{pop}\ p^{a\cdot\langle u_1 \dots u_n\rangle}} p^{a\cdot\langle u_1\dots u_n\rangle}$$
$$p^{a\cdot\langle u_1\dots u_n\rangle} \xrightarrow{\mathsf{pop}\ u_1} p^{a\cdot\langle u_2\dots u_n\rangle}u_1$$
$$\vdots$$
$$p^{a\cdot\langle u_n\rangle}u_1\dots u_{n-1} \xrightarrow{\mathsf{pop}\ u_n} p^a u_1 \dots u_n$$

3. For each right-hand side $\langle q^b; v_1 \cdots v_n \rangle$ of a rule in $\Delta_\mathcal{C}$, add the transitions

$$q^b v_1 \dots v_n \xrightarrow{\mathsf{push}\ v_n} q^{b\cdot\langle v_n\rangle} v_1 \dots v_{n-1}$$
$$\vdots$$
$$q^{b\cdot\langle v_2\dots v_n\rangle} v_1 \xrightarrow{\mathsf{push}\ v_1} q^{b\cdot\langle v_1\dots v_n\rangle}$$
$$q^{b\cdot\langle v_1\dots v_n\rangle} \xrightarrow{\mathsf{push}\ q^{b\cdot\langle v_1\dots v_n\rangle}} {}^{\#}\textsc{End}$$

4. For each rule $\langle p^a; u\rangle \hookrightarrow \langle q^b; v\rangle$, add the transition

$$p^a u \xrightarrow{\underline{1}} q^b v$$

Definition D.5. Following Carayol and Hague [5], we call a sequence of transitions $\mathsf{p}_1 \cdots \mathsf{p}_n$ *productive* if the corresponding element $\mathsf{p}_1 \otimes \cdots \otimes \mathsf{p}_n$ in $\mathsf{StackOp}_\Sigma$ is not $\underline{0}$. A sequence is productive if and only if it contains no adjacent terms of the form $\mathsf{pop}\ y, \mathsf{push}\ x$ with $x \neq y$.

Lemma D.5. There is a one-to-one correspondence between productive transition sequences accepted by $\mathcal{A}_\mathcal{C}$ and elementary proofs in $\mathcal{C}$ that do not use the axiom S-Pointer.

Proof. A productive transition sequence must consist of a sequence of pop edges followed by a sequence of push edges, possibly with insertions of unit edges or intermediate sequences of the form

$$\mathsf{push}\ (u_n) \otimes \cdots \otimes \mathsf{push}\ (u_1) \otimes \mathsf{pop}\ (u_1) \otimes \cdots \otimes \mathsf{pop}\ (u_n)$$

Following a push or pop edge corresponds to observing or forgetting part of the stack. Following a $\underline{1}$ edge corresponds to applying a PDS rule from $\mathcal{P}_\mathcal{C}$. □

D.3 Saturation

The label sequences appearing in lemma D.5 are tantalizingly close to having the simple structure of building up a pop sequence representing an initial state of the pushdown automaton, then building up a push sequence representing a final state. But the intermediate terms of the form "push u, then pop it again" are unwieldy. To remove the necessity for those sequences, we can *saturate* $\mathcal{A}_\mathcal{C}$ by adding additional $\underline{1}$-labeled transitions providing shortcuts to the push/pop subsequences. We modify the standard saturation algorithm to also lazily instantiate transitions which correspond to uses of the S-Pointer-derived rules in Δ_{ptr}.

Lemma D.6. The reachable states of the automaton $\mathcal{A}_\mathcal{C}$ can be partitioned into *covariant* and *contravariant* states, where a state's variance is defined to be the variance of any sequence reaching the state from $^{\#}\textsc{Start}$.

Proof. By construction of $\Delta_\mathcal{C}$ and $\mathcal{A}_\mathcal{C}$. □

Lemma D.7. There is an involution $n \mapsto \overline{n}$ on $\mathcal{A}_\mathcal{C}$ defined by

$$\overline{x^a u} = x^{\ominus\cdot a} u$$
$$\overline{^{\#}\textsc{Start}} = {}^{\#}\textsc{End}$$
$$\overline{^{\#}\textsc{End}} = {}^{\#}\textsc{Start}$$

Proof. Immediate due to the use of the rule constructors $\mathsf{rule}^{\oplus}$ and $\mathsf{rule}^{\ominus}$ when forming $\Delta_\mathcal{C}$. □

In this section, the automaton $\mathcal{A}_\mathcal{C}$ will be *saturated* by adding transitions to create a new automaton $\mathcal{A}^{\mathsf{sat}}_\mathcal{C}$.

Definition D.6. A sequence is called *reduced* if it is productive and contains no factors of the form $\mathsf{pop}\ x \otimes \mathsf{push}\ x$.

Reduced productive sequences all have the form of a sequence of pops, followed by a sequence of pushes.
The goal of the saturation algorithm is twofold:

1. Ensure that for any productive sequence accepted by $\mathcal{A}_\mathcal{C}$ there is an equivalent reduced sequence accepted by $\mathcal{A}^{\mathsf{sat}}_\mathcal{C}$.
2. Ensure that $\mathcal{A}^{\mathsf{sat}}_\mathcal{C}$ can represent elementary proofs that use S-Pointer.

The saturation algorithm D.2 proceeds by maintaining, for each state $q \in \mathcal{A}_\mathcal{C}$, a set of reaching-pushes $R(q)$. The reaching push set $R(q)$ will contain the pair (ℓ, p) only if there is a transition sequence in $\mathcal{A}_\mathcal{C}$ from p to q with weight $\mathsf{push}\ \ell$. When q has an outgoing $\mathsf{pop}\ \ell$ edge to q' and $(\ell, p) \in R(q)$, we add a new transition $p \xrightarrow{\underline{1}} q'$ to $\mathcal{A}_\mathcal{C}$.

A special propagation clause is responsible for propagating reaching-push facts as if rules from Δ_{ptr} were instantiated, allowing the saturation algorithm to work even though the corresponding unconstrained pushdown system has infinitely many rules. This special clause is justified by considering the standard saturation rule when $x.\mathsf{store} \sqsubseteq x.\mathsf{load}$

Algorithm D.1 Converting a transducer from a set of pushdown system rules

procedure ALLPATHS(V, E, x, y) ▷ Tarjan's path algorithm. Return a finite state automaton recognizing the label sequence for all paths from x to y in the graph (V, E).
    ...
    return Q
end procedure

procedure TRANSDUCER($\Delta_{\mathcal{C}}$) ▷ $\Delta_{\mathcal{C}}$ is a set of pushdown system rules $\langle p^a; u\rangle \hookrightarrow \langle q^b; v\rangle$
    $V \leftarrow \{^{\#}\textsc{Start}, ^{\#}\textsc{End}\}$
    $E \leftarrow \emptyset$
    for all $\langle p^a; u_1 \cdots u_n\rangle \hookrightarrow \langle q^b; v_1 \cdots v_m\rangle \in \Delta_{\mathcal{C}}$ **do**
        $V \leftarrow V \cup \{p^{a\cdot\langle u_1 \cdots u_n\rangle}, q^{b\cdot\langle v_1 \cdots v_m\rangle}\}$
        $E \leftarrow E \cup \{(^{\#}\textsc{Start}, p^{a\cdot\langle u_1 \cdots u_n\rangle}, \mathsf{pop}\ p^{a\cdot\langle u_1 \cdots u_n\rangle})\}$
        $E \leftarrow E \cup \{(q^{b\cdot\langle v_1 \cdots v_m\rangle}, ^{\#}\textsc{End}, \mathsf{push}\ q^{b\cdot\langle v_1 \cdots v_m\rangle})\}$
        for $i \leftarrow 1 \ldots n-1$ **do**
            $V \leftarrow V \cup \{p^{a\cdot\langle u_{i+1} \cdots u_n\rangle} u_1 \ldots u_i\}$
            $E \leftarrow E \cup \{(p^{a\cdot\langle u_i \cdots u_n\rangle} u_1 \ldots u_{i-1}, p^{a\cdot\langle u_{i+1} \cdots u_n\rangle} u_1 \ldots u_i, \mathsf{pop}\ u_i)\}$
        end for
        for $j \leftarrow 1 \ldots m-1$ **do**
            $V \leftarrow V \cup \{q^{b\cdot\langle v_{j+1} \cdots u_m\rangle} v_1 \ldots v_j\}$
            $E \leftarrow E \cup \{(q^{b\cdot\langle v_{j+1} \cdots v_m\rangle} v_1 \ldots v_j, q^{b\cdot\langle v_j \cdots v_m\rangle} v_1 \ldots v_{j-1}, \mathsf{push}\ v_j)\}$
        end for
        $E \leftarrow E \cup \{(p^a u_1 \ldots u_n, q^b v_1 \ldots v_m, \underline{1})\}$
    end for
    $E \leftarrow \textsc{Saturated}(V, E)$
    $Q \leftarrow \textsc{AllPaths}(V, E, ^{\#}\textsc{Start}, ^{\#}\textsc{End})$
    return Q
end procedure

is added as an axiom in $\mathcal{C}$. An example appears in figure 10: a saturation edge is added from $x^{\oplus}.\mathsf{store}$ to $y^{\oplus}.\mathsf{load}$ due to the pointer saturation rule, but the same edge would also have been added if the states and transitions corresponding to $p.\mathsf{store} \sqsubseteq p.\mathsf{load}$ (depicted with dotted edges and nodes) were added to $\mathcal{A}_{\mathcal{C}}$.

Once the saturation algorithm completes, the automaton $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$ has the property that, if there is a transition sequence from p to q with weight *equivalent* to

$$\mathsf{pop}\ u_1 \otimes \cdots \otimes \mathsf{pop}\ u_n \otimes \mathsf{push}\ v_m \otimes \cdots \otimes \mathsf{push}\ v_1,$$

then there is a path from p to q that has, ignoring $\underline{1}$ edges, *exactly* the label sequence $\mathsf{pop}\ u_1, \ldots, \mathsf{pop}\ u_n, \mathsf{push}\ v_m, \ldots, \mathsf{push}\ v_1$.

D.4 Shadowing

The automaton $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$ now accepts push/pop sequences representing the changes in the stack during any legal derivation in the pushdown system Δ. After saturation, we can guarantee that every derivation is represented by a path which first pops a sequence of tokens, then pushes another sequence of tokens.

Unfortunately $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$ still accepts unproductive transition sequences which push and then immediately pop token sequences. To complete the construction, we form an automaton Q by intersecting $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$ with an automaton for the language of words consisting of only pops, followed by only pushes.

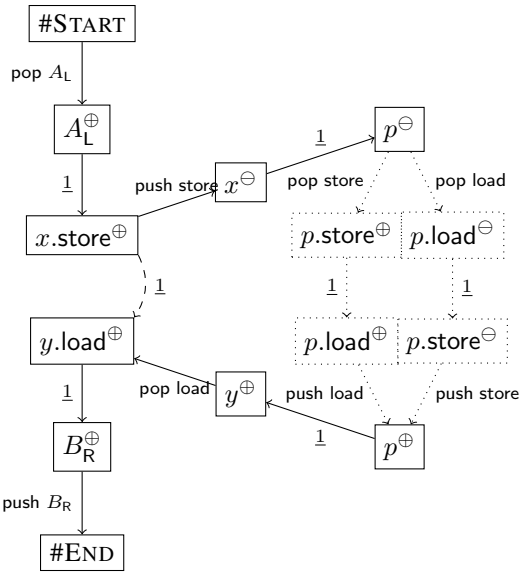
This final transformation yields an automaton Q with the property that for every transition sequence s accepted by $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$, Q accepts a sequence s' such that $[\![s]\!] = [\![s']\!]$ in $\mathsf{StackOp}$ and s' consists of a sequence of pops followed by a sequence of pushes. This ensures that Q only accepts the productive transition sequences in $\mathcal{A}_{\mathcal{C}}^{\mathsf{sat}}$. Finally, we can treat Q as a finite-state transducer by treating transitions labeled $\mathsf{pop}\ \ell$ as reading the symbol ℓ from the input tape, $\mathsf{push}\ \ell$ as writing ℓ to the output tape, and $\underline{1}$ as an ε-transition.

Q can be further manipulated through determinization and/or minimization to produce a compact transducer representing the valid transition sequences through $\mathcal{P}_{\mathcal{C}}$. Taken as a whole, we have shown that $Xu \overset{Q}{\mapsto} Yv$ if and only if X and Y are interesting variables and there is an elementary derivation of $\mathcal{C} \vdash X.u \sqsubseteq Y.v$.

We make use of this process in two places during type analysis: first, by computing Q relative to the type variable of a function, we get a transducer that represents all elementary derivations of relationships between the function's inputs and outputs. Algorithm D.3 is used to convert the transducer Q back to a pushdown system $\mathcal{P}$, such that Q describes all valid derivations in $\mathcal{P}$. Then the rules in $\mathcal{P}$ can be interpreted

Algorithm D.2 Saturation algorithm

procedure SATURATED(V, E) ▷ V is a set of vertices partitioned into $V = V^{\oplus} \amalg V^{\ominus}$
  $E' \leftarrow E$ ▷ E is a set of edges, represented as triples $(\mathsf{src}, \mathsf{tgt}, \mathsf{label})$
  for all $x \in V$ **do**
    $R(x) \leftarrow \emptyset$
  end for
  for all $(x, y, e) \in E$ with $e = \mathsf{push}\ \ell$ **do** ▷ Initialize the reaching-push sets $R(x)$
    $R(y) \leftarrow R(y) \cup \{(\ell, x)\}$
  end for
  repeat
    $R_{\text{old}} \leftarrow R$
    $E'_{\text{old}} \leftarrow E'$
    for all $(x, y, e) \in E'$ with $e = \underline{1}$ **do**
      $R(y) \leftarrow R(y) \cup R(x)$
    end for
    for all $(x, y, e) \in E'$ with $e = \mathsf{pop}\ \ell$ **do**
      for all $(\ell, z) \in R(x)$ **do** ▷ The standard saturation rule.
        $E' \leftarrow E' \cup \{(z, y, \underline{1})\}$
      end for
    end for
    for all $x \in V^{\ominus}$ **do** ▷ Lazily apply saturation rules corresponding to S-POINTER.
      for all $(\ell, z) \in R(x)$ with $\ell = .\mathsf{store}$ **do** ▷ See figure 10 for an example.
        $R(\overline{x}) \leftarrow R(\overline{x}) \cup \{(.\mathsf{load}, z)\}$
      end for
      for all $(\ell, z) \in R(x)$ with $\ell = .\mathsf{load}$ **do**
        $R(\overline{x}) \leftarrow R(\overline{x}) \cup \{(.\mathsf{store}, z)\}$
      end for
    end for
  until $R = R_{\text{old}}$ and $E' = E'_{\text{old}}$
  return E'
end procedure




```
{
  p =  y;
  x =  p;
 *x =  A;
  B = *y;
}
```

Figure 10. Saturation using an implicit application of $p.\mathsf{store} \sqsubseteq p.\mathsf{load}$. The initial constraint set was $\{y \sqsubseteq p, \quad p \sqsubseteq x, \quad A \sqsubseteq x.\mathsf{store}, \quad y.\mathsf{load} \sqsubseteq B\}$, modeling the simple program on the right. The dashed edge was added by a saturation rule that only fires because of the lazy handling in algorithm D.2. The dotted states and edges show how the graph would look if the corresponding rule from Δ_{ptr} were explicitly instantiated.

as subtype constraints, resulting in a simplification of the constraint set relative to the formal type variables.

Second, by computing Q relative to the set of type constants we obtain a transducer that can be efficiently queried to determine which derived type variables are bound above or below by which type constants. This is used by the SOLVE procedure in algorithm F.2 to populate lattice elements decorating the inferred sketches.

Algorithm D.3 Converting a transducer to a pushdown system

```
procedure TypeScheme(Q)
    Δ ← new PDS
    Δ.states ← Q.states
    for all p -t-> q ∈ Q.transitions do
        if t = pop ℓ then
            AddPDSRule(Δ, ⟨p; ℓ⟩ ↪ ⟨q; ε⟩)
        else
            AddPDSRule(Δ, ⟨p; ε⟩ ↪ ⟨q; ℓ⟩)
        end if
    end for
    return Δ
end procedure
```

E. The lattice of sketches

Throughout this section, we fix a lattice $(\Lambda, <:, \vee, \wedge)$ of atomic types. We do not assume anything about Λ except that it should have finite height so that infinite subtype chains eventually stabilize. For example purposes, we will take Λ to be the lattice of semantic classes depicted in figure 11.

Our initial implementation of sketches did not use the auxiliary lattice Λ. We found that adding these decorations to the sketches helped preserve high-level types of interest to the end user during type inference. This allows us to recover high-level C and Windows typedefs such as `size_t`, `FILE`, `HANDLE`, and `SOCKET` that are useful for program understanding and reverse engineering, as noted in Lin et al. [17].

Decorations also enable a simple mechanism by which the user can extend Retypd's type system, adding semantic purposes to the types for known functions. For example, we can extend Λ to add seeds for a tag `#signal-number` attached to the less-informative `int` parameter to `signal()`. This approach also allows us to distinguish between opaque typedefs and the underlying type, as in `HANDLE` and `void*`. Since the semantics of a `HANDLE` are quite distinct from those of a `void*`, it is important to have a mechanism that can preserve the typedef name.

E.1 Basic definitions

Definition E.1. A *sketch* is a regular tree with edges labeled by elements of Σ and nodes labeled with elements of Λ. The set of all sketches, with Σ and Λ implicitly fixed, will be denoted Sk. We may alternately think of a sketch as a prefix-closed regular language $\mathcal{L}(S) \subseteq \Sigma^*$ and a function $\nu : S \to \Lambda$ such that each fiber $\nu^{-1}(\lambda)$ is regular. It will be convenient to write $\nu_S(w)$ for the value of ν at the node of S reached by following the word $w \in \Sigma^*$.

By collapsing equal subtrees, we can represent sketches as deterministic finite state automata with each state labeled by an element of Λ, as in figure 13. Since the regular language associated with a sketch is prefix-closed, all states of the associated automaton are accepting.

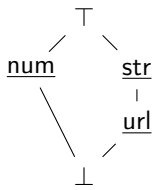


Figure 11. The sample lattice Λ of atomic types.

Language

$$\mathcal{L}(X \sqcap Y) = \mathcal{L}(X) \cup \mathcal{L}(Y)$$

$$\mathcal{L}(X \sqcup Y) = \mathcal{L}(X) \cap \mathcal{L}(Y)$$

Node labels

$$\nu_{X \sqcap Y}(w) = \begin{cases} \nu_X(w) \wedge \nu_Y(w) & \text{if } \langle w \rangle = \oplus \\ \nu_X(w) \vee \nu_Y(w) & \text{if } \langle w \rangle = \ominus \end{cases}$$

$$\nu_{X \sqcup Y}(w) = \begin{cases} \nu_X(w) & \text{if } w \in \mathcal{L}(X) \setminus \mathcal{L}(Y) \\ \nu_Y(w) & \text{if } w \in \mathcal{L}(Y) \setminus \mathcal{L}(X) \\ \nu_X(w) \vee \nu_Y(w) & \text{if } w \in \mathcal{L}(X) \cap \mathcal{L}(Y), \\ & \quad \langle w \rangle = \oplus \\ \nu_X(w) \wedge \nu_Y(w) & \text{if } w \in \mathcal{L}(X) \cap \mathcal{L}(Y), \\ & \quad \langle w \rangle = \ominus \end{cases}$$

Figure 12. Lattice operations on the set of sketches.

Lemma E.1. The set of sketches Sk forms a lattice with meet and join operations $\sqcap, \sqcup$ defined according to figure 12. Sk has a top element given by the sketch accepting the language $\{\varepsilon\}$, with the single node labeled by $\top \in \Lambda$.

If Σ is finite, Sk also has a bottom element accepting the language Σ^*, with label function $\nu_\bot(w) = \bot$ when $\langle w \rangle = \oplus$, or $\nu_\bot(w) = \top$ when $\langle w \rangle = \ominus$.

We will use $X \trianglelefteq Y$ to denote the partial order on sketches compatible with the lattice operations, so that $X \sqcap Y = X$ if and only if $X \trianglelefteq Y$.

E.1.1 Modeling constraint solutions with sketches

Sketches are our choice of entity for modeling solutions to the constraint sets of section 3.1.

Definition E.2. A *solution* to a constraint set $\mathcal{C}$ over the type variables $\mathcal{V}$ is a set of bindings $S : \mathcal{V} \to \mathsf{Sk}$ such that

- If $\overline{\kappa}$ is a type constant, $\mathcal{L}(S_\kappa) = \{\varepsilon\}$ and $\nu_{S_\kappa}(\varepsilon) = \kappa$.
- If $\mathcal{C} \vdash \textsc{Var}\ X.v$ then $v \in \mathcal{L}(X)$.
- If $\mathcal{C} \vdash X.u \sqsubseteq Y.v$ then $\nu_X(u) <: \nu_Y(v)$.
- If $\mathcal{C} \vdash X.u \sqsubseteq Y.v$ then $u^{-1}S_X \trianglelefteq v^{-1}S_Y$, where $v^{-1}S_X$ is the sketch corresponding to the subtree reached by following the path v from the root of S_X.

The main utility of sketches is that they are almost a free tree model of the constraint language. Any constraint set $\mathcal{C}$ is satisfiable over the lattice of sketches, as long as $\mathcal{C}$ cannot prove an impossible subtype relation in Λ.

Theorem E.1. Suppose that $\mathcal{C}$ is a constraint set over the variables $\{\tau_i\}_{i\in I}$. Then there exist sketches $\{S_i\}_{i\in I}$ such that $w \in S_i$ if and only if $\mathcal{C} \vdash \textsc{Var}\ \tau_i.w$.

Proof. The languages $\mathcal{L}(S_i)$ can be computed an algorithm that is similar in spirit to Steensgaard's method of almost-linear-time pointer analysis [30]. Begin by forming a graph with one node $n(\alpha)$ for each derived type variable appearing in $\mathcal{C}$, along with each of its prefixes. Add a labeled edge $n(\alpha) \xrightarrow{\ell} n(\alpha.\ell)$ for each derived type variable $\alpha.\ell$ to form a graph G. Now quotient G by the equivalence relation $\sim$ defined by $n(\alpha) \sim n(\beta)$ if $\alpha \sqsubseteq \beta \in \mathcal{C}$, and $n(\alpha') \sim n(\beta')$ whenever there are edges $n(\alpha) \xrightarrow{\ell} n(\alpha')$ and $n(\beta) \xrightarrow{\ell'} n(\beta')$ in G with $n(\alpha) \sim n(\beta)$ where either $\ell = \ell'$ or $\ell = .\mathsf{load}$ and $\ell' = .\mathsf{store}$.

The relation $\sim$ is the symmetrization of $\sqsubseteq$, with the first defining rule roughly corresponding to T-InheritL/T-InheritR and the second rule corresponding to S-Field$_\oplus$/S-Field$_\ominus$. The unusual condition on ℓ and ℓ' is due to the S-Pointer rule.

By construction, there exists a path with label sequence u through $G/\!\sim$ starting at the equivalence class of τ_i if and only if $\mathcal{C} \vdash \textsc{Var}\ \tau_i.u$. We can take this as the definition of the language accepted by S_i. □

Working out the lattice elements that should label S_i is a trickier problem; the basic idea is to use the same pushdown system construction that appears during constraint simplification to answer queries about which type constants are upper and lower bounds on a given derived type variable. The computation of upper- and lower- lattice bounds on a derived type variable can be found in appendix D.4.

E.1.2 Sketch narrowing at function calls

It was noted in section 3.2 that the rule T-InheritR leads to a system with structural typing: any two types in a subtype relation must have the same fields. In the language of sketches, this means that if two type variables are in a subtype relation then the corresponding sketches accept exactly the same languages. Superficially, this seems problematic for modeling typecasts that narrow a pointed-to object as motivated by the idioms in section 2.4.

Algorithm E.1 Computing sketches from constraint sets

```
procedure InferShapes(C_initial, B)
    C ← Substitute(C_initial, B)
    G ← ∅                          ▷ Compute constraint graph modulo ∼
    for all p.ℓ1 ... ℓn ∈ C.derivedTypeVars do
        for i ← 1 ... n do
            s ← FindEquivRep(p.ℓ1 ... ℓ_{i−1}, G)
            t ← FindEquivRep(p.ℓ1 ... ℓ_i, G)
            G.edges ← G.edges ∪ (s, t, ℓ_i)
        end for
    end for
    for all x ⊑ y ∈ C do
        X ← FindEquivRep(x, G)
        Y ← FindEquivRep(y, G)
        Unify(X, Y, G)
    end for
    repeat                         ▷ Apply additive constraints and update G
        C_old ← C
        for all c ∈ C_old with c = Add(_) or Sub(_) do
            D ← ApplyAddSub(c, G, C)
            for all δ ∈ D with δ = X ⊑ Y do
                Unify(X, Y, B)
            end for
        end for
    until C_old = C
    for all v ∈ C.typeVars do      ▷ Infer initial sketches
        S ← new Sketch
        L(S) ← AllPathsFrom(v, G)
        for all states w ∈ S do
            if ⟨w⟩ = ⊕ then
                ν_S(w) ← ⊤
            else
                ν_S(w) ← ⊥
            end if
        end for
        B[v] ← S
    end for
end procedure

procedure Unify(X, Y, G)           ▷ Make X ∼ Y in G
    if X ≠ Y then
        MakeEquiv(X, Y, G)
        for all (X′, ℓ) ∈ G.outEdges(X) do
            if (Y′, ℓ) ∈ G.outEdges(Y) for some Y′ then
                Unify(X′, Y′, G)
            end if
        end for
    end if
end procedure
```

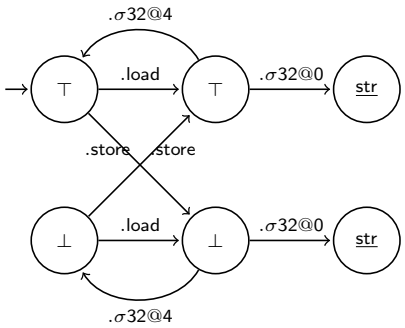


Figure 13. Sketch representing a linked list of strings `struct LL { str s; struct LL * a; }*`.

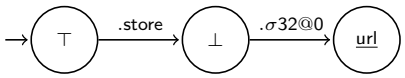


Figure 14. Sketch for Y representing `_Out_ url * u;`

The missing piece that allows us to effectively narrow objects is instantiation of callee type schemes at a callsite. To demonstrate how polymorphism enables narrowing, consider the example type scheme $\forall F.\mathcal{C} \Rightarrow F$ from figure 2. The function `close_last` can be invoked by providing any actual-in type α such that $\alpha \sqsubseteq F.\mathsf{in}_{\mathsf{stack0}}$; in particular, α can have *more* capabilities than $F.\mathsf{in}_{\mathsf{stack0}}$ itself. That is, we can pass a more constrainted ("more capable") type as an actual-in to a function that expected a less constrained input. In this way, we recover aspects of the physical, nonstructural subtyping utilized by many C and C++ programs via the pointer-to-member or pointer-to-substructure idioms described in section 2.4.

Example E.1. Suppose we have a reverse DNS lookup function `reverse_dns` with C type signature `void reverse_dns(num addr, url* result)`. Furthermore, assume that the implementation of `reverse_dns` works by writing the resulting URL to the location pointed to by `result`, yielding a constraint of the form $\underline{\mathsf{url}} \sqsubseteq$ `result`$.\mathsf{store}.\sigma32@0$. So `reverse_dns` will have an inferred type scheme of the form

$$\forall \alpha, \beta.(\alpha \sqsubseteq \underline{\mathsf{num}}, \underline{\mathsf{url}} \sqsubseteq \beta.\mathsf{store}.\sigma32@0) \Rightarrow \alpha \times \beta \to \mathsf{void}$$

Now suppose we have the structure `LL` of figure 13 representing a linked list of strings, with instance `LL * myList`. Can we safely invoke `reverse_dns(addr, (url*) myList)`?

Intuitively, we can see that this should be possible: since the linked list's payload is in its first field, a value of type `LL*` also looks like a value of type `str*`. Furthermore, `myList` is not `const`, so it can be used to store the function's output.

Is this intuition borne out by Retypd's type system? The answer is yes, though it takes a bit of work to see why. Let us write S_β for the instantiated sketch of β at the callsite of `reverse_dns` in question; the constraints $\mathcal{C}$ require that S_β satisfy $.\mathsf{store}.\sigma32@0 \in \mathcal{L}(S_\beta)$ and $\underline{\mathsf{url}} <: \nu_{S_\beta}(.\mathsf{store}.\sigma32@0)$. The actual-in has the type sketch S_X seen in figure 13, and the copy from actual-in to formal-in will generate the constraint $X \sqsubseteq \beta$.

Since we have already satisfied the two constraints on S_β coming from its use in `reverse_dns`, we can freely add other words to $\mathcal{L}(S_\beta)$, and can freely set their node labels to almost any value we please subject only to the constraint $S_X \trianglelefteq S_\beta$. This is simple enough: we just add every word in $\mathcal{L}(S_X)$ to $\mathcal{L}(S_\beta)$. If w is a word accepted by $\mathcal{L}(S_X)$ that must be added to $\mathcal{L}(S_\beta)$, we will define $\nu_{S_\beta}(w) := \nu_{S_X}(w)$. Thus, both the shape and the node labeling on S_X and S_β match, with one possible exception: we must check that X has a nested field $.\mathsf{store}.\sigma32@0$, and that the node labels satisfy the relation

$$\nu_{S_\beta}(.\mathsf{store}.\sigma32@0) <: \nu_{S_X}(.\mathsf{store}.\sigma32@0)$$

since $.\mathsf{store}.\sigma32@0$ is contravariant and $S_X \trianglelefteq S_\beta$. X does indeed have the required field, and $\underline{\mathsf{url}} <: \underline{\mathsf{str}}$; the function invocation is judged to be type-safe.

F. Additional algorithms

This appendix holds a few algorithms referenced in the main text and other appendices.

Algorithm F.1 Type scheme inference

```
procedure INFERPROCTYPES(CallGraph)
    T ← ∅   ▷ T is a map from procedure to type scheme.
    for all S ∈ POSTORDER(CallGraph.sccs) do
        C ← ∅
        for all P ∈ S do
            T[P] ← ∅
        end for
        for all P ∈ S do
            C ← C ∪ CONSTRAINTS(P, T)
        end for
        C ← INFERSHAPES(C, ∅)
        for all P ∈ S do
            V ← P.formalIns ∪ P.formalOuts
            Q ← TRANSDUCER(C, V ∪ Λ̄)
            T[P] ← TYPESCHEME(Q)
        end for
    end for
end procedure

procedure CONSTRAINTS(P, T)
    C ← ∅
    for all i ∈ P.instructions do
        C ← C ∪ ABSTRACTINTERP(TypeInterp, i)
        if i calls Q then
            C ← C ∪ INSTANTIATE(T[Q], i)
        end if
    end for
    return C
end procedure
```

Algorithm F.2 C type inference

procedure INFERTYPES(CallGraph, T)
  $B \leftarrow \emptyset$ ▷ B is a map from type variable to sketch.
  for all $S \in$ REVERSEPOSTORDER(CallGraph.sccs) **do**
    $\mathcal{C} \leftarrow \emptyset$
    for all $P \in S$ **do**
      $T[P] \leftarrow \emptyset$
    end for
    for all $P \in S$ **do**
      $\mathcal{C}_\partial \leftarrow T[P]$
      SOLVE($\mathcal{C}_\partial, B$)
      REFINEPARAMETERS(P, B)
      $\mathcal{C} \leftarrow$ CONSTRAINTS(P, T)
      SOLVE($\mathcal{C}, B$)
    end for
  end for
  $A \leftarrow \emptyset$
  for all $x \in B$.keys **do**
    $A[x] \leftarrow$ SKETCHTOAPPXCTYPE($B[x]$)
  end for
  return A
end procedure

procedure SOLVE($\mathcal{C}, B$)
  $\mathcal{C} \leftarrow$ INFERSHAPES($\mathcal{C}, B$)
  $Q \leftarrow$TRANSDUCER($\mathcal{C}, \overline{\Lambda}$)
  for all $\lambda \in \Lambda$ **do**
    for all Xu such that $\lambda \overset{Q}{\mapsto} Xu$ **do**
      $\nu_{B[X]}(u) \leftarrow \nu_{B[X]}(u) \vee \lambda$
    end for
    for all Xu such that $Xu \overset{Q}{\mapsto} \lambda$ **do**
      $\nu_{B[X]}(u) \leftarrow \nu_{B[X]}(u) \wedge \lambda$
    end for
  end for
end procedure

Algorithm F.3 Procedure specialization

procedure REFINEPARAMETERS(P, B)
  for all $i \in P$.formalIns **do**
    $\lambda \leftarrow \top$
    for all $a \in P$.actualIns(i) **do**
      $\lambda \leftarrow \lambda \sqcup B[a]$
    end for
    $B[i] \leftarrow B[i] \sqcap \lambda$
  end for
  for all $o \in P$.formalOuts **do**
    $\lambda \leftarrow \bot$
    for all $a \in P$.actualOuts(o) **do**
      $\lambda \leftarrow \lambda \sqcap B[a]$
    end for
    $B[o] \leftarrow B[o] \sqcup \lambda$
  end for
end procedure

G. Other C type resolution policies

Example G.1. The initial type simplification stage results in types that are as general as possible. Often, this means that types are found to be more general than is strictly helpful to a (human) observer. A policy called REFINEPARAMETERS is used to specialize type schemes to the most *specific* scheme that is compatible with all uses. For example, a C++ object may include a getter function with a highly polymorphic type scheme, since it could operate equally well on any structure with a field of the right type at the right offset. But we expect that in every calling context, the getter will be called on a specific object type (or perhaps its derived types). By specializing the function signature, we make use of contextual clues in exchange for generality before presenting a final C type to the user.

Example G.2. Suppose we have a C++ class

```
class MyFile
{
 public:
   char * filename() const {
     return m_filename;
   }
 private:
   FILE * m_handle;
   char * m_filename;
};
```

In a 32-bit binary, the implementation of `MyFile::filename` (if not inlined) will be roughly equivalent to the C code

```
typedef int32_t dword;
dword get_filename(const void * this)
{
   char * raw_ptr = (char*) this;
   dword * field_ptr =
       (dword*) (raw_ptr + 4);
   return *field_ptr;
}
```

Accordingly, we would expect the most-general inferred type scheme for `MyFile::filename` to be

$$\forall \alpha, \beta.(\beta \sqsubseteq \texttt{dword}, \alpha.\mathsf{load}.\sigma 32@4 \sqsubseteq \beta) \Rightarrow \alpha \rightarrow \beta$$

indicating that `get_filename` will accept a pointer to anything which has a value of some 32-bit type β at offset 4, and will return a value of that same type. If the function is truly used polymorphically, this is exactly the kind of precision that we wanted our type system to maintain.

But in the more common case, `get_filename` will only be called with values where `this` has type `MyFile*` (or perhaps a subtype, if we include inheritance). If every call-site to `get_filename` passes it a pointer to `MyFile*`, it may be best to specialize the type of `get_function` to the monomorphic type

$$\texttt{get\_filename} : \texttt{const MyFile*} \rightarrow \texttt{char*}$$

The function REFINEPARAMETERS in algorithm F.3 is used to specialize each function's type *just enough* to match

how the function is actually used in a program, at the cost of reduced generality.

Example G.3. A useful but less sound heuristic is represented by the `reroll` policy for handling types which look like unrolled recursive types:

```
reroll(x):
 if there are u and ℓ with x.ℓu = x.ℓ,
      and sketch(x) ⊴ sketch(x.ℓ):
   replace x with x.ℓ
 else:
   policy does not apply
```

In practice, we often need to add other guards which inspect the shape of x to determine if the application of reroll appears to be appropriate or not. For example, we may require x to have at least one field other than ℓ to help distinguish a pointer-to-linked-list from a pointer-to-pointer-to-linked-list.

H. Details of the Figure 2 example

The results of constraint generation for the example program in Figure 2 appears in Figure 16. The constraint simplification algorithm builds the automaton Q (Figure 15) to recognize the simplified entailment closure of the constraint set. Q recognizes exactly the input/output pairs of the form

$$(\mathsf{close\_last.in}_{\mathsf{stack0}}(.\mathsf{load}.\sigma 32@0)^*.\mathsf{load}.\sigma 32@4, (\mathrm{int} \mid \#\mathrm{FileDescriptor}))$$

and

$$((\mathrm{int} \mid \#\mathrm{SuccessZ}), \mathsf{close\_last.out}_{\mathsf{eax}})$$

To generate the simplified constraint set, a type variable τ is synthesized for the single internal state in Q. The path leading from the start state to τ generates the constraint

$$\mathsf{close\_last.in}_{\mathsf{stack0}} \sqsubseteq \tau$$

The loop transition generates

$$\tau.\mathsf{load}.\sigma 32@0 \sqsubseteq \tau$$

and the two transitions out of τ generate

$$\tau.\mathsf{load}.\sigma 32@4 \sqsubseteq \mathrm{int}$$
$$\tau.\mathsf{load}.\sigma 32@4 \sqsubseteq \#\mathrm{FileDescriptor}$$

Finally, the two remaining transitions from start to end generate

$$\mathrm{int} \sqsubseteq \mathsf{close\_last.out}_{\mathsf{eax}}$$
$$\#\mathrm{SuccessZ} \sqsubseteq \mathsf{close\_last.out}_{\mathsf{eax}}$$

To generate the simplified constraint set, we gather up these constraints (applying some lattice operations to combine inequalities that only differ by a lattice constant) and close over the introduced τ by introducing an $\exists\tau$ quantifier. The result is the constraint set of Figure 2.

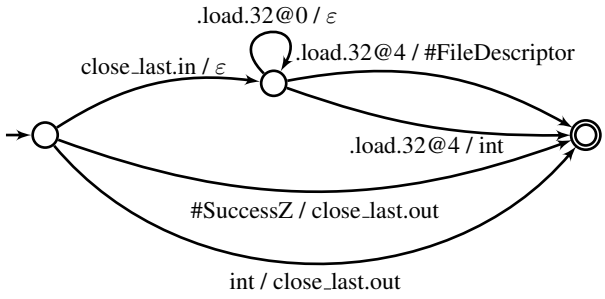


Figure 15. The automaton Q for the constraint system in Figure 2.

```
_text:08048420  close_last proc near
_text:08048420  close_last:
_text:08048420          mov     edx,dword [esp+fd]
                           AR_close_last_INITIAL[4:7] <: EDX_8048420_close_last[0:3]
                           close_last.in@stack0 <: AR_close_last_INITIAL[4:7]
                           EAX_804843F_close_last[0:3] <: close_last.out@eax
_text:08048424          jmp     loc_8048432
_text:08048426          db 141, 118, 0, 141, 188, 39
_text:0804842C          times 4 db 0
_text:08048430
_text:08048430  loc_8048430:
_text:08048430          mov     edx,eax
                           EAX_8048432_close_last[0:3] <: EDX_8048430_close_last[0:3]
_text:08048432
_text:08048432  loc_8048432:
_text:08048432          mov     eax,dword [edx]
                           EDX_8048420_close_last[0:3] <: unknown_loc_106
                           EDX_8048430_close_last[0:3] <: unknown_loc_106
                           unknown_loc_106.load.32@0 <: EAX_8048432_close_last[0:3]
_text:08048434          test    eax,eax
_text:08048436          jnz     loc_8048430
_text:08048438          mov     eax,dword [edx+4]
                           EDX_8048420_close_last[0:3] <: unknown_loc_111
                           EDX_8048430_close_last[0:3] <: unknown_loc_111
                           unknown_loc_111.load.32@4 <: EAX_8048438_close_last[0:3]
_text:0804843B          mov     dword [esp+fd],eax
                           EAX_8048438_close_last[0:3] <: AR_close_last_804843B[4:7]
_text:0804843F          jmp     __thunk_.close
                           AR_close_last_804843B[4:7] <: close:0x804843F.in@stack0
                           close:0x804843F.in@stack0 <: #FileDescriptor
                           close:0x804843F.in@stack0 <: int
                           close:0x804843F.out@eax <: EAX_804843F_close_last[0:3]
                           int <: close:0x804843F.out@eax
_text:08048443
_text:08048443  close_last endp
```

Figure 16. The constraints obtained by abstract interpretation of the example code in Figure 2.