An expressive type system for binary code

Carlo Angiuli Matthew Maurer

May 2, 2012

1 Introduction

In this report, we describe an expressive type system for the SSA version of the BAP Intermediate Language (BIL). BAP [2] is a binary analysis platform in development by Prof. David Brumley's research group, which lifts program binaries into an intermediate language by elaborating instructions into their precise semantics. BAP, written in OCaml, currently includes limited type inference capabilities [4] for a simple type system with a hierarchy of sized integer, pointer, and record types.

Type reconstruction on binary code is used to recover high-level structure (e.g., for reverse engineering) and statically verify basic safety properties of a binary (e.g., only valid pointers are dereferenced), especially when a program's source code is not present. There are a number of techniques for binary type reconstruction, including observation of dynamic memory access patterns [6], and value-set analysis [1], a static dataflow analysis.

However, these projects all reconstruct very simple types. In contrast, the Typed Assembly Language project [5] is a family of very expressive type systems for realistic assembly languages, including stack-based reasoning, polymorphism, and recursive types. These type systems permit high-level reasoning, such as reachability of heap objects for garbage collection, type safety, and validity of stack accesses [3].

We have developed an expressive type theory for BIL modeled after TAL, along with a type-checker to verify that a user's suggested function signatures are valid. These expressive signatures allow us to check precise constraints at function call sites, letting us type even more functions. Our theory includes features like ABI compliance, recursive types, bounded polymorphism, and non-null pointers.

This theory is geared toward binaries generated by compilers; there are many valid functions which do not typecheck under our system. Conversely, however, we believe our system to be sound, i.e., to not typecheck any functions which are unsafe. Our theory specifically assumes x86 calling conventions, but it would not be difficult to support additional architectures.

Figure 1 contains the formal grammar of the BIL language considered in this paper. This is a very slightly simplified version of the grammar specified in `ocaml/ssa.ml` and `ocaml/type.ml`, omitting variable endianness. The remainder of this paper uses the concrete syntax defined in Figure 2, which presents a slight restriction of the grammar presented in Figure 1.

Each statement in a program has a unique label ℓ. The set of labels is equipped with a $+1$ operation which, given the label of a statement, returns the label of the next one in sequence. Some expressions are annotated with the bit-width w of their operand. In Figures 1 and 2, x ranges over a set of *variables* which can be compared for equality. We guarantee that a program is in SSA form, i.e., any variable is assigned to (by **Move**) at most once.

$$
\begin{array}{lcl}
\textit{Prgm} & ::= & [\textit{Stmt}] \\
\textit{Value} & ::= & \textbf{Int}\ n\ w \\
& | & \textbf{Var}\ x \\
& | & \textbf{Lab}\ \ell \\
\textit{Exp} & ::= & \textbf{Load}\ \textit{Value}\ \textit{Value}\ w \\
& | & \textbf{Store}\ \textit{Value}\ \textit{Value}\ \textit{Value}\ w \\
& | & \textbf{Ite}\ \textit{Value}\ \textit{Value}\ \textit{Value} \\
& | & \textbf{Extract}\ n\ n\ \textit{Value} \\
& | & \textbf{Concat}\ \textit{Value}\ \textit{Value} \\
& | & \textbf{BinOp}\ \textit{binop}\ \textit{Value}\ \textit{Value} \\
& | & \textbf{UnOp}\ \textit{unop}\ \textit{Value} \\
& | & \textbf{Val}\ \textit{Value} \\
& | & \textbf{Phi}\ [x] \\
\textit{Stmt} & ::= & \textbf{Move}\ x\ \textit{Exp} \\
& | & \textbf{Call}\ x\ \ell\ [x]\ \textit{Value}\ x\ \textit{Value}\ [\textit{Value}] \\
& | & \textbf{Jmp}\ \textit{Value} \\
& | & \textbf{CJmp}\ \textit{Value}\ \textit{Value}\ \textit{Value} \\
& | & \textbf{Label}\ \ell \\
\textit{binop} & ::= & \textsc{plus} \mid \textsc{minus} \mid \textsc{times} \mid \textsc{divide} \mid \textsc{sdivide} \mid \textsc{mod} \mid \textsc{smod} \mid \\
& & \textsc{lshift} \mid \textsc{rshift} \mid \textsc{arshift} \mid \textsc{and} \mid \textsc{or} \mid \textsc{xor} \mid \\
& & \textsc{eq} \mid \textsc{neq} \mid \textsc{lt} \mid \textsc{le} \mid \textsc{slt} \mid \textsc{sle} \\
\textit{unop} & ::= & \textsc{neg} \mid \textsc{not}
\end{array}
$$

Figure 1: SSA BIL formal grammar.

$$
\begin{array}{lcl}
p & ::= & [\ell \times s] \\
v & ::= & n_w \mid x \mid \ell \\
e & ::= & m[v] \\
& | & m[v \mapsto v'] \\
& | & v\ ?\ v' : v'' \\
& | & v_{(n,m)} \\
& | & v \cdot v' \\
& | & v\ \mathsf{binop}\ v' \\
& | & \mathsf{unop}\ v \\
& | & v \\
& | & \phi(v, \ldots) \\
s & ::= & x := e \\
& | & x, m' := \ell(m, v)(v', \ldots) \\
& | & \mathsf{jmp}\ \ell \\
& | & \mathsf{jmp}\ (\ell\ ?\ \ell' : \ell'')
\end{array}
$$

Figure 2: BIL concrete syntax.

2 Expression types

The grammar of our expression type system is specified in Figure 3. Our judgments take the form $\Xi\ ;\ \Delta\ ;\ \Gamma \vdash \mathcal{J}$, where Ξ represents a context of equational constraints, Δ a context of type variables, and Γ a variable typing context.

Several types merit additional explanation. $\mathbf{1}$ is the unit of $\times$; $\cdots$ is the type of infinitely-expandable stack space. $\mathsf{initptr}_{m,\alpha}\ \tau$ is the type of a pointer which is in the process of being initialized, and whose type can thus change; $\mathsf{ptr}_m\ \tau$ is the type of a concrete pointer, whose contents have fixed type. junk_τ is the type of an $\mathsf{initptr}$ field which can be initialized to any subtype of τ. $\mathsf{nullable}\ \tau$ is either zero or an element of τ. $\forall \tau_1 \trianglelefteq A \trianglelefteq \tau_2 . \tau_A$ is a bounded polymorphic type, and $\mu A . \tau_A$ is a recursive type.

$$\begin{aligned}\tau ::= \ & A \mid \mathsf{int}_w \mid \mathsf{mem} \mid \mathsf{zero} \mid \mathsf{junk}_\tau \mid \top_w \mid \bot_w \mid \tau \times \tau' \mid \mathbf{1} \mid \cdots \mid \\ & \mathsf{ptr}_m\ \tau \mid \mathsf{initptr}_{m,\alpha}\ \tau \mid \mathsf{nullable}\ \tau \mid \\ & \mu A . \tau_A \mid \forall \tau_1 \trianglelefteq A \trianglelefteq \tau_2 . \tau_A \mid \tau \to \tau'\end{aligned}$$

Figure 3: Grammar of types.

Ξ is a list of equations. The auxiliary judgment $\Xi \Rightarrow e = e'$ indicates that $e = e'$ is derivable from the constraints in Ξ along with general arithmetic reasoning. (The rules we specify are meant to be taken modulo this equational reasoning.) Δ contains a list of type variables free in the judgment, along with the range of types they are valid over, e.g., $\tau_1 \trianglelefteq A \trianglelefteq \tau_2$. Γ is an ordinary typing context which specifies, e.g., that variable x has type τ in the current context, denoted $x : \tau$.

The judgment $x : \tau$ means that the variable x can be treated as an element of type τ, and $\tau \trianglelefteq \tau'$ means that τ is a subtype of τ'. In an abuse of notation, we combine these in a judgment $x \trianglelefteq \tau$ (resp., $x \trianglerighteq \tau$) which means that the type of x is guaranteed to be a subtype of τ, i.e.,

$$x \trianglelefteq \tau \iff x : \tau' \text{ and } \tau' \trianglelefteq \tau.$$

2.1 Type sizes

Although our type system is mostly meant to capture high-level information about expressions, it also captures the size, in bytes, of each expression. We define a sizeof operation on (most) types, which maps each type to the number of bytes used by the representation of any expression of that type. Henceforth we denote by W our architecture's word size, which is 4 for the purposes of our examples.

$$\frac{}{\mathsf{sizeof}(\mathsf{int}_w) = w} \qquad \frac{}{\mathsf{sizeof}(\top_w) = w} \qquad \frac{}{\mathsf{sizeof}(\bot_w) = w} \qquad \frac{}{\mathsf{sizeof}(\mathbf{1}) = 0}$$

$$\frac{}{\mathsf{sizeof}(\mathsf{ptr}_m\ \tau) = W} \qquad \frac{}{\mathsf{sizeof}(\mathsf{initptr}_{m,\alpha}\ \tau) = W} \qquad \frac{}{\mathsf{sizeof}(\mathsf{zero}) = W}$$

$$\frac{}{\mathsf{sizeof}(\mathsf{nullable}\ \tau) = \mathsf{sizeof}(\tau)} \qquad \frac{}{\mathsf{sizeof}(\mathsf{junk}_\tau) = \mathsf{sizeof}(\tau)}$$

$$\frac{}{\mathsf{sizeof}(\tau \times \tau') = \mathsf{sizeof}(\tau) + \mathsf{sizeof}(\tau')}$$

$$\frac{}{\mathsf{sizeof}(\mu A.\tau_A) = \mathsf{sizeof}(\tau_A[(\mu A.\tau_A)/A])}$$

$$\frac{\mathsf{sizeof}(\tau_1) = \mathsf{sizeof}(\tau_2)}{\mathsf{sizeof}(\forall \tau_1 \trianglelefteq A \trianglelefteq \tau_2.\tau_A) = \mathsf{sizeof}(\tau_A[\tau_1/A])}$$

Note that sizeof is not defined on mem, $\cdots$, or $\tau \to \tau'$ as elements of these types do not actually exist on the machine.[1] In our type system, a recursive type is sensible only when it has a well-defined size after being unrolled once; a prototypical example is the type of linked list pointers, $\mu A.\mathsf{nullable}\ (\mathsf{ptr}_m\ (\tau \times A))$, which has size W. Similarly, a quantified type is only sensible if the type variable has a fixed size. (The subtyping rules, which follow, ensure that $\mathsf{sizeof}(\tau) = \mathsf{sizeof}(\tau')$ for $\tau \trianglelefteq \tau'$.)

2.2 Subtyping rules

Most of our subtyping rules are straightforward. $\bot_w$ and $\top_w$ are a subtype and supertype, respectively, of all w-sized types, except for junk_τ. (We will show later why this restriction is needed.) The zero type is a subtype of int_W and $\mathsf{nullable}\ \tau$ for any τ of size W. In a departure from the TIE system, $\mathsf{int}_w \not\trianglelefteq \mathsf{int}_{w'}$ for any $w \neq w'$.

$$\frac{}{\tau \trianglelefteq \tau} \qquad \frac{\tau \trianglelefteq \tau' \quad \sigma \trianglelefteq \sigma'}{\tau \times \sigma \trianglelefteq \tau' \times \sigma'} \qquad \frac{}{\cdots \times \tau \trianglelefteq \tau}$$

$$\frac{}{\mathsf{ptr}_m\ \tau \times \tau' \trianglelefteq \mathsf{ptr}_m\ \tau} \qquad \frac{}{\mathsf{initptr}_{m,\alpha}\ \tau \trianglelefteq \mathsf{ptr}_m\ \tau} \qquad \frac{}{\tau \trianglelefteq \mathsf{nullable}\ \tau}$$

$$\frac{\tau \trianglelefteq \tau'}{\mathsf{ptr}_m\ \tau \trianglelefteq \mathsf{ptr}_m\ \tau'} \qquad \frac{\tau \trianglelefteq \tau'}{\mathsf{initptr}_{m,\alpha}\ \tau \trianglelefteq \mathsf{initptr}_{m,\alpha}\ \tau'} \qquad \frac{\tau \trianglelefteq \tau'}{\mathsf{nullable}\ \tau \trianglelefteq \mathsf{nullable}\ \tau'}$$

$$\frac{\tau \trianglelefteq \tau'}{\mathsf{junk}_\tau \trianglelefteq \mathsf{junk}'_\tau} \qquad \frac{}{\tau \trianglelefteq \mathbf{1} \times \tau} \qquad \frac{}{\mathbf{1} \times \tau \trianglelefteq \tau} \qquad \frac{}{\mathsf{junk}_{\top_m} \times \mathsf{junk}_{\top_n} \trianglelefteq \mathsf{junk}_{\top_{m+n}}}$$

$$\frac{\mathsf{sizeof}(\tau) = w \quad \tau \neq \mathsf{junk}_\sigma}{\bot_w \trianglelefteq \tau} \qquad \frac{\mathsf{sizeof}(\tau) = w \quad \tau \neq \mathsf{junk}_\sigma}{\tau \trianglelefteq \top_w}$$

$$\frac{\tau_A[(\mu A.\tau_A)/A] \trianglelefteq \tau}{\mu A.\tau_A \trianglelefteq \tau} \qquad \frac{\tau \trianglelefteq \tau_A[(\mu A.\tau_A)/A]}{\tau \trianglelefteq \mu A.\tau_A}$$

$$\frac{}{\mathsf{zero} \trianglelefteq \mathsf{int}_W} \qquad \frac{\mathsf{sizeof}(\tau) = W}{\mathsf{zero} \trianglelefteq \mathsf{nullable}\ \tau}$$

To define subtyping on quantified types, we eliminate the quantifiers by introducing type variables. (The previous rules take place in arbitrary fixed contexts.)

[1] While functions do exist on the machine, they are represented as code pointers, not as the abstract functions which we type $\tau \to \tau'$.

$$\frac{}{\Xi\,;\,\Delta, \tau_1 \trianglelefteq A \trianglelefteq \tau_2\,;\,\Gamma \vdash A \trianglelefteq \tau_2} \qquad \frac{}{\Xi\,;\,\Delta, \tau_1 \trianglelefteq A \trianglelefteq \tau_2\,;\,\Gamma \vdash \tau_1 \trianglelefteq A}$$

$$\frac{\Xi\,;\,\Delta, \tau_1 \trianglelefteq A \trianglelefteq \tau_2\,;\,\Gamma \vdash \tau_A \trianglelefteq \sigma}{\Xi\,;\,\Delta\,;\,\Gamma \vdash (\forall \tau_1 \trianglelefteq A \trianglelefteq \tau_2 . \tau_A) \trianglelefteq \sigma} \qquad \frac{\Xi\,;\,\Delta, \tau_1 \trianglelefteq A \trianglelefteq \tau_2\,;\,\Gamma \vdash \sigma \trianglelefteq \tau_A}{\Xi\,;\,\Delta\,;\,\Gamma \vdash \sigma \trianglelefteq (\forall \tau_1 \trianglelefteq A \trianglelefteq \tau_2 . \tau_A)}$$

For the purposes of comparing partially-initialized $\mathsf{initptr}$s, we also define a binary operation $\vee$ which returns a type compatible with both given types.

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash \tau' \trianglelefteq \tau \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau'' \trianglelefteq \tau}{\Xi\,;\,\Delta\,;\,\Gamma \vdash \tau' \vee \tau'' = \tau}\ \vee$$

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau' \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau''}{\Xi\,;\,\Delta\,;\,\Gamma \vdash \mathsf{junk}_{\tau'} \vee \tau'' = \mathsf{junk}_{\tau}}\ \vee_{\mathsf{junk}}$$

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau' \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau''}{\Xi\,;\,\Delta\,;\,\Gamma \vdash \mathsf{junk}_{\tau'} \vee \mathsf{junk}_{\tau''} = \mathsf{junk}_{\tau}}\ \vee_{\mathsf{junk,junk}}$$

2.3 Expressions

We derive for each expression a *largest* type; the conclusion $e \trianglerighteq \tau$ means that it is valid to give e any supertype of τ. In particular, we may introduce a new type variable A with $\tau' \trianglelefteq A \trianglelefteq \tau''$ for any $\tau \trianglelefteq \tau', \tau''$. Similarly, the premise $e \trianglelefteq \tau$ means that e must have a subtype of τ as its type. In practice, when we may conclude $e \trianglerighteq \tau$, we instead conclude $e : A$ for $\tau \trianglelefteq A \trianglelefteq \top_{\mathsf{sizeof}(\tau)}$, and dynamically shrink the upper bound as later constraints are discovered (e.g., by premises $e \trianglelefteq \tau'$).

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash x \trianglerighteq \tau \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau' \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau''}{\Xi\,;\,\Delta, \tau' \trianglelefteq A \trianglelefteq \tau''\,;\,\Gamma \vdash x : A}\ \forall$$

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash x : \tau \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash \tau \trianglelefteq \tau'}{\Xi\,;\,\Delta\,;\,\Gamma \vdash x : \tau'}\ \trianglelefteq$$

$$\frac{x : \tau \in \Gamma}{\Xi\,;\,\Delta\,;\,\Gamma \vdash x : \tau}\ \textbf{Var}$$

$$\frac{}{\Xi\,;\,\Delta\,;\,\Gamma \vdash n_w \trianglerighteq \mathsf{int}_w}\ \textbf{Int}$$

$$\frac{}{\Xi\,;\,\Delta\,;\,\Gamma \vdash 0_W \trianglerighteq \mathsf{zero}}\ \textbf{Int}_0$$

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash m : \mathsf{mem} \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau}{\Xi\,;\,\Delta\,;\,\Gamma \vdash m[x] \trianglerighteq \tau}\ \textbf{Load}$$

$$\frac{\Xi\,;\,\Delta\,;\,\Gamma \vdash m : \mathsf{mem} \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \quad \Xi\,;\,\Delta\,;\,\Gamma \vdash y \trianglelefteq \tau}{\Xi\,;\,\Delta\,;\,\Gamma \vdash m[x \mapsto y] : \mathsf{mem}}\ \textbf{Store}$$

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_{1/8} \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash z \trianglelefteq \tau}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x\ ?\ y : z \trianglerighteq \tau}\ \textbf{Ite}$$

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_w \quad 0 \le n \le m \le 8w}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x_{(n,m)} \trianglerighteq \mathsf{int}_{(m-n+1)/8}}\ \textbf{Extract}$$

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_w \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_{w'}}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \cdot y \trianglerighteq \mathsf{int}_{w+w'}}\ \textbf{Concat}$$

The unop rules are straightforward.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_w}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash -x \trianglerighteq \mathsf{int}_w}\ \textsc{neg} \qquad \frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_{1/8}}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash \neg x \trianglerighteq \mathsf{int}_{1/8}}\ \textsc{not}$$

The binop rules can be grouped into classes of similar operators. The comparison operators EQ, NEQ, LT, LE, SLT, and SLE all behave as follows:

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_w \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_w}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x = y \trianglerighteq \mathsf{int}_{1/8}}\ \textsc{eq}$$

For EQ and NEQ we additionally allow pointer operands.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{ptr}_m\ \tau'}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x = y \trianglerighteq \mathsf{int}_{1/8}}\ \textsc{eq}'$$

The bitwise operators LSHIFT, RSHIFT, ARSHIFT, AND, OR, and XOR behave as follows:

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_w \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_w}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x\ \&\ y \trianglerighteq \mathsf{int}_w}\ \textsc{and}$$

The arithmetic operators PLUS, MINUS, TIMES, DIVIDE, SDIVIDE, MOD, and SMOD follow

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{int}_{w_x} \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_{w_y}}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + y \trianglerighteq \mathsf{int}_w}\ \textsc{plus}$$

where w, w_x, w_y depend on the operation, e.g., for PLUS and MINUS, $w = w_x = w_y$, and for TIMES, $w = w_x + w_y$.

However, in machine code, PLUS and MINUS also serve the very important role of allowing pointer arithmetic, which requires an additional set of rules.[2] First, we define rules allowing word-sized integers to be added and subtracted from pointers.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_W}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + y \trianglerighteq \mathsf{ptr}_m\ \tau'}\ \textsc{plus}'$$

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_W}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x - y \trianglerighteq \mathsf{ptr}_m\ \tau'}\ \textsc{minus}'$$

[2]Because compilers typically generate only positive offsets for non-stack pointers, we do not include rules for negative indexing into struct pointers. These rules could be added to the system without much effort.

Known nonnegative offsets into record types can be resolved.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \times \tau'}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + \mathsf{sizeof}(\tau) \trianglerighteq \mathsf{ptr}_m\ \tau'} \qquad \frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{zero}}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + y \trianglerighteq \mathsf{ptr}_m\ \tau}$$

We also resolve known nonnegative offsets into stack pointers.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \cdots \times \tau \times \tau'}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + \mathsf{sizeof}(\tau) \trianglerighteq \mathsf{ptr}_m\ \tau'} \qquad \frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{ptr}_m\ \cdots \times \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{zero}}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x + y \trianglerighteq \mathsf{ptr}_m\ \tau}$$

Stack pointers are unusual in that they can be accessed with negative offsets to automatically allocate new space.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{initptr}_{m,\alpha}\ \cdots \times \tau}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x - n_W \trianglerighteq \mathsf{initptr}_{m,\alpha}\ \cdots \times \mathsf{junk}_{\top_n} \times \tau}$$

It remains only to type ϕ nodes.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \tau}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash \phi(x,y) \trianglerighteq \tau}\ \phi$$

Unlike ordinary ptrs, initptrs are allowed to change type as they are initialized; hence it is important to allow ϕ nodes between initptrs of different types which originate from the same malloc.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x : \mathsf{initptr}_{m,\alpha}\ \tau \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y : \mathsf{initptr}_{m,\alpha}\ \tau'}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash \phi(x,y) : \mathsf{initptr}_{m,\alpha}\ (\tau \vee \tau')}\ \phi_{\mathsf{initptr}}$$

Because ϕ nodes arise in loops, there often arises a circularity in which the type of $\phi(x,y)$ depends on the type of x, which in turn depends on the type of $\phi(x,y)$. To assign this expression a type, we must consider it in the context of other statements, so we handle this case later.

3 Statement types

We type sequences of statements using Hoare triples with partial correctness semantics. The judgment $\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\}$ indicates that, if $\{\Xi\ ;\ \Delta\ ;\ \Gamma\}$ hold before the statement at ℓ is executed, then execution starting at ℓ with these assumptions will continue until ℓ' contains the next statement to be executed, at which point $\{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\}$ will hold.

We use the auxiliary judgment $[\ell] = s$ to indicate that statement s is located at label ℓ. This judgment is implicitly parametrized over a single program; we do not allow mixing these judgments between programs.

3.1 Structural rules

As in traditional Hoare logics, compositional reasoning about statement sequences is a core principle.

$$\frac{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\} \quad \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\}\ \ell' \to \ell''\ \{\Xi''\ ;\ \Delta''\ ;\ \Gamma''\}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell''\ \{\Xi''\ ;\ \Delta''\ ;\ \Gamma''\}}\ \text{Composition}$$

It is also sound to replace the precondition with a stronger one, or the postcondition with a weaker one. Type variable contexts Δ cannot shrink, to ensure the well-formedness of the typing contexts Γ.

$$\frac{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi'', \Xi'''\ ;\ \Delta''\ ;\ \Gamma'', \Gamma'''\}}{\{\Xi, \Xi'\ ;\ \Delta, \Delta'\ ;\ \Gamma, \Gamma'\}\ \ell \to \ell'\ \{\Xi''\ ;\ \Delta''\ ;\ \Gamma''\}}\ \text{Consequence}$$

Because we consider only programs in SSA form, the Hoare system is not burdened with detailed reasoning about control flow; instead, ϕ nodes handle propagating and unifying type information when control flow joins back up.

3.2 Statements

The control flow statements **Jmp** and **CJmp** are relatively straightforward. Either branch of an unknown conditional jump can be considered, with an additional symbolic constraint. Only the correct branch of a known conditional jump can be considered.

$$\frac{[\ell] = \mathsf{jmp}\ \ell'}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi\ ;\ \Delta\ ;\ \Gamma\}}\ \mathbf{Jmp}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \Rightarrow x = 1}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi\ ;\ \Delta\ ;\ \Gamma\}}\ \mathbf{CJmp}_{\text{known}}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \Rightarrow x = 0}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell''\ \{\Xi\ ;\ \Delta\ ;\ \Gamma\}}\ \mathbf{CJmp}'_{\text{known}}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \not\Rightarrow x = 0 \quad \Xi \not\Rightarrow x = 1}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi, x = 1\ ;\ \Delta\ ;\ \Gamma\}}\ \mathbf{CJmp}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \not\Rightarrow x = 0 \quad \Xi \not\Rightarrow x = 1}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell''\ \{\Xi, x = 0\ ;\ \Delta\ ;\ \Gamma\}}\ \mathbf{CJmp}'$$

These rules also serve to handle statically resolvable indirect jumps, because our rules are taken modulo equational reasoning. For example, the following is a derivable rule for unconditional indirect jumps.

$$\frac{[\ell] = \mathsf{jmp}\ x \quad \Xi \Rightarrow x = \ell'}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell'\ \{\Xi\ ;\ \Delta\ ;\ \Gamma\}}$$

We have no rules to handle unresolved indirect jumps, as it is very difficult to reason about these.

The most common statement is **Move**, which encompasses assignments, memory reads and writes, and ϕ nodes. The assigned variable gets the type of the right hand side, and a constraint equating their values. We do not allow the assigned variable to already exist in the typing context.

$$\frac{[\ell] = x := y \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \tau \quad x \notin \Gamma}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi, x = y\ ;\ \Delta\ ;\ \Gamma, x \trianglerighteq \tau\}}\ \mathbf{Move}$$

Note that, if x already does exist in the typing context, it is still possible to derive the same triple, so long as it can be done without the x assumption. For example, if $[\ell] = x := 0_W$ and we want $\{\cdot\ ;\ \cdot\ ;\ x : \mathsf{int}_W\}\ \ell \to \ell + 1\ \{\cdot\ ;\ \cdot\ ;\ x : \mathsf{int}_W\}$, by the consequence rule it suffices to show $\{\cdot\ ;\ \cdot\ ;\ \cdot\}\ \ell \to \ell + 1\ \{\cdot\ ;\ \cdot\ ;\ x : \mathsf{int}_W\}$, which follows from the **Move** rule.

The rule above contains insufficient information in some cases. Because pointers are indexed by the memory in which they are valid, we need to track propagation of memories through stores and ϕ nodes. To do this, we introduce a $m \rightsquigarrow m'$ constraint which means that the contents of memory m have been propagated through to memory m'.

$$\frac{[\ell] = x := y \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y : \mathsf{mem}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi, y \rightsquigarrow x\ ;\ \Delta\ ;\ \Gamma, x : \mathsf{mem}\}}\ \mathbf{Move}_{\mathsf{mem}}$$

$$\frac{\Xi \Rightarrow \phi(y, z) \rightsquigarrow x}{\Xi \Rightarrow y \rightsquigarrow x}\ \rightsquigarrow_{\phi_1} \qquad \frac{\Xi \Rightarrow \phi(y, z) \rightsquigarrow x}{\Xi \Rightarrow z \rightsquigarrow x}\ \rightsquigarrow_{\phi_2} \qquad \frac{\Xi \Rightarrow y[a \mapsto b] \rightsquigarrow x}{\Xi \Rightarrow y \rightsquigarrow x}\ \rightsquigarrow_{\mapsto}$$

This constraint is used to update pointer typing information.

$$\frac{\Xi \Rightarrow m \rightsquigarrow m' \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash x : \mathsf{ptr}_m\ \tau}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x : \mathsf{ptr}_{m'}\ \tau}\ \rightsquigarrow_{\mathsf{ptr}}$$

$$\frac{\Xi \Rightarrow m \rightsquigarrow m' \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash x : \mathsf{initptr}_{m,\alpha}\ \tau}{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash x : \mathsf{initptr}_{m',\alpha}\ \tau}\ \rightsquigarrow_{\mathsf{initptr}}$$

We also have a **Move** rule which allows the types of initptr contents to change, unlike ptrs.

$$\frac{\begin{array}{l}\Xi\ ;\ \Delta\ ;\ \Gamma \vdash m : \mathsf{mem} \\ [\ell] = m' := m[x \mapsto y] \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{initptr}_{m,\alpha}\ \mathsf{junk}_\tau \times \sigma \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \tau\end{array}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi\ ;\ \Delta, \Delta^*\ ;\ \Gamma, \Gamma^*, m' : \mathsf{mem}\}}\ \mathbf{Move}_{\mathsf{initptr}}$$

In the rule above, Δ^*, Γ^* replace each occurrence of $\mathsf{initptr}_{m,\alpha}\ \rho \times \mathsf{junk}_{\tau'} \times \sigma$ for $\tau \trianglelefteq \tau'$ in Δ, Γ with $\mathsf{initptr}_{m',\alpha}\ \rho \times \tau \times \sigma$. The absence of $m \rightsquigarrow m'$ in Ξ implicitly invalidates the previous versions of the α-labeled initptrs.

One other **Move** rule remains, to allow typing ϕ nodes whose arguments' types depend on the type of the ϕ node itself.

$$\frac{[\ell] = x := \phi(xs) \quad \Xi'\ ;\ \Delta'\ ;\ \Gamma' \vdash \phi(xs) \trianglelefteq \tau \quad \{\Xi\ ;\ \Delta\ ;\ \Gamma, x \trianglelefteq \tau\}\ \ell + 1 \to \ell\ \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi\ ;\ \Delta\ ;\ \Gamma, x \trianglerighteq \tau\}}\ \mathbf{Move}_{\phi}$$

To understand this rule, consider the code snippet in Figure 4. We would like to conclude

$$\{\cdot\ ;\ \cdot\ ;\ \cdot\}\ \ell \to \ell + 3\ \{\cdot\ ;\ \cdot\ ;\ x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\},$$

i.e., that execution starting at ℓ reaches $\ell + 3$ with all three variables having type int. However, if we attempt reasoning about this code, we notice that execution starting at ℓ has no type for z, so $\phi(y, z)$ cannot be given a type.

If we hypothesized $z : \mathsf{int}$ before $\ell + 1$, then we could conclude $x : \mathsf{int}$, which would then imply our hypothesis $z : \mathsf{int}$ after $\ell + 2$, and in particular, before $\ell + 1$. In other words, if we assume what

$$\begin{array}{lcl} [\ell] & = & y := 0 \\ [\ell+1] & = & x := \phi(y, z) \\ [\ell+2] & = & z := x \\ [\ell+3] & = & \mathsf{jmp}\ \ell+1 \end{array}$$

Figure 4: A motivating example for $\mathbf{Move}_\phi$.

we would like to prove, and show it is sufficient to derive that assumption, then it is a sound type assignment.

Hence, the $\mathbf{Move}_\phi$ rule states that, if by assuming that a ϕ node at ℓ has type τ after ℓ, we can show that execution reaches ℓ where the ϕ node does have type τ, then it is safe to assume that it does indeed have type τ.

3.3 Function calls

The only remaining statement is **Call**, which performs function calls. This is a high-level statement which we added to sidestep complications arising from typing function arguments via stack slots. A function call

$$\mathsf{eax}, m''' := \ell(m, \mathsf{esp})(x, y)$$

replaces a sequence of BIL instructions

$$\begin{array}{l} m' := m[\mathsf{esp} \mapsto x] \\ \mathsf{esp}' := \mathsf{esp} + 4 \\ m'' := m'[\mathsf{esp}' \mapsto y] \\ \mathsf{jmp}\ \ell \end{array}$$

and specifies a result eax, output and input memories m''' and m respectively, input stack pointer esp to locate the arguments, and a list of arguments x, y. Internally, a **Call** also maintains a list of callee-saved (clobbered) registers for the purposes of the SSA transformation.

We only allow **Call**s to labels ascribed a function type in this type system. Having such a type indicates not only the argument and result types of the function, but that a call to the function is well-behaved and, if it returns, will return to the next instruction in the current function. We define and discuss function types in more detail in Section 4.

$$\frac{[\ell] = z, m' := \ell'(m, x)(ys) \quad \begin{array}{c} \Xi\ ;\ \Delta\ ;\ \Gamma \vdash m : \mathsf{mem} \\ \Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{initptr}_{m,\alpha}\ \cdots \times \sigma \end{array} \quad \begin{array}{c} \Xi\ ;\ \Delta\ ;\ \Gamma \vdash ys \trianglelefteq \tau^* \\ \Xi\ ;\ \Delta\ ;\ \Gamma \vdash \ell' : \tau^* \to \tau \end{array}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell+1\ \{\Xi, m \rightsquigarrow m'\ ;\ \Delta\ ;\ \Gamma, m' : \mathsf{mem}, z \trianglerighteq \tau\}}\ \mathbf{Call}$$

To type $z, m' := \ell'(m, x)(ys)$ where $\ell' : \tau^* \to \tau$, we require that m is a memory, the arguments ys form a record of the correct type τ^*, and that x is a stack pointer. (Because the argument stack writes have been omitted, we do not specify any stack slot types.) Then we conclude the result z has the correct output type τ, and memory m' inherits pointers from m.

There is a distinguished function malloc that cannot be normally typed. malloc is treated as a distinguished label determined statically by the type checker.

$$\frac{\Xi\ ;\ \Delta\ ;\ \Gamma \vdash m : \mathsf{mem}}{[\ell] = z, m' := \mathsf{malloc}(m, x)(y) \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash x \trianglelefteq \mathsf{initptr}_{m,\alpha}\ \cdots \times \sigma \quad \Xi\ ;\ \Delta\ ;\ \Gamma \vdash y \trianglelefteq \mathsf{int}_W}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi, m \rightsquigarrow m'\ ;\ \Delta\ ;\ \Gamma, m' : \mathsf{mem}, z \trianglerighteq \mathsf{nullable}\ \mathsf{initptr}_{m',\beta}\ \top_y\}}\ \mathbf{Call}_{\mathsf{malloc}}$$

In the rule above, β is a fresh logic variable which does not occur elsewhere in the contexts. malloc is the only function which creates a new initptr; because no user-defined function can have an output type of a fresh pointer mid-initialization, all other functions must output concretized ptr pointers whose types are immutable.

4 Function types

The central type to our system is the function type $\tau^* \to \tau$, which is ascribed to a label ℓ if it subscribes to the ABI such that, when it is called with an argument list of type τ^*, it either fails to terminate or eventually returns a value of type τ.

This judgment requires some machinery in addition to the expression and statement typing described in the previous sections. In particular, we need to capture the notion of *live-ins and live-outs* for a sequence of instructions, and express the guarantee that jumping to a label will either not terminate or return to the call site.

Thus far, we have avoided the abstraction of registers entirely, using SSA variables exclusively. This has proved a boon, as SSA variables cannot change type, but it precludes our speaking of "the type of the output eax" of a function. We handle this problem in the implementation by adding a phantom **Move** instruction to a distinguished live-out (resp., live-in) variable for the value of each register at the end (resp., start) of a function. This guarantees that the $\mathsf{eax}_{\mathsf{exit}}$ variable will be bound at the end, and $\mathsf{esp}_{\mathsf{entry}}$ at the start, of every function.

Although the **Call** statement suppresses function arguments being moved onto the stack in callers, functions still access their arguments in this way, so the argument types are propagated from only the input stack pointer. This stack pointer is given the type $\mathsf{initptr}_{m,\alpha}\ \cdots \times \tau^*$ to also allow functions to allocate private stack space. Our function judgment is defined by the following straightforward rule, where $\ell \to \star$, defined below, enforces the safety property described above.

$$\frac{\{\cdot\ ;\ \cdot\ ;\ \mathsf{esp}_{\mathsf{entry}} : \mathsf{initptr}_{m,\alpha}\ \cdots \times \tau^*\}\ \ell \to \ell'\ \{\Xi\ ;\ \Delta\ ;\ \Gamma, \mathsf{eax}_{\mathsf{exit}} : \tau\} \quad \{\cdot\ ;\ \cdot\ ;\ \cdot\}\ \ell \to \star}{\ell : \tau^* \to \tau}\ \to$$

4.1 Safety

We would like to say that a function between ℓ and ℓ' has type $\Pi_i \tau_i \to \tau$ so long as we can derive

$$\{\cdot\ ;\ \cdot\ ;\ y_i : \tau_i\}\ \ell \to \ell'\ \{\Xi\ ;\ \Delta\ ;\ \Gamma, \mathsf{eax}_{\mathsf{exit}} : \tau\}$$

Indeed, the burden of typing ϕ nodes in the function body ensures that such a judgment captures the fact that along *all* execution paths $\ell \to \ell'$, $\mathsf{eax}_{\mathsf{exit}} : \tau$ is derivable. Crucially, however, this judgment does *not* capture that all executions starting at ℓ are well-typed (when they cannot reach ℓ'), nor even that all paths $\ell \to \ell'$ are well-typed (when they are dead code with respect to $\mathsf{eax}_{\mathsf{exit}}$).

An example of the first deficiency can be found in Figure 5. All paths $\ell \to \ell' + 1$ are safe, but there is an execution path $(\ell, \ell + 1, \dots)$ which is unsafe. For an example of the latter deficiency,

simply remove the statement at $\ell + 2$; even though both paths of the conditional jump reach ℓ', there is no ϕ node forcing the $\ell + 1$ branch to be typechecked at all.

$$\begin{array}{lcl} [\ell] & = & \mathsf{jmp}\ (y_0\ ?\ \ell' : \ell + 1) \\ [\ell + 1] & = & \text{(ill-typed exploitable code)} \\ [\ell + 2] & = & \mathsf{jmp}\ \ell + 1 \\ [\ell'] & = & \mathsf{eax}_{\mathsf{exit}} := 0_W \end{array}$$

Figure 5: An unsafe function which satisfies $\mathsf{eax}_{\mathsf{exit}} : \mathsf{zero}$ for $\ell \to \ell' + 1$.

We solve this problem by introducing a new judgment $\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star$ which indicates that all executions starting at ℓ with $\{\Xi\ ;\ \Delta\ ;\ \Gamma\}$ are typesafe. In particular, if the statement at ℓ is a **Jmp** or **CJmp**, we require that all possible next labels have this property. Otherwise, we require that the current instruction is well-typed, and require that the next label have this property with respect to the contexts after this instruction. To correctly mark infinite loops as safe, we always add the current safety goal to the equational context, and automatically derive a safety goal already in the context.

$$\frac{\Xi \Rightarrow \ell \to \star}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe}$$

$$\frac{[\ell] = \mathsf{jmp}\ \ell' \quad \{\Xi, \ell \to \star\ ;\ \Delta\ ;\ \Gamma\}\ \ell' \to \star}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{Jmp}}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \Rightarrow x = 1 \quad \{\Xi, \ell \to \star\ ;\ \Delta\ ;\ \Gamma\}\ \ell' \to \star}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{CJmp_{known}}}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \Rightarrow x = 0 \quad \{\Xi, \ell \to \star\ ;\ \Delta\ ;\ \Gamma\}\ \ell'' \to \star}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{CJmp'_{known}}}$$

$$\frac{[\ell] = \mathsf{jmp}\ (x\ ?\ \ell' : \ell'') \quad \Xi \not\Rightarrow x = 0 \quad \Xi \not\Rightarrow x = 1 \quad \begin{array}{l}\{\Xi, \ell \to \star\ ;\ \Delta\ ;\ \Gamma\}\ \ell' \to \star \\ \{\Xi, \ell \to \star\ ;\ \Delta\ ;\ \Gamma\}\ \ell'' \to \star\end{array}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{CJmp}}$$

$$\frac{[\ell] = x := y \quad \{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\} \quad \{\Xi', \ell \to \star\ ;\ \Delta'\ ;\ \Gamma'\}\ \ell + 1 \to \star}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{Move}}$$

$$\frac{\begin{array}{c}[\ell] = z, m' := \ell'(m, x)(ys) \\ \{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \ell + 1\ \{\Xi'\ ;\ \Delta'\ ;\ \Gamma'\} \quad \{\Xi', \ell \to \star\ ;\ \Delta'\ ;\ \Gamma'\}\ \ell + 1 \to \star\end{array}}{\{\Xi\ ;\ \Delta\ ;\ \Gamma\}\ \ell \to \star}\ \mathbf{Safe_{Call}}$$

5 Example

Figure 6 contains the BIL lifting of a function which takes an integer and returns a newly-allocated linked list node populated with that integer:

```
typedef struct node {
  int value;
  struct node* next;
} node;

node *alloc(int value) {
  node *new_node = malloc(sizeof(node));
  if (!new_node)
    return null;
  new_node->value = value;
  new_node->next = null;
  return new_node;
}
```

Our system derives the type $\mathsf{int} \to \mu x.\mathsf{nullable\ ptr}_m\ (\mathsf{int} \times x)$ for this function, since a linked list node has this recursive type. We could certainly derive other types, like the supertype $\mathsf{int} \to \mathsf{nullable\ ptr}_m\ (\mathsf{int} \times \mathsf{zero})$, but this type would not let us type other functions on linked lists which use the recursive structure of the lists.

6 Implementation

We have partially implemented a typechecker for the system described above, on top of the BAP platform. While our system cannot yet type code with the full power of our type system, we can type short sequences of code for safety.

This implementation has three basic components. The first is a series of transformations to the vanilla BAP IL itself, to give us the form we need to be able to work with. In particular, BAP has no concept of calls, returns, or functions. Next, we encoded the subtyping system. While the rules are clear on paper, the actual implementation avoid the guessing required for the instantiation of some of the subtyping rules. Finally, the biggest challenge was to manage the contexts in the rules, and handle the firing of the rules.

6.1 IL Modifications

To implement our version of BIL with calls, we first extract the instruction range of interest from the target binary. We implemented an algorithm to turn code which exits through an indirect jump into a single-exit graph, by locating sequences of instructions similar to `ret`s, and ensuring that they were all jumping out via the same address. Once this was done, an exit node was added to the graph, and the returns rewritten to direct jumps for the purposes of analysis.

As BIL is often quite verbose (and uses a number of unusual operations, especially for flag setting), we run a dead code elimination pass at this point to reduce the code's complexity. We tag the live-in callee-saved registers at the beginning and end, to guarantee that effects on them are tracked; we also tag all memory stores as live, and tag the live-out `eax` as the return value.

```
entry:
R_EBX_212:u32 = R_EBX_6:u32 @set "liveout"
R_ESP_213:u32 = R_ESP_1:u32 @set "liveout"
R_EBP_214:u32 = R_EBP_0:u32 @set "liveout"
R_EDI_215:u32 = R_EDI_3:u32 @set "liveout"
R_ESI_216:u32 = R_ESI_2:u32 @set "liveout"
T_t_217:u32 = R_EBP_214:u32
R_ESP_218:u32 = R_ESP_213:u32 - 4:u32
mem_219:?u32 = mem_56:?u32 with [R_ESP_218:u32, e_little]:u32 = T_t_217:u32
R_EBP_220:u32 = R_ESP_218:u32
R_ESP_222:u32 = R_ESP_218:u32 - 0x18:u32
R_EAX_252:u32, mem_255:?u32 =
  addr 0x804dc70@(mem_219:?u32, R_ESP_222:u32)(8:u32)
T_t_256:u32 = R_EAX_252:u32 & R_EAX_252:u32
R_ZF_277:bool = 0:u32 == T_t_256:u32
cjmp R_ZF_277:bool, "cjmp2", "nocjmp2"

nocjmp2:
temp_291:u32 = R_EBP_220:u32 + 8:u32
R_EDX_292:u32 = mem_255:?u32[temp_291:u32, e_little]:u32
mem_293:?u32 =
  mem_255:?u32 with [R_EAX_252:u32, e_little]:u32 = R_EDX_292:u32
temp_294:u32 = R_EAX_252:u32 + 4:u32
mem_295:?u32 = mem_293:?u32 with [temp_294:u32, e_little]:u32 = 0:u32

cjmp2:
mem_279:?u32 = phi(mem_255:?u32, mem_295:?u32)
R_ESP_280:u32 = R_EBP_220:u32
R_EBP_281:u32 = mem_279:?u32[R_ESP_280:u32, e_little]:u32
R_ESP_282:u32 = R_ESP_280:u32 + 4:u32
R_ESP_284:u32 = R_ESP_282:u32 + 4:u32

exit:
R_EAX_285:u32 = R_EAX_252:u32 @set "liveout"
R_EBX_286:u32 = R_EBX_212:u32 @set "liveout"
R_ESP_287:u32 = R_ESP_284:u32 @set "liveout"
R_EBP_288:u32 = R_EBP_281:u32 @set "liveout"
R_EDI_289:u32 = R_EDI_215:u32 @set "liveout"
R_ESI_290:u32 = R_ESI_216:u32 @set "liveout"
```

Figure 6: `alloc` function for linked list nodes.

Next, we template call sequences, as if all calls have arity zero. We insert calls here to ensure that, for the purposes of the SSA lifting pass, fresh memories are used after the calls, and caller-save registers and the return value have been redefined after a call.

After SSA lifting occurs, we remove writes to the stack region at known offsets from the function's input esp, and pack these as function arguments. As discussed earlier, this allows us to sidestep the x86 calling conventions to guarantee our assumption that memory cells remain the same type throughout a function. (Otherwise, sequential calls would violate this invariant.)

Finally, we had to add support for the **Call** statement throughout the BAP infrastructure, to ensure that critical analysis tools and passes, like dead code elimination and control flow graph operations, produce the correct results. As a result, the **Call** statement is now fully integrated and supported by other transformations in BAP.

6.2 Subtyping

To avoid the existential search problem posed by our rules as written, we concretize the idea of subtyping into a series of lattices, one per size. We then enforce that subtyping is confined to individual lattices; our type inference engine rejects signatures which require this. Defining meet and join operations over these lattices suffices to generate all needed upper and lower bounds on type variables. When we need simply to check $\tau \trianglelefteq \tau'$, we take the join of τ, τ' and check whether it is τ'.

Additionally, in practice, the vast majority of types are represented as type variables with upper and lower lattice bounds, and are concretized only at the end. This practice helps keep our inference uniform.

6.3 Rules

We have not yet implemented every rule in the system, nor the path-dependent contexts needed for casting nullable τ to τ, but our overall structure is quite elegant. Each attempt to typecheck the program maintains three contexts:

1. Type context (assignment from variables to types)
2. Subtyping context (upper and lower bounds on type variables)
3. Path-dependent context (a copy of both other contexts, on a per-block level)

We use a worklist over all statements to attempt to show they are all safe. Because the control flow graph already encapsulates the reachability portion of the $\ell \to \star$ judgment, it suffices to show that each statement is safe.

Some rules, like $\mathbf{Move}_\phi$, have subgoals which use additional facts. In these situations, all the contexts are pushed up a level on the stack, copied, and we recursively call the checker. When complete, the new copies of the contexts can be removed with no modification to the state.

While the rules deal explicitly with pre- and post-conditions, most assertions about variable types are true everywhere, which makes most of the reasoning easier. This property is largely owed to the SSA form, and our pointer tagging. However, conditional jumps create explicitly path-dependent knowledge, in which the upper bound on some types is lowered in certain basic blocks (e.g., casing on whether a pointer is null, causing the pointer's upper bound to drop from nullable pointer to pointer).

We handle this situation with a domination relation. If a virtual node along the edge coming from a **CJmp** would dominate a block, then the information from that **CJmp** is added to its per-block context. Then, when trying to fire a rule, we first try with the normal context. If this fails, we re-attempt with the current block context, but any results must only be added to the current block context, and the block contexts of blocks dominated by the current one. In this way, we avoid the need to talk about paths and pre- and post-conditions by appealing to the dominator tree and control flow graph we already have.

Our implementation efforts are still ongoing, but things look promising. We can currently type graphs of instructions which don't force initptr concretization or require the per-block contexts; once we complete implementing the logic described in this paper, our typechecker should be quite powerful. We will then hook up an SMT solver as a general equational reasoning system rather than just using a few built-in reasoning tools; this will further aid our system, paving the way for future work with refinement types and variable-size arrays, once we obtain reliable context-sensitive information from this system.

The SMT solver will also allow us to improve our function call templating to work across basic blocks. Overall, while our implementation cannot currently type as much as we hoped, no major obstacles to continuing the work are apparent, and our implementation thus far provides quite substantial support to future implementation efforts.

7 Conclusion

We have detailed an expressive type system for binary code, in particular, for the SSA version of BIL. We have demonstrated that it is suitable for typechecking purposes, and have described our implementation strategy for a typechecker. As far as we know, this is the first example of a TAL-like type system which can be realistically checked on existing binaries, rather than being intended for correct-by-construction programs.

We intend to continue our work on this project, first by completing and improving our typechecker, then by extending the type system with additional constructs, particularly to support dynamically allocated arrays, existential types, and refinement types. We hope our system will eventually become a realistic binary static analysis tool for reconstruction of high-level type information.

Carlo and Matt designed the concept of the type system in tandem. Carlo is primarily responsible for the theoretical implementation—precise rules and judgments—while Matt is primarily responsible for the typechecker implementation in OCaml.

References

[1] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[2] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Karl Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 198–212, New York, NY, USA, 2003. ACM.

[4] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Symposium*, pages 251–268, February 2011.

[5] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *J. Funct. Program.*, 12(1):43–88, January 2002.

[6] M. Sharif, A. Lanzi, J. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94 –109, may 2009.