State of the Union: Type Inference via Craig Interpolation [⋆]

Ranjit Jhala[1] Rupak Majumdar[2] Ru-Gang Xu[2]

[1]UC San Diego [2]UC Los Angeles

Abstract. The ad-hoc use of unions to encode disjoint sum types in C programs and the inability of C's type system to check the safe use of these unions is a long standing source of subtle bugs. We present a dependent type system that rigorously captures the ad-hoc protocols that programmers use to encode disjoint sums, and introduce a novel technique for automatically inferring, via Craig Interpolation, those dependent types and thus those protocols. In addition to checking the safe use of unions, the dependent type information inferred by interpolation gives programmers looking to modify or extend legacy code a precise understanding of the conditions under which some fields may safely be accessed. We present an empirical evaluation of our technique on 350KLOC of open source C code. In 80 out of 90 predicated edges (corresponding to 1472 out of 1684 union accesses), our type system is able to infer the correct dependent types. This demonstrates that our type system captures and explicates programmers' informal reasoning about unions, without requiring manual annotation or rewriting.

1 Introduction

We present a type system and inference algorithm for statically checking the safety of downcasts in imperative programs. Our type system is motivated by the problem of checking the safety of union accesses in C programs. C programmers extensively use unions to encode disjoint sum types in an ad-hoc manner. The programmer uses the *value* of a *tag field* to determine which element of the union an instance actually corresponds to. For example, Figure 1 shows networking code that manipulates packets represented as a C structure (`packet`) which contains an union (`icmp_hun`) to represent different types of packets. The packet is interpreted as a *parameter* message (field `ih_gwaddr`) when the field `icmp_type` $= 12$, as a *redirect* message (field `ih_pptr`) when the field `icmp_type` $= 5$, and as an *unreachable* message (field `ih_pmtu`) when the field `icmp_type` $= 3$. This ad-hoc protocol determining the mapping between tag values and the union elements is informally documented in the protocol description, but not enforced by the type system. The absence of static checking for the correctness of these accesses can be a source of subtle bugs.

⋆ This research was funded in part by the grants NSF-CCF-0427202 and NSF-CCF-0546170.

The problem of checking the safety of union accesses is an instance of the more general problem of checking the safety of *downcasts* in a language with subtyping —consider each possible "completion" of a structure with the different elements of the union as subtypes of that structure, and view union accesses as downcasts to the appropriate completion. At run-time, each instance of a supertype corresponds to an instance of *one of* its immediate subtypes. To ensure safety, programmers typically associate with each subtype, a *guard predicate* over some tag fields. The predicates for the different subtypes are pairwise inconsistent. Before performing a downcast (*i.e.,* accessing the union), the programmer tests the tag fields to ensure that the corresponding subtypes' guard predicate holds, and similarly before performing an upcast (*i.e.,* constructing the union), the programmer sets the tag field to ensure the guard predicate holds.

We formalize this idiom in a type system comprising two ingredients. The first ingredient is a type hierarchy corresponding to a directed tree of types, where the nodes correspond to types, and children to immediate subtypes. The second is a *predicated refinement* of the hierarchy, where the edges of the type hierarchy tree are labeled with *edge predicates* over the fields of the structure that hold when the supertype can be safely downcast to the subtype corresponding to the target of the edge, and conversely, must be established when the subtype is upcast to the supertype. By requiring that the edge predicates for the different children of a supertype be pairwise inconsistent, we ensure that there is a single subtype of which the supertype is an instance at runtime.

Given a predicated refinement for the subtype hierarchy of the program, we can statically type check the program by verifying that at each occurrence of an upcast or downcast, the edge predicate for the cast holds. Instead of a general invariant generator, we present a simple *syntax-directed* system that is scalable, captures the idiomatic ways in which programmers test fields, and concisely specifies the set of programs that are accepted by our type system. The technique converts the programs to SSA form, and then *conjoins* the statements *dominating* each cast location to obtain a *cast predicate* that is an invariant at the cast location. Our type checking algorithm verifies that at each cast location the edge predicate corresponding to the cast holds by using a decision procedure to check that the cast predicate *implies* the edge predicate.

We eliminate the burden of explicitly providing the predicated type refinement by devising a technique to infer types via interpolation. Our inference algorithm generates a system of *predicate constraints* with variables representing the unknown edge predicates. These constraints force the solutions for the variables to have the following key properties: (1) they are over the fields of the structure, (2) the edge predicates for the subtypes are pairwise inconsistent, and, (3) the edge predicates hold at each cast point, *i.e.,* at each (up- or down-) cast point, the cast predicate implies the edge predicate. We use *pairwise Craig interpolation*, a variant of Craig interpolation [3, 16], to solve these constraints. We show that a predicated refinement exists if for each type, the cast predicates for its subtypes are pairwise inconsistent. Thus, to solve the predicate constraints and

```
struct packet{
  u_char  icmp_type;
  u_char  icmp_code;
  u_short icmp_cksum;
  union {
    int ih_gwaddr;
    short ih_pptr;
    short ih_pmtu
  }  icmp_hun; };
int type, dest, code;
struct packet icp;
. . .
type = icp.icmp_type;
if (type == 5) {
  icp.icmp_hun.ih_gwaddr = dest;
}
else {
  if (type == 12) {
    icp.icmp_hun.ih_pptr = 0;
    code = 0;
 } else if (type == 3) {
    icp.icmp_hun.ih_pmtu = 0; } }
```

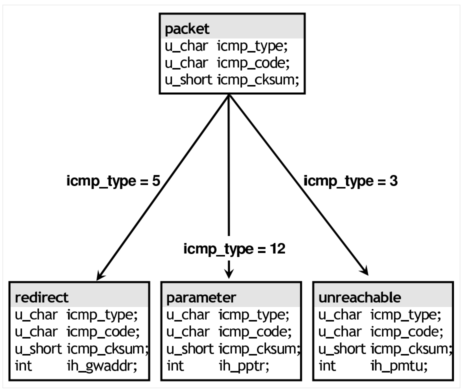


Fig. 1. (a) ICMP Example (b) (Union) Subtype Hierarchy and its Predicated Refinement

infer the predicated refinement, we compute the edge predicates for the subtypes of each type as pairwise interpolants of the corresponding cast predicates.

We have implemented the predicated subtype inference algorithm for C, and used it to infer the edge predicates for subtype hierarchies obtained from unions, for a variety of open source C programs totaling 350K lines of code. We empirically show that our inference algorithm is effective. In 80 out of 90 predicated edges (corresponding to 1472 out of 1684 union access points), our algorithm finds the correct predicate guards (which we then manually verified).

2 Language and Type System

We formalize our approach with a core imperative language with simple types. We first describe the language, then define our type system. Our core language capture C programs such as Figure 1(a). In the converted program, union fields are accessed after casting the lvalue down to the subtype containing the field. Thus, the problem of checking the correct use of unions is reduced to that of checking the safety of downcasts.

2.1 Syntax and Semantics

Types. Figure 2(b) shows the types in our language. The set of types include base types `bool` and `int`, and structure types where each structure is defined by a list of fields that are pairs of a label l and a type t. We write `void` as an abbreviation for the type `s`$\{\}$. The set of types is equipped with a partial order: we say $t' \preceq t$, or t' is a *subtype* of t, if both t, t' are structures and fields of t are a prefix of the fields of t'. Note that every structure type is a subtype of `void`.

Syntax. Figure 2(a) shows the grammar for expressions and statements in our imperative language. An *lvalue* lv is either an integer, structure or a field access, together with an explicit type cast. The `new`(t) statement is used to allocate a structure of type t. For ease of exposition, in our language every lvalue lv includes a type-cast (t) which specifies how lv is interpreted. This captures explicit upcasts, downcasts and the trivial cast to the statically declared type of lv. Arithmetic expressions are constructed from constants and integer lvalues using arithmetic operations. Boolean expressions comprise arithmetic comparisons. Statements are `skip` (or no-op), assignments, sequential composition, conditionals, and while loops. A program P is a tuple (T, Γ_0, s) where T is a set of types, Γ_0 is a map from the program lvalues to their declared types, and s is a statement corresponding to the body of the program. While we present the intraprocedural, pointer-free case, our implementation, described in Section 5, handles both procedures and pointers.

Static Single Assignment Form. For convenience in describing the type checking and type inference rules, we shall assume that the programs are converted to static single assignment (SSA) form [4], where each variable in the program is defined exactly once. Programs in SSA form have special Φ-assignment operations of the form $lv := \Phi(lv_1, \ldots, lv_\ell)$ that capture the effect of control flow joins. A Φ-assignment $lv := \Phi(lv_1, \ldots, lv_n)$ for lvalues lv, $lv_1, \ldots, lv_n$ at a node `n` implies: (1) `n` has exactly n predecessors in the control flow graph, (2) if control arrives at `n` from its jth predecessor, then lv has the value lv_j at the beginning of `n`. Formally, we extend the syntax with Φ-assignments:

$$\text{Statements } s ::= \ldots \mid lv := \Phi(lv_1, \ldots, lv_n)$$

We assume that the program has first been transformed into SSA form. We describe type checking and inference on programs in this form.

Semantics. We define the operational semantics of the language using a store and a memory in the standard way but additionally taking into account the runtime type information [15]. We assume a *store* Σ mapping variables to values, a partial mapping *memory* M from addresses to values, and a partial mapping *runtime type information* (RTTI) W from variables and addresses to types. When a structure is created during execution using the `new`(t) operation, it is tagged with the (leaf) type t that remains with it during the remainder of the execution. This value can be cast up or down along the path from the leaf t to the root type `void`, and any attempt to downcast it to a type not along this path leads the program into a "stuck" state. The (small step) operational semantics is defined using a relation $(\Sigma, M, W; s) \rightarrow (\Sigma', M', W'; s')$. The rules take into account the RTTI W, and execution gets "stuck" if a bad cast is made (i.e., an lvalue is cast to a type incompatible with its RTTI). We write $\rightarrow^*$ for the reflexive transitive closure of $\rightarrow$. For store Σ, memory M, RTTI W, and statement s, we say $(\Sigma, M, W; s)$ *diverges* if there is an infinite sequence $(\Sigma, M, W; s) \rightarrow (\Sigma_1, M_1, W_1; s_1) \rightarrow \ldots$. We say $(\Sigma, M, W; s)$ is *stuck* if (1) s is not `skip`, and (2) there is no $(\Sigma', M', W'; s')$ such that $(\Sigma, M, W; s) \rightarrow (\Sigma', M', W'; s')$.

Lvalues lv	$::=$	$(t)lv.l \mid (t)v$
Expressions e	$::=$	$n \mid \texttt{new}(t) \mid lv \mid e_1 \oplus e_2$
Boolean p	$::=$	$e_1 \sim e_2$
Statements s	$::=$	$\texttt{skip} \mid lv := e \mid s_1; s_2$
		$\mid \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2$
		$\mid \texttt{ while } p \texttt{ do } s_1$

Types t	$::=$	$\texttt{int} \mid \texttt{bool} \mid s\{m_1, ..., m_k\}i$
Fields m	$::=$	(l, t)
Declarations	$::=$	$t\ v$

Fig. 2. Syntax and Types. (a) Expressions and Statements (b) Types and declarations. n is an integer constant, v a variable, l a string label, $\sim \in \{<, >, \leq, \geq, =, \neq\}$.

2.2 Predicated Refinements of Subtype Hierarchies

Programs in our language are type checked by the standard typing rules dealing with booleans, integers and structures. However, we also want to show that each runtime downcast executes safely. To do so, we shall assume we are given a *predicated refinement* of the subtype hierarchy of the program.

Subtype Hierarchy. A *Subtype Hierarchy* is a forest (T, E) where *nodes* correspond to a set of types T, and *edges* $E \subseteq T \times T$ are such that $(t, t') \in E$ if t' is the immediate subtype of t, *i.e.,* $t' \preceq t$ and there is no t'' such that $t' \preceq t''$ and $t'' \preceq t$. Consider the structure definitions of the program of Figure 1(a). We can "unroll" the union definitions to obtain three subtypes of the type `packet`, namely `redirect`, `parameter` and `unreachable`, which correspond, respectively to instances of `packet` where the union field is actually a `ih_gwaddr`, `ih_pptr` or `ih_pmtu`. Thus, as shown in Figure 1(b), each of the subtypes is a structure containing all the fields of the supertype `packet` together with the extra field from the union. In this setting, $t' \preceq t$ if the fields of t form a prefix of the fields of t'.

Therefore, we reduce the problem of checking the safety of union accesses to checking the safety of *downcasts* in our system by converting each union access into a downcast to the subtype containing the particular union field being accessed, followed by a standard field access on the subtype. Next, we see how to *refine* the subtype hierarchy to enable the static checking of the safety of downcasts and thus, union accesses.

Predicated Refinement and Tags. We say that (T, E, ϕ) is a *predicated (refinement of the) subtype hierarchy* (T, E) if ϕ is a map from the edges E to first-order *edge predicates* such that:

R1 For each edge $(t, t') \in E$, the edge predicate $\phi(t, t')$ has one free variable `this` that refers to a structure of type t, *i.e.,* all lvalues are field accesses of a structure of type t.

R2 For each node $t \in T$, for each pair of its children t', t'' the predicates $\phi(t, t')$ and $\phi(t, t'')$ are inconsistent, *i.e.,* $\phi(t, t') \wedge \phi(t, t'')$ is *unsatisfiable.*

The tag fields of a type t are the fields that occur in the edge predicates for any edge in the subtree rooted at t in the subtype hierarchy. Formally, the *tag fields* of $t \in T$ are defined as $\texttt{tag}(t, \phi) \equiv \{l \mid \exists t' \preceq t\ : \mathit{this}.l \text{ occurs in } \phi(\cdot, t')\}$.

We use the predicate refinement to statically check the safety of downcasts and thus, union accesses. A predicated refinement captures the intuition that the programmer performs a downcast from t to t' only when a certain "tag" condition on the fields of t is met, and this tag condition is disjoint from the conditions under which downcasts are made from t to subtypes other than t'. Our type system checks that the first time a leaf type structure is upcast, the edge predicate for the structure holds, and that subsequently, the fields occurring in the edge predicate are not modified. As this is done for all structures, and the edge predicates for different downcasts are disjoint, we can statically deduce that if the edge predicate for that subtype holds at the downcast point, the downcast is safe. In Figure 1(b) each edge of the subtype hierarchy is labeled with its edge predicate. For example, `ih_gwaddr` field can be safely accessed only after the `packet` structure has been downcast to a `redirect` struct, which is permissible only when the `icmp_type` field equals 5.

3 Type Checking

Informally, a program is type safe when all structures of type t (created by $new(t)$) are accessed only as the type t or a supertype of t. Given a predicated subtype hierarchy (T, E, ϕ), a program is type safe if the hierarchy meets requirements **R1**, **R2**, and at each point in the program where an expression `e` of type t is cast to the type t', we have: (1) either t' is a supertype of t, *i.e.,* we have an upcast, or (2) t' is a subtype of t, *i.e.,* we have a downcast. In either case, the predicate obtained by substituting `this` with the variable `e` in the edge predicate $\phi(t, t')$ holds at that point. Thus, to type check the program, the edge predicates must satisfy:

R3 The edge predicate $\phi(t, t')$ with `this` substituted with `e` must hold at each program location where an expression `e` is downcast from a type t to a subtype t', or upcast from t' to t.
R4 The tag fields of a structure are not modified.

Our type checking algorithm proceeds in three steps. First, we use standard type checking to verify that each field access is to a field in the type of the expression, and that each cast conforms to the subtype hierarchy, *i.e.,* is either an upcast to a supertype or a downcast to a subtype. Second, we use a decision procedure to check that the edge predicates satisfy requirements **R1**, **R2**. Third, we perform a flow sensitive analysis to check that the edge predicates hold at each upcast or downcast. We now describe the last step in detail.

Judgments. A *judgment* in the type system for a statement s is of the form $\Gamma, \phi, I \vdash s \triangleright I'$. The judgment states: using the edge predicate map ϕ from the predicated subtype hierarchy (T, E, ϕ), we can deduce that if the program begins execution from a state satisfying the type environment Γ and the precondition I, the execution of a statement s proceeds without getting stuck (cast errors) and results in a state satisfying postcondition I'.

Our syntax-directed *derivation rules* for inferring type judgments are shown in Figure 3. At each cast point, the rules check, using a decision procedure, that the invariants imply the corresponding edge predicate. A typing rule transforms the invariant by adding the effect of the current statement on the invariant. Since our program is in SSA form, we have an invariant by taking the conjunction of the predicate representing the current statement with the previous derived invariant [12, 6]. Assignments are represented by equality as shown in rules Var-Assign and Field-Assign of Figure 3). The latter rule also stipulates that tag fields should not be assigned to, once the structure has been upcast. The rule permits the usual C idiom of appropriately "initializing" the structure by setting the data and tag fields *before* casting up to the supertype, as the tag field only appears on the parent edge of the (base) subtype, and not in the edges of the subtree rooted at the subtype. Conditionals on some predicate p are represented by p on `then` branch and $\neg p$ on the else branch (rule If in Figure 3).

Example 1. In Figure 1, consider the implicit cast (at the union access) from `packet` (t) to the redirect message (t') at line `05`. The statement 05 is dominated by the `then` branch at 04 and the assignment 03, and so the invariant at 05 is:

$$(\texttt{icp.icmp\_type} = \texttt{type}) \wedge (\texttt{type} = 5) \tag{1}$$

Similarly, the invariants at 09 and 11 are respectively:

$$(\texttt{icp.icmp\_type} = \texttt{type}) \wedge (\texttt{type} \neq 5) \wedge (\texttt{type} = 12), \text{ and,}$$
$$(\texttt{icp.icmp\_type} = \texttt{type}) \wedge (\texttt{type} \neq 5) \wedge (\texttt{type} \neq 12) \wedge (\texttt{type} = 3)$$

Thus, for each statement s where a downcast or upcast occurs, we compute, using the constraints generated by the type checking rules, the invariant at s.

Checking using Access Predicates. From the invariant, we construct an *access predicate* $\psi_s(t, t')$ by syntactically renaming all local variables in the invariant to fresh names, and renaming the cast expression with `this`. By replacing `icp` with `this` and `type` with a fresh, subscripted version, we have the access predicate $\psi_{05}(\texttt{packet}, \texttt{redirect})$:

$$\texttt{this.icmp\_type} = \texttt{type}_1 \wedge \texttt{type}_1 = 5 \tag{2}$$

To ensure that condition **R3** is met, we use a decision procedure[5] to check that at each downcast s of t to a subtype t', or upcast of t' to t, the access predicate $\psi_s(t, t')$ *implies* the edge predicate $\phi(t, t')$ (Rules Var-Up, Var-Down in Figure 3). So, for the downcast of `icp` from `packet` to `redirect` at line 05, we use a decision procedure to check the validity of the implication: $\texttt{this.icmp\_type} = \texttt{type}_1 \wedge \texttt{type}_1 = 5 \Rightarrow (\texttt{this.icmp\_type} = 5)$. In the given code snippet, at each downcast statement (there are no upcasts), the access predicate implies the corresponding edge predicate and so we conclude that the program is type safe.

Intuitively, the soundness of our type system follows from the following observations. First, we ensure that every new structure is a "leaf" of the type hierarchy. Thus, at run time, any instance that is ever downcast from, must

$$\frac{\Gamma(x) = t \qquad t' \preceq t \qquad \boxed{I[\mathtt{this}/x] \Rightarrow \phi(t,t')}}{\Gamma, \phi, I \vdash_l (t')x : t'}\ \textsf{Var-Down}$$

$$\frac{\Gamma(x) = t' \qquad t' \preceq t \qquad \boxed{I[\mathtt{this}/x] \Rightarrow \phi(t,t')}}{\Gamma, \phi, I \vdash_l (t)x : t}\ \textsf{Var-Up}$$

$$\frac{\Gamma, \phi, I \vdash_e e : t \qquad \Gamma, \phi, I \vdash_l (t)x : t}{\Gamma, \phi, I \vdash (t)x := e \rhd I \wedge (lv = e)}\ \textsf{Var-Assign}$$

$$\frac{\begin{array}{c}\Gamma, \phi, I \vdash_e e : t \\ \Gamma, \phi, I \vdash_e (t)lv.l : t \qquad \Gamma', \phi, I \vdash_l lv : t' \qquad l \notin \mathsf{tag}(t', \phi)\end{array}}{\Gamma, \phi, I \vdash (t)lv.l := e \rhd I \wedge (lv.l = e)}\ \textsf{Field-Assign}$$

$$\frac{\Gamma, \phi, I \vdash_l lv_i : t \text{ for all } i \qquad \Gamma, \phi, I \vdash_l lv : t}{\Gamma, \phi, I \vdash lv := \Phi(lv_1, \ldots, lv_n) \rhd I}\ \textsf{Assign-}\Phi$$

$$\frac{\Gamma, \phi, I \vdash_e p : \mathtt{bool} \qquad \Gamma, \phi, I \wedge p \vdash s \rhd I' \qquad \Gamma, \phi, I \wedge \neg p \vdash s' \rhd I''}{\Gamma, \phi, I \vdash \mathtt{if}\ \ p\ \ \mathtt{then}\ s\ \mathtt{else}\ s' \rhd I}\ \textsf{If}$$

$$\frac{\Gamma, \phi, I \vdash_e p : \mathtt{bool} \qquad \Gamma, \phi, I \wedge p \vdash s \rhd I'}{\Gamma, \phi, I \vdash\ \mathtt{while}\ \ p\ \ \mathtt{do}\ s \rhd I \wedge \neg p}\ \textsf{While}$$

Fig. 3. Relevant type checking rules. Hypotheses in boxes correspond to queries to the decision procedure made in the checking phase, or the predicate constraints in the inference phase. A complete set of rules is in [11]

have been upcast to at some point in the past. Second, our type system ensures that the tag fields are not altered, and therefore, any edge predicate that held at the upcast in the past, will continue to hold till the downcast. Thus, by checking the edge predicates at upcasts, and by requiring that edge predicates for sibling edges be pairwise inconsistent, our type system ensures there is a unique subtype that each supertype value is an instance of (and therefore, can be safely downcast to), namely the subtype whose edge predicate holds at the downcast point.

4 Type Inference via Interpolation

In the previous section, we assumed that we were *given* a predicated refinement of the subtype hierarchy with which the program could be type checked to ensure statically that all casts were safe. In practice, these annotations are not available. We now present an algorithm that given a program and the subtype hierarchy, *automatically infers* a predicated refinement of the hierarchy such that the program type checks, if indeed the program is type safe. In other words, given a program (T, Γ_0, s), the inference algorithm computes an edge predicate map ϕ that satisfies conditions **R1-R4** or reports that no such map exists, *i.e.,* the program is not type safe.

To find the predicate map ϕ, we introduce for each edge (t, t') in E a *predicate variable* $\pi_{t,t'}$. Next, using the syntax-directed type checking rules, we generate a set of *predicate constraints* on the predicate variables, such that a solution for the constraints will give us edge predicates that satisfy the three requirements. Finally, we describe how to solve the constraints and thus infer ϕ.

4.1 Generating Predicate Constraints

We use the syntax-directed typing rules of Figure 3 to generate the predicate constraints. The constraint generation is done in two phases.

In the first phase, we make a syntax-directed pass over the program to compute the set of fields that *cannot* be tag fields because they are modified *after* an upcast. This information is captured by computing a map $\overline{\mathsf{tag}}(t)$ from types t to the sets of fields that cannot be used in the edge predicates for edges $(t, \cdot)$.

In the second phase, we use type checking rules to compute the invariants at each access point. For a predicate I and a set of field names F, and a location s, define $\mathsf{rename}(I, F, s)$ as the predicate where all occurrences of free variables x other than `this` are substituted with a fresh name x_s and all occurrences of field names $l \in F$ are substituted with a fresh name l_s. At each downcast and upcast location s, *i.e.,* where one of the rules Var-Down, Var-Up (Figure 3) applies, instead of checking that the access predicate $I[\mathtt{this}/lv]$ implies the edge predicate for the cast, we introduce a predicate constraint:

$$\mathsf{rename}(I[\mathtt{this}/lv], \overline{\mathsf{tag}}(t), s) \Rightarrow \pi_{t,t'}$$

We call the LHS of the constraint above the *renamed access predicate* at location s. The renaming does not get in the way of inferring appropriate ϕ as the fields in $\overline{\mathsf{tag}}(t)$ cannot appear in $\phi(t, t')$. Instead, it will force the inferred predicates to not contain the fields in $\overline{\mathsf{tag}}(t)$, thus yielding a ϕ that suffices to type check the program, if one exists. Given a program $P \equiv (T, \Gamma_0, s)$, let $\mathsf{Cons}(P)$ be the set of predicate constraints generated by the algorithm described above.

We can always make the only upcasts and downcasts in the program be between immediate subtypes. Thus, the constraint generation introduces predicate constraints for $\pi_{t,t'}$ for edges $(t, t') \in E$.

Example 2. The downcast on line `05` in Figure 1(a) generates the constraint:

$$(\mathtt{type}_{05} = \mathtt{this.icmp\_type} \wedge \mathtt{type}_{05} = 5) \Rightarrow \pi_{\mathtt{packet},\mathtt{redirect}}$$

Similarly, the downcasts on line `09` and `12` generate constraints:

$$(\mathtt{type}_{09} = \mathtt{this.icmp\_type} \wedge \mathtt{type}_{09} \neq 5 \wedge \mathtt{type}_{09} = 12) \Rightarrow \pi_{\mathtt{packet},\mathtt{parameter}}$$

$$(\mathtt{type}_{12} = \mathtt{this.icmp\_type} \wedge \mathtt{type}_{12} \neq 5 \wedge \mathtt{type}_{12} \neq 12 \wedge \mathtt{type}_{12} = 3) \Rightarrow \pi_{\mathtt{packet},\mathtt{unreachable}}$$

Notice that the substitution renames `icp` to `this` and the variable `type` in each constraint.

Solutions. A *solution* to a set of constraints $\mathsf{Cons}(P)$ is a mapping Π from each predicate variable $\pi_{t,t'}$ to a predicate such that:

S1 For each predicate variable $\pi_{t,t'}$, the predicate $\Pi(\pi_{t,t'})$ has a single free variable `this`.
S2 For each triple t, t', t'', the predicates $\Pi(\pi_{t,t'})$ and $\Pi(\pi_{t,t''})$ are inconsistent.
S3 For each constraint $\psi_s \Rightarrow \pi_{t,t'}$ in $\mathsf{Cons}(P)$, the implication $\psi_s \Rightarrow \Pi(\pi_{t,t'})$ is valid.
S4 For each t, t', the predicate $\Pi(\pi_{t,t'})$ should not contain any field name in $\overline{\mathsf{tag}}(t)$.

Every solution Π for the set of constraints $\mathsf{Cons}(P)$, yields a predicated subtype hierarchy for P with which we can prove the safety of P.

Theorem 1. [Soundness of Constraint Generation] *For every program* $P \equiv (T, \Gamma_0, s)$*, if* Π *is a solution for the constraints* $\mathsf{Cons}(P)$ *then* $\phi \equiv \lambda(t,t').\Pi(\pi_{t,t'})$ *is such that:* $\Gamma_0, \phi, true \vdash s \rhd \cdot$.

4.2 Solving Predicate Constraints

We now give an algorithm to find a solution to a set of constraints $\mathsf{Cons}(P)$ if one exists. We define for each edge $(t,t') \in E$ a *cast predicate* $\psi(t,t')$ as:

$$\psi(t,t') \equiv \bigvee_{\psi_s \Rightarrow \pi_{t,t'} \in \mathsf{Cons}(P)} \psi_s$$

The cast predicate for an edge is the disjunction over all the renamed access predicates ψ_s for the locations where a t is downcast to t' or a t' is upcast to t. Note that by the properties of disjunction and implication, a map Π from the type variables to predicates is a solution for the constraints $\mathsf{Cons}(P)$ iff it satisfies conditions **S1, S2** and **S4**, and in addition

S3' For each (t,t') we have $\psi(t,t') \Rightarrow \Pi(\pi_{t,t'})$.

For each $\psi_s \Rightarrow \pi_{t,t'}$ we have $\psi_s \Rightarrow \psi(t,t')$ as the RHS cast predicate is the disjunction of all the corresponding access predicates ψ_s. Thus, by the properties of disjunction and implication, any solution Π satisfies requirement **S3'** iff it satisfies **S3**.

Existence of a Solution. A solution can only exist if for each triple t, t', t'', the conjunction $\psi(t,t') \wedge \psi(t,t'')$ is unsatisfiable. If not, *i.e.,* if there are t, t', t'' such that: $\psi(t,t') \wedge \psi(t,t'')$ is satisfiable, then for any candidate solution such that $\psi(t,t') \Rightarrow \Pi(\pi_{t,t'})$ and $\psi(t,t'') \Rightarrow \Pi(\pi_{t,t''})$, the conjunction $\Pi(\pi_{t,t'}) \wedge \Pi(\pi_{t,t''})$ is satisfiable, thus violating **S2**. Intuitively, if the conjunction of the cast predicates for t', t'' is satisfiable, it means that there is some condition under which the program casts to (or from) type t' as well as to (or from) t'' thus one of those casts may be unsafe, or depends on a modified field *i.e.,* a field in $\overline{\mathsf{tag}}(t)$. In this case, the type inference fails with an error message pointing out the two conflicting casts.

Algorithm 1 PredTypeInference

```
Input: Program $P = (T, \Gamma_0, s)$
Output: Refinement $(T, E, \phi)$ or ERROR
$E$ = edges induced by $\preceq$ on $T$; $C = \mathsf{Cons}(P)$
for all $(t, t') \in E$ do $\psi(t, t') = \vee\{\psi_s \mid \psi_s \Rightarrow \pi_{t,t'} \in C\}$
for all $t \in T$ with immediate subtypes $t_1, \ldots, t_n$ do
  if $\psi(t, t_1) \wedge \ldots \wedge \psi(t, t_n)$ is unsatisfiable then
    $\phi(t, t_1), \ldots, \phi(t, t_n) := \mathsf{ITP}(\psi(t, t_1), \ldots, \psi(t, t_n))$
  else return ERROR
return $(T, E, \phi)$
```

Constraint Solving via Interpolation. Dually, we show that if for each triple t, t', t'' the cast predicates $\psi(t, t')$ and $\psi(t, t'')$ are inconsistent, then through Craig interpolation [3] we can infer a solution to the constraints, and thus a predicated subtype hierarchy that suffices to type check the program. Given a sequence of predicates $A_1, \ldots, A_n$ such that for all i, j, the predicate $A_i \wedge A_j$ is unsatisfiable, a *pairwise interpolant* for the sequence is the sequence $\hat{A_1}, \ldots, \hat{A_n} \equiv \mathsf{ITP}(A_1, \ldots, A_n)$ such that **(I1)** For each i, the variables of $\hat{A_i}$ occur in each of $A_1, \ldots, A_n$, **(I2)** for each pair i, j, the predicate $\hat{A_i} \wedge \hat{A_j}$ is unsatisfiable, and **(I3)** for each i, the implication $A_i \Rightarrow \hat{A_i}$ is valid. For predicates over theories of equality and arithmetic, pairwise interpolants can be computed from the *proof of unsatisfiability* of conjunctions of two predicates [16].

For each node $t \in T$ with immediate subtypes $t_1, \ldots, t_n$, we define:

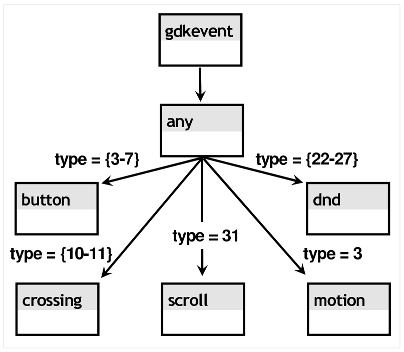
$$\Pi(t, t_1), \ldots, \Pi(t, t_n) \equiv \mathsf{ITP}(\psi(t, t_1), \ldots, \psi(t, t_n))$$

The properties of pairwise interpolants suffice to show that Π is indeed a solution to the constraints $\mathsf{Cons}(P)$. The only variable common to $\psi(t, t_1), \ldots, \psi(t, t_n)$ is `this` and hence, by **I1** each $\Pi(t, t')$ contains the sole free variable `this`, thus enforcing requirement **S1**. In addition, as we renamed all the fields in $\overline{\mathsf{tag}}(t)$, there is no field name in $\overline{\mathsf{tag}}(t)$ that is in any $\psi(t, t')$ and thus Π meets condition **S4**. Property **I2** of interpolants ensure requirement **S2**. Finally, property **I3** of interpolants ensures requirement **S3'** and hence, **S3**.

By Theorem 1, we have inferred an edge map ϕ and thus, a predicated subtype hierarchy that suffices to show that all casts are safe. The inference algorithm runs in time quadratic in the number of constraints, and thus, the program, and makes a quadratic (in the size of T) calls to an interpolating decision procedure.

We summarize the predicated type inference algorithm PredTypeInference in Algorithm 1. The correctness of the algorithm is stated in the following theorem.

Theorem 2. **[Correctness of Type Inference]** *For every program $P \equiv (T, \Gamma_0, s)$,* PredTypeInference$(P)$ *terminates. If* PredTypeInference(P) *returns (T, E, ϕ) then $\Gamma_0, \phi, true \vdash s \rhd \cdot$. If* PredTypeInference$(P)$ *returns* ERROR *then there is no ϕ such that $\Gamma_0, \phi, true \vdash s \rhd \cdot$.*



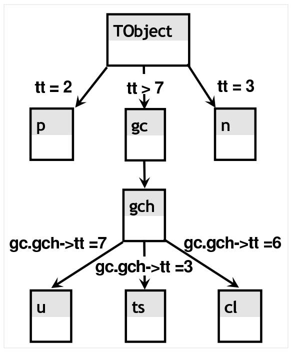


Fig. 4. Predicate subtype hierarchy for (a)gdkevent (b)lua

Example 3. For the constraints from Example 2, we get the cast predicates:

$$\begin{array}{ll}
\psi(\texttt{packet},\texttt{redirect}) & \texttt{type}_{05} = \texttt{this.icmp\_type} \wedge \texttt{type}_{05} = 5 \\
\psi(\texttt{packet},\texttt{unreachable}) & \texttt{type}_{09} = \texttt{this.icmp\_type} \wedge \texttt{type}_{09} \neq 5 \wedge \texttt{type}_{09} = 12 \\
\psi(\texttt{packet},\texttt{parameter}) & \texttt{type}_{12} = \texttt{this.icmp\_type} \wedge \texttt{type}_{12} \neq 5 \wedge \texttt{type}_{12} \neq 12 \wedge \texttt{type}_{12} = 3
\end{array}$$

The only common names are `this` and the allowed fields. The pairwise interpolant of these predicates yields the edge predicates: $\texttt{this.type} = 5$, $\texttt{this.type} = 12$ and $\texttt{this.type} = 3$.

5 Implementation and Experiences

Implementation. We have implemented the predicated type inference algorithm for C. We use CIL [17] to parse and manipulate the C program, the structural invariant package [12] to generate constraints, and the theorem prover Foci [16] to generate interpolants and check implications at cast locations. We first use physical subtyping [20] to get subtyping hierarchy based on structural prefixes. Next we model union accesses as casts. We add types representing each field in an union. If a structure t contains a union $t.u$ with fields f_i, we create a immediate subtype t'_i representing the same structure t but only allowing access to $t.u.f_i$. In the implementation, access to the union field $t.u.f_i$ is the same as a downcast from t to t'_i. We extend the invariants generation algorithm as described previously with pointers and functions using the techniques in [12].

Experimental Results. We summarize our experimental results on nine open source programs in Table 1. Our algorithm identified 1,684 downcasts requiring predicate guards. These accesses were determined by a predicated subtyping hierarchy of 90 edges. We were able to infer 77 predicate edges corresponding to union fields correctly. We also correctly inferred the 3 predicate edges corresponding to explicit C type casts in `moapsource`. Our tool can derive complex predicated subtyping hierarchies. Figure 4 shows the two partial predicated subtyping hierarchies for `gdkevent` and `lua`. Some subtypes are dropped because

Program	Description	LOC	Predicate Edges Inferred	Actual	Accesses	Time
`ip_icmp`	FreeBSD ICMP	7K	7	7	15	1s
`xl`	SPEC Lisp interpreter	12K	8	8	428	875s
`moapsource`	Emstar packet processor	14K	3	3	5	1s
`gdkevent`	GDK events	16K	12	13	90	38s
`lua`	Lua compiler	18K	13	15	274	151s
`snort`	Intrusion detector	42K	7	7	26	12s
`sendmail`	Mail server	106K	17	24	406	995s
`ssh`	ssh secure shell	35K	0	0	0	12s
`bash`	Bourne again shell	101K	13	13	440	1157s
`Total`		351K	80	90	1684	3242s

Table 1. Experimental Results: **LOC** is lines of code. **Time** is the number of seconds spent on inference. **Predicate Edges** is the number of predicated edges in the predicated subtype hierarchy. **Inferred** is the number such edges for which our tool inferred an edge predicate, and **Actual** is the number of edges constructed by manual inspection of the code. **Accesses** is the number of predicated cast points. Experiments were run on a Dell PowerEdge 1800 with two 3.6GHz processors and 5GB memory.

of space. Note that predicate edges are not simply single tag assignments but rather more complex predicates involving ranges.

Conflicts and Bugs. There are two sources of conflicts: casts on edges for which the predicated access idiom is not followed, and casts on edges where a predicate was inferred but where the generated access predicate was not strong enough to establish the edge predicate.

For the first case, we use the following heuristic to determine which edges have no predicates. Given a type node, if *all* cast instances of an outgoing predicate edge conflict with all cast instance of any other outgoing predicate edge, then we conclude that type node has no guards. An example of this is the `memset` macro which uses a union to access different bytes in memory. Our heuristic correctly distinguishes all such accesses from predicated casts.

For the second case, when the access predicate was not strong enough to imply the edge predicate, the downcast instance typically conflicted with many other downcast instances. Here either our generated invariant was too weak, the predicate guard used variables that were not part of the structure, or there was a bug. For the few cases where our invariants are too weak, more precise invariant generators (e.g., Blast [10]) can be used to statically verify that the edge predicates hold at the cast points, or alternatively, dynamic checking can be inserted to ensure type-safety at runtime [18, 7, 15]. One bug is when the programmer forget to check the predicate before the access, leaving the possibility of an unsafe access. For example, there are 23 cases in Lua where two different variables are assumed to have the same predicate hold. An union field in one of those variables is accessed after the appropriate predicate is checked. However, the same union field in the other variable is also accessed without checking if the appropriate predicate holds as well. That is, there is an unchecked assumption that two different variables always satisfy the same predicate.

6 Related Work

Language support. Functional programming languages like ML and Haskell provide disjoint sum types within the language. The Cyclone language [13] provides mechanisms such as sum types and subtyping within C, allowing safer programs to be written within a C-like language. Our goal on the other hand is to check for safe usage in a large body of legacy code written in C or in low level code where bytes "off the wire" must be cast to proper data types (as in networking code).

Static analysis. There is a large body of recent work on statically proving memory safety of C programs (augmented with adding runtime checks) to make them execute safely [15, 18, 2]. CCured [18] performs a pointer-kind inference and adds runtime checks to make C programs memory safe. However, CCured leaves open the question of statically checking proper usage of unions or downcasts of pointers: either putting in additional tags or removing unions altogether and replacing them with structures. The former technique ignores checks the programmer already has in place, the latter technique may not work for applications such as network packet processors where the data layout cannot be changed. Runtime type information has been used for bug finding and providing debugging information for bad casts or union access [15], but the inference problem has not been studied. Identifying correct use of datatypes in the presence of memory layout and casts has been studied in [2, 20]. However, these type systems do not correlate guards to ensure correctness of downcasts.

Dependent types. There is substantial previous work in dependent types [22, 9, 21]. The predicate subtyping scheme of PVS [19] is more general than our system. All these systems require interactive theorem proving as the type systems are undecidable. By restricting our target properties and proof strategies, we provide an automatic mechanism. Closer to our work, [9] provides dependent record types to encode safety properties such as array bound checks and null pointer dereferences. The type system of [8] infers dependent types for representing ML values passed to C programs through the foreign language interface. Unlike our algorithm, they fix a set of dataflow facts for the guards.

Our type system is closest to the type systems in [1] and [14]. The type system in [1] only tracks the evaluation of ML-style pattern-matching statements. Our type system tracks all assignments and conditionals dominating the access. In [14], the authors consider the problem of identifying record types and guarded disjoint unions in COBOL programs. However, both approaches infer types by using a dataflow analysis to track equalities between variables and constants appearing in branch statements. In many of our experiments we have found that this simple language of guards is insufficient (because, for example, programmers use guards of the form `tag` ≥ 5). Further, the problem of identifying guards in terms of the *in scope* fields is not considered.

Acknowledgments. We thank Todd Millstein and Pat Rondon for carefully reading drafts and providing valuable feedback.

References

1. A. Aiken, E. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *POPL 94*, pages 163–173. ACM, 1994.
2. S. Chandra and T. Reps. Physical type checking for c. In *PASTE 99*, pages 66–75. ACM, 1999.
3. W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.
4. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadek. Efficiently computing static single assignment form and the program dependence graph. *ACM TOPLAS*, 13:451–490, 1991.
5. D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
6. Y. Fang. *Translation validation of optimizing compilers.* PhD thesis, New York University, 2005.
7. C. Flanagan. Hybrid type checking. In *POPL 06*, pages 245–256. ACM, 2006.
8. M. Furr and J. Foster. Checking type safety of foreign function calls. In *PLDI 05: Programming Language Design and Implementation*, pages 62–72. ACM, 2005.
9. M. Harren and G.C. Necula. Using dependent types to certify the safety of assembly code. In *SAS 05*, LNCS 3672, pages 155–170. Springer, 2005.
10. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
11. R. Jhala, R. Majumdar, and R. Xu. Type inference using Craig interpolation. http://www.cs.ucla.edu/∼rxu/techrep.ps.
12. R. Jhala, R. Majumdar, and R. Xu. Structural invariants. In *SAS 06*. Springer, 2006.
13. T. Jim, J.G. Morrisett, D. Grossman, M.W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Usenix Tech. Conf.*, pages 257–288, 2002.
14. R. Komondoor, G. Ramalingam, S. Chandra, and J. Fields. Dependent types for program understanding. In *TACAS 05*, LNCS 3440, pages 157–173. Springer, 2005.
15. A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *FASE 01*, LNCS 2029, pages 217–232. Springer, 2001.
16. K.L. McMillan. An interpolating theorem prover. *TCS*, 345:101–121, 2005.
17. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*, LNCS 2304, pages 213–228. Springer, 2002.
18. G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM TOPLAS*, 27(3):477–526, 2005.
19. J.M. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Trans. Software Eng.*, 24(9):709–720, 1998.
20. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *ESEC/FSE 99*, pages 180–198. ACM, 1999.
21. H. Xi and R. Harper. A dependently typed assembly language. In *ICFP 01: International Conference on Functional Programming*, pages 169–180. ACM, 2001.
22. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL 99: Principles of Programming Languages*, pages 214–227. ACM, 1999.