State of the Union: Dependent Type Inference via Craig Interpolation

Ranjit Jhala
CSE Department, UC San Diego
jhala@cs.ucsd.edu

Rupak Majumdar
CS Department, UC Los Angeles
rupak@cs.ucla.edu

Ru-Gang Xu
CS Department, UC Los Angeles
rxu@cs.ucla.edu

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 be safely 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 approximately 1500 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 for statically checking the safety of downcasts in imperative programs, and a novel technique for inferring dependent types based on Craig Interpolation. 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 a 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 accesses is a common source of subtle bugs due to memory corruption.

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 —we can consider each possible "completion" of a structure with the different elements of the union as subtypes of the structure, and we can 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 dependent 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 ingredient is a *predicated refinement* of the hierarchy, where we label the edges of the type hierarchy tree with *edge predicates* over the fields of the structure that are true 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. Though there are several static verification engines that can be used for this purpose, 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 new technique to infer dependent types via Craig Interpolation. To infer the refinement, we first generate a system of *predicate constraints* with variables representing the unknown edge predicates. The predicate constraints force the solutions for the variables to have the key properties of edge predicates, namely: (1) that they be over the fields of the structure, (2) that the edge predicates for the subtypes be pairwise inconsistent, and, (3) that the edge predicates hold at each cast point, *i.e.,* at each (up- or down-) cast point, the cast predicate implies the edge predicate.

To solve these constraints, we observe that the *pairwise Craig Interpolant* for a sequence of formulas $A_1, \ldots, A_n$ that are pairwise inconsistent, is a sequence of formulas $\hat{A}_1, \ldots, \hat{A}_n$ such that: (a) each $\hat{A}_i$ contains variables that occur in all of $A_1, \ldots, A_n$, (b) each pair $\hat{A}_i, \hat{A}_j$ is inconsistent, and, (c) each A_i implies $\hat{A}_i$. Pairwise Craig interpolants are guaranteed to exist for recursively enumerable theories, and can be efficiently computed for many the-

ories of practical interest [22]. Note that the properties (a),(b) and (c) of interpolants correspond directly to the requirements (1),(2) and (3) of the edge predicates. We show that a predicated refinement exists iff for each type, the cast predicates for its subtypes are pairwise inconsistent. Thus, to solve the predicate constraints, and infer the (dependent) predicated refinement, we compute the edge predicates for the subtypes of each type as the pairwise interpolants of the cast predicates for the subtypes.

We have implemented the predicated subtype inference algorithm for the C language, 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 approximately 1500 union access points), our algorithm finds the correct predicate guards (which we manually verified a posteriori).

2. Related Work

Language support. Functional programming languages like ML and Haskell provide disjoint sum types within the language. The Cyclone language [19] 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. Moreover, our techniques are also useful 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 properties of C programs (augmented with adding runtime checks) to make them execute safely [3, 21, 14, 8, 3]. CCured [24] 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 [21], 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 [4, 27]. However, these type systems do not correlate guards to ensure correctness of downcasts.

Dependent types. There is a large body of work in dependent types [30, 28, 16, 29]. There are several recent attempts to add dependent types within a programming language. For example, Xanadu [28] and ATS [6] provide expressive dependent types within the language and can express more general program invariants than predicated unions. Similarly, the predicate subtyping scheme of PVS [25] is more general than our system. However, all these systems requires interactive theorem proving as the type system becomes undecidable. By restricting our target properties and proof strategies, we provide an automatic mechanism. Closer to our work, [16] provides dependent record types to encode safety properties such as array bound checks and null pointer dereferences. These general systems require annotations at the basic block level to type check programs. Our inference algorithm is based on interpolating decision procedures [22] and can automatically infer dependent type information based on visible variables.

The type system of [15] infers dependent types for representing ML values passed to C programs through the foreign language interface. They infer types via a dataflow analysis that uses a specific lattice of facts derived from the OCAML foreign function interface, unlike our algorithm, which infers generic predicates.

Our type system is closest to the type systems in [1] and [20]. 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 [20], 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 $\texttt{tag} \geq 5$).

3. Overview

We now give an overview of our dependent type system and our technique for using interpolation for inference, by showing how we automatically infer the predicates that determine how the union of the example in Figure 1 is used. In particular, we show how we infer the predicated type refinement that captures the informal intuition that the packet can be interpreted as a *parameter* message (*i.e.,* the `ih_pptr` field can be accessed) when the field $\texttt{icmp\_type} = 12$, as a *redirect* message (*i.e.,* the `ih_gwaddr` field can be accessed) when the field $\texttt{icmp\_type} = 5$, and as an *unreachable* message (*i.e.,* the `ih_pmtu` field can be accessed) when the field $\texttt{icmp\_type} = 3$.

The type system has two ingredients: a *subtype hierarchy* corresponding to a set of (simple) types and a subtyping relation, and a *predicate refinement* of the subtype hierarchy that specifies when a supertype can be downcast to a subtype.

Ingredient 1: Subtype Hierarchy. A *(simple) type* is either a base type `int` or `bool`, or a *structure* which is a list of pairs of field names and their (simple) types. For two (simple) types t, t' we say $t' \preceq t$, or t' is a *subtype of* t if t is a prefix of t'. A *Subtype Hierarchy* is a forest (T, E) where the *nodes* correspond to a set of (simple) types T, and the set of *edges* $E \subseteq T \times T$ is 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 $t' \prec t''$ and $t'' \prec t$.

Consider the structure definitions of the program of Figure 1(a). We can "unroll" the union definitions to obtain three (simple) subtypes of the type `packet`, namely `redirect`, `param` and `unreach`, 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 (simple) 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'.

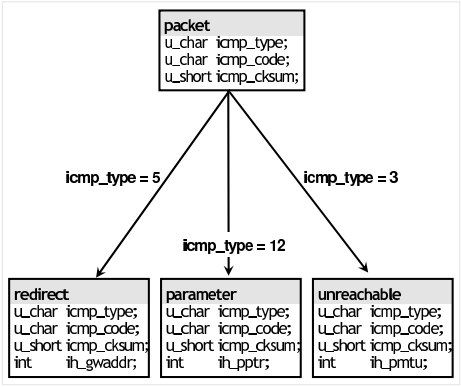
Safe Unions via Safe Downcasts. We reduce the problem of checking the safety of union accesses to checking the safety of *downcasts* in our system by converting each union access into a downcast to the subtype containing the particular union field being accessed, followed by a standard field access on the subtype. Figure 1(c) shows how the converted version of the program of Figure 1(a), with all union access replaced with explicit downcasts followed by field accesses, at statements 05, 09, and 11. Next, we see how to *refine* the subtype hierarchy to enable the static checking of the safety of downcasts and thus, union accesses.

Ingredient 2: Predicated Refinement. 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.

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*.

```
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;
};
00 int type, dest, code;
01 struct packet icp;
02 . . .
03 type = icp.icmp_type;
04 if (type == 5) {
05   icp.icmp_hun.ih_gwaddr = dest
06 }
07 else {
08   if (type == 12) {
09     icp.icmp_hun.ih_pptr = 0;
10     code = 0;
11  } else if (type == 3) {
12     icp.icmp_hun.ih_pmtu = 0;
13  }
14 }
```




```
type packet =
  s{(icmp_type,int),(icmp_code,int),
    (icmp_cksum,int),(icmp_hun,void)};
type redirect =
  s{(icmp_type,int),(icmp_code,int),
    (icmp_cksum,int),(ih_gwaddr,int)};
type packet =
  s{(icmp_type,int),(icmp_code,int),
    (icmp_cksum,int),(ih_pptr,int)};
type packet =
  s{(icmp_type,int),(icmp_code,int),
    (icmp_cksum,int),(ih_pmtu,int)};

00 int type, dest, code;
01 packet icp;
02 ...
03 (int)type := (int)((packet)icp).icmp_type;
04 if ((int)type = 5) then
05   (int)((redirect)icp).ih_gwaddr := (int)dest;
07 else
08   if ((int)type = 12)
09     (int)((parameter)icp).ih_pptr := 0;
10     (int)code := 0;
11   else if ((int)type = 3)
12     (int)((unreachable)icp).ih_pmtu := 0;
13   else ...
```

Figure 1. (a) ICMP Example (b) (Union) Subtype Hierarchy and Predicated Refinement (c) Code translated to our core language with casts explicated

In this case, we refer to (T, E, ϕ) as a *Predicated Subtype Hierarchy*.

We use the predicate refinement to statically check the safety of downcasts and thus, union accesses. Intuitively, the edge predicate $\phi(t, t')$ specifies the conditions under which a value of the supertype t actually corresponds to an instance of type t' and can be safely downcast. Before performing a downcast (*i.e.*, accessing a union field) the programmer must determine which downcast is safe by determining which of the edge predicates of the immediate subtypes holds. **R1** ensures these predicates are over *tag* fields of the supertype, and **R2** ensures that only one of them holds. Thus, the edge predicates formalize and explicate the informal "tagging protocol" used by the programmer.

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. We now show how, *given* a predicated refinement for a subtype hierarchy, we can *statically check* the safety of downcasts.

3.1 Type Checking

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, and (3) 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 a third requirement:

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.

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 (simple) 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.

Invariants. A typing judgment in our type system carries along an *invariant* in addition to the type environment. The invariant is a predicate on the program state that is guaranteed to hold at the program point (for all program executions reaching that point). A typing rule additionally transforms the invariant by adding the effect of the current statement on the invariant. Intuitively, the invariant tracks the set of all facts that directly *dominate* a particular statement, *i.e.*, the set of all program facts that are guaranteed to be executed on all paths from the entry point of the function to the program point. This captures the idiom that the tag fields for a union are checked in a conditional dominating the access of the union.

In Figure 1(c), 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 of 05 is:

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

Similarly, the invariants for 09 and 11 are respectively:

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

$\psi_{05}(\texttt{packet}, \texttt{redirect})$:

$$\texttt{this.icmp\_type} = \texttt{type}_1 \wedge \texttt{type}_1 = 5 \quad (2)$$

To ensure that condition **R3** is met, we use a decision procedure[9] 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 $\phi(t, t')$. 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.

3.2 Type Inference via Craig Interpolation

Given a subtype hierarchy and a program, the *type inference* problem is to find a predicated refinement of the subtype hierarchy that suffices to type check the program, if one exists. Thus, the problem is to infer the edge predicates, *i.e.,* the mapping ϕ from E to predicates that satisfies requirements **R1**,**R2** and **R3**, and thus ensures that the program type checks.

For a subtype hierarchy (T, E), let the *cast predicate for* $(t, t') \in E$, written $\psi(t, t')$, be the disjunction of all access predicates $\psi_s(t, t')$ over all program statements s where t is downcast to t', or t' is upcast to t. For the types `packet` and `redirect`, there is a single cast, so the cast predicate $\psi(\texttt{packet}, \texttt{redirect})$ is just $\psi_{05}(\texttt{packet}, \texttt{redirect})$ shown in formula (2).

Using a decision procedure we check that for each pair of immediate subtypes t' and t'' of a type t that $\psi(t, t') \wedge \psi(t, t'')$ is unsatisfiable. If not, *i.e.,* if the conjunctions of the cast predicates is satisfiable, there exists a pair of program executions for which the same type is downcast or upcast from two different subtypes which cannot be distinguished by the generated invariants, and so we conclude that there is no suitable predicated refinement.

Instead, suppose that we have found the conjunction of all the pairs cast predicates to be unsatisfiable. As the predicates $\psi(t, t')$ contain local variables, they do not satisfy the requirement that predicates are exclusively over the fields of the supertype. To get predicates over the supertype fields, we use *interpolation* [5].

Pairwise Interpolants. 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.

If predicates are over theories of equality and arithmetic, interpolants can be computed from the *proof of unsatisfiability* of conjunctions of two predicates [22].

To infer appropriate edge predicates we compute the pairwise interpolant for the sequence of cast predicates for the immediate subtypes of t. Thus, if the immediate subtypes of a type t are $t_1, \ldots, t_n$, then:

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

As the conjunction of the pairs of cast predicates is unsatisfiable, the interpolant is guaranteed to exist. Due to the renaming, the only variable in common to the different A_i predicates is `this`. Condition **I1** implies that the interpolants, and thus, the inferred edge predicates are over `this` and fields that are reachable from `this`, *i.e.,* fields of the structure t, thus enforcing requirement **R1**. Condition **I2** ensures that the pairwise conjunction of the predicates for the different immediate subtypes are unsatisfiable, thus enforcing requirement **R2**. Finally, **I3** ensures the cast predicate implies the inferred edge predicate

In our example, the cast predicates $\psi(\texttt{packet}, \texttt{redirect})$, $\psi(\texttt{packet}, \texttt{param})$ and $\psi(\texttt{packet}, \texttt{unreach})$ are respectively:

$\texttt{this.icmp\_type} = \texttt{type}_{05} \wedge \texttt{type}_{05} = 5,$

$\texttt{this.icmp\_type} = \texttt{type}_{05} \wedge \texttt{type}_{05} \neq 5 \wedge \texttt{type}_{12} = 12,$ and,

$\texttt{this.icmp\_type} = \texttt{type}_{12} \wedge \texttt{type}_{12} \neq 5 \wedge \texttt{type}_{12} \neq 12 \wedge \texttt{type}_{12} = 3$

These cast predicates are pairwise unsatisfiable, and so we compute the edge predicates:

$$\phi(\texttt{packet}, \texttt{redirect}), \phi(\texttt{packet}, \texttt{param}), \phi(\texttt{packet}, \texttt{unreach})$$

as the pairwise interpolant of

$$\psi(\texttt{packet}, \texttt{redirect}), \psi(\texttt{packet}, \texttt{param}), \psi(\texttt{packet}, \texttt{unreach})$$

which yields the edge predicates:

$$\texttt{this.icmp\_type} = 5, \quad \texttt{this.icmp\_type} = 12, \quad \texttt{this.icmp\_type} = 3$$

3.3 Soundness via Checking Upcasts and Downcasts

In order to ensure soundness, our type system ensures that the edge predicates hold both at upcasts and at downcasts. To see why checks are necessary at both places, consider the unsafe example shown in Figure 2, which is a version of the ICMP fragment from Figure 1 where additionally (lines `X4`–`X1`) an instance of the subtype `redirect` is created (corresponding to the creation of a `packet` instance where the union element is an `ih_gwaddr`), which is then upcast to the supertype `packet`. The program is unsafe, as the "wrong" tag value is written in line `X2` (or, depending on ones point of view, the wrong values are checked further down).

Our type system catches this, because we check (resp. infer) the edge predicates using (resp. from) the access predicates at downcasts *and upcasts*. In this unsafe example, the cast predicate $\psi(\texttt{packet}, \texttt{redirect})$ is

$$(\texttt{this.icmp\_type} = 12 \wedge \texttt{this.ih\_gwaddr} = \ldots) \vee (\texttt{this.icmp\_type} = \texttt{type}_{05} \wedge \texttt{type}_{05} = 5)$$

where the first disjunct comes from the access predicate due to the upcast on line `X1` and the second disjunct comes from the access predicate due to the downcast on line `05`. The cast predicate for $\psi(\texttt{packet}, \texttt{param})$ is the same as before. However, the cast predicates for the two immediate subtypes `redirect` and `param` are *consistent*, *i.e.,* their conjunction is satisfiable, and so, no predicate refinement can be inferred, and our type system rejects this program as unsafe.

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 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. Language and Type System

We formalize our approach with a core imperative language with simple types. We first describe the language, then define our dependent type system, and finally, present our type checking algorithm. Recall that C programs with unions can be translated into our core

```
int type, dest, code;
packet icp;
redirect rp;
X4 (redirect)rp = new(redirect);
X3 (int)((redirect) rp).ih_gwaddr = ...;
X2 (int)((redirect) rp).icmp_type = 12;
X1 (packet) icp = (packet) rp;
02 ...
03 (int)type := (int)((packet)icp).icmp_type;
04 if ((int)type = 5) then
05   (int)((redirect)icp).ih_gwaddr := (int)dest;
07 else
08   if ((int)type = 12)
09     (int)((parameter)icp).ih_pptr := 0;
10     (int)code := 0;
11   else if ((int)type = 3)
12     (int)((unreachable)icp).ih_pmtu := 0;
13   else ...
```

Figure 2. Unsafe version of ICMP

language as shown in Figure 1. In the converted program, union fields are accessed after casting the lvalue down to the subtype containing the union. Thus, in our setting, the problem of checking the correct use of unions is reduced to that of checking the safety of downcasts.

4.1 Syntax and Semantics

For ease of exposition, we present the intraprocedural, pointer-free case – our implementation, described in Section 6, handles both procedures and pointers.

Types. Figure 3(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 $\mathtt{s}\{\}$. 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 3(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 $\mathtt{new}(t)$ statement creates a new structure of type t, and is used to model allocation. For ease of exposition, in our language every lvalue lv includes a type cast (t), label l that 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.

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 [7], 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,

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

(a) Expressions and Statements

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

(b) Types and declarations

Figure 3. Syntax and Types. n is an integer constant, v a variable, l a string label, $\sim \in \{<, >, \leq, \geq, \neq\}$, and $\oplus \in \{+, -\}$.

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 [21]. 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. For a predicate p, we write $\Sigma, M, W \models p$ to mean that the predicate p evaluates to true in the state defined by Σ, M, W. When a structure is created during execution using the $\mathtt{new}(t)$ 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 intuitively, any attempt to downcast it to a type not along this path leads the program into a "stuck" state. We assume for simplicity that each base type takes exactly one memory word. 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 define the predicate $\mathsf{WF}(\Sigma, M, W)$ which defines a program state that is compatible with the runtime type information, i.e., for each variable or address p, the value of p ($\Sigma(p)$ or $M(p)$) is a valid element of the type $W(p)$. 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 $(\Sigma, M, W; s) \rightarrow (\Sigma', M', W'; s')$.

4.2 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. We represent the subtype hierarchy for the set of types T in a program as a directed forest (T, E), where the nodes correspond to the types T correspond to nodes and there is an edge $(t, t') \in E$ iff t' is an immediate subtype of t, i.e., $t' \preceq t$ and there is no t'' different from t and t' such that $t' \preceq t'' \preceq t$. We write $N(t) = \{t' \mid (t, t') \in E\}$ for the set of neighbors (immediate

subtypes) of a type $t \in T$. We say that t is a *leaf type* if $N(t)$ is empty, i.e., t has no subtypes. We shall require that in our programs, whenever a structure is created, it belongs to a leaf type. For ease of exposition, we shall assume that all casts in a program are between neighbors: whenever a value of type t is cast to t', either $t \in N(t')$ or $t' \in N(t)$. We can enforce this by converting the program to a normal form by introducing temporary variables into the program to hold the values at intermediate cast points.

Predicated Subtype Hierarchy. A *predicated refinement* of a subtype hierarchy (T, E) is a triple (T, E, ϕ) where ϕ is a map from edges in E to quantifier free predicates that satisfies properties **R1**,**R2** and **R3** (Section 3).

Tag Fields. The *tag fields* of a type $t \in T$ as:

$$\mathsf{tag}(t, \phi) \equiv \{l \mid \exists t' \prec t \ : this.l \text{ occurs in } \phi(\cdot, t')\}$$

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.

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.

4.3 Type Checking using Predicated Refinements

This intuition is formalized in our type checking algorithm that takes a program and a predicated subtype hierarchy, and follows a three step process to check the safety of all the downcasts.

In the first step, we check that each access conforms to the subtype hierarchy. This is standard type checking, where we assume that all the casts are safe and use the explicated cast information to ensure that at each access $(t)lv.l$ that t is a supertype or subtype of lv and that the field l is indeed a field of the structure t. In the second phase, we use a decision procedure (e.g., SIMPLIFY[9]) to check that the supplied type hierarchy meets requirements **R1**, **R2**. In the third phase, we check condition **R3** and also:

R4 That the tag fields of a structure are not modified.

We now describe the third phase in detail.

Checking the Edge Predicates. We present a dependent type and effect system that checks the edge predicates hold at each cast location. In addition, our type system ensures that the tag fields are not updated after a type has been upcast from a leaf type. This, together with the disjointness of the edge predicates, enables us to statically type check the safety of downcasts using a predicated subtype hierarchy. Our type system is flow sensitive: judgments carry an *invariant* that is updated with the effect of each statement. An invariant is a predicate over program states. We assume that the function and atomic relation symbols appearing in the predicates are interpreted by a decidable theory, and all validity checks are done using a decision procedure for the theory.

Judgments. A *judgment* in the type system for a statement s is of the form $\Gamma, \phi, I \vdash s \rhd I'$. 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'.

The judgment uses auxiliary relations where t is the checked type: $\Gamma, \phi, I \vdash_e e : t$ to type expressions and $\Gamma, \phi, I \vdash_l l : t$ to type lvalues. These judgments state that under the type assumptions Γ and using the edge predicate map ϕ we can deduce that the expression e (resp. the lvalue lv) has the type t and there are no type errors. Our syntax-directed *derivation rules* for inferring type judgments are shown in Figure 4. At each cast point, the rules check, using a decision procedure, that the invariants imply the corresponding edge predicate. In the figure, the queries made to the decision procedure are highlighted using boxes.

Derivation Rules: Invariants. Intuitively, the rules accumulate an invariant consisting of all program facts that dominate a particular statement (*i.e.*, the set of facts that hold on every execution up to a statement). The SSA form ensures that this set of facts form an invariant, *i.e.*, every execution to this program point satisfies the formula. The rules for statements accumulate the invariants by conjoining the statements that dominate each cast point.

- Var-Assign for an assignment $(t)x := e$ requires that the lvalue on the left hand side has the same type as the right hand side, but strengthens the invariant I to $I \wedge x = e$ thus capturing the value flow due to the assignment.
- Field-Assign for an assignment to a field $lv.l := e$ is similar to the previous rule, but in addition ensures that the l is not a tag field of lv, thus ensuring that the tag fields are *not modified* after an upcast.
- Assign-Φ checks that all the values being joined at the Φ-node have the same type as the lvalue being assigned to.
- Seq for sequencing combines the effects sequentially.
- If collects additional facts in the then and else branches: along the then branch, the new invariant is $I \wedge p$ ensuring p holds; along the else branch, the invariant is $I \wedge \neg p$. At the end of the if statement, these additional facts are removed and the resulting invariant is again I. Note that this loses path correlation information.
- While combines the invariant with the loop condition inside the body of the loop and throws away the effect of the loop body at the end.

The SSA form is critical for ensuring that the formulas gathered in the invariant I are indeed guaranteed to hold before each statement executes [18, 11], thus enabling sound type checking. Though there are more powerful techniques for generating invariants, the method we use is highly scalable, essential for a type checker, and it is *syntax-directed* thereby giving the programmer a clear specification of what programs will be accepted by the type system. Notice that the rules are syntax directed and thus essentially work by traversing the AST of the program

Derivation Rules: Casts. The key rules are those pertaining to the casting of lvalues. The lvalue rules check that the access predicate implies the edge predicate for the corresponding cast (**R3**). The implication checks, highlighted using boxes, are done using a theorem prover [9]. For a predicate I and an lvalue lv, we write $I[\texttt{this}/lv]$ for the predicate obtained by replacing all occurrences of lv with `this`.

- Var-Down allows a downcast only if the current invariant, when substituted with `this`, implies the edge predicate for the downcast. The substituted invariant at this point is called the *access predicate* for that particular access, and thus, the rule permits downcasts from t to t' only if the access predicate implies the edge predicate $\phi(t, t')$.
- Var-Up allows an upcast from subtype t' to type t only if the renamed invariant implies the edge predicate $\phi(t', t)$. This rule ensures that after the t' instance is *created*, at the first point at

which it is cast up to a supertype t, it satisfies the edge predicate. As the tag fields appearing in the predicate cannot change, and as only one of the edge predicates of siblings can be true, if at some point in the future we know that the edge predicate still holds, then the supertype instance must be the result of an up cast from a t' instance, and therefore it is safe to cast down to t'.

- Field-Down and Field-Up are analogous rules for field accesses.
- Var-Eq and Field-Eq handle the explicit trivial casts put into the program.

Soundness. We can show that a program $P \equiv (T, \Gamma_0, s)$ is type safe if there exists a predicated subtype hierarchy (T, E, ϕ) such that using the rules we can derive the judgment $\Gamma_0, \phi, \mathit{true} \vdash s \rhd \cdot$. The type soundness theorem states the informal idea that well typed programs do not have unsafe casts.

THEOREM 1. **[Type Soundness]** *Let $P = (T, \Gamma_0, s)$ be a program and (T, E, ϕ) be a predicated subtype hierarchy. Let Σ (resp. M) be an arbitrary store (resp. memory) such that* $\mathsf{WF}(\Sigma, M, \Gamma_0)$. *If $\Gamma_0, \phi, \mathit{true} \vdash s \rhd \cdot$ then either $(\Sigma, M, \Gamma_0; s)$ diverges or $(\Sigma, M, \Gamma_0; s) \rightarrow^* (\Sigma', M', W'; \mathtt{skip})$ and* $\mathsf{WF}(\Sigma', M', W')$.

We use the initial typing environment Γ_0 obtained from the static declarations in a program as the initial RTTI. We exploit the critical fact, proved in [18], that the invariant carried along in a judgment is an over-approximation of the set of the program states before that statement is executed, that is, if $\Gamma_0, \phi, \mathit{true} \vdash s \rhd I$ then the invariant I is a *postcondition* [10] of true w.r.t. to the statement s.

LEMMA 1. **[Invariant Generation [18]]** *Let $P = (T, \Gamma_0, s)$ be a program and (T, E, ϕ) be a predicated subtype hierarchy. Let Σ (resp. M) be an arbitrary store (resp. memory) such that* $\mathsf{WF}(\Sigma, M, \Gamma_0)$. *Let Σ', M', W' be such that $(\Sigma, M, \Gamma_0; s) \rightarrow^* (\Sigma', M', W'; \mathtt{skip})$. If $\Gamma_0, \phi, \mathit{true} \vdash s \rhd I$ then $\Sigma', M', W' \models I$.*

The proof of the soundness theorem follows by using induction on the number of steps of the execution to show that there are no unsafe downcasts. Recall that each structure that is created is of a leaf type. Consider the first step at which an unsafe downcast occurs. Prior to this step, the structure must have been upcast several times to an (ancestor) supertype that is not a leaf type. The type system ensures, that for each upcast (a) the edge predicate for the upcast holds (via the lemma), and, (b) that the fields in the edge predicate were not modified subsequent to the upcast. Whenever a downcast from t to t' is performed, the disjointness of the edge predicates ensures that only one of the immediate subtypes' edge predicates is satisfied. As the fields in the edge predicate $\phi(t, t')$ were not modified after the structure was previously upcast, it must have been that at the *most recent* upcast $\phi(t, t')$ was true, in other words, the instance being upcast was in fact of type t'. Thus, as at the downcast point, our type system ensures (using the lemma), that $\phi(t, t')$ still holds, the structure can indeed be safely downcast to t'.

EXAMPLE 1: We will show that the downcast at line 5 in Figure 1 is safe. The type declaration Γ_0 maps `icp`, `type` and `dest` to `packet`, `int` and `int` respectively. At statement 3, the assignment and variable access rules are applied, and the casts are trivial, so no implication checks are done, but the invariant I becomes $\mathtt{icp.icmp\_type} = \mathit{type}$. We apply the conditional rule for statement 4; inside the then branch, I_4 is $(\mathtt{icp.icmp\_type} = \mathtt{type}) \wedge (\mathtt{type} = 5)$. In statement 5, there is a downcast from `packet` to `redirect`). To check the safety of this downcast, we use the Var-Down rule on `icp` using the access predicate obtained from I_4 by substituting `icp` with `this`, and check the validity of:

$$(\mathtt{this.icmp\_type} = \mathtt{type} \wedge \mathtt{type} = 5) \Rightarrow (\mathtt{this.icmp\_type} = 5)$$

As the theorem prover tells us this implication is valid, we conclude that the downcast at 5 is safe. □

5. 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 downcasts 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.

5.1 Generating Predicate Constraints

We use the syntax-directed typing rules of Figure 4 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 $(t, \cdot)$. We start with an initial map corresponding to the empty set for all types t. Next, we do the type checking without making the implication queries at the casts (as there is no ϕ). Instead, at each occurrence of the Field-Assign rule (Figure 4), we add the field l to $\overline{\mathsf{tag}}(t'')$, for all $t'' \preceq t'$, *i.e.,* all subtypes of t'. Intuitively, the assignment implies that no such field can be used to distinguish which of the subtypes of t' the structure is at runtime. At the end of this phase, all the fields that cannot be tag fields of t are in the set $\overline{\mathsf{tag}}(t)$.

In the second phase, we make another syntax-directed pass using the 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, Field-Down, Var-Up and Field-Up (Figure 4) 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, as we shall see, 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.

Recall that by our assumption that the only upcasts and downcasts in the program are between immediate subtypes. Thus, the

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

$$\frac{\Gamma, \phi, I \vdash_l lv : s\{\cdot, (l,t), \cdot\} \qquad t' \prec t \qquad \boxed{I[\texttt{this}/lv] \Rightarrow \phi(t,t')}}{\Gamma, \phi, I \vdash_l (t')lv.l : t'} \textsf{Field-Down}$$

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

$$\frac{\Gamma, \phi, I \vdash_l lv : s\{\cdot, (l,t'), \cdot\} \qquad t' \prec t \qquad \boxed{I[\texttt{this}/lv] \Rightarrow \phi(t,t')}}{\Gamma, \phi, I \vdash_l (t)lv.l : t} \textsf{Field-Up} \qquad \frac{\Gamma, \phi, I \vdash_l lv : t}{\Gamma, \phi, I \vdash_l (t)lv : t} \textsf{Lval-Eq}$$

(a) Lvalues

$$\frac{}{\Gamma, \phi, I \vdash_e \textsf{New}(t) : t} \textsf{New} \qquad \frac{\Gamma, \phi, I \vdash_l lv : t}{\Gamma, \phi, I \vdash_e lv : t} \textsf{Lval}$$

$$\frac{\Gamma, \phi, I \vdash_e e_1 : \texttt{int} \qquad \Gamma, \phi, I \vdash_e e_2 : \texttt{int}}{\Gamma, \phi, I \vdash_e e_1 \oplus e_2 : \texttt{int}} \textsf{Aop}$$

$$\frac{\Gamma, \phi, I \vdash_e e_1 : t \qquad \Gamma, \phi, I \vdash_e e_2 : t}{\Gamma, \phi, I \vdash_e e_1 = e_2 : \texttt{bool}} \textsf{Eq}$$

$$\frac{\Gamma, \phi, I \vdash_e e_1 : \texttt{int} \qquad \Gamma, \phi, I \vdash_e e_2 : \texttt{int}}{\Gamma, \phi, I \vdash_e e_1 \sim e_2 : \texttt{bool}} \textsf{Acomp}$$

(b) Expressions and booleans

$$\frac{}{\Gamma, \phi, I \vdash \texttt{skip} \rhd I} \textsf{Skip}$$

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

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

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

$$\frac{\Gamma, \phi, I \vdash s \rhd I' \qquad \Gamma, \phi, I' \vdash s' \rhd I''}{\Gamma, \phi, I \vdash s; s' \rhd I''} \textsf{Seq}$$

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

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

(c) Statements

Figure 4. 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.

constraint generation introduces predicate constraints for $\pi_{t,t'}$ for edges $(t, t') \in E$.

EXAMPLE 2: The constraint generated from the downcast on line 05 in Figure 1(a) is:

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

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

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

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

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

Solutions. A *solution* to a set of constraints $\textsf{Cons}(P)$ 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 $\textsf{Cons}(P)$, the implication $\psi_s \Rightarrow \Pi(\pi_{t,t'})$ is valid, and,
- **S4** For each t, t', the predicate $\Pi(\pi_{t,t'})$ should not contain any field name in $\overline{\textsf{tag}}(t)$.

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

THEOREM 2. **[Soundness of Constraint Generation]** *For every program $P \equiv (T, \Gamma_0, s)$, if Π is a solution for the constraints $\textsf{Cons}(P)$ then*

$$\phi \equiv \lambda(t,t').\Pi(\pi_{t,t'})$$

is such that: $\Gamma_0, \phi, true \vdash s \rhd \cdot$.

The theorem follows by observing that **S1**, **S2**, and **S4** enforce that requirements **R1, R2** and **R4** of the edge predicate map ϕ are met, and condition **S3** ensures **R3**, *i.e.,* that the resulting ϕ is such that the implications required for type checking are valid. The predicated type system we have described does not have a principal typing property. Consider a structure with a tag field t and two physical subtypes with fields f_1 and f_2. If f_1 is accessed only when $t = 0$, and f_2 is accessed only when $t = 5$, then both $(t \leq 2, t > 2)$ and $(t \leq 1, t > 1)$ are predicate refinements, but neither subsumes the other.

5.2 Solving Predicate Constraints

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

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

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 $\textsf{Cons}(P)$ 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.

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 we can infer a solution to the constraints, and thus a predicated subtype hierarchy that suffices to type check the program. We construct the solution using a variant of Craig Interpolation [5]. Recall from Section 3.2, that 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 $\hat{A}_1, \ldots, \hat{A}_n \equiv \mathsf{ITP}(A_1, \ldots, A_n)$ satisfying conditions **I1**,**I2**, and **I3**.

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 2, 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 linear in the number of constraints, and thus, the program, and makes a linear (in the size of T) calls to an interpolating decision procedure.

Algorithm 1 PredTypeInference

Input: Program $P = (T, \Gamma_0, s)$
Output: Refinement (T, E, ϕ) 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, ϕ)

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

THEOREM 3. **[Correctness of Type Inference]** *For every program* $P \equiv (T, \Gamma_0, s)$, $\mathsf{PredTypeInference}(P)$ *terminates. If* $\mathsf{PredTypeInference}(P)$ *returns* (T, E, ϕ) *then* $\Gamma_0, \phi, true \vdash s \rhd \cdot$. *If* $\mathsf{PredTypeInference}(P)$ *returns* ERROR *then there is no* ϕ *such that* $\Gamma_0, \phi, true \vdash s \rhd \cdot$.

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

$\texttt{type}_{05} = \texttt{this.icmp\_type} \wedge \texttt{type}_{05} = 5$

$\texttt{type}_{09} = \texttt{this.icmp\_type} \wedge \texttt{type}_{09} \neq 5 \wedge \texttt{type}_{09} = 12$

$\texttt{type}_{12} = \texttt{this.icmp\_type} \wedge \texttt{type}_{12} \neq 5 \wedge \texttt{type}_{12} \neq 12 \wedge \texttt{type}_{12} = 3$

corresponding to $\psi(\texttt{packet}, \texttt{redirect})$, $\psi(\texttt{packet}, \texttt{unreach})$ and $\psi(\texttt{packet}, \texttt{param})$, respectively. Note that 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 $\texttt{this.type} = 3$ respectively. □

6. Implementation and Experiences

We have implemented the predicated type inference algorithm for C. Our tool uses CIL [23] to parse and manipulate the C program, and the theorem prover FOCI [22] to generate interpolants and check implications at cast locations. Our tool follows Algorithm 1 and focuses on the safety of union accesses and explicit casts on fields. Also, our tool extends the invariant generation to handle pointers and functions. Further, although our type system mimics a common C idiom, there are cases where this idiom is not followed. Our system gracefully handles these issues by giving a best effort to generate predicate edges when programmers follow the idiom and by identifying cases where programmers do not follow the idiom.

6.1 Implementation Issues

We create a subtyping hierarchy to 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. We shall informally describe our technique. For pointers, we run an flow-insensitive aliasing analysis and replace all pointer accesses with abstract locations guarded by predicates *i.e.,* if a pointer `p` may point to `a` or `b`, we replace `*p = 5` with `if (p = &a) a = 5 else if (p = &b) b = 5`. In Figure 4, the typing rules (Var-Assign, Field-Assign, Assign-Φ, If, While) conjoin predicates from the start of the surrounding function. However, some functions that access a specific union field are only called when a predicate check has been performed at the call site. We have a parameter that specifies the depth of the disjunction of invariants derived from all callsites. For functions being called in the surrounding function, we inline those functions based on another depth parameter.

Resolving Conflicts. We say that two cast instances *conflict* if they are casts from the same supertype to different subtypes, such that the conjunction of the access predicates at the two locations are satisfiable. As discussed in Section 5, if this happens, our algorithm cannot find a predicated refinement of the subtype hierarchy that suffices to check the program is correct. Conflicts arise either because there is no predicated refinement for that specific cast, or because our invariant generation is too weak.

In order to get useful results even in the presence of conflicts, we extend the algorithm for inference described in Section 5, by using heuristics to greedily restrict the set of access predicates to those that do not conflict with downcasts to different subtypes. We found in our experiments that despite conflicts, out of a total of 90 edge predicates that could be discerned by a close manual inspection, our implementation was able to infer 80 predicates correctly (*i.e.,* 89% of the edge predicates were correctly inferred). Whenever our method found an edge predicate, it was, in all cases but one, exactly the one that was revealed by close manual inspection.

Program	LOC	Predicate Edges		Accesses		Conflicts	Time
		Inferred	Actual	Predicated	Other		
ip_icmp	7K	7	7	15	7	1	1s
xl	12K	8	8	428	0	200	875s
moapsource	14K	3	3	5	6	2	1s
gdkevent	16K	12	13	90	5	31	38s
lua	18K	13	15	274	8	130	151s
snort	42K	7	7	26	120	0	12s
sendmail	106K	17	24	406	17	138	995s
ssh	35K	0	0	0	2105	0	12s
bash	101K	13	13	440	3914	162	1157s
Total	351K	80	90	1684	6182	664	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** gives the number of syntactic cast points in the program. Among these, **Predicated** is the number of predicated accesses and **Other** is the number of other accesses. **Conflicts** is the number of conflicts.

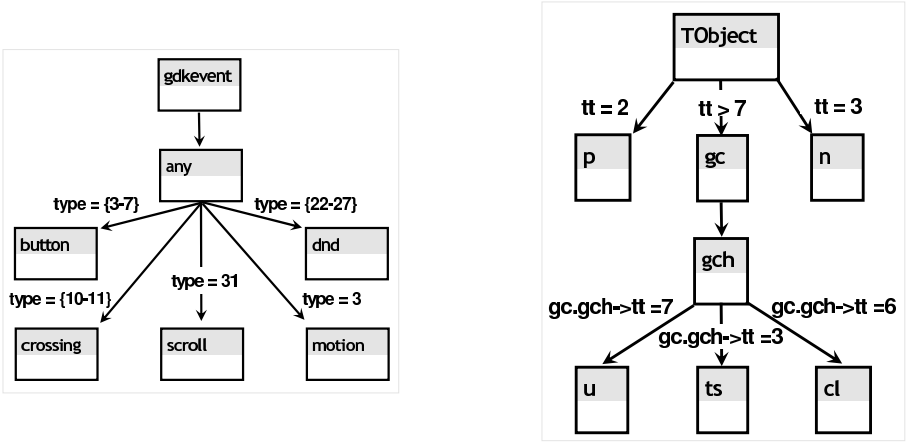


Figure 5. Predicate subtype hierarchy for (a)gdkevent (b)lua

We also have heuristics identifying downcast edges that do not have predicates. For the case where programmers do not use downcast edges with predicates, our tool identified all 131 such edges. All these edges were manually confirmed to have no edge predicates. Our results show that by identifying these conflicts and by not including those access predicates in our inference, we infer the majority of correct edge predicates. The heuristics to identify conflicts are described in Section 6.2 and a more detailed breakdown of conflicts is shown in Section 6.3.

6.2 Experimental Results

We have used our tool to investigate nine open source programs. `ip_icmp` is the ICMP implementation in the FreeBSD kernel. `moapsource` is the packet processing code of Emstar, a sensor network development tool. `gdkevent` is how events are encoded in the GDK graphics library. `lua` is a dynamically typed language interpreter. `snort` is an opensource intrusion detection tool. `xl` is a small lisp interpreter from the SPEC benchmarks. `ssh` is the widely used secure shell client. `bash` is the Bourne Again shell. `sendmail` is the Sendmail email server.

We summarize our results in Table 1. Our experiments were all run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. All experiments except for `moapsource` were run without inlining functions. `moapsource` required looking at one level of callers to generate sufficiently precise invariants. 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`.

The only case involving explicit predicated casts occurred in `moapsource`. Packets are encapsulated within other packets. These packets contain a header that contains an 8 bit field that identifies the next header type. The next header is explicitly cast to the right header subtype.

Our tool can derive complex predicated subtyping hierarchies. Figure 5 shows the two partial predicated subtyping hierarchies for `gdkevent` and `lua`. Some subtypes are dropped because of space. The subtypes shown are representative of the output of our tool. We show that predicate edges are not simply single tag assignments but rather more complex predicates involving ranges.

In `gdkEvent`, checking the type of an event can be done by looking at the `type` field in the `GdkEventAny` union field or the `type` union field. In Figure 5(a) we use the `type` union field to distinguish accesses. Our tool handles this by knowing that the `type` field in the `GdkEventAny` and the `type` union field represent the same location in memory. Edges are disjunctions of tag values,

i.e. to access the `crossing` field, the `type` field must be 10 or 11. Also, just an $=$ operator is not enough as `lua` requires the $\geq$ operator as seen from the access of the `gc` field.

Conflicts and Bugs. Recall there are two sources of conflicts: downcasts on edges for which no predicate could be inferred, and downcasts on edges where a predicate was inferred, but where the access predicate invariant was not strong enough to establish the edge predicate, either because the inferred invariant was too weak, or because there is a bug.

For the first case, we used the following heuristic to determine which edges have no predicates. Given a type node, if *all* downcast instances of an outgoing downcast edge conflict with all downcast instance of any other outgoing cast edge, then it is likely that type node has no predicated downcasts. Using this heuristic, our tool inferred 84 unions types and explicit casts of 47 different fields corresponding to the 6,182 accesses as not having downcast predicate edges. We then manually confirmed that for every one of these edges, there was no downcast predicate.

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. To facilitate faster identification of possible bugs, the downcasts that conflict the most with others are presented first. In such cases, the programmer forgot 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. A union field in one of those variables is accessed after the appropriate predicate is checked for that `struct`. However, the same union field in the other variable is also accessed without checking if the appropriate predicate holds on that `struct` as well. In these 23 cases, there is an assumption that two different variables always have the same predicate hold. Although our tool presents conflicting statements, we have not fully investigated this tool as a bug finder.

6.3 Limitations

Table 1 also summarizes how often predicated guarded accesses and downcasts are used. Although we found that 1,633 casts corresponding to union field accesses and 46 on explicit casts can be described by our type system, there were 6,182 access that did not follow our idiom. The majority of these accesses (6,018 of them) occurs when unions are used to simplify memory access. Our simple heuristic from the last section correctly distinguishes all such accesses from predicated casts. The remaining 164 accesses either required a predicate over other lvalues as opposed to just fields of the downcasted lvalue, or required stronger invariants than the ones generated by our type system.

1. Overlapping Layout. For 6,018 conflicting accesses, programmers used unions to easily access memory locations. For example, in the header file `in6.h` (used in `icmp` and `ssh`) we saw the following structure:

```
struct in6_addr {
    union { uint8_t  u6_addr8[16];
            uint16_t u6_addr16[8];
            uint32_t u6_addr32[4];
    } in6_u;
  };
```

In this case, the union is just a sequence of several bytes and the union fields are used to access bytes at different offsets safely.

Memsets. The `memset` macro is a more complex example of this. 3,886 union accesses involved the `memset` macro. `Memset` is a widely used macro that takes three parameters n, c, s, and sets n bytes to some character c starting at memory address s. The macro is implemented by the use of an anonymous union. Each use of `memset` causes a new anonymous union, leading to 3,886 non-predicated downcasts in `bash`.

2. External Correlation. Some union field accesses or explicit casts depend on a wide range of variables, limiting the predicate over fields of a single data structure is not sufficient. We found a few cases where the union accesses were safe, but the reason for safety could not be expressed within our type system. For 3 union fields, we found that there were edge predicates for the union that used variables belonging to a different structure, and thus could not be expressed using `this`. We believe that most of the 20 downcast predicate edges involving explicit casts fall in this category. In future work we intend to look at ways to infer this external correlation and "pack" the other structure together with the structure containing the downcasted field in order to be able to extend our technique to this setting. In the `bh` benchmark from the Olden suite (not shown in the Table), we found that the reason for safety depended not on other fields, but on the *control* location. In different phases of execution, the program was in different locations, and the value in the union is directly correlated to the phase and hence the program location. In `ssh` we found a variant of this, where there were many conflicts involving the `ciphercontext` union, which includes different types of ciphers as fields. Different encryption and decryption functions choose the appropriate element of the union without establishing any edge predicate because an external global data structure ensures that the function pointers corresponding to the encryption and decryption functions are correlated with the field inside the ciphertext union.

3. Operation Ordering. In our imperative language, **S4** specifies that predicates must be over fields that have not been modified. In C programs, this constraint is relaxed. Fields in predicates are modified when a data structure is being created. These fields are usually sequentially written to. Our type system assumes that the assignments to the tag fields (*i.e.,* the fields appearing in the edge predicate) are done, thus establishing the edge predicates, *before* the structure is cast. However, in the example of Figure 1, the programmer may create a `redirect` message, and assign to the `ih_gwaddr`, and *after* that, write the appropriate value in the tag field `icmp_type` indicating which union subtype the structure had. Thus, when the `ih_gwaddr` field was accessed, the edge predicate did not hold as it was established afterward, thus leading to a spurious conflict. Such code is easily modified to make it pass our type checker, simply by moving the tag assignment.

4. Invariant Imprecision. Many conflicts arose because even though we inferred the correct edge predicates, the algorithm described that accumulates the dominating statements as invariants did not generate invariants strong enough to imply the edge predicates. Though the presented algorithm is fast and scalable, many complexities in the program cause it to not always be precise enough. These include aliasing, the need for loop invariants, dynamic data structures, and interprocedural reasoning. For the few cases where our invariants are too weak, more precise safety analyzers such as BLAST [17], ESC [13], and TVLA [26] 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 [24, 12, 21].

7. Conclusions

We have presented a dependent type system for the verification of union accesses, and an algorithm based on Craig Interpolation to infer the dependent refinements, thereby yielding a fully automatic way to check the safety of union accesses. Our type system captures the common idiom where the programmer checks tag fields before accessing data in a union, and is able to infer the tag fields as well as the protocol used by the programmer to ensure safety. Though

we could instantiate our approach with more advanced invariant generation techniques like abstract interpretation (to compute the cast predicates), our experiments demonstrate that for most cases, our simple syntax-directed invariants suffice to determine how the programmer ensures the safety of union accesses, almost always inferring exactly the same type as that found by a close manual inspection of the code. In places where the inferred edge predicate cannot be statically proven to hold, we can use the edge predicate to insert runtime checks, without modifying the program to insert fields carrying type information. In future work, we shall generalize the technique with parameterized invariant generation schemes, consider more expressive dependent type formalisms that let us capture external correlations, and extend our predicate subtype approach to statically verify the safety of other kinds of downcasts, for example in Object Oriented code.

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

References

[1] A. Aiken, E. Wimmers and T.K. Lakshmam. Soft typing with conditional types. In *POPL 94*, pp. 163–173. ACM, 1994

[2] S. Artzi and M.D. Ernst. Using predicate fields in a highly flexible industrial control system. In *OOPSLA 05*, pp. 319–330. 2005.

[3] T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02*, pp. 1–3. ACM, 2002.

[4] S. Chandra and T. Reps. Physical type checking for c. In *PASTE 99*, pp. 66–75. ACM, 1999.

[5] W. Craig. Linear reasoning. *J. Symbolic Logic*, 22:250–268, 1957.

[6] S. Cui, K. Donnelly, and H. Xi. ATS: A language that combines programming with theorem proving. In *FroCos 05*, LNCS 3717, pp. 310–320. Springer, 2005.

[7] 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.

[8] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02*, pp. 57–68. ACM, 2002.

[9] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[10] E.W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976

[11] Y. Fang. *Translation validation of optimizing compilers*. PhD thesis, New York University, 2005.

[12] C. Flanagan. Hybrid type checking. In *POPL 06*, pp. 245–256. ACM Press, 2006.

[13] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02*, pp. 234–245. ACM, 2002.

[14] J.S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI 02*, pp. 1–12. ACM, 2002.

[15] M. Furr and J. Foster. Checking type safety of foreign function calls. In *PLDI 05*, pp. 62–72. ACM, 2005.

[16] M. Harren and G.C. Necula. Using dependent types to certify the safety of assembly code. In *SAS 05*, LNCS 3672, pp. 155–170. Springer, 2005.

[17] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02*, pp. 58–70. ACM, 2002.

[18] R. Jhala, R. Majumdar, and R. Xu. Structural Invariants In *SAS 06*, Springer, 2006.

[19] T. Jim, J.G. Morrisett, D. Grossman, M.W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Usenix*, pp. 257–288. 2002.

[20] R. Komondoor, G. Ramalingam, S. Chandra, and J. Fields. Dependent Types for Program Understanding. In *TACAS 05*, LNCS 3440, pp. 157–173. Springer, 2005.

[21] A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *FASE 01*, LNCS 2029, pp. 217–232. Springer, 2001.

[22] K.L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[23] 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*, LNCS 2304, pp. 213–228. Springer, 2002.

[24] 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.

[25] J.M. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE TSE*, 24(9):709–720, 1998.

[26] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[27] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *ESEC/FSE 99*, pp. 180–198. ACM, 1999.

[28] H. Xi. Imperative programming with dependent types. In *LICS 00*, pp. 375–387. IEEE, 2000.

[29] H. Xi and R. Harper. A dependently typed assembly language. In *ICFP 01*, pp. 169–180. ACM, 2001.

[30] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL 99*, pp. 214–227. ACM, 1999.