Physical Type Checking for C

Satish Chandra
Software Production Research Dept.
Bell Laboratories, Lucent Technologies
chandra@research.bell-labs.com

Thomas Reps
Computer Sciences Dept.
University of Wisconsin-Madison
reps@cs.wisc.edu

Abstract

The effectiveness of traditional type checking in C is limited by the presence of type conversions using type casts. Because the C standard allows arbitrary type conversions between pointer types, neither C compilers, nor tools such as *lint*, can guarantee type safety in the presence of such type conversions. In particular, by using casts involving pointers to structures (C `struct`s), a programmer can interpret any memory region to be of any desired type, further compromising C's weak type system. Not only do type casts make a program vulnerable to type errors, they hinder program comprehension and maintenance by creating latent dependencies between seemingly independent pieces of code.

To address these problems, we have developed a stronger form of type checking for C programs, called *physical type checking*. Physical type checking takes into account the layout of C `struct` fields in memory. This paper describes an inference-based physical type checking algorithm. Our algorithm can be used to perform static safety checks, as well as compute useful information for software engineering applications.

1 Introduction

In C, a pointer of a given type can be *cast* into any other pointer type. Because of this, a programmer can interpret any region of memory to be of any type. Traditional type checking for C cannot enforce that such reinterpretation of memory is done in a meaningful way, because the C standard allows arbitrary type conversions between pointer types. For this reason, C compilers and tools such as *lint* do not provide any warnings against potential runtime errors arising from the use of casts. We motivate the problem of type safety in C programs using the following two examples.

Example 1. Consider the code fragment in Figure 1. Because of the cast `(ColorPoint*)`, a structure of type `Point` can be interpreted as a structure of type `ColorPoint`. If the sole dereference of pointer `pcp` in the program were `pcp->x`, the program would work correctly, because the `x` field is present in both structures at the same offset. However, the dereference `pcp->color` can cause unexpected behavior. Neither *cc* nor *lint* issues a warning for this program. An overly conservative type checker for C could disallow the cast from a value of type `Point*` to a `ColorPoint*`, regardless of whether the field `color` is dereferenced from `pcp`.

Unfortunately, C programs contain casts with surprising frequency, particularly in systems software (see Table 1), and a type checker that disallows all casts would, in practice, outlaw too many programs.

```
typedef struct {
  int x,y;
} Point;

typedef struct {
  int x, y, color;
} ColorPoint;
```

```
main() {
 Point p;
 ColorPoint *pcp;

 pcp = (ColorPoint *)&p;
 pcp->x = 1;
 pcp->color = RED;
}
```

Figure 1: A program to illustrate problem with type casts.

Example 2. Consider the code fragment in Figure 2. In this code, the cast `(Radio*)` converts a type `Clock*` (the type of `c + 1` is the same as the type of `c`) to type `Radio*`. This program would be declared unsafe by a conservative type checker that rejects all casts. At first glance, this decision appears to be reasonable, because the cast seems to make no sense. However, notice that `c` points to the first field of the structure `ClockRadio`. Because of C's pointer-arithmetic rules, the expression `c + 1` yields the address of—i.e., a pointer to—the beginning of the second field of `ClockRadio`. A pointer to this field can correctly be dereferenced for a `frequency` field, as is done in this example. This program relies on the fact that `c` points to a region in memory that contains a `Clock` structure followed by a `Radio` structure (and the fact that the size of `Clock` is such that no padding is required between fields `clock` and `radio`). Although this usage appears contrived, we have found it used frequently in production code (see [15]).

These examples show that C programmers implicitly rely on the physical layout of `struct`s in memory. This makes type checking difficult: a type checker that is based only on manifest types in the program would either be too conservative, or, like a C compiler, would permit potential run-time errors to go undetected.

```
typedef struct {
  int hour, minute;
} Clock;

typedef struct {
  double frequency;
} Radio;

typedef struct {
  Clock clock;
  Radio radio;
} ClockRadio;
```

```
main() {
  ClockRadio cr;
  Clock *c;
  Radio *r;

  c = &(cr.clock);
  r = (Radio *)(c + 1);
  r->frequency = 91.5;
}
```

Figure 2: An example of the "+1" casting idiom.

The use of type casts can also make programs difficult to understand and extremely fragile to modify. Casts involving pointers to `structs` introduce hidden dependences between types. That is, in the presence of casts involving pointers to `structs` it is not safe for a programmer to simply add a new field to a `struct` because the code may rely on the memory layout of the original `struct` definition. Therefore, automatic techniques to analyze physical layouts of data are important also for program comprehension and maintenance.

Program	kLOC	Void-Struct	Struct-Struct
binutils	516	1109	188
xemacs	288	1662	70
gcc	208	410	137
telephone	110	126	430
bash	76	123	47
vortex	67	592	50
ijpeg	31	74	601
perl	27	101	15
xkernel	37	1882	179
Total	1360	6079	1717

Table 1: Count of casts in a suite of C programs [15]—SPEC95 benchmarks *gcc*, *ijpeg*, *perl*, *vortex*, GNU utilities *bash*, *binutils*, *xemacs*, networking code *xkernel*, and portions from a Lucent Technologies's product code *telephone*. kLOC is the number of source lines (in thousands). The Void-Struct column gives the total number of casts in which a `void*` was converted to a pointer to a `struct` (or vice versa), and the Struct-Struct column gives the number of casts in which both the types involved were pointers to `structs`. These numbers include both implicit and explicit casts.

Physical Type Checking. In this paper, we develop a new form of type checking for C programs, called *physical type checking*, that is based on the physical layout of `structs` in memory. The goal of physical type checking is to provide static safety checks for pointer dereferences in a program. A program that passes these static safety checks is declared to be *physically type safe*.

In Example 1, our type-checking algorithm would declare the program to be unsafe, because the declared type of `p`, `Point`, does not have a field `color`, which is required because of the dereference `pcp->color`. In Example 2, our algorithm would declare the program to be type safe, because the requirements on the structure `cr` are satisfied by its declared type.

Physical type checking is carried out by a flow-insensitive, context-insensitive, interprocedural analysis algorithm. In terms of how the approach relates to previous work, the most significant aspects are as follows:

- The physical-type-checking algorithm is cast as a type-inference problem: For the most part, the analysis ignores the declared types in a program, and relies on inference to compute a "required" type for each variable in the program whose address is taken. In recent years, a number of other papers have also used type inference as a mechanism for specifying flow-insensitive analyses (see Section 5).
- There are some similarities between the physical-type-checking algorithm and previous work on flow-insensitive points-to analysis [1, 18, 14, 21]. The relationship between physical type checking and points-to analysis is addressed in Section 4.3.

Physical type checking can be useful for a number of purposes. Its most obvious application is to discover potential *physical type errors* caused by inconsistent type interpretations of memory. Just as a tool such as *LClint* [4] statically identifies certain classes of errors in programs, physical type checking detects another class of errors that traditional C type checking misses. Physical type checking also has a number of applications in software-engineering tools: The information obtained can help a programmer to understand the ways type casts are used in programs, to uncover hidden dependences between different C types, and to retrofit more stringent type declarations to variables or function arguments.

Contributions. The starting point for our work is the observation that C programs that use type casts require type checking that is more powerful than the standard type-checking system for C (which is based only on variables' declared types). Our contributions are as follows:

- We give an inference-based algorithm to perform such type checking for C programs. (An implementation of the algorithm that uses an off-the-shelf constraint solver is described in [3].)
- In previous work, we had introduced a notion of *physical subtyping* between `struct` types. However, this work did not handle pointer fields inside structures. In this paper, we introduce a way to handle subtyping in the presence of pointer fields.

Outline. The remainder of the paper is organized as follows: Section 2 reviews some ideas from our previous work [15] that we draw upon in later sections. Section 3 presents the basic approach behind our physical-type-checking algorithm, and describes certain problems with pointers that hinder our approach. Section 4 presents the actual algorithm and shows how it solves these problems. Section 5 discusses related work. Due to space constraints, much detail has been left out from this paper, but is covered in a more complete technical report [3].

2 Preliminaries

In this section, we define what we mean by physical type safety, and review the notion of physical subtyping, which was introduced in [15].

It is useful to define a number of auxiliary functions:
- $stype(e)$ is the compiler-assigned type of a C expression e;
- $sizeof(t)$ is the standard C `sizeof` function for type t;
- $offset(t, x)$ is the offset of field x in `struct` or `union` type t.

We assume that the statements in an input program have been normalized to consist of only a few simple forms, shown in Table 2. The purpose of normalization is to limit the number of cases our analysis must consider. A procedure

Address-Of	$p = \&x$
Assignment	$p = cast_{opt}\ q$
Pointer Dereference on rhs	$p = *q$
Pointer Dereference on lhs	$*p = q$
Field Dereference on lhs	$p \to a = q$
Field Dereference on rhs	$p = q \to a$
Field Address	$p = \&(q \to a)$
Plus One	$p = q + 1$
Other Arithmetic	$p = q \text{ op } k$

Table 2: Statements in the normal form.

to convert a program into this normal form may need to introduce temporary variables. We also assume that all assignment statements copy values only of ground types, i.e., `struct` copies are transformed to element-wise copies. Notice that a cast may appear only in the *Assignment* statement in the normal form. Such casts are only needed if we want the normalized program to be a legal C program; in all other respects, casts do not have any significance at all in the physical-type-checking algorithm. The declared types of variables are used only to compute the appropriate offset corresponding to a field dereference. We assume that targets of indirect function calls have been precomputed, e.g., by using a points-to analysis on function pointers.

2.1 Physical Type Safety

Intuitively,[1] to be physically type safe, each pointer dereference should point to "valid memory", and refer to a "valid type". By valid memory, we mean that the address computed for the load of the specified field from memory must be within the bounds of the allocation unit that the pointer currently points to. (Each stack-allocated variable constitutes an allocation unit, as does a chunk of memory returned by `malloc`.) By valid type, we mean that the ground type being referred to must be the same as the one stored at the memory location.

For example, in Figure 1 in the previous section, a dereference of `color` from `pcp` is not physically type safe, because the field `color` lies outside the valid memory of a `Point` variable. Suppose we cast `&p` to a pointer to `struct { int x; float y; }`. We would consider a dereference of the `y` field as unsafe, because the ground type being referred to (i.e., `float`) does not match the type stored at the memory location (i.e., `int`), although the address of the field does lie within the allocation unit, i.e., refers to valid memory.

Note that physical type safety is still not a guarantee of absence of runtime errors. For example, it says nothing about errors related to management of heap storage and to the bounds of array references.

[1] For a precise definition, based on a runtime notion of physical type correctness, see [3].

2.2 Physical Subtyping

A cast-free C program that type checks (and that does not use `union`s inconsistently) will be physically type safe. However, many C programmers find it useful to use casts, and this motivated us to find alternative conditions under which physical type safety could be guaranteed. A key concept in defining such conditions is physical subtyping. The idea behind physical subtyping is that a value of one type t may be operated upon as if it had another type t', if in the memory layout of the two types, the values stored in corresponding locations "make sense". Consider the following code:

```
Point pt; ColorPoint cp;

pt.x = 3;  pt.y = 41;
cp.x = 5;  pt.y = 17;  pt.color = RED;
```

A picture of how `pt` and `cp` are represented in memory might look like:

pt	3	41	
cp	5	17	RED

`cp` can be thought of as being of the same type as `pt` simply by *ignoring its last field.*

Figure 3 presents rules for inferring that a C type t is a physical subtype of C type t' (denoted by $t \preccurlyeq t'$). A `struct` containing k members is denoted by $s\{m_1, \ldots, m_k\}$. Members are assumed to carry a type and an offset from the beginning of the structure. This offset computation is platform-dependent, as the layout of the fields of a `struct` or a `union` depends on the compiler and the architecture. Currently, we do not handle bit fields in our type system. (In order to handle bit fields, the type system could be augmented to carry around a more elaborate notion of offsets.)

[Reflexivity] $$\overline{t \preccurlyeq t}$$

[Void Pointers] $$\overline{\texttt{t ptr} \preccurlyeq \texttt{void*}}$$

[First members] $$\frac{t \preccurlyeq t' \quad m_1 = (l, t, 0)}{s\{m_1, \ldots, m_k\} \preccurlyeq t'}$$

[Structures] $$\frac{k' \leq k \quad m_1 \preccurlyeq m_1' \ldots m_{k'} \preccurlyeq m_{k'}'}{s\{m_1, \ldots, m_k\} \preccurlyeq s\{m_1', \ldots, m_{k'}'\}}$$

[Members] $$\frac{m = (l, t, i) \quad m' = (l', t', i') \quad l = l' \quad i = i' \quad t \preccurlyeq t'}{m \preccurlyeq m'}$$

Figure 3: Inference rules for physical subtypes.

The rule for structures attempts to match up one structure as a prefix of another structure. (In practice, we use certain relaxations in matching `struct` types.) Note that there is no rule for physical subtyping that involves comparing two `union` types. This is because with `union`s, determining which branch within a `union` is the active one is an orthogonal problem. Note also that we have no rule for comparing two pointer types, except that we consider any pointer type to be a physical subtype of `void*`. In [15], we report on how these subtyping rules "explain" several patterns of cast usage in C programs. In particular, casts in C programs are often used to simulate an object-oriented style (e.g., inheritance).

3 Physical Type Checking

Our type-checking algorithm works by performing a *backwards* propagation of type requirements to the program points at which a memory address is created and bound to a pointer variable. We first describe and provide the rationale for a new domain of types that is used in our type-checking system. We then describe the main ideas behind our physical type-checking algorithm. This subsection essentially presents a type-checking algorithm that works on a restricted subset of programs. Finally, we describe the difficulties we face when trying to extend the algorithm for general programs.

In Section 4, we will present the complete type-checking algorithm that works on unrestricted programs. We split the presentation into two sections, because we wish to first present the essence of physical type checking in a somewhat simplified setting in which we do not have to face up to the full range of complications that pointers cause us.

3.1 Type Obligations

We define a domain T of *type obligations* as follows:

$$\begin{array}{rcl} T & = & \{t_1, \ldots, t_n\} \\ & | & \Lambda \end{array}$$

where t_i is a C type. Λ denotes the null type obligation.

We say T is a *type obligation* on a pointer-valued variable p, if in the context provided by the rest of the program, either

- $T = \Lambda$, and p occurs *unconstrained*, i.e., the value of p is not used as an address, or,
- $T = \{t\}$, and dereferences from p would be physically type safe if p points to a `struct` of type t, or any physical subtype of t, or,
- $T = \{t_1, \ldots, t_n\}, n > 1$, and no single C type could guarantee that all dereferences from p would be physically type safe; however, each of $\{t_1\}, \ldots, \{t_n\}$ are type obligations on p.

Example 3. In Example 1, `ColorPoint` is a type obligation on `pcp`, due to the dereferences `pcp->x` and `pcp->color`. (Any physical subtype of `ColorPoint` is also a type obligation, but `ColorPoint` is the smallest one.) Had the only dereference through `pcp` been `pcp->x`, any physical subtype of `struct {int x;}` would be a type obligation.

Type obligations express sufficient, but perhaps more than necessary conditions for physical type safety. In the systems of constraints that we generate, we denote the type obligation on a variable p by $\langle p \rangle$. (Our intention is to find the *least restrictive* type obligation for each $\langle p \rangle$ that satisfies the system of constraints.) We also introduce type obligations for *pointer sources*. The following syntactic occurrences are pointer sources:
– An address-of `&var`
– A call to `malloc` (or other storage allocation primitive)
– An array arithmetic operation `a+i`, where a is an array.
In our constraint system, the type obligation of `&v` is denoted by $\langle \&v \rangle$, of `a+i` by $\langle \&a \rangle$, and of an occurrence of `malloc` at a particular program site s by $\langle malloc_s \rangle$.

We define the domain of type obligations in this manner because sometimes it is not possible to infer that $\langle p \rangle$ is a single C type for each pointer-valued variable p in a procedure, even though the procedure is physically type safe. This can happen for two reasons. First, the use of C `union`s can introduce incompatible type requirements if the dereferences refer to different branches of the `union`. Second, the flow-insensitive nature of our inference algorithm can give rise to spurious type requirements if the same pointer variable is (safely) used to refer to different structures. It may be tempting to infer a C `union` type as the least restrictive type in both of these cases. However, the interpretation of a `union` of types is an "or" of the constituent types: it provides space to store any one of the constituent types, but it may hold only one type at a time. By contrast, the set form $\{t_1, \ldots, t_n\}$ represents an "and" of types, in the sense that it represents a type that can be all of its constituent type at the same time. This gives us a way of recording information about certain kinds of inconsistent types, which is helpful in a tool that reports anomalous usages.

We also define a binary relation $\sqsubseteq$ on type obligations, which captures the fact that, if $A \sqsubseteq B$ and B is a type obligation for p, then A is also a type obligation for p. That is, if the right-hand-side term is sufficient to be $\langle p \rangle$ for some p, then so is the left-hand-side term, by either being a physical subtype of the right-hand-side term, or including more types in the set. The relation $\sqsubseteq$ is reflexive and transitive, and also admits the following "resolution" rules:

$$(\langle e \rangle \sqsubseteq \{t_1\}) \wedge (\langle e \rangle \sqsubseteq \{t_2\}) \wedge (t_1 \preccurlyeq t_2) \Rightarrow (\langle e \rangle \sqsubseteq \{t_1\})$$

$$(\langle e \rangle \sqsubseteq \{t_1\}) \wedge (\langle e \rangle \sqsubseteq \{t_2\}) \wedge (t_1 \not\preccurlyeq t_2) \wedge (t_2 \not\preccurlyeq t_1) \Rightarrow (\langle e \rangle \sqsubseteq \{t_1, t_2\})$$

In the remainder of this paper, we will usually skip the surrounding curly braces when we mention a singleton C type in the $\sqsubseteq$ relation.

3.2 Inference-Based Physical Type Checking

For this subsection, assume that the program being checked does not contain *second-level pointers*. Second-level pointers hold pointers to variables that may themselves be pointers to other variables, or to `struct` variables that may contain a pointer as a member field.

The type-checking algorithm works in two steps. First, it infers a least-restrictive type obligation for each pointer source (i.e., a type obligation that is greatest with respect to $\sqsubseteq$). The second step of the algorithm performs type checking: it checks whether these pointer sources *satisfy* their type obligations. A pointer source p satisfies its type obligation if either of the following two cases hold:

1. $\langle p \rangle \sqsubseteq \{t\}$, and the declared type of the variable associated with the pointer source is a physical subtype of t.
2. $\langle p \rangle \sqsubseteq \{t_1, \ldots, t_n\}$. No declared type can, in fact, be a physical subtype of two incomparable (under $\preccurlyeq$) types. In this case, a user must use other information, such as control flow, to make a determination of physical type safety.

In the second case, if the declared type is a `union`, we could provide an option that assumes a user-provided guarantee that the use of `union` is ANSI conformant, i.e., its branches are active in a mutually-exclusive manner. Under this option, we can check if each t_i has at least one branch in the `union` that is a physical subtype of t_i. (This exploits the property of C `union`s that common prefixes of members are laid out identically.)

By checking that type obligations hold at pointer sources in a program, we can determine whether or not the program is physically type safe. Note that we do not perform any checks at the actual points of dereference, because the type-inference step propagates the "needs" of these dereferences back to the pointer sources. Least-restrictive type obligations are, in essence, *preconditions* for actual points of dereference to be safe.

Generating Type Constraints. The type-inference procedure traverses the abstract syntax tree of the program to generate a set of relations, or *constraints*, on type obligations. It generates a set of constraints using the rules in the second column of Figure 7. (Ignore the third column for this section.) The $\langle e \rangle$ terms appear as "unknowns" in this set of relations—the procedure to "solve" for these terms is described later.

The relation-generation procedure also employs three auxiliary functions for constructing new C types from existing C types. Let t be a C `struct` type:

$$t = s\{m_1, \ldots, m_{k-1}, (t_x, x, i_x), \ldots, m_n\}$$

We define

$$PrefixExclusive(t, x) = s\{m_1, \ldots m_{k-1}\}$$

$$PrefixInclusive(t, x) = s\{m_1, \ldots (t_x, x, i_x)\}$$

$$Concat(t, t_{new}) = s\{t, (t_{new}, l_{new}, i_{new})\}$$

where l_{new} is a new field label and i_{new} is an appropriate offset. The function *PrefixExclusive* forms a new `struct` type that contains all fields in the same order as t up to,[2] but not including field x; the function *PrefixInclusive* is similar except it includes the field x. (These definitions are motivated by our physical subtyping rules.) The function *Concat* augments a `struct` with a new field of the specified type t_{new}. We normally use a type-obligation term, e.g., $\langle e \rangle$, in the second argument of *Concat*. When this is the case, *Concat* results in a set of C types, which is the result of mapping the function defined above onto all the constituent C types of the type-obligation argument.

The constraint-generation rules have the following explanation:

- *Assignment*: This rule propagates the type obligation for the left-hand side of an assignment to the right-hand side. In this sense, the analysis is a *backwards* analysis (albeit flow insensitive) that propagates the "needs" originating from dereferences back to the pointer sources.
- *Address-Of*: This rule is similar to *Assignment*.
- *Pointer Dereference*: For $*q$, the type obligation on q constraints q to point to the (only) valid ground type. Note that, because of our assumption of only single-level pointers, $*q$ itself cannot contain a pointer value.

[2]including the "holes" introduced by padding

```
struct S {
  int a;
  int b;                      struct {
  struct T {                    int a;
    int x;                      int b;
    int y;                      struct {
    int z;                        int x;
  } u;                          } u;
} *p;                         }
struct T *q;
q = &(p->u); q->x = 3;
```

Figure 4: Example to explain the *Field Address* constraint-generation rule. See the text for details.

- *Field Dereference*: For $q \to a$, the type obligation on q should be a `struct` that has space for an a field at the right offset. The constructed type $PrefixInclusive(t, a)$ expresses exactly this criterion. Note that the field $q \to a$ itself cannot contain a pointer value.
- *Field Address*: This rule propagates the type obligation of a pointer into the middle of a structure back to the pointer to the top of the structure. For instance, in Figure 4, `q` is a pointer into the middle of the structure that `p` points to. The type obligation of `q`, based on the dereference `q->x`, is `struct { int x;}`. The type obligation on `p` must capture the fact that the field at the offset $offset(\texttt{struct S}, \texttt{u})$ must satisfy the type obligation of `q`. The constructed type shown on the right in Figure 4 constrains $\langle p \rangle$ accordingly.
- *Plus One*: For arbitrary arithmetic, the algorithm would declare the type obligation on q to be a divergent type ($\top$). However, our algorithm takes a special interest in the case of $q + 1$, because we can track the type obligation precisely. The rational for this rule is much as in the *Field Address* case: the type obligation of an internal pointer in a structure is propagated to a pointer that points to the "previous field" in the same structure. Note that in this rule, the static type of q need not be a pointer to a `struct`.

Solving Constraints. The result of constraint generation is a set of $\sqsubseteq$ relations involving $\langle e \rangle$ terms. We now combine the constraints so we can arrive at the values for $\langle e \rangle$ terms. We use transitivity and the resolution rules mentioned in Section 3.1, until we have only a single relation involving any given $\langle e \rangle$ term on the left-hand side. The r.h.s. value in the final $\langle e \rangle \sqsubseteq T$ is the answer we get for $\langle e \rangle$.[3]

Example 4. Consider again the code in Figure 1. The following constraints are generated for the three statements in the program (using the *Assignment* rule for the first statement and *Field Dereference* for the next two statements):

$$\langle \&p \rangle \sqsubseteq \langle pcp \rangle \quad (1)$$

$$\langle pcp \rangle \sqsubseteq \texttt{struct \{int x\}} \quad (2)$$

$$\langle pcp \rangle \sqsubseteq \texttt{struct \{int x, y, color\}} \quad (3)$$

By transitivity, we obtain

$$\langle \&p \rangle \sqsubseteq \texttt{struct \{int x\}} \quad (4)$$

[3]We have side-stepped certain details such as an occurs-check test in constructed type terms. These implementation issues are covered in the technical report [3].

$$\langle \&p \rangle \;\sqsubseteq\; \texttt{struct \{int x, y, color\}} \tag{5}$$

Since `struct {int x,y,color}` $\preccurlyeq$ `struct {int x,y}`, the simplified constraint for $\langle \&p \rangle$ is also given by (5). Accordingly, the solution for $\langle \&p \rangle$ is `struct {int x, y, color}`. Since the declared type of `p`, `Point`, is not a physical subtype of $\langle \&p \rangle$, the program is unsafe. Notice that the algorithm did not attach any significance to the type cast used in Figure 1. (The technical report also shows how our algorithm works for the code in Figure 2).

An attractive feature of inference-based type checking is that it propagates use information back to the source. This has an obvious advantage from the standpoint of reverse-engineering and program-comprehension applications—one can figure out exactly in which ways a given `struct` variable is used in a program. This information can also be used to refine the type declarations of procedure parameters.

One problem with this method is the use of polymorphic functions in C. For example, a programmer might write a function with a `void*` formal parameter; inside the function, the formal parameter is cast to one of several different types, depending on certain conditions (e.g., a tag value). Our analysis back-propagates the type obligations of each of these cases to the formal parameter. This may generate spurious error messages, since the type obligation to be satisfied by each actual argument to the function will be the combined set of type obligations generated inside the function. Context-sensitive extensions of our analysis could alleviate this problem. Another problem is to handle polymorphic library functions, such as `fread`. One way to model the effect of such library functions is to conceptually replace each instance of a call with a new copy of the called function. (This is similar to the way in which `malloc` is handled.)

3.3 The Pointer Subtyping Problem

We now turn to the restrictions on pointers that we placed on the program in the algorithm in Section 3.2. Suppose we wish to type check programs in which a member field of a structure itself is a pointer. Consider the following candidate rule for *Pointer Field Dereference*:

$$\frac{l = e \rightarrow x \qquad \mathit{stype}(e) = \mathit{struct}\ t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq \mathit{Concat}(\mathit{PrefixExclusive}(t, x), \langle l \rangle\ \texttt{ptr})}$$

The rationale for this rule is that if the type obligation on $e \rightarrow x$ is $\langle l \rangle$, then there are two requirements on e: there must exist an x field at the appropriate offset, and the type at the pointer-valued field x must respect the type obligation $\langle l \rangle$. A difficulty with this rule is that there is no obvious subtyping rule for comparing `struct` $\{t$ `ptr`$\}$ to `struct` $\{t'$ `ptr`$\}$, where t and t' are related by physical subtyping. This, in turn, is because there is no subtyping rule for comparing t `ptr` to t' `ptr`. Suppose we attempt to remedy this situation by allowing the following pointer subtyping rule:

$$[\text{Pointers}]\frac{t \preccurlyeq t'}{t\ \texttt{ptr} \preccurlyeq t'\ \texttt{ptr}}$$

The following example shows that this rule is not sound.

Example 5. Consider the code in Figure 5. We generate the following constraints:

$$\langle \&cps \rangle \;\sqsubseteq\; \langle psp \rangle \tag{1}$$

```
typedef struct {
  int common;
  Point *p;
} PS;

typedef struct {
  int common;
  ColorPoint *p;
} CPS;

PS *psp;
CPS cps, *cpsp;
Point pt, *q;
ColorPoint *cp;

main() {
  psp = (PS *) &cps;
  cpsp = &cps;
  q = &pt;
  psp->p = q;
  cp = cpsp->p;
  cp->color = RED;
}
```

Figure 5: An unsafe program. The definitions for `Point` and `ColorPoint` are the same as in Figure 1.

$$
\begin{aligned}
\langle \&cps \rangle &\sqsubseteq \langle cpsp \rangle && (2)\\
\langle \&pt \rangle &\sqsubseteq \langle q \rangle && (3)\\
\langle psp \rangle &\sqsubseteq \texttt{struct \{ int common, Point *p \}} && (4)\\
\langle cpsp \rangle &\sqsubseteq \mathit{Concat}(\mathit{PrefixExclusive}(\texttt{CPS}, p), \langle cp \rangle\ \texttt{ptr}) && (5)\\
\langle cp \rangle &\sqsubseteq \texttt{struct \{ int x, y, color \}} && (6)
\end{aligned}
$$

From (6), the value of $\langle cp \rangle$ is `ColorPoint`. From (5), which comes from the *Pointer Field Dereference* rule, $\langle cpsp \rangle$ is `struct { int common, ColorPoint *p }`. From (4), $\langle psp \rangle$ is `struct {int common, Point *p }`. From (1) and (2), and the fact that `struct { int common, ColorPoint *p }` is a physical subtype of `struct {int common, Point *p }` (by the (unsound) pointer subtyping rule), $\langle \&cps \rangle$ is `struct { int common, ColorPoint *p }`. Since `CPS` is a physical subtype of the last constructed type, the program type checks. However, the program is actually unsafe because it accesses the `color` field of `pt`, which is only a `Point`.

Intuitively, the pointer subtyping rule is not sound because it cannot track indirect modifications. In the example just presented, we indirectly modified the value of `cpsp->p` (by assigning to `psp->p`, rather than `cpsp->p`). Had we "visibly" assigned a `Point` pointer value to `cpsp->p`, we would have been able to catch the error. (Incidentally, the void pointers rule of Figure 3 is sound, because void pointers cannot be dereferenced.)

4 An Algorithm for Physical Type Checking

In this section, we present an algorithm for physical type checking that works on all programs, no matter how many levels of pointers are used. The algorithm follows the same pattern as the restrictive algorithm presented in Section 3.2: it generates a set of constraints involving type obligations, and uses the final type obligations to check the pointer sources. We add a new kind of type-obligation term into our constraint system, and we use a different set of rules to generate the constraints.

We first describe these two changes, and then illustrate the algorithm with an example. We also compare this algorithm with points-to analysis.

```
Point p, **pp;
ColorPoint *q;

main() {
  pp = (Point **) &q;
  *pp = &p;
  q->color = RED;
}
```

Figure 6: An example to illustrate *set* type obligations.

4.1 Type Obligations Revisited

The $\langle p \rangle$ type obligations presented so far are, by construction, the smallest (ordered by $\sqsubseteq$) type that can be safely *read* via a dereference of p. This is p's *get* type obligation. We now introduce a complementary *set* type obligation. The *set* type obligation of p, denoted by $[p]$, is the largest type (ordered by $\sqsubseteq$)—or least restrictive type—that must be *written* through a pointer.

Example 6. Consider the code fragment in Figure 6. The *set* type obligation on `pp` is `ColorPoint`, because, when we assign through `pp`, as in `*pp = &p`, we require a particular constraint on the right-hand-side pointer value. This is because the assignment indirectly modifies `q`, and $\langle q \rangle =$ `ColorPoint`. Thus, $[pp] =$ `ColorPoint`.

Intuitively, the *set* type obligation for a pointer variable represents the *get* type obligations of the elements in its points-to set. By tracking the *set* type obligation of a pointer variable in addition to its *get* type obligation, we are able to deal with the problem of pointer subtyping.

4.2 Get-Type and Set-Type Inference

Physical type checking is expressed in terms of the inference rules presented in Figure 7, which now involve constraints over unknowns of the form $\langle p \rangle$ and $[p]$. Note that in the new rules, we not only generate constraints from the statements in the program (as before), but also infer new constraints from previously generated constraints.

We first explain the new items of notation used in the third column of Figure 7. (1) $r.\widehat{a}$ stands for the base-offset representation of the field a in a structure r. Because of type casting, one can view a structure of type S as a structure of type T. When a field named a that belongs to type T is accessed, it does not follow that a corresponding a field exists in the type S. Yong et al. [21] have addressed this issue by using a symbolic representation of fields and their positions. We have adopted a simpler, though non-portable approach of computing the offset of a field from the beginning of the structure. The notation $\widehat{a}$ stands for this offset (in bytes). When r itself has a non-zero offset, i.e., it is a field of some containing structure (e.g., $s.\widehat{b}$), $r.\widehat{a}$ stands for the same base, and a new offset that is the sum of r's offset and $\widehat{a}$ (viz., $s.(\widehat{b}+\widehat{a})$). (2) $r.all$ stands for all offsets within the structure r. $[p] \sqsubseteq [r.all]$ is a shorthand for the set of relations $[p] \sqsubseteq [r.1]$, $[p] \sqsubseteq [r.2]$ etc.

The rationale for the rules given in the third column of Figure 7 is as follows:

- *Address-Of:* This rule considers an assignment of an address-of-variable expression as a special case of assignment. It links the get obligation on the right-hand-side variable x to the set obligation on the left-hand-side variable p via the constraint $[p] \sqsubseteq \langle x \rangle$.
- *Assignment:* The assignment rule now also propagates the set-type obligations in a *forward* direction. Set-type obligations flow forwards, originating from address-of statements (e.g., $p = \&x$) through simple assignments (e.g., $q = p$) to indirect modification statements (e.g., $*q = v$).
- *Pointer Dereference:* Consider the R case (the L case is analogous). This rule can be thought of as an assignment to p of all of the variables that q might point to. If q points to some variable r, we have $[q] \sqsubseteq \langle r \rangle$, either generated directly, or inferred. We infer the constraints that we would generate for the assignment $p = r$.
- *Field Dereference:* Consider the R case (the L case is analogous). For each structure r that q may point to, we infer the constraints that we would generate for the assignment $p = r.\widehat{a}$.
- *Field Address:* This statement essentially translates the pointer q by a particular offset within the structure r that q points to. The field $r.\widehat{a}$ denotes the field whose address is assigned to p. By the reasoning of the *Address-of* case, p's set obligation must satisfy the get obligation of that field.
- *Plus One:* If q points to (any field of) a structure r, in general we do not know which field might $q+1$ point to. To be conservative, we assume that p may point to anywhere in the structure (which is the same behavior that we want on arbitrary arithmetic). This now reduces to the *Field Address* case, but with all possible offsets in r taken into account.

After we perform inference (until no more new judgments are obtained),[4] we use the values of the get-type obligation on pointer sources to perform actual type safety checking. The latter process is the same as in the restrictive algorithm of Section 3. Our inference rules can be encoded in a class of set constraints that is known to be solvable in cubic time. The actual implementation that has been created harnesses the *Bane* system [5] as our constraint-solving engine. The encoding of our constraints in *Bane* is discussed in the technical report [3].

Example 7. Reconsider the code in Figure 5. Using the rules in Figure 7, we generate the following chain of inferences:

$$(1)\ \frac{cp \rightarrow color = \mathtt{RED} \quad stype(cp) = \mathtt{ColorPoint}}{\langle cp \rangle \sqsubseteq \mathtt{ColorPoint}}$$

$$(2)\ \frac{\dfrac{cp = cpsp \rightarrow p \quad \dfrac{cpsp = \&cps}{[cpsp] \sqsubseteq \langle cps \rangle}}{\langle csp.\widehat{p} \rangle \sqsubseteq \langle cp \rangle} \quad \langle cp \rangle \sqsubseteq \mathtt{ColorPoint}\ (1)}{\langle csp.\widehat{p} \rangle \sqsubseteq \mathtt{ColorPoint}}$$

$$(3)\ \frac{\dfrac{psp \rightarrow p = q \quad \dfrac{psp = \&cps}{[psp] \sqsubseteq \langle cps \rangle}}{\langle q \rangle \sqsubseteq \langle cps.\widehat{p} \rangle} \quad \langle csp.\widehat{p} \rangle \sqsubseteq \mathtt{ColorPoint}\ (2)}{\langle q \rangle \sqsubseteq \mathtt{ColorPoint}}$$

$$(4)\ \frac{\dfrac{q = \&pt}{\langle \&pt \rangle \sqsubseteq \langle q \rangle} \quad \langle q \rangle \sqsubseteq \mathtt{ColorPoint}\ (3)}{\langle \&pt \rangle \sqsubseteq \mathtt{ColorPoint}}$$

[4] We have skipped the discussion of how to tackle a special situation in which the inference process, as presented, may not terminate. The technical report shows how to handle this situation [3].

[Address-Of]	$\dfrac{p = \&x}{\langle \&x \rangle \sqsubseteq \langle p \rangle}$	$\dfrac{p = \&x}{[p] \sqsubseteq \langle x \rangle}$
[Assignment]	$\dfrac{p = q}{\langle q \rangle \sqsubseteq \langle p \rangle}$	$\dfrac{p = q}{[p] \sqsubseteq [q]}$
[Pointer Dereference (R)]	$\dfrac{p = *q \quad stype(q) = t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq t}$	$\dfrac{p = *q \quad [q] \sqsubseteq \langle r \rangle}{\langle r \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq [r]}$
[Pointer Dereference (L)]	$\dfrac{*p = q \quad stype(p) = t\ \texttt{ptr}}{\langle p \rangle \sqsubseteq t}$	$\dfrac{*p = q \quad [p] \sqsubseteq \langle r \rangle}{\langle q \rangle \sqsubseteq \langle r \rangle \quad [r] \sqsubseteq [q]}$
[Field Dereference (L)]	$\dfrac{p \to a = q \quad stype(p) = struct\ t\ \texttt{ptr}}{\langle p \rangle \sqsubseteq PrefixInclusive(t, a)}$	$\dfrac{p \to a = q \quad [p] \sqsubseteq \langle r \rangle}{\langle q \rangle \sqsubseteq \langle r.\widehat{a} \rangle \quad [r.\widehat{a}] \sqsubseteq [q]}$
[Field Dereference (R)]	$\dfrac{p = q \to a \quad stype(q) = struct\ t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq PrefixInclusive(t, a)}$	$\dfrac{p = q \to a \quad [q] \sqsubseteq \langle r \rangle}{\langle r.\widehat{a} \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq [r.\widehat{a}]}$
[Field Address]	$\dfrac{p = \&(q \to a) \quad stype(q) = struct\ t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq Concat(PrefixExclusive(t, a), \langle p \rangle)}$	$\dfrac{p = \&(q \to a) \quad [q] \sqsubseteq \langle r \rangle}{[p] \sqsubseteq \langle r.\widehat{a} \rangle}$
[Plus One]	$\dfrac{p = q + 1 \quad stype(q) = t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq Concat(t, \langle p \rangle)}$	$\dfrac{p = q + 1 \quad [q] \sqsubseteq \langle r \rangle}{[p] \sqsubseteq \langle r.all \rangle}$

Figure 7: Constraint generation and inference rules.

Consequently, $\langle \&pt \rangle$ must be a physical subtype of `ColorPoint`. Because the declared type of `pt`, namely `Point`, is not a physical subtype of `ColorPoint`, we flag an error.

4.3 Comparison with Flow-Insensitive Points-To Analysis

In this section, we explore the connection between the algorithm presented above and previous work on flow-insensitive points-to analysis [1, 6, 18, 14, 21]. The goal of points-to analysis is to compute, for each pointer variable p, a set of variables whose address p might contain. The physical-type-checking algorithm has most in common with algorithms for points-to analysis that distinguish between fields of structures [18, 20, 21, 22]. Like much of this work, our analysis tracks fields in terms of a base pointer and a numeric offset. (Consequently, the information obtained from the analysis is specific to a given platform.)

The first point to note is that the nature of the information obtained from physical type checking and points-to analysis is different: In points-to analysis, the information obtained is that a variable p might contain the address of a variable q; in contrast, with physical type checking the information obtained is that the type of a variable q (where the address of q is taken somewhere in the program) needs to have a certain collection of fields for pointer dereferences in the program to be safe.

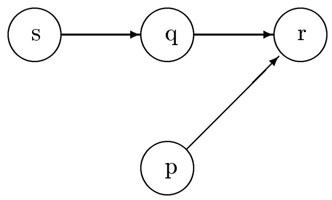
There is also a difference in "philosophy" behind the two kinds of analyses. To obtain points-to information, one makes an *a priori* assumption that the given types of variables are correct, at least insofar as determining the size of a variable is concerned. If the declared type of a variable is inadequate with respect to actual dereferences in the program (e.g., the declared type is `Point` when the dereferences demand a `ColorPoint`), a points-to analysis would either (i) quit, (ii) assume (pessimistically) that an arbitrary piece of memory in the activation record has been clobbered, or (iii) assume (optimistically) that the out-of-bound access does not clobber other variables in the activation record. For example, in such a scenario, Steensgaard's algorithm [18] will fail to "type check" or produce a points-to graph. In contrast, in our approach, all accesses are treated as if they *must* fall within bounds; that is, the essence of our approach is to *infer the types that would make all accesses fall within bounds*. The algorithm discriminates between structure fields to maintain precision, but does not "trust" the declared types.

A key technical difference between physical type checking and points-to analysis is that physical type checking involves a *backwards* propagation of *needs* as opposed to a *forwards* propagation of *points-to information*.

Despite these differences, there are some similarities between physical type checking and points-to analysis. For one thing, the augmented rules of Section 4.2 must account for the effect of indirect modifications via pointers. In addition, a judgement $[p] \sqsubseteq \langle r \rangle$ that arises in our analysis is somewhat similar to a points-to fact $\textit{points-to}(p, r)$, and some of the rules by which our analysis infers such judgements are similar to the rules for inferring points-to facts in Andersen's pointer analysis [1]. However, judgements of the form $[p] \sqsubseteq \langle r \rangle$ do *not* represent *exactly* the same information as points-to facts: If $\textit{points-to}(p, r)$, then our analysis generates the judgement $[p] \sqsubseteq \langle r \rangle$, but the converse does not necessarily hold. For example, in the following set of statements,

```
1. q = &r
2. s = &q
3. p = q
```

although $\textit{points-to}(s, p)$ does not hold, our rules infer the judgment $[s] \sqsubseteq \langle p \rangle$, which is appropriate given the points-to facts that do hold (shown in the following diagram).



Furthermore, consider the following three statements:

```
1. p = &r
2. s = &q
3. q = p
```

This set of statements generates the same points-to relations as the previous set, but the judgment $[s] \sqsubseteq \langle p \rangle$ does not hold. This example shows that the information being computed by our algorithm is incomparable to points-to information.

Points-to analysis does give alternative ways to do physical type checking. (1) One possible physical type-checking algorithm can work in two phases, by performing an *alias analysis* in the first phase, and the type inference of Section 3.2 in the second phase. Recall from Section 3.3, that the "obvious" pointer subtyping rule was not sound, because it could not track indirect modifications. The results of a points-to analysis can be used to compute alias relationships. Given alias information, we can augment the restricted algorithm of Section 3 by making the following change: In each case of an assignment $lhs = rhs$, we assume that all aliases of the left-hand side are also assigned the right-hand-side value. (2) Another possibility is to use the results of points-to analysis to "directly" perform physical type-checking, without invoking the type-inference step of Section 3.2. Given the points-to relation, we can verify the validity of each field dereference $p \to a$ by looking for the field a in each points-to target of p. This approach, however, does not actually construct an expected type for the target. Starting from the results of points-to analysis, one will need to perform a computation similar to the one given in Section 3.2 to construct expected types.

Although the results from a points-to analysis can be used to achieve the goals of physical type checking, this involves working "outside the type system". In particular, it addresses the issue of physical subtyping in the presence of pointers indirectly, at best. One of our contributions is the formulation of a rule for physical subtyping in the presence of pointers. The key idea in our approach involves introducing two distinct variables in the constraint system per program variable. This approach may have applications beyond physical type checking.

5 Related Work

In addition to the work on pointer analysis, which we discussed in Section 4.3, related work falls into four categories:

Static semantic checking tools Physical type-checking is related to, but complementary to, such tools as *lint* [9] and *LCLint* [4]. Our algorithm, as well as *lint* and *LCLint*, can be used in static detection of type errors that escape the notice of many C compilers. *LCLint* can identify problems and constructs that our system cannot, for example problems with dereferencing null pointers, but only by adding explicit annotations to the source code. On the other hand, neither *lint* nor *LCLint* has a notion of subtyping.

Alternative type systems. The idea of applying alternative type systems to C appears in several places, among them [7, 17, 11, 12, 16, 18]. Most of these references discuss the application of *parametric polymorphism* to C. In particular, [17] concerns a new dialect of C that is polymorphic and type safe. [11] uses polymorphic type inference on existing C programs, but for determining information about the transfer of values. [12] and [18] present algorithms to infer a certain kind of typing on structures. Both infer (though for different purposes) a maximal allowable "coarsening" of a structure: at one end, a structure may be treated atomically, and at the other end, each field may be referred to separately, depending on the accesses in a program.

Comparison to Record Subtyping Our work has some similarity to the work on record subtyping by Cardelli [2]. In both cases, a structure (or record) that contains a superset of the fields of another structure is considered a subtype of the second structure. The primary difference is that we take into account the physical layout of data types when determining subtype relationships, while in Cardelli's work the notion of a physical layout does not apply. The problem of subtyping in the presence of pointer fields inside structures appears to be related to the problem of record subtyping in the presence of mutable fields.

Constraint-based analyses The constraint-based analysis for inferring most general physical types bears some relationship to certain kinds of (backwards, flow-sensitive) need-based analyses developed in the functional-programming community, including algorithms for neededness [8], strictness analysis [19], program slicing [13], and dependence analysis [10]. These all use the idea of treating the accesses on a variable as a "contract" to limit attention to certain portions of the variable in question; the minimal obligations on a variable are determined by accounting for all of the accesses on it. Our work has applied this idea in the context of a flow-insensitive analysis for C (an imperative language that supports destructive updating of heap-allocated storage). Our algorithm incorporates the notion of a "set-type obligation" in order to handle destructive updating.

Acknowledgements

Michael Siff and Thomas Ball made many contributions to this work. The second author was supported, in part, by the National Science Foundation under grants CCR-9625667 and CCR-9619219, by the United States-Israel Binational Science Foundation under grant 96-00337, by a grant from IBM, and by a Vilas Associate Award from the University of Wisconsin.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] Luca Cardelli. A semantics of multiple inheritance. In G.Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, number 173 in Lecture Notes in Computer Science, pages 51–68. Springer-Verlag, 1984.

[3] Satish Chandra and Thomas Reps. Physical type checking for C. Technical Report BL0113590-990302-04, Lucent Technologies, Bell Laboratories, March 1999. Available at `http://www.bell-labs.com/~schandra/pubs/checking-tr.ps`.

[4] David Evans. Static detection of dynamic memory errors. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 44–53, May 1996.

[5] Manuel Fähndrich and Alex Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, 1997.

[6] J. Foster, M. Fähndrich, and A. Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB//CSD-97-964, University of California, Berkeley, July 1997.

[7] F.-J. Grosch and G. Snelting. Polymorphic components for monomorphic languages. In R. Prieto-Diaz and W.B. Frakes, editors, *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*, pages 47–55, Lucca, Italy, March 1993. IEEE Computer Society Press.

[8] J. Hughes. Backwards analysis of functional programs. *Partial Evaluation and Mixed Computation: Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation, (Gammel Avernaes, Denmark, Oct. 18-24, 1987)*, pages 187–208, 1988.

[9] S. C. Johnson. Lint, a C program checker, July 1978.

[10] Y.A. Liu. Dependence analysis for recursive data. *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 206–215, May 1998.

[11] Robert O'Callahan and Daniel Jackson. Detecting shared representations using type inference. Technical Report CMU-CS-95-202, Carnegie Mellon University, September 1995.

[12] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[13] T. Reps and T. Turnidge. Program specialization via program slicing. *Proc. of the Dagstuhl Seminar on Partial Evaluation, (Schloss Dagstuhl, Wadern, Ger., Feb. 12-16, 1996), Lec. Notes in Comp. Sci., Vol. 1110*, pages 409–429, 1996.

[14] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 1997.

[15] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. Technical Report BL0113590-990202-03, Lucent Technologies, Bell Laboratories, February 1999. Available at `http://www.bell-labs.com/~schandra/pubs/coping-tr.ps`.

[16] Michael Siff and Thomas Reps. Program generalization for software reuse: From C to C++. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 135–146, San Francisco, October 1996.

[17] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *1996 European Symposium on Programming*, April 1996.

[18] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 1996 International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150. Springer-Verlag, April 1996.

[19] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. *Third Conf. on Func. Prog. and Comp. Arch. (Portland, OR, Sept. 14-16, 1987), Lec. Notes in Comp. Sci., Vol. 274*, pages 385–407, 1987.

[20] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 1995.

[21] Suan Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[22] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, San Francisco, October 1996.