Appears as Bell Laboratories Technical Report BL0113590-990302-04.

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

April 6, 1999

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 `structs`), 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 and its implementation. 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[SCKR99], and a type checker that disallows all casts would, in practice, outlaw too many programs.

Example 2. Consider the code fragment in Figure 2. In this code, the cast `(Radio *)` converts a type `Clock *` (the type of `c + 1` as 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

```
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. The dereference `pcp->color` can cause unexpected behavior in this program.

```
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. By C's type rules, the expression `(c + 1)` has the same type as `c`. Therefore, the cast at the second statement is from `Clock*` to `Radio*`. Because `Radio` follows `Clock` in `struct ClockRadio`, the cast is actually safe.

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 also on the fact that the size of `Clock` is such that no padding is required between the fields `clock` and `radio`). Although this usage appears contrived, we have found it used frequently in production code (see [SCKR99]).

These examples show that C programmers implicitly rely on the physical layout of `structs` 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.

Moreover, casts are used heavily in C programs, particularly in systems software. Table 1 presents empirical data from a suite of C programs.

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 [SCKR99]—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.

1.1 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 the first example (Figure 1), our type-checking algorithm would declare the program to be unsafe, because the actual type of `p`, `Point`, does not have a field `color`, which is required because of the dereference `pcp->color`. If the program did not contain the statement `pcp->color = RED`, our algorithm would declare the program to be type safe, because the requirements on the structure `pcp` would be satisfied by its actual type. In the second example (Figure 2), our algorithm would declare the program to be type safe, because the requirements on the structure `cr` are satisfied by its actual type.

Physical type checking is carried out by a flow-insensitive, context-insensitive, interprocedural analysis algorithm. In terms of how this analysis 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 6).
- There are some similarities between the physical-type-checking algorithm and previous work on flow-insensitive points-to analysis [And94, Ste96, SH97, YHR99]. The relationship between physical type checking and points-to analysis is addressed in Section 4.3.

Physical type checking is 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* [Eva96] 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.

t ::

ground	
$\mid$ $t[n]$	// array of type t of size n
$\mid$ t `ptr`	// pointer to t
$\mid$ $s\{m_1, \ldots, m_k\}$	// `struct`
$\mid$ $u\{\|m_1, \ldots, m_k\|\}$	// `union`

m ::

(t, l, i)	// member labeled l of type t at offset i
$\mid$ $(l : n, i)$	// bit field labeled l of size n at offset i

ground ::

$e\{id_1, \ldots, id_k\}$	// `enum`

`void*` | `char` | `unsigned char` | `short` | `int` | `long` | `double` | ...

Figure 3: A slightly simplified type system for C. Type qualifiers are ignored (e.g., `const int` and `volatile int` are treated as `int`). Furthermore, typedefs are considered to be synonyms for the types they redefine.

1.2 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). The paper's contributions can be categorized as follows:

- We formulate an alternative type system for C based on the physical layout of `structs` in memory.
- We give an inference-based algorithm to perform such type checking for C programs. We also describe an implementation of the algorithm that uses an off-the-shelf constraint solver.
- 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.

1.3 Outline

The remainder of the paper is organized as follows: Section 2 reviews some ideas from our previous work [SCKR99] 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. We describe our implementation in Section 5. Section 6 discusses related work.

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 [SCKR99].

Our work addresses a slightly simplified version of the ANSI C type system, as shown in Figure 3. Members and bit-fields of `struct` and `union` types are annotated with an *offset*. In a `struct`, the offset of a member indicates the difference in bytes between the storage location of this member and the first member of the `struct`. The first member is, by definition, at offset 0. All union members have offset 0.

The computation of other offsets is compiler dependent, but must follow a number of requirements, as set down in the ANSI C standard. In particular, C compilers lay out *compatible* prefixes of two structures identically. We discuss the data-layout issue further in Appendix A.

It is useful to define a number of auxiliary functions for a type t:

- $stype(e)$ is the compiler-assigned type of a C expression e;
- $sizeof(t)$ is the standard C `sizeof` function;
- $offset(t, x)$ 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 to convert a program into this normal form may need to introduce temporary

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	$p \rightarrow a = q$
Field Dereference on rhs	$p = q \rightarrow a$
Field Address	$p = \&(q \rightarrow a)$
Plus One	$p = q + 1$
Other Arithmetic	$p = q$ op k

Table 2: Statements in the normal form.

variables. We also assume that all assignment statements copy values only of ground types, i.e., `struct` copies are transformed to element-wise copies.[1] 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.

Table 3 shows the normalization of a few sample statements.

$p = s.a$	$tmp = \&s$ $p = tmp \rightarrow a$
$p = \&(s.a)$	$tmp = \&s$ $p = \&(tmp \rightarrow a)$
$p \rightarrow a \rightarrow b = q$	$tmp = p \rightarrow a$ $tmp \rightarrow b = q$

Table 3: Examples of normalization.

2.1 Physical Type Safety

Since the semantics of C is not formally specified, we must state precisely what we mean by physical type safety in the context of C programs. Here, we provide only an intuitive notion of safety requirements on pointer dereferences, leaving a more formal characterization to an appendix. Appendix B describes a runtime notion of physical type safety that corresponds to these intuitive safety requirements.

Intuitively, 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

[1] For this transformation to fully simulate a call to *memcpy*, it may be necessary to introduce manufactured field names to stand for holes introduced by padding.

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.

2.2 Physical Subtyping

A cast-free C program that type checks (and that does not use `unions` 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.c = 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.*

We write $t \preccurlyeq t'$ to denote that t is a *physical subtype* of type t'. The intuition behind physical subtypes can be summarized as follows:

- The size of a type is no larger than the size of any of its subtypes.
- Ground types are physical subtypes of themselves and not of other ground types. For example:
  - `int`$\preccurlyeq$`int`
  - `int`$\not\preccurlyeq$`double`
  - `double`$\not\preccurlyeq$`char`
  - an enumerated type is not a physical subtype of a different enumerated type (or any other ground type)
- If a `struct` type is a physical subtype of another `struct` type then the members of two types line up in some sensible fashion. In matching two `struct` types, we allow a number of "relaxations", for example, flattening out the `struct` types, or renaming their member labels.

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

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

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

[Structures] $$\frac{k' \leq k \quad m_1 \preccurlyeq m'_1 \ldots m_{k'} \preccurlyeq m'_{k'}}{\{m_1, \ldots, m_k\} \preccurlyeq \{m'_1, \ldots, m'_{k'}\}}$$

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

Figure 4: Inference rules for physical subtypes.

Figure 4 presents rules for inferring that a C type t is a physical subtype of C type t' (denoted by $t \preccurlyeq t'$), in the style of [Gun92].

The rule for structures attempts to match up one structure as a prefix of another structure. (The rule does not show the relaxations that we allow in matching `struct` types.) Note that there is no rule for physical subtyping that involves comparing two `union` types. This is because with `unions`, 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 [SCKR99], we report on how these subtyping rules "explain" several patterns of cast usage in C programs.

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. Our motivation for splitting the presentation into two sections is that 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:

$$\begin{aligned} T &= \{t_1, \ldots, t_n\} \\ &\mid \Lambda \end{aligned}$$

where t_i is a C type as defined in Figure 3. 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,

```
union { int i; double d; } *u;
...
u->i = 3;
...
u->d = 3.14;
```

```
struct t { int i; } a;
struct s { double d;} b;
void *p;
...
p = &a;
((struct t *)p)->i = 3;
p = &b;
((struct s *)p)->d = 3.14
```

Figure 5: This figure shows two examples where no single C type would guarantee physical type safety.

- $T = \{t\}$, 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 (see Figure 5); however, each of $\{t_1\}, \ldots, \{t_n\}$ are type obligations on p.

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 constraint.) We also introduce type obligations for *pointer sources.* The following syntactic occurrences are pointer sources:

- An address-of `&var`
- A `malloc` call site
- An array arithmetic operation `a+i`, where a is an array. (Multidimensional arrays also yield a pointer if the number of subscripts is less than the declared arity.)

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:

- Use of C `unions` can introduce incompatible type requirements if the dereferences refer to multiple interpretations of the `union`. In the first code fragment in Figure 5, there is no single C type that is a physical subtype of both an `int` and a `double`, and could guarantee type safety, therefore $\langle u \rangle$ cannot be assigned any C type (see the discussion below). This could happen even if our algorithm were flow-sensitive.[2]
- The flow-insensitive nature of our inference algorithm can give rise to spurious type requirements. In the second code fragment in Figure 5, $\langle p \rangle$ must be a physical subtype of both `struct` s and `struct` t, even though p is used in only one way each time it is assigned. There is no C type that is a physical subtype of both `struct` s and `struct` t.

It may be tempting to infer a C `union` type as the least restrictive type in such 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.[3] For example, `union {int i; double j;}` is not a physical subtype of either `int` or `double`, because we do not know which of the two

[2]We note again that analyzing whether or not `unions` are treated in a "consistent" manner (e.g., as variant records) is an orthogonal issue.

[3]There exist programs that abuse `unions` to do type casting.

fields was last stored in the `union`. By contrast, $\{t_1, \ldots, t_n\}$ 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. Thus, $\{\texttt{int}, \texttt{double}\}$ is a physical subtype of both `int` and `double`. The set form of type obligations 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 $\sqsubseteq$ has the following properties:

$$\begin{aligned} T &\sqsubseteq T \\ T &\sqsubseteq \Lambda \\ T_1 \sqsubseteq T_2 \wedge T_2 \sqsubseteq T_3 &\Rightarrow T_1 \sqsubseteq T_3 \\ \{\ldots, t_i, \ldots\} &\sqsubseteq \{t_i\} \\ T \cup \{t\} &\sqsubseteq T \\ t_1 \preccurlyeq t_2 &\Leftrightarrow \{t_1\} \sqsubseteq \{t_2\} \end{aligned}$$

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

In this subsection, we describe the intuition behind our inference-based approach to physical type checking. Our description is expressed as a simpler algorithm for physical type checking, and works only for a *restricted subset* of programs; the full algorithm is described in Section 4. 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.[4]

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 process of inferring least-restrictive type obligations is described below in Section 3.2.1 and Section 3.2.2. 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 static 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 static 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.

We provide some examples of this process in Section 3.2.3. 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.

[4]Since this algorithm is presented only for expository purposes, we are not concerned with testing whether a program qualifies for use with this restrictive algorithm. One possibility, however, is to perform a conservative points-to analysis on the program.

[Field Dereference] $$\frac{l = e \to x \quad or \quad e \to x = r \qquad stype(e) = struct\ t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq PrefixInclusive(t, x)}$$

[Pointer Dereference] $$\frac{l = *e \quad or \quad *e = r \qquad stype(e) = t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq t}$$

[Field Address] $$\frac{l = \&(e \to x) \qquad stype(e) = struct\ t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq Concat(PrefixExclusive(t, x), \langle l \rangle)}$$

[Plus One] $$\frac{l = e + 1 \qquad stype(e) = t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq Concat(t, \langle l \rangle)}$$

[Assignment] $$\frac{l = e}{\langle e \rangle \sqsubseteq \langle l \rangle}$$

[Address-Of] $$\frac{l = \&e}{\langle \&e \rangle \sqsubseteq \langle l \rangle}$$

Figure 6: The relation generation rules.

3.2.1 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 Figure 6. The $\langle e \rangle$ terms appear as "unknowns" in this set of relations—the procedure to "solve" for these terms is described in the Section 3.2.2.

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:

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

Define

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

and

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

and

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

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,[5] but not including field x; the function *PrefixInclusive* is similar except it includes the field x. 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 rationale for the rules in Figure 6 as follows:

- *Field Dereference*: The type obligation on e should be a `struct` that has space for an x field at the right offset. The constructed type $PrefixInclusive(t, x)$ expresses exactly this criterion. Note that the field $e \to x$ itself cannot contain a pointer value.
- *Pointer Dereference*: The type obligation on e is *typeof* $(*e)$, which constraints e to point to the (only) valid ground type. Note that $*e$ itself cannot contain a pointer value.

[5] This may include unnamed "holes" introduced by padding that alignment restrictions may require. We do not explicitly mention these holes in our examples.

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

```
struct {
  int a;
  int b;
  struct {
    int x;
  } u;
}
```

Figure 7: Example to explain the *Field Address* constraint-generation rule. The `struct` on the right hand side is the constructed type that constrains $\langle p \rangle$. See the text for details.

- *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 7, `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*(`struct S, u`) must satisfy the type obligation of `q`. The constructed type shown on the right in Figure 7 constrains $\langle p \rangle$ accordingly.

- *Plus One*: For arbitrary arithmetic, the algorithm would declare the type obligation on e to be a divergent type ($\top$). However, our algorithm takes a special interest in the case of $e+1$, 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 e need not be a pointer to a `struct`.

- *Assignment*: The type obligation for the left-hand side of an assignment is propagated to the right-hand side. That is, the assignment rule propagates type obligations counter to the direction in which values flow during execution. 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 case is similar to *Assignment*.

3.2.2 Solving for $\langle e \rangle$

The result of the constraint-generation procedure 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 following resolution rules, until we have only a single relation involving any given $\langle e \rangle$ term on the left-hand side.

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

These rules, which follow from the properties of $\sqsubseteq$ mentioned in Section 3.1, either eliminate types that are already covered by other types because of physical subtyping, or if they are not related by subtyping, combine them into a multi-element set. The final right-hand-side value in $\langle e \rangle \sqsubseteq T$ is the answer we get for $\langle e \rangle$.[6]

[6]We have side-stepped certain details such as an occurs-check test in constructed type terms. These implementation issues will be covered in Section 5.

3.2.3 Examples of Inference-Based Type Checking

Example 3. 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

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

Since `struct {int x, y, color}` $\preccurlyeq$ `struct {int x, y}`, the simplified constraint for $\langle \&p \rangle$ is $\langle \&p \rangle \sqsubseteq$ `struct {int x, y, color}`. 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 (cf. (1) above).

Example 4. Consider the code in Figure 2. We normalize the first statement `c = &(cr.clock)` to `pcr = &cr` and `c = &(pcr->clock)`. The constraints generated for the four statements are:

$$\langle \&cr \rangle \sqsubseteq \langle pcr \rangle \quad (1)$$
$$\langle pcr \rangle \sqsubseteq \mathit{Concat}(\mathit{PrefixExclusive}(\texttt{Clock}, \texttt{clock}), \langle c \rangle) \quad (2)$$
$$\langle c \rangle \sqsubseteq \mathit{Concat}(\texttt{Clock}, \langle r \rangle) \quad (3)$$
$$\langle r \rangle \sqsubseteq \texttt{struct \{double frequency\}} \quad (4)$$

From constraint (4), the solution for $\langle r \rangle$ is `struct {double frequency}`. From (3), the solution for $\langle c \rangle$ is *Concat*(`Clock`, `struct {double frequency}`), or `struct {Clock c, struct {double frequency} r}`.[7] From (2), the solution for $\langle pcr \rangle$ is `struct { struct {Clock c, struct {double frequency} r} s}`,[8] which, by (1), becomes the solution to $\langle \&cr \rangle$. Since `ClockRadio` is a physical subtype of the last constructed type, the program type checks.

Example 5. Figure 8 presents an example of a code where the flow-insensitive nature of our algorithm forces us to infer multi-set types as least restrictive type obligations. In this code fragment, a `void*` variable `p` is used to hold, at one time, address of a `Point` variable, and at another time, address of a `ColorPoint` variable. $\langle p \rangle$ evaluates to { `Point, RealPoint` }. However, when { `Point, RealPoint` } is compared against `&pt` and `&rpt`, we will be forced to announce a possible unsafety. As an aside, if prior to running our algorithm, a conversion to single-static assignment form performed on the program (so independent occurrences of `p` are named differently), the algorithm would declare the program to be safe.

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.

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 the case in which a member field of a structure itself is a pointer. Consider the following candidate rule for pointer fields:

[7] The field names such as `c` and `r` are actually offsets in the implementation and thus specific names are unimportant. (We use symbolic names to keep the examples comprehensible.)

[8] *PrefixExclusive*(`Clock`,`clock`) is null.

```
typedef struct {
  float x;
  float y;
} RealPoint;

Point pt;
RealPoint rpt;
void *p;

p = (Point *) &pt;
p->y = 3;
...
p = (RealPoint *) &rpt;
p->y = 3.5;
...
```

Figure 8: Inferring a multi-element set type. The definition for `Point` is the same as in Figure 1.

$$\text{[Pointer Field Dereference]} \quad \frac{l = e \to x \qquad stype(e) = struct\ t\ \texttt{ptr}}{\langle e \rangle \sqsubseteq Concat(PrefixExclusive(t, x), \langle l \rangle\ \texttt{ptr})}$$

The rationale for this rule is that if the type obligation on $e \to x$ 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 $\langle l \rangle$. In other words, e should be pointing to a physical subtype of some structure that has an x field at the appropriate offset, and that x field must contain a pointer to a structure that is a physical subtype of $\langle l \rangle$.

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:

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

The following example shows that this rule is not sound.

Example 6. Consider the code in Figure 9. We generate the following constraints:

$$\begin{aligned}
\langle \&cps \rangle &\sqsubseteq \langle psp \rangle && (1)\\
\langle \&cps \rangle &\sqsubseteq \langle cpsp \rangle && (2)\\
\langle \&pt \rangle &\sqsubseteq \langle q \rangle && (3)\\
\langle psp \rangle &\sqsubseteq \texttt{struct \{ int common, Point *p \}} && (4)\\
\langle cpsp \rangle &\sqsubseteq Concat(PrefixExclusive(\texttt{CPS}, p), \langle cp \rangle\ \texttt{ptr}) && (5)\\
\langle cp \rangle &\sqsubseteq \texttt{struct \{ int x, y, color \}} && (6)
\end{aligned}$$

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

```
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;
}
```

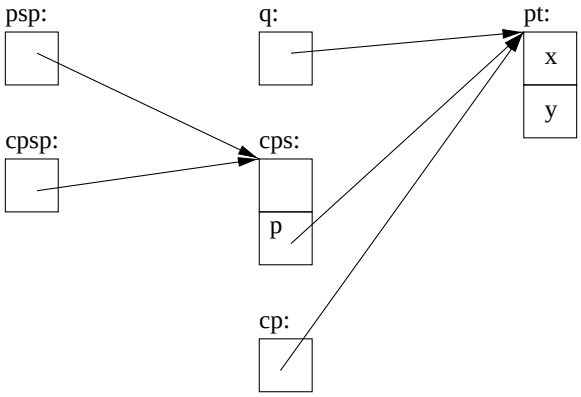

Figure 9: An unsafe program. The definitions for `Point` and `ColorPoint` are the same as in Figure 1. The picture shows the state just before the execution of the last statement.

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 4 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.

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. The following example illustrates this idea.

Example 7. Consider the code fragment in Figure 10. 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

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

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

Figure 10: An example to illustrate *set* type obligations.

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 11, 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 Figure 11.

- $r.\widehat{a}$ 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. [YHR99] 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., $s.(\widehat{b} + \widehat{a})$).
- $r.all$ 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 Figure 11 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:* This rule is similar to the assignment rule in Figure 6, except that now it 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., $*q = v$).

In the remaining rules, the first parts are the same as in Figure 6 in each case. We describe the rationale only for the second parts.

- *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 $p = r.\widehat{a}$.

[Address-Of] $$\frac{p = \&x}{\langle \&x \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq \langle x \rangle}$$

[Assignment] $$\frac{p = q}{\langle q \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq [q]}$$

[Pointer Dereference (R)] $$\frac{p = *q \quad stype(q) = t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq t} \quad \frac{p = *q \quad [q] \sqsubseteq \langle r \rangle}{\langle r \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq [r]}$$

[Pointer Dereference (L)] $$\frac{*p = q \quad stype(p) = t\ \texttt{ptr}}{\langle p \rangle \sqsubseteq t} \quad \frac{*p = q \quad [p] \sqsubseteq \langle r \rangle}{\langle q \rangle \sqsubseteq \langle r \rangle \quad [r] \sqsubseteq [q]}$$

[Field Dereference (L)] $$\frac{p \to a = q \quad stype(p) = struct\ t\ \texttt{ptr}}{\langle p \rangle \sqsubseteq PrefixInclusive(t, a)} \quad \frac{p \to a = q \quad [p] \sqsubseteq \langle r \rangle}{\langle q \rangle \sqsubseteq \langle r.\widehat{a} \rangle \quad [r.\widehat{a}] \sqsubseteq [q]}$$

[Field Dereference (R)] $$\frac{p = q \to a \quad stype(q) = struct\ t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq PrefixInclusive(t, a)} \quad \frac{p = q \to a \quad [q] \sqsubseteq \langle r \rangle}{\langle r.\widehat{a} \rangle \sqsubseteq \langle p \rangle \quad [p] \sqsubseteq [r.\widehat{a}]}$$

[Field Address] $$\frac{p = \&(q \to a) \quad stype(q) = struct\ t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq Concat(PrefixExclusive(t, a), \langle p \rangle)} \quad \frac{p = \&(q \to a) \quad [q] \sqsubseteq \langle r \rangle}{[p] \sqsubseteq \langle r.\widehat{a} \rangle}$$

[Plus One] $$\frac{p = q + 1 \quad stype(q) = t\ \texttt{ptr}}{\langle q \rangle \sqsubseteq Concat(t, \langle p \rangle)} \quad \frac{p = q + 1 \quad [q] \sqsubseteq \langle r \rangle}{[p] \sqsubseteq \langle r.all \rangle}$$

Figure 11: Constraint generation and inference rules.

- *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 *Field Offset* case, but with all possible offsets in r taken into account.

After we perform inference (until no more new judgments are obtained), 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 (we sketch this encoding in Section 5).

Example 8. Reconsider the code in Figure 9. Using the rules in Figure 11, we generate the following chain of inferences:

$$(1)\ \frac{\dfrac{cp = cpsp \to p \quad \dfrac{cpsp = \&cps}{[cpsp] \sqsubseteq \langle cps \rangle}}{\langle csp.\widehat{p} \rangle \sqsubseteq \langle cp \rangle} \quad \dfrac{cp \to color = \texttt{RED} \quad stype(cp) = \texttt{ColorPoint}}{\langle cp \rangle \sqsubseteq \texttt{ColorPoint}}}{\langle csp.\widehat{p} \rangle \sqsubseteq \texttt{ColorPoint}}$$

$$(2)\ \frac{\dfrac{psp \to p = q \quad \dfrac{psp = \&cps}{[psp] \sqsubseteq \langle cps \rangle}}{\langle q \rangle \sqsubseteq \langle cps.\widehat{p} \rangle} \quad \langle csp.\widehat{p} \rangle \sqsubseteq \texttt{ColorPoint}\ \text{(from (1))}}{\langle q \rangle \sqsubseteq \texttt{ColorPoint}}$$

$$(3)\ \frac{\dfrac{q = \&pt}{\langle \&pt \rangle \sqsubseteq \langle q \rangle} \quad \langle q \rangle \sqsubseteq \texttt{ColorPoint}\ \text{(from (2))}}{\langle \&pt \rangle \sqsubseteq \texttt{ColorPoint}}$$

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.

Example 9. The purpose of this example is to point out a possibility of divergence during constraint generation and inference, and one way to avoid it. Consider the following three statements:

```
1. q = &M;
2. p = &(q->a);
3. q = p;
```

Statement 1 will generate the constraint $[q] \sqsubseteq \langle M \rangle$. Statement 2, using the *Field Offset* rule, will generate the constraint $[p] \sqsubseteq \langle M.\widehat{a} \rangle$. From statement 3, we have $[q] \sqsubseteq [p]$. By transitivity, $[q] \sqsubseteq \langle M.\widehat{a} \rangle$. The steps just mentioned can now be repeated, leading to increasingly deeper cumulative offsets into M ($M.\widehat{a}$, $M.(2 * \widehat{a})$, etc.), and non-termination of constraint generation. Since we do not rely on declared types in the inference phase (except for evaluating $\widehat{a}$), the algorithm must assume that all offsets—no matter how deep—are possible. To circumvent this problem, an implementation could check for the offset of r in the *Field Offset* case. If it is above a prespecified threshold, then, instead of inferring the usual constraint $[p] \sqsubseteq \langle r.\widehat{a} \rangle$, it should infer $[p] \sqsubseteq \langle r.all \rangle$. An appropriate threshold would be the size of the largest `struct` type in the program.

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 [And94, Ste96, SH97, YHR99].

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 [Ste96, WL95, YHR99, ZRL96]. 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 (although we are confident that a portable, platform-independent version could be developed).

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 are also differences 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 [Ste96] 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.

- Most points-to analyses attempt to track the consequences of whatever casts a programmer has made use of in his or her program. The motivation behind physical type checking is different. We identify a class of "sensible" casts that we are prepared to handle—e.g., upcasts from a subtype to a supertype—and deem all others unacceptable. The notion of "sensible" is motivated by a number of idioms that C programmers use to simulate object-oriented language features [SCKR99].

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 judgment $[p] \sqsubseteq \langle r \rangle$ that arises in our analysis is somewhat similar to a points-to fact *points-to*(p, r), and some of the rules by which our analysis infers such judgments are similar to the rules for inferring points-to facts in Andersen's pointer analysis [And94]. However, judgments of the form $[p] \sqsubseteq \langle r \rangle$ do *not* represent *exactly* the same information as points-to facts: If *points-to*(p, r), then our analysis generates the judgment $[p] \sqsubseteq \langle r \rangle$, but the converse does not necessarily hold, as shown by the following example.

Example 10. Consider the following three statements:

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

A pointer-analysis would infer the following points-to facts: *points-to*(q, r), *points-to*(s, q), and *points-to*(p, r). In particular, Andersen's analysis would generate the following proof tree:

$$\frac{p = q \quad \dfrac{q = \&r}{\textit{points-to}(q, r)}}{\textit{points-to}(p, r)}$$

Correspondingly, our analysis would generate the following proof tree:

$$\frac{\dfrac{p = q}{[p] \sqsubseteq [q]} \quad \dfrac{q = \&r}{[q] \sqsubseteq \langle r \rangle}}{[p] \sqsubseteq \langle r \rangle}$$

Our proof tree has a slightly different shape, because in our system, $p = q$ does not insist on an immediate judgment of the form $[q] \sqsubseteq \langle r \rangle$. However, our analysis also admits another proof tree:

$$\frac{\dfrac{s = \&q}{[s] \sqsubseteq \langle q \rangle} \quad \dfrac{p = q}{\langle q \rangle \sqsubseteq \langle p \rangle}}{[s] \sqsubseteq \langle p \rangle}$$

Although *points-to*(s, p) does not hold, the inferred judgment $[s] \sqsubseteq \langle p \rangle$ is appropriate. It says that p's get-type obligation must be larger that s's set-type obligation, which makes sense given the points-to relations that do hold. This example shows that the judgment $[p] \sqsubseteq \langle r \rangle$ is not the same as *points-to*(p, r).

It should be noted that pointer analysis does give alternative ways to do physical type checking.

- 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 the problem we faced in formulating a subtyping rule for pointers (Section 3.3). We found that the "obvious" pointer subtyping rule was not sound, because it could not track indirect modifications. Two l-values in a program are *aliases* if they may denote the same memory location. The results of a points-to analysis can be used to compute alias relationships:
  - If p points to x, then $*p$ and x are aliases.
  - If p and q are variables that have a common member in their points-to sets, $*p$ and $*q$ are aliases.
  - If e_1 and e_2 are aliases, so are $e_1.a$ and $e_2.a$, where a is a field in a structure that both e_1 and e_2 may denote (as l-values).

  Given alias information, we can augment the restricted algorithm of Section 3 in the following manner: 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.

In the example of Figure 9, once aliasing information is taken into account, the algorithm would expose the type-safety violation as follows: Because `cpsp->p` is an alias of `psp->p`, the assignment `psp->p = q` also requires us to consider the effect of `cpsp->p = q`. We now have

$$\langle \&pt \rangle \sqsubseteq \langle q \rangle \sqsubseteq \langle cpsp \rightarrow p \rangle \sqsubseteq \langle cp \rangle \sqsubseteq \texttt{struct \{int x, y, color\}},$$

which propagates the type obligation on `cp` to `&pt`.

- 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 $p \rightarrow a$ by looking for the field a in each points-to target of p. This approach, however, is not as well-suited to reverse engineering, because it 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 Implementation

Our implementation first uses a C front end to build an abstract syntax tree (AST) of a program. It then traverses the AST and generates constraints in a form suitable for being solved by *Bane* [FA97], a publicly available constraint solver from University of California, Berkeley. In this section, we describe several implementation details that arise in this process.

5.1 Dealing with structure fields

Our algorithm relies on a base-offset scheme to refer to the fields of a `struct` and therefore discriminates between fields precisely (though not portably). In order to follow the algorithm faithfully, an implementation must store separately the type-obligation information for every offset that can be referred to in each structure.

In practice, a given implementation may trade precision for lower memory requirements and higher speed. An implementation may choose not to discriminate between the fields of a `struct` by deliberately taking each offset (an $\widehat{a}$ term) to be zero. An implementation might also choose to maintain precise information only as long as the maximum cumulative offset remains below a prespecified threshold.

5.2 Handling of Arrays and Function Pointers

Variables of an array type are treated as pointers to the memory the array occupies. Thus, an array `a` is treated as $\&a$, where a is the name of the block of memory corresponding to the array. (A syntactic address-of operator applied to an array is treated as a no-op.) A read or write to any individual array element is treated as a read or write to a, the entire array.

Function names are lifted to be pointers to function definitions. (A syntactic address-of operator applied to a function is treated as a no-op.) Each function definition is also associated with a special return variable. The algorithm tracks function pointers in 0-CFA style. At call sites, formal parameters are

considered to be assigned the values of actual parameters,[9] and a (possibly dummy) variable gets assigned the value of the function's return variable.

Our treatment of arrays and function pointers is similar to that of Foster et al. [FFA97].

5.3 Constraint Solving

We now sketch how to express our algorithm in *Bane*. Our intention here is only to give a flavor of how our algorithm can be implemented with *Bane* to the reader. An actual implementation of our algorithm includes more details, e.g. handling of `struct` fields, which we do not describe here. This is also not a complete description of *Bane*: for further details on *Bane* and on how to use it to perform program analyses (including points-to analysis), we refer the reader to [FA97, FFA97].

We use the part of *Bane* that deals with *sets*. To begin with, *Bane* has variables and constructors, both of which are expressions of type set. For example, one might declare a 0-ary constructor $R : \text{unit} \to s$, and a binary constructor $C : (s, -s) \to s$. The binary constructor declaration specifies that its first argument is covariant and second argument contravariant. An example of an expression using this constructor is $C(R, 0)$, where the constant 0 denotes the empty set (1 denotes the universal set). Finally, *Bane* defines an inclusion relation $\leq$ on expressions. Thus, $C(R, 0) \leq p$ says that the set denoted by variable p includes the set denoted by the constructed term $C(R, 0)$. The *Bane* interpretation of inclusion is the same as the usual subset relation on sets. For example, $\leq$ is transitive, and $0 \leq p$ for all p.

In encoding our constraints in *Bane*, we exploit constructors that have both covariant and contravariant positions. *Bane* resolves constructed terms as follows. Two constructed terms cannot be compared with $\leq$ unless their constructor is the same (a violation of this leads to an error). With the same constructor, resolution introduces element-wise inclusion constraints, with the sense of inclusion reversed for contravariant arguments. Thus, $C(R, q) \leq C(p, 0)$ result in, $R \leq p$ and $0 \leq q$.

We now show how to write out an initial set of constraints as *Bane* constraints, starting from a program's AST. We will illustrate that the inferred constraints emerge automatically from *Bane*'s resolution procedure. We first define a 4-ary constructor $F : (-s, s, -s, s) \to s$. (The role of the four positions will be explained shortly.) In the following, keep in mind that the ordering imposed by our $\sqsubseteq$ connective is in the opposite sense of *Bane*'s $\leq$. See Table 4. For the purposes of the present discussion, we denote the variables of the *Bane* specification by $[p]$, $\langle \&p \rangle$, etc. Note that the $[p]$ variables in this formulation are connected to constructed terms $F(\ldots)$, whereas in the previous sections, $[p]$ was connected to $\langle q \rangle$ terms; however, values corresponding to the familiar meaning of $[p]$ can be obtained by projecting out the fourth argument from the constructed terms appearing in the solution for $[p]$ in this formulation.

	Abstract Syntax	Bane Constraints
(i)	$p = \&x$	$F([x], [x], \langle x \rangle, \langle x \rangle) \leq [p]$ and $\langle p \rangle \leq \langle \&x \rangle$
(ii)	$p = q$	$[q] \leq [p]$ and $\langle p \rangle \leq \langle q \rangle$
(iii)	$p = *q$	$[q] \leq F(0, [p], \langle p \rangle, 1)$
(iv)	$*p = q$	$[p] \leq F([q], 1, 0, \langle q \rangle)$
(v)	$p \to a$	$\text{PrefixInclusive}(\text{typeof}(*p), a) \leq \langle p \rangle$

Table 4: *Bane* constraints generated for normalized statements. The constructed type in the last row is actually concealed from *Bane* behind a 0-ary constructor unique to the constructed type.

The first two positions in the constructor F are used to store the set-type obligations, and the remaining two positions are used to store the get-type obligations. Within each pair, the first position is used as a contravariant argument and the second as a covariant argument. We select which of the two positions to use in such a way that the flow of set-type obligations is forwards and that of get-type obligations is backwards. It is easiest to explain this using an example.

Example 11. Consider the following two statements:

[9] As a matter of further detail, when a function's formal parameter is declared to be an array type, we must *un*lift the type of the actual parameters, to avoid lifting an array type to a pointer twice.

```
1. q = &p;
2. t = *q;
```

In Figure 11, the rule for *Dereference (R)* would propagate the get-type obligation on t to p by inferring a new constraint. We can see that a similar propagation in *Bane* occurs by contravariance: From (1), by rule (i) of Table 4, we have $F([p],[p],\langle p\rangle,\langle p\rangle) \leq [q]$ and from (2), by rule (iii) of Table 4, we have $[q] \leq F(0,[t],\langle t\rangle,1)$. By transitivity and the resolution of F, we have $[p] \leq [t]$ and $\langle t\rangle \leq \langle p\rangle$. These are equivalent to the relations that the rules in Figure 11 would infer:

$$\frac{t = *q \quad \dfrac{q = \&p}{[q] \sqsubseteq \langle p\rangle}}{\langle p\rangle \sqsubseteq \langle t\rangle \quad [t] \sqsubseteq [p]}$$

We chose to put $[t]$ and $\langle t\rangle$ in those particular positions in F so that the contravariance reverses the sense of the inclusion for $\langle t\rangle$, but covariance maintains the sense for $[t]$, giving us the desired relations. (Compare how rules (i) and (iii) of Table 4 interact, as illustrated above, with how rules (i) and (iv) interact.)

5.4 Circularity

The results of constraint solving might need post-processing to account for circularity. Consider the following two statements:

```
1. s = t + 1
2. t = s
```

These statements will generate the following constraints:

$$\langle s\rangle \sqsubseteq \mathit{Concat}(\mathit{typeof}\,(*t),\langle t\rangle)$$

$$\langle t\rangle \sqsubseteq \langle s\rangle$$

In this example, one cannot simply read off the answers for $\langle s\rangle$ and $\langle t\rangle$ from the outcome of constraint solving. (The constraint system treats the *Concat*-term as a constant and leaves it as such in the final set of constraints.) An implementation must perform an occurs-check of variable names inside constructed terms to detect cycles. In this particular example, both $\langle s\rangle$ and $\langle t\rangle$ evaluate to a divergent value ($\top$).

We have created a prototype implementation of our algorithm using *Bane*. Our plan is to tune the implementation and run it on large code bases, in particular, code from a Lucent Technologies product, and report our experiences.

6 Related Work

Related work falls into the five categories: static semantic checking tools; alternate type systems for C; record and variant subtyping; points-to and alias analysis in the presence of `structs` and `unions`; constraint-based analysis.

Static semantic checking tools Physical type-checking is related to, but complementary to, such tools as *lint* [JR78, Joh78] and *LCLint* [Eva96]. Our algorithm, as well as *lint* and *LCLint*, can be used to assist in the 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 [GS93, SV96, OJ95, SR96, Ste96]. Most of these references discuss the application

of *parametric polymorphism* to C. In particular, [SV96] concerns a new dialect of C that is polymorphic and type safe. [OJ95] uses polymorphic type inference on existing C programs, but for determining information about the transfer of values.

Physical type safety may also be contrasted with the work on typed assembly language of Morrisett et al. [MWCG98]. In that work, a type system is imposed on a RISC-like assembly language in such a way that individual registers and memory locations are assigned types to provide run-time type safety.

Ramalingam et al. [RFT99] have presented an algorithm to infer a particular kind of typing on structures. Their type system allows various degrees of coarsening of a structure: at one end, a structure may be treated atomically in a program, and at the other end, each field may be referred to separately. Their analysis attempts to find the coarsest admissible type for each structure in a program.

Comparison to Record Subtyping Our work has some similarity to record subtyping proposed by Cardelli [Car84]. 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 [AC96]. We plan to investigate this relationship in future work.

Relation to Points-to and Alias Analysis We already presented a comparison between flow-insensitive points-to analysis and physical-type-safety analysis in Section 4.3. In [ZRL96], which also presents an alias-analysis algorithm, a subtype-like relationship called a *weakly right-regular relation* is defined for pairs of C expressions. The definition of the relation has some of the flavor of the physical-subtyping rules discussed here. However, the system described in [ZRL96] has no provision for handling type casts and considers two `structs` to be related only if they are of equal type, whereas in our system the `structs` are related if one is a physical subtype of the other.

We would like to clear up a possible terminological confusion that may arise when our work is compared to that of Foster et al. [FFA97]. In that work, the use of "get" and "set" components of a type variable in the type-inference scheme corresponds to the covariant and contravariant aspects, respectively, of the same type variable. Our use of "get" and "set" corresponds to two different type variables, each of which themselves have covariant and contravariant components in a *Bane* implementation.

Constraint-based analyses The constraint-based analysis for inferring most general physical types gathers information about "access obligations" that a variable's type must satisfy. This bears some relationship to certain kinds of (backwards, flow-sensitive) need-based analyses developed in the functional-programming community, including algorithms for neededness [Hug88], strictness analysis [WH87], program slicing [RT96], and dependence analysis [Liu98]. 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.

7 Conclusions

We have presented an algorithm to perform physical type checking on C programs. Physical type checking can be useful for static safety checks, and also has application to program comprehension and reverse engineering.

In the near future, we plan to experiment with physical type checking on several large software systems, particularly on a proprietary product of Lucent Technologies that consists of several hundred thousands of lines of code.

Acknowledgments

We thank Michael Siff and Thomas Ball for their many contributions to this work. In particular, the material presented in Section 2 and the appendices is based on our joint previous work.

The second author is 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

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Notices*, 29(6):290–301, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer, 1996.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Car84] 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.

[Eva96] 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.

[FA97] Manuel Fähndrich and Alex Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, 1997.

[FFA97] 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.

[GS93] 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.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* The MIT Press, 1992.

[HS91] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice Hall, 1991.

[Hug88] 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.

[ISO90] ISO/IEC. *Programming langauges—C.* Number 9899. ISO/IEC, 1990.

[Joh78] S. C. Johnson. Lint, a C program checker, July 1978.

[JR78] S. C. Johnson and D. M. Ritchie. UNIX time-sharing system: Portability of C programs and the UNIX system. *Bell Systems Technical Journal*, 57(6):2021–2048, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[Liu98] Y.A. Liu. Dependence analysis for recursive data. *Proceedings of the 1998 IEEE International Conference on Computer Languages*, pages 206–215, May 1998.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, January 1998.

[OJ95] Robert O'Callahan and Daniel Jackson. Detecting shared representations using type inference. Technical Report CMU-CS-95-202, Carnegie Mellon University, September 1995.

[RFT99] 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.

[RT96] 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.

[SCKR99] 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`.

[SH97] 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.

[SR96] 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.

[Ste96] 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.

[SV96] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *1996 European Symposium on Programming*, April 1996.

[WH87] 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.

[WL95] 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.

[YHR99] 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.

[ZRL96] 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.

A Data Layout in the ANSI C standard

In this section, we briefly review the storage model for C data structures. For a more detailed account, the reader is directed to [HS91, KR88, ISO90].

All data objects in C are represented by an integral number of bytes in memory. The *size* of a data object is the number of bytes occupied by the data object [HS91]. A character type (`char`, `signed char`, `unsigned char`) is defined to occupy one byte of memory. The sizes of other C types are implementation dependent, but must conform to the following guidelines:

- `signed`, `unsigned`, `const`, and `volatile` qualifiers do not affect the size of a type. For example, an `unsigned int` is of the same size as a `const int`.
- `shorts` and `ints` require at least 16 bits.
- `shorts` are no longer than `ints`.
- `longs` require at least 32 bits.
- `ints` are no longer than `longs`.
- `floats`, `doubles`, and `long doubles` require at least 32 bits.
- a `union` requires at least as much storage as its largest member.
- a `struct` requires at least as much storage as the sum of the storage of its members, respecting the alignments of its members (see below).

Some computers allow an object to be stored at any address in memory, regardless of the type of the object. However, many computers impose *alignment restrictions* on certain data types. Some types are required to be stored at addresses that are integral multiples of bytes. If a data object is stored in accordance with its type's alignment restriction, expressions (other than casts) involving that object are *portable* across all C compilers compliant with the C standard. However, because of the ability of C programmers to cast an expression of one type to be of another type, it is possible to make an end-run around alignment restrictions resulting in *non-portable code.*

Character types have no alignment restrictions since they occupy only one byte of memory. The alignment restrictions of other C types are implementation dependent, but conform to the following guidelines:

- `signed`, `unsigned`, `const`, and `volatile` qualifiers do not affect the alignment of a type. For example, an `unsigned int` has the same alignment as a `const int`.
- The alignment of a `struct` or `union` is no less than the maximum alignment of its members.

The storage rules of `struct` types dictate that the first member be stored at the beginning of the `struct`. All subsequent members are stored in the order they are declared within the `struct`. The alignments of the types of the members of a `struct` may require that unused space be placed between members. For example, suppose a machine requires integers to be stored at four-byte multiples. Consider the following `struct`:

```
struct { char a;  int b;  char c; } x;
```

If the `struct` is stored at address 0, then `x.a` is at address 0, `x.b` is at address 4 (not address 1), and `x.c` is at address 8. Furthermore, the entire `struct` is padded to be a multiple of its largest alignment, resulting in a total size of 12 bytes.

We say that the number of bytes between the address of a field of a `struct` and the `struct` itself is the *offset* of the field. C guarantees that the offset of the first member (assuming it is not a bit field) of a `struct` is 0 and the offset of any other member (again, assuming it is not a bit field) is a multiple of the alignment of the type of that member. C guarantees that all non-bit-field members of unions are placed at offset 0.

The storage of bit-field members is implementation dependent. In our type system (see Figure 3) we assume that offsets are supplied explicitly. In our implementation (see Section 5) we assume that for `struct` types without bit fields, the members are stored as close together as possible without violating alignment restrictions.

B Defining Physical Type Safety

We define physical type safety using a scheme in which each pointer and pointer expression carries with it (as instrumentation) a *dynamic* type. This dynamic type is a pair (t, o), where t is a C type (i.e., from Figure 3) and o is an offset in this type. Given a pointer expression e, let $dtype(e)$ be e's dynamic type (t, o), where $dtype(e).t = t$ and $dtype(e).o = o$. The instrumentation scheme has been inspired by the SafeC tool [ABS94].

We define an auxiliary function $typeat(t, o)$ that returns the type of the object beginning at offset o for a `struct` t. If t is a ground type, pointer, or array then $typeat(t, o) = t$ if and only if $o = 0$, and is $\perp$ otherwise. If offset o is contained in a union u inside type t, we assume that the correct member of the union is chosen. Enforcing that the correct member of a union is chosen is an orthogonal issue that we do not deal with in this paper.

Dynamic types are created and propagated by the following rules:

- $dtype$(`malloc(sizeof(T))`) $= (T, 0)$
- $dtype$(`&(e->x)`) $= (dtype(e).t,\ dtype(e).o + offset(stype(e), x))$
- $dtype$(`&e`) $= (stype(e), 0)$.[10]
- If $dtype(e) = (t, o)$, then $dtype(e + c) = (t, o + c * sizeof(stype(e)))$
- After an assignment $p = e$, $dtype(p) = dtype(e)$

Note that pointer casts, explicit or implicit, do not affect the dynamic type of a pointer expression.

Given this instrumentation, a memory reference `e->x` is *physically type safe* if the offset of x is a *valid offset* with respect to $dtype(e)$. Let $o_x = offset(stype(e), x)$ and let $dtype(e) = (t_e, o_e)$. Offset o_x is a valid offset with respect to (t_e, o_e) if the following conditions are met:

- $o_x + sizeof(stype($`e->x`$)) \leq sizeof(typeat(t_e, o_e))$. The intuition behind this condition is that the referred memory location should lie within the "chunk" of memory that e currently points to.
- $typeat(stype(e), o_x) = typeat(t_e, o_e + o_x)$. The refered location must contain the same type of data as the type of field x.

Consider the application of this instrumentation to the last two statements in Figure 1. The dynamic type of `&p` is (`Point`,0), which is propagated to become the dynamic type of the pointer `pcp`. The dereference `pcp->x` is physically type safe because the type `Point` contains an integer at offset 0. The dereference `pcp->color` is not physically type safe because the offset of the field `color` is outside of the type `Point`.

We illustrate how pointers are tracked as they move within a `struct` using Figure 2. After execution of the first statement, the dynamic type of `c` is (`ClockRadio`,0). After execution of the second statement, the dynamic type of `r` is (`ClockRadio`,$sizeof$(`Clock`)), which represents the type `Radio`. Thus, the dereference `r->frequency` is physically type safe.

[10] This implies that this scheme does not track pointers that point into the middle of arrays.