Appears as Bell Laboratories Technical Report BL011350-990202-03.

Coping with Type Casts in C

Michael Siff*, Satish Chandra⊥, Thomas Ball⊥
Krishna Kunchithapadam⊤, Thomas Reps⊤

*Mathematics Dept
Wesleyan Univ.
msiff@wesleyan.edu

⊥Software Production Research Dept.
Bell Laboratories
{schandra,tball}@bell-labs.com

⊤Computer Sciences Dept.
Univ. of Wisconsin-Madison
{krisna,reps}@cs.wisc.edu

February 2, 1999

Abstract

The use of type casts is pervasive in C. Although casts provide great flexibility in writing code, their use obscures the meaning of a piece of code, and can present obstacles during maintenance. Casts involving pointers to structures (C `structs`) are particularly problematic, because by using them, a programmer can interpret any memory region to be of any desired type, thereby compromising C's already weak type system.

This paper presents an approach towards making sense of such casts, in terms of understanding their purpose and identifying fragile code. We base our approach on the observation that casts are often used to simulate object-oriented language features not supported directly in C. We first describe a variety of ways—*idioms*—in which this is done in C programs. We then develop a notion of *physical subtyping*, which provides a model that explains these idioms.

We have created tools that automatically analyze casts appearing in C programs. Experimental evidence collected by using these tools on a large amount of C code (over a million lines) shows that, of the casts involving `struct` types, most (over 90%) can be associated meaningfully—and automatically—with physical subtyping. Our results indicate that the idea of physical subtyping is useful in coping with casts and can lead to valuable software productivity tools.

1 Introduction

In the C programming language, a programmer can use a *cast* to coerce the type of a given expression into another type. Casts offer great flexibility to a programmer. In particular, because C allows a pointer of a given type to be cast into any other pointer type, a programmer can *reinterpret* the value at a memory location to be of any desired type. As a consequence, C programmers can—and often do—exploit the physical layout of structures (`structs`) in memory in various ways. Moreover, casts come with little or no performance cost, as most casts do not require extra machine code to be generated. The use of casts is pervasive in C programs.

A major problem with using casts is that it makes programs *difficult to understand*. Casts diminish the usefulness of type declarations in providing clues about the code. For example, a pointer variable can made to point to memory of a type unrelated to the variable's declared type. Another major problem is that casts make programs *fragile to modify*. Casts induce relationships between types that, at first glance, may appear to be unrelated to one another. As a result, it may not be safe for a programmer to add new fields to a `struct S`, because the code may rely on the memory layout of *other* `structs` that share a relationship with `S` through pointer casts.

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

typedef enum { RED, BLUE } color;

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

void translateX(Point *p, int dx)
{
  p->x += dx;
}

main()
{
  Point pt;
  ColorPoint cpt;
  ...
  translateX(&pt, 1);
  translateX((Point *) &cpt, 1);
}
```

Figure 1: A simple example of subtypes in C: `ColorPoint` can be thought of as a subtype of `Point`.

The preceding problems are exacerbated by that fact that there are currently no tools that assist a programmer in analyzing casts appearing in a piece of source code. C compilers do not check that the reinterpretation of memory via casts is done in a meaningful way. As stated before, C allows casts between any pair of pointer types. For the same reason, tools such as *lint* do not provide any help on seemingly inexplicable, yet legal casts.

This paper presents a semi-automatic approach to making sense of casts that involve pointers to `struct` types. We base our approach on the observation that casts involving pointers to `struct` types can often be considered as simulating subtyping, a language feature not found in C. This observation is supported by an analysis of over 1.3 million lines of C code containing over 7000 such casts. Our analysis examines each cast appearing in a program, and computes the relationship between the pair of C types involved in the cast. This relationship is usually an *up*-cast or a *down*-cast, but occasionally neither of the two. In the less frequent last case (we found 1053 total occurrences involving 127 unique pairs), the user must inspect the participating types manually. We have identified several patterns of usage occurring in C code, and have found that the seemingly unrelated types in the last case usually fall into one of these patterns.

Consider the C code shown in Figure 1. The function `translateX` is defined to take two arguments: `p` (a pointer to a `Point`) and `dx` (an integer). The function translates the horizontal component of the object `p` points to by `dx` units. Since `pt` is declared to be a `Point`, the expression `translateX(&pt,1)` is legal in C. We may also wish to apply `translateX` to a variable `cpt` of type `ColorPoint`, but the statement `translateX(&cpt,1)` is *not* (strictly speaking) legal in C. However, we can *cast* a pointer to a `ColorPoint` to be a pointer to a `Point` as shown in Figure 1. This works because of the way values of the types `Point` and `ColorPoint` are laid out in memory;[1] in effect, the cast of actual parameter `&cpt` in this call on `translateX` causes one type (i.e., `ColorPoint`) to be treated as a *subtype* of another type (i.e., `Point`).

The cast from `ColorPoint` to `Point` is an example of treating an instance of one type as an instance of another type — an "is a" relationship. In many programming languages (for example, C++), this relationship can be captured explicitly with subtyping. However, C has no such mechanism, so users who wish to capture the "is a" relationship rely on two things: type casts and the layout of data in memory.

In this example, our analysis explains the cast by reporting that `Colorpoint` and `Point` are involved in a subtype relationship. It also points out that the types `Colorpoint` and `Point` may not be modified independently of each other. For example, one cannot add a new field at the beginning of `Point`, and continue to use the function `translateX` on `ColorPoint`, unless the same field is also added to `ColorPoint`.

The contributions of this paper are as follows:

- We identify how type casts and the layout of data structures are used to simulate various object-oriented features in C. In particular, we present several commonly used *idioms* in C programs that represent C++-style object-oriented constructs and discuss the role of type casts in these idioms.
- We define the notion of *physical subtyping* and present rules by which the physical-subtype relationship may be inferred. Physical subtypes are important because they provide a model that captures most of the object-oriented casting idioms found in C.
- We describe a pair of software tools based on physical subtyping. `Cast-analyzer` classifies all the type casts in a program using the physical-subtype relationship. `Struct-analyzer` captures the physical subtyping relationships between all pairs of types in a C program. As we shall discuss later, a programmer can use these tools in combination, both for understanding the purpose of casts appearing in the program, and for discovering related types in the program that must be modified consistently.

We have run these tools on a number of large C programs taken from a variety of sources, including C programs from the SPEC95 benchmark suites, various GNU programs, and telephone call processing code from Lucent Technologies. Our tools and experimental results point the way to several software engineering applications of these tools:

- *To help programmers quickly learn about the relationships between data types in their programs.* The physical-subtype relationship can be naturally shown as a directed acyclic graph, which can be presented to a programmer to provide a visualization of the relationship between data types in their programs. We have created such visualizations for a few programs from our benchmarks (Section 4.3.1).
- *To identify fragile code.* In our experience, code containing casts that violate the physical subtype relationship is very fragile, because a programmer may introduce erroneous data references by using inconsistent type declarations. We present a detailed study of one such fragile cast identified in the telephone code (Section 4.2.2).
- *To aid in the conversion of C to object-oriented languages such as Java and C++.* The identification of physical subtypes in C programs provides a seed to the process of converting C programs to C++ or Java.

[1] The ANSI C standard makes certain guarantees about the layout of the fields of `structs` in memory. In particular, the first field of all `structs` is always allocated at offset zero, and *compatible* common prefixes of two `structs` are laid out identically. See Appendix B for more details.

Section 2 explains several common casting idioms by which C programmers emulate object-oriented programming. Section 3 presents a type system for C and formalizes physical subtypes by presenting a collection of inference rules. Section 4 describes our implementation of two complementary tools that identify physical subtypes, and the results of applying these tools to our benchmarks. Section 5 discusses related work.

Appendix A contains background material on type casts in C and Appendix B describes data layout in ANSI C.

2 Object-Oriented Idioms in C

In this section, we consider several object-oriented *idioms* that can be found with perhaps surprising frequency in C programs. These idioms emulate C++ features, such as *inheritance*, *class hierarchies*, and *virtual functions*.

2.1 Inheritance

Redundant declarations

C programmers can emulate public inheritance in a variety of ways. Perhaps the most common, at least for data types with a small number of members, is by declaring one `struct` type's member list to have another `struct` type's member list as a *prefix*. For example:

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

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

Instances of `ColorPoint` can be used in any context that allows the use of an instance of `Point`. Any valid context expecting a `Point` can, at most, refer to the `Point`'s `x` and `y` members. Any instance of `ColorPoint` has such `x` and `y` members at the same relative offsets as for an instance of `Point`.

First members

The use of redundant declarations is perhaps the simplest method of implementing subtyping in C. However, making a textual copy of the members of a *base* class in the body of each *derived* class is both cumbersome and error-prone. The *first-member idiom* represents an improvement that alleviates both of these problems.

Subtype relationships often characterize *is-a relationships*, as in "a color point *is-a* point". Members of `struct` types often characterize *has-a* relationships. For example, a `Person` *has-a* `name`:

```
typedef struct {
  ...
  char *name;
  ...
} Person;
```

However, because C guarantees that the first member of an object of a `struct` type begins at the same address at which the object itself begins, the first member can also reflect an is-a relationship. For example, consider this alternative definition of `ColorPoint`:

```
typedef struct {
  Point p;
  color c;
} ColorPoint;
```

Now a `ColorPoint` can be used where a `Point` is expected in two equivalent ways:

```
  ColorPoint cp;
  void translateX(Point *, int);

  translateX((Point *)&cp, 1);
  translateX(&(cp.p), 1);
```

In the second call to `translateX`, the reference to the `Point` component of `cp` is made more explicit (at the cost of having the programmer remember the names of the first member and modifying such code if and when the member names change).

Array padding

The first-member idiom may also be implemented in a slightly different manner, in which the allocation of storage space for the members of the *base* class is separated from the access to those members. Consider another definition for `ColorPoint`:

```
typedef struct {
  char base[sizeof (Point)];  /* storage space for a point */
  Color c;
} ColorPoint;
```

In this definition of `ColorPoint`, sufficient space is allocated to hold an entire `Point` rather than explicitly declaring a member of type `Point` (as in the first-member idiom) or using all the members of `Point` (as in the redundant-declaration idiom). The space is allocated by using a byte (`char`) array of the same size as `Point`.

This idiom is prevalent in several large systems that we have analyzed with our tools (described in Section 4), most notably *telephone* and *gcc*.

Flattening

We say that `struct` type s is a flattening of a `struct` type s' if the list of members of s is the result of replacing occurrences of `struct` members m within the list of members of s' by the list of the members of m. Sometimes subtyping relationships are implied by flattening. For example:

```
  typedef struct {
    Point p;
    color c;
  } ColorPoint;
```

flattens to

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

In this example, flattening can be seen as a transformation from the first-member idiom to the redundant-declaration idiom. However, flattening need not be confined to predefined `struct`s or to first members. For example:

```
typedef struct {
  int x;
  struct {
    int y;
    color c;
  } s;
} ColorPoint;
```

also flattens to:

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

Although this technique is not a very good way to implement subtypes or polymorphism (the idiom is complicated to understand and is rather error-prone), it has been used in some software projects (where, perhaps, two structures may have different logical groupings of the same list of fields). Identifying this idiom allows us to issue a diagnostic to the programmer (perhaps suggesting better ways to reorganize the data structures involved to make the object-oriented idiom more explicit and less error-prone).

Flattening is discussed further in Section 3.3.

2.2 Class hierarchies

It is not uncommon to find implicit class hierarchies in C programs using one or more of the inheritance idioms discussed above. One interesting combination is to use the first-member idiom for the top level of inheritance and then to use the redundant-declaration idiom for deeper levels of inheritance. For example:

```
typedef struct { .... } Point;

typedef struct {
  color c;
} AuxColor;
typedef struct {
  Point p;
  AuxColor aux;
} ColorPoint;

typedef struct {
  char name[10];
} AuxName;
typedef struct {
  Point p;
  AuxColor aux;
  AuxName aux2;
} NamedColorPoint;
```

Notice that NamedColorPoint can be thought of as a subclass of Point by the first-member idiom and as a subclass of ColorPoint by the redundant-declaration idiom.

Using the tools described in Section 4, we found that this idiom is prevalent in *xemacs* (a graphical-user-interface version of the text editor *emacs*).

2.3 Downcasts

It is a common object-oriented practice to allow objects of a derived class to be treated as if they are objects of a base class. This notion is referred to as an *upcast*—as in casting up from a subclass (subtype) to a superclass (supertype). The complementary notion of a *downcast* is not as common, but still very useful in object-oriented programming. A downcast causes an object of a base class to be treated as an object of a derived class, or in C, casts an expression of supertype down to a subtype.

The following is a simple example of a downcast:

```
void make_red(ColorPoint* cp) {
  cp->c = RED;
}
...
ColorPoint cp0;
Point* pp;
...
pp = &cp0;                      /* upcast from ColorPoint to Point */
...
make_red((ColorPoint *) pp);   /* downcast from Point to ColorPoint */
```

As this example illustrates, downcasts can be sensible in cases where type information has been lost through a previous upcast. The problem of identifying cases where downcasts are used without a preceding upcast is an aim of our future research.

2.4 Virtual functions

Downcasts are necessary in order to implement *virtual functions*, which are one of the most powerful aspects of object-oriented programming.

There are several ways in which virtual functions can be simulated in C. The most common is probably via the addition of run-time type information (RTTI) to data types in conjunction with switch statements that choose how a function should behave based on the RTTI.

As an example, consider the code fragment shown in Figure 2. In this example, Shape corresponds to an abstract base class, Circle and Rectangle are derived classes (using the first-member idiom), and the area function behaves as a virtual function in that it dynamically selects a specific area function to call depending on run-time type information stored in the kind field.

2.4.1 The +1 idiom

Another similar, but more complicated, mechanism for simulating virtual functions involves the use of pointer arithmetic. This idiom is illustrated in Figure 3. The example is based on a common idiom found in the *telephone* code discussed in Section 4. The idea in the example is that there are several kinds of messages that use a common message header (which includes run-time type information indicating the kind of message the header is attached to). In the process_msg function, the argument hdr is a pointer to a message header. hdr + 1 is a pointer-arithmetic expression referring to the address hdr plus (one times) the size of the object pointed to by hdr. In other words, hdr + 1 says

```
typedef enum { CIRCLE, RECTANGLE } shape_kind;

typedef struct {
  shape_kind kind;
} Shape;

typedef struct {
  Shape s;
  double radius;
} Circle;

typedef struct {
  Shape s;
  double length, width;
} Rectangle;

double circle_area(Circle *c) {
  return (3.14 * c->radius * c->radius);
}

double rectangle_area(Rectangle *r) {
  return (r->length * r->width);
}

double area(Shape *s) {
  switch(s->kind) {
  case CIRCLE:
    return (circle_area((Circle *)s)); /* downcast */
  case RECTANGLE:
    return (rectangle_area((Rectangle *)s)); /* downcast */
  }
}



int main() {
  int i;
  Shape *shape_array[5];
  double area_array[5];
  Circle c;
  Rectangle r;

  /* ... */

  shape_array[0] = (Shape *)&c; /* upcast */
  shape_array[1] = (Shape *)&r; /* upcast */
  /* ... */

  for(i = 0; i < 5; i++)
    area_array[i] = area(shape_array[i]);

  /* ... */
}
```


Figure 2: An example illustrating the use of explicit run-time type information to simulate virtual functions.

```
typedef struct {
  msg_hdr hdr;
  msg1_body body;
} msg1;

typedef struct {
  msg_hdr hdr;
  msg2_body body;
} msg2;

void processMsg1(msg1_body *);
void processMsg2(msg2_body *);

void processMsg(msg_hdr *hdr) {
  switch(hdr->kind) {
  case MSG1:
    processMsg1((msg1_body*)(hdr + 1));
    break;
  case MSG1:
    processMsg2((msg2_body*)(hdr + 1));
    break;
  /* ... */
  }
}
```

Figure 3: The `+1` idiom: An example illustrating the use of pointer arithmetic and run-time type information to simulate virtual functions.

"point to the the next member in the `struct` containing what `hdr` points to". An advantage to using the `+1` idiom is that the programmer need only remember the types of members, not member names.

By C's type rules, the expression `hdr + 1` has the same type as `hdr`. So the cast causes a pointer to type `msg_hdr` to be treated as either a pointer to `msg1_body` or a pointer to `msg2_body`. Because `msg_hdr` need not have anything in common with `msg1_body` or `msg2_body`, this idiom is rather confusing when first encountered. However, because of the way C lays out data in memory, the cast makes sense.[2]

It is one thing to identify an instance of the `+1` idiom, it is another thing entirely to determine if such an instance makes sense in terms of subtypes. The problem of making sense of casts such as these is outside the scope of this paper; however, we plan to address it in future research.

2.5 Generic Pointers in C

C programmers have long made use of generic pointers to achieve a limited form of polymorphism and subtyping. A generic pointer is much like the class `Object` in Java, for which all classes are subclasses: all pointer types may be thought of as subtypes of the generic pointer type. Prior to ANSI standardization, C programmers used pointers to a scalar type (usually `char*`) to represent generic pointers; now `void*` is the accepted type for generic pointers. The use of generic pointers is discussed further in Section 3.2 and Section 4.

[2] This assumes that the sizes and alignments of the `msg_hdr` and `msg_body` types are such that no padding is required between the `hdr` and `body` fields.

3 Physical Subtypes

Casts allow expressions of one type to be substituted for expressions of another type. In this respect, casts between types are reminiscent of subtype relationships often used in other programming languages (like C++).[3] In this section, we define the notion of *physical subtyping* and present rules for determining if one type is a physical subtype of another. The motivation for these rules is to be able to automatically identify *upcasts*: type casts from t to t', where t can be thought of as a subtype of t'.

The idea behind physical subtyping is that an expression of one type may be substituted for an expression of another type if, when the two are laid out in memory, 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 $t \prec t'$ 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`$\prec$`int`
  - `int`$\not\prec$`double`
  - `double`$\not\prec$`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.

3.1 A Type System for C

Our work addresses a slightly simplified version of the C type system:

- We ignore type qualifiers (e.g., `const int` and `volatile int` are treated as `int`).
- We consider typedefs to be synonyms for the types they redefine.

Types are described by the following language of type expressions:

[3] Substitution is a weaker notion than subtyping since it comes with no guarantee of expected behavior. The fact that a compiler allows an expression of one type to be used in place of an expression of another type does not preclude the occurrence of run-time errors.

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

[Void pointers] $$\overline{t \; \texttt{ptr} \prec \texttt{void*}}$$

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

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

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

Figure 4: Inference rules for physical subtypes.

t ::
  ground
| $t[n]$ // array of type t of size n
| t `ptr` // pointer to t
| $s\{m_1, \ldots, m_k\}$ // `struct`
| $u\{|m_1, \ldots, m_k|\}$ // `union`
| $(t_1, \ldots, t_k) \rightarrow t$ // function

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

ground ::
  $e\{id_1, \ldots, id_k\}$ // `enum`
| `void*` | `char` | `unsigned char`
| `short` | `int` | `long` | `double` | ...

Non-bit-field members 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 members of `union` types are considered to have offset 0. (See Appendix B)

3.2 Physical Subtyping Rules

Figure 4 presents rules in the style of [6] for inferring that one type is a physical subtype of another. We consider each of the physical-subtype rules individually:

- *Reflexivity*: Any type is a physical subtype of itself.
- *Void pointers*: A pointer to type t is a physical subtype of `void*`. Void pointers are generic: they can, by definition, only be used in contexts where any other pointer can be used. It is illegal

to dereference an object of type `void*`. In fact, the only legal operations on a void pointer that are cause for concern are type casts. For example:

```
Bar *b
Foo *f;
void *vp;
...
vp = (void *)b; /* upcast: Bar* is a subtype of void* */
...
f = (Foo *)vp;  /* downcast: Foo* is a supertype of void */
```

The cast from `void*` to `Foo*` is an example of a downcast, discussed in Section 2.3.

- *First members*: If t is a physical subtype of t' then a `struct` with a first member (the member at offset 0) of type t is a physical subtype of t'. This captures the first-member idiom described in Section 2. For example, assuming `ColorPoint` is a physical subtype of `Point`, then:

  ```
  typedef struct {
    ColorPoint cp;
    char *name;
  } NamedColorPoint;
  ```

  is a physical subtype of `ColorPoint` as well as a physical subtype of `Point`. (This example also illustrates the *transitivity* of the physical-subtype relation.)

- *Structures*: `struct` s with k members is a physical subtype of `struct` s' with k' members if:

  - s has no fewer members than s' (i.e., $k' \leq k$). (Note the *contravariance* between the direction of the subtype relation and the direction of the inequality between k and k'.)
  - Each member of s' has the same label, and the same offset as each of the corresponding members of s, and the types of members of s' are physical supertypes of the types of the corresponding members of s.

  For example:

  ```
  struct {
    int a;
    struct { double d1,d2,d3; } b;
    char c;
  }
  ```

  is a physical subtype of:

  ```
  struct {
    int a;
    struct { double d1,d2; } b;
  }
  ```

  This is in contrast with Cardelli-style structural subtyping between *record* types ([3, 1]). A record type is like a `struct` type, but the order of members (and therefore the layout in memory) is unimportant. A record type $\{l_1 : t_1, \ldots, l_k : t_k\}$ is a subtype of record type $\{l'_1 : t'_1, \ldots, l'_{k'} : t'_{k'}\}$ iff for each label l_i, there is a j such that $l_i = l'_j$ and t_i is a subtype of t_j.

3.3 Relaxing the Rules

The definition of physical subtypes in the previous section is not quite broad enough to capture some of the more complicated object-oriented idioms that arise in practice (see Section 2). In this section, we consider a somewhat relaxed definition of physical subtypes. The relaxed rules, which are tried only when the original ones are inconclusive, are both more expensive to implement and less precise than the original subtyping rules:

1. *Complexity.* By the original definition, the algorithm for determining if a type t is a physical subtype of t' has worst-case running time that is polynomial in the maximum of the sizes of t and t'. (Here, the "size of the type" refers to the size of representation of the type in our type system, not the amount of memory required to store the type.) For the more relaxed definition, for some cases (albeit rare ones), the running time complexity can be exponential.

2. *False Positives.* We can never be *sure* that a user intended for a cast to be an upcast or a downcast. Relaxing the rules increases the possibility that some casts will erroneously be identified as upcasts (or downcasts).

Member labels

It is sometimes the case that two `struct` types look very similar but differ in one or more member labels, as in the following definitions of `ColorPoint` and `MyColorPoint`:

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

This case can arise when more than one programmer is responsible for creating similar data types.

Flattening

Recall from Section 2.1 that a `struct` type s is a flattening of a `struct` type s' if the list of members of s is the result of replacing occurrences of `struct` members m within the list of members of s' by the list of the members of m, as in the following:

```
typedef struct {
  Point p;
  color c;
} ColorPoint;
```

flattens to

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

$$\text{[Flatten left]} \quad \frac{\begin{array}{l}\exists i \leq k : (\quad m_i = (l_i, \{m'_1, \ldots, m'_{k'}\}, d_i) \ \wedge \\ \qquad \forall j \leq k' : (m'_j = (l'_j, t'_j, d'_j) \ \wedge \ m''_j = (l'_j, t'_j, d'_j + d_i)) \ \wedge \\ \qquad \{m_1, \ldots, m_{i-1}, m''_1, \ldots, m''_{k'}, m_{i+1}, \ldots, m_k\} \prec t)\end{array}}{\{m_1, \ldots, m_k\} \prec t}$$

$$\text{[Flatten right]} \quad \frac{\begin{array}{l}\exists i \leq k : (\quad m_i = (l_i, \{m'_1, \ldots, m'_{k'}\}, d_i) \ \wedge \\ \qquad \forall j \leq k' : (m'_j = (l'_j, t'_j, d'_j) \ \wedge \ m''_j = (l'_j, t'_j, d'_j + d_i)) \ \wedge \\ \qquad t \prec \{m_1, \ldots, m_{i-1}, m''_1, \ldots, m''_{k'}, m_{i+1}, \ldots, m_k\})\end{array}}{t \prec \{m_1, \ldots, m_k\}}$$

Figure 5: Inference rules for physical subtypes with flattening.

There are numerous ways to flatten a `struct`. To name a few: flatten all member `structs`, no matter how deeply nested within the outer `struct`; flatten the first member that is a `struct`; or flatten all member `structs` that are not nested. Here, we consider two relaxed physical-subtyping rules for `structs` that incorporate flattening of member `structs` (but not member `unions`). Formally, we add the pair of subtype rules shown in Figure 5. Informally, the rule "Flatten left" states that a `struct` type s is a physical subtype of t if the flattening of s is a physical subtype of t. "Flatten right" states that a type t is a physical subtype of a `struct` type s if t is a physical subtype of the flattening of s.

For example:

```
struct S {
  int i;
  struct {
    int j,k;
  };
};
```

is a physical subtype of

```
struct T {
  int i, j;
};
```

because the flattening of `struct S`, namely

```
struct S {
  int i, j, k;
};
```

is a physical subtype of

```
struct T {
  int i, j;
};
```

4 Implementation and Results

In this section, we describe the basic implementation of the physical-subtype-analysis algorithm, as well as the cast-analyzer tool that is based on this algorithm. We then present our experimental results and discuss other applications of the physical-subtype-analysis algorithm.

4.1 Implementation

Our analysis tools were written in Standard ML. The tools act on data structures representing C types and abstract syntax trees. The abstract syntax trees can be generated from any preprocessed C program.

The core physical-subtype-analysis algorithm takes as input two types, t and t', and compares them to determine *to what extent* is t a physical subtype of t'. If t is not a physical subtype of t' according to the rules presented in Section 3.2, then *relaxations* of the subtyping rules are considered, as spelled out in Section 3.3. The algorithm returns a result in one of two forms:

1. *t is a physical subtype of t'*, together with numbers indicating how many of each of the relaxations rules have been invoked in order to identify the subtype relationship.

2. *t is not a physical subtype of t'.*

Given an abstract syntax tree representation of a C program, the `cast-analyzer` tool proceeds by traversing the abstract syntax tree and collecting the pairs of types associated with every implicit and explicit cast. For each pair of types, t and t', involved in a cast,[4] it returns one of three possible results:

1. *Upcast*: If the core physical-subtype-analysis algorithm returns that t is a subtype of t', then the tool returns "upcast".

2. *Downcast*: If the core algorithm returns that t is not a subtype of t', then the core algorithm is applied to see to what extent t' is a subtype of t. If the algorithm returns that t' is a subtype of t then the tool returns "downcast".

3. *Mismatch*: If the core algorithm determines that t is not a physical subtype of t' and t' is not a physical subtype of t (no matter how many relaxations are allowed), then the tool returns "mismatch".

The output of the cast analyzer is a list consisting of the following, for each occurrence of a cast:

- The location in the file where the cast occurred.
- The type being cast from.
- The type being cast to.
- The result of the physical-type-cast analyzer: upcast, downcast, or mismatch.
- The number of relaxations needed to identify the subtype relationship.

					Void-Struct			Struct-Struct		
Benchmark	**kLOC**	**Casts**	**Scalar**	**FunPtr**	**U**	**D**	**M**	**U**	**D**	**M**
binutils	516	3,426	2,088	41	399	678	32	32	156	0
xemacs	288	5,273	3,407	134	598	985	79	39	26	5
gcc	208	5,448	4,882	19	268	139	3	0	0	137
telephone	110	598	42	0	103	23	0	32	66	332
bash	76	642	346	126	44	78	1	17	22	8
vortex	67	3,827	3,143	42	495	83	14	40	9	1
ijpeg	31	1,260	571	14	15	59	0	464	137	0
perl	27	325	204	5	60	41	0	0	3	12
xkernel	37	3,148	1,021	66	771	702	409	64	95	20
Total	1,360	23,947	15,704	447	2,753	2,788	538	688	514	515

Table 1: **Total counts of casts in benchmarks**. **kLOC** is the number of source lines (in thousands) in the program, including comments. **Casts** is the total number of occurrences of casts, implicit and explicit, in the program. The remaining columns break this total down as follows: **Scalar** is the number of casts not involving a `struct union` or function pointer. **FunPtr** is the number of function pointer casts. **Void-Struct** represents casts in which exactly one of the types includes a `struct` or `union` type (the other being a pointer type such as `void *` or `char *`. **Struct-Struct** represents casts in which both types include a `struct` or `union` type. Each of these two categories is further classified as an upcast (U), downcast (D), or mismatch (M), as specified by the physical-subtype algorithm.

4.2 Experimental Results

We applied the cast analyzer to a number of C programs from the SPEC95 benchmarks (*gcc*, *ijpeg*, *perl*, *vortex*), as well as networking code *xkernel*, GNU's *bash*, *binutils*, and *xemacs*, and portions from a Lucent Technologies' product code *telephone*.

Table 1 summarizes the various benchmarks analyzed, in terms of their size, the total number occurrences of casts, and types of casts, as classified by the type-cast analyzer. Table 2 presents the cast numbers, but only counts casts between unique pairs of types. The number of casts in these programs, which represents a wide-variety of application domains, is non-trivial. Furthermore, we see that a large number of the casts are between pointers to `struct`s, evidence that programmers must reason about the physical-type relationships between `struct`s. Of these casts, the majority are upcasts and downcasts, but a substantial number are mismatches as well (i.e., there is no physical-subtype relationship between the two types at a cast between pointer-to-struct types).

Notice that a very high percentage (91 %) of the 1487 unique casts involving a `struct` type (see the last size columns of Table 2) can be classified automatically as either upcasts or downcasts. Furthermore, on simple manual inspection of mismatches, most of them turned out to be idioms indirectly involving physical subtyping (see Section 4.2.1). In a very small number of cases (fewer than 20) did a cast involving a pointer to `struct` appear completely unrelated to physical subtyping. These numbers provide evidence that the idea of physical subtyping is very useful in coping with casts appearing in C programs, and that the process of relating casts to subtyping can largely be automated.

[4]The cast appears in the program as $t*$ to $t'*$, because in C, references to `struct`s are stored as pointers.

					Void-Struct			Struct-Struct		
Benchmark	**kLOC**	**Casts**	**Scalar**	**FunPtr**	**U**	**D**	**M**	**U**	**D**	**M**
binutils	516	501	91	17	142	202	22	12	15	0
xemacs	288	409	67	55	119	135	20	3	7	3
gcc	208	129	44	10	20	48	2	0	0	5
telephone	110	135	18	0	49	11	0	11	4	42
bash	76	91	11	12	21	30	1	4	4	8
vortex	67	215	74	12	92	16	5	13	2	1
ijpeg	31	166	51	5	11	41	0	28	30	0
perl	27	40	4	2	15	15	0	0	1	3
xkernel	37	334	28	32	141	108	10	7	3	5
Total	1,360	2,020	388	145	610	606	60	78	66	67

Table 2: **Cast counts, for unique pairs of types.**

4.2.1 Examination of Mismatch Casts

After running the cast analyzer, we examined manually all of the cases for which the cast analyzer reported a mismatch. This found a number of questionable usages and interesting idioms (including the "`+1`" idiom described in Section 2.4.1), some of which we report on below.

The mismatches reported under the **Void-Struct** category were primarily due to the use of the type qualifier `const`: the cast analyzer reports a mismatch when an `struct S *` is cast into a `const void *` (or vice versa). There are a small number of mismatches due to other reasons. In *gcc* and *xemacs*, sometimes there is a cast to or from a partially defined structure,[5] which is reported under the **Void-Struct** category. We believe that in these cases, partially defined structures are used as a substitute for `void *`. In *xkernel*, the return type of certain functions ought to be valid pointers under normal conditions, but carry an `enum` signifying a status code under special conditions. Pointer values are thus compared to `enum` constants using a cast. Clearly, this usage is unrelated to physical subtyping.

The mismatches under the **Struct-Struct** category are more interesting. Most of them fall into one of the following patterns.

- A pointer to a `union` is cast to (or from) a pointer to one of the possible fields within the `union`. The cast analyzer does not compare a `union` type to any other type except a `void *`, because the selection of the "current interpretation" of the `union` is an important but orthogonal issue. We also found variations on this theme, such as a `struct` with a `union` as its last field, being cast into another `struct` with the same sequence of fields except the last one; the last field of the latter `struct` was one of the possible fields in the `union`.

- Upcast and downcast in the presence of bit-fields. The cast analyzer does not identify physical subtype relationships in the presence of bit-fields, because their memory layout is implementation dependent.

- The two `structs` participating in the cast have a common prefix. Consider an example from *bash*. There are several variants of a `struct command`, such as `for_command`, `while_command`, and `simple_command`. All these `structs` have a common first field. A function that needs to

[5] In C, one can reserve a structure tag name by declaring `struct t;` The name `t` is reserved as a tag name in the scope in which this declaration appears. The structure need not actually be defined anywhere at all. Such names are called partially defined structures, and are used, for example, to define a pair of structures that contain pointers to each other.

examine only the first field accepts all the variants of the command structs by the following trick: it declares its formal argument to type simple_command *, and at calls sites the actual argument is cast to type simple_command *. An alternative would be to declare a new base struct type containing only the first field. All the command variants would then appear as subtypes of the base type, and the function itself would be more evidently polymorphic by declaring its formal parameter to be a pointer to the base type.

- The "+1" idiom, as described previously in Section 2.4.1.
- The array padding idiom, as described previously in Section 2.1.

The last three patterns also relate to physical subtyping, albeit indirectly. In each of them we can identify a pair of types in play, such that one type acts as a base type and another a physical subtype. For example, in the "+1" idiom, if the cast converts a type A into type B, we can think of the base type as A and the subtype as struct { A a; B b; }. The subtyping relation in these patterns cannot be inferred by the rules in Section 3.2 (or their relaxations in Section 3.3). For a small number of exceptions (4 mismatches in *gcc*, 1 in *telephone*, 3 in *xkernel*, and 3 in *perl* code), we could not find any explanation at all. Perhaps a deeper understanding of the way these programs work is required to make sense of these few casts.

4.2.2 Telephone Code

This section presents an example of a potential type error found by the cast analyzer when applied to a large software system for telephone call processing. (The code presented here is not the actual code analyzed, but a distilled version that illustrates the essential features.)

Message passing is the common communication mechanism for telephone switching systems, which are massive distributed systems. Such a system may contain over a thousand different kinds of messages. Message formats in these systems generally follow the header-body paradigm: a header contains meta-information about the message; the body contains the contents of the message. The body itself may consist of another message, and so on. Messages are specified using structs and unions.

Typically, a "dispatch" procedure receives a message from the operating system. Depending on the contents of the header, the dispatcher will call other procedures that deal with specialized sets of messages and expect a pointer to a particular kind of message to be passed as an argument. Often, the dispatcher will "look ahead" into the body of message to find a commonly occurring case that requires immediate handling. For example, we found such a dispatch procedure that declared its view of messages as:

```
struct {
  header hdr;
  union {
    Msg1 m1;
    Msg2 m2;
    struct { int x; int y; } m3;
  } body;
} M;
```

There are three kinds of messages that can be nested inside Message, represented by M.body.m1, M.body.m2, and M.body.m3. The first and second messages reference typedef'd structs. Message m3 is declared inline. Now, the dispatcher contains the following code:

```
if (M.hdr.tag == 3 && M.body.m3.x == 1)
   process_m3(&M);      /* implicit cast */
```

where the function `process_m3` expects a pointer to the following structure:

```
typedef struct { header h; int x; char c; int y; } Msg3;
```

The cast analyzer flagged the implicit cast at the call as a "mismatch" because the second field of `Msg3` does not match the type of the second field of the anonymous `struct` represented by `M.body.m3`. Clearly, the code implies that these two types represent the same message, yet they are incompatible. If the dispatcher were to access field `m3.y` and the procedure `process_m3` had accessed `(&M)->c`, a physical type error would occur. A programmer simply examining the dispatcher may easily insert a reference to `m3.y`, oblivious to this problem.

4.2.3 Identifying Virtual Functions in *ijpeg*

The *ijpeg* benchmark suite (taken from SPECINT95) provides a good illustration of the use of abstract base classes and virtual functions (i.e., type-based code dispatch).

ijpeg provides a set of generic image-manipulation routines that convert an image from any one of a set of input formats to any one of a set of output formats (although the JPEG file format is the usual input or output type). Each image format is represented internally as an abstraction—images have a size, a colormap, a pixel array, etc. The image-manipulation routines are written in a fairly generic fashion, without reference to any specific image format. Components of an image are accessed or changed via calls through function pointers that are associated with each image object. The program initially sets up the input-image and the output-image objects with functions that are format-specific, and then passes pointers to the image objects to the generic image-manipulation routines.

The `main` function and the various *ijpeg* functions that it calls have no notion of the specific input-image type with which they are dealing. The selection of the input image type and the initialization of the relevant function pointers and data structures of instances of the `jpeg_compress_struct` type are done during the call to the `select_file_type` function. This separation simulates the object-oriented idiom of using abstract base classes and virtual functions to build extensible software libraries.

Each of the image-format-specific functions performs a *downcast* when the function is entered. By examining these downcasts (which were identified by the cast analyzer) we were able to track down the virtual-function idiom in *ijpeg*. After the downcast, the format-specific functions execute code that is particular to a given format.

4.3 Other Applications of Physical Subtypes

Given an abstract syntax tree representation of a C program, the `struct-analyzer` tool proceeds by traversing the abstract syntax tree and collecting a list of every `struct` type defined in the program. For each pair of `struct` types, t and t', the physcial-subtype algorithm is used to determine to what extent, if any, t is a subtype of t'. The result of a `struct` analysis is a list consisting of the following, for each pair of `struct` types for which some subtype relation has been identified:

- The subtype.
- The supertype.
- The size (in bytes) of the subtype.
- The size (in bytes) of the supertype.
- The difference between the two sizes.
- A list of numbers indicating the number of relaxations needed to identify the subtype relationship.

For the C-to-C++ conversion problem, the `struct` analyzer can help identify potential class hierarchies. The `struct` analyzer is also a good complement to the cast analyzer. Sometimes, implied subtype relationships are obfuscated by casts to and from generic types (usually `void*`).

`Struct` analysis can assist the manual tracking of such relationships. For example, given the following code:

```
typedef struct {
  int x,y;
} Point;
typedef enum BLUE, GREEN, RED color;
typedef struct {
  int xx,yy;
  color c;
} ColorPoint;

void foo()
{
  Point *pp;
  ColorPoint *cpp;
  void *vp;
  ...
  vp = cpp;  /*  upcast from ColorPoint* to void*  */
  ...
  pp = vp;   /*  downcast from void* to Point*     */
}
```

the cast anayzer produces the following output:

```
Line#  From Type       To Type     Result
-------------------------------------------
  16   ptr(ColorPoint) ptr(void)   upcast
  18   ptr(void)       ptr(Point)  downcast
```

This leaves us with a question: Is there a more interesting subtype relationship hiding beneath the `void` pointers? The result of applying the `struct` analyzer to the same program can shed some light:

```
                             Sizes          Relaxations
Subtype      Supertype   Sub  Super  Diff     L
---------------------------------------------------
ColorPoint   Point        12    8      4      2
```

This indicates that `ColorPoint` is a subtype of `Point` assuming two label-mismatch relaxations.

4.3.1 Visualizing the Results

For analzying larger systems, it is often useful to visualize the results of physical-subtype analysis graphically. The output of the `struct` analyzer can be displayed as a graph where vertices represent `structs` and there is an edge from t to t' if t is a physcial subtype of t'. Figure 6 shows a small example of such a graph from the SPEC95 benchmark *vortex*. This graph shows a small "class hierarchy": The class hierarchy is a tree with base class `typebasetype` and derived classes `integerdesctype`,

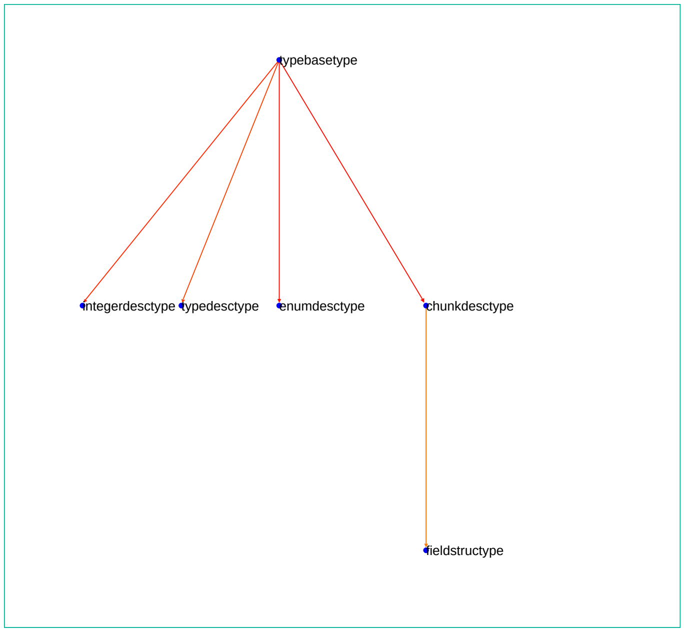

Figure 6: An example of the physical-subtype relation for *vortex*.

`typedesctype`, `enumdesctype`, and `chunkdesctype`, which has as a physical subtype the `struct fieldstructype`.

The output of the cast analyzer is also suitable for visualization as a graph. In this case, the vertices might represent types and edges upcast and downcast relationships. Figure 7 shows a set of upcasts found in the *vortex* benchmark. In this graph, a number of pointer types are cast to `Ptr(DrawObj)`, which is a pointer to `struct DrawObj`.

5 Related Work

Some of the ideas of this paper are also discussed in [13].

The idea of applying alternate type systems to C appears in several places, among them [5, 15, 12, 14, 16]. Most of these references discuss the application of *parametric polymorphism* to C, while in this paper we discuss the application of *subtype polymorphism* to C. The related work section in [14] describes related work pertaining to the application of parametric polymorphism to C.

The type system developed in this paper has similarities with several type systems proposed by Cardelli [2, 3, 1]. 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 physical layout does not apply. In particular, there are differences between our notion of `struct` subtyping and Cardelli's notion of record subtyping. In Cardelli's formulation, a record r is a subtype of a record r' if the set of labels occurring in r' is a subset of those occurring in r and if the type of the members of r' are supertypes of their corresponding members in r. In our system, a `struct` s is a subtype of a `struct` s' if the set of labels *and their offsets* of the members of s' is a subset of those occuring in s, the types of all but the last member of s' *match* the corresponding types in s (i.e., are supertypes and subtypes of the corresponding types in s), and the type of the last member of s' is a supertype of the

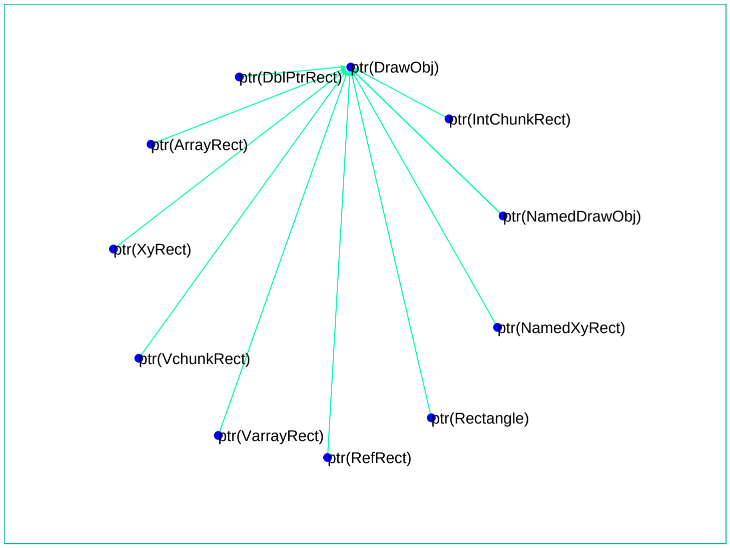

Figure 7: An example of a set of upcasts found in *vortex*.

corresponding member of s. (See Section 3.2.)

Steensgaard imposes a "non-standard" type system upon C programs and then uses type inference for the purpose of improving the efficiency and accuracy of points-to analysis [16].

The tools we have developed based on physical-subtype analysis (see Section 4) are related to, but complementary to, such tools as *lint* [10, 9] and *LCLint* [4]. Our tools, 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 requiring the user to add explicit annotations to the source code. On the other hand, neither *lint* nor *LCLint* has any notion of subtyping. *Lint* and *LCLint* can improve *cleanliness* of programs. The tools described herein can not only improve cleanliness, but can also help recognize fragile code.

6 Summary

This paper has discussed the problem of understanding type casts in C programs, and identifying fragile code. We have formally defined a notion of physical subtypes and built two tools using the definition that can automatically uncover physical-subtype relationships. We have also identified several idioms in C programs that represent C++-style object-oriented constructs.

In the future, we plan to explore program-analysis techniques that combine ideas from pointer analysis and physical subtyping to perform type-safety analysis on cast-intensive programs. We will also explore how to use these techniques to build other interesting software productivity tools.

References

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

[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] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[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] 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.

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

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

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

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

[10] 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.

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

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

[13] Michael Siff. *Techniques for Software Renovation.* PhD thesis, University of Wisconsin-Madison, August 1998.

[14] 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.

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

[16] 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.

A Type Casts in C

In C, a value that is of any pointer type may be substituted in an expression that expects an object of type `void*`. Because of implicit coercions and promotions, values of arithmetic types can be substituted in contexts that expect values of different arithmetic types. C does not allow all substitutions. For example, the following is illegal in C:

```
Point p;
ColorPoint cp;
...
p = cp;  /* error: incompatible types */
```

However, by the use of type-cast expressions, any value that is of a pointer type can be used in a context where a value of another pointer type is expected. So the following is legal (and reasonable) in C:

```
Point *pp;
ColorPoint *cpp;
...
pp = (Point *)cpp;  /* okay: explicit type cast */
```

Using pointer-type casts in conjunction with the address-of operator (`&`) and the indirection operator (`*`), any type can be cast to any other. For example:

```
Point p;
ColorPoint cp;
...
p = *((Point *)(&cp));
```

Many C compilers do not require explicit casts from one pointer type to another. A pointer of one type may be *implicitly type cast* to a pointer of another type (at most causing the compiler to issue a warning). Implicit type casts can occur in three places:

- Assignments:

```
Point *pp;
ColorPoint *cpp;

pp = cpp;   /*  implicit type cast  */
```

- Function calls:

```
ColorPoint *cpp;
void translateX(Point *, int);

translateX(cpp, 1);    /*  implicit type cast  */
```

- Function returns:

```
ColorPoint *cpp;

Point *getPoint() {
  /* ... */
  return cpp;    /*  implicit type cast  */
}
```

Implicit casts cause one commonly used compiler (`gcc`) to issue the warning "assignment from incompatible pointer type," but the code compiles nevertheless.

B Data Layout in the ANSI C standard

In this section, we briefly review the storage model for C data structures [7, 11, 8].

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 [7]. 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`.
- `short`s and `int`s require at least 16 bits.
- `short`s are no longer than `int`s.
- `long`s require at least 32 bits.
- `int`s are no longer than `long`s.
- `float`s, `double`s, and `long double`s 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 another type, it is possible to make an end-run around alignment restrictions, resulting in *non-portable code.* One of the advantages of physical subtypes is the following property: *a cast expression from type t to type t' where t is a physical subtype of t' can be used without loss of portability.*

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` accomodates 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 almost entirely implementation dependent. In particular, the presence of bit fields in a `struct` make it nearly impossible to predict how a compiler will choose the offsets of other members of a `struct`. In our type system (see Section 3.1) we assume that offsets are supplied explicitly. In our experiments (see Section 4) we assume that for `struct` types without bit fields, the members are stored as close together as possible without violating alignment restrictions.