Coping with Type Casts in C

Michael Siff¹, Satish Chandra², Thomas Ball², Krishna Kunchithapadam³, and Thomas Reps³

¹ Mathematics Department, Sarah Lawrence College, Bronxville, NY 10708-5999.  
msiff@mail.slc.edu  
² Bell Laboratories, Lucent Technologies, Naperville, IL 60566.  
{chandra,tball}@research.bell-labs.com  
³ Computer Sciences Department, University of Wisconsin, Madison, WI 53706.  
{krisna,reps}@cs.wisc.edu

Abstract. The use of type casts is pervasive in C. Although casts provide great flexibility in writing programs, their use obscures the meaning of programs, 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 for 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.
typedef struct {
  int x, y;
  Point;
} 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);
}

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

A major problem with casts is that they make programs difficult to understand. Casts diminish the usefulness of type declarations in providing clues about the code. For example, a pointer variable can be 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.

The preceding problems are exacerbated by the fact that there are currently no tools that assist a programmer in analyzing casts in C programs. 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 \textit{hint} 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 occurrences of 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 upcast or a downcast, but sometimes 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 Fig. 1. The function \textit{translateX} is defined to take two arguments: \textit{p} (a pointer to a \textit{Point}) and \textit{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 Fig. 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 sev-
  eral commonly used idioms in C programs that represent C++-style object-
  oriented concepts 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 impor-
tant 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. The cast-
  analyzer tool classifies all the type casts in a program using the physical-
  subtype relationship. The struct-analyzer tool captures the physical subtyp-
  ing 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 dis-
covering 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:

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.
- 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 graph, which can be presented to a programmer to provide a visualization of the relationship between data types in their programs. (See Section 4.3).
- 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).
- 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.

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.

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. This is illustrated by the Point and ColorPoint structs appearing in Fig. 1. 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 every 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 can 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 Sect. 4), most notably telephone and gcc.

Due to space limitations, another interesting inheritance idiom — flattening — is not discussed here. The reader is directed to [10] for more details.
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. An example appears in Fig. 2. Observe 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 Sect. 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:

```c
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.
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;

**Fig. 3.** An example illustrating the use of explicit run-time type information to simulate virtual functions

### 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 Fig. 3. 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.

**The +1 idiom** Another similar, but more complicated, mechanism for simulating virtual functions involves the use of pointer arithmetic. This idiom is illustrated in Fig. 4. The example is based on a common idiom found in the **telephone** code discussed in Sect. 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 “point to the the next member in the **struct** containing what **hdr** points to”.

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 MSG2:  
      processMsg2((msg2_body*)(hdr + 1));  
      break;  
    /* ... */  
  }  
}

Fig. 4. The +1 idiom: An example illustrating the use of pointer arithmetic and run-
time type information to simulate virtual functions

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 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 Sect. 3.2 and Sect. 4.

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++).² In this section, we

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 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
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; cp.y = 17; cp.c = RED;

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

<table>
<thead>
<tr>
<th>pt</th>
<th>3</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp</td>
<td>5</td>
<td>17</td>
</tr>
</tbody>
</table>

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 $\prec$ double
  - double $\prec$ 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 language of type expressions appearing in Fig. 5.
\[ t :: \\
  \text{ground} \\
  | \{n\} \quad /\text{array of type}\ t\ \text{of size} \ n \\
  | t\ \text{ptr} \quad /\text{pointer to}\ t \\
  | s\{m_1, \ldots, m_k\} \quad /\text{struct} \\
  | u\{m_1, \ldots, m_k\} \quad /\text{union} \\
  | (t_1, \ldots, t_n) \to t \quad /\text{function} \\
\]

\[ m :: \\
  | (t, l, i) \quad /\text{member labeled}\ l\ \text{of type}\ t\ \text{at offset}\ i \\
  | (l: n) \quad /\text{bit field labeled}\ l\ \text{of size}\ n \\
\]

\[ \text{ground} :: \\
  | e\{id_1, \ldots, id_k\} \quad /\text{enum} \\
  | \text{void*} | \text{char} | \text{unsigned char} | \text{short} | \text{int} | \text{long} | \text{double} | \ldots \\
\]

Fig. 5. A type system for C

Non-bit-field members of \text{struct} and \text{union} types are annotated with an \text{offset}. In a \text{struct}, the offset of a member indicates the difference in bytes between the storage location of this member and the first member of the \text{struct}. The first member is, by definition, at offset 0. All members of \text{union} types are considered to have offset 0.

3.2 Physical Subtyping Rules

Figure 6 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:

\textit{Reflexivity:} Any type is a physical subtype of itself.

\textit{Void pointers:} A pointer to type \( t \) is a physical subtype of \text{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 \text{void*}. In fact, the only legal operations on a void pointer that are cause for concern are type casts. For example:

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

The cast from \text{void*} to \text{Foo*} is an example of a downcast, discussed in Sect. 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 Sect. 2. For example, assuming ColorPoint is a physical subtype of Point, then:

define 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,

    \[
    \begin{align*}
    \text{struct} & \quad \text{struct} \\
    \text{int } a; & \quad \text{int } a; \\
    \text{struct} & \quad \text{struct} \\
    \{ \text{double } d1,d2,d3; \} & \quad \{ \text{double } d1,d2; \} \\
    \text{char } c; & \\
    \} & \\
    \} \\
    \end{align*}
    \]

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

In this section, we describe the basic implementation of the physical-subtype-analysis algorithm, as well as the cast-analysis 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

The analysis tools are 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 if \( t \) is a physical subtype of \( t' \) according to the rules presented in Sect. 3.2. The algorithm returns a result in one of two forms:

1. \( t \) is a physical subtype of \( t' \), together with numbers indicating how many times each of the subtyping 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-analysis 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,

\[ \text{it returns one of three possible results:} \]

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 physical subtype of \( t' \), then the core algorithm is applied to see to whether \( t' \) is a physical 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 \( t \), then the tool returns “mismatch”.

The output of the cast-analysis tool 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 cast analysis: upcast, downcast, or mismatch.

If the cast analysis results in an upcast or downcast, then the tool outputs, along with the above information, numbers indicating how many times each of the subtyping rules have been invoked in order to identify the physical-subtype relationship.

\[ \text{The cast appears in the program as } t \text{ ptr to } t' \text{ ptr, because in C, references to structs are stored as pointers.} \]
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. There are a total of 7,796 Void-Struct and Struct-Struct casts. Of these casts, 1,053 are classified as mismatches.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kLOC</th>
<th>Casts</th>
<th>Scalar</th>
<th>FunPtr</th>
<th>Void-Struct</th>
<th>Struct-Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>binutils</td>
<td>516</td>
<td>3,426</td>
<td>2,088</td>
<td>41</td>
<td>399</td>
<td>678</td>
</tr>
<tr>
<td>xemacs</td>
<td>288</td>
<td>5,273</td>
<td>3,407</td>
<td>134</td>
<td>598</td>
<td>985</td>
</tr>
<tr>
<td>gcc</td>
<td>208</td>
<td>5,448</td>
<td>4,882</td>
<td>19</td>
<td>268</td>
<td>139</td>
</tr>
<tr>
<td>telephone</td>
<td>110</td>
<td>598</td>
<td>42</td>
<td>0</td>
<td>103</td>
<td>23</td>
</tr>
<tr>
<td>bash</td>
<td>76</td>
<td>642</td>
<td>346</td>
<td>126</td>
<td>44</td>
<td>78</td>
</tr>
<tr>
<td>vortex</td>
<td>67</td>
<td>3,827</td>
<td>3,143</td>
<td>42</td>
<td>495</td>
<td>83</td>
</tr>
<tr>
<td>jpeg</td>
<td>31</td>
<td>1,260</td>
<td>571</td>
<td>14</td>
<td>15</td>
<td>59</td>
</tr>
<tr>
<td>perl</td>
<td>27</td>
<td>325</td>
<td>204</td>
<td>5</td>
<td>60</td>
<td>41</td>
</tr>
<tr>
<td>xkernel</td>
<td>37</td>
<td>3,148</td>
<td>1,021</td>
<td>66</td>
<td>771</td>
<td>702</td>
</tr>
<tr>
<td>Total</td>
<td>1,360</td>
<td>23,947</td>
<td>15,704</td>
<td>447</td>
<td>2,753</td>
<td>2,788</td>
</tr>
</tbody>
</table>

4.2 Experimental Results

We applied the cast-analyzer tool to a number of C programs from the SPEC95 benchmarks (gcc, jpeg, perl, vortex), as well as networking code xkernel, GNU’s bash, binutils, and xemacs, and portions of a Lucent Technologies’ product (identified here as 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 cast-analyzer tool. Table 2 presents the cast numbers, but only counts casts between unique pairs of types. The number of casts in these programs, which represent a wide-variety of application domains, is non-trivial. Furthermore, we see that a large number of the casts are between pointers to structs, evidence that programmers must reason about the physical-type relationships between structs. 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 six columns of Table 2) can be classified automatically as either upcasts or downcasts. Furthermore, on simple manual inspection of mismatches (discussed next), most of them turned out to be idioms indirectly involving physical subtyping. In only a very small number of cases (fewer than
Table 2. Cast counts, for unique pairs of types

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>kLOC</th>
<th>Casts</th>
<th>Scalar</th>
<th>FunPtr</th>
<th>Void-Struct</th>
<th>Struct-Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>U</td>
<td>D</td>
</tr>
<tr>
<td>binutils</td>
<td>516</td>
<td>501</td>
<td>91</td>
<td>17</td>
<td>142</td>
<td>202</td>
</tr>
<tr>
<td>xemacs</td>
<td>288</td>
<td>409</td>
<td>67</td>
<td>55</td>
<td>119</td>
<td>135</td>
</tr>
<tr>
<td>gcc</td>
<td>208</td>
<td>129</td>
<td>44</td>
<td>10</td>
<td>20</td>
<td>48</td>
</tr>
<tr>
<td>telephone</td>
<td>110</td>
<td>135</td>
<td>18</td>
<td>0</td>
<td>49</td>
<td>11</td>
</tr>
<tr>
<td>bash</td>
<td>76</td>
<td>91</td>
<td>11</td>
<td>12</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>vortex</td>
<td>67</td>
<td>215</td>
<td>74</td>
<td>12</td>
<td>92</td>
<td>16</td>
</tr>
<tr>
<td>ipage</td>
<td>31</td>
<td>166</td>
<td>51</td>
<td>5</td>
<td>11</td>
<td>41</td>
</tr>
<tr>
<td>perl</td>
<td>27</td>
<td>40</td>
<td>4</td>
<td>2</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>xkernel</td>
<td>37</td>
<td>334</td>
<td>28</td>
<td>32</td>
<td>141</td>
<td>108</td>
</tr>
<tr>
<td>Total</td>
<td>1,360</td>
<td>2,020</td>
<td>388</td>
<td>145</td>
<td>610</td>
<td>606</td>
</tr>
</tbody>
</table>

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.

Examination of Mismatch Casts After running the cast-analyzer tool, we examined manually all of the cases for which it reported a mismatch. This exposed a number of questionable usages and interesting idioms (including the “+1” idiom described in Sect. 2.4), 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 tool does not compare a union type 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.
any other type except a void*. The selection of the “current interpretation” of the union is an orthogonal but important 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 tool 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 but then diverge. 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 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 call sites the actual argument is cast to type simple_command*. (An alternative would have been 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 it would then be possible to make the polymorphic nature of the function more explicit, by declaring its formal parameter to be a pointer to the base type.)
- The “+1” idiom, as described previously in Sect. 2.4.
- The array padding idiom, as described previously in Sect. 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 Sect. 3.2.

For a small number of exceptions (4 mismatches in gcc, 1 in telephone, 3 in zkerneld, and 3 in perl code), we could not find any explanation at all.

**Telephone Code** This section discusses a mismatch found by the cast-analyzer tool when applied to telephone, a large software system for call processing. (The code presented here is not the actual code analyzed, but a distilled version that illustrates the essential features.) This mismatch highlights a potentially dangerous coding style that exists in this code.

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 a 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:

```c
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:

```c
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:

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

The `cast-analyzer` tool flagged the implicit cast at the call as a “mismatch” because the type of the field `c` of `Msg3` does not match type of the field `y` 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, oblivious to this problem, could easily insert a reference to `m3.y`.

**Identifying Virtual Functions in `ijpeg`**

`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). 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` tool, we were able to track down the virtual-function idiom in `jpeg`.

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 physical-subtype algorithm is used to determine if \( t \) 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.
- A list of numbers indicating how many times each of the subtyping rules have been invoked in order to identify the subtype relationship.

For the C-to-C++ conversion problem, the `struct-analyzer` tool can help identify potential class hierarchies. It is also a good complement to the `cast-analyzer` tool. 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 definitions of `Point` and `ColorPoint` shown in Fig. 1, `struct-analyzer` produces the following output:

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Supertype</th>
<th>Reflex Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ColorPoint</td>
<td>Point</td>
<td>2 1</td>
</tr>
</tbody>
</table>

This indicates that `ColorPoint` is a subtype of `Point` by one use of the structure rule and two uses of the reflexivity rule.

For analyzing larger systems, it is often useful to visualize the results of physical-subtype analysis graphically. The output of the `struct-analyzer` tool can be displayed as a graph where vertices represent `structs` and there is an edge from \( t \) to \( t' \) if \( t \) is a physical subtype of \( t' \). Figure 7(a) 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`, `typedesctype`, `enumdesctype`, and `chunkdesctype`, which has as a physical subtype `fieldstructype`.

The output of the `cast-analyzer` tool is also suitable for visualization as a graph. In this case, the vertices might represent types and edges upcast and
downcast relationships. Figure 7(b) 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

The idea of applying alternate type systems to C appears in several places, among them [5,12,9,11,13]. 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 [11] 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 occurring 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 corresponding member of \( s \). (See Sect. 3.2.)

The tools we have developed based on physical-subtyping are related to, but complementary to, such tools as *lint* [8,7] and *LCLint* [4]. Our tools, as
well as lint and LCLint, can be used to assist in static detection of type errors that escape the notice of many C compilers. LCLint can identify problems and constructs that our system cannot – for example, problems with dereferencing null pointers – but only by 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. Our tools can not only improve cleanliness, but can also help recognize fragile code.

Acknowledgements T. Reps is supported in part by the NSF under grants CCR-9625667 and CCR-9619219.

References