Unifying Type Checking and Property Checking for Low-Level Code

Jeremy Condit
Microsoft Research
jcondit@microsoft.com

Brian Hackett
Stanford University
bhackett@cs.stanford.edu

Shuvendu K. Lahiri
Microsoft Research
shuvendu@microsoft.com

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

Abstract

We present a unified approach to type checking and property checking for low-level code. Type checking for low-level code is challenging because type safety often depends on complex, program-specific invariants that are difficult for traditional type checkers to express. Conversely, property checking for low-level code is challenging because it is difficult to write concise specifications that distinguish between locations in an untyped program's heap. We address both problems simultaneously by implementing a type checker for low-level code as part of our property checker.

We present a low-level formalization of a C program's heap and its types that can be checked with an SMT solver, and we provide a decision procedure for checking type safety. Our type system is flexible enough to support a combination of nominal and structural subtyping for C, on a per-structure basis. We discuss several case studies that demonstrate the ability of this tool to express and check complex type invariants in low-level C code, including several small Windows device drivers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Type safety, property checking

Keywords SMT solvers, decision procedures, low-level code

1. Introduction

Despite the availability of safe, high-level languages, many of our most critical software systems are still written in low-level languages such as C and C++. Although these languages are very expressive and can be used to write high-performance code, they do not enforce type and memory safety, which makes them much less robust and much harder to analyze than higher-level languages.

Existing approaches to this problem, including *type checking* and *property checking*, have encountered a number of key challenges. Sound type checking for low-level code is challenging because type safety often depends on subtle, program-specific invariants. Although previous low-level type systems can be quite expressive [12, 23, 25], they are typically designed for a fixed set of programming idioms and are hard to adapt to the needs of a particular program. Likewise, property checking tools for low-level code can be quite powerful and quite general, but they either ignore types for soundness [10, 16] or rely on unproven type safety assumptions in order to achieve the necessary level of precision [4, 18].

In this paper, we address these challenges by implementing a *unified* type checker and property checker for low-level C code. The type checker can use the full power of the property checker to express and verify subtle, program-specific type and memory safety invariants, well beyond what the native C type system can check. Meanwhile, the property checker can rely on the type checker to provide structure and disambiguation for the program's heap, enabling more concise and more powerful type-based specifications. Our approach makes use of a fully automated Satisfiability Modulo Theories (SMT) [30] solver, which means that the programmer's only duty is to provide high-level type and property annotations as part of the original program's source.

The core idea behind our unified type and property checker is that we provide an explicit, low-level model of types and the type safety invariant. Our tool models the C program's heap using two maps that represent the data in the program's heap and the types at which each heap location was allocated:

$$\begin{gathered}\mathsf{Mem} : \mathsf{int} \rightarrow \mathsf{int} \\ \mathsf{Type} : \mathsf{int} \rightarrow \mathsf{type}\end{gathered}$$

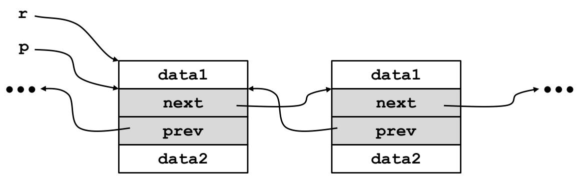
Our checker also defines a predicate called HasType, which indicates whether a given value corresponds to a given type, and we use this predicate to state the type safety invariant for the heap:

$$\forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$$

By asserting and checking this simple invariant at each program point, we can use the property checker to verify type safety in a flow-sensitive and path-sensitive manner. We also provide a decision procedure for the resulting type safety assertions.

This low-level representation of types and type safety has many benefits. First, the programmer can provide additional information about program-specific type invariants using the language of the property checker. Second, we can allow the programmer to define custom types as appropriate for a given program. Third, we can refine the types stored in the Type map in order to identify and distinguish structure fields that are important for checking higher-level properties of the code. In fact, when checking C structures, we can effectively choose between a nominal and a structural definition of type equivalence on a per-structure basis. Finally, because we encode the meaning of types directly in our translated program instead of relying on rules for deriving type judgments, our system does not require a complex, offline proof of soundness.

We implemented this technique as part of the HAVOC property checker [2], and we have applied it to a number of microbenchmarks and to several Windows device drivers of up to 1,500 lines of code, which can be verified, modulo a handful of unchecked assumptions, in about one minute each. This technique has allowed



```
struct list   { list *next; list *prev; }
struct record { int data1; list node; int data2; }

#define container(p) ((record*)((int*)(p) - 1))

void init_record(list *p) {
    record *r = container(p);
    r->data2 = 42;
}

void init_all_records(list *p) {
  while (p != NULL) {
    init_record(p);
    p = p->next;
  }
}
```

Figure 1. Example C code. The diagram shows two `record` structures in a linked list, with the embedded `list` shown in gray.

us to check complex spatial type and memory safety properties (i.e., safety in the presence of pointer arithmetic, casts, and linked data structures) that previous tools were incapable of expressing or checking; in addition, our property checker is now capable of exploiting concise, type-based annotations in proving properties of low-level code.

The contributions of this paper are:

- A low-level encoding of a C program's heap and types that allow a property checker to verify strong type safety properties.
- A semantics for C types that can be used in specifications for a sound property checker.
- A decision procedure for the type safety assertions generated by our encoding.
- Case studies evaluating the effectiveness of this technique on real C code, including small Windows device drivers.

In the next section, we present an example that demonstrates our technique in more detail. Then, we present the formal translation, our decision procedure for the resulting type safety assertions, and extensions that handle additional C features. Finally, we present case studies, discuss related work, and conclude.

2. Overview

In this section, we provide an overview of our technique using the sample code shown in Figure 1, which demonstrates a C language idiom commonly found in code such as the Windows kernel. The structure `list` represents a doubly-linked list with `prev` and `next` pointers, and it is intended to be embedded in the middle of a larger structure, such as the `record` structure. When initializing the elements of the list (`init_record` and `init_all_records`), the programmer must use pointer arithmetic and trusted type casts to compute the address and type of the enclosing record for each list node (as encapsulated by the `container` macro).

```
let Mem : int → int
let Type : int → type

let ctor Int : type
let ctor Ptr : type → type
let ctor List : type
let ctor Record : type

let Match : int × type → bool
Match($a$, Int) $\triangleq$ Type[$a$] $=$ Int
Match($a$, Ptr($t$)) $\triangleq$ Type[$a$] $=$ Ptr($t$)
Match($a$, List) $\triangleq$
  Match($a$, Ptr(List)) $\wedge$ Match($a+1$, Ptr(List))
Match($a$, Record) $\triangleq$
  Match($a$, Int) $\wedge$ Match($a+1$, List) $\wedge$ Match($a+3$, Int)

let HasType : int × type → bool
HasType($v$, Int) $\triangleq$ true
HasType($v$, Ptr($t$)) $\triangleq$ $v = 0 \vee (v > 0 \wedge$ Match($v, t$))

pre ($\forall a$ : int.HasType(Mem[$a$], Type[$a$]))
pre (HasType($p$, Ptr(List)))
post ($\forall a$ : int.HasType(Mem[$a$], Type[$a$]))
fun $init\_record$($p$ : int) : unit =
  let $r$ : int in
  $r := p - 1$;
  assert $\forall a$ : int.HasType(Mem[$a$], Type[$a$]);
  assert HasType($p$, Ptr(List));
  assert HasType($r$, Ptr(Record));
  Mem[$r + 3$] := 42;
  assert $\forall a$ : int.HasType(Mem[$a$], Type[$a$]);
  assert HasType($p$, Ptr(List));
  assert HasType($r$, Ptr(Record));
```

Figure 2. Translated BPL code for `init_record`.

The diagram at the top of Figure 1 illustrates a typical use of these data structures. In the diagram, we have two `record` objects in a list, with their embedded `list` objects shown in gray. Note that `next` and `prev` point to the embedded `list` objects, not the containing `record` objects. The `init_record` function computes the pointer `r` from the pointer `p` using the `container` macro.

There are two challenges presented by this example:

1. *Type checking.* Because of the arithmetic performed by the `container` macro, it is not obvious to a naive type checker that this code is well-typed; in fact, the well-typedness of this code relies on an unstated precondition about the lists that can be passed to `init_all_records`. Existing tools for enforcing type safety in C programs would have difficulty reasoning about this code because it relies on a program-specific invariant.
2. *Property checking.* If a property checker wanted to prove that `init_all_records` does not alter the contents of field `data1`, it would need to prove that the `data1` and `data2` fields of two structures can never be aliased. This fact is hard to prove without enforcing a strong type invariant throughout the program.

This section will present a step-by-step example showing how we address these challenges. Our technique translates the original C code (plus some optional user-supplied annotations) into a lower-level language called BPL that is suitable for input to a property checker. BPL has no notion of a heap or of C types, so this translation must model these constructs explicitly in BPL. Once generated, the BPL code can be passed to a property checking tool, which attempts to verify all assertions in this code.

2.1 Translating from C to BPL

Figure 2 shows the BPL translation produced for the `init_record` function. BPL has four built-in sorts: int, which represents values in the original C program, type, which represents types in the original C program, bool, which represents formulas, and unit, which is used by functions that do not return a value.

The core of our translation involves the maps Mem and Type, which are defined at the top of Figure 2. As mentioned in Section 1, Mem models the C program's heap as a mapping from integer addresses to integer values, and Type models the program's allocation state as a mapping from integer addresses to types. Our translation will enforce type safety by explicitly asserting the following type safety invariant at every program point:

$$\forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$$

This type safety invariant says that for every address a in the program's heap, the value at $\mathsf{Mem}[a]$ corresponds to the type at $\mathsf{Type}[a]$ according to the predicate HasType.

In order to define HasType, we must first discuss how our translation models C types. As shown in Figure 2, our translation defines nullary type constructors Int, List, and Record, which correspond to the built-in integer type and the user-defined `list` and `record` types. We also define the unary type constructor Ptr, which is used to construct pointer types such as Ptr(Int). Each type expression created by applying these constructors has a unique value.

Some of these type constants represent types that consume one word in memory (Int and $\mathsf{Ptr}(t)$), whereas others consume more than one word in memory (List and Record). Since Type gives the type for each individual word in memory, we define a new predicate, $\mathsf{Match}(a, t)$, that holds if and only if the values of Type starting at address a match the layout of type t. In other words, Match lifts Type to types that may span multiple addresses.

In our model, integers and pointers span only a single address in the heap, so Match for integers and pointers simply checks that Type has the appropriate value at address a. For List and Record, we define Match inductively by checking each field of the structure using its declared type. For example, $\mathsf{Match}(a, \mathsf{Record})$ holds if and only if the values of $\mathsf{Type}[a]$ through $\mathsf{Type}[a + 3]$ correspond to the declared types for the fields of the structure `record`.

Finally, we must define HasType itself. Since HasType only applies to types that span a single address, we define it only for Int and Ptr. For integers, all values are considered valid values of type Int. For pointers, the valid values include zero (the null pointer) and positive heap addresses that match the pointer's base type, as defined by Match.

Now that we have formalized the program's heap, the program's types, and our notion of type safety, we can translate the `init_record` function itself. The translated function has two preconditions: first, the type safety invariant holds on entry to the function, and second, the argument variable p has its declared type. This function also has a single postcondition, which simply says that the type safety invariant holds on exit as well.

Inside the body of the function, we declare a variable r corresponding to the variable in the original C program, and we translate the arithmetic and assignments in the C program into the corresponding operations on r and Mem. Note that all type casts have been eliminated during this translation. However, at every program point, we re-assert the type safety invariant, and we also assert that our local variables, r and p, still have their declared types.

2.2 Checking the Program

Now that we have a complete translation, we use a standard Floyd-Hoare verification condition generator, and we pass the verification condition to our SMT solver. Unfortunately, the code shown in Figure 2 has a problem: after executing the statement $r := p-1$, the assertion $\mathsf{HasType}(r, \mathsf{Ptr}(\mathsf{Record}))$ does not hold. Moreover, the next statement, which assigns to $\mathsf{Mem}[r+3]$, would violate the type safety invariant, since we cannot deduce a value for $\mathsf{Type}[r+3]$ and thus cannot prove that $\mathsf{HasType}(42, \mathsf{Type}[r + 3])$.

To address this problem, the programmer needs to provide more type information in the form of an extra precondition in the C code:

```
pre(hastype(container(p), record*) &&
    container(p) != 0)
```

This precondition is then translated into BPL as the following additional precondition on $init\_record$:

$$\mathsf{pre}\ (\mathsf{HasType}(p - 1, \mathsf{Ptr}(\mathsf{Record})) \wedge p - 1 \neq 0)$$

Note that this precondition is completely consistent with the existing preconditions, because the `record` type contains a `list` type at offset 1. With this precondition, we can prove that r has type Ptr(Record) after it is initialized. In addition, we can prove that $\mathsf{HasType}(r + 3, \mathsf{Int})$ using the type safety invariant and the definition of Match for Record.

So, by allowing the programmer to supply additional type information in the form of a precondition that refers to our HasType predicate, we have allowed the user to explain why their code is type-safe, and we have mechanically proven that type safety holds when given this precondition.

Although this example uses a very simple precondition, our technique exposes the full power of the SMT solver to the type checker. For example, when annotating `init_all_records`, we must not only state that the argument `p` points to a `list` embedded inside a `record`, but that all `list` objects reachable from `p` by following a `next` pointer also have this property. We can state this precondition in the C program as follows:

```
pre(forall(q, reach(p, next),
    q != 0 ==> hastype(container(q), record*) &&
               container(q) != 0))
```

Essentially, our technique allows us to describe complex program-specific invariants that are needed to prove the code to be type-safe.

2.3 Field Sensitivity

So far, we have focused on type safety alone; however, this technique also has many advantages for the property checking tool itself. Let's say that we want to prove that the `data1` fields of the `record` structures are untouched by `init_all_records`.

Unfortunately, because we represent C's heap as a single array of integers, such assertions are notoriously difficult to prove. For example, our theorem prover has no way to prove that the `data1` field does not happen to overlap with `data2` of some other record, and since `data2` is modified by `init_record`, we might inadvertently modify another record's `data1` field as well.

To address this problem, we allow the programmer to use a *field-sensitive* translation that introduces a new type constant for each word-sized field in the program. Our translation in Figure 2 would be extended with the following definitions:

$$\begin{array}{l}\mathsf{let\ ctor\ Data1 : type}\\ \mathsf{let\ ctor\ Data2 : type}\end{array}$$

$$\begin{array}{l}\mathsf{Match}(a, \mathsf{Data1}) \triangleq \mathsf{Type}[a] = \mathsf{Data1}\\ \mathsf{Match}(a, \mathsf{Data2}) \triangleq \mathsf{Type}[a] = \mathsf{Data2}\\ \mathsf{Match}(a, \mathsf{Record}) \triangleq\\ \quad \mathsf{Match}(a, \mathsf{Data1}) \wedge \mathsf{Match}(a + 1, \mathsf{List}) \wedge \mathsf{Match}(a + 3, \mathsf{Data2})\end{array}$$

$$\begin{array}{l}\mathsf{HasType}(v, \mathsf{Data1}) \triangleq \mathsf{HasType}(v, \mathsf{Int})\\ \mathsf{HasType}(v, \mathsf{Data2}) \triangleq \mathsf{HasType}(v, \mathsf{Int})\end{array}$$

Now we have two new type constants, Data1 and Data2, which represent the two integer fields of the `record` structure. Their

HasType definition is the same as the definition for Int, so they can still hold the same set of values. However, the Match definition for Record is altered so that Type must specify Data1 and Data2 at the appropriate offsets instead of just Int. The `next` and `prev` fields could also be made field-sensitive in the same way.

With this change, we can now prove that Data1 is not modified by these functions, because we can show that the only heap locations that are updated are locations a such that $\mathsf{Type}[a] = \mathsf{Data2}$.

This level of precision is extremely important for proving higher-level properties of C programs. For languages such as Java, disambiguation by field is taken for granted; our technique makes field disambiguation feasible in C programs as well. Furthermore, by tying disambiguation to our type safety invariant, we have a convenient way to enforce our invariant throughout the program, making use of the programmer's original type declarations.

From the type safety point of view, field sensitivity represents *nominal* type equivalence as opposed to *structural* type equivalence. In our original translation, any structure with the same layout as `record` would have matched that location, which corresponds to structural type equivalence; however, in the field-sensitive translation, only `record` structures may match that location. Both disciplines have their uses in C programs, and our technique allows the user to select the appropriate one on a per-structure basis.

Now that we have provided an overview of our technique and the associated contributions, the rest of this paper will define our translation formally, show that the resulting verification conditions are decidable, and provide extensions that can help the programmer address many common idioms in real C programs.

3. Translation

In this section, we will formally define our translation from C to our property checker's input language, BPL.

3.1 Languages

First, we define our input and output languages. The input language, shown in Figure 3, is a simplified version of the C language. The key C features modeled by this language are pointer types, structure types, the address-of operator (&), pointer arithmetic on a pointer to a type of size n ($\oplus_n$), and type casts ($(\tau)\ e$).

We have three primitive types: integer types (`int`), pointer types ($\sigma*$), and named structure types (`t`). The non-terminal τ stands for all types whose run-time representation fits in a single word (integers and pointers), and the non-terminal σ represents all types. For simplicity, we assume that the size of a word is 1.

Next we have l-expressions[1] and expressions, which include pointer dereference, field reference, address-of, pointer arithmetic, and casts. The symbol op represents binary operations, including both arithmetic and boolean operations. Commands include allocation, function call, assignment, and variable declaration.

At the top level, we have type definitions and procedures. Type definitions allow users to create named structure types that can be referenced within σ. Procedures take a single argument with a word-sized type, and they return a single value with a word-sized type. We annotate procedures with preconditions and postconditions that use expressions drawn from the same language; postconditions can refer to the return value via a special variable, r.

For the time being, we omit several other C features, such as scalar types of various sizes (e.g., `char`, `short`), union types, function pointers, and memory deallocation. We will revisit these features in Section 5. We disallow taking the address of local variables; in practice, our front-end replaces any local variables

[1] L-expressions are expressions that evaluate to locations and can therefore appear on the left-hand side of an assignment.

Types (one word)	τ	::=	$\mathtt{int} \mid \sigma*$
Types (general)	σ	::=	$\tau \mid \mathtt{t}$
L-expressions	l	::=	$*e \mid l.f$
Expressions	e	::=	$x \mid n \mid l \mid \&l$
		$\mid$	$e_1\ \mathtt{op}\ e_2 \mid e_1 \oplus_n e_2 \mid (\tau)\ e$
Commands	c	::=	$\mathtt{skip} \mid c_1; c_2 \mid x := \mathtt{new}\ \sigma$
		$\mid$	$x := f(e) \mid x := e \mid l := e$
		$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ c$
		$\mid$	$\mathtt{while}\ e\ \mathtt{do}\ c$
		$\mid$	$\mathtt{let}\ x : \tau\ \mathtt{in}\ c \mid \mathtt{return}\ e$
Type definitions	d	::=	$\mathtt{type\ t} = \{f_1 : \sigma_1; \ldots; f_n : \sigma_n\}$
Procedures	p	::=	$\mathtt{pre}\ e_1\ \mathtt{post}\ e_2$
			$f(x : \tau_x) : \tau_f = c$

Figure 3. Our C-like input language.

Sorts	$\hat{s}$	::=	$\mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \mathsf{type}$
		$\mid$	$\hat{s}_1 \times \hat{s}_2 \mid \hat{s}_1 \rightarrow \hat{s}_2$
Expressions	$\hat{e}$	::=	$x \mid x[\hat{e}] \mid n \mid \mathsf{C}(\hat{e}_1, \ldots, \hat{e}_n) \mid \hat{e}_1\ \mathsf{binop}\ \hat{e}_2$
Formulas	$\hat{b}$	::=	$\mathsf{true} \mid \mathsf{false} \mid \hat{e}_1\ \mathsf{relop}\ \hat{e}_2$
		$\mid$	$\mathsf{P}(\hat{e}_1, \ldots, \hat{e}_n) \mid \neg\hat{b} \mid \hat{b}_1 \wedge \hat{b}_2 \mid \forall x : \hat{s}.\hat{b}$
Commands	$\hat{c}$	::=	$\mathsf{skip} \mid \hat{c}_1; \hat{c}_2$
		$\mid$	$x := \mathsf{call}\ f(\hat{e}) \mid x := \hat{e} \mid x[\hat{e}_1] := \hat{e}_2$
		$\mid$	$\mathsf{if}\ \hat{e}\ \mathsf{then}\ \hat{c} \mid \mathsf{while}\ \hat{e}\ \mathsf{do}\ \hat{c}$
		$\mid$	$\mathsf{let}\ x : \hat{s}\ \mathsf{in}\ \hat{c} \mid \mathsf{return}\ \hat{e}$
		$\mid$	$\mathsf{assert}\ \hat{b} \mid \mathsf{assume}\ \hat{b} \mid \mathsf{havoc}\ x$
Procedures	$\hat{p}$	::=	$\mathsf{pre}\ \hat{b}_1\ \mathsf{post}\ \hat{b}_2$
			$\mathsf{fun}\ f(x : \hat{s}) : \hat{s} = \hat{c}$

Figure 4. BPL, our output language.

whose address is taken with a heap allocation. We do not mention global variables, but they are a trivial addition to our translation.

Our output language, BPL, is shown in Figure 4. This language has four built-in sorts, the most important of which are int and type. We also include product sorts and function sorts.

Expressions in BPL are of sort int or type, and include variable reference, map lookup ($x[\hat{e}]$), integer constants (n), type constructors (C), and binary operations on integers. Type constructors (C) typically include nullary type constants such as Int as well as the unary type constructor Ptr.

Formulas are of sort bool and contain relational operators on integers, predicate symbols (P), negation, conjunction, and universal quantification. Predicate symbols P typically include HasType and Match, which are used to define our notion of type safety. This language allows relatively unrestricted use of quantifiers, but in practice, our translation will be limited to a subset of these uses.

Commands contain assignment and control flow, plus assume $\hat{b}$, assert $\hat{b}$, and havoc x, the latter of which scrambles the value of x.

The most important differences between C and BPL are:

1. BPL has no notion of heap allocation. Thus, we model the C heap as a map Mem from integer addresses to integer values, and we use select-update reasoning to model reads and writes to the heap.

$$
\begin{aligned}
T(\texttt{int}) &= \mathsf{Int}\\
T(\tau*) &= \mathsf{Ptr}(T(\tau))\\
T(\texttt{t}) &= \mathsf{T}\\[1em]
L(*e) &= E(e)\\
L(l.f) &= L(l) + \mathsf{Offset}(f)\\[1em]
E(x) &= x\\
E(n) &= n\\
E(l) &= \mathsf{Mem}[L(l)]\\
E(\&l) &= L(l)\\
E(e_1 \;\mathsf{op}\; e_2) &= E(e_1) \;\mathsf{op}\; E(e_2)\\
E(e_1 \oplus_n e_2) &= E(e_1) + n * E(e_2)\\
E((\tau)\; e) &= E(e)\\[1em]
C(\Gamma, \texttt{skip}) &= \mathsf{skip}\\
C(\Gamma, c_1; c_2) &= C(\Gamma, c_1); C(\Gamma, c_2)\\
C(\Gamma, x := \texttt{new}\; \sigma) &= \mathsf{havoc}\; x;\\
&\; \mathsf{assume\ HasType}(x, \mathsf{Ptr}(T(\sigma)))\\
C(\Gamma, x := f(e)) &= x := \mathsf{call}\; f(E(e));\\
&\; \mathsf{assert\ HasType}(x, T(\tau_f))\\
C(\Gamma, x := e) &= x := E(e);\\
&\; \mathsf{assert\ HasType}(x, T(\Gamma(x)))\\
C(\Gamma, l := e) &= \mathsf{Mem}[L(l)] := E(e);\\
&\; \mathsf{assert}\; \forall a{:}\mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])\\
C(\Gamma, \texttt{if}\; e\; \texttt{then}\; c) &= \mathsf{if}\; E(e)\; \mathsf{then}\; C(\Gamma, c)\\
C(\Gamma, \texttt{while}\; e\; \texttt{do}\; c) &= \mathsf{while}\; E(e)\; \mathsf{do}\; C(\Gamma, c)\\
C(\Gamma, \texttt{let}\; x : \tau\; \texttt{in}\; c) &= \mathsf{let}\; x : \mathsf{int\ in}\; C(\Gamma[x \mapsto \tau], c)\\
C(\Gamma, \texttt{return}\; e) &= \mathsf{return}\; E(e)
\end{aligned}
$$

$$
P\begin{pmatrix} \texttt{pre}\; e_1\; \texttt{post}\; e_2 \\ f(x : \tau_x) : \tau_f \\ = c \end{pmatrix} =
\begin{array}{l}
\mathsf{pre}\; (E(e_1) \wedge \mathsf{HasType}(x, T(\tau_x)) \wedge \\
\quad \forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a]))\\
\mathsf{post}\; (E(e_2) \wedge \mathsf{HasType}(r, T(\tau_f)) \wedge \\
\quad \forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a]))\\
\mathsf{fun}\; f(x : \mathsf{int}) : \mathsf{int} = C(\emptyset[x \mapsto \tau_x], c)
\end{array}
$$

Figure 5. Translation from C to BPL.

2. BPL has no notion of pointer types or structure types. Instead, BPL provides the sorts int and type, which we use to represent the original program's values and types, respectively. That is, all word-sized values in the original program map to values of sort int, and all C types in the original program map to values of sort type.

Figure 5 shows our translation from C to BPL. The translation involves five functions, as follows. First, T maps C types to BPL expressions of sort type. Note that each named type `t` in the C program is mapped to a distinct constant T in the BPL program.

L and E map l-expressions and expressions to BPL expressions of sort int. L yields integers that stand for heap locations, so E translates the expression l as a memory reference and $\&l$ as the location itself. Note that C's binary operations map to a BPL operator in op = binop ∪ relop. Field references and pointer arithmetic are compiled down to integer arithmetic; casts are compiled away entirely. Offset(f) is a compile-time function giving the offset of field f in its structure; we assume field names are unique.

C and P map commands and procedures in C to their respective constructs in BPL. C takes an additional argument, Γ, that maps C variables to C types. Also, we assume that τ_f is the declared return type of function f, that the variable r in a postcondition refers to the function's return value, and that procedure calls scramble all of Mem. Ignoring the assumptions and assertions in gray boxes, which will be discussed in the next section, this translation is a straightforward modeling of C's operational semantics. For simplicity, allocation is modeled conservatively by scrambling the value in x; however, our implementation models allocation more precisely, as discussed in Section 5.5.

After this translation, we can compute a verification condition from the BPL program using standard techniques [7, 14]. Then we can pass it to our SMT solver, which indicates whether the program fails any of the assertions.

Definitions for Int

$$
\begin{aligned}
&\mathsf{Match}(a, \mathsf{Int}) \triangleq \mathsf{Type}[a] = \mathsf{Int} && (A)\\
&\mathsf{HasType}(v, \mathsf{Int}) \triangleq \mathsf{true} && (B)
\end{aligned}
$$

Definitions for Ptr(t)

$$
\begin{aligned}
&\mathsf{Match}(a, \mathsf{Ptr}(t)) \triangleq \mathsf{Type}[a] = \mathsf{Ptr}(t) && (C)\\
&\mathsf{HasType}(v, \mathsf{Ptr}(t)) \triangleq v = 0 \vee (v > 0 \wedge \mathsf{Match}(v, t)) && (D)
\end{aligned}
$$

Definitions for $\texttt{type t} = \{f_1 : \sigma_1; \ldots; f_n : \sigma_n\}$

$$
\mathsf{Match}(a, \mathsf{T}) \triangleq \bigwedge_i \mathsf{Match}(a + \mathsf{Offset}(f_i), T(\sigma_i)) \qquad (E)
$$

Figure 6. Definition of HasType and Match for a, v of sort int and t of sort type.

3.2 Modeling Type Safety

We now discuss our approach to enforcing type safety, which involves the assumptions and assertions shown in gray boxes in Figure 5. First, however, we must discuss our representation of the heap and of C types in BPL.

We assume the presence in BPL of the following two maps:

$$
\begin{aligned}
\mathsf{Mem} &: \mathsf{int} \to \mathsf{int}\\
\mathsf{Type} &: \mathsf{int} \to \mathsf{type}
\end{aligned}
$$

As described earlier, Mem[a] represents the value in the heap at address a, and Type[a] represents the type at which address a was allocated. Although Mem is mutable, Type is fixed at allocation time and cannot later be changed.

C types are modeled in BPL as inductive data types with sort type. We have a nullary constructor Int for integer types as well as a unary constructor Ptr(t) for pointer types. We also introduce nullary constructors T for every user-defined type name `t`.

Now, we must assign a meaning to these types, and we do so by introducing two new predicates:

$$
\begin{aligned}
\mathsf{Match} &: (\mathsf{int} \times \mathsf{type}) \to \mathsf{bool}\\
\mathsf{HasType} &: (\mathsf{int} \times \mathsf{type}) \to \mathsf{bool}
\end{aligned}
$$

As described earlier, the Match predicate lifts Type to types that span multiple addresses. Formally, for address a and type t, Match(a, t) holds if and only if the Type map starting at address a matches the type t. The HasType predicate gives the meaning of a type. For a word-sized value v and a word-sized type t, HasType(v, t) holds if and only if the value v has type t.

The definitions of Match and HasType are given in Figure 6. For Match, the definitions are straightforward: if a given type is a word-sized type, we check Type at the appropriate address, and for structure types, we apply Match inductively to each field. For HasType, we only need definitions for word-sized types. For integers, we allow all values to be of integer type, and for pointers, we allow either zero (the null pointer) or a positive address such that the allocation state (as given by Match) matches the pointer's base type. HasType is the core of our technique, since it explicitly defines the correspondence between values and types.

Now that we have defined HasType, we can state our type safety invariant for the heap:

$$\forall a : \mathsf{int}.\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$$

In other words, for all addresses a in the heap, the value at $\mathsf{Mem}[a]$ must correspond to the type at $\mathsf{Type}[a]$ according to the HasType axioms. We can also extend this type safety invariant to local variables by saying that for all locals x with compile-time type τ_x, then $\mathsf{HasType}(x, T(\tau_x))$ must hold, where T is our translation from C types to BPL terms.

Our translation enforces this invariant at all program points via the gray boxes shown in Figure 5. P adds the type safety invariant to the preconditions and postconditions of each procedure. C asserts the type safety invariant after every update to a local variable or a heap location, and it assumes the type safety invariant for any newly allocated heap location.

3.3 Field Sensitivity

In addition to proving type safety for our input program, we would also like to check properties that are specified by the user as preconditions and postconditions for each function. Property checking in the presence of heap-allocated structures often requires us to be able to distinguish between two fields of a structure; for example, in Figure 1, we would like to be able to show that writing to `data2` does not affect the values in the `next`, `prev`, and `data1` fields of other records in the program's heap.

As described in Section 2, our approach to this problem is to introduce a new type constant for every word-sized structure field in the program. In effect, we refine the types stored in Type so that it captures information about specific structure fields in addition to the types of those fields. For example, we introduce constants Data1 and Data2, and we use these constants in Type to correspond to the `data1` and `data2` fields. The definition of HasType for these fields is the same as that of the underlying type, Int, which means that the type safety invariant provides the same amount of information about the values stored in these fields as it did before. However, because the Type map now differs for these two fields, our property checker knows that the `data1` and `data2` fields of two different structures cannot overlap in memory. We can perform the same refinement on `next` and `prev` as well.

Using this field-sensitive translation involves a trade-off between precision and flexibility. On the one hand, field sensitivity provides a stronger invariant to the theorem prover, which can often be useful in distinguishing one heap location from another. On the other hand, field sensitivity restricts the ways in which two C structures can overlap in the heap.

This trade-off corresponds to the trade-off between *nominal* and *structural* type systems. In the field-sensitive translation, equivalence between structure types is determined by the name of that structure (or, more precisely, by the names of its fields). In the original field-insensitive translation, equivalence is determined by structure—that is, by the types of the fields alone.

Note that our translation does not require the user to choose field-sensitive or field-insensitive behavior for the entire program. Rather, the programmer is free to choose a field-sensitive or field-insensitive translation on a field-by-field basis. This flexibility is often quite useful when checking C programs, since many C programs use a combination of these two approaches. In our experience, the majority of structures in C programs are handled using nominal type equivalence, where overlapping structures must be of the exact same named structure type; structural type equivalence is only used in relatively rare cases where the programmer deliberately overlaps two distinct structures types that share a common header. Thus, our implementation uses the more precise and more common field-sensitive translation by default, and it allows the programmer to specify the fields and structures that should be handled using the field-insensitive translation instead.

$$\begin{array}{lcl}
b & \in & BoolConst = \{\mathsf{false}, \mathsf{true}\} \\
c & \in & IntConst = \{\ldots, -1, 0, 1, \ldots\} \\
t & \in & ITypeConst = \{\mathsf{Int}, \mathsf{List}, \mathsf{Record}, \ldots\} \\
d & \in & TypeConst \supset ITypeConst \\
\\
w & \in & BoolVar \\
x & \in & IntVar \\
y & \in & TypeVar \\
\\
\varphi & ::= & b \mid w \mid p < p \mid p = p \mid \mathsf{Match}(p, q) \mid \mathsf{HasType}(p, q) \mid \\
 & & \neg\varphi \mid \varphi \wedge \varphi \\
p & ::= & c \mid x \mid p + p \mid p - p \mid \mathsf{Mem}[p] \\
q & ::= & d \mid y \mid \mathsf{Ptr}(q) \mid \mathsf{Type}[p]
\end{array}$$

Figure 7. Grammar for verification conditions generated by our translation.

4. Decision Procedure

In this section, we describe the decision procedure used to check the verification conditions generated from the output of the translation described in Section 3. The verification condition corresponds to a formula that encodes the partial correctness of a loop-free and a call-free code fragment annotated with a precondition and a postcondition, where the annotations can optionally refer to quantifier-free assertions apart from the type safety assertion. Each verification condition is represented by a formula in the logic of Figure 7. Without loss of generality, we assume that updates to Mem have been compiled away by introducing case-splits to model the select-update reasoning for arrays. Our logic contains three sorts: bool, int, and type. Terms of these sorts are generated by the non-terminals φ, p, and q, respectively. $BoolConst$, the set of constants of sort bool and $IntConst$, the set of constants of sort int are defined in the usual way. $ITypeConst$ is the set of (interpreted) type constants that occur in the program and which are referred to in the definitions of the predicates Match and HasType in Figure 6. For example, $ITypeConst = \{\mathsf{Int}, \mathsf{List}, \mathsf{Record}\}$ for our running example from Figure 2. $TypeConst$ is the set of all type constants of sort type and is a countably infinite set that contains $ITypeConst$.

A formula in our logic is evaluated in a model that provides a domain for each sort bool, int, and type. The domains for the sorts bool and int are standard. The domain for the sort type is the unique infinite set whose elements are in one-to-one correspondence with the least set of terms containing all type constants in $TypeConst$ and closed under the application of Ptr. In this interpretation, each type constant in $TypeConst$ is interpreted as a *distinct* type value and Ptr is interpreted as a one-one map from type into type whose range is *disjoint* from the interpretations of the type constants in $TypeConst$. Let $PtrTypeVals = \mathsf{type} \setminus TypeConst$ be the set of all type values that are in the range of Ptr. Let $UTypeConst = TypeConst \setminus ITypeConst$ be the set of (uninterpreted) type constants that do not occur in the program. Then, the disjoint union $ITypeConst \uplus UTypeConst \uplus PtrTypeVals$ equals type.

In addition, a model also provides an interpretation for constants, variables, and functions. The interpretation of the arithmetic and Boolean terms is standard. Functions Mem and Type are interpreted as arbitrary maps from int to int and int to type, respectively. Predicates Match and HasType are interpreted as maps from $\mathsf{int} \times \mathsf{type}$ to bool. Given a model Γ, we denote the interpretation of a symbol s in the signature of our logic as $\Gamma(s)$. For ease of exposition, we often use s rather than $\Gamma(s)$ for those symbols, such as $+$, whose interpretation does not vary from one model to another.

A model Γ is *well-typed* if the following conditions are satisfied:

1. $\Gamma(\mathsf{Match})$ and $\Gamma(\mathsf{HasType})$ are consistent with the definitions of Match and HasType in Figure 6.
2. For all $a \in \mathsf{int}$, the evaluation of $\mathsf{HasType}(\mathsf{Mem}[a], \mathsf{Type}[a])$ in Γ returns $true$.

A model Γ *satisfies* a formula φ if φ evaluates to $true$ in Γ.

The logic in Figure 7 is the quantifier-free combination of three theories—uninterpreted functions, arithmetic, and inductive data types—with disjoint sets of symbols. Each of these theories is stably infinite[2] and individually decidable. Hence, their combination is also decidable using the Nelson-Oppen method [27] of theory combination. Satisfiability Modulo Theories (SMT) solvers such as Z3 [13] can be used to efficiently check whether there exists a model of φ. However, deciding the existence of an arbitrary model of φ does not suffice; instead we need to determine whether there exists a well-typed model of φ. Conjoining the type-invariant and the definitions of Match and HasType to φ as universally-quantified axioms is unlikely to work well because the performance of SMT solvers on formulas with quantifiers is unpredictable and typically bad. To get good performance, we have designed a new decision procedure that conjoins a small number of instantiations of the universally-quantified facts to φ to get a quantifier-free formula ψ with the following property:

> There is a well-typed model satisfying φ iff there is a model satisfying ψ.

Thus, it suffices to feed ψ to an SMT solver. We now show how to construct ψ.

4.1 Quantifier Instantiation

Let $P(\varphi)$ denote the set containing the constant 0 and every term p in φ such that for some term q either $\mathsf{HasType}(p, q)$ or $\mathsf{Match}(p, q)$ is present in φ. Let $Q(\varphi)$ denote the set containing $ITypeConst$ and every term of the form $\mathsf{Ptr}(q')$ in φ. We will use the terms in $P(\varphi)$ and $Q(\varphi)$ to instantiate the definitions A, B, C, D, E from Figure 6.

First, we preprocess each definition E so that every use of Match on the right side of the definition is expanded out by the application of other such definitions. Note that this expansion will terminate because the definition of Match follows the hierarchical structure of types in a C program and is consequently non-recursive. After each definition of Match has been expanded out, we proceed to conjoin the following formulas with φ:

1. $\mathsf{HasType}(\mathsf{Mem}[p], \mathsf{Type}[p])$ for each term $\mathsf{Mem}[p]$ in φ.
2. Instantiations of definitions A, B, and E on each term in $P(\varphi)$.
3. Instantiations of definitions C and D for each term p and q such that $p \in P(\varphi)$ and $\mathsf{Ptr}(q) \in Q(\varphi)$.

Let the resulting formula be ψ. Since the size of $P(\varphi)$ and $Q(\varphi)$ is bounded by $|\varphi|$, we generate at most $|\varphi|^2$ conjuncts, each of constant size. Therefore $|\psi| \in O(|\varphi|^2)$. If the solver concludes that ψ is unsatisfiable, then φ does not have a well-typed model because we only added facts related to the characterization of well-typed models to φ to get ψ. We now argue that if the solver concludes that ψ is satisfiable, then φ has a well-typed model. The following property of ψ is crucial for the correctness of this claim:

LEMMA 1. *If either* $\mathsf{Match}(p, q)$ *or* $\mathsf{HasType}(p, q)$ *is a term in* ψ, *then* $p \in P(\varphi)$.

[2] A theory is stably infinite if any satisfiable formula in the theory has a countably infinite model.

4.2 Model Construction

A *satisfying assignment* of ψ is a map W from terms in ψ to a value of the appropriate sort—bool, int, or type—such that evaluating ψ according to W returns $true$. A satisfying assignment W of ψ is *minimal* if for all terms q of sort type in ψ, either $W(q) \in TypeConst$ or $W(q) = W(\mathsf{Ptr}(q'))$ for some term $\mathsf{Ptr}(q')$ in ψ. We assume that if the SMT solver returns satisfiable, it provides a minimal satisfying assignment W for ψ. This assumption essentially requires the solver to not create any fresh Ptr terms while creating a model for ψ, which is reasonable because typical SMT solvers only create fresh constants during model generation. We will extend W to a well-typed model Γ satisfying φ.

Consider the set of integers $a \in \mathsf{int}$ such that W does not provide an assignment to $\mathsf{Type}[a]$. There exists a one-to-one map from this set into $UTypeConst$ because $UTypeConst$ is countably infinite. We use this map to complete the assignment of Type in Γ.

The interpretation of Match depends only on the interpretation of Type. For all integers a and for all type values $t \in ITypeConst \cup PtrTypeVals$, we evaluate $\mathsf{Match}(a, t)$, starting from $t \in PtrTypeVals$ and then for the $t \in ITypeConst$ in a bottom up fashion. To complete the assignment for Match, for all integers a and for all type values $t \in UTypeConst$, we assign $true$ to $\mathsf{Match}(a, t)$ everywhere it is not defined by W.

The definition of HasType depends only on the definition of Match. For all integers v and for all type values $t \in \{\mathsf{Int}\} \cup PtrTypeVals$, we evaluate $\mathsf{HasType}(v, t)$ bottom up, as with the evaluation of Match. To complete the assignment for HasType, for all integers v and for all type values $t \in TypeConst \setminus \{\mathsf{Int}\}$, we assign $true$ to $\mathsf{HasType}(a, t)$ everywhere it is not defined by W.

Our method of extending HasType yields the following lemma:

LEMMA 2. *For each* $t \in \mathsf{type}$, *there exists* $v \in \mathsf{int}$ *such that* $\mathsf{HasType}(v, t)$ *is assigned* $true$.

This property will help us complete the assignment for Mem. If Mem is not defined for some $a \in \mathsf{int}$, extend the assignment of Mem at a to some integer v such that $\mathsf{HasType}(v, \mathsf{Type}[a])$ is assigned $true$.

LEMMA 3. Γ *is a model satisfying* ψ *and hence satisfies* φ.

PROOF. To prove this lemma, it suffices to show that the assignments to Match and HasType in Γ are consistent with the assignments in W. The assignments to Type and Mem in Γ simply extend W by our definition of Γ. Here we present the proof for Match only, omitting the proof for HasType, which is similar.

We prove by contradiction that the assignment to $\mathsf{Match}(a, t)$ obtained under Γ is consistent with W. If not, then there exists a term $\mathsf{Match}(p, q)$ in ψ such that $W(p) = a$, $W(q) = t$, and $W(\mathsf{Match}(p, q))$ is inconsistent with the evaluation of $\Gamma(\mathsf{Match}(a, t))$. Since W is a minimal satisfying assignment, either $W(q) \in TypeConst$ or $W(q) = W(\mathsf{Ptr}(q'))$ for some term $\mathsf{Ptr}(q')$ in ψ. Since Γ extends W for any $t \in UTypeConst$ (by definition), we can strengthen the first case to $W(q) \in ITypeConst$. In either case, we get $q \in Q(\varphi)$. Since $\mathsf{Match}(p, q)$ is present in ψ, we also get $p \in P(\varphi)$. Therefore ψ contains an instantiation for the definition of $\mathsf{Match}(p, q)$. Because $\mathsf{Match}(p, q)$ is defined in ψ and W satisfies ψ, $\Gamma(\mathsf{Match}(a, t))$ must be consistent with $W(\mathsf{Match}(p, q))$, which is a contradiction; thus Γ satisfies ψ. Since φ is a conjunct in ψ, Γ satisfies φ too. □

Γ is also well-typed, which yields our main theorem:

THEOREM 1. Γ *is a well-typed model satisfying* φ.

The complexity of checking the satisfiability of ψ is NP-complete [27]. Since the translation from φ to ψ results in at most a quadratic blowup, the complexity of checking whether φ has a well-typed model is also NP-complete.

5. Extensions

In this section, we discuss a number of extensions to our translation that address additional features of the C language or additional requirements for verifying type safety in existing C code. Except where noted, these extensions were implemented and used in the case studies described in Section 6.

5.1 Unions

C's union types allow fields of several unrelated types to be stored at the same location in memory, with only one such field in use at a given time. Unfortunately, C does not provide any mechanism for keeping track of which field is currently in use, which means that the programmer could easily violate type safety by storing a value in one field and retrieving it from another field of a different type.

The use of unions varies widely from program to program. In some cases, each instance of a given union type uses only one field for the entire lifetime of the union; that is, the dynamic type of the union is fixed at allocation time. In other cases, a given instance of a union type uses many fields over its lifetime, and the dynamic type of that union cannot be fixed at allocation time.

Because the use of unions varies so widely, our approach is to leave unions completely undefined during translation. That is, our default translation says nothing about the meaning of HasType or the value of Type for a union type. If the programmer wishes to use unions safely, they must introduce additional assertions that state the appropriate invariants explicitly.

For example, consider the following C code:

```
union foo { int n; int *p; }
int getnum(foo *f, tag t) {
    return (t == 1) ? f->n : *f->p;
}
```

In this example, we have a union containing two types, `int` and `int*`, which means that the `foo*` argument is either an `int*` or an `int**`. Our default translation does not indicate which field is selected, so the user must specify a precondition on this function, such as:

```
pre((t == 1) ==> hastype(f, int*)) &&
    (t != 1) ==> hastype(f, int**)))
```

Here, we have extended C's syntax with an implication operator (`==>`) and a predicate `hastype` that will be translated into HasType in the input to the theorem prover. This precondition provides enough information to verify that the body of `getnum` is well-typed.

5.2 Function Pointers

The translation described in Section 3 only allows calls to known functions; however, most C programs use function pointers to invoke functions indirectly. Many property checking tools model function pointers by associating each function in the program with a distinct integer value, and then they model function pointer invocation as a case split on the integer representing the function pointer or as nondeterministic choice. However, when checking large C programs, it is often difficult, if not impossible, to know at compile time all functions that might be invoked at a given call site. Instead, we address this problem by adding a function type to our language.

We extend our input language with a function type and an indirect function call:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \tau_1 e_1 \rightarrow \tau_2 e_2 \\ c & ::= & \ldots \mid x := y(e) \end{array}$$

The function type $\tau_1 e_1 \rightarrow \tau_2 e_2$ represents a function from type τ_1 to type τ_2 that has precondition e_1 and postcondition e_2. Naturally, the precondition e_1 can refer to the argument x, and the postcondition e_2 can refer to the argument x and the return value r. Note that by allowing the programmer to refer to expressions in function types, we have introduced a form of dependent type. In the indirect call, we invoke a function stored in the variable y.

We extend BPL with a new data type constructor:

$$\mathsf{Func} : (\mathsf{type} \times \mathsf{int} \times \mathsf{type} \times \mathsf{int}) \rightarrow \mathsf{type}$$

We also extend the translation as follows:

$$\begin{array}{rcl} T(\tau_1 e_1 \rightarrow \tau_2 e_2) & = & \mathsf{Func}(T(\tau_1), \phi(E(e_1)), T(\tau_2), \phi(E(e_2))) \\ C(\Gamma, x := y(e)) & = & x := \mathsf{call}\ stub_\tau(E(e)) \\ & & \text{where } y \text{ has type } \tau \end{array}$$

The first part of this translation maps an annotated C function type to its BPL representation. The ϕ function is a one-to-one function from BPL expressions to integers, which is created by assigning a unique integer to every expression in the program text; this function allows us to encode the precondition and postcondition of a BPL function as integer arguments to Func.

The second part of the translation implements a call to y by calling the stub corresponding to y. If the type of y is $\tau = \tau_1 e_{pre} \rightarrow \tau_2 e_{post}$, then $stub_\tau$ is declared as follows:

$$\begin{array}{l} \mathsf{pre}\ E(e_{pre}) \wedge \mathsf{HasType}(x, \tau_1) \\ \mathsf{post}\ E(e_{post}) \wedge \mathsf{HasType}(r, \tau_2) \\ \mathsf{fun}\ stub_\tau(x : \mathsf{int}) : \mathsf{int} \end{array}$$

This stub summarizes the entire class of functions represented by the function type $\tau_1 e_{pre} \rightarrow \tau_2 e_{post}$. Thus, by calling this stub, we will check the preconditions given the argument e_2, and we will assume the postcondition on the caller's return variable x. Note that we do not need to perform any checking on $stub_\tau$ itself; it exists solely to represent function pointer invocations.

A subtle but important point is that the translation of function pointer invocations depends upon the types assigned by the original (unsound) C type system. However, because we enforce the declared type of y in our translation, we can use it to translate this function call.

The final piece of the translation is the HasType and Match axioms for the Func constructor. In order to define HasType, we associate a unique integer with every function in the program. For a given function type Func(...), we define HasType as the set of integers corresponding to all functions of that type. This set necessarily includes all such functions that are visible in the current compilation unit; however, it is not limited to those functions, since we may be calling a function of that type in a different compilation unit. For Match, we provide a definition that corresponds directly to the definitions for other word-sized types, since the function type is itself a word-sized type.

So, by associating preconditions and postconditions with C function types, and by using these preconditions and postconditions in the translation of C function calls, we can correctly translate and type-check C programs that use function pointers, even without knowing all possible values for every function pointer.

5.3 Parametric Polymorphism

Many existing C programs can be more effectively type-checked if the programmer is allowed to indicate code that uses parametric polymorphism. Consider the following example program:

```
typedef void *arg_t;  // Type variable!
typedef void (*fn_t)(arg_t a);
void create_thread(fn_t f, arg_t a);

void thread1(int n) { ... }
void thread2(foo *p) { ... }

foo *p = ...;
create_thread(thread1, 42);  // arg_t = int
create_thread(thread2, p);  // arg_t = foo*
```

In this example, we declare a function called `create_thread` that takes two arguments: a pointer to a function that should be executed on the new thread, and an argument to pass to that function. We then define two additional functions, `thread1` and `thread2`, which represent the main functions for two different threads. Finally, we invoke `create_thread` on each of these functions with different arguments; one takes an `int` and the other takes a `foo*`.

Although the type of `arg_t` is given as `void*` in this example, this type is actually being treated as a type variable. That is, for a particular call to `create_thread`, the programmer can consider `arg_t` to be any word-sized type, as long as the thread function and its argument have consistent types. This code is an example of how C programmers frequently use concepts from higher-level type systems even in lower-level code.

In our translation, we can provide polymorphism by explicitly passing the type for any type variables involved in the function call. For example, the above code would be translated into:

```
pre $\mathsf{HasType}(f, \mathsf{Func}(t, \ldots))$
pre $\mathsf{HasType}(a, t)$
fun $create\_thread(t : \mathsf{type}, f : \mathsf{int}, a : \mathsf{int}) = \ldots$

assume $\mathsf{HasType}(thread1, \mathsf{Func}(\mathsf{Int}, \ldots))$
assume $\mathsf{HasType}(thread2, \mathsf{Func}(\mathsf{Ptr}(\mathsf{Foo}), \ldots))$

assume $\mathsf{HasType}(p, \mathsf{Ptr}(\mathsf{Foo}))$
call $create\_thread(\mathsf{Int}, thread1, 42)$
call $create\_thread(\mathsf{Ptr}(\mathsf{Foo}), thread2, p)$
```

Note that both calls to $create\_thread$ satisfy the two preconditions on the types. In the first call, we pass a function $thread1$ that has a type whose first argument is Int, and we pass 42, which can be determined to have type Int according to the HasType axioms. The second call satisfies the preconditions for a similar reason.

In practice, our translator allows the programmer to identify types that should be treated as type variables. When these types appear in the arguments or return types of a function being invoked, we compare the formal types to the actual types to determine an appropriate substitution, and then we pass the substituted types as arguments to the translated function. Similarly, when translating a function with arguments of polymorphic type, we add a suitable number of formal type parameters to the translated function.

As with function pointers, we have found this feature to be quite useful in establishing type safety for existing C code due to examples like the one above. This example is particularly noteworthy because our type invariant allows us to state an important fact about the arguments to `create_thread` (i.e., that `f` will only be called with `a` as an argument) that would be difficult to state succinctly in a more traditional property checking tool.

Finally, note that our decision procedure can be extended to handle polymorphic types by treating the type variable t as a new, opaque type constant.

5.4 User-Defined Types and Dependent Types

In addition to the above extensions, we also allow the programmer to introduce new type constants with user-provided HasType definitions. For example, a programmer could use this feature to define a non-null type. Although this invariant could also be expressed by writing preconditions and postconditions on functions, it is often convenient to be able to add such global invariants to the type safety invariant, which is implicitly enforced at each program point.

When providing user-defined types, it is often convenient to have HasType depend upon Mem in addition to Type. Types that are defined in this manner can be considered a form of dependent type, since their meaning depends upon values stored in the heap. For example, consider the following structure:

```
struct string { char *buf; int len; }
```

This structure represents a string, where `len` is the number of characters appearing in `char`. However, our default type definition does not express this invariant:

$$\mathsf{HasType}(v, \mathsf{Ptr}(\mathsf{String})) \triangleq v = 0 \lor (v > 0 \land \mathsf{Match}(v, \mathsf{String}))$$

However, if HasType can depend on Mem, we can write a much stronger definition:

$$\begin{aligned}&\mathsf{HasType}(v, \mathsf{Ptr}(\mathsf{String}), \mathsf{Mem}) \triangleq\\ &\quad v = 0 \lor (v > 0 \land \mathsf{Match}(v, \mathsf{String}) \land\\ &\qquad\qquad \forall i : \mathsf{int}. 0 \leq i < \mathsf{Mem}[v + 1] \implies\\ &\qquad\qquad\quad \mathsf{HasType}(\mathsf{Mem}[v] + i, \mathsf{Ptr}(\mathsf{Char}), \mathsf{Mem}))\end{aligned}$$

This new definition is more powerful than the previous one, but it also places an additional burden on the theorem prover, and we must rely on the programmer to create type definitions that preserve the completeness guarantees discussed in Section 4.

5.5 Allocation and Sub-Word Access

Currently, our translation models memory allocation by scrambling the target variable and assuming the appropriate type. We can improve precision by introducing two additional maps: Alloc, which keeps track of whether each word of memory has been allocated or deallocated, and Base, which maps each allocated word to the base address for that allocation. These maps provide additional precision for our property checker, and they also allow us to express and check temporal type and memory safety properties, such as the lack of dangling pointers.

Another imprecision is our assumption that each word in memory is of size 1. To model a 32-bit machine, we can set the word size to 4 and allow Mem to map byte addresses to values. We maintain an additional map, Span, to keep track of how many byte addresses a given value spans. For example, when writing word-sized values to address a, we will assert that $\mathsf{Span}(a) = 4$ and that $\mathsf{Span}(a + 1) = \mathsf{Span}(a + 2) = \mathsf{Span}(a + 3) = 0$.

Our current prototype implements Alloc and Base, but it does not implement Span, and we do not check for dangling pointer errors. These features are explained in more detail in the appendix and will be explored further in future work.

6. Evaluation

Here we present several case studies that demonstrate the effectiveness of our technique on real code, including property examples and experiments with type checking in Windows device drivers.

We implemented the combined type and property checking tool described in this paper inside HAVOC [2], a property checker for C code that plugs into Microsoft's Visual C compiler. After HAVOC translates C code to BPL, we use Boogie [5] to generate a verification condition, which we check using the Z3 SMT solver [13]. HAVOC previously supported reasoning about linked lists [20] and arrays using SMT solvers.

6.1 Property Checking

To evaluate the usefulness of adding types to a property checker, we have applied our tool to a set of small to medium-sized C benchmarks in the HAVOC regression suite [20]. These examples range between 10 and 100 lines of code, and they include various low-level list algorithms (e.g., adding or removing elements from a doubly linked list, reversing or sorting a list) and various array sorting algorithms (e.g., insertion sort, bubble sort). The list routines use the `list` structure from Figure 1. For each of these examples, we proved partial correctness properties (e.g., bubble sort yields a sorted array, reversing a list preserves the list), in addition to the type safety assertions. The runtimes ranged from a few seconds on

	cancel	event	kbfiltr	vserial	headers	total
Lines of code	1186	1259	1174	1452		5071
Procedures checked	13	9	12	22		56
Time to check (sec)	57	61	49	52		219
Function pre & post	22	17	20	15	1	75
Loop invariants	1	1		3		4
Field sensitivity (§3.3)	2	3		3		8
Custom types (§5.4)	24	12				36
Type variables (§5.3)					3	3
Type changes	14	2	1	5	5	27
Code changes	4	5	7	2	17	35
Assumptions	9	15	8	3	1	36
Total changes	76	55	36	30	27	224

Table 1. Results from our Windows device driver experiments.

the smaller examples to around 8 minutes on the largest example. In the absence of types, earlier verification of these examples included ad-hoc annotations to obtain disambiguation.

We use one of the examples `list_appl` to illustrate the benefits of using types in the annotations. The example (about 100 lines) contains two circular doubly-linked lists hanging off a parent object; each node in the two lists has a pointer to the parent. The objects in the two lists have distinct C types and have different data structure invariants. The example performs various operations such as initialization, insertion/deletion from the lists, and updating the data values in the lists. The data structures in the example are fairly representative of low-level systems code.

The main challenge in this example is to preserve the global invariants of the lists despite updates to the heap. To do so, we must ensure that the set of addresses in the two lists are disjoint, and we must have field-based disambiguation in order to show that certain fields are not updated. Previously, stating these invariants required us to construct a set for the addresses of *each* of the fields in the two lists and specify pairwise disjointness of these sets. These specifications were very cumbersome and required a quadratic number of annotations in the number of fields.

In contrast, our field-sensitive type safety assertion ensures that the fields of two different types do not alias. To state that the two lists are disjoint, we simply state an invariant for each list describing the type of the object in which the list is embedded. The specifications are local to each list and hence grow linearly in the number of lists. The conciseness of specification is crucial for verifying larger systems programs where multiple lists hang off a parent object with a few hundred fields.

We are currently working on using type safety assertions to improve the soundness of property checking for real-world code. Among these, we are working towards justifying the field disambiguation that was *assumed* when HAVOC checked complex synchronization protocols in a 300 KLOC Windows component [3].

6.2 Device Drivers

We applied our tool to several Windows device drivers for the purpose of verifying type safety. These device drivers (`cancel`, `event`, `kbfiltr`, and `vserial`) are publicly-available sample drivers included with the Windows Driver Kit (WDK) 1.7 [22] that demonstrate several common idioms in Windows device drivers.

The process of annotating a driver is iterative, much like the traditional edit-compile-debug cycle. We ran our tool on the unmodified driver, and we added annotations, introduced new types, or otherwise modified the code in order to resolve the reported type errors. We also modified the WDK header files where appropriate, and we had HAVOC automatically add non-null assumptions for all pointers. Each conversion took approximately 1-2 hours.

The flexibility of the technique described in this paper was crucial for checking these drivers successfully. The most prominent example is the `LIST_ENTRY` structure, which is embedded in structures to form a linked list as demonstrated in Figure 1. This idiom appears commonly in Windows code, including two of the four drivers tested. We were able to annotate and check these linked lists without specifically customizing our tool for this case; this example demonstrates the ability of our technique to capture program-specific invariants that are important for enforcing type safety.

Another common example that demonstrates the usefulness of our technique is the dispatch mechanism that the kernel uses to invoke drivers. A simplified version of the code is as follows:

```
void MyRead(Driver *driver) {
    MyContext *ctx = (MyContext*) driver->ctx;
    ...
}
void MyWrite(Driver *driver) {
    MyContext *ctx = (MyContext*) driver->ctx;
    ...
}
void MyInit(Driver *driver) {
    MyContext *ctx = ...;
    driver->ctx = (void*) ctx;
    driver->read = MyRead;
    driver->write = MyWrite;
}
```

In this example, our driver defines two dispatch routines, `MyRead` and `MyWrite`, and an initialization function, `MyInit`. Each routine is called with a kernel object of type `Driver*` representing the driver. This kernel object contains a `ctx` field of type `void*` that is used by each driver to store the driver's private data. In the `MyRead` and `MyWrite` functions, the driver casts this pointer to a `MyContext*` in order to access its private data.

We would like to prove this cast safe, but we cannot simply add a precondition to `MyRead` and `MyWrite` saying that the type of the `ctx` field is `MyContext*`, since the caller (i.e., the kernel) does not know about the internals of each driver and could not prove this precondition. In fact, the real precondition for `MyRead` is `driver->read == self`, where `self` is a special keyword representing the current function (i.e., `MyRead`). In other words, the invariant is that the kernel will only call the `driver->read` function with `driver` itself as an argument, which is a common invariant in low-level type systems for object-oriented code. Since we can prove that `driver->read == MyRead`, we can use the `read` field as a tag indicating the run-time type of `ctx`; that is, we add a global invariant that says that a `driver` whose `read` field is `MyRead` must also have a `ctx` of type `MyContext*`.

A final example where we made use of this technique is in the `vserial` driver. Several complicated routines that read and write buffers of data required preconditions and loop invariants in order to prove that all array references were in bounds. Because we can express the necessary invariants directly using the property checker, we did not require any customized type annotations, as other type checking tools would require.

The results of our driver experiments are shown in Table 1. The columns in this table show the results for each driver, the results for the common header files, and the totals.

The first two rows show the number of lines of code in each example as well as the time it took for each example to be checked. Each driver is slightly over 1,000 lines of code and takes about 1 minute to be checked using our tool. Because our tool is completely modular, we expect this figure to scale in proportion to the number of lines of code; however, more complex annotations may result in more significant slowdowns. At these speeds, it is feasible to run our type checker occasionally during development, though not at every compilation; however, we believe that there is still a significant amount of room for optimization.

The next section of Table 1 shows the number and kind of changes to the program, roughly amounting to the number of lines changed or added. Changes are broken down according to the features used, with a reference to a section of the paper where relevant. "Type changes" refers to any refinement of the program's existing types (e.g., changing `void*` to something more specific), and "code changes" refers to any changes to the code itself.

The first six lines represent "good" annotations that add more precision to the existing C code. The most frequently used annotations are the function annotations, which specify function types and contracts. The type extension feature is also used frequently in `cancel` and `event`, primarily for specifying private device data structures containing linked lists. Overall, there are 153 such annotations for 5,000 lines of code, or 3.0%.

The last two lines represent "bad" or undesirable annotations, including changes to the code and unchecked assumptions. Most code changes were the result of gaps in our C analysis infrastructure, and assumptions were typically made when types were determined by obscure rules that were difficult to formalize succinctly without deep knowledge about driver invariants. An example of the latter case is the `IRP` data structure's `Tail.Overlay.ListEntry` field, which is part of a union that has no obvious tag indicating its current type. These assumptions, though unchecked, serve a valuable purpose in flagging code for future review. We also believe that further annotation effort could reduce the number of unchecked assumptions significantly. These "bad" changes accounted for 71 annotations, or 1.4% of the lines of code, and do not include the automatically-generated non-null pointer assumptions.

Overall, these results demonstrate that our translation is an effective tool for expressing and checking important type invariants in C code without customizing the tool to a particular code base. There is still additional room for improvement in terms of inference and expressiveness so that we minimize the need for explicit "good" annotations and eliminate the "bad" ones.

7. Related Work

7.1 Proof Carrying Code

Proof Carrying Code [24] combined type checking for Java-like languages with an SMT solver. For example, Touchstone [26] compiles Java into native x86 code along with a proof that the resulting code is type and memory safe, which is generated by an SMT solver with specially-designed decision procedures. However, the type system formalized in PCC was based on a set of axiomatized type rules, whereas our lower-level approach explicitly defines the set of values that correspond to each type. Also, we provide a much lower-level model of the program's semantics (e.g., using the Mem map instead of more abstract objects), and we expose a significant portion of this model to the programmer by allowing the programmer to write preconditions, postconditions, and custom types. Finally, we show these features can improve the precision of property checking on low-level code.

Further work on PCC has attempted to minimize the trusted computing base for these verifiers to the model of the underlying hardware [9, 15]. These tools are typically quite powerful, and they allow the user to apply many different proof techniques in the context of a single program. Our work focuses on type checking legacy C code using an SMT solver, which allows us to build a more scalable and more automated system.

7.2 Type Checking for C

CCured [25] provides strong type checking for C by inferring refined pointer "kinds" and instrumenting the program appropriately, and Deputy [12] uses dependent types in place of CCured's pointer annotations. Both systems use compile-time and run-time checks to enforce type safety. Unfortunately, CCured and Deputy provide only a fixed set of types, so it is difficult to check programs that make use of C idioms not covered by these types. We provide a much more flexible annotation system that is based on our property checking tools; as a result, it is often easier for users to "explain" to the type checker why an existing program is safe. We have found that stating preconditions, postconditions, and type invariants can often be simpler and more flexible than using dependent types.

The type systems for CCured and Deputy provided the inspiration for the low-level types described in this paper. By implementing these types in the context of an SMT solver, we gain additional expressiveness that was not available in either of these tools, and we allow the SMT solver to leverage the existing C types.

Cyclone [19] provides a sound C-like type system that can handle a wide range of existing C idioms. Our approach requires less porting effort and has more expressive types, but we are also less scalable. Semantic type qualifiers [11] allow C types to be refined using type rules whose soundness is checked at compile time. As with our system, this approach allows the user to extend C types to express common program invariants. Our approach gains additional expressiveness at the cost of some scalability.

7.3 Dependent Types

Dependent ML [33] and Xanadu [32] provide dependent types for functional and imperative programs, respectively. These types provide a clean mechanism for refining ML types to provide additional information about properties such as array bounds. However, these type systems do not reason about updates to mutable state. Our approach overcomes the problem of mutable state by modeling it directly in our translation; our SMT solver handles these updates cleanly through the use of standard select-update reasoning. Also, we have found that specifying preconditions and postconditions of functions can be a useful alternative to specifying these properties using dependent types.

Liquid types [29] use property checking tools to infer dependent types for DML programs. This approach is complementary to ours; the same techniques that can be used to infer refinements for ML types could conceivably be used to infer important type-based invariants in our system as well.

7.4 Property Checking

ESC/Java [17] and Spec# [6] add checked contracts to Java and C# in the same style as our work. However, such contracts are more difficult to write in C due to its lower-level memory model. Enforcing type safety in C bridges this gap, enabling higher-level property checking even in the context of a low-level language.

Property checking tools for C fall under two categories. Sound verifiers for C such as Compcert [21] and VCC [31] take a low-level view of C's memory model ignoring types, and use higher-order theorem provers (e.g., Coq [1]) in conjunction with SMT solvers to discharge the verification conditions. Although these tools offer more expressiveness compared to our work for the property language, they place significant annotation burden on the users to express disambiguation of the heap and also guide the theorem prover to construct the proofs. Calcagno et al. [8] check memory-safety of low-level code manipulating linked-lists using separation logic [28], but have limited expressivity to check the properties we check in this work.

On the other hand, several property checking tools for C assume type safety to perform scalable analysis of low-level code, thereby introducing unsoundness in the analysis. SLAM [4] and BLAST [18] use predicate abstraction to check control-oriented properties (e.g., lock usage) on device drivers. Caduceus [16] is a modular verifier for C that assumes field disambiguation to partition the heap. Yang et al. [34] prove memory-safety of programs manipulating linked lists, but require unsound assumptions for arrays and pointer arithmetic. Our work shows that these tools require a field-sensitive type safety invariant to justify the field disambiguation used in these tools. Most of these works are aimed at inferring annotations and can be seen complementary to our work. Using these techniques in the context of our low-level memory model would reduce the annotation burden in our tool.

8. Conclusion

This paper has presented a technique for checking types and properties in tandem on low-level code. Using a property checker to implement a type checker gives us the power to express and check program-specific type invariants. In addition, proving type safety for low-level code allows us to provide disambiguation between heap locations that is required by the property checker. Our results suggest that this approach is an effective way to improve the power and expressiveness of verification tools for low-level code.

References

[1] The Coq proof assistant. `http://coq.inria.fr/`.

[2] The HAVOC property checker. `http://research.microsoft.com/projects/havoc/`.

[3] T. Ball, B. Hackett, S. K. Lahiri, and S. Qadeer. Annotation-based property checking for systems software. Technical Report MSR-TR-2008-82, Microsoft Research, 2008.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, 2001.

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, 2005.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, 2004.

[7] M. Barnett and R. Leino. Weakest-precondition of unstructured programs. In *Program Analysis for Software Tools and Engineering (PASTE)*, 2005.

[8] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis Symposium (SAS)*, 2006.

[9] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The open verifier framework for foundational verifiers. In *Types in Language Design and Implementation (TLDI)*, 2005.

[10] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[11] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation (PLDI)*, 2005.

[12] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. Necula. Dependent types for low-level programming. In *European Symposium on Programmig (ESOP)*, 2007.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[14] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communcations of the ACM*, 18, 1975.

[15] X. Feng, Z. Ni, Z. Shao, and Y. Guo. An open framework for foundational proof-carrying code. In *Types in Language Design and Implementation (TLDI)*, 2007.

[16] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Computer Aided Verification (CAV)*, 2007.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2002.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, 2002.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[20] S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL)*, 2008.

[21] X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Principles of Programming Languages (POPL)*, 2006.

[22] Microsoft. Windows driver kit. `http://www.microsoft.com/whdc/devtools/wdk/default.mspx`.

[23] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *Transactions on Programming Languages and Systems (TOPLAS)*, 21:3, 1999.

[24] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages (POPL)*, 1997.

[25] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), May 2005.

[26] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Language Design and Implementation (PLDI)*, 1998.

[27] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *Transactions on Programming Languages and Systems (TOPLAS)*, 1(2), 1979.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.

[29] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, 2008.

[30] Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.

[31] W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler. In *C/C++ Verification Workshop*, 2007.

[32] H. Xi. Imperative programming with dependent types. In *Logic in Computer Science (LICS)*, 2000.

[33] H. Xi and F. Pfenning. Dependent types in practical programming. In *Principles of Programming Languages (POPL)*, 1999.

[34] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification (CAV)*, 2008.

A. Additional Extensions

This section provides further details on some of the extensions proposed in Section 5. Only some portions of these extensions have been implemented and tested, and they were not used in the experiments described in Section 6.

A.1 Deallocation

The translation described in Section 3 mentions memory allocation but does not mention memory deallocation. In this section, we extend the translation to allow us to verify correct deallocation. Specifically, we would like to verify that the program never accesses deallocated memory and never frees memory twice. These properties are sometimes referred to as *temporal memory safety*. (Bounds checks, which largely fall under *spatial memory safety*, are covered by the type safety assertions discussed earlier in this paper.)

To add the appropriate assertions, we need two new maps:

$$\begin{array}{rcl} \mathsf{Alloc} & : & \mathsf{int} \rightarrow \{\mathsf{unallocated}, \mathsf{allocated}, \mathsf{deallocated}\} \\ \mathsf{Base} & : & \mathsf{int} \rightarrow \mathsf{int} \end{array}$$

The first map, Alloc, keeps track of the allocation state for each address in memory, which can be unallocated (i.e., never allocated), allocated (i.e., in use), and deallocated. The second map, Base, maps integer addresses to the first word of the allocated object that contains it. For example, if we allocate a pointer p to an object of size n, then $\mathsf{Base}(p)$ through $\mathsf{Base}(p + n - 1)$ will be equal to p. The Alloc map is mutable, but the Base map is not.

Our translation refers to these maps as follows:

- When we allocate an object of size n at address p, our translation inserts an assumption stating that Alloc is $\mathsf{unallocated}$ at addresses p through $p + n - 1$, and then it sets these values to $\mathsf{allocated}$. We also insert an assumption stating that the Base map is set to p for all addresses in this range.
- When we deallocate an object p, we verify that $\mathsf{Base}(p) = p$, which says that p was allocated at some point in the past. We also check that $\mathsf{Alloc}(p) = \mathsf{allocated}$, which means that it has not already been freed, and we set $\mathsf{Alloc}(p)$ to $\mathsf{deallocated}$.
- When we dereference a memory address a, we assert that $\mathsf{Alloc}[\mathsf{Base}(a)]$ is $\mathsf{allocated}$.

This formalization allows us to check that all memory locations accessed by the program were properly allocated and have not yet been deallocated. Note that deallocation only changes the first memory address to $\mathsf{deallocated}$, but because we check the status of the *base* address on each memory reference, this one change is sufficient to mark the entire object as freed.

Note that checking these assertions is considerably more challenging than verifying the type safety assertions discussed in this paper. Although we can verify temporal memory safety for small examples, our large experiments currently do not use this portion of the translation.

Finally, there is still one source of imprecision in this approach: we assume that deallocated memory is never subsequently reallocated, which is not the case in real C programs. To relax this assumption, we can change the translation of deallocation to set the Alloc map back to $\mathsf{unallocated}$. However, we must also make Base and Type mutable, since their values can be changed when memory is reallocated. These changes can make verification of the program even more challenging.

A.2 Sub-Word Access

Our translation currently assumes that all pointers and integers are of size 1. However, modern processors can access integer data of multiple sizes, so we must handle memory accesses that affect multiple addresses in memory. For example, if we write a 32-bit integer to address p, we actually change the values at byte addresses $p, p + 1, p + 2$, and $p + 3$.

For the sake of efficiency, we wish to avoid modeling writes at such a low level wherever possible. For example, if the program never accesses $p + 1$, $p + 2$, and $p + 3$ directly, we can store the integer value in p only. However, if any of these addresses are accessed by the program, we would like our translation to model these sub-word reads and writes faithfully.

To further illustrate this challenge, consider this example:

```
int *p = ...;
*p = 1337;

char *q = ((char *) p) + 2;
*q = 42;

assert(*p != 1337);
```

Even though `p` and `q` have different values, the write to `q` alters the value of `*p`. If we model these pointer writes as updates to the Mem map at distinct locations (e.g., $\mathsf{Mem}[p] := 1337; q := p + 2; \mathsf{Mem}[q] := 42$), then we will not correctly model the behavior of the real C program.

To address this problem, we introduce an additional map:

$$\mathsf{Span} : \mathsf{int} \rightarrow \mathsf{int}$$

For a given address a, $\mathsf{Span}(a)$ indicates the number of addresses spanned by the value stored at a. For example, if we write a four-byte value to address a, we will assert that $\mathsf{Span}(a) = 4$ and that $\mathsf{Span}(a + 1) = \mathsf{Span}(a + 2) = \mathsf{Span}(a + 3) = 0$. A span of zero indicates that that address is invalid because it is spanned by a lower address.

In the common case, every address will be given a span that is determined by the type allocated at that address. For example, `int` types will be assigned span 4, and `short` types will be assigned span 2. However, for addresses that are accessed at multiple sizes and offsets (as in the example above), we can assign each address in the relevant range a span of 1. Writes to these addresses are then modeled using separate updates to each address; for example, writing a `short` to an address a that is modeled with byte-level access would result in an update to addresses a and $a + 1$.

In essence, we use Span to indicate whether a location in memory should be treated with byte-level precision or with a higher-level representation, and we assert as part of the translation that each memory location accessed has been modeled at an appropriate level of precision.

Due to time constraints, we have left the implementation and evaluation of this scheme to future work.