CSolve: Verifying C With Liquid Types[⋆]

Patrick Rondon, Alexander Bakst, Ming Kawaguchi, and Ranjit Jhala

University of California, San Diego
{prondon, abakst, mwookawa, jhala}@cs.ucsd.edu

Abstract. We present CSolve, an automated verifier for C programs based on Liquid Type inference. We show how CSolve verifies memory safety through an example and describe its architecture and interface.

1 Introduction

Verifying low-level programs is challenging due to the presence of mutable state, pointer arithmetic, and unbounded heap-allocated data structures. In recent years, dependent refinement types have emerged as a promising approach to verification in general [1,7] and low-level software in particular [2]. In a refinement type system, each program variable and expression is given a type of the form $\{\nu : \tau \mid p\}$ where τ is a conventional type such as `int` or `bool` and p is a logical predicate over the program variables describing the values ν which belong to the type, called the *refinement predicate*. To keep type checking decidable, refinement predicates are typically drawn from a quantifier-free logic; by combining SMT-based logical reasoning and type theory-based data structure reasoning, refinement type systems are easily able to synthesize and reason using facts about the contents of unbounded data structures.

While powerful, refinement types have typically been associated with a high annotation burden on the programmer. We present CSolve, an automated verifier for C programs based on the Low-Level Liquid Types [6] technique for refinement type inference. We show how CSolve accommodates refinement type checking with little necessary annotation.

2 Architecture, Use, and Availability

Type inference in CSolve is split into four phases. In the first phase, the input C program is read by CIL [4], which generates an AST. This AST is then simplified in various ways, the most significant of which is that the code is transformed to SSA so that local variables are never mutated. The second phase generates physical types for each declared function and global variable and checks that the program code respects these types. The third phase walks the CIL AST and assigns each expression and variable in the program a refinement type with

[⋆] This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

a distinct *refinement variable* representing its as-yet-unknown refinement predicate. The same phase generates subtyping constraints over these refinement types such that solving for the refinement variables within the constraints yields a valid typing for the program. The fourth phase attempts to solve the subtyping constraints using a fixed-point procedure based on predicate abstraction, using the Z3 SMT solver [3] to discharge the logical validity queries that arise in constraint solving.

Input CSolve takes as input a C source code file and a file specifying the set of logical predicates to use in refinement inference. Predicates are also read from a standard library of predicates that have proven to be useful on a large variety of programs, further easing the programmer's annotation burden.

Output If the program is type-safe, CSolve outputs "Safe". Otherwise, the program may be type-unsafe, according to either the physical type system or the refinement type system. In either case, for each error, CSolve prints the name of the file and line number where the error occurs, as well as a description of the error. In the case where the error is in refinement type inference, CSolve prints the subtyping constraints which cannot be solved. Whether the program typechecks or not, CSolve produces a mapping of program identifiers to their inferred types which can be viewed using the tag browsing facilities provided by common editors, *e.g.* Vim and Emacs.

Compatibility With C Infrastructure Thanks to the infrastructure provided by CIL, CSolve is able to work as a drop-in replacement for GCC. Hence, to check a multi-file program one need only construct or slightly modify a makefile which builds the program from source.

Availability The CSolve source code and an online demo are available at `http://goto.ucsd.edu/csolve`.

3 Example

In the following, we demonstrate the use of CSolve through a series of functions which manipulate text containing comma-separated values. We begin by showing how CSolve typechecks library functions against their stated specifications. We then show how CSolve infers function types to check an entire program.

We begin with a string library function, `strntolower`, shown in Figure 1, which lowercases each letter in a string. Its type signature is a C type augmented with annotations that are used by the CSolve typechecker. The `CHECK_TYPE` annotation tells CSolve to check `strntolower` against its type signature, rather than attempting to infer its type from its uses.

Type checking `strntolower` proceeds in two phases. First, because C is untyped, a *physical type checking* pass recovers type information describing heap layout and the targets of pointer-valued expressions. Next, a *refinement type checking* pass computes refinement types for all local variables and expressions.

Physical Type Checking We begin by describing how the type annotations in the example are used by CSolve to infer enriched physical types. The `char` $*$ `STRINGPTR` portion of the type ascribed to `strntolower`'s parameter `s`

```
void
strntolower (char * STRINGPTR SIZE_GE(n) s,
             int NONNEG n)
CHECK_TYPE {
  for (; n-- && *s != '\0'; s++)
    *s = tolower (*s);
}

extern char * NNSTRINGPTR LOC(L)
       NNREF(&& [s <= V; V < s + n; InB(s)])
strnchr (char * STRINGPTR LOC(L) SIZE_GE(n) s,
         int NONNEG n,
         char c);

typedef struct _csval {
  int                 len;
  char * ARRAY LOC(L) str;
  struct _csval *     next;
} csval;

void lowercase_csvals (csval *v) {
  while (v) {
    strntolower (v->str, v->len);
    v = v->next;
  }
}
```

```
csval INST(L, S) *
revstrncsvals (char * ARRAY LOC(S) s,
               int n)
{
  csval *last = NULL;
  while (n > 0) {
    csval *v =
      (csval *) malloc (sizeof (*v));
    v->next  = last;
    v->str   = s;
    char *c  = strnchr (s, n, ',');

    if (!c) c = s + n - 1;

    *c      = '\0';
    v->len  = c - s;
    n      -= v->len + 1;
    s       = c + 1;
    last    = v;
  }

  return last;
}

...
1. csval *vs =
     revstrncsvals (line, len);
2. lowercase_csvals (vs);
...
```

Fig. 1. Running example: splitting a string into comma-separated values

indicates to CSolve that `s` is a reference to a location l which contains an array of characters, *i.e.,* a string (and not a single `char`). Concretely, the type of `s` is $\texttt{ref}(l, \{0 + 1*\})$, which indicates that `s` points into a region of memory named by l. The notation $\{0 + 1*\}$, which is equivalent to $\{0, 1, 2, \ldots\}$, indicates that `s` may point to any nonnegative offset from the start of the region l, *i.e.,* anywhere in the array. Based on `s`'s type, CSolve describes the heap as

$$l \mapsto \{0 + 1*\} : \texttt{int}(1, \{0 \pm 1*\}).$$

The above heap contains a single location, l, whose elements are offsets from l in the set $\{0 + 1*\}$, defined as above. Each array element has the type $\texttt{int}(1, \{0 \pm 1*\})$, which is the type of one-byte integers (`char`s) whose values are in the set $\{\ldots, -1, 0, 1, \ldots\}$ (*i.e.,* any `char`). Similarly, the physical type of `n` is $\texttt{int}(4, \{0 \pm 1*\})$.

CSolve then determines, through straightforward abstract interpretation in a domain of approximate integer values and pointer offsets [6], that the physical types of `s`, `n`, and the heap are preserved by the loop within the body of `strntolower`; we note only that the return type of `tolower` indicates that it returns an arbitrary `char`, as above. Thus, physical typechecking succeeds, and we proceed to refinement type checking.

Refinement Type Checking We next explain how CSolve typechecks the body of `strntolower`—in particular, to verify that `strntolower`'s type signature implies the safety of the array accesses in its body.

We begin by describing how the annotations ascribed to `strntolower` are translated to a refinement type by CSolve. The type of `s` uses the convenience macros `STRINGPTR` and `SIZE_GE`, defined as:

$$\texttt{STRINGPTR} \doteq \texttt{ARRAY REF(SAFE(}\nu\texttt{))}$$

$$\texttt{SIZE\_GE(n)} \doteq \texttt{REF(}BE(\nu) - \nu \geq \texttt{n)}.$$

In the above, `ARRAY` indicates that the refined type points to an array, used in physical type checking. The `REF` macro is used to attach a refinement predicate to a type. Refinement predicates can themselves be constructed using macros; `SAFE` is a macro defined as the predicate

$$\mathtt{SAFE}(p) \doteq 0 < p \wedge BS(p) \leq p \wedge p < BE(p).$$

In this definition, the functions $BS(p)$ and $BE(p)$ indicate the beginning and end of the memory region assigned to pointer p, respectively. Thus, the $\mathtt{SAFE}(p)$ predicate states that p is a non-NULL pointer that points within the memory region allocated to p, *i.e.,* p is within bounds. The predicate $\mathtt{SIZE\_GE}(\mathtt{n})$ states that the decorated pointer points to a region containing at least `n` bytes; note that this expresses a *dependency* between the type of `s` and the value of the parameter `n`. We decorate the type of `n` with `NONNEG`, which expands to $\mathtt{REF}(\nu \geq 0)$.

We now describe how CSolve uses the given types for `s` and `n` to verify the safety of `strntolower`. To do so, CSolve infers *liquid types* [5] for the variables `s` and `i` within the body of `strntolower`, as well as the contents of the heap. A liquid type is a refinement type whose refinement predicate is a conjunction of user-provided *logical qualifiers*. Logical qualifiers are logical predicates ranging over the program variables, the wildcard $\star$, which CSolve instantiates with the names of program variables, and the *value variable* ν, which stands for the value being described by the refinement type. Below, we assume the logical qualifiers

$$\mathbb{Q}_0 \doteq \{0 \leq \nu,\ \mathtt{SAFE}(\nu),\ \star \leq \nu,\ \nu + \star \leq BE(\nu),\ \nu \neq 0 \Rightarrow \mathtt{InB}(\nu, \star)\}$$
$$\text{where } \mathtt{InB}(p, q) \doteq BS(p) = BS(q) \wedge BE(p) = BE(q)$$

where $\mathtt{InB}(p, q)$ means p and q point into the same region of memory.

From the form of the loop and the given type for the parameter `n`, CSolve infers that, within the body of the loop, `n` has the liquid type

$$\mathtt{n} :: \{\nu : \mathtt{int}(4, \{0 \pm 1*\}) \mid 0 \leq \nu\}.$$

Based on the type given for the parameter `s` and the form of the loop, CSolve infers that, within the loop, `s` has the liquid type

$$\mathtt{s} :: \{\nu : \mathtt{ref}(l, \{0 + 1*\}) \mid \mathtt{s} \leq \nu \wedge \nu + \mathtt{n} \leq BE(\nu) \wedge \nu \neq 0 \Rightarrow \mathtt{InB}(\nu, \mathtt{s})\}.$$

The predicates $\mathtt{s} \leq \nu$, $\nu + \mathtt{n} \leq BE(\nu)$, and $\nu \neq 0 \Rightarrow \mathtt{InB}(\nu, \mathtt{s})$ are instantiations of qualifiers $\star \leq \nu$, $\nu + \star \leq BE(\nu)$, and $\nu \neq 0 \Rightarrow \mathtt{InB}(\nu, \star)$ from $\mathbb{Q}_0$, respectively, where the $\star$ has been instantiated with `s`, `n`, and `s`, respectively. By using an SMT solver to check implication, CSolve can then prove that `s` has the type

$$\mathtt{s} :: \{\nu : \mathtt{ref}(l, \{0 + 1*\}) \mid \mathtt{SAFE}(\mathtt{s})\}$$

and thus that the accesses to $*$`s` within `strntolower` are within bounds.

External Definitions If the user specifies a type for a function with the `extern` keyword, CSolve will use the provided type when checking the current source file, allowing the user to omit the body of the external function. This allows for modular type checking and, by abstracting away the details of other source files, it permits the user to work around cases where a function may be too complex for CSOLVE to typecheck.

In the sequel, we use the library function `strnchr`, which attempts to find a character `c` within the first `n` bytes of string `s`, returning a pointer to the character within `s` on success and NULL otherwise. Its type, declared in Figure 1, illustrates two new features of CSOLVE's type annotation language. First, macros that begin with `NN` are analogous to the versions without the `NN` prefix, but the refinement predicates they represent are guarded with the predicate $(\nu \neq 0)$. Such macros are used to indicate properties that are only true of a pointer when it is not NULL. Second, the annotation `LOC(L)` is used to provide may-aliasing information to CSOLVE. The annotation `LOC(L)` on both the input and output pointers of `strnchr` indicates that both point to locations in the same may-alias set of locations, named `L`. This annotation is necessary because CSOLVE assumes by default that all pointers passed into or out of a function refer to distinct heap locations. This assumption that pointers do not alias is *checked*: if the annotation were not given, CSOLVE would alert the user that locations that were assumed distinct may become aliased within the body of `strnchr`.

Whole-Program Type Inference The remainder of Figure 1 shows a fragment of a program which reads lines of comma-separated values from the user, splits each line into individual values (`revstrncsvals`), and then transforms each value to lowercase (`lowercase_csvals`). In the following, we describe how CSOLVE performs refinement type inference over the whole program to determine that all of its memory accesses are safe.

The remainder of the program manipulates linked lists of comma-separated values, described by the structure type `csval`. Note that the field `str` is annotated with the `ARRAY` attribute, as before, as well as a may-alias set, `L`. By declaring that the `str` field points to may-alias set `L`, we *parameterize* the `csval` structure by the location set `L` that its `str` field points into. The programmer can then instantiate the parameterized location set according to context to indicate potential aliasing between the `str` field and other pointers. For example, the annotation `INST(L, S)` in the type of `revstrncsvals` instantiates `csval`'s may-alias set parameter `L` in that type to the location set `S`, indicating that the input string `s` and the strings stored in the list of `csval`s reside in heap locations in the same may-alias set, `S`.

In the following, we assume the set of qualifiers is

$$\mathbb{Q} \doteq \mathbb{Q}_0 \cup \{\nu \neq 0 \Rightarrow \star \leq \nu,\ \nu \neq 0 \Rightarrow \nu < \star + \star,\ \nu \neq 0 \Rightarrow \nu = BS(\nu),\\ \nu \neq 0 \Rightarrow BE(\nu) - BS(\nu) = 12,\ \nu = \star + (\star - \star)\}.$$

At the line marked 1, we assume CSOLVE has inferred the types

$$\begin{aligned} \texttt{line} &:: \{\nu :\ \texttt{ref}(l, \{0 + 1*\})\ \mid\ \texttt{SAFE}(\nu) \wedge \texttt{SIZE\_GE}(\texttt{len})\} \\ \texttt{len} &:: \{\nu :\ \texttt{int}(4, \{0 \pm 1*\})\ \mid\ \text{true}\}. \end{aligned}$$

From line 1, CSolve infers that argument `s` of `revstrncsvals` of has type

$$\mathtt{s} :: \{\nu : \ \mathtt{ref}(l, \{0 + 1*\}) \ \mid \ \mathtt{SAFE}(\nu) \wedge \mathtt{SIZE\_GE}(\mathtt{n})\}.$$

CSolve infers that this type is a loop invariant, and thus that the call to `strnchr` is type-correct. CSolve infers from the type of `malloc` that `v` has type

$$\begin{aligned} \mathtt{v} :: \{\nu : \ \mathtt{ref}(l_{\mathtt{v}}, \{0\}) \ \mid \ \mathtt{VALPTR}(\nu)\} \\ \mathtt{VALPTR}(p) \doteq \nu = BS(\nu) \wedge BE(\nu) - BS(\nu) = 12 \wedge \nu > 0, \end{aligned}$$

indicating that `v` is a non-NULL pointer to a 12-byte allocated region; this allows CSolve to verify the safety of the indirect field accesses.

Finally, CSolve infers that `v` and `last` refer to elements within a set of run-time locations, collectively named $l_{\mathtt{v}}$. Each location in $l_{\mathtt{v}}$ has type

$$\begin{aligned} l_{\mathtt{v}} \mapsto 0 &: \{\nu : \ \mathtt{int}(4, \{0 \pm 1*\}) \ \mid \ 0 \leq \nu\}, \\ 4 &: \{\nu : \ \mathtt{ref}(l, \{0 + 1*\}) \ \mid \ \mathtt{SAFE}(\nu) \wedge \mathtt{SIZE\_GE}(@0)\}, \\ 8 &: \{\nu : \ \mathtt{ref}(l_{\mathtt{v}}, \{0\}) \ \mid \ \nu \neq 0 \Rightarrow \mathtt{VALPTR}(\nu)\}. \end{aligned}$$

Offsets 0, 4, and 8 in the type of $l_{\mathtt{v}}$ correspond to the fields `len`, `str`, and `next`, respectively. The notation $@n$ is used in refinement predicates to refer to the value stored at offset n within the location; in this case, @0 is used to indicate that the `str` field points to an allocated region of memory whose size is at least the value given in the `len` field. The type of heap location l is as given earlier. The type of the `next` field indicates that it contains a pointer to the location $l_{\mathtt{v}}$, *i.e.,* that the `next` field contains a pointer to the same kind of structure. Thus, CSolve that lists constructed by `revstrncsvals` satisfy the above invariant.

Because the pointer `last` is returned from `revstrncsvals` and due to the call on line 2, as well as the type of $l_{\mathtt{v}}$ given above, CSolve is able to determine that the array accesses and call to `strntolower` within `lowercase_csvals` are safe, and thus prove the program is memory safe.

References

1. J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF*, 2008.
2. J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.
3. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
4. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC 02: Compiler Construction*.
5. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
6. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, pages 131–144, 2010.
7. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.