Don't Sweat the Small Stuff

Formal Verification of C Code Without the Pain

David Greenaway Japheth Lim June Andronick Gerwin Klein

NICTA and UNSW, Sydney, Australia

{first name.last name}@nicta.com.au

Abstract

We present an approach for automatically generating *provably correct* abstractions from C source code that are useful for practical implementation verification. The abstractions are easier for a human verification engineer to reason about than the implementation and increase the productivity of interactive code proof. We guarantee soundness by automatically generating proofs that the abstractions are correct.

In particular, we show two key abstractions that are critical for verifying systems-level C code: automatically turning potentially overflowing machine-word arithmetic into ideal integers, and transforming low-level C pointer reasoning into separate abstract heaps. Previous work carrying out such transformations has either done so using unverified translations, or required significant proof engineering effort.

We implement these abstractions in an existing proof-producing specification transformation framework named AutoCorres, developed in Isabelle/HOL, and demonstrate its effectiveness in a number of case studies. We show scalability on multiple OS microkernels, and we show how our changes to AutoCorres improve productivity for total correctness by porting an existing high-level verification of the Schorr-Waite algorithm to a low-level C implementation with minimal effort.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs

Keywords C verification; Isabelle/HOL

1. Introduction

Recent successes like the verified CompCert compiler [17] and the seL4 microkernel [15] show that the verification of low-level systems code has become feasible, although the effort expended for these verifications is still high: the seL4 team reports 20 person years for 10 000 source lines of C code. Formal verification at this low level is especially important in safety and security critical systems, such as operating systems, real-time systems, or language runtimes, where higher-level programming languages are not appropriate.

There are two broad approaches to low-level software verification. The first is push-button verification, where even a relatively unskilled user can use an automated verification tool to quickly verify specific properties about her program, e.g., using software model checking [3, 12]. While theoretically any property might be checkable, practical success at scale has been achieved only for specific domains, such as using APIs in correct sequence, avoiding buffer overflows, or undefined behaviour. The second approach is interactive verification, which requires skilled users and greater time investment, but is able to verify deeper properties about the system, such as the two landmark verifications [15, 17] mentioned above. This approach is more flexible; the verification engineer has complete freedom in style and form of properties as well as semantic depth. She can for instance reason simultaneously about a program with a VCG [30], prove refinement to a higher-level specification [7], and prove more complex properties such as non-interference [20].

Much work has focused on making the automated systems in the first approach more powerful. The goal of our work is to simplify the interactive second approach and to drastically increase its productivity, while simultaneously maintaining the soundness guarantees the approach provides. C verification projects report that a large part of C verification deals with mechanical complexities of the C semantics, not with key ideas of the underlying algorithms [30]. In our AutoCorres tool[1], we deal with the uninteresting complexities of C automatically, allowing the user to focus her human creativity on the algorithmic parts of verification. We do this by performing automatic, proof-producing *specification abstraction*, transforming a low-level specification in multiple steps into a more abstract one. AutoCorres presents to the end user a representation of the program that is simpler to reason about, while additionally producing a proof that the original program is a formal refinement of the final representation.

Specification abstraction has constraints that do not apply to push-button verification. Firstly, we generate a single output per program that has to be general enough not to sacrifice the flexibility and freedom of the verification engineer in her choice of property, program logic, or semantic depth. Secondly, our output specification needs to be usable and readable by humans. CEGAR, for instance, performs automatic abstraction and refinement by avoiding counter examples generated by an SMT solver. This provides good results, but a new model is automatically generated for each property of interest, and these final models are unlikely to be human readable. Finally, we aim at higher correctness assurance than other typical automated methods: our transformation steps produce formal LCF-style proofs in Isabelle/HOL [21].

PLDI '14, June 9–11, 2014, Edinburgh, UK.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594296

[1] `http://www.ssrg.nicta.com.au/projects/TS/autocorres/`

This paper extends the existing AutoCorres tool by Greenaway et al. [11] that transforms a C program into a shallow monadic embedding in the specification abstraction style described above. While previously AutoCorres performed well at control flow abstraction, it lacked abstraction support for primitive C data types such as 32- or 64-bit machine words and for pointer reasoning. Both are the source of considerable effort in C verification. Reasoning about ideal integers instead of finite words is a key difference between algorithm verification and C implementation verification, and low-level heap-based programs are of course prevalent in systems code. In a correct C implementation of an abstract algorithm, these details will typically not play a major role. This means, ideally, the user should automatically be presented with integers instead of machine words and with an abstract heap model instead of low-level C memory detail. Both are hard problems to solve automatically.

The contribution of this paper is to solve both of them by extending AutoCorres with two key abstractions that make it applicable to large-scale reasoning about heap-based C programs. Namely, the novelty lies in the combination of: (*i*) automatically carrying out word and heap abstractions on input specifications, without requiring user input; and (*ii*) simultaneously, generating a full LCF-proof that these abstractions are sound. Previous work has either axiomatised the abstraction, or required the abstractions and associated proofs to be carried out manually. Using AutoCorres, one can carry out algorithmic verification at a high level of abstraction, while obtaining a machine-checked proof that the verification holds for the C source program, or even the compiled binary when chained with recent translation validation work [25].

Additionally, we demonstrate in two case studies that AutoCorres scales to code bases of at least tens of thousands of lines, and that it indeed achieves a significantly higher level of abstraction than direct C code verification. For the former, we show statistics of running AutoCorres on a number of larger code bases used in independent ongoing verification projects. For the latter, we examine the Schorr-Waite graph marking algorithm [24], a popular pointer verification benchmark. We show that a previous verification [18] of partial correctness of the high-level algorithm requires only minimal changes to become a low-level C implementation verification of total correctness based on AutoCorres.

We begin in Sec 2 by briefly summarising the existing AutoCorres tool. Sec 3 discusses the integer abstraction for machine words and Sec 4 how we automatically eliminate low-level heap reasoning. Finally, Sec 5 summarises the case studies that substantiate the practical usability of our extensions to AutoCorres.

2. The AutoCorres Tool

Greenaway et al.'s *AutoCorres* tool [11] automatically abstracts low-level C semantics into a monadic shallow embedding in the interactive proof assistant Isabelle/HOL [21], and produces a proof that it did so correctly. In particular, AutoCorres takes as input a program in sequential C [14] which has been translated into Schirmer's *Simpl* language [23] using Norrish's C-to-Isabelle parser [27].

Because AutoCorres uses Norrish's parser's output as its input, it supports the same C subset as the parser. In particular, we support loops, function calls, type casting, pointer arithmetic, structures and recursion.

We do not support references to local variables, `goto` statements, expressions with uncontrolled side-effects, `switch` statements using fall-through, unions, floating point arithmetic, or calls to function pointers. Integer arithmetic is architecture-defined, and in our examples matches a two's-complement 32-bit system.

Fig 2 shows the example of a C function `max`, its translation max_body into Simpl by the C parser, and its abstracted functional Isabelle/HOL specification specification max′ operating on signed machine words produced by the existing AutoCorres tool. AutoCorres additionally produces a proof that max_body refines max′.

```
int max(int a, int b) {
    if (a < b)
        return b;
    return a;
}
```

```
max′ a b ≡
   if a < b then b else a
```

```
max_body ≡
   TRY
     IF {| ´a___int <_s ´b___int |} THEN
       ´ret__int :== ´b___int;;
       ´global_exn_var :== Return;;
       THROW
     ELSE
       SKIP
     FI;;
     ´ret__int :== ´a___int;;
     ´global_exn_var :== Return;;
     THROW;;
     GUARD DontReach ∅
       SKIP
   CATCH
     SKIP
   END
```

Figure 2. C function `max`, its translation max_body into Simpl by the C parser, and its AutoCorres abstraction max′.

Simpl	Monad	Definition
–	return x	$\lambda s.$ ({(Normal x, s)}, False)
Skip	skip	return ()
Basic m	modify m	$\lambda s.$ ({(Normal (), $m\ s$)}, False)
Throw	throw x	$\lambda s.$ ({(Except x, s)}, False)
Cond $c\ L\ R$	condition $c\ L\ R$	$\lambda s.$ if $c\ s$ then $L\ s$ else $R\ s$
–	fail	$\lambda s.$ ($\emptyset$, True)
Guard $t\ g\ B$	guard g	condition g skip fail

Table 1. Monadic functions with corresponding Simpl commands.

The Simpl language is a general intermediate language for sequential imperative programs, geared towards program verification. It is intentionally verbose and full of low-level detail: the parser is a trusted piece of the verification chain, and thus aims to produce the most literal and conservative translation of C possible. These details include abrupt termination as in `return`, `break` and `continue`, and a ghost variable global_exn_var_′ for recording the reason for the current exception.

The C parser emits inline Guard commands to rule out undefined behaviour in C, such as divide-by-zero, signed integer overflow, or, in this case, execution falling off the end of the (non-`void`) function.

This faithful, literal translation is not easy to reason about. AutoCorres alleviates this problem for control flow by performing a number of proof-producing abstraction steps, as depicted in Fig 1. The first of these is a plain translation from deep to shallow embedding, which enables equivalence transformations on the program by rewriting. This shallow embedding forms the monadic execution model that AutoCorres abstracted programs work in. It uses an *exception monad*, which is a *state monad* with additional support for non-determinism, exceptions and failure (the latter representing irrecoverable program failure). The concrete type of this monad in Isabelle is:

$$('s,\ 'a,\ 'e)\ monad_E \quad\equiv\quad 's \Rightarrow ((('e + 'a) \times 's)\ set \times bool$$

The $\Rightarrow$ stands for the function type that takes a single state $'s$ as input and returns a pair (results, failed), made of the set of results and a boolean failure flag. Non-deterministic functions are modelled by returning a set of possible results. Each such result contains the return value (either a normal value Normal $'a$, or an exception value Except $'e$) and the resulting state $'s$.

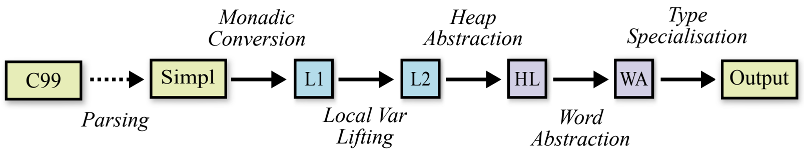

Figure 1. The steps AutoCorres takes to translate a Simpl program to an abstracted output program. Dashed arrows represent unverified translations, while solid arrows represent verified translations. Each phase beyond parsing is in Isabelle/HOL. Our extensions in this paper are the Heap Abstraction and Word Abstraction phases.

Table 1 shows the corresponding monad for each imperative Simpl construct. bind is written as $\gg=$ (or using the **do** . . . **od** syntax), and loops are represented by the combinator whileLoop *c B i* with loop condition c, body B, and initial *loop iterator* value i.

This first step is followed by a number of further transformations, for instance simplifying control flow for abrupt return, eliminating conservative translation artefacts, discharging guards against undefined behaviour, lifting local variables from the imperative state into lambda-bound variables in the monad, and type specialisation for code that can be proved to never fail or never throw an exception.

While clearly useful to the human verification engineer, all of these transformations are mostly syntactic. The semantic complexity of C's data types and reasoning about pointers, which are two of the largest effort drivers in C source code verification, remain unchanged. These are what we address in this paper. In the remainder of this paper, we describe extensions to AutoCorres that add semantic data abstractions with refinement theorems that chain down to the C code.

3. Word Abstraction

The first key contribution of this paper is automatic and verified integer abstraction for machine words, allowing mathematical reasoning over unbounded numbers.

3.1 Arithmetic in C

As a low-level language, C makes no attempt to hide details of hardware arithmetic to the programmer. For instance, on a 32-bit system, the range of the signed `int` datatype is -2^{31} to $2^{31}-1$, while the range of `unsigned int` is 0 to $2^{32}-1$. An *overflow* occurs when the result of a calculation exceeds this range, while an *underflow* occurs when the result is below the minimum value. When the context is clear, we simply use the term *overflow* to refer to both overflow and underflow.

The C standard [14] dictates different behaviours for signed and unsigned datatypes when overflow occurs. For unsigned datatypes, the result of the operation is simply calculated modulo 2^{32}; for example $2^{31} \times 2 = 0$. Signed arithmetic has stricter rules. The C standard states that it is *undefined behaviour* for a program to perform signed arithmetic that overflows: the compiler is free to assume that such behaviour will never occur and, if it does occur, is free to exhibit any behaviour it desires. In modern C compilers, this is not an academic issue: for instance, `gcc-4.7` will happily optimise the signed expression `s + 1 > s` to `true` [29].

In the context of program verification, this means that a program specification must precisely model unsigned overflow, and ensure that signed arithmetic operations will not overflow. Norrish's C parser ensures this by translating variables to Isabelle/HOL's finite word types; `unsigned int`'s are translated into the unsigned *word32* type, while `int`'s are translated into the signed *sword32* type. Additionally, for signed operations, the C parser emits guard statements to check that the result does not overflow. For example, the

Incorrect Equation	Counter-example	
$s = s + 1 - 1$	$s = 2^{31} - 1$	*(undefined)*
$s = -(-s)$	$s = -2^{31}$	*(undefined)*
$u + 1 > u$	$u = 2^{32} - 1$	*(incorrect)*
$u \times 2 = 4 \longrightarrow u = 2$	$u = 2^{31} + 2$	*(incorrect)*
$-u = u \longrightarrow u = 0$	$u = 2^{31}$	*(incorrect)*

Table 2. Examples of incorrect mathematical reasoning in C. Variable s is a 32-bit signed `int`, while u is a 32-bit `unsigned int`.

signed C expression `a + b` is translated into:

$$\begin{array}{l}\textbf{do}\ \mathsf{guard}\ (\lambda s.\ \mathsf{INT\_MIN} \le \mathsf{sint}\ a + \mathsf{sint}\ b);\\ \quad \mathsf{guard}\ (\lambda s.\ \mathsf{sint}\ a + \mathsf{sint}\ b \le \mathsf{INT\_MAX});\\ \quad \mathsf{return}\ (a +_s b)\\ \textbf{od}\end{array}$$

Here, the function sint of type *sword32* ⇒ *int* converts the finite 32-bit signed word type into the unbounded Isabelle/HOL integer type. The analogous function unat similarly converts unsigned words into natural numbers, and is used for unsigned expressions. We use the s suffix (such as on the operator $+_s$) to indicate that the operation is being carried out on signed words, and use a similar w prefix for unsigned word arithmetic.

While this approach allows C arithmetic to be correctly modelled, actually *reasoning* about it remains burdensome. Table 2 lists some 'obvious' mathematical identities that are simply not true when reasoning about C programs. Further, while Isabelle/HOL contains extensive libraries of theorems about natural numbers and integers, these theorems do not apply when verifying C programs. Larger verification projects also experience the burden of word-proofs: approximately 25% of the 30 000 lines of proof library developed in the seL4 project [15] were dedicated to word-arithmetic proofs.

3.2 Word abstraction

Ideally, we would like to abstract *word32* and *sword32* data types into unbounded natural numbers and integers respectively; this would avoid the corner cases described above, and also allow Isabelle/HOL's existing proof libraries to be freely used in program proofs. The question is how this can be done in a sound manner. We can't simply pretend that the underlying hardware can perform arithmetic on arbitrarily large numbers, nor can we ignore C's requirement that signed arithmetic never overflows—or can we?

We observe that the program verifier must *already* prove that signed arithmetic doesn't fall out of the range -2^{31} to $2^{31}-1$, because the C standard demands it. The C parser already inserts corresponding proof obligations. We can thus abstract *sword32* types into *int* types, utilising the existing guard statements to know that the abstract values will always remain in the range of representable values at the concrete level.

Unsigned arithmetic is slightly more difficult; the Simpl program will not contain any guards to ensure that overflow doesn't occur, and, more importantly, the source C program may actually *rely* on overflow to occur. Despite this, for many functions unsigned overflow is not expected and—if the program verifier is willing to prove that it does not occur by having additional guard statements in their abstracted output—we can abstract unsigned arithmetic to natural numbers. We allow the user to select whether to use word abstraction or not on a per-function basis.

An example where such abstraction makes sense is in a binary search that calculates the middle element of an array:

$$\textbf{unsigned int}\ \mathsf{m} = (\mathsf{l} + \mathsf{r})\ /\ 2;$$

A typical verification condition that arises is showing the selected element remains between the elements *l* and *r*:

$$l <_w r \longrightarrow l \le_w (l +_w r)\ \mathsf{div}_w\ 2 \wedge (l +_w r)\ \mathsf{div}_w\ 2 <_w r$$

If the terms *l*, *m* and *r* were of type *nat*, this theorem is solved automatically using Isabelle/HOL's built-in `auto` tactic. On the original *word32* type, however, an additional precondition $\mathsf{unat}\ l + \mathsf{unat}\ r < 2^{32}$ is required and the proof term must be manually lifted into the naturals before it can finally be solved using existing theorems in Isabelle/HOL's library[2].

3.3 Performing the abstraction

Our implementation of word abstraction converts local variables and arguments of functions, but does not attempt to modify values stored in memory or global variables. This means that the program's state remains unmodified, and the abstraction process only has to adjust expressions in the program.

We generate a refinement theorem showing that the original concrete program *C* refines our generated abstract program *A*:

$$\begin{array}{l}\mathsf{abs\_w\_stmt}\ P\ rx\ ex\ A\ C \equiv \\ \quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ s) \longrightarrow \\ \quad\quad (\forall (r, t){\in}\mathsf{results}\ (C\ s). \\ \quad\quad\quad \mathsf{case}\ r\ \mathsf{of} \\ \quad\quad\quad\quad \mathsf{Normal}\ v \Rightarrow (\mathsf{Normal}\ (rx\ v), t) \in \mathsf{results}\ (A\ s) \\ \quad\quad\quad\ \mid \mathsf{Except}\ e \Rightarrow (\mathsf{Except}\ (ex\ e), t) \in \mathsf{results}\ (A\ s)) \wedge \\ \quad\quad \neg\ \mathsf{failed}\ (C\ s)\end{array}$$

The precondition *P* states under which conditions our abs_w_stmt assertion will hold. The theorem states that, assuming the abstract program doesn't fail, then (*i*) if *A* returns a value, it will be the same value as *C* abstracted through the function *rx*; (*ii*) similarly, if *A* raises an exception, it will have the same value as *C* abstracted through *ex*; (*iii*) finally, if *A* doesn't fail, then neither will *C*.

Our algorithm for generating an abstracted version of the program is in the form of a set of syntax-directed rules. These translation rules can be applied in any setting, but in our context of Isabelle/HOL, we use them by (*i*) first proving the translation rules correct, and then (*ii*) using Isabelle/HOL's resolution engine to apply these rules. By carrying out these two steps in Isabelle/HOL, we simultaneously obtain both the abstracted program and an LCF-style proof of correctness that the abstraction is sound.

We begin this process by generating a *schematic lemma* where *C* is instantiated to the program we want to abstract, while *rx* and *ex* are set to an appropriate abstraction function. The abstract program *A* and the precondition *P* are left unspecified (or *schematic*) and are given the notation $?A_1$ and $?P_1$ respectively. As our algorithm proceeds, these values will be incrementally instantiated. For our midpoint example above, for instance, we start with the tautology:

$$\frac{\mathsf{abs\_w\_stmt}\ ?P_1\ \mathsf{unat}\ \mathsf{id}\ ?A_1\ (\mathsf{return}\ ((l +_w r)\ \mathsf{div}_w\ 2))}{\mathsf{abs\_w\_stmt}\ ?P_1\ \mathsf{unat}\ \mathsf{id}\ ?A_1\ (\mathsf{return}\ ((l +_w r)\ \mathsf{div}_w\ 2))}$$

[2] A challenge to solve this seemingly trivial goal was issued to 3 experienced verification engineers, with 10 minutes being the median time required to discharge the goal. The human effort for the *nat* version is effectively zero.

Our goal is to discharge the assumption, leaving only the conclusion. We find a rule from our ruleset that pattern-matches the concrete program. Table 3 shows a sample of the word abstraction rules used. In this example, we wish to abstract the return expression in our concrete program, using the rule $\mathrm{W_{RET}}$. This instantiates A_1 to return $?A_2$, where $?A_2$ is a new schematic variable. Similarly, $?P_1$ in instantiated to $(\lambda\_.\ ?P_2)$:

$$\frac{\mathsf{abs\_w\_val}\ ?P_2\ \mathsf{unat}\ ?A_2\ ((l +_w r)\ \mathsf{div}_w\ 2)}{\begin{array}{l}\mathsf{abs\_w\_stmt}\ (\lambda s.\ ?P_2)\ \mathsf{unat}\ \mathsf{id} \\ \quad (\mathsf{return}\ ?A_2)\ (\mathsf{return}\ ((l +_w r)\ \mathsf{div}_w\ 2))\end{array}}$$

where $\mathsf{abs\_w\_val}\ P\ f\ a\ c \equiv P \longrightarrow a = f\ c$. That is, under precondition *P*, *a* is the abstract version of *c* using the abstraction function *f*. We again find a rule that matches this new proposition; in this case, $\mathrm{W_{DIV}}$:

$$\frac{\begin{array}{c}\mathsf{abs\_w\_val}\ ?P_3\ \mathsf{unat}\ ?A_3\ (l +_w r) \\ \mathsf{abs\_w\_val}\ ?P_4\ \mathsf{unat}\ ?A_4\ 2\end{array}}{\begin{array}{l}\mathsf{abs\_w\_stmt}\ (\lambda s.\ ?P_3 \wedge ?P_4)\ \mathsf{unat}\ \mathsf{id} \\ \quad (\mathsf{return}\ (?A_3\ \mathsf{div}\ ?A_4))\ (\mathsf{return}\ ((l +_w r)\ \mathsf{div}_w\ 2))\end{array}}$$

Applying the rule leaves us with two new assumptions to discharge. Solving the first will instantiate $?A_3$, the left-hand side of the division, while discharging the second will instantiate $?A_4$, the right-hand side. We continue this process of discharging assumptions and creating new ones, using the rules $\mathrm{W_{SUM}}$ and $\mathrm{W_{TRIV}}$, until we have no assumptions left and are left with just the conclusion:

$$\begin{array}{l}\mathsf{abs\_w\_stmt}\ (\lambda s.\ \mathsf{unat}\ l + \mathsf{unat}\ r \le \mathsf{UINT\_MAX})\ \mathsf{unat}\ \mathsf{id} \\ \quad (\mathsf{return}\ ((\mathsf{unat}\ l + \mathsf{unat}\ r)\ \mathsf{div}\ 2))\ (\mathsf{return}\ ((l +_w r)\ \mathsf{div}_w\ 2))\end{array}$$

The values unat *l* and unat *r* correspond to the abstract versions of our concrete program's input parameters. To convert this theorem into an abstract function, we replace unat *l* and unat *r* with fresh variables. Additionally, the theorem only holds under the precondition that $\mathsf{unat}\ l + \mathsf{unat}\ r \le \mathsf{UINT\_MAX}$; we prepend a guard statement ensuring that this holds. The generated abstraction thus becomes:

$$\begin{array}{l}\mathbf{do}\ \mathsf{guard}\ (\lambda s.\ l + r \le \mathsf{UINT\_MAX}); \\ \quad \mathsf{return}\ (l + r\ \mathsf{div}\ 2) \\ \mathbf{od}\end{array}$$

AutoCorres has approximately 40 rules built-in to process all C statements and expressions, and uses an additional 11 for each type that needs to be abstracted (e.g., signed words and unsigned words). While typically these rules need not be modified (or even understood) by users of AutoCorres, the rule sets can be extended if the user wishes to abstract code-specific idioms that are sound at the concrete level but become unprovable after abstraction. For instance, the expression `x > x + y` can be used in C to determine if the unsigned addition of `x` and `y` overflows; after unsigned word abstraction, however, the user will be obliged to prove that `x + y` doesn't overflow, making the test useless. By extending the word-abstraction rule-set with the custom rule:

$$\frac{\mathsf{abs\_w\_val}\ P\ \mathsf{unat}\ x\ x' \qquad \mathsf{abs\_w\_val}\ Q\ \mathsf{unat}\ y\ y'}{\mathsf{abs\_w\_val}\ (P \wedge Q)\ \mathsf{id}\ (\mathsf{UINT\_MAX} < x + y)\ (x' +_w y' <_w x')}$$

the test will be abstracted into the expression $x + y \le \mathsf{UINT\_MAX}$, allowing the original intent of the concrete code to be captured in the abstraction.

Word abstraction is effective: for example, AutoCorres's output of the `max` function in Fig 2 precisely matches Isabelle's built-in definition of max on the *nat*'s; further, AutoCorres's abstraction of the standard C implementation of Euclid's greatest-common-denominator algorithm is equal to return (gcd *a b*), where gcd is Isabelle's implementation on the *nat*'s. More complex usages to word arithmetic invariably cause the abstracted program to also be more complex, as the user becomes obliged to prove the arithmetic does not overflow. In our experience, however, the abstracted version tends to be far simpler to reason about than the original input program.

$$\frac{\begin{array}{c}\mathsf{abs\_w\_stmt}\ P\ rx_1\ ex\ L\ L' \\ \forall\, r\ r'.\ \mathsf{abs\_w\_val}\ \mathsf{True}\ rx_1\ r\ r' \longrightarrow \\ (\mathsf{abs\_w\_stmt}\ (Q\ r)\ rx_2\ ex\ (R\ r)\ (R'\ r'))\end{array}}{\begin{array}{l}\mathsf{abs\_w\_stmt}\ P\ rx_2\ ex \\ \quad (\mathbf{do}\ v \leftarrow L;\ \mathsf{guard}\ (Q\ v);\ R\ v\ \mathbf{od})\ (L' \gg\!= R')\end{array}}\ \mathrm{W_{BIND}}$$

$$\frac{\forall\, s.\ \mathsf{abs\_w\_val}\ (Q\ s)\ rx\ a\ c}{\mathsf{abs\_w\_stmt}\ Q\ rx\ ex\ (\mathsf{return}\ a)\ (\mathsf{return}\ c)}\ \mathrm{W_{RET}}$$

$$\frac{\mathsf{abs\_w\_val}\ P\ \mathsf{unat}\ a\ a' \qquad \mathsf{abs\_w\_val}\ Q\ \mathsf{unat}\ b\ b'}{\mathsf{abs\_w\_val}\ (P \wedge Q \wedge a + b \leq \mathsf{UINT\_MAX})\ \mathsf{unat}\ (a + b)\ (a' +_w b')}\ \mathrm{W_{LE}}$$

$$\frac{\mathsf{abs\_w\_val}\ P\ \mathsf{unat}\ a\ a' \qquad \mathsf{abs\_w\_val}\ Q\ \mathsf{unat}\ b\ b'}{\mathsf{abs\_w\_val}\ (P \wedge Q \wedge a + b \leq \mathsf{UINT\_MAX})\ \mathsf{unat}\ (a + b)\ (a' +_w b')}\ \mathrm{W_{SUM}}$$

$$\frac{\mathsf{abs\_w\_val}\ P\ \mathsf{unat}\ a\ a' \qquad \mathsf{abs\_w\_val}\ Q\ \mathsf{unat}\ b\ b'}{\mathsf{abs\_w\_val}\ (P \wedge Q)\ \mathsf{unat}\ (a\ \mathsf{div}\ b)\ (a'\ \mathsf{div}_w\ b')}\ \mathrm{W_{DIV}}$$

$$\frac{}{\mathsf{abs\_w\_val}\ \mathsf{True}\ f\ (f\ b)\ b}\ \mathrm{W_{TRIV}}$$

Table 3. A selection of word abstraction rules. The Operators $+_w$, div_w and $\leq_w$ operate on *word32* types, while unannotated operators are over the natural numbers.

Finally, this proof-producing method of abstracting programs gives end-users an assurance that the output of AutoCorres is sound: the word abstraction proofs fit into a chain of proofs linking the original C-Simpl input to the final AutoCorres output.

4. Heap Abstraction

This section presents the second key contribution of this paper. We automatically lift reasoning from low-level C memory to a more abstract notion of multiple separate heaps that is commonly used in high-level reasoning about pointer algorithms, while still allowing the user to carry out low-level reasoning on specific functions that require it. As with the previous section, we do so while generating a proof of correctness showing that the abstraction is sound.

4.1 Byte-level versus typed-heap reasoning

C programs frequently require low-level access to memory. This includes accessing memory in a byte-by-byte manner (such as `memcpy` or `memset` implementations) or reusing the same region of memory as a different type (such as in unions, `malloc` or `free`).

When reasoning about C programs, one of the first questions to arise is how memory, or the system's *heap*, should be formally modelled. A typical approach in higher-level languages such as Java [28] is to represent the contents of memory locations as a datatype:

$$\textbf{datatype}\ \mathit{value} = \mathsf{Int}\ \mathit{int} \mid \mathsf{Float}\ \mathit{float} \mid \mathsf{IntPtr}\ \mathit{addr} \mid \cdots$$

The heap can then be modelled as a function of type $\mathit{addr} \Rightarrow \mathit{value}$ converting addresses to their values. However, such a heap representation prevents us from performing the byte-level reasoning described above.

An alternative approach is to simply model memory as a function from addresses to individual bytes. For example, on a 32-bit system, the heap would have type $\mathit{word32} \Rightarrow \mathit{word8}$. On the surface, this naïve approach has many benefits: it is easy to understand, faithful to how the hardware functions at a low level, and allows low-level C code that interacts with memory at a low level (such as `memcpy`, `memset`, casting pointers between types, etc.) to be reasoned about. While not perfect (for example, there is no way to represent unmapped

```
void swap(unsigned *a, unsigned *b)
{
    unsigned t = *a;
    *a = *b;
    *b = t;
}
```

$$\begin{array}{l}\mathsf{swap}'\ a\ b \equiv \\ \quad \mathbf{do}\ \mathsf{guard}\ (\lambda s.\ \mathsf{ptr\_aligned}\ a \wedge 0 \notin \{a\ ..\ {+}\mathsf{obj\_size}\ a\}); \\ \quad\quad t \leftarrow \mathsf{gets}\ (\lambda s.\ \mathsf{read}\ (\mathsf{heap}'\ s)\ a); \\ \quad\quad \mathsf{guard}\ (\lambda s.\ \mathsf{ptr\_aligned}\ b \wedge 0 \notin \{b\ ..\ {+}\mathsf{obj\_size}\ b\}); \\ \quad\quad \mathsf{modify}\ (\lambda s.\ \mathsf{heap}'\mathsf{\_update}\ (\lambda s'.\ \mathsf{write}\ s'\ a\ (\mathsf{read}\ (\mathsf{heap}'\ s)\ b))\ s); \\ \quad\quad \mathsf{modify}\ (\mathsf{heap}'\mathsf{\_update}\ (\lambda s.\ \mathsf{write}\ s\ b\ t)) \\ \quad \mathbf{od}\end{array}$$

Figure 3. A simple implementation of `swap` in ANSI C, and its translation by AutoCorres, without heap abstraction.

or invalid addresses), if we are unable to reason about this simple model, we are going to struggle with anything more sophisticated.

Even this simple heap model, unfortunately, is difficult to work with. Consider the simple word-swap function shown in Fig 3 with its translation into Isabelle/HOL using an unmodified version of AutoCorres. We may wish to prove the Hoare-triple:

$$\begin{array}{l}\{\!|\lambda s.\ \mathsf{read}\ (\mathsf{heap}'\ s)\ a = v_a \wedge \mathsf{read}\ (\mathsf{heap}'\ s)\ b = v_b|\!\} \\ \quad \mathsf{swap}'\ a\ b \\ \{\!|\lambda rv\ s.\ \mathsf{read}\ (\mathsf{heap}'\ s)\ a = v_b \wedge \mathsf{read}\ (\mathsf{heap}'\ s)\ b = v_a|\!\}\end{array}$$

Here, the function read takes the current state of the system's heap and a pointer of type $'a\ ptr$, and decodes the bytes at this location into an object of type $'a$ (in this example, a *word32*). The Hoare-triple states that if a contains value v_a and b contains value v_b in our initial state, then the two values will be swapped in the final state.

This statement is not correct as written, however. For the post-condition to hold, the precondition must be strengthened to ensure that: (*i*) the pointers a and b are aligned to a 4-byte boundary; (*ii*) the pointers a and b are not `NULL`; (*iii*) the pointers a and b do not wrap around the end of the address space; and (*iv*) the pointers a and b do not *partially* overlap (though if the pointers are equal, the function remains correct). The first three of these additional conditions are required by the C standard, while the fourth is required for the post-condition to hold[3].

Taking these additional preconditions into account, the correct Hoare-triple for this function is:

$$\begin{array}{l}\{\!|\lambda s.\ (\mathsf{read}\ (\mathsf{heap}'\ s)\ a = x \wedge \mathsf{read}\ (\mathsf{heap}'\ s)\ b = y)\ \wedge \\ \quad\quad (\mathsf{ptr\_aligned}\ a \wedge \mathsf{ptr\_aligned}\ b)\ \wedge \\ \quad\quad (0 \notin \{a\ ..\ {+}\mathsf{obj\_size}\ a\} \wedge 0 \notin \{b\ ..\ {+}\mathsf{obj\_size}\ b\})\ \wedge \\ \quad\quad (a \neq b \longrightarrow \{a\ ..\ {+}\mathsf{obj\_size}\ a\} \cap \{b\ ..\ {+}\mathsf{obj\_size}\ b\} = \emptyset)|\!\} \\ \quad \mathsf{swap}'\ a\ b \\ \{\!|\lambda rv\ s.\ \mathsf{read}\ (\mathsf{heap}'\ s)\ a = y \wedge \mathsf{read}\ (\mathsf{heap}'\ s)\ b = x|\!\}\end{array}$$

Here, $\mathsf{ptr\_aligned}$ indicates that a pointer of type $'a\ ptr$ has the correct alignment required by type $'a$, while $\{a\ ..\ {+}\mathsf{obj\_size}\ a\}$ indicates the range of memory addresses occupied by the object in memory. With this strengthened precondition, this Hoare-triple can now be proven correct, with a little manual reasoning showing that updating parts of the heap disjoint to a read don't affect that read.

Clearly, if we wish to verify functions with significantly more complexity than `swap`, a better approach is needed.

[3] Low-level language aficionados will observe that in this simple example not all the preconditions are strictly required: for instance, if the pointers a and b are both aligned, then they can't wrap around the edge of memory, nor can they partially overlap. However, in more complex examples—such as swapping larger `structs`—all of these preconditions are required.

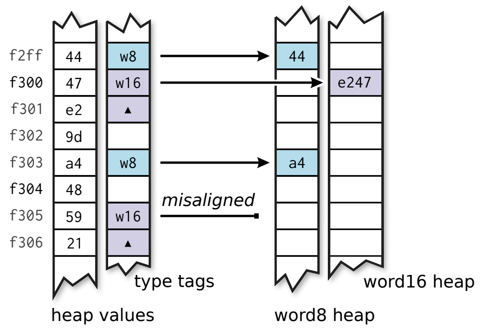

Figure 4. The heap lifting function.

4.2 The heap lifting approach

As we will see with the related work in Sec 6, most C verification frameworks aim at a high-level memory model, easing the reasoning about programs that interact with memory. Keeping the ability to do byte-level reasoning, and providing formal argument about the soundness of the high-level model are two essential characteristics, which Tuch et al.'s *heap lifting* logic [27] provides. Our work builds on Tuch's approach, while addressing its scalability problem explained further below. We hide the underlying C memory and provide the user with pure abstracted heaps, exposing the underlying model only when explicitly requested to.

In Tuch's model, ghost state annotations are added to the C source code to indicate what concrete type each address in the heap should be interpreted as. Each address in memory can be marked as either the first byte of a C type, such as an `int` or `struct node`; the *footprint* of an earlier type, where the address simply continues a previous type; or untyped memory. Such *type tag* source annotations are typically required only after particular addresses in memory need to be treated as a different type, such as calls to `malloc` or `free`; the rest can be inferred automatically.

Tuch next defines a class of functions heap_lift (see Fig 4) that projects the byte-level heap (of type $word32 \Rightarrow word8$) into a partial object-level heap (of type $'a\ ptr \Rightarrow\ 'a\ option$):

$$\begin{aligned}&\mathsf{heap\_lift}\ s\ p \equiv\\ &\quad \mathsf{if\ type\_tag\_valid}\ s\ p \wedge \mathsf{ptr\_aligned}\ p \wedge 0 \notin \{p\ ..\ {+}\mathsf{obj\_size}\ p\}\\ &\quad \mathsf{then\ Some\ (read}\ s\ p)\ \mathsf{else\ None}\end{aligned}$$

The Isabelle $'a\ option$ type has two possible values: None or Some a, where a is an element of type $'a$. The function "the" is defined such that the (Some x) $= x$ holds.

In the lifted heap, a particular address contains a valid object if and only if (*i*) the entire range of addresses occupied by the object are correctly tagged; (*ii*) the pointer being accessed is correctly aligned; and (*iii*) the pointer is not `NULL` and does not wrap around the end of the address space. If any of these conditions fail to hold, the address resolves to None. The definition of the projected heap immediately gives rise to the following rules:

$$\frac{\mathsf{heap\_lift}\ s\ p = \mathsf{Some}\ v}{\mathsf{ptr\_aligned}\ p} \qquad \frac{\mathsf{heap\_lift}\ s\ p = \mathsf{Some}\ v}{0 \notin \{p\ ..\ {+}\mathsf{obj\_size}\ p\}}$$

Informally, if a value is non-None on the projected heap, then the pointer to it is valid on the concrete heap.

Moreover, the user can reason that objects in the projected heap cannot partially overlap other objects or alias objects of different types: for two objects to partially overlap, one object's type tags would be incorrect. This means that writes to valid addresses are equivalent to functional updates on the projected heap:

$$\frac{\mathsf{heap\_lift}\ s\ p = \mathsf{Some}\ v'}{\mathsf{heap\_lift}\ (\mathsf{write}\ s\ p\ v) = (\mathsf{heap\_lift}\ s)(p := \mathsf{Some}\ v)}$$

Here, $f(n := v)$ indicates function f is updated at n to value v.

Reasoning at the level of lifted heaps greatly simplifies proofs interacting with the heap. The correctness statement of our `swap` function becomes:

$$\begin{aligned}&\{\!|\lambda s.\ \mathsf{heap\_lift}\ (\mathsf{heap}'\ s)\ a = \mathsf{Some}\ v_a \wedge\\ &\qquad (\mathsf{heap\_lift}\ (\mathsf{heap}'\ s)\ b = \mathsf{Some}\ v_b)|\!\}\\ &\quad \mathsf{swap}'\ a\ b\\ &\{\!|\lambda rv\ s.\ \mathsf{heap\_lift}\ (\mathsf{heap}'\ s)\ a = \mathsf{Some}\ v_b \wedge\\ &\qquad\quad (\mathsf{heap\_lift}\ (\mathsf{heap}'\ s)\ b = \mathsf{Some}\ v_a)|\!\}\end{aligned}$$

This can be proved by simply unfolding the definition of swap$'$, executing a VCG and running Isabelle/HOL's `auto` tactic with the above rules.

4.3 Limitations of the heap lifting approach

Tuch's heap lifting approach relies heavily on Isabelle's simplifier to automatically apply recursive conditional rewrite rules to change low-level C operations into high-level heap updates. As programs become more complex, so does application of the lifting predicates. For instance, consider Suzuki's challenge [26] to prove that the following fragment returns 4 under the assumption that the four pointers `w`, `x`, `y` and `z` are distinct:

```
w−>next = x; x−>next = y; y−>next = z; x−>next = z;
w−>data = 1; x−>data = 2; y−>data = 3; z−>data = 4;
return w−>next−>next−>data;
```

On this fragment of code, Isabelle/HOL fails to apply the heap-lifting rules described above. The primary problem is the deep nesting of write operations, preventing Isabelle's simplifier from identifying which rewrite rules to apply, because their recursive preconditions become too large and too deep. Basically, at even a moderately larger scale, the prover is already overloaded just applying heap abstraction, and never proceeds to reasoning about the actual semantics of the program.

Ad hoc heap lifting is also unsatisfactory on a more fundamental usability level: while C programs need byte-level access to memory on occasion, most C functions are type-safe. Ideally, for the majority of type-safe code, we should present the user with a specification that operates directly on the lifted heap, instead of requiring the user to manually appeal to heap abstraction predicates.

4.4 Adding a state abstraction step

The approach we propose in this paper is to add an abstraction step where we automatically translate the byte-level heap model within the state into a multiple, typed-heaps state. We abstract the record type *globals*, generated by the C parser to represent programs states, into a new record type *abs_globals* containing one heap per type used in the program. Reasoning on such an abstracted state removes the need to invoke lifting rewrite rules.

We start by analysing the source program to determine which types the program accesses on the heap, i.e., which types are used as arguments to the read and write functions described above. For each heap type $'a$ required, we place two functions into the *abs_globals* record: an `is_valid` function of type $'a\ ptr \Rightarrow bool$ and a `heap` function of type $'a\ ptr \Rightarrow\ 'a$. The former function determines if a particular address contains a valid value (that is, $\exists x.$ heap_lift $s\ p =$ Some x), while the latter function contains the actual values of each address (the (heap_lift $s\ p$)):

```
record abs_globals =
        is_valid_w32 :: word32 ptr ⇒ bool
        heap_w32 :: word32 ptr ⇒ word32
        is_valid_node_C :: node_C ptr ⇒ bool
        heap_node_C :: node_C ptr ⇒ node_C
        . . .
```

While splitting data and validity information initially appears more complex than simply having a partial function $'a\ ptr \Rightarrow\ 'a\ option$, we

$$\frac{\mathsf{abs\_h\_stmt}\ L'\ L \qquad \forall r.\ \mathsf{abs\_h\_stmt}\ (R'\ r)\ (R\ r)}{\mathsf{abs\_h\_stmt}\ (L' \gg\!= R')\ (L \gg\!= R)}\ \text{H}_{\text{BIND}}$$

$$\frac{\mathsf{abs\_h\_val}\ P\ a\ c}{\mathsf{abs\_h\_stmt}\ (\mathbf{do}\ \mathsf{guard}\ P;\quad \mathsf{gets}\ a\ \mathbf{od})\ (\mathsf{gets}\ c)}\ \text{H}_{\text{GETS}}$$

$$\frac{\mathsf{abs\_h\_modifies}\ P\ a\ c}{\mathsf{abs\_h\_stmt}\ (\mathbf{do}\ \mathsf{guard}\ P;\ \mathsf{modify}\ a\ \mathbf{od})\ (\mathsf{modify}\ c)}\ \text{H}_{\text{MODIFY}}$$

$$\frac{\mathsf{abs\_h\_val}\ P\ a\ c}{\begin{array}{l}\mathsf{abs\_h\_val}\ (\lambda s.\ P\ s \wedge \mathsf{is\_valid\_w32}\ s\ (a\ s))\ (\lambda s.\ \mathsf{True})\\ \quad (\lambda s.\ \mathsf{ptr\_aligned}\ (c\ s) \wedge 0 \notin \{c\ s\ ..\ +\mathsf{obj\_size}\ (c\ s)\})\end{array}}\ \text{H}_{\text{PTR}}$$

Table 4. Syntax-directed rules used while abstracting `swap`.

have found this approach allows a greater separation of concerns between *what* data is contained at an address, and *which* addresses are valid. Further, while the data at a particular address frequently changes, the validity of an address rarely changes. Splitting the two dimensions makes it clear that changes to one are independent of the other.

4.5 Abstracting programs to typed-heaps states

We wish to generate both a new version of the input program using the new abstract state and a proof showing that our abstraction is correct: that the input *concrete* program C refines the *abstract* one A (here *concrete* just refers to the state before applying heap abstraction). For this, we need a refinement relation between their states. So for each C, we generate a *state abstraction* function st of type *globals* $\Rightarrow$ *abs_globals*, where the abstract validity functions have the value heap_lift $s\ p \neq$ None and the abstract data functions have the value the (heap_lift $s\ p$). We then show refinement using:

$$\begin{array}{l}\mathsf{abs\_h\_stmt}\ A\ C \equiv\\ \quad \forall s.\ \neg\ \mathsf{failed}\ (A\ (\mathsf{st}\ s)) \longrightarrow\\ \quad\quad (\forall (r, t){\in}\mathsf{results}\ (C\ s).\\ \quad\quad\quad \mathsf{case}\ r\ \mathsf{of}\\ \quad\quad\quad\quad \mathsf{Normal}\ r \Rightarrow (\mathsf{Normal}\ r, \mathsf{st}\ t) \in \mathsf{results}\ (A\ (\mathsf{st}\ s))\\ \quad\quad\quad\ \mid \mathsf{Except}\ r \Rightarrow (\mathsf{Except}\ r, \mathsf{st}\ t) \in \mathsf{results}\ (A\ (\mathsf{st}\ s))) \wedge\\ \quad\quad \neg\ \mathsf{failed}\ (C\ s)\end{array}$$

Informally, the abs_h_stmt predicate states that a monadic program C is a refinement of an abstract program A when the state is abstracted using st if—for all states where A does not fail—then, for every concrete state that program C produces, program A produces a corresponding abstract state. Additionally, for each abstract state, if program A does not fail, then neither will C. The first of these conditions states that program A produces a superset of states of program C. Thus, if a property holds for all states in program A, then we can reason that it also holds on program C. The second condition allows us to reason that program C will never fail by proving that program A never fails. With these two conditions, we can typically prove useful properties about our original concrete program without needing to ever reason on it directly, as we will demonstrate in Sec 5.

Our algorithm to generate the abstract program again uses a set of syntax-directed rules to translate statements and expressions in our concrete program to abstract equivalents. As in word abstraction, we apply the rules using Isabelle/HOL's unification engine to simultaneously generate the abstract version of the program and a proof that the abstraction is sound. Table 4 shows a subset of the 35 rules built into AutoCorres used for heap abstraction.

The rules split into three categories: (*i*) abs_h_stmt rules, which abstract statements; (*ii*) abs_h_modifies rules, which abstract abs_h_modifies statements that perform heap updates; and (*iii*) abs_h_val rules, which abstract expressions, where:

$$\mathsf{abs\_h\_val}\ P\ a\ c \equiv \forall s.\ P\ (\mathsf{st}\ s) \longrightarrow c\ s = a\ (\mathsf{st}\ s)$$

$$\begin{array}{l}\mathbf{do}\ \mathsf{guard}\ (\lambda s.\ \mathsf{is\_valid\_w32}\ s\ a);\\ \quad t \leftarrow \mathsf{gets}\ (\lambda s.\ s[a]);\\ \quad \mathsf{guard}\ (\lambda s.\ \mathsf{is\_valid\_w32}\ s\ b);\\ \quad \mathsf{modify}\ (\lambda s.\ s[a := s[b]]);\\ \quad \mathsf{modify}\ (\lambda s.\ s[b := t])\\ \mathbf{od}\end{array}$$

Figure 5. The `swap` function translated by AutoCorres with heap abstraction enabled.

$$\mathsf{abs\_h\_modifies}\ P\ a\ c \equiv \forall s.\ P\ (\mathsf{st}\ s) \longrightarrow \mathsf{st}\ (c\ s) = a\ (\mathsf{st}\ s)$$

Informally, when carrying out heap abstraction, three primary cases must be handled:

Heap reads: Operations on the concrete specification that access the heap, such as read, need to be abstracted into an access of the equivalent abstract function. One difficulty is that for many concrete operations, the abstract equivalent is only sound under side-conditions: a read is only equivalent to a functional access if the pointer being read from is valid. To resolve this, when translating an expression, we emit suitable abstract guard statements for each translated expression.

Heap modifies: Operations that modify the heap are handled similarly to heap reads. A write operation becomes a functional update on the appropriate heap, again with the side-condition that the pointer being written to is valid.

Guard statements: Guards on the concrete specification are abstracted into accesses of the appropriate `is_valid` function, in particular guard statements that pointers are aligned (ptr_aligned), are non-`NULL` and don't wrap memory ($0 \notin \{p\ ..\ +\mathsf{obj\_size}\ p\}$).

Some care must also be taken with C structures. The C parser translates field accesses such as p−>data into pointer offset values, such as read s (Ptr (ptr_val $p +_w$ offset ′′data′′)). A naïve translation of such a term will convert this into direct pointer access of `data`, while we would prefer to present to the user this statement as accessing the data_C field of the *node_C* record.

Fig 5 shows the `swap` program after heap abstraction has been applied. We use the notation $s[p]$ and $s[p := v]$ for reading and writing to the appropriate heap for pointer p in state s, respectively. As expected, read and write operations on the heap are converted to functional accesses and updates, while checks of pointers have been abstracted into checks on the is_valid_w32 function. Our correctness statement for `swap` is as follows:

$$\begin{array}{l}\{\!|\lambda s.\ \mathsf{is\_valid\_w32}\ s\ a \wedge \mathsf{is\_valid\_w32}\ s\ b \wedge s[a] = x \wedge s[b] = y|\!\}\\ \quad \mathsf{swap}'\ a\ b\\ \{\!|\lambda rv\ s.\ \mathsf{is\_valid\_w32}\ s\ a \wedge \mathsf{is\_valid\_w32}\ s\ b \wedge s[a] = y \wedge s[b] = x|\!\}\end{array}$$

This goal is automatically discharged by applying a VCG and running Isabelle/HOL's built-in `auto` tactic, without needing to appeal to further rules. Since we have eliminated any application of complex conditional rewrites for the prover in a separate guided abstraction phase, Isabelle can now work directly on simpler data types with all the power of its built-in automation.

More complex examples, such as Suzuki's challenge described above, can also be solved simply: Isabelle/HOL's `auto` immediately discharges the generated verification conditions. This is a first indication that we have solved the inherent scalability problem of Tuch's approach. Sec 5 will show in more detail that our abstracted programs are now as easy to reason about as higher-level specifications that have been manually tuned for formal verification.

Our abstraction is integrated into AutoCorres, which generates heap-lifted specifications with a formal proof that they are correct abstractions of their input C programs. The internal rules used to carry out the abstraction need not be understood by users of AutoCorres, but, like with word abstraction, can be extended by end-users

Program	LoC	Functions	CPU Time (s)		Lines of Spec		Avg. Term Size	
			C Parser	AutoCorres	C Parser	AutoCorres	C Parser	AutoCorres
seL4 kernel	10 121	551	1443.6	2368.1	20 576	11 928	318	112
CapDL SysInit	2 079	163	130.9	743.3	3 353	2 183	184	72
Piccolo kernel	936	56	37.6	214.9	1 748	1 198	372	182
eChronos	563	40	11.7	62.8	715	537	180	108
Schorr-Waite	19	1	2.9	8.7	120	57	766	311

Table 5. Comparison of the specifications generated by Norrish's C parser and our modifications to AutoCorres of 5 larger C programs. CPU time measurements are recorded on a 3.3GHz Intel Xeon E5-2643 with 128GiB of RAM.

to add additional support for abstracting code-level idioms used by a particular program.

4.6 Mixing low-level and high-level code

One of our original motivations for using C was its ability to access the heap at a low-level. Heap abstracted code, however, requires that memory is firmly tagged to being accessed only as a single type, preventing type-unsafe functions such as `memset` from being used.

Our solution is to allow the user to indicate which functions should be abstracted and which should remain in the low-level memory model. The former gain the benefits of simplified reasoning with heap abstraction, while the latter allows type-unsafe operations.

Calls from abstracted code to low-level code use the function $\mathsf{exec\_concrete}\ M$, where M is the low-level function to be executed. $\mathsf{exec\_concrete}$ non-deterministically selects a low-level state corresponding to the current high-level state, executes the monad M, and then translates the resulting low-level state back into an abstracted state. The following Hoare rule allows reasoning about calls to $\mathsf{exec\_concrete}$ (where st is the state translation function):

$$\frac{\{\!|\lambda s.\ P\ (\mathsf{st}\ s)|\!\}\ M\ \{\!|\lambda r\ s.\ Q\ r\ (\mathsf{st}\ s)|\!\}}{\{\!|P|\!\}\ \mathsf{exec\_concrete}\ M\ \{\!|Q|\!\}}$$

An analogous function $\mathsf{exec\_abstract}$ allows calls to abstracted functions from low-level code. With these constructs, we have proven properties such as:

$$\begin{array}{l}\{\!|\lambda s.\ \mathsf{is\_valid\_w32}\ s\ p|\!\}\\ \quad \mathsf{exec\_concrete}\ (\mathsf{memset}'\ p\ 0\ 4)\\ \{\!|\lambda\_\ s.\ \mathsf{is\_valid\_w32}\ s\ p \wedge s[p] = 0|\!\}\end{array}$$

This triple states that the call to the type-unsafe `memset` function, which sets 4 bytes at location p to 0, has the effect of performing an update on the abstracted `word32` heap. While proving this rule requires low-level reasoning, once in place, the user can simply reason about the abstract behaviour of `memset` without further low-level reasoning.

5. Case Studies

In this section, we give statistics on applying AutoCorres in larger projects, and examine two smaller examples to demonstrate that we indeed achieve abstraction and increased verification productivity.

5.1 Automatic abstraction in the large

While the examples shown in this paper are very simple, AutoCorres is designed to scale to non-trivial applications, and is being used in several ongoing verification projects. These projects are: (*i*) *CapDL SysInit*, a user-level program designed to bootstrap complex systems running on the seL4 microkernel; (*ii*) *eChronos*, a high-assurance real-time operating system for small micro-controllers without memory protection; (*iii*) *Piccolo*, a prototype separation kernel; and (*iv*) the *seL4 microkernel* [15], a formally verified operating system kernel. The first three of these projects are aiming to prove full functional verification of low-level C code from existing high-level specifications. AutoCorres is being used to reduce the effort of such proofs by minimising the gap between function specification and the code level. For the seL4 microkernel, AutoCorres is being trialled as a method of reducing proof maintenance costs when new functionality is added to the kernel.

Table 5 gives statistics for these projects, including number of functions and lines of code. We additionally provide the metrics *lines of specification* and *term size* for both the output of Norrish's C parser and AutoCorres. Since both tools directly emit terms in Isabelle/HOL's internal representation, we estimate the former number by using Isabelle/HOL's pretty printer for the generated definitions. The *term size* metric measures the number of nodes in the abstract syntax tree of a specification. While neither metric is a perfect match for specification complexity, the numbers reinforce our intuition that the output of AutoCorres is significantly simpler than that of the C parser with lines of spec ranging from 25% to 53% smaller and the term sizes ranging from 40% to 61% smaller. While AutoCorres has a longer running time than the C parser, for both tools this cost tends to be a one-off, where the results of the translation can be saved and reused. Further, AutoCorres translates functions in parallel, so real time is significantly less than CPU time.

We know of two further larger-scale projects that use AutoCorres in ongoing C verifications: the verification of a flash file system, and the verification of LEDA graph library checkers [22]. Statistics have not yet been obtained for these projects.

While there is clear interest in using the tool from independent verification projects, the numbers above can only give an indirect indication of effort reduction; more conclusive results on this scale will only be available when these longer-running projects are completed. In the meantime, to demonstrate that AutoCorres indeed achieves significant abstraction and reduces verification effort, we examine two concrete examples: in place-list reversal and the Schorr-Waite algorithm.

The Schorr-Waite algorithm, Richard Bornat famously argued, is "the first mountain that any formalism for pointer aliasing should climb." [5]. His advice has not gone unheeded, with many papers demonstrating new program verification techniques on the algorithm: Hubert and Marché, for instance, already verified a concrete C implementation using the Caduceus verification condition generator and the Coq theorem prover [13]. Earlier still, Mehta and Nipkow [18] verified the algorithm on a simple high-level imperative language in Isabelle/HOL, producing a readable machine-checked proof.

From our perspective, the latter proof is interesting because it verifies the Schorr-Waite algorithm on an idealised imperative language. The heap uses a Burstall-Bornat split heap memory model [5]; there is no concept of invalid pointers, such as unaligned pointers or unmapped memory, and the address space is infinite. All of these assumptions fail to hold on a low-level language such as C.

A useful benchmark, then, is to determine if we can implement this algorithm in standard C, automatically abstract the program with AutoCorres, and apply Mehta and Nipkow's existing proofs—

```
struct node *reverse(struct node *list) {
    struct node *rev = NULL;
    while (list) {
        struct node *next = list−>next;
        list−>next = rev; rev = list; list = next;
    }
    return rev;
}

 reverse' list ≡
    do (list, rev) ←
        whileLoop (λ(list, rev) s. list ≠ NULL)
          (λ(list, rev).
             do guard (λs. is_valid_node_C s list);
                next ← gets (λa. a[list]→next);
                modify (λs. s[list→next := rev]);
                return (next, list)
             od)
          (list, NULL);
       return rev
    od
```

Figure 6. ANSI C implementation of in-place linked list reversal, and its translation by AutoCorres.

written nearly a decade before AutoCorres was even conceived and carried out on a very abstract heap—to the result. We do not expect the proof to apply unchanged, but the goal is for any changes to be minimal.

5.2 In-place list reversal

Before we examine Schorr-Waite, we set the stage with the simpler example of in-place linked list reversal, which has become the *hello world* of pointer aliasing programs. The list-reversal function takes a singly-linked list, destructively modifies it so that the order of the nodes is reversed, and returns a pointer to the head of the new list.

Mehta and Nipkow used this example as an introductory exercise in their work [18], so it serves as a good example for demonstrating the differences between their program representations and those of AutoCorres. To carry out their proof, Mehta and Nipkow developed a simple while language. The language doesn't have a concept of a heap; instead, memory is modelled as global variables with the function type $'a\ ref \Rightarrow 'a$, where $'a\ ref$ can either be a Ref pointer or Null. These heaps are modified by assigning a new value to the variables, typically with only a single address updated. There is one such heap variable for each record field and type in the program.

For in-place list reversal, Mehta and Nipkow's algorithm and proof statement is as follows:

```
{List next p Ps}
 q := Null;
 WHILE p ≠ Null INV {...} DO
    t := p; p := next (addr p); next := next(t → q); q := t
 OD
{List next q (rev Ps)}
```

The predicate List $next\ p\ Ps$ indicates that there is a valid linked list in the heap $next$ starting from address p. The list Ps states the pointers of every node in the linked list, and has a definition equivalent to:

$$\begin{aligned}&\text{List}\ h\ p\ [] = (p = \text{Null})\\&\text{List}\ h\ p\ (x{\cdot}xs) = (p = \text{Ref}\ x \wedge \text{List}\ h\ (h\ x)\ xs)\end{aligned}$$

Their proof proceeds by building a library of theorems about the behaviour of the List predicate, providing an invariant for the while loop, running a VCG on the statement, and finally discharging the goals using built-in Isabelle/HOL tactics. The bulk of the work is in the library of list theorems, with just a few lines of main proof.

```
{R = reachable (relS {l, r}) {root} ∧ (∀x. ¬ m x) ∧ iR = r ∧ iL = l}
 t := root; p := Null;
 WHILE p ≠ Null ∨ t ≠ Null ∧ ¬ t^.m INV {...} DO
    IF t = Null ∨ t^.m THEN
       IF p^.c THEN
          q := t; t := p; p := p^.r; t^.r := q
       ELSE
          q := t; t := p^.r; p^.r := p^.l; p^.l := q; p^.c := True
       FI
    ELSE
       q := p; p := t; t := t^.l; p^.l := q; p^.m := True; p^.c := False
    FI
 OD
{(∀x. (x ∈ R) = m x) ∧ r = iR ∧ l = iL}
```

Figure 7. Mehta and Nipkow's correctness statement of the Schorr-Waite algorithm, reproduced from [18].

Fig 6 shows our C implementation of Mehta and Nipkow's list reversal, together with the corresponding output of AutoCorres. To port the proof, we had to resolve the following three differences:

(*i*) Valid addresses in the original proof are differentiated from the Null pointer using the $'a\ ref$ data type, while C uses the `NULL` sentinel value. This difference alone required tweaks in the majority of list definitions and proof statements; despite this, we could use the original proof scripts mostly unchanged.

(*ii*) In Mehta and Nipkow's proof there is no concept of an invalid heap access as there is in C. In contrast, the output of AutoCorres contains guard statements to ensure that pointer accesses are valid. We were able to adjust the definition of List to additionally assert that all elements in the list are valid pointers; this was enough to automatically discharge the guards in the final proof.

(*iii*) The original proof shows partial correctness, while AutoCorres requires total correctness for its refinement theorem to hold. We extended Mehta and Nipkow's proof to include a termination argument; in particular, we showed that the size of the list yet to be reversed is decreasing. This argument carries over to AutoCorres.

With these adjustments, we could complete the same main proof of correctness using the same loop invariant as Mehta and Nipkow. Overall, only minor effort was required to get a C-level correctness guarantee out of a high-level algorithmic proof.

5.3 Schorr-Waite algorithm

The Schorr-Waite algorithm [24] enumerates all nodes in a graph. While such a problem can be trivially solved using a stack linear in the size of the graph, the Schorr-Waite algorithm requires only two bits of storage per node. The algorithm achieves this by reversing pointers as it walks the graph so that it is able to backtrack in memory, and later restoring the pointers so that the original input graph is restored by the end of the algorithm. Because of its low memory requirements, the Schorr-Waite algorithm was originally proposed as the core element of a mark-and-sweep garbage collector; graph nodes not reached by the algorithm are no longer live, so can be reallocated. For brevity, we do not attempt to describe the algorithm in detail, but instead refer interested readers to one of the many descriptions available [13, 18, 24].

We base our C implementation shown in Fig 8 directly off the high-level imperative implementation of Mehta and Nipkow, reproduced in Fig 7. Each graph node contains two pointers `l` and `r` pointing to the *left* and *right* child respectively. Additionally, each graph node contains two bits. The *marked* bit `m` is set when a node has been visited by the algorithm. When the algorithm completes, all nodes reachable from the root will have their marked bit set. The *child* bit `c` is used to track which children of the current node have already been visited. Mehta and Nipkow's proof states that, assuming R is the set of nodes reachable from the root of the graph *root* and that no nodes are marked, then, after the algorithm finishes, all nodes in the set of addresses R will be marked, and the pointers of all nodes will match what they started as. The implementation uses the same while-language as in the previous example, but with the additional syntax $a\hat{}.f$ for accessing the heap f at location a and $a := f\hat{}.v$ for updating the heap f at location a to value v.

After translating our C implementation of Schorr-Waite using our extended version of AutoCorres, we reuse Mehta and Nipkow's existing proof script to verify the algorithm. This reuse presents the same set of differences as in the list reversal problem:

(*i*) Again, replacing *'a ref* with `NULL` sentinel C pointers causes no significant semantic changes to the proof, but does require updates to the base definitions.

(*ii*) Since the original model has no concept of invalid addresses, *it is not possible* to construct an invalid graph in Mehta and Nipkow's memory model, as every graph shape is supported by the algorithm. We have no such luxury, so we need to add a new precondition that all nodes in the set of reachable addresses R are valid and a new invariant repeating this fact.

(*iii*) Again, the original proof is a partial correctness result, while we desire total correctness. We modify Mehta and Nipkow's proof to include a new termination argument, requiring around 160 lines of new proof script. In particular, we annotate the main loop body with the measure used in Bornat's proof [5], and show that it decreases.

Overall, while several changes were required to the base definitions of Mehta and Nipkow's proof script to apply it to the output of AutoCorres, most of these changes were simple, while the main body of the original proof could be used unchanged. Table 6 shows the number of lines for our modified proof and for Mehta and Nipkow's original. The list definitions shared 48 lines (76%) while the main body of the proof shared 335 lines (66%). Of the remaining lines, the majority of changes were differences in program syntax and subtle differences in the output of the VCG tools.

We also include numbers from Hubert and Marché's Coq proof of a C implementation semantically equivalent to our implementation. Lines of proof script are not directly comparable between the two provers: Isabelle/HOL tends to provide more automation than Coq (often resulting in smaller proof scripts); however, Mehta and Nipkow's proof was intended to be highly readable (leading to a longer, more verbose proof script). Generously assuming Isabelle to be twice as productive as Coq, the size reduction compared to the previous C verification is striking.

Overall, in both examples we were able to apply an existing proof of an abstract algorithm almost directly to our automatically produced abstraction of a low-level C implementation. Even without taking into account that we proved stronger statements than the original, the size increase was moderate at most, and, more importantly, the proof complexity remained unchanged, with unchanged or only minimally adjusted invariants. This clearly shows that C verification can be performed from first principles at a comfortable level of abstraction.

```
void schorr_waite(struct node *root) {
  struct node *t = root, *p = NULL, *q;
  while (p != NULL || (t != NULL && !t->m)) {
    if (t == NULL || t->m) {
      if (p->c) {
        q = t; t = p; p = p->r; t->r = q;
      } else {
        q = t; t = p->r; p->r = p->l;
        p->l = q; p->c = 1;
      }
    } else {
      q = p; p = t; t = t->l; p->l = q;
      p->m = 1; p->c = 0;
    }
  }
}
```

Figure 8. Our implementation of the Schorr-Waite algorithm in C.

Component	*This Work*	M/N	H/M
List definitions	64	62	~ 900
Partial correctness	528	489	~ 1 400
Fault freedom	44	—	
Termination	160	—	~ 900
Miscellaneous	11	26	—
Total	807	577	3317

Table 6. A comparison of the lines of proof required for our work, for Mehta and Nipkow's proof [18] in Isabelle/HOL (*M/N*), and Hubert and Marché's proof [13] in Coq (*H/M*).

6. Related Work

Abstracting finite machine words into ideal integers is a standard technique for manual algorithm verification, formal or informal, which is famously [4, 29] easy to get wrong. Most modern C verification frameworks work directly with finite machine arithmetic, preferring the incurred cost over potential unsoundness. Of the large source code verifications to date, the Verisoft project [1] abstracted finite machine types to ideal integers in a manual proof at the ISA level, inducing similar guards to ours on the source code level. In comparison, our abstraction is fully automatic and can be selective.

As mentioned, our heap abstraction approach builds on Tuch et al.'s [27] base memory model, bypassing their shallow version of heap lifting, but retaining the foundational aspect of deriving all higher-level concepts from first principle. Later work indicated significant manual effort in the use of Tuch's model at larger scale [30]. Our main contribution over Tuch is to break the barrier to larger-scale application and provide a scalable, fully automatic, and easy to use abstraction on top of this foundational model. We inherit the ability to soundly drop down to byte-level reasoning for type-unsafe code.

Other memory models include reasoning on the low level only, for instance the C compiler verification in CompCert [17]. Similarly, Ellison and Roşu [9] define a highly complete and validated C semantics with a memory model that is essentially a map to blocks of bytes. This is closer to the C standard than our base model, but verification relying on the standard alone is insufficient, since it is routinely violated on purpose in systems code [15]. Our model makes explicit compiler and architecture assumptions, and we use binary translation validation [25] to prove that the compiler correctly implements our model even when the code violates the standard.

Most C verification frameworks strive for abstraction and some aim to consolidate low- and high-level views. Usually this is not done foundationally, but axiomatically. Moy's translation of C into the FramaC framework with the Jessie-plugin for instance [19], provides an axiomatic typed-heap memory model with limited support for pointer casts. The SMT-based VCC tool [8], originally with an untyped memory model, also supports a typed semantics, but the abstraction is justified by a pen and paper argument only.

Other approaches use separation logic to reason about C memory [2], which can be defined directly on a low-level memory view. The program logic used to reason on C is largely orthogonal to the memory model. For instance, it is trivial to instantiate our abstracted heaps into existing separation logic frameworks [16].

Chlipala [6] proposes manually-guided proof-producing abstraction directly from machine code. Our work aims for full automation. The approaches are not incompatible: one could imagine a fully automated abstraction in our style as far as it carries and continuing manually in Chlipala's style from there on.

To summarise, the key differentiator of our heap abstraction work is that it allows reasoning (*i*) at *both* a typed-heap model and byte level, (*ii*) that the lifting is *automatic*, (*iii*) that the tool supports *any* C Standard retyping/casting operations, and (*iv*) provides an LCF-style foundational proof.

7. Conclusion

We have presented two automatic, proof-producing abstractions that significantly ease the interactive formal verification of C programs. Both are implemented in the AutoCorres tool, on top of an existing C verification framework with a proof chain down to binary code [25].

We have demonstrated in two case studies how the verification of low-level C code can now proceed on the same level of abstraction as previous verifications of idealised algorithms.

The automated abstractions provide an obvious reduction of cognitive complexity for the verification engineer, with an associated increase in human productivity and potential for further mechanical automation. This is not the only benefit of the approach. Compared to just a better VCG with more automation for a specific logic, the automated abstraction approach is agnostic about the logic and the kind of verification to be conducted. For instance, it is easy to use a separation logic on top, to connect to a VCG, to make our output an intermediate step in a larger refinement proof, or to use it as a base-level model for proving non-interference in the style of Murray et al [20]. These are diverse logics that can be chosen freely depending on the application area. A purely VCG-based approach would have to decide up front to support a specific one.

AutoCorres is currently being used in a number of larger C verification projects at different institutions, including the verification of a complex large-scale graph library, the verification of a file system, and the verification of a real-time operating system for high-assurance systems.

Finally, the AutoCorres tool described in this paper is available under an open-source BSD-style license at [10].

Acknowledgments

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In *VSTTE 2008*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.

[2] A. Appel. Verified software toolchain. In *20th ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.

[3] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *2010 FMCAD*, pages 35–42, Lugano, Switzerland, 2010. FMCAD Inc.

[4] J. Bloch. Nearly all binary searches and mergesorts are broken. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`, Jun 2006.

[5] R. Bornat. Proving pointer programs in Hoare Logic. In *5th MPC*, volume 1837 of *LNCS*, pages 102–126. Springer, Jul 2000.

[6] A. Chlipala. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *18th ICFP*, pages 391–402. ACM, 2013.

[7] D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.

[8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer.

[9] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *39th POPL*, pages 533–544. ACM, 2012.

[10] D. Greenaway. Autocorres tool. `http://ssrg.nicta.com.au/projects/TS/autocorres/`, 2014.

[11] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *3rd ITP*, volume 7406 of *LNCS*, pages 99–115, Princeton, New Jersey, Aug 2012. Springer.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *10th SPIN*, volume 2648 of *LNCS*, pages 235–239, Portland, OR, USA, May 2003. Springer.

[13] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *3rd Int. Conf. Softw. Engin. & Formal Methods*, pages 190–199. IEEE, Sep 2005.

[14] ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.

[15] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[16] G. Klein, R. Kolanski, and A. Boyton. Mechanised separation algebra. In *3rd ITP*, volume 7406 of *LNCS*, pages 332–337. Springer, 2012.

[17] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7): 107–115, 2009.

[18] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *19th CADE*, volume 2741 of *LNCS*, pages 121–135, 2003.

[19] Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Paris, France, Jan 2009.

[20] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, pages 415–429, San Francisco, CA, May 2013.

[21] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[22] L. Noschinski, C. Rizkallah, and K. Mehlhorn. Verification of certifying computations through AutoCorres and Simpl. In *NASA Formal Methods*, volume 8430 of *LNCS*. Springer, 2014. To appear.

[23] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[24] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *CACM*, 10 (8):501–506, 1967.

[25] T. Sewell, M. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *2013 PLDI*, pages 471–481. ACM, 2013.

[26] N. Suzuki. *Automatic Verification of Programs with Complex Data Structures*. Garland Publishing, New York, 1980.

[27] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *34th POPL*, pages 97–108, Nice, France, Jan 2007. ACM.

[28] D. von Oheimb and T. Nipkow. Machine-checking the java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 119–156. Springer, 1999.

[29] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: what happened to my code? In *3rd APSys*, pages 9:1–9:7, New York, NY, USA, 2012. ACM.

[30] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In *22nd TPHOLs*, volume 5674 of *LNCS*, pages 500–515. Springer, 2009.