Bridging the Gap: Automatic Verified Abstraction of C

David Greenaway[1,2], June Andronick[1,2], and Gerwin Klein[1,2]

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{first-name.last-name}@nicta.com.au

Abstract. Before low-level imperative code can be reasoned about in an interactive theorem prover, it must first be converted into a logical representation in that theorem prover. Accurate translations of such code should be conservative, choosing safe representations over representations convenient to reason about. This paper bridges the gap between conservative representation and convenient reasoning. We present a tool that automatically abstracts low-level C semantics into higher level specifications, while generating proofs of refinement in Isabelle/HOL for each translation step. The aim is to generate a verified, human-readable specification, convenient for further reasoning.

1 Introduction

Low-level imperative C is still the most widely used language for developing software with high performance and precise memory requirements, especially in embedded and critical high-assurance systems. The challenge of formally verifying C programs has been attacked with approaches ranging from static analysis for eliminating certain runtime errors to full functional correctness with respect to a high-level specification. This paper addresses the latter by improving automation in the verification process while preserving the strength of the correctness proof.

The first step required to formally reason about a program is to parse the code into a formal logic. The parser is necessarily trusted, giving rise to two approaches: either the parser is kept simple, minimising the assumption we make about its correctness, but resulting in a low-level formal model; or the parser generates a specification that is more pleasant to reason about, but resulting in a weaker trust chain.

We advocate the first approach, increasing the level of trustworthiness of the final proof. In this context, existing solutions either bear the burden of working with the low level C semantics, as for instance in Verisoft [2], or manually bridge

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

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

int gcd(int a, int b) {
  int c;
  while (a != 0) {
    c = a;
    a = b % a;
    b = c;
  }
  return b;
}
```

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

gcd a b ≡
  do
    (a, b) ← while (λ(a, b) s. a ≠ 0)
        (λ(a, b). return (b mod a, a))
        (a, b);
    return b
  od
```

Fig. 1. C functions `max` and `gcd` and their corresponding abstractions.

the gap by abstracting the low-level specification to an intermediate functional specification as in the L4.verified project [7], which showed correctness of the seL4 microkernel.

The contribution of this paper is a new tool[1] that automatically abstracts low-level C semantics into higher level specifications, while generating proofs in Isabelle/HOL for each translation step. The aim is to generate a human-readable specification that is easier and more convenient to reason about than the original code. Simpler specifications are more amenable to proving further high-level properties: instead of 25 person years reasoning on the code level, establishing integrity and authority confinement for seL4 merely took 10 person months, because it could be proved about the much simpler, abstract specification instead (with the functional correctness proof guaranteeing that it is then true down to the C code level). One third of these 25 person years were dedicated to refinement proofs of the form we envision our tool to eventually automate. The novelty here lies in providing both automated abstraction and correctness proofs.

As a running example, we will consider two simple functions, computing the maximum and the greatest common divisor respectively of two numbers. The C implementation of these two functions is given in Fig 1 on the left and the translation output of our tool on the right. In comparison, Fig 2 shows the output of the L4.verified C parser by Norrish [14,11] in the Simpl language [12] embedded in Isabelle/HOL [10]. The output of this parser is the starting point of our translation.

Compared to the C parser output, the result of our tool is significantly simpler and more abstract. The raw parser output is so complex because the C semantics have to deal with abrupt termination (e.g., `return` statements), with ensuring the C standard is obeyed (*guard* statements), with non-terminating loops, etc. Modelling these conservatively and precisely with a minimal trusted computing base induces overhead. Our tool aims to automatically distill the interesting semantic content without sacrificing trust.

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

```
TRY
  IF {| ´a <s ´b |} THEN
    ´ret-int :== ´b;
    ´exn-var :== Return;
    THROW
  ELSE
    SKIP
  FI;
  ´ret-int :== ´a;
  ´exn-var :== Return;
  THROW;
  GUARD DontReach ∅
    SKIP
CATCH
  SKIP
END
```

(a) `max` Simpl Translation

```
TRY
  NonDetInit c-′ c-′-update;
  WHILE {| ´a ≠ 0 |} DO
    ´c :== ´a;
    GUARD Div-0 {| ´a ≠ 0 |}
      ´a :== ´b mod ´a;
    ´b :== ´c
  OD;
  ´ret-int :== ´b;
  ´exn-var :== Return;
  THROW;
  GUARD DontReach ∅
    SKIP
CATCH
  SKIP
END
```

(b) `gcd` Simpl Translation

Fig. 2. The C functions from Fig 1 parsed into Simpl.

While the above are toy examples for presentation, the tool is not: it successfully translates, for instance, the seL4 microkernel with ca. 8 700 lines of code, a malloc-style allocator, and a real-time operating system task scheduler. Where insightful, we will mention results of applying the tool to these code bases.

Current limitations of the tool are: recursion is not supported, and a limited number of features of the C language are not supported, most notably taking the address of local variables. The first limitation is planned for future work, while the second limitation stems from the C parser front-end.

In the following, Sec 2 describes the supported C subset, the input language Simpl and the monadic framework the tool is working in. Sec 3 presents the core of the tool by explaining the translations in the abstraction process and their proofs, while Sec 3.7 describes the final theorem between tool input and output.

2 Background

2.1 Parsing C

Before code can be reasoned about, it must first be translated into the theorem prover. In this work, we consider programs in C99 [6] translated into Isabelle/HOL using Norrish's C parser [14,11]. This parser supports a subset of C, including loops, function calls, type casting, pointer arithmetic and structures. Integer arithmetic is defined to match a two's-complement 32-bit system. The parser emits inline *guards* to ensure that undefined operations, such as divide-by-zero or signed integer overflow, do not occur. As the parser must be trusted, it attempts to be simple, giving the most literal translation of C wherever possible.

The parser does not support `goto` statements, expressions with side-effects, references to local variables, `switch` statements using fall-through, unions, floating point arithmetic, or calls to function pointers. Finally, while the parser does support recursion, our tool does not yet handle such inputs. Our tool remains useful despite these limitations as embedded and systems code is often stack depth-constrained and typically avoids recursion.

2.2 Simpl

The parser translates C source code into Schirmer's *Simpl* language [12] embedded in Isabelle/HOL. Simpl is a generic imperative language with deeply embedded statements and shallowly embedded expressions, designed to be a target for embedding programs in a variety of languages, such as C, Java and Ada.

The Simpl language consists of 11 commands. The commands of interest are:

$$c \equiv \mathsf{SKIP} \mid \mathsf{BASIC}\ m \mid c_1\ ;\ c_2 \mid \mathsf{IF}\ e\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \mathsf{FI} \mid \mathsf{WHILE}\ e\ \mathsf{DO}\ c\ \mathsf{OD} \\ \mid \mathsf{TRY}\ c_1\ \mathsf{CATCH}\ c_2\ \mathsf{END} \mid \mathsf{THROW} \mid \mathsf{CALL}\ f \mid \mathsf{GUARD}\ F\ P\ c \mid \mathsf{SPEC}\ r$$

The statement $\mathsf{BASIC}\ m$ modifies the state by applying function m to it; in the common case where m is a function that updates a variable a to the value b, we use the notation $\acute{}\mathsf{a} :== b$. $\mathsf{GUARD}\ F\ P\ c$ asserts property P before executing c, otherwise aborting execution with fault F. $\mathsf{SPEC}\ r$ non-deterministically selects a new state s' based on the current state s such that $(s, s') \in r$ holds; we use such non-determinism to model hardware and uninitialised memory.

Fig 2(a) shows an example of a simple C function `max` parsed into Simpl. Input parameters a and b are set up by the caller and otherwise treated as local variables, while the return value of the function is recorded in the ghost variable $\mathsf{ret\text{-}int}$. The function body is surrounded by an exception handler with the empty SKIP body; this pattern is used to model abrupt termination as in `return`, `break` and `continue`. The ghost variable $\mathsf{exn\text{-}var}$ records the reason for the current exception, so that, for instance, `return` statements inside loops are not handled by the `break` handler surrounding the loop. Finally, the GUARD command rules out particular undefined behaviour in C; in this case asserting that execution does not fall off the end of the (non-`void`) function. Fig 2(b) is similar in structure, but additionally initialises the variable c to a non-deterministically chosen value.

All Simpl programs execute on a particular state type. In our case it always contains a record with local variables and a record with global variables, among them the heap, a partial function mapping addresses in memory to their byte values. We use Schirmer's notation $\Gamma \vdash \langle C, \mathsf{Normal}\ s\rangle \Rightarrow \mathsf{Normal}\ t$ to specify that the Simpl program C starting in state $\mathsf{Normal}\ s$ has at least one execution path resulting in $\mathsf{Normal}\ t$. Other state types include $\mathsf{Abrupt}\ s$, indicating the program is currently propagating an exception; $\mathsf{Fault}\ f$, indicating an irrecoverable failure f; or Stuck, indicating stuck execution. The variable Γ maps function names to function bodies, and is used for making function calls in Simpl. We additionally use Schirmer's notation $\Gamma \vdash C \downarrow \mathsf{Normal}\ s$ to specify that all execution paths of the program C starting in state $\mathsf{Normal}\ s$ terminate.

Simpl	Monad	Monadic Definition
–	returnE x	$\lambda s.$ ({(Norm x, s)}, False)
SKIP	skipE	$\lambda s.$ ({(Norm (), s)}, False)
BASIC m	modifyE m	$\lambda s.$ ({(Norm (), m s)}, False)
THROW	throwE ()	$\lambda s.$ ({(Exc (), s)}, False)
IF c THEN L ELSE R FI	condE c L R	$\lambda s.$ if c s then L s else R s
GUARD t g B	guardE g	condE g skipE failE
WHILE c DO B OD	whileE c B ()	*(see text)*

Fig. 3. A selection of monadic functions with corresponding Simpl commands.

2.3 Monadic Framework

Our goal is to abstract imperative programs encoded in Simpl into a representation that eases reasoning. But which representation is best suited to such reasoning? Any representation we choose must encode the same functionality as Simpl, including programs that read and write global state; contain loops that potentially do not terminate; raise and catch exceptions; are non-deterministic; and have execution paths that result in irrecoverable failure.

Our chosen representation is a *state monad* with additional support for non-determinism, exceptions and failure (representing irrecoverable program failure). We name this monad the *exception monad* which has type $'s \Rightarrow (('e + 'a) \times 's)$ *set* $\times$ *bool* abbreviated as $('s, 'a, 'e)$ *monadE*. The monad accepts a single input state $'s$ and returns a tuple. The first half of this tuple contains the results of the execution: a set of pairs containing a return value and state. The result is a set so that functions may return more than one resulting state, modelling non-determinism. Each return value is either a standard value of type $'a$ indicating normal execution, or an exception value of type $'e$. The second half of the tuple is a flag indicating whether any execution of the monad failed. We name the first and second halves of this tuple results and failed respectively. A full description of the motivation for and formalisation of this monad with VCG support are presented in earlier work [3].

Fig 3 lists the monadic commands used in this paper and their Simpl equivalents where applicable. Monadic commands are suffixed with the character E to indicate they operate on the exception monad. Monadic functions may be joined together by the *bind* operator, where $a \gg=_{\mathsf{E}} (\lambda x.\ b\ x)$ denotes that a is executed with its return value passed into b, bound to the variable x. We additionally use the notation doE $x \leftarrow\ a$; $b\ x$ odE as alternative syntax for bind. returnE f simply returns the value f, allowing it to be used later as a bound variable.

To represent loops, we define a combinator whileE $c\ B\ i$ with type:

$$('a \Rightarrow 's \Rightarrow \mathit{bool}) \Rightarrow ('a \Rightarrow ('s, 'a, 'e)\ \mathit{monadE}) \Rightarrow 'a \Rightarrow ('s, 'a, 'e)\ \mathit{monadE}$$

The combinator takes a loop condition c, a loop body B, and an initial *loop iterator* value i. While the condition c remains true, the loop body will be executed with the current loop iterator value. The return value from each iteration of the

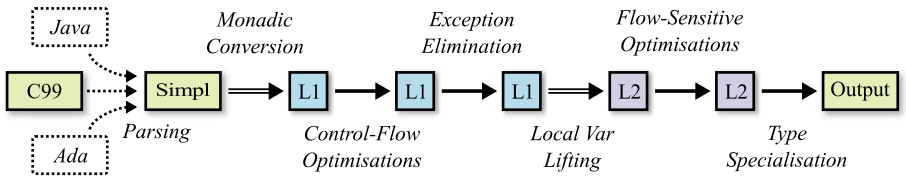

Fig. 4. The process of converting Simpl to an abstracted output program. Dashed arrows represent trusted translations, white arrows represent refinement proofs, while solid arrows represent term rewriting. Each phase beyond parsing is in Isabelle/HOL.

loop body will become the loop iterator value for the next loop iteration, or the return value for the whileE block if the loop condition is false. This allows us to bind variables in one iteration of the loop and use them in either the next iteration of the loop or after the loop completes. Formally, whileE returns the set of all results that can be reached in a finite number of loop iterations. Additionally, whileE fails if any computation within the loop fails or if there exists any non-terminating computation.

3 Abstraction Process

This section describes in detail the transformations that take place from our input Simpl specification to the output specification presented to the user, as well as the proofs of correctness generated at each step. Fig 4 depicts the transformations applied, each of which is described below.

3.1 Conversion to Shallow Embedding

When C code is parsed into Isabelle/HOL, it is converted into the Simpl language with deeply-embedded statements. While such a deep embedding is sufficient for reasoning about program behaviour, in practice it is a frustrating experience: standard Isabelle mechanisms such as term rewriting, which can replace sub-terms of a program with equivalent alternatives, cannot be used, as two semantically equivalent programs are only considered equal if they are *structurally* identical. While tools can be developed to alleviate some of this burden [15], still much of the support provided by Isabelle remains unavailable.

Our first step towards generating an abstraction is thus converting the deeply-embedded Simpl input into a monadic shallow embedding. We name the output of this translation stage *L1*.

The conversion process is conceptually easy: Simpl constructs are simply substituted with their monadic equivalents shown in Fig 3. Our goal, however, is to also generate a proof that the conversion is sound. We achieve this by proving a property $\mathsf{corres}_{\mathsf{L1}}$ stating that the original Simpl program is a refinement of our translated program, defined as follows:

$$\mathsf{corres_{L1}}\ \Gamma\ (\mathsf{modifyE}\ m)\ (\mathsf{BASIC}\ m) \qquad \text{L1CorresSkip}$$

$$\frac{\mathsf{corres_{L1}}\ \Gamma\ L\ L' \qquad \mathsf{corres_{L1}}\ \Gamma\ R\ R'}{\mathsf{corres_{L1}}\ \Gamma\ (L \gg\!=_{\mathsf{E}} (\lambda y.\ R))\ L';\ R'} \qquad \text{L1CorresSeq}$$

$$\frac{\mathsf{corres_{L1}}\ \Gamma\ L\ L' \qquad \mathsf{corres_{L1}}\ \Gamma\ R\ R'}{\mathsf{corres_{L1}}\ \Gamma\ (\mathsf{condE}\ c\ L\ R)\ (\mathsf{IF}\ c\ \mathsf{THEN}\ L'\ \mathsf{ELSE}\ R'\ \mathsf{FI})} \qquad \text{L1CorresCond}$$

$$\frac{\mathsf{corres_{L1}}\ \Gamma\ B\ B'}{\mathsf{corres_{L1}}\ \Gamma\ (\mathsf{whileE}\ (\lambda\text{-}.\ c)\ (\lambda\text{-}.\ B)\ ())\ (\mathsf{WHILE}\ c\ \mathsf{DO}\ B'\ \mathsf{OD})} \qquad \text{L1CorresWhile}$$

Fig. 5. Selection of rules, compositionally proving $\mathsf{corres_{L1}}$ in the Simpl to L1 translation.

$$\begin{array}{l}
\mathsf{corres_{L1}}\ \Gamma\ A\ C \equiv \\
\quad \forall s.\ \neg\ \mathsf{failed}\ (A\ s) \longrightarrow \\
\qquad (\forall t.\ \Gamma \vdash \langle C, \mathsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\qquad\quad (\mathsf{case}\ t\ \mathsf{of}\ \mathsf{Normal}\ s' \Rightarrow (\mathsf{Norm}\ (), s') \in \mathsf{results}\ (A\ s) \\
\qquad\quad\ \mid \mathsf{Abrupt}\ s' \Rightarrow (\mathsf{Exc}\ (), s') \in \mathsf{results}\ (A\ s)\ \mid \text{-} \Rightarrow \mathsf{False})) \wedge \\
\qquad \Gamma \vdash C \downarrow \mathsf{Normal}\ s
\end{array}$$

The definition reads as follows: Given a Simpl context Γ mapping function names to function bodies, a monadic program A and a Simpl program C, then, assuming that the monadic program A does not fail: (i) for each normal execution of the Simpl program there is an equivalent normal execution of the monadic program; (ii) similarly, for each execution of the Simpl program that results in an exception, there is an equivalent monadic execution also raising an exception; and, finally (iii) every execution of the Simpl program terminates.

The final termination condition may initially seem surprising. Recall, however, that these conditions must only hold if A does not fail, while our definition of whileE ensures that infinite loops will raise the failure flag. Consequently, proving termination of C is reduced to proving non-failure of A.

We prove $\mathsf{corres_{L1}}$ automatically using a set of syntax-directed rules such as those listed in Fig 5. The final L1 output is a program that has the same structure as the source Simpl program, but is in a more convenient representation.

3.2 Control Flow Peephole Optimisations

The Simpl output generated by the C parser is, by design, as literal a conversion of C as possible. This frequently leads to clutter such as: (i) unnecessary skipE statements, generated from stray semicolons (which remain after the preprocessor strips away debugging code); (ii) guardE statements that are always true; (iii) dead code following throwE or failing guardE statements; or (iv) conditional condE statements where the condition is True or False. As the L1 specification is a shallow embedding, we are able to use Isabelle's rewrite engine to apply a series of *peephole optimisations* consisting of 21 rewrite rules, removing significant amounts of unnecessary code from the L1 programs. Table 1 at the end of this paper measures the size reduction in each translation stage.

$$\frac{\mathsf{no\text{-}throw}\ A}{\mathsf{catchE}\ A\ E = A}\ \textsc{CatchNoThrow} \qquad \frac{\mathsf{always\text{-}throw}\ A}{A \gg\!=_{\mathsf{E}} B = A}\ \textsc{SeqAlwaysThrow} \qquad \mathsf{catchE}\ (\mathsf{throwE}\ a)\ E = E\ a\ \textsc{CatchThrow}$$

$$\frac{\mathsf{no\text{-}throw}\ A}{\mathsf{catchE}\ (A \gg\!=_{\mathsf{E}} B)\ C = A \gg\!=_{\mathsf{E}} (\lambda x.\ \mathsf{catchE}\ (B\ x)\ C)}\ \textsc{SeqNoThrow}$$

$$\mathsf{catchE}\ (\mathsf{condE}\ c\ L\ R)\ E = \mathsf{condE}\ c\ (\mathsf{catchE}\ L\ E)\ (\mathsf{catchE}\ R\ E)\ \textsc{CatchCond}$$

$$\mathsf{catchE}\ (\mathsf{condE}\ C\ L\ R \gg\!=_{\mathsf{E}} B)\ E = \mathsf{condE}\ C\ (\mathsf{catchE}\ (L \gg\!=_{\mathsf{E}} B)\ E)\ (\mathsf{catchE}\ (R \gg\!=_{\mathsf{E}} B)\ E)\ \textsc{CatchCondSeq}$$

Fig. 6. Rewrite rules to reduce exceptions in control flow

3.3 Exception Rewriting

Statements in C that cause abrupt termination such as `return`, `continue` or `break` are modelled in Simpl with exceptions, as described in Sec 2.2. While exceptions accurately model the behaviour of abrupt termination, their presence complicates reasoning about the final program: each block of code now has *two* exit paths that must be considered.

Fortunately, most function bodies can be rewritten to avoid the use of exceptions. Fig 6 shows the set of rewrite rules we use to reduce exceptional control flow. CatchNoThrow eliminates exception handlers surrounding code that never raises exceptions (denoted by no-throw). Analogously, SeqAlwaysThrow removes code trailing a block that *always* raises an exception (denoted by always-throw). The no-throw and always-throw side-conditions are proved automatically using a syntax-directed set of rules.

Not all rules in this set can be applied blindly. In particular, the rules CatchCond and CatchCondSeq duplicate blocks of code, which may trigger exponential growth in pathological cases. For CatchCond, which duplicates the exception handler, knowledge of our problem domain saves us: inputs originating from C only have trivial exception handlers generated by the parser, and hence duplicating them is of no concern.

The rule CatchCondSeq, however, also duplicates its tail B, which may be arbitrarily large. We carry out the following steps to avoid duplication: (*i*) if neither branch of the condition throws an exception, then SeqNoThrow is applied; (*ii*) if both branches throw an exception, then SeqAlwaysThrow is applied; (*iii*) if one branch always throws an exception, then the rule CatchCondSeq is applied followed by SeqAlwaysThrow on that branch, resulting in only a single instance of B in the output; finally (*iv*) if the body B is trivial, such as a simple returnE or throwE statement, we apply CatchCondSeq and duplicate B under the assumption the rewritten specification will still be simpler than the original. Otherwise, we leave the specification unchanged, and let the user reason about the exception rather than a larger output specification.

```
doE
  modifyE (λs. s(| a-' := 3 |));
  condE (λs. 5 ≤ a-' s)
     (modifyE (λs. s(| b-' := 5 |)))
     (modifyE (λs. s(| c-' := 4 |)));
  modifyE (λs. s(| ret-int-' := a-' s |))
odE
```

(a) Locals in state

```
doE
  a ← returnE 3;
  (b, c) ← condE (λs. 5 ≤ a)
     (returnE (5, c))
     (returnE (b, 4));
  returnE a
odE
```

(b) Local lifted form

Fig. 7. Two program listings. The first stores locals in the state, while the second uses bound variables. The shaded region does not affect the final return value; this is clearly apparent in the second representation.

Using these rules, all exceptions can be eliminated other than those in nested condition blocks described above, or those caused by `break` or `return` statements inside loop bodies. Applying the transformation to the seL4 microkernel, 96% of functions could be rewritten to eliminate exceptional control flow. Of the remaining 4%, 10 could not be rewritten due to nested condition blocks, 13 because of either `return` or `break` statements inside a loop, and one function for both reasons independently.

3.4 Local Variable Lifting

Both the Simpl embedding of our original input programs and our L1 translation represent local variables as part of the state: each time a local is read it is extracted from the state, and each time a local is written the state is modified. While this representation is easy to generate, it complicates reasoning about variable usage. An example of this is shown in Fig 7(a): the variable a is set to the value 3 at the top of the function and later returned by the function. However, to prove that the function returns the value 3, the user must first prove that the shaded part of the program preserves a's value.

An alternative approach to representing locals is using the bound variables feature provided by our monadic framework that we have so far ignored. To achieve this, we remove locals from the state type and instead model them as bound Isabelle/HOL variables. We name this representation *lifted local form* and the output of this translation *L2*. The representation is analogous to *static single-assignment* (SSA) form used by many compilers as an intermediate representation [9], where each variable is assigned precisely once.

Fig 7(b) shows the same program in lifted local form. The function returns the variable a, which is bound to the value 3 in the first line of the function. As variables cannot change once bound, the user can trivially determine that the function returns 3 without inspecting the shaded area.

Two complications arise in representing programs in local lifted form. The first is that variables bound inside the bodies of condE and catchE blocks are not available to statements after the block. To overcome this, we modify the

bodies of such blocks to return a tuple of all variables modified in the bodies and subsequently referenced, as demonstrated in Fig 7(b); statements following the block can then use the names returned in this tuple. The second, similar complication arises from loops, where locals bound in one iteration not only need to be accessible after the loop, but also accessible by statements in the *next* iteration. We solve this by passing all required locals between successive iterations of the loop as well as the result of the loop in the iterator of the whileE combinator. The `gcd` function in Fig 1 shows an example. In both cases, the tool must perform program analysis to determine which variables are modified. The emitted proofs imply correctness of this analysis as we shall see below.

For the soundness proof of the translation from L1 to L2 we use a refinement property $\mathsf{corres}_{\mathsf{L2}}$ defined as follows:

$$
\begin{array}{l}
\mathsf{corres}_{\mathsf{L2}}\ st\ rx\ ex\ P\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ (st\ s)) \longrightarrow \\
\qquad (\forall (r,\ t) \in \mathsf{results}\ (C\ s). \\
\qquad\quad \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Exc}\ () \Rightarrow (\mathsf{Exc}\ (ex\ t),\ st\ t) \in \mathsf{results}\ (A\ (st\ s)) \\
\qquad\quad \mid \mathsf{Norm}\ () \Rightarrow (\mathsf{Norm}\ (rx\ t),\ st\ t) \in \mathsf{results}\ (A\ (st\ s))) \wedge \\
\qquad \neg\ \mathsf{failed}\ (C\ s)
\end{array}
$$

The predicate has several parameters: st is a state translation function, converting the L1 state type to the L2 state type by stripping away local variable data; P is a precondition used to ensure that input bound variables in the L2 program match their L1 values; and A and C are the abstract L2 and concrete L1 programs respectively. The values rx and ex are a *return extraction function* and an *exception extraction function* respectively; they are required because the L2 monads return or throw variables, while the corresponding L1 monads store these values in their state. The return extraction function rx extracts a value out of the L1 state to compare with the return value of the L2 monad, while ex is used to compare an exception's payload with the corresponding L1 state.

The $\mathsf{corres}_{\mathsf{L2}}$ definition can be read as: for all states matching the precondition P, assuming that A executing from state $st\ s$ does not fail, then the following holds: (i) for each normal execution of C there is an equivalent execution of A whose return value will match the value extracted using rx from C's state; (ii) similarly, every exceptional execution of the C will have an equivalent execution of A with an exception value that matches the value extracted using ex from C's state; and, finally (iii) the execution of C will not fail.

The first two conditions ensure that executions in L2 match those of L1 with locals bound accordingly. The last condition allows us to later reduce non-failure of L1 programs to non-failure of L2 programs.

As a concrete example, Fig 9 shows our example `max` function after local variable lifting has taken place. The generated $\mathsf{corres}_{\mathsf{L2}}$ predicate for `max` is:

$$\mathsf{corres}_{\mathsf{L2}}\ \mathsf{globals}\ \mathsf{ret\text{-}int\text{-}'}\ (\lambda s.\ ())\ (\lambda s.\ \mathsf{a\text{-}'}\ s = a \wedge \mathsf{b\text{-}'}\ s = b)\ (\mathsf{max}_{\mathsf{L2}}\ a\ b)\ \mathsf{max}_{\mathsf{L1}}$$

In this example the state translation function globals strips away local variables from the L1 state; the return extraction function rx ensures the value returned by $\mathsf{max}_{\mathsf{L2}}$ matches the variable $\mathsf{ret\text{-}int\text{-}'}$ of $\mathsf{max}_{\mathsf{L1}}$, while the exception extraction

$$\frac{\forall s.\ P\ s \longrightarrow st\ s = st\ (M'\ s) \qquad \forall s.\ P\ s \longrightarrow rx\ (M'\ s) = v}{\mathsf{corres_{L2}}\ st\ rx\ ex\ P\ (\mathsf{returnE}\ v)\ (\mathsf{modifyE}\ M')} \quad \textsc{L2CorresReturn}$$

$$\frac{\forall s.\ P\ s \longrightarrow M\ (st\ s) = st\ (M'\ s)}{\mathsf{corres_{L2}}\ st\ rx\ ex\ P\ (\mathsf{modifyE}\ M)\ (\mathsf{modifyE}\ M')} \quad \textsc{L2CorresModify}$$

$$\frac{\begin{array}{c}\mathsf{corres_{L2}}\ st\ rx\ ex\ Q_A\ A\ A' \qquad \forall x.\ \mathsf{corres_{L2}}\ st\ rx'\ ex\ (Q_B\ x)\ (B\ x)\ B' \\ \{\!|P|\!\}\ A'\ \{\!|\lambda\text{-}\ s.\ Q_B\ (rx\ s)\ s|\!\}, \{\!|\lambda\text{-}\ \text{-}.\ \mathsf{True}|\!\} \qquad \forall s.\ P\ s \longrightarrow Q_A\ s\end{array}}{\mathsf{corres_{L2}}\ st\ rx'\ ex\ P\ (A \gg\!=_{\mathsf{E}} B)\ (A' \gg\!=_{\mathsf{E}} (\lambda x.\ B'))}$$
$\textsc{L2CorresSeq}$

$$\frac{\begin{array}{c}\forall x.\ \mathsf{corres_{L2}}\ st\ rx\ ex\ (Q'\ x)\ (A\ x)\ B \\ \{\!|\lambda s.\ Q\ (rx\ s)\ s|\!\}\ B\ \{\!|\lambda\text{-}\ s.\ Q\ (rx\ s)\ s|\!\}, \{\!|\lambda\text{-}\ \text{-}.\ \mathsf{True}|\!\} \\ \forall s.\ Q\ (rx\ s)\ s \longrightarrow c'\ s = c\ (rx\ s)\ (st\ s) \\ \forall s\ x.\ Q\ x\ s \longrightarrow Q'\ x\ s \qquad \forall s.\ Q\ x\ s \longrightarrow rx\ s = x \qquad \forall s.\ P\ x\ s \longrightarrow Q\ x\ s\end{array}}{\mathsf{corres_{L2}}\ st\ rx\ ex\ (P\ x)\ (\mathsf{whileE}\ c\ A\ x)\ (\mathsf{whileE}\ (\lambda\text{-}.\ c')\ (\lambda\text{-}.\ B)\ ())}$$
$\textsc{L2CorresWhile}$

Fig. 8. Selected rules used in the $\mathsf{corres_{L2}}$ proofs.

function ex is unused and simply returns unit, as no exceptions are thrown by the `max` function. The remainder of the predicate states that, assuming the inputs a and b to our $\mathsf{max_{L2}}$ function match those of the L1 state, then the return value of our $\mathsf{max_{L2}}$ function will match the L1 state variable ret-int after executing $\mathsf{max_{L1}}$.

We prove the predicate $\mathsf{corres_{L2}}$ compositionally using a syntax-directed approach similar to our rule set for $\mathsf{corres_{L1}}$. Fig 8 shows a sample of the rules used to carry out the proofs. We use the Hoare-style syntax $\{\!|P|\!\}\ C\ \{\!|Q|\!\}, \{\!|E|\!\}$ to state that program C starting in a state satisfying P ensures Q in the event of normal termination or E in the event of an exception.

The rule L2CorresReturn shows that the L1 statement $\mathsf{modifyE}\ M'$ refines the L2 statement $\mathsf{returnE}\ v$ if the state-update function M' only modifies locals, and the L2 return value v corresponds to the local updated in L1, extracted using rx. The rule L2CorresModify is similar, but is used when an L1 modifyE statement updates non-local state. Automating such proofs requires: (i) parsing the term M'; (ii) determining if it has a local or non-local effect; (iii) emitting the corresponding abstract statement; and finally (iv) generating the corresponding proof term. If an L2 term is correctly generated then the side-conditions of the rule are discharged automatically by Isabelle's simplifier.

The composition rules are more involved. For instance, L2CorresSeq states the L1 program fragment $A' \gg\!=_{\mathsf{E}} (\lambda\text{-}.\ B')$ refines the L2 program fragment $A \gg\!=_{\mathsf{E}} B$. For the rule to apply, A' must refine A under the precondition Q_A, and B' must refine B under precondition Q_B. This latter precondition has an additional parameter x representing the return value from A. We must prove that executing A' from a state satisfying P leads to a state s where the second

```
condE (λs. a <_s b)
   (doE
      ret ← returnE b;
      exn-var ← returnE Return;
      returnE ret
    odE)
   (doE
      ret ← returnE a;
      exn-var ← returnE Return;
      returnE ret
    odE)
```

Fig. 9. The `max` function after local variable lifting.

```
condE (λs. a <_s b)
   (returnE b)
   (returnE a)
```

Fig. 10. The `max` function after flow-sensitive optimisations.

```
returnE ( if a <_s b then b else a )
```

Fig. 11. The `max` function after type-strengthening.

precondition Q_B $(rx\ s)$ is satisfied. This second parameter to Q_B is what ensures that the locals stored in the L1 state match the bound variables used in L2.

To automatically prove an application of the L2CorresSeq rule, we must calculate a suitable precondition P that both implies the first precondition Q_A and will lead to Q_B being satisfied. We generate such a P stating that all bound variables required by A match their L1 state; and all bound variables required by B and not modified by A match their L1 state. Using this P, we can discharge the Hoare-style side-condition by showing that A' preserves all variables required by B which it does not otherwise pass in by bound variables; these proofs are again automated using a syntax-directed rule-set.

3.5 Flow-Sensitive Simplifications

A significant benefit of lifted local form is that it allows us to easily determine how local variables are used, and carry out simplifications based on this. Such simplifications include: (i) removing code that writes to locals that are never subsequently read from; (ii) using assumptions from guardE, condE and whileE statements to simplify later expressions; and (iii) collapsing variables that are only used once into the locations where they are used. By allowing constant valued expressions to be folded into the location they are used, we are also able to discharge many more guardE statements not previously provable and determine that some condE conditions always have the same value.

Fig 10 shows the `max` function after flow-sensitive optimisations: the redundant exn-var variable is detected, and the two returnE terms in each branch of the condE are collapsed into a single statement, resulting in a much simpler program.

3.6 Type Specialisation

So far, all of our generated programs have been written using our exception monad. Sec 2.3 outlined some of the motivations for using this monad, including our aim to represent C that supports reading and writing from global state; abrupt termination; non-determinism; or code that may fail.

For the majority of the code we translate, many of these features are not required. For example, Sec 3.3 describes how the majority of exception usage can be eliminated from specifications; the use of non-determinism is mostly limited to setting up uninitialised variables, many of which are eliminated using the simplifications in Sec 3.5; further, many functions do not modify the state of the system at all, either only reading the global state or having results that depend entirely on their input parameters. In these cases, the exception monad is far more expressive than required. Less expressive monads would constrain program behaviour by type and give the user free theorems by notation alone.

We therefore specialise the type of individual functions to contain only the features they require. The types we use are as follows, in decreasing strength:

Pure functional: These are standard Isabelle functions, where the function returns a deterministic output depending only on its input parameters. Our example `max` function falls into this category.

Option monad: The C standard is littered with restrictions that result in guardE statements that cannot be automatically discharged. Unfortunately, such a guardE statement will prevent a function from being translated into a pure Isabelle function, as we must consider failed executions. We can, however, use the *option monad*, where every computation either results in a single value a (represented as Some a), or failure (represented as None). Any intermediate failure results in failure of the entire computation.
Functions that may potentially fail, but are deterministic, have simple control flow, and only read from global state can be transformed into the option monad. Callers of such functions will translate a result of None into failure.

State monad: Functions which need to modify global state or use non-determinism but do not use exceptional control flow are translated into a state monad without exceptions.

Type specialisation takes place using a series of rewrite rules that attempt to strengthen individual parts of the program, and then combine partial results to strengthen larger parts of the program. For instance, the rewrite rules we use to strengthen the exception monad to a pure Isabelle function are as follows:

$$\mathsf{skipE} = \mathsf{returnE}\ ()$$
$$\mathsf{condE}\ (\lambda\text{-}.\ c)\ (\mathsf{returnE}\ A)\ (\mathsf{returnE}\ B) = \mathsf{returnE}\ (\mathsf{if}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B)$$
$$\mathsf{returnE}\ A \gg\!=_{\mathsf{E}} (\lambda x.\ \mathsf{returnE}\ (B\ x)) = \mathsf{returnE}\ (\mathsf{let}\ x = A\ \mathsf{in}\ B\ x)$$

Fig 11 shows the result of applying these rules to our example `max` function.

To determine which type we can strengthen each function to, we attempt to apply each set of strengthening rules in order from strongest type to weakest type. If a particular function can be completely rewritten, the transformation was successful and a new definition for the function is emitted. Otherwise, we continue to try alternative, more expressive, representations. If all translations fail, we simply continue to use the exception monad.

Table 2 shows statistics of type strengthening used on the seL4 microkernel source code, with almost 96% of functions being strengthened into another type. The remaining 4% of functions correspond to the functions unable to be rewritten to avoid using exceptions in Sec 3.3.

Table 1. Average function term size after each translation phase of the seL4 source.

Specification	Avg. Size
Simpl	356.7
Shallow Embedding	362.5
Control-Flow Peephole	286.2
Exception Rewriting	281.1
Lifted Local Vars	249.2
Flow-Sensitive Opts.	173.6
Type Strengthening	173.5
Polish	168.9

Table 2. Number of functions in the seL4 microkernel translated into each type.

Type	Count
Pure function	151
Option monad	51
State monad	309
Exception monad	24
Total	535

3.7 Final Theorem

Along with the abstracted program specification, the tool emits a proof of correctness. In particular, the final refinement theorem between the input Simpl and output monadic program is as follows:

$$
\begin{array}{l}
\mathsf{corres_F}\ st\ \Gamma\ rx\ P\ A\ C \equiv \\
\quad \forall s.\ P\ s \wedge \neg\ \mathsf{failed}\ (A\ (st\ s)) \longrightarrow \\
\qquad (\forall t.\ \Gamma \vdash \langle C, \mathsf{Normal}\ s\rangle \Rightarrow t \longrightarrow \\
\qquad\quad (\exists s'.\ t = \mathsf{Normal}\ s' \wedge \\
\qquad\qquad (\mathsf{Norm}\ (rx\ s'), st\ s') \in \mathsf{results}\ (A\ (st\ s)))) \wedge \\
\qquad \Gamma \vdash C \downarrow \mathsf{Normal}\ s
\end{array}
$$

It composes the two previous theorems, and is proved with the following rule:

$$
\frac{\mathsf{corres_{L2}}\ st\ rx\ (\lambda\text{-}.\ ())\ P\ A\ B \qquad \mathsf{corres_{L1}}\ \Gamma\ B\ C \qquad \mathsf{no\text{-}throw}\ A}{\mathsf{corres_F}\ st\ \Gamma\ rx\ P\ A\ C}
$$

Our final $\mathsf{corres_F}$ definition, while differing from the definition given in previous work on C abstraction in L4.verified [15], is strong enough to prove it.

In summary, we have shown the following transformations: (*i*) from deep into shallow embedding, which enables us to use rewriting; (*ii*) simple control flow peephole rewrites, exploiting the shallow embedding; (*iii*) exception rewriting, which further simplifies control flow; (*iv*) local variable lifting, which allows us to make use of Isabelle's built-in bound variables, substitution, and unification; (*v*) flow-sensitive rewrites, enabled by explicit bound variables; (*vi*) and, type specialisation, giving the user convenient notation and implicit free theorems.

The final additional step is a polishing phase which rewrites internal terms into a more human-friendly form. Table 1 quantifies the effect of each transformation by showing the average term size after each phase for the translation of seL4.

4 Related Work

The motivation for this work is the paper *Mind the Gap: A verification framework for low-level C* by Winwood et al. [15] in the context of the seL4 microkernel

verification. They showed that formal, interactive verification of low-level C code at scale is possible, but noted that automation could be improved. Our final refinement theorem implies their ccorres statement. Some of the automation ideas are present in early forms in this previous work, such as lifting a single variable from the state into a bound variable in the monad. In addition to our other transformations, we generalise this approach to completely automatically lift all variables of all functions in a program. On the technical side, our rule sets and refinement statements are tuned for full automation. The idea is to remove all unnecessary manual work in low-level C verification and enable the human to concentrate on the interesting reasoning instead.

Two further projects have treated large low-level code bases interactively. The Verisoft project [1] reasoned directly about Simpl using a VCG that translates Hoare triples about Simpl code into proof obligations. While the VCG provides some automation, it performs less abstraction. Consequently, the verification overhead in Verisoft was similarly high as in L4.verified.

The Verisoft-XT project applied the VCC tool [4]. VCC does not attempt automated abstraction of this form either, but instead uses a powerful SMT solver as backend reasoner to increase productivity. The increased automation comes at the cost of reduced expressiveness in annotations and explicit ghost state to guide the reasoner. While our focus is on interactive reasoning, we believe the approach is complementary: our tool could be used to generate a higher-level, less detailed model, and automated reasoners could then be used on top.

The FramaC framework [5] with the Jessie plug-in [8] also supports deductive verification of C. Annotated C code is translated into the functional language Why on which verification then proceeds. The translation touches on some transformations that are close to ours. The main difference is that these transformations need to be trusted whereas our work produces proofs. The necessarily trusted translation step from C into a formal logic is much smaller in our work.

5 Conclusions

We have presented a tool that automatically abstracts low-level C semantics into higher-level specifications with automatic proofs of correctness for each of the transformation steps. The tool consists of 3 300 lines of ML code and 5 000 lines of Isabelle proof script, on top of existing libraries for monads, Simpl and parsing.

While our main case study is the seL4 microkernel, because it provides a convenient known target for comparison, the tool is not specific to this kernel. We have also applied it to Tuch's memory allocator case study [14], and other projects such as the scheduler of a small commercial real-time system. We believe that the general idea can be applied to languages other than C, and that the tool may even be directly applicable to these as long as a front-end to Simpl exists.

The tool accepts anything the C parser front-end accepts, but presently does not translate recursive functions. While not a problem for embedded code, this is one of the obvious next steps for future work. The second direction for future work is to provide further translation steps, for instance exploiting Tuch's interactive

framework [13] to automatically generate a more abstract heap format for type safe fragments of the program.

Our experience indicates a significant improvement in clarity and ease of reasoning for the output of the tool. Our long term goal is to completely automate the low-level C verification phase in Winwood et al. [15] for projects like L4.verified.

Acknowledgements We are grateful to Matthias Daum, Daniel Matichuk, Thomas Sewell and the anonymous reviewers for their feedback on drafts of this paper.

References

1. E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. Balancing the load — leveraging a semantics stack for systems verification. *JAR: Special Issue Operat. Syst. Verification*, 42, Numbers 2–4:389–454, 2009.
2. E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In P. O'Hearn, G. T. Leavens, and S. Rajamani, editors, *VSTTE 2010*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, UK, Aug 2010. Springer.
3. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer.
4. 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 S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer.
5. The Frama-C platform. `http://www.frama-c.cea.fr/`, 2008.
6. ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.
7. 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 *22nd SOSP*, pages 207–220. ACM, 2009.
8. Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Paris, France, Jan 2009.
9. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
10. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. M. Norrish. C-to-Isabelle parser, version 0.7.2. `http://ertos.nicta.com.au/software/c-parser/`, Jan 2012.
12. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
13. H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug 2008.
14. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108. ACM, 2007.
15. S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 500–515, Munich, Germany, Aug 2009. Springer.