Deeply Integrating C11 Code Support into Isabelle/PIDE

Frédéric Tuong
LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
ftuong@lri.fr

Burkhart Wolff
LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
wolff@lri.fr

We present a framework for C code in C11 syntax deeply integrated into the Isabelle/PIDE development environment. Our framework provides an abstract interface for verification back-ends to be plugged-in independently. Thus, various techniques such as deductive program verification or white-box testing can be applied to the same source, which is part of an integrated PIDE document model. Semantic back-ends are free to choose the supported C fragment and its semantics. In particular, they can differ on the chosen memory model or the specification mechanism for framing conditions.

Our framework supports semantic annotations of C sources in the form of comments. Annotations serve to locally control back-end settings, and can express the term focus to which an annotation refers. Both the logical and the syntactic context are available when semantic annotations are evaluated. As a consequence, a formula in an annotation can refer both to HOL or C variables.

Our approach demonstrates the degree of maturity and expressive power the Isabelle/PIDE subsystem has achieved in recent years. Our integration technique employs Lex and Yacc style grammars to ensure efficient deterministic parsing. We present two case studies for the integration of (known) semantic back-ends in order to validate the design decisions for our back-end interface.

Keywords: User Interface, Integrated Development, Program Verification, Shallow Embedding

1 Introduction

Recent successes like the Microsoft Hypervisor project [16], the verified CompCert compiler [17] and the seL4 microkernel [13, 14] show that the verification of low-level systems code has become feasible. However, a closer look at the underlying verification engines VCC [8], or Isabelle/AutoCorres [10] show that the road is still bumpy: the empirical cost evaluation of the L4.verified project [13] reveals that a very substantial part of the overall effort of about one third of the 28 man years went into the development of libraries and the associated tool-chain. Accordingly, the project authors [13] express the hope that these overall investments will not have to be repeated for "similar projects".

In fact, none of these verifying compiler tool-chains capture all aspects of "real life" programming languages such as C. The variety of supported language fragments seem to contradict the assumption that we will all converge to one comprehensive tool-chain soon. There are so many different choices concerning memory models, non-standard control flow, and execution models that a generic framework is desirable: in which verified compilers, deductive verification, static analysis and test techniques (such as [12], [1]) can be developed and used inside the Isabelle platform as part of an integrated document.

In this paper we present Isabelle/C [1], a generic framework in spirit similar to Frama-C [7]. In contrast to the latter, Isabelle/C is deeply integrated into the Isabelle/PIDE document model [21]. Based on the

[1] The current developer snapshot is provided in `https://gitlri.lri.fr/ftuong/isabelle_c`.

R. Monahan, V. Prevosto, J. Proença (Eds.): Formal Integrated Development Environment 2019 (F-IDE 2019)
EPTCS 310, 2019, pp. 13–28, doi:10.4204/EPTCS.310.3

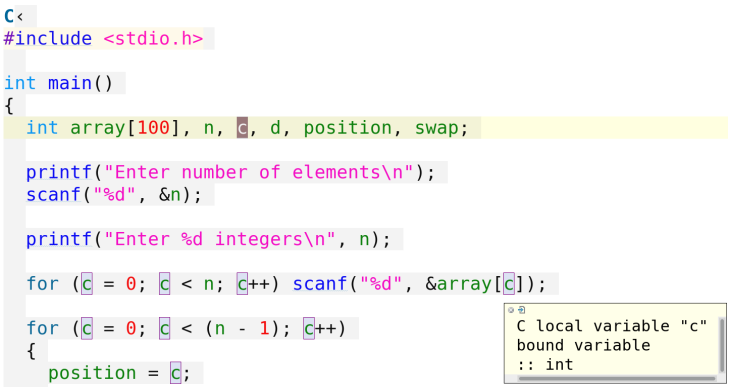


Figure 1: A C11 sample in Isabelle/jEdit

C11 standard (ISO/IEC 9899:2011), Isabelle/C parses C11 code inside a rich IDE supporting static scoping. SML user-programmed extensions can benefit from the parallel evaluation techniques of Isabelle. The plug-in mechanism of Isabelle/C can integrate diverse semantic representations, including those already made available in Isabelle/HOL [18]: AutoCorres [10], IMP2 [15], Orca [4], or Clean (discussed in this paper). A particular advantage of the overall approach compared to systems like Frama-C or VCC is that all these semantic theories are conservative extensions of HOL, hence no axiom-generators are used that produce the "background theory" and the verification conditions passed to automated provers. Isabelle/C provides a general infrastructure for semantic annotations specific for back-ends, i.e. modules that generate from the C source a set of definitions and derive automatically theorems over them. Last but not least, navigation features of annotations make the logical context explicit in which theorems and proofs are interpreted.

The heart of Isabelle/C, the new **C**‹ . . › command, is shown in Figure 1. Analogously to the existing **ML**‹ . . › command, it allows for editing C sources inside the ‹ . . › brackets, where C code is parsed on the fly in a “continuous check, continuous build” manner. A parsed source is coloured according to the usual conventions applying for Isabelle/HOL variables and keywords. A static scoping analysis makes the bindings inside the source explicit such that editing gestures like hovering and clicking may allow the user to reveal the defining variable occurrences and C type information (see yellow sub-box in the screenshot Figure 1). The C source may contain comments to set up semantic back-ends. Isabelle/C turns out to be sufficiently efficient for C sources such as the seL4 project.

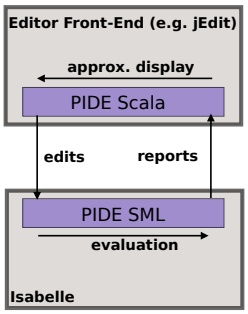
This paper proceeds as follows: in section 2, we briefly introduce Isabelle/PIDE and its document model, into which our framework is integrated. In section 3 and section 4, we discuss the build process and present some experimental results on the integrated parser. The handling of semantic annotations comments — a vital part for back-end developers — is discussed in section 5, while in section 6 we present some techniques to integrate back-ends into our framework at the hand of examples.

2 Background: PIDE and the Isabelle Document Model

The Isabelle system is based on a generic document model allowing for efficient, highly-parallelized evaluation and checking of its document content (cf. [2, 21, 22] for the fairly innovative technologies underlying the Isabelle architecture). These technologies allow for scaling up to fairly large documents: we have seen documents with 150 files be loaded in about 4 min, and individual files — like the x86

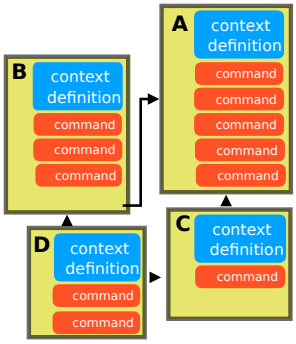
model generated from Antony Fox' L3 specs — have 80 kLoC and were loaded in about the same time.[2]

The PIDE (prover IDE) layer consists of a part written in SML and another in Scala. Roughly speaking, PIDE implements "continuous build and continuous check" functionality over a textual albeit generic document model. It transforms user modifications of text elements in an instance of this model into increments — *edits* — and communicates them to the Isabelle system. The latter reacts by the creation of a multitude of light-weight reevaluation threads resulting in an asynchronous stream of *reports* containing *markup* that is used to annotate text elements in the editor front-end. For example, such markup is used to highlight variables or keywords with specific colours, to hyperlink bound variables to their defining occurrences, or to annotate type information to terms which become displayed by specific user gestures on demand (such as hovering). Note that PIDE is not an editor, it is the framework that coordinates these asynchronous information streams and optimizes their evaluation to a certain extent: outdated markup referring to modified text is dropped, and corresponding re-calculations are oriented to the user focus, for example. For PIDE, several editor applications have been developed, where Isabelle/jEdit (`https://www.jedit.org`) is the most commonly known. More experimental alternatives based on Eclipse or Visual Studio Code exist.



2.1 The PIDE Document Model

The document model foresees a number of atomic sub-documents (files), which are organized in the form of an acyclic graph. Such graphs can be grouped into sub-graphs called *sessions* which can be compiled to binaries in order to avoid long compilation times — Isabelle/C as such is a session. Sub-documents have a unique name (the mapping to file paths in an underlying file-system is done in an integrated build management). The primary format of atomic sub-documents is `.thy` (historically for "theory"), secondary formats can be `.sty`, `.tex`, `.c` or other sub-documents processed by Isabelle and listed in a configuration of the build system.



A `.thy` file consists of a *context definition* and a body consisting of a sequence of *commands*. The context definition includes the sections **imports** and **keywords**. For example our context definition states that *`C_Command`* is the name of the sub-document depending on *`C_Eval`* which transitively includes the parser sources as (ML files) sub-documents, as well as the C environment and the infrastructure for defining C level annotations. *Keywords* like **C** or **C_file** must be declared before use.

```
theory C_Command
  imports C_Eval
  keywords "C" :: thy_decl
       and "C_file" :: thy_load
```

For this work, it is vital that predefined commands allow for the dynamic creation of *user-defined* commands similarly to the definition of new functions in a shell interpreter. Semantically, commands are transition functions $\sigma \to \sigma$ where σ represents the system state called *logical context*. The logical context in interactive provers contains — among many other things — the declarations of types, constant symbols as well as the database with the definitions and established theorems. A command starts with a pre-declared keyword followed by the specific syntax of this command; an *evaluation* of a command parses the input till the next command, and transfers the parsed input to a transition function, which can be configured in a late binding table. Thus, the evaluation of the generic document model allows for user programmed extensions including IDE and document generation.

[2] On a modern 6-core MacBook Pro with 32Gb memory, these loading times were counted *excluding* proof checking.

Note that the Isabelle platform supports multiple syntax embeddings, i.e. the possibility of nesting different language syntaxes inside the upper command syntax, using the ⟨ .. ⟩ brackets (such parsing techniques will be exploited in section 5). Accordingly, these syntactic sub-contexts may be nested. In particular, in most of these sub-contexts, there may be a kind of semantic macro — called antiquotation and syntactically denoted in the format `@{name ⟨ .. ⟩}` — that has access to the underlying logical context. Similar to commands, user-defined antiquotations may be registered in a late-binding table. For example, the standard *term*-antiquotation in **ML**`⟨ val t = @{term "3 +"} ⟩` parses the argument `"3 +"` with the Isabelle/HOL term parser, attempts to construct a λ-term in the internal term representation and to bind it to `t`; however, this fails (the plus operation is declared infix in logical context) and therefore the entire command fails.

2.2 Some Basics of PIDE Programming

A basic data-structure relevant for PIDE is *positions*; beyond the usual line and column information they can represent ranges, list of continuous ranges, and the name of the atomic sub-document in which they are contained. It is straightforward to use the antiquotation `@{here}` to infer from the system lexer the actual position of the antiquotation in the global document. The system converts the position to a markup representation (a string representation) and sends the result via `writeln` to the interface.

```
ML⟨ val pos = @{here};
    val markup = Position.here pos;
    writeln ("And a link to the declaration\
             \ of 'here' is " ^ markup) ⟩
```

```
And a link to the declaration of 'here' is ⌂
```

In return, the PIDE output window shows the little house-like symbol ⌂, which is actually hyperlinked to the position of `@{here}`. The ML structures `Markup` and `Properties` represent the basic libraries for annotation data which is part of the protocol sent from Isabelle to the front-end. They are qualified as "quasi-abstract", which means they are intended to be an abstraction of the serialized, textual presentation of the protocol. A markup must be tagged with a unique id; this is done by the library `serial` function. Typical code for taking a string `cid` from the editing window, together with its position `pos`, and sending a specific markup referring to this in the editing window managed by PIDE looks like this:

```
ML⟨ fun report_def_occur pos cid = Position.report pos (my_markup true cid (serial ()) pos) ⟩
```

Note that `my_markup` (not shown here) generates the layout attributes of the link and that the `true` flag is used for markup declaring `cid` as a defining occurrence, i.e. as *target* (rather than the *source*) in the hyperlink animation in PIDE.

3 The C11 Parser Generation Process and Architecture

Isabelle uses basically two parsing technologies:

1. Earley parsing [9] intensively used for mixfix-syntax denoting λ-terms in mathematical notation,
2. combinator parsing [11] typically used for high-level command syntax.

Both technologies offer the dynamic extensibility necessary for Isabelle as an interactive platform geared towards incremental development and sophisticated mathematical notations. However, since it is our goal to support *programming languages* in a fast parse-check-eval cycle inside an IDE, we opt for a

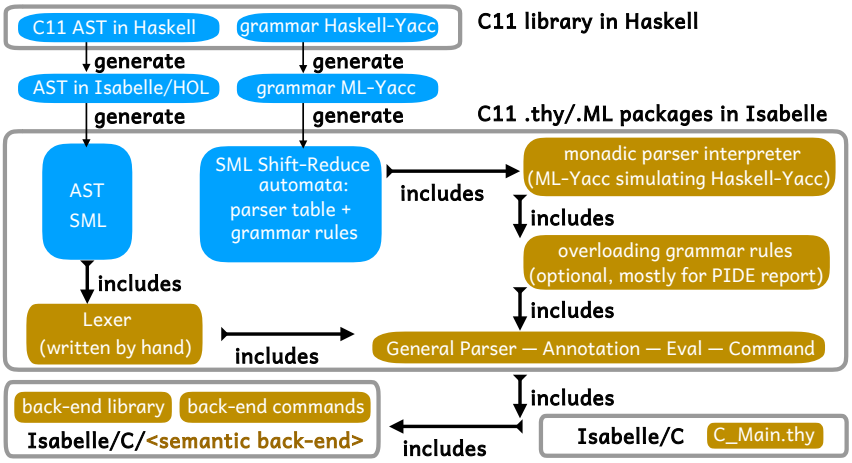


Figure 2: The architecture of Isabelle/C

Lex and Yacc deterministic grammar approach. It turns out the resulting automata based parser performs well enough for our purpose; the gain in performance is discussed in the next section.

In the following, we describe a novel technique for the construction and integration of this type of parser into the Isabelle platform. Since it is mostly relevant for integrators copying our process to similar languages such as JavaScript or Rust [3], users of the Isabelle/C platform may skip this section: for them, the take-home message is that the overall generation process takes about 1 hour, the compilation of the generated files takes 15s, and that the generated files should be fairly portable to future Isabelle versions.

We base our work on the C11 parsing library `http://hackage.haskell.org/package/language-c` implemented in Haskell by Huber, Chakravarty, Coutts and Felgenhauer; we particularly focus on its open-source Haskell Yacc grammar as our starting point. We would like to emphasize that this is somewhat arbitrary, our build process can be easily adapted to more recent versions when available.

The diagram in Figure 2 presents the architecture of Isabelle/C. The original Haskell library was not modified, it is presented in blue together with generated sources, in particular the final two blue boxes represent about 11 kLoC. In output, the glue code in brown constitutes the core implementation of Isabelle/C, amounting to 6 kLoC (without yet considering semantic back-ends).

3.1 Generating the AST

In the following, we refer to *languages* by $\mathscr{L}$, $\mathscr{I}$. The notation $\text{AST}^{\mathscr{L}}_{\mathscr{I}}$ refers to abstract syntaxes for language $\mathscr{L}$ implemented in language $\mathscr{I}$. For example, we refer by $\text{AST}^{\text{C11}}_{\text{ML}}$ to an AST implementation of C11 implemented in SML. Indices will be dropped when no confusion arises, or to highlight the fact that our approach is sufficiently generic.

For our case, we exploit that from a given Haskell source AST_{HS}, Haskabelle generates to a maximum extent an Isabelle/HOL theory. Via the Isabelle code generator, an AST_{ML} can be obtained from a

[3] E.g. `http://hackage.haskell.org/package/language-javascript` or `http://hackage.haskell.org/package/language-rust`

constructive AST_{HOL} representation. However, the process is challenging for technical reasons in practice due to the enormous size of AST^{C11} (several hundreds of constructors), and due to certain type declarations not initially supported by Haskabelle (we have to implement here the necessary features). Ultimately, the process to compile AST_{HS} to AST_{ML} is done only once at build time, it comprises:

1. the generation of AST_{HOL} from AST_{HS}, represented as a collection of **datatype**,
2. the execution of the **datatype** theory for AST_{HOL} and checking of all their proofs,[4]
3. the generation of an AST_{ML} from AST_{HOL}.

3.2 Constructing a Lexer for C11

We decided against the option of importing the equivalent Haskell lexer, as it is coming under-developed compared to the existing PIDE lexer library, natively supporting Unicode-like symbols (mostly for annotations). Using a more expressive position data-structure, our C lexer is also compatible with the native ML lexer regarding the handling of errors and backtracking (hence the perfect fit when nesting one language inside the other). Overall, the modifications essentially boil down to taking an extreme care of comments and directives which have intricate lexical conventions (see subsection 4.1).

3.3 Generating the Shift-Reduce Parser from the Grammar

In the original C11 library, together with AST_{HS}, there is a Yacc grammar file $\text{G}_{\text{HS-YACC}}$ included, which we intend to use to conduct the C parsing. However due to technical limitations of Haskabelle (and advanced Haskell constructs in the associated G_{HS}), we do not follow the same approach as subsection 3.1. Instead, an ultimate grammar G_{ML} is obtained by letting ML-Yacc participate in the generation process. In a nutshell, the overall grammar translation chain becomes: $\text{G}_{\text{HS-YACC}} \longrightarrow_{\text{HS}} \text{G}_{\text{ML-YACC}} \longrightarrow_{\text{ML}} \text{G}_{\text{ML}}$.

$\longrightarrow_{\text{HS}}$ is implemented by modifying the Haskell parser generator Happy, because Happy is already natively supporting the whole $\mathscr{L}_{\text{HS-YACC}}$. Due to the close connection between Happy and ML-Yacc, the translation is even almost linear. However cares must be taken while translating monadic rules [5] of $\text{G}_{\text{HS-YACC}}$, as $\mathscr{L}_{\text{ML-YACC}}$ does not support such rules. In G^{C11}, monadic rules are particularly important for scoping analyses, or while building new informative AST nodes (in contrast to disambiguating non-monadic rules, see `@` vs. `&` in section 5). Consequently, applying ML-Yacc $\longrightarrow_{\text{ML}}$ on $\text{G}_{\text{ML-YACC}}$ is not enough: after compiling G_{ML} to an efficient Shift-Reduce automaton, we substantially modified the own grammar interpreter of ML-Yacc to implement all features of $\mathscr{L}_{\text{HS-YACC}}$ presented as used in $\text{G}_{\text{HS-YACC}}$.

4 Isabelle/C: Syntax Tests and Experimental Results

The question arises, to what extent our construction provides a faithful parser for C11, and if Isabelle/C is sufficiently stable and robust to handle real world sources. A related question is the treatment of `cpp` preprocessing directives: while a minimal definition of the preprocessor is part of C standards since C99, practical implementations vary substantially. Moreover, `cpp` comes close to be Turing complete: recursive computations can be specified, but the expansion strategy bounds the number of unfolding.

[4] Large mutually recursive datatypes in AST_{HOL} might lead to worse performance time, see for instance `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-March/msg00034.html` and `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2017-April/msg00000.html`.

[5] `https://www.haskell.org/happy/doc/html/sec-monads.html`

Therefore, a complete cpp reimplementation contradicts our objective to provide efficient IDE support inside Isabelle. Instead, we restrict ourselves to a common subset of macro expansions and encourage, whenever possible, Isabelle specific mechanisms such as user programmed C annotations. C sources depending critically on a specific cpp will have to be processed outside Isabelle.[6]

4.1 Preprocessing Lexical Conventions: Comments and Newlines

A very basic standard example taken from the GCC / CPP documentation [7] shows the quite intricate mixing of comment styles that represents a challenge for our C lexer. A further complication is that it is allowed and common practice to use backslash-newlines \↩ *anywhere* in C sources, be it inside comments, string denotations, or even regular C keywords like `i`\↩`n`\↩`t` (see also Figure 4).

In fact, many C processing tools assume that all comments have already been removed via cpp before they start any processing. However, annotations in comments carry relevant information for back-ends as shown in section 5. Consequently, they must be explicitly represented in $\text{AST}^{\text{C11}}_{\text{ML}}$, whereas the initial $\text{AST}^{\text{C11}}_{\text{HS}}$ is not designed to carry such extra information. Annotations inside comments may again contain structured information like programming code, formulas, and proofs, which implies the need for nested syntax. Fortunately, Isabelle is designed to manage multiple parsing layers with the technique of *cascade sources* [8] (see also Figure 3). We exploit this infrastructure to integrate back-end specific syntax and annotation semantics based on the parsing technologies available.

```
C — ‹Nesting of comments› ‹
/* inside /* inside */ int a = "outside";
// inside // inside until end of line
int a = "outside";
/* inside
  // inside
inside
*/ int a = "outside";
// inside /* inside until end of line
int a = "outside";
›
```

4.2 Preprocessing Side-Effects: Antiquoting Directives vs. Pure Annotations

Whereas *comments* can be safely removed without affecting the meaning of C code, *directives* are semantically relevant for compilation and evaluation.

1. Classical directives: `#define x TOKS` makes any incoming C identifier `x` be replaced by some *arbitrary* tokens `TOKS`, even when included via the `#include` directive.
2. Typed (pseudo-)directives as commands: It is easy to overload or implement a new `#define'` acting only on a decided subset of well-formed `TOKS`. There are actually no differences between Isabelle/C directives and Isabelle commands: both are internally of type $\sigma \to \sigma$ (see section 2).
3. Non-expanding annotations: Isabelle/C annotations `/*@` $\mathscr{L}_{\text{annot}}$ `*/` or `//@` $\mathscr{L}_{\text{annot}}$ can be freely intertwined between other tokens, even inside directives. In contrast to (antiquoting) directives and similarly as C comments, their designed intent is to not modify the surrounding parsing code.

A limitation of Isabelle and its current document model is that there is no way for user programmed extensions to exploit implicit dependencies between sub-documents. Thus, a sub-document referred to via `#include <some_file>` will not lead to a reevaluation of a **C**⟨ . . ⟩ command whenever modified. (The only workaround is to open all transitively required sub-documents *by hand*.)

[6] Isabelle/C has a particular option to activate (or not) an automated call to cpp before any in-depth treatment.
[7] https://gcc.gnu.org/onlinedocs/cpp/Initial-processing.html
[8] http://isabelle.in.tum.de/repos/isabelle/file/83774d669b51/src/Pure/General/source.ML

4.3 A Validation via the seL4 Test Suite

The AutoCorres environment contains a C99 parser developed by Michael Norrish [14]. Besides a parser test-suite, there is the entire seL4 codebase (written in C99) which has been used for the code verification part of the seL4 project. While the parser in itself represents a component belonging to the trusted base of the environment, it is arguably the most tested parser for a semantically well-understood translation in a proof environment today.

It is therefore a valuable reference for a comparison test, especially since AST^{C99} and AST^{C11} are available in the same implementation language. From $\text{AST}^{\text{C11}}_{\text{HOL}}$ to $\text{AST}^{\text{C99}}_{\text{HOL}}$ we construct an abstraction function $C^{\downarrow}$. A detailed description of $C^{\downarrow}$ is out of the scope of this paper; we would like to mention that it was 4 man-months of work due to the richness of AST^{C11}. As such, the abstraction function $C^{\downarrow}$ is at the heart of the AutoCorres integration into our framework described in subsection 6.2. Note that AST^{C99} seems to be already an abstraction compared to the C99 standard. This gives rise to a particular testing methodology: we can compile the test suites as well as the seL4 source files by both ML parsers $\text{PARSE}^{\text{C99}}_{stop}$ and $\text{PARSE}^{\text{C11}}_{report}$, abstract the output of the latter via $C^{\downarrow}$ and compare the results.

Our test establishes that both parsers agree on the entire seL4 codebase. However trying to compare the two parsers using other criteria is not possible, for example we had to limit ourselves to C programs written in a subset of C99. Fundamentally, the two parsers are achieving different tasks: the one of PARSE_{stop} is to just return a parsed AST. In contrast, PARSE_{report} intends to maximize markup reporting, irrespective of a final parsing success or failure, and reports are provided in parallel during its (monadic) parsing activity. Thus, in the former scenario, the full micro-kernel written in 26 kLoC can be parsed in 0.1s. In the latter, all reports we have thought helpful to implement are totally rendered before 20s. Applying $C^{\downarrow}$ takes 0.02 seconds, so our PARSE_{report} gives an average of 2s for a 2-3 kLoC source. By interweaving a source with proofs referring to the code elements, the responsiveness of PIDE should therefore be largely sufficient.

5 Generic Semantic Annotations for C

With respect to interaction with the underlying proof-engine, there are essentially two lines of thought in the field of deductive verification techniques:

1. either programs and specifications — i.e. the pre- and post-condition contracts — are clearly separated, or
2. the program is annotated with the specification, typically by using some form of formal comment.

Of course, it is possible to inject the essence of annotated specifications directly into proofs, e.g. by instantiating the `while` rule of the Hoare calculus by the needed invariant inside the proof script. The resulting clear separation of programs from proofs may be required by organisational structures in development projects. However, in many cases, modelling information may be interesting for programmers, too. Thus, having pre- and post-conditions locally in the source close to its point of relevance increases its maintainability. It became therefore common practice to design languages with annotations, i.e. structured comments *inside* a programming source. Examples are ACSL standardized by ANSI/ISO (see `https://frama-c.com/download/acsl.pdf`) or UML/OCL [6] for static analysis tools. Isabelle/C supports both the inject-into-proof style and annotate-the-source style in its document model; while the former is kind of the default, we address in this section the necessary technical infrastructure for the latter.

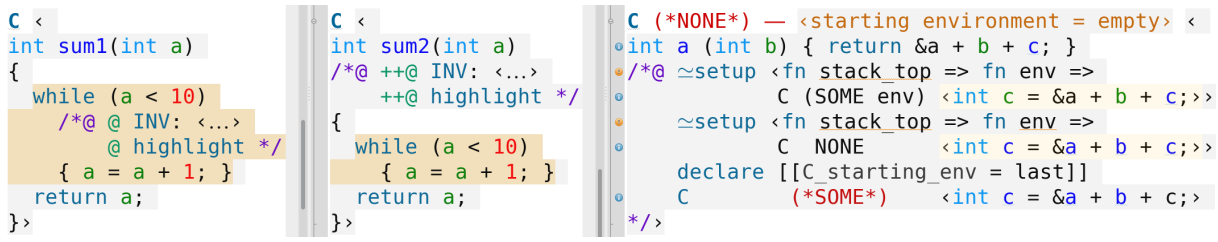


Figure 3: Advanced annotation programming

Generally speaking, a generic annotation mechanism which is sufficiently expressive to capture idioms used in, e.g., Frama-C, Why3, or VCC is more problematic than one might think. Consider this:

```
for (int i = 0; i < n; i++) a+= a*i /*@ annotation */
```

To which part of the AST does the annotation refer? To `i`? `a*i`? The assignment? The loop? Some verification tools use prefix annotations (as in Why3 for procedure contracts), others even a kind of parenthesis of the form:

```
/*@ annotation_begin */ ... /*@ annotation_end */
```

The matter gets harder since the C environment — a table mapping C identifiers to their type and status — changes according to the reference point in the AST. This means that the context relevant to type-check an annotation such as */*@ assert ⟨a > i⟩ */* strongly differs depending on the annotation's position. And the matter gets even further complicated since Isabelle/C lives inside a proof environment; here, local theory development (rather than bold ad-hoc axiomatizations) is a major concern.

The desire for fast impact analysis resulting from changes may inspire one to annotate local proofs near directives, which is actually what is implemented in our Isabelle/C/AutoCorres example (section 6).

```
C ‹
#define SQRT_UINT_MAX 65536
/*@ lemma uint_max_factor [simp]:
"UINT_MAX = SQRT_UINT_MAX * SQRT_UINT_MAX - 1"
by (clarsimp simp: UINT_MAX_def SQRT_UINT_MAX_def)
*/›
```

In the example, the semantic back-end converts the cpp macro into a HOL *definition*, i.e. an extension of the underlying theory context by the conservative axiom *SQRT_UINT_MAX* ≡ *65536* bound to the name *SQRT_UINT_MAX_def*. This information is used in the subsequent proof establishing a new theory context containing the lemma *uint_max_factor* configured to be used as rewrite rule whenever possible in future proofs. This local lemma establishes a relation of *SQRT_UINT_MAX* to the maximally representable number *UINT_MAX* for an unsigned integer according to the underlying memory model.

Obviously, the scheduling of these transformations of the underlying theory contexts is non-trivial.

5.1 Navigation for Annotation Commands

In order to overcome the problem of syntactic ambiguity of annotations, we slightly refine the syntax of semantic annotations by the concept of a navigation expression:

$\mathscr{L}_{\text{annot}} = \varnothing \mid$ <navigation-expr> <annotation-command> $\mathscr{L}_{\text{annot}}$

A <navigation-expr> string consists of a sequence of + symbols followed by a sequence consisting of @ or & symbols. It allows for navigating in the syntactic context, by advancing tokens with several +, or taking an ancestor AST node with several @ (or & which only targets monadic grammar rules). This

corresponds to a combination of right-movements in the AST, and respectively parent-movements. This way, the "focus" of an `<annotation-command>` can be modified to denote any C fragment of interest.

As a relevant example for debugging, consider Figure 3. The annotation command **highlight** is a predefined Isabelle/C ML-library function that is interpreted as C annotation. Its code is implicitly parameterized by the syntactical context, represented by `stack_top` whose type is a subset of AST^{C11}, and the lexical environment `env` containing the lexical class of identifiers, scopes, positions and serials for markup. The navigation string before **highlight** particularly influences which `stack_top` value gets ultimately selected. The third screenshot in Figure 3 demonstrates the influence of the static environment: an Isabelle/C predefined command ≃**setup** allows for "recursively" calling the C environment itself. This results in the export of definitions in the surrounding logical context, where the propagation effect may be controlled with options like *C_starting_env*. ≃**setup** actually mimics standard Isabelle **setup** command, but extends it by `stack_top` and `env` [9]. In the example, the first recursive call uses `env` allowing it to detect that `b` is a local parameter, while the second ignores it which results in a treatment as a free global variable. Note that bound global variables are not green but depicted in black.

5.2 Defining Annotation Commands

Extending the default configuration of commands, text and code antiquotations from the Isabelle platform to Isabelle/C is straightforward. For example, the central Isabelle command definition:

```
Outer_Syntax.command: K_cmd -> (σ -> σ) parser -> unit
```

establishes the dynamic binding between a command keyword K_{cmd} = **definition** | **lemma** | ... and a parser, whose value is a system transition.[10] The `parser` type stems from the aforementioned parser combinator library: `'a parser = Token.T list -> 'a * Token.T list`.

Analogously, Isabelle/C provides an internal late-binding table for *annotation commands*:

```
C_Annotation.command : K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> unit
C_Annotation.command': K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> σ -> σ
C_Token.syntax': 'a parser -> 'a c_parser
```

where in this paper we define $R_{cmd} = \sigma \text{ -> } \sigma$ as above.[11] Since the type `c_parser` is isomorphic to `parser`, but accepting C tokens, one can use `C_Token.syntax'` to translate and carry the default Isar commands *inside* the **C**〈 .. 〉 scope, such as **lemma** or **by**. Using ≃**setup**, one can even define an annotation command **C** taking a C code as argument, as the ML code of ≃**setup** has type α^{AST} `-> env ->` R_{cmd} (which is enough for calling `C_Annotation.command'` in the ML code). Here, whereas the type `env` is always the same, the type $\alpha^{\text{AST}} \subseteq \text{AST}^{\text{C11}}$ varies depending on `<navigation-expr>` (see subsection 5.3).

Note, however, that the user experience of the IDE changes when nesting commands too deeply. In terms of error handling and failure treatment, there are some noteworthy implementation differences between the outermost commands and C annotation commands. Naturally, the PIDE toplevel has been optimized to maximize the error recovery and parallel execution. Inside a command, the possibilities to mimic this behaviour are somewhat limited. As a workaround useful during development and debugging, we offer a further pragma for a global annotation, namely `*` (in complement to the violet `@`), that controls a switch between a strict and a permissive error handling for nested annotation commands.

[9] cf. `https://isabelle.in.tum.de/doc/isar-ref.pdf`

[10] σ has actually the internal Isabelle type `Toplevel.transition`.

[11] In some parallel work, we focus on running commands in native efficient speed with R_{cmd} = `(`K_{cmd}`*(σ->σ))list`. [20]

5.3 Evaluation Order

We will now explain why positional languages are affecting the evaluation time of annotation commands in Figure 3. This requires a little zoom on how the parsing is actually executed.

The LALR parsing of our implemented C11 parser can be summarized as a sequence of alternations between Shift and Reduce actions. By definition of LALR, whereas a unique Shift action is performed for each C token read from left to right, some unlimited number of Reduce actions are happening between two Shifts. Internally, the parser manages a stack-like data-structure called α^{AST} `list` representing all already encountered Shift and Reduce actions (SR). A given α^{AST} `list` can be seen as a *forest of SR nodes*: all leafs are tagged with a Shift, and any other parent node is a Reduce node. After a certain point in the parsing history, the top stack element α^{AST} (cast with the right type) is returned to $\simeq$**setup**.

Since a SR-forest is a list of SR-trees, it is possible to go forward and backward at will in the actually unparsed SR-history, and execute a sequence of SR parsing steps only when needed. While every annotation command like $\simeq$**setup** is by default attached to a closest previous Shift leaf, navigation expressions modify the attached node, making the presentation of α^{AST} referring to another term focus.

Instead of visiting the AST in the default bottom-up direction during parsing, it is possible to store the intermediate results, so that it can be revisited by using another direction strategy, for example top-down after parsing (where a parent node is executed before any of its children, and knows how they have been parsed thanks to α^{AST}). This enables commands to decide if they want to be executed during parsing, or after the full AST has been built. This gives rise to the implementation of different versions of annotation commands that are executed at different moments, relative to the parsing process. For example, the annotation command $\simeq$**setup** has been defined for being executed at bottom-up time, whereas the execution of the variant $\simeq$**setup**$\Downarrow$ happens at top-down time. In the above example, **C1** is a new command defined by `C_def`, a shorthand antiquotation for `C_Annotation.command'`. Since **C1** is meant to be executed during bottom-up time (during parsing), it is executed before **C2** is defined (which is directly after parsing).

```
C ‹int _;
/*@ @ C ‹//@ C1 ‹int _; //@ @ ≃setup⇓ ‹@{C_def ⇑ C2}› \
                              @ C1  ‹//* C2 ‹int _;››   \
                              @ C1⇓ ‹//* C2 ‹int _;››    ››
   @ C ‹//* C2 ‹int _;›                                   ›
     ≃setup ‹@{C_def ⇑ (* bottom-up *)  C1  }›
     ≃setup ‹@{C_def ⇓ (* top-down  *) "C1⇓"}›
*/›
```

Note that the C11 grammar has enough scoping structure for the full inference of the C environment `env` be at bottom-up time. In terms of efficiency, we use specific *static* rule wrappers having the potential of overloading default grammar rules (see Figure 2), to assign a wrapper to be always executed as soon as a Shift-Reduce rule node of interest is encountered. The advantage of this construction is that the wrappers are statically compiled, which results in a very efficient reporting of C type information.

6 Semantic Back-Ends

In this section, we briefly present two integrations of verification back-ends for C. We chose Clean used for program-based test generation [12], and AutoCorres [10], arguably the most developed deductive verification environment for machine-oriented C available at present.

Note that we were focusing on keeping modifications of integrated components minimal, particularly for the case of AutoCorres. Certain functionalities like position propagation of HOL terms in annotations, or "automatic" incremental declarations [12] may require internal revisions on the back-end side. This is out of the scope of this paper.

[12] `https://github.com/seL4/l4v/blob/master/tools/autocorres/tests/examples/Incremental.thy`

6.1 A Simple Typed Memory Model: Clean

Clean (pronounced as: "C lean" or "Céline" [selin]) is based on a simple, shallow-style execution model for an imperative target language. It is based on a "no-frills" state-exception monad **type_synonym** *('o, 'σ) MON$_{SE}$ = ⟨'σ ⇀ ('o × 'σ)⟩* with the usual definitions of *bind* and *unit*. In this language, sequence operators, conditionals and loops can be integrated. From a concrete program, the underlying state *'σ* is constructed by a sequence of extensible record definitions:

1. Initially, an internal control state is defined to give semantics to *break* and *return* statements:

   ```
   record control_state = break_val :: bool return_val :: bool
   ```

   control_state represents the σ_0 state.

2. Any global variable definition block with definitions $a_1 : \tau_1 \ \ldots \ a_n : \tau_n$ is translated into a record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1; ...; a_n :: τ_n
   ```

3. Any local variable definition block (as part of a procedure declaration) with definitions $a_1 : \tau_1 \ \ldots \ a_n : \tau_n$ is translated into the record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1 list; ...; a_n :: τ_n list; result :: τ_{result−type} list;
   ```

   where the *list*-lifting is used to model a *stack* of local variable instances in case of direct recursions and the *result* used for the value of the *return* statement.

The **record** package creates an *'σ* extensible record type *'σ control_state_ext* where the *'σ* stands for extensions that were subsequently "stuffed" in them. Furthermore, it generates definitions for the constructor, accessor and update functions and automatically derives a number of theorems over them (e.g., "updates on different fields commute", "accessors on a record are surjective", "accessors yield the value of the last update"). The collection of these theorems constitutes the *memory model* of Clean. This model might be wrong in the sense that it does not reflect the operational behaviour of a particular compiler, however, it is by construction *logically consistent* since it is impossible to derive falsity from the entire set of rules.

On this basis, assignments, conditionals and loops are reformulated into *break*-aware and *return*-aware versions as shown in the figure aside. The Clean theory contains about 600 derived theorems containing symbolic evaluation and Hoare-style verification rules.

```
definition assign :: "('σ control_state_scheme  ⇒
                       'σ control_state_scheme) ⇒
                      (unit, 'σ control_state_scheme) MONSE"
  where   "assign 𝒰 = (λσ. if break_val σ ∨ return_val σ
                           then Some((), σ) else Some((), 𝒰 σ))"

definition ifClean :: "['σ control_state_ext ⇒ bool,
                        ('β, 'σ control_state_ext) MONSE,
                        ('β, 'σ control_state_ext) MONSE] ⇒
                       ('β, 'σ control_state_ext) MONSE"
  where   "ifClean ℬ T F = (λσ. if break_val σ ∨ return_val σ
                                then Some (undefined, σ) —
                               ‹state unchanged, return arbitrary›
                                else if ℬ σ then T σ else F σ)"
```

Importing Clean into a theory with its activated back-end proceeds as in Figure 4. Clean generates for the C program a common type for the state, based on two generated extensible records — in the figure: just a global variable *k* and a local variable with a stack of result values for *prime$_C$*. Clean maps machine integers simply and naively on the HOL type *int*. The core of this program is represented by two generated definitions available subsequently in the logical context, where they are ready to be used in symbolic executions or proofs.

```
theory Prime imports Isabelle_C_Clean.Backend
                  — ‹Clean back-end is now on›
begin
C ‹
//@ definition ‹prime_HOL (p :: nat) =                    \
          (1 < p ∧ (∀ n ∈ {2..<p}. ¬ n dvd p))›
#   define SQRT_UINT_MAX 65536
unsigned int k = 0;
unsigned int prime_C(unsigned int n) {
//@ pre_Clean  ‹C‹n› ≤ UINT_MAX›
//@ post_Clean ‹C‹prime_C(n)› ≠ 0 ⟷ prime_HOL C‹n››
    if (n < 2) return 0;
    for (unsigned i = 2; i < SQRT_UINT_MAX
                         && i * i <= n; i++) {
      if (n % i == 0) return 0;
      k++;
    }
    return 1;
}›
```

```
prime_C_core_def: "prime_C_core n ≡
  if_Clean ⟨(n < 2) ⟩then return 0 else skip;-
  ⟨ i := 2 ⟩;-
  while_Clean ⟨i < SQRT_UINT_MAX ∧ i * i ≤ n⟩
    (if_Clean ⟨n mod i = 0⟩
      then return 0 else skip;
     ⟨k:=k+1⟩; assert ⟨k≤UINT_MAX ⟩
     ⟨i:=i+1⟩; assert ⟨i≤UINT_MAX ⟩) ;-
  return 1"

prime_C_def: "prime_C n ≡
  block_Clean push_local_prime_C_state
              (is_prime_core n)
              pop_local_prime_C_state"
```

Figure 4: Activating the Isabelle/C/Clean back-end triggers the generation of theorems

Generated definitions include push and pop operations for local variable blocks, for the entire variable space of procedures. Additionally, a specific syntax is introduced to represent assignments on global and local variables. For example, *i := 2* internally rewrites to *assign (λσ. ((i_upd o map_hd) (λ_. 2)) σ)*. The *return* operation is syntactically equivalent to the assignment of the result variable in the local state (stack) and sets the *return_val* flag. On this representation of the C program, the HOL term *prime_C n* can be decomposed into program test-cases according to a well-established coverage criterion. Technically, this is done by a variant of the program-based testing method

```
apply (branch_and_loop_coverage "Suc (Suc (Suc 0))")
```

developed in [12], which also uses Clean as semantic basis. Note that the testing approach does not need the formulation of an invariant, which is already non-trivial in the given example.

Finally, we will have a glance at the code for the registration of the annotation commands used in the example. Thanks to Isabelle/C's function `C_Annotation.command'`, the registration of user-defined annotations is very similar to the registration of ordinary commands in the Isabelle platform.

```
ML ‹fun command keyword f =
  C_Annotation.command' keyword ""
    (C_Token.syntax'
      (Parse.token Parse.cartouche)
     >>> toplevel f)›
setup ‹command ("pre_Clean", ⌂) Spec
    #> command ("post_Clean", ⌂) End_spec
    #> command ("inv_Clean", ⌂) Invariant›
```

6.2 The Case of AutoCorres

The AutoCorres environment consists of a C99 parser, compiling to a deepish embedding of a generic imperative core programming language, over a refined machine word oriented memory model, and a translator of this presentation into a shallow language based on another Monad for non-deterministic computations. This translator has been described in [10, 23] in detail. However, the original use of AutoCorres implies a number of protocol rules to follow, and is only loosely integrated into the Isabelle document model, which complicates the workflow substantially.

Our running example $prime_C$ for Isabelle/C/AutoCorres basically differs in what the theory is importing in its header. Similarly to Clean, AutoCorres constructs a memory model and represents the program as a monadic operation on it. Actually, it generates even two presentations, one on a very precise word-level memory model taking aspects of the underlying processor architecture into account, and another one more abstract, then it automatically proves the correspondence in our concrete example. Both representations become the definitions *prime_C_def* and *prime_C'_def*. A Hoare-calculus plus a derived verification generator *wp* from the AutoCorres package leverage finally the correctness proof.

```
theory Prime
  imports Isabelle_C_AutoCorres.Backend
begin
C ‹
  ⋮
theorem (in primeC) primeC'_correct:
  ‹{| λ _. n ≤ UINT_MAX |} primeC' n
   {| λ primeC' _. primeC' ≠ 0 ⟷ primeHOL n |}!›
proof (rule validNF_assume_pre)
 assume 1: ‹n ≤ UINT_MAX›
 have   2: ‹n = 0 ∨ n = 1 ∨ n > 1› by linarith
 show ?thesis
  proof (insert 2, elim disjE)
   assume ‹n = 0›
   then show ?thesis
        by (clarsimp simp: primeC'_def, wp, auto)
  next
  ⋮
```

Note that the integration of AutoCorres crucially depends on the conversion $\text{AST}^{\text{C11}} \Rightarrow \text{AST}^{\text{C99}}$ of $C^{\downarrow}$ discussed in subsection 4.3. In particular, for the overall seL4 annotations **INVARIANT**, **INV**, **FNSPEC**, **RELSPEC**, **MODIFIES**, **DONT_TRANSLATE**, **AUXUPD**, **GHOSTUPD**, **SPEC**, **END-SPEC**, **CALLS**, and **OWNED_BY**, we have extended our implementation of $C^{\downarrow}$ in such a way that the conversion places the information at the right position in the target AST. Obviously, this works even when navigation is used, as in Figure 3 left.

7 Conclusions

We presented Isabelle/C a novel, generic front-end for a deep integration of C11 code into the Isabelle/PIDE framework. Based on open-source Lex and Yacc style grammars, we presented a build process that constructs key components for this front-end: the lexer, the parser, and a framework for user-defined annotations including user-defined annotation commands. While the generation process is relatively long, the generated complete library can be loaded in a few seconds constructing an environment similar to the usual **ML** environment for Isabelle itself. 20 kLoC large C sources can be parsed and decorated in PIDE within seconds.

Our framework allows for the deep integration of the C source into a global document model in which literate programming style documentation, modelling as well as static program analysis and verification co-exist. In particular, information from the different tools realized as plugin in the Isabelle platform can flow freely, but based on a clean management of their semantic context and within a framework based on conservative theory development. This substantially increases the development agility of such type of sources and may be attractive to conventional developers, in particular when targeting formal certification [5].

Isabelle/C also forms a basis for future semantically well-understood combinations of back-ends based on different semantic interpretations: inside Isabelle, bridge lemmas can be derived that describe the precise conditions under which results from one back-end can be re-interpreted and used in another. Future tactic processes based on these bridge lemmas may open up novel ways for semantically safe tool combinations.

Acknowledgments. The authors warmly thank David Sanán and Yang Liu for encouraging the development and reuse of $C^{\downarrow}$, initially started in the Securify project [19] (`http://securify.sce.ntu.edu.sg/`).

References

[1] Romain Aïssat, Frédéric Voisin & Burkhart Wolff (2016): *Infeasible Paths Elimination by Symbolic Execution Techniques - Proof of Correctness and Preservation of Paths*. In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pp. 36–51, doi:10.1007/978-3-319-43144-4_3.

[2] Bruno Barras, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel & Burkhart Wolff (2013): *Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems*. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka & Wolfgang Windsteiger, editors: *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, *Lecture Notes in Computer Science* 7961, Springer, pp. 359–363, doi:10.1007/978-3-642-39320-4_29.

[3] Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors (2009): *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. *Lecture Notes in Computer Science* 5674, Springer, doi:10.1007/978-3-642-03359-9.

[4] Joshua A Bockenek, Peter Lammich, Yakoub Nemouchi & Burkhart Wolff (2018): *Using Isabelle/UTP for the Verification of Sorting Algorithms A Case Study*. `https://easychair.org/publications/preprint/CxRV`. Isabelle Workshop 2018, Colocated with Interactive Theorem Proving. As part of FLOC 2018, Oxford, GB.

[5] Achim D. Brucker, Idir Aït-Sadoune, Paolo Crisafulli & Burkhart Wolff (2018): *Using the Isabelle Ontology Framework - Linking the Formal with the Informal*. In: *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pp. 23–38, doi:10.1007/978-3-319-96812-4_3.

[6] Achim D. Brucker, Frédéric Tuong & Burkhart Wolff (2014): *Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5*. *Archive of Formal Proofs* 2014. `https://www.isa-afp.org/entries/Featherweight_OCL.shtml`.

[7] CEA-List (2019): *The Frama-C Home Page*. `https://frama-c.com`. Accessed 2019-03-24.

[8] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In Berghofer et al. [3], pp. 23–42, doi:10.1007/978-3-642-03359-9_2.

[9] Jay Earley (1970): *An Efficient Context-Free Parsing Algorithm*. *Commun. ACM* 13(2), pp. 94–102, doi:10.1145/362007.362035.

[10] David Greenaway, Japheth Lim, June Andronick & Gerwin Klein (2014): *Don't sweat the small stuff: formal verification of C code without the pain*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 429–439, doi:10.1145/2594291.2594296.

[11] Graham Hutton (1992): *Higher-Order Functions for Parsing*. *J. Funct. Program.* 2(3), pp. 323–343, doi:10.1017/S0956796800000411.

[12] Chantal Keller (2018): *Tactic Program-Based Testing and Bounded Verification in Isabelle/HOL*. In: *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018,*

Toulouse, France, June 27-29, 2018, Proceedings, pp. 103–119, doi:10.1007/978-3-319-92994-1_6.

[13] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski & Gernot Heiser (2014): *Comprehensive formal verification of an OS microkernel*. *ACM Trans. Comput. Syst.* 32(1), pp. 2:1–2:70, doi:10.1145/2560537.

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: formal verification of an OS kernel*. In Jeanna Neefe Matthews & Thomas E. Anderson, editors: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, ACM, pp. 207–220, doi:10.1145/1629575.1629596.

[15] Peter Lammich & Simon Wimmer (2019): *IMP2 - Simple Program Verification in Isabelle/HOL*. *Archive of Formal Proofs* 2019. `https://www.isa-afp.org/entries/IMP2.html`.

[16] Dirk Leinenbach & Thomas Santen (2009): *Verifying the Microsoft Hyper-V Hypervisor with VCC*. In Ana Cavalcanti & Dennis Dams, editors: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, *Lecture Notes in Computer Science* 5850, Springer, pp. 806–809, doi:10.1007/978-3-642-05089-3_51.

[17] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[18] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[19] David Sanán, Yongwang Zhao, Zhe Hou, Fuyuan Zhang, Alwen Tiu & Yang Liu (2017): *CSimpl: A Rely-Guarantee-Based Framework for Verifying Concurrent Programs*. In Axel Legay & Tiziana Margaria, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, *Lecture Notes in Computer Science* 10205, pp. 481–498, doi:10.1007/978-3-662-54577-5_28.

[20] Frédéric Tuong & Burkhart Wolff (2015): *A Meta-Model for the Isabelle API*. *Archive of Formal Proofs* 2015. `https://www.isa-afp.org/entries/Isabelle_Meta_Model.shtml`.

[21] Makarius Wenzel (2014): *Asynchronous User Interaction and Tool Integration in Isabelle/PIDE*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, *Lecture Notes in Computer Science* 8558, Springer, pp. 515–530, doi:10.1007/978-3-319-08970-6_33.

[22] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014.*, *EPTCS* 167, pp. 84–94, doi:10.4204/EPTCS.167.10.

[23] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock & Michael Norrish (2009): *Mind the Gap*. In Berghofer et al. [3], pp. 500–515, doi:10.1007/978-3-642-03359-9_34.