

A Certifiable Formal Semantics of C

Maksym Bortin

Christoph Lüth

Dennis Walter

1 Introduction

This paper presents a formalisation of a subset of the C programming language, and a corresponding verification calculus, in the theorem prover Isabelle. There are of course many and varied approaches to the verification of safety-critical programs. The characteristics of our approach stem from the application domain¹: the certification of control software for autonomous mobile robots [3]. This means that firstly, our verification techniques need to stand up to certification by an external agency such as the TÜV; secondly, we can restrict ourselves to a subset of C tailored for safety-critical applications, such as MISRA C [5] (in fact, this is even required by the relevant standard IEC 61508); and thirdly, the algorithms to be verified are comparatively sophisticated for a safety function, involving the calculation of safety zones from a model of the braking behaviour of the robot.

Our verification is based on a formalisation of a subset of MISRA C in the theorem prover Isabelle in typed higher-order logic (HOL). Using Isabelle is crucial to our approach: based on the C semantics, we can build a proof calculus and verify its correctness inside Isabelle. Thus, the validation of our verification can focus on the semantics of C as presented here. Further, using Isabelle allows us to use higher-order logic to express our specifications, so we are not tied to a specific specification language.

2 A formal semantics for a C subset

Overall, our model is close to a text-book semantics as found in e.g. [7], but of course some extensions are necessary to cover features of the C programming language like pointers and their arithmetic, nested structures, arrays and expressions with side-effects.

2.1 The memory model

A *state* represents a program's memory. In contrast to the usual memory model as a stack of local variables and a heap containing allocated objects, we use a flat model where all objects are given a *base location*. The state maps these base locations to object representations. Fig. 1 depicts the structure of a state.

Concretely, a state is a partial function $\Sigma : BaseLoc \rightarrow (Type \times (\mathbb{N} \rightarrow Val))$ mapping base locations to representations of scalar or structured values and their (run-time) type *Type*. Objects are represented as sequences of what the C standard calls *scalar* values, i.e. integer and floating-point numbers and addresses. These sequences are modelled as partial functions from \mathbb{N} to *Val*, the type of scalar values. To access scalar values (possibly inside structures or arrays) we use *locations*, which are pairs $(BaseLoc \times \mathbb{N})$. Thus, locations represent addresses. They allow a limited form of arithmetic, as defined in the standard: we can add and subtract the offsets of locations sharing the same base location.

Our model precludes the use of structured values in expressions, so they cannot occur as arguments to assignment or functions. The basic operations on states are allocation, deallocation, reading and writing:

$$\begin{array}{ll} extend : BaseLoc \rightarrow \Sigma \rightarrow \Sigma & read : Loc \rightarrow \Sigma \rightarrow Val \\ dealloc : BaseLoc \rightarrow \Sigma \rightarrow \Sigma & update : Loc \rightarrow Val \rightarrow \Sigma \rightarrow \Sigma \end{array}$$

Deallocation is currently used for local variables on function exit; *malloc* and *free* could easily be supported by our model as well. State updates always succeed, i.e. at the state level we don't perform type checks, array bounds checks or pointer validity checks. Sanity checks are instead inserted into the semantics of pointer dereferencing and array access.

¹<http://www.sams-project.org/>

$BaseLoc_0$	$Type_0$	$Val_{0,0}$	$Val_{0,1}$	$Val_{0,2}$	\dots
$BaseLoc_1$	$Type_1$	$Val_{1,0}$			
\vdots	\vdots	\vdots	\vdots	\vdots	
$BaseLoc_n$	$Type_n$	$Val_{n,0}$	$Val_{n,1}$		

Figure 1: A state maps base locations to types and sequences of scalar values

2.2 Language features

This is a brief and incomplete list of supported (●) and unsupported (○) features of the C language.

- The address-of operator `&`
- Arrays and nested structures
- Pointer offsets and subtraction
- `sizeof` operator
- Side-effects in expressions
- Casts to and from `void *`
- Function pointers
- `union`
- `switch`, `goto`, `break`, `continue`
- Dynamic memory management (`malloc`)

2.3 State transformer semantics

The abstract syntax of C programs is modelled as a collection of Isabelle/HOL datatypes. We provide a deterministic denotational semantics for the language that identifies all kinds of faults like invalid memory access, non-termination or division by zero as complete failure. Hence, the semantics of functions, statements and expressions map the corresponding data type to partial state transformers:

$$\begin{aligned} \llbracket stmt \rrbracket &: \Gamma \rightarrow \Sigma \rightarrow 1 \times \Sigma \\ \llbracket expr \rrbracket &: \Gamma \rightarrow \Sigma \rightarrow Val \times \Sigma \\ \llbracket f(x_1, \dots, x_n) \rrbracket &: \Gamma \rightarrow Val^n \rightarrow \Sigma \rightarrow Val \times \Sigma \end{aligned}$$

An environment Γ maps a variable to its allocated base location, and function identifiers to their semantics:

$$\Gamma \cong (Id \rightarrow BaseLoc) \times (FunId \rightarrow (Val^n \rightarrow \Sigma \rightarrow Val \times \Sigma))$$

Since expressions can have side-effects, evaluation order is important. However, the MISRA-C guidelines [5, Rule 12.2] require that an expression must yield the same value under every evaluation order. As we only consider MISRA-conformant programs, it is appropriate to fix the evaluation order, proceeding from left to right for both function argument lists and expression trees.

A rather drastic simplification from a theoretical point of view is the lack of support for recursive functions (also a MISRA requirement). This allows us to give semantics to functions in a sequential fashion without the need for a fixed-point operator. Thus, the computational power of our language subset rests on the presence of while-loops, the semantics of which are given in terms of the least number of unfoldings of the loop body.

3 Proving Properties

We use preconditions and postconditions of functions and invariants of while-loops for program specifications. Additionally, as a means to express framing properties — i.e. to specify parts of the state that do not change— we use *assignment lists*. They resemble JML’s assignable-clauses [1], and are lists of expression patterns which evaluate to a set of locations that are allowed to be modified.

Very classically, total correctness of a function (or block of statements) means (i) the function (block) terminates for all states satisfying the precondition; (ii) every state satisfying the precondition is transformed by the semantics of the function (block) to a state satisfying the postcondition; (iii) all locations not in the set of modifiable locations have kept their initial values after program execution. Total correctness is derived through a Hoare-style calculus [4]. We have correctness assertions of the form $\Gamma \vdash_{stmt} [P] c [Q]$, where P and Q are *state predicates*, i.e. functions $\Sigma \rightarrow bool$. We decided to embed predicates shallowly because this gives us the full expressiveness of HOL without tying us to a specific specification language. As a consequence, the assignment

rule in our calculus (A) uses state change operations instead of syntactic substitution in the predicate as usual (B), a characteristic shared with e.g. Schirmer’s approach [6]:

$$(A) \quad \frac{\Gamma \vdash_{expr} [P] E [\lambda v S. Q (update (\Gamma x) v S)]}{\Gamma \vdash_{stmt} [P] x := E [Q]} \quad \Gamma \vdash_{stmt} [P[x/E]] x := E [P] \quad (B)$$

An important consequence is that we obtain a rather simple representation and uniform proof goals with regards to aliasing. Terms of the form $read\ l_1\ (update\ l_2\ v\ \Sigma)$ either simplify to v for $l_1 = l_2$, to $read\ l_1\ S$ for $l_1 \neq l_2$, or they lead to a case split for cases where the equality cannot be derived.

Practically, the calculus is used in a weakest precondition fashion. We formulated all proof rules so that they can automatically be applied by a proof procedure that finishes with a number of verification conditions. Isabelle’s metavariable mechanism is used to this end. For this to work, in all rules the preconditions in the premisses and the postcondition of the conclusion consists of a single metavariable. The verification conditions are then to be proved by a domain expert.

4 A Verification Environment

The encoding as sketched above forms the core of a verification environment built around it. A syntactic front-end is used to parse and typecheck the C programs, check conformance to the MISRA guidelines, and translate the abstract syntax into the Isabelle datatype; since this is mainly convenience and syntactic translation, it does not impinge on correctness.

The specifications are embedded into the program by using annotations as in JML or Caduceus [2], and translated along with the program. This way, programs and specifications are kept in sync automatically: we can either translate a given program to obtain a running program, or feed it through the verification tool.

References

- [1] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
- [2] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, Nov. 2004. Springer.
- [3] U. Frese, D. Hausmann, C. Lüth, H. Täubig, and D. Walter. The importance of being formal. In H. Hungar, editor, *International Workshop on the Certification of Safety-Critical Software Controlled Systems Safe-Cert’08*, To appear in *Electronic Notes in Theoretical Computer Science*, 2008.
- [4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [5] MISRA-C: 2004. Guidelines for the use of the C language in critical systems., 2004.
- [6] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 398–414. Springer, 2004.
- [7] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. The MIT Press, 1993.