Formal verification of C code at Radboud University Nijmegen, an overview

Robbert Krebbers Freek Wiedijk Marko van Eekelen Sjaak Smetsers Ken Madlener Dan Frumin Marc Schoolderman Freek Verbeek **Herman Geuvers**

Radboud University Nijmegen
The Netherlands

contents

- EUTypes COST Action CA15123
- The CH_2O project
- The Sovereign project

EUTypes

The European research network on types for programming and verification

The main objective:

> *To develop and use expressive type systems as a basis for improved programming techniques and for methods and tools to implement computer artifacts and verify them.*

Webpage

`eutypes.cs.ru.nl`

4 Working Groups

1. Theoretical Foundations
   Andrej Bauer (Slovenia)
2. Type-theoretic tools
   Assia Mahboubi (France)
3. Types for programming
   Andreas Abel (Sweden)
4. Types for verification
   Keiko Nakata (Germany)

- Chair: Herman Geuvers (Netherlands)
- Vice Chair: Tarmo Uustalu (Estonia)
- STSM Coordinator: Silvia Ghilezan (Serbia)
- Dissemination Coordinator: Aleksy Schubert (Poland)

WG1 Theoretical Foundations

- How to deal with isomorphic/equal structures? (HoTT, Homotopy Type Theory)
- Dependent type theory as an integrated environment for certified programming (WG2)
- Type-theoretic mechanisms to capture non-functional behavior of systems (WG3)

WG2 Type-theoretic tools

- Methods for high-level human computer communication
- Library reuse and modularity.
- Techniques for stronger proof automation (e.g. using machine learning)
- Deployment of advanced system architecture and parallelisation.

WG3 Types for programming

- Deployment in a concrete programming environment of type theories that capture other properties beyond functional correctness, for example, resource usage, matching communications, secure multi party computation, and modularity. (WG1)
- New strongly typed programming languages
- Program correctness by design, via type inference

WG4 Types for verification

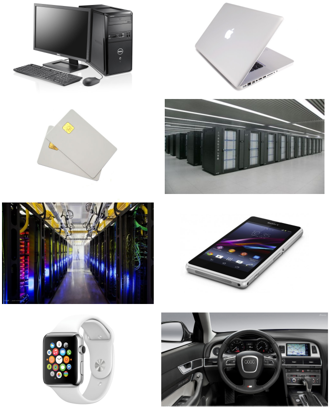
- Formalisation of industrial programming languages and their specification languages in different type based verification environments
- Proof automation techniques specific for the formal verification (verification condition generators, proof tactics etc.)
- High-level logics and type theories that make it easier to express and verify particular properties of interest in program verification.

Activities

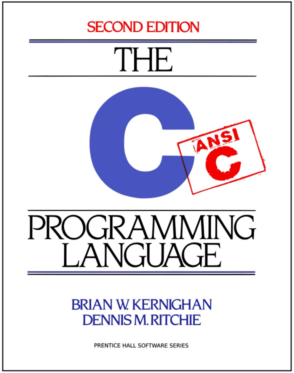
1. Plenary Meetings and WG meetings:
   - Types Conference Novi Sad (May 2016)
   - Joint meeting with Cost Betty (Oct 2016)
   - WG2,3,4 meeting co-located with POPL/CPP in Paris (Jan 2017)
   - Plenary MC meeting + WG1 meeting co-located with HoTT meeting at Ljubljana (Feb 2017)
2. Short-Term Scientific Missions:
   - 10-15 STSM per year, about 4 calls per year
3. Training (Summer) Schools: none in year 1, but one in every year 2, 3, 4.

why C?

performance / portability / control



=



the sweet spot between abstraction and concreteness

properties of C

- **performance**
- **portability**
- **control**

realized by a specific combination of

- **abstraction**
  underspecification

  allows compiler optimizations
  allows many architectures
- **concreteness**
  close to the hardware

  allows inline assembly
  explicit data representation as bytes

the four dimensions of software

let's compare C and Haskell...

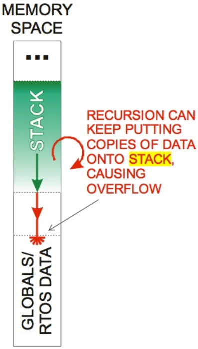
building a program is a trade-off between:

- **features**
  C gives more control
- **performance**
  C gives better performance
- **reliability**
  Haskell gives more reliability
- **cost**
  C is idiosyncratic, but easy
  Haskell needs monads: only for PhDs = expensive!

accelerating cars

SUA = sudden unintended acceleration

2005 Toyota Camry



embedded software

256.6K (non-comment) lines of C code
11,000 global variables

recursion $\Longrightarrow$ stack overflow
against the MISRA-C rules

stack is 94% full *plus* any recursion
incorrect assumption that overflow always results in a system reset
memory just past stack is OSEK RTOS area

four levels of applying formal methods to C

- **static analysis**
- static analysis + **annotations**

  model checkers
  SMT solvers
  automated theorem provers
- static analysis + annotations + **interactive proof**

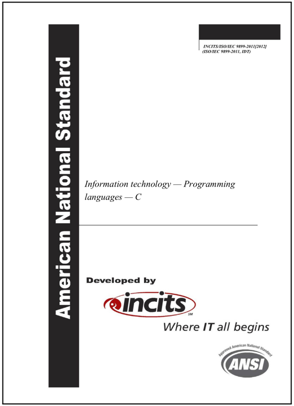
  proof assistants

  Why3/Jessie/Frama-C
  Jean-Christophe Filliâtre/Claude Marché, LRI, Paris France
- verification against **explicit semantics** inside a proof assistant

  Verified Software Toolchain Andrew Appel e.a., Princeton US

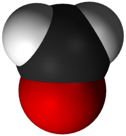
CH_2O

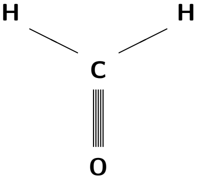
formalizing the C11 standard

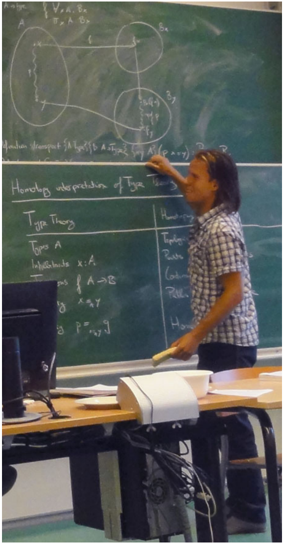


Robbert Krebbers and the formalin molecule

formalin $=$ CH_2O







- not *exactly* the standard
- only Coq
- executable semantics
- separation logic
- **metatheory**
  validate formal definitions

CompCert and related projects

- **CompCert** Xavier Leroy e.a., INRIA France

  compiler from C to x86/ARM/PowerPC
  implemented using Coq's functional language
  verified using using Coq's proof language

  CompCert C
  small step operational semantics

  does not match C11 as precisely as CH_2O
  CH_2O compliant (future work)

- **CompCertTSO** Jaroslav Ševčík, Viktor Vafeiadis, e.a.

- **Compositional CompCert** Gordon Stewart e.a., Princeton US

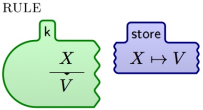
- **CerCo** Claudio Sacerdoti Coen e.a., Bologna Italy

Chucky Ellison's `kcc`

$\mathbb{K}$ = semantic framework built on rewriting logic

`kcc`
Chucky Ellison, A Formal Semantics of C with Applications
2012 PhD thesis with Grigore Roșu, Illinois US

executable semantics implemented in $\mathbb{K}$



does not match C11 as precisely as CH_2O
more features from C11 than CH_2O

`kcc`	aims to be a formal version of the C11 standard does *not* allow proof in a proof assistant
CompCert C	more specific than the C11 standard *does* allow proof in a proof assistant

Chung-Kil Hur's group

Seoul, Korea

Jeehoon Kang's PhD project

- **CompCertSep**

  separate compilation

- Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, Viktor Vafeiadis

  A Formal C Memory Model Supporting Integer-Pointer Casts

  PLDI 2015

Peter Sewell's group

Cambridge, UK

Kayvan Memarian's PhD project

- C memory **quiz**

  unfinished web survey that escaped to the web

- Kayvan Memarian, Justus Matthiesen, Kyndylan Nienhuis, Peter Sewell

  Cerberus,
  a semantic basis for sequential and concurrent C11

  unpublished paper that escaped to the web

less is more

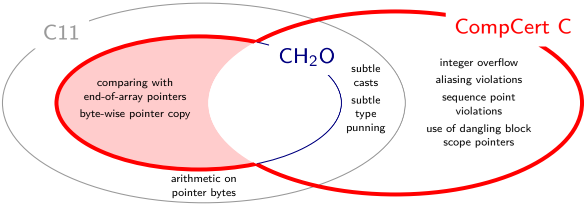
CH_2O compared to the C11 standard:

- fewer features
- fewer programs behave well

  in case of doubt: don't choose = everything may happen

  if one *can* prove a program behaves in a certain way, it should behave that way with *any* conforming compiler

CompCert C compared to C11 / CH_2O:



underspecification in the C standard

- **implementation-defined** behavior
  = you *do* know what will happen, but not from the standard
  each implementation documents how the choice is made

  number of bits in the integer types

- **unspecified** behavior
  = you do *not* know what will happen, but it will be reasonable

  order of evaluation of function arguments

- **undefined** behavior
  = you do not know *at all* what will happen = might crash

  accessing an array out-of-bounds
  signed integer overflow
  multiple updates to a variable in one statement

underspecification in the CH_2O formalization

- **implementation-defined** behavior
  semantics parametrized by an environment

```
Class IntCoding (K : Set) := {
  char_rank : K;
  char_signedness : signedness; ...
  char_bits : nat; ...
}.
Class IntEnv (K : Set) := {
  int_coding :> IntCoding K; ...
}.
Class Env (K : Set) := {
  env_type_env :> IntEnv K;
  size_of : env K → type K → nat;
  align_of : env K → type K → nat;
  field_sizes : env K → list (type K) → list nat;
  alloc_can_fail : bool
}.
```

- **unspecified** behavior
  non-determinism in the semantics
- **undefined** behavior = *is allowed to crash*
  undef states in the semantics

multiple updates in one statement

```
int x, y = (x = 3) + (x = 4);
printf("x = %d y =%d \n", x, y);
```

output:

```
x=4 y=7
```

- another "natural" output is `x=3 y=7`
- compiled with `gcc -O3` this prints `x=4 y=8` (!)
- anything is allowed, also a computer a crash.

`j` modified twice in the same statement $\Longrightarrow$ undefined behavior

not allowed to read or write a variable after writing it
between two sequence points

sequence point = boundary of expression evaluation

C11 standard, 6.5p2:

> *If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.*

C11 is inconsistent

the example from Defect Report 260

```
short a[2] = {6, 7}, b[2] = {-1, 9};
short *p = a + 2, *q = b;
if memcmp(&p, &q, sizeof(p)) == 0) {
  /* the bits of p and q are identical */
  printf("%d ", p == q);
  *q = 8;
  printf("%d %d\n", *p, *q);
}
```

compiled with `gcc -O3` this prints

0 8 -1

questions:

- is this very strange output allowed by the C11 standard? yes!
- is this program allowed to crash? yes!

who cares for a contrived example?

many examples

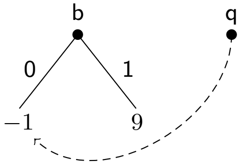
- end-of-array pointers
- unions (type punning)
- uninitialized memory (padding bytes in structs)
- dangling pointers

fundamental inconsistency between:

- **abstract** way of looking at data

  arrays, structs, unions
  effective types

  CH_2O: trees / paths in trees



- **concrete** way of looking at data

  unstructured, untyped

  CH_2O: lists of bits

11111111	11111111	10010000	00000000	00100100	01100110	10110110	11010110

the CH_2O memory model

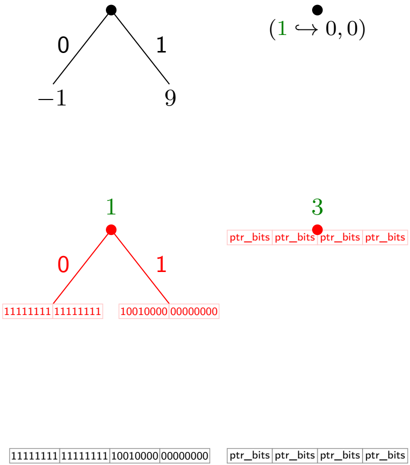
trees of lists of bits

- **values** in the semantics

- **memory values**

  memory $=$
  finite partial function
  from indexes
  to memory values

- **flattened** values
  (for unions)



the CH_2O semantics

two abstract variants of C

$$
\begin{array}{c}
\text{C source code} \\
\Big\downarrow {\scriptstyle \text{C} \,|\, \text{cpp}} \\
\text{preprocessed C} \\
\Big\downarrow {\scriptstyle \text{OCaml} \,|\, \text{parser from CIL}} \\
\text{CIL parse tree} \\
\Big\downarrow {\scriptstyle \text{OCaml}} \\
\mathsf{CH_2O\ abstract\ C} \\
\Big\downarrow {\scriptstyle \text{Coq} \dashrightarrow \text{OCaml}} \\
\mathsf{CH_2O\ core\ C}
\end{array}
$$

CH_2O counterpart to "CompCert C"

```c
struct s {char c; short h;} x;
int i;
int main() {
  int j = 0, k = j;
  return (j = (i = j++));
}
```

$(\texttt{"s"}, \mathsf{struct}\,((\mathsf{char}\,\texttt{"c"})\,(\mathsf{short}\,\texttt{"h"})))$
$(\texttt{"x"}, \mathsf{global} : \mathsf{struct}\,\texttt{"s"})$
$(\texttt{"i"}, \mathsf{global} : \mathsf{int})$
$(\texttt{"main"}, \mathsf{fun}\,($
$\quad \mathsf{int}\,\texttt{"j"} := (\mathsf{const}_{\mathsf{int}}\,0) ; \mathsf{int}\,\texttt{"k"} := \texttt{"j"};$
$\quad \mathsf{return}\,(\texttt{"j"} := (\texttt{"i"} := (\texttt{"j"} \mathrel{+:=} (\mathsf{const}_{\mathsf{int}}\,1))))$
$) : \epsilon \to \mathsf{int})$

$0 \mapsto (\mathsf{struct}_{\texttt{"s"}}\,(((\mathsf{base}_{\mathsf{signed\ char}}(\mathsf{full}, 0)^8)(\mathsf{full}, \lightning)^8)$
$\qquad ((\mathsf{base}_{\mathsf{signed\ short}}(\mathsf{full}, 0)^{16}))), \mathsf{false})$
$1 \mapsto (\mathsf{base}_{\mathsf{signed\ int}}(\mathsf{full}, 0)^{32}, \mathsf{false})$

$\texttt{"main"} \mapsto \mathsf{local}_{\mathsf{signed\ int}}\,(x_0 := [\mathsf{int}_{\mathsf{signed\ int}}\,0]_\emptyset ;$
$\quad (\mathsf{local}_{\mathsf{signed\ int}}\,(x_0 := \mathsf{load}\,x_1 ;$
$\quad (\mathsf{return}\,(x_1 := ([(1 : \mathsf{signed\ int}, \epsilon, 0)_{\mathsf{signed\ int} >_* \mathsf{signed\ int}}]_\emptyset$
$\qquad := (x_1 \mathrel{+:=} [\mathsf{int}_{\mathsf{signed\ int}}\,1]_\emptyset)))))))$

three variants of the CH_2O semantics

semantics of CH_2O abstract C

- **translation** to CH_2O core C

semantics of CH_2O core C

- **operational** semantics

  small step

- **executable** semantics

  'interpreter'

  extracted to OCaml $\longrightarrow$ standalone experimentation tool

- **axiomatic** semantics

  separation logic $\longrightarrow$ proving small programs correct

operational semantics: running around a zipper

state of the program $\mathcal{P}[s]$ in the semantics:

$$\mathbf{S}(\mathcal{P}[\square], (d, s), m)$$

(d, s)	running around the syntax tree
e	evaluating an expression
$\overline{\text{call}}\ f\ v$	calling a function
$\overline{\text{return}}\ f\ v$	returning from a function
$\overline{\text{undef}}\ \phi_U$	crashed!

s	statement that is the current focus		
$\mathcal{P}[\square]$	context of the statement, annotated with stack addresses		
d	direction of execution	$\searrow$	down
		$\nearrow$	up
		$\curvearrowright l$	executing goto l
		$\uparrow n$	breaking from a loop
		$\uparrow\uparrow v$	returning from a function
m	current state of the memory		
e	expression being evaluated		

calculates trace of sets of states
all the states, but modulo renumbering of memory indexes

axiomatic semantics: separation logic for C

multiple writes to a variable in one statement $\Longrightarrow$ undefined behavior

matches well with separation logic:

$$\frac{\{P_1\}\, e_1\, \{Q_1\} \quad \{P_2\}\, e_2\, \{Q_2\}}{\{P_1 * P_2\}\, e_1 \circledcirc e_2\, \{Q_1 * Q_2\}}$$

metatheory

- **the three kinds of semantics match**
  executable semantics: modulo renaming of indexes
  axiomatic semantics: not complete

  separation logic through shallow embedding

C typing:

$$\Gamma, \Delta, \vec{\tau} \vdash e : \tau_{\mathsf{lr}}$$
$$\Gamma \vdash \mathbf{S}(\mathcal{P}[\Box], \phi, m) : f_{\mathsf{main}}$$

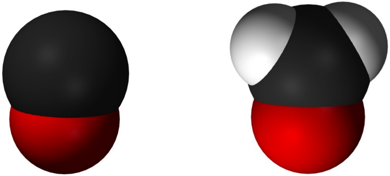
- **translation to CH_2O core C always type correct**
  (*if* the translation succeeds)

- **subject reduction** and **progress**

extend it?

- I/O and external functions
- multi-threading
- extend semantics to be closer to C11 standard
  - `exit`
  - untyped `malloc`
  - `setjmp` / `longjmp`
  - signals
  - floats
  - bitfields
  - variadic functions
  - variable-length arrays
  - `const`, `volatile`, `restrict`, *etc.*
  - header files and the preprocessor
  - *etc. etc.*
- stack overflow (for Toyota)

other proof assistants?



use it? improve it?

- verification condition generation
- static analysis++

- replace CIL parser with verified parser from CompCert
- formally show that CompCert C is an instance of CH_2O

STW project Sovereign

- Sovereign: A Framework for Modular Formal Verification of Safety Critical Software.
- PI: Marko van Eekelen. Co-applicants HG, Sjaak Smetsers, Freek Wiedijk.

Motto: *Scalability through modularity*
Verification should be

1. scalable (costs should not grow exceedingly as the size of the system increases),
2. compositional (global properties are directly inferrable from local properties of the subsystems),
3. incremental (verification can be performed iteratively while previous intermediate results are still usable),
4. effective (the proposed methodology will be applied successfully in some real-world case studies).

Sovereign project users

Companies involved

- **Rijkswaterstaat** RWS: Maeslantkering, tunnels, bridges

- **Nuclear Research Group** NRG: Borssele, Petten

Other companies (potentially) interested: NASA, TNO, ASML

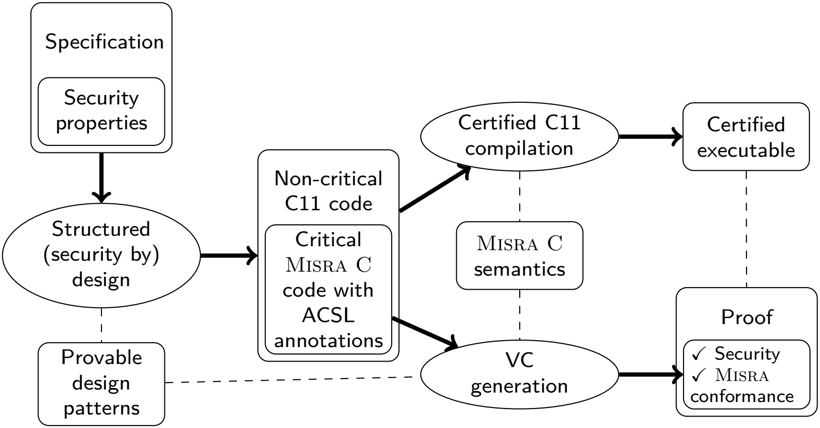


Figure: The Sovereign Framework

questions?