A Typed C11 Semantics for Interactive Theorem Proving

Robbert Krebbers Freek Wiedijk

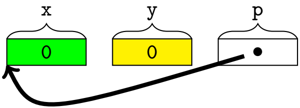
ICIS, Radboud University Nijmegen, The Netherlands

January 13, 2015 @ CPP, Mumbai, India

What is this C program supposed to do?

```
int x = 0, y = 0, *p = &x;
int f() { p = &y; return 17; }
int main() {
  *p = f();
  printf("x=%d,y=%d\n", x, y);
}
```

Initial state:



Let us try some compilers

- Clang prints `x=0,y=17`
  `f` is called first, thereafter `p` is evaluated to `&y`
- GCC prints `x=17,y=0`
  `p` is evaluated to `&x` first, then `f` is called

More subtle: `*p = (p = &y, 17);` has undefined behavior

Contribution

CH_2O (Krebbers & Wiedijk)

- Compiler independent C11 semantics in Coq
- Operational, executable, and axiomatic semantics

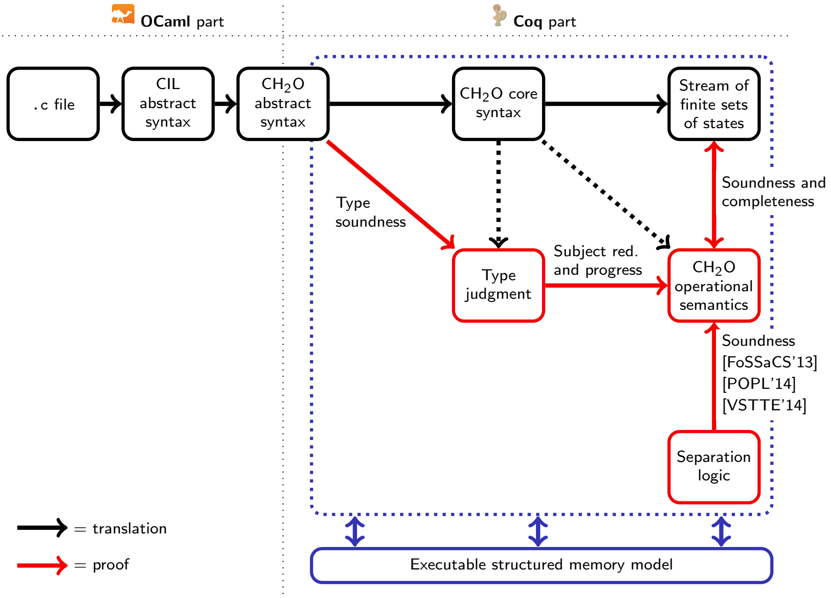
CPP'15 contribution: a verified interpreter to explore the non-deterministic behaviors of CH_2O

- Type system & *weak type safety*
- Executable semantics & *soundness/completeness*
- Formal translation from AST & *type soundness*

Recent related work

	CompCert	KCC	CH_2O
Compiler indep/close to C11	○	◐	●
Size of C fragment	●	●	◐
Proof assistant support	●	○	●
Type system	○	○	●
Principled core language	●	○	●
Formal translation from AST	○	n/a	●

Overview of the CH_2O project



CH_2O abstract C

$$
\begin{aligned}
k \in \mathsf{cintrank} &::= \mathsf{char} \mid \mathsf{short} \mid \mathsf{int} \\
&\mid \mathsf{long} \mid \mathsf{long\ long} \mid \mathsf{ptr} \\
si \in \mathsf{signedness} &::= \mathsf{signed} \mid \mathsf{unsigned} \\
\tau_i \in \mathsf{cinttype} &::= si^? \ k \\
\tau \in \mathsf{ctype} &::= \mathsf{void} \mid \mathsf{def}\ x \mid \tau_i \mid \tau * \\
&\mid \tau[e] \mid \mathsf{struct}\ x \mid \mathsf{union}\ x \\
&\mid \mathsf{enum}\ x \mid \mathsf{typeof}\ e \\
\alpha \in \mathsf{assign} &::= \ := \ \mid \circledcirc := \ \mid \ := \circledcirc \\
e \in \mathsf{cexpr} &::= x \mid \mathsf{const}_{\tau_i}\ z \mid \mathsf{sizeof}\ \tau \\
&\mid \tau_i\ \mathsf{min} \mid \tau_i\ \mathsf{max} \mid \tau_i\ \mathsf{bits} \\
&\mid \&e \mid *e \\
&\mid e_1\ \alpha\ e_2 \\
&\mid x(\vec{e}) \mid \mathsf{abort} \\
&\mid \mathsf{alloc}_\tau\ e \mid \mathsf{free}\ e \\
&\mid \circledcirc_u\ e \mid e_1 \circledcirc e_2 \\
&\mid e_1\ \&\&\ e_2 \mid e_1\ ||\ e_2 \\
&\mid e_1\ ?\ e_2 : e_3 \mid (e_1, e_2) \\
&\mid (\tau)\ I \mid e\ .\ x \\
r \in \mathsf{crefseg} &::= [e] \mid .x
\end{aligned}
$$

$$
\begin{aligned}
I \in \mathsf{cinit} &::= e \mid \{\overrightarrow{\vec{r} := I}\} \\
sto \in \mathsf{cstorage} &::= \mathsf{static} \mid \mathsf{extern} \mid \mathsf{auto} \\
s \in \mathsf{cstmt} &::= e \mid \mathsf{skip} \\
&\mid \mathsf{goto}\ x \mid \mathsf{return}\ e^? \\
&\mid \mathsf{break} \mid \mathsf{continue} \\
&\mid \{s\} \\
&\mid \overrightarrow{sto}\ \tau\ x := I^?\ ;\ s \\
&\mid \mathsf{typedef}\ x := \tau\ ;\ s \\
&\mid s_1\ ;\ s_2 \mid x : s \\
&\mid \mathsf{while}(e)\ s \\
&\mid \mathsf{for}(e_1\ ;\ e_2\ ;\ e_3)\ s \\
&\mid \mathsf{do}\ s\ \mathsf{while}(e) \\
&\mid \mathsf{if}\ (e)\ s_1\ \mathsf{else}\ s_2 \\
d \in \mathsf{decl} &::= \mathsf{struct}\ \overrightarrow{\tau\ x} \mid \mathsf{union}\ \overrightarrow{\tau\ x} \\
&\mid \mathsf{typedef}\ \tau \\
&\mid \mathsf{enum}\ \overrightarrow{x := e^?} : \tau_i \\
&\mid \mathsf{global}\ I^? : \overrightarrow{sto}\ \tau \\
&\mid \mathsf{fun}\ (\overrightarrow{\tau\ x^?})\ s^? : \overrightarrow{sto}\ \tau \\
\Theta \in \mathsf{decls} &:= \mathsf{list}\ (\mathsf{string} \times \mathsf{decl})
\end{aligned}
$$

CH_2O abstract C

Formal translation to core C

Conversions include:

- Named variables to De Bruijn indices
- Sound/complete constant expression evaluation, *e.g.* in $\tau[e]$
- Simplification of loops, *e.g.*

$$\text{while}(e)\ s \Rightarrow \text{catch (loop (if } (e) \text{ skip else throw } 0\,;\ \text{catch } s))$$

- Expansion of `typedef` and `enum` declarations
- Translation of constants like `INT_MIN`
- Translation of compound literals, *e.g.* `(struct S){ .x=1, {4,r}, .y[4+1]=0, q }`

Theorem (Type soundness)

The translator only produces well-typed CH_2O core programs

CH_2O operational semantics

- **Zippers** are used to describe non-local control flow
- **Structured memory model** (as separation algebra) to accurately describe low- versus high-level subtleties of C11
- **Permissions** (as separation algebra) are used for:
  - Ruling out expressions like `(x = 1) + (x = 2)`
  - Connection with separation logic
- **Evaluation contexts** for non-deterministic redex selection
- **Stuck states** for undefined behavior

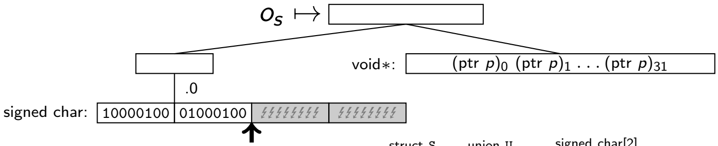
CH$_2$O operational semantics

Example of memory state

Consider:

```
struct S {
  union U {
    signed char x[2]; int y;
  } u;
  void *p;
} s = { { .x = {33,34} }, s.u.x + 2 }
```

The object in memory may look like:



$p = (o_S : \text{struct } \texttt{S}, \xhookrightarrow{\texttt{struct S}} 0 \xhookrightarrow{\texttt{union U}} \bullet\ 0 \xhookrightarrow{\text{signed char[2]}} 0, 16)_{\text{signed char} > \text{void}}$

Typing of CH_2O core C

Expression judgment $\Gamma, \Gamma_f, \Delta, \vec{\tau} \vdash e : \tau_{\mathbf{lr}}$

- Struct/union fields: $\Gamma \in \text{tag} \to_{\text{fin}} \text{list type}$
- Functions: $\Gamma_f \in \text{funname} \to_{\text{fin}} (\text{list type} \times \text{type})$
- Memory layout: $\Delta \in \text{index} \to_{\text{fin}} (\text{type} \times \text{bool})$
- De Bruijn variables: $\vec{\tau} \in \text{list type}$

For example:

$$\frac{\vec{\tau}(i) = \tau}{x_i^{\tau} : \tau_{\mathbf{l}}} \qquad \frac{e : \tau_{\mathbf{l}}}{\& e : (\tau *)_{\mathbf{r}}} \qquad \frac{\Gamma_f(f) = (\vec{\tau}, \sigma) \qquad \vec{e} : \vec{\tau}_{\mathbf{r}}}{f(\vec{e}) : \sigma_{\mathbf{r}}}$$

Statement judgment $\Gamma, \Gamma_f, \Delta, \vec{\tau} \vdash s : (\beta, \tau^{?})$

$$\frac{}{\text{skip} : (\text{false}, \bot)} \qquad \frac{e : \tau_{\mathbf{r}}}{\text{return } e : (\text{true}, \tau)} \qquad \frac{}{\text{goto } l : (\text{true}, \bot)}$$

State judgment $\Gamma, \Gamma_f, \Delta \vdash S : g$ (typically $g = \texttt{main}$)

Typing of CH_2O core C

Type preservation

Lemma (Type preservation)

If $S_1 : g$ and $S_1 \rightharpoonup S_2$, then $S_2 : g$

Theorem (Weak type safety)

If S_1 initial for $g(\vec{v})$, then if $S_1 \rightharpoonup^* S_2$ we have either:

1. Not finished: $S_2 \rightharpoonup S_3$ for some S_3
2. Undefined behavior: $S_2 = \mathbf{S}(\mathcal{P}, \overline{\text{undef}}\, \phi_U, m)$
3. Final state: $S_2 = \mathbf{S}(\epsilon, \overline{\text{return}}\, g\, v, m)$

Executable semantics

Goal: define $\mathsf{exec} : \mathsf{state} \to \mathcal{P}_{\mathsf{fin}}(\mathsf{state})$ and extract to OCaml

Problems:

1. Decomposition $\mathcal{E}[\, e_1 \,]$ of expressions is non-deterministic:

$$\mathbf{S}(\mathcal{P},\, \mathcal{E}[\, e_1 \,],\, m_1) \rightharpoondown \mathbf{S}(\mathcal{P},\, \mathcal{E}[\, e_2 \,],\, m_2) \quad \text{if} \quad (e_1, m_1) \rightharpoondown_{\mathsf{h}} (e_2, m_2)$$

2. Object identifiers o for newly allocated memory are arbitrary:

$$\begin{aligned} &\mathbf{S}(\mathcal{P},\, (\searrow,\, \mathsf{local}_{\tau}\, s),\, m) \\ \rightharpoondown\ &\mathbf{S}((\mathsf{local}_{o:\tau}\, \square)\, \mathcal{P},\, (\searrow,\, s),\, \mathsf{alloc}_{\Gamma}\, o\, \tau\, \mathsf{false}\, m) \quad \text{if} \quad o \notin \mathsf{dom}\, m \end{aligned}$$

Solutions:

1. Enumerate all possible decompositions $\mathcal{E}[\, e_1 \,]$
2. Pick a canonical object identifier $\mathsf{fresh}\; m$ for o (makes completeness difficult!)

Executable semantics

Soundness and completeness

Theorem (Soundness)

If $S_2 \in \mathrm{exec}\ S_1$, then $S_1 \rightarrowtriangle S_2$

Definition (Permutation)

We let $S_1 \sim_f S_2$, if S_2 is obtained by renaming S_1 with respect to $f : \mathrm{index} \to \mathrm{option\ index}$

Theorem (Completeness)

If $S_1 \rightarrowtriangle^* S_2$, then there exists an f and S_2' such that:

$$\begin{tikzcd}
 & & S_2' \arrow[d, dash, dashed, "\sim_f"] \\
S_1 \arrow[urr, dashed, "\mathrm{exec}", "*"'] \arrow[rr, "*"] & & S_2
\end{tikzcd}$$

Formalization in Coq

Interpreter extracted to OCaml from Coq

- **Error monad** for failure of type checking
- **Set monad** for non-determinism
- Verified **hash sets** for efficiency

All essential properties proven in Coq:

- Weak type safety
- Soundness and completeness of executable semantics
- Type soundness of translation from AST

Part of ∼40.000 LOC constructive and axiom free development

Conclusion

A programming language semantics should consist of:

- **Operational semantics**
  Reasoning about program transformations
- **Axiomatic semantics**
  Correctness proofs of concrete programs
- **Executable semantics**
  Debugging and testing

Extremely challenging to develop matching versions for C11

Future work: still many parts of C11 left to be explored

Demo and questions

Sources: `http://robbertkrebbers.nl/research/ch2o/`