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?

```c
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);
}
```

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$_2$O (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$_2$O
- Type system & weak type safety
- Executable semantics & soundness/completeness
- Formal translation from AST & type soundness
## Recent related work

<table>
<thead>
<tr>
<th>Feature</th>
<th>CompCert</th>
<th>KCC</th>
<th>CH$_2$O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler indep/close to C11</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Size of C fragment</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Proof assistant support</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Type system</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Principled core language</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Formal translation from AST</td>
<td>○</td>
<td>n/a</td>
<td>●</td>
</tr>
</tbody>
</table>
Overview of the CH$_2$O project

**OCaml part**

- `.c file` → CIL abstract syntax → CH$_2$O abstract syntax → CH$_2$O core syntax → Stream of finite sets of states

- Type soundness

**Coq part**

- CH$_2$O operational semantics

- Subject red. and progress

- Soundness and completeness

**Soundness**

- [FoSSaCS'13]
- [POPL'14]
- [VSTTE'14]

**Separation logic**

**Executable structured memory model**

= translation

= proof
\[
\begin{align*}
k \in \text{cintrank} & := \text{char} | \text{short} | \text{int} \\
& \quad | \text{long} | \text{long long} | \text{ptr} \\
si \in \text{signedness} & := \text{signed} | \text{unsigned} \\
\tau_i \in \text{cintype} & := si? k \\
\tau \in \text{ctype} & := \text{void} | \text{def} x | \tau_i | \tau \ast \\
& \quad | \tau[e] | \text{struct} x | \text{union} x \\
& \quad | \text{enum} x | \text{typeof} e \\
\alpha \in \text{assign} & := := | \odot := | := \odot \\
e \in \text{cexpr} & := x | \text{const}_{\tau_i} z | \text{sizeof} \ \tau \\
& \quad | \tau_i \text{min} | \tau_i \text{max} | \tau_i \text{bits} \\
& \quad | \&e | \ast e \\
& \quad | e_1 \alpha e_2 \\
& \quad | x(\vec{e}) | \text{abort} \\
& \quad | \text{alloc}_\tau e | \text{free} e \\
& \quad | \odot_u e | e_1 \odot e_2 \\
& \quad | e_1 \&\& e_2 | e_1 \mid\mid e_2 \\
& \quad | e_1 ? e_2 : e_3 | (e_1, e_2) \\
& \quad | (\tau) \ ! | e \cdot x \\
r \in \text{crefseg} & := [e] | .x \\
l \in \text{cinit} & := e | \{ \vec{r} := \vec{l} \} \\
sto \in \text{cstorage} & := \text{static} | \text{extern} | \text{auto} \\
s \in \text{cstmt} & := e | \text{skip} \\
& \quad | \text{goto} x | \text{return} e? \\
& \quad | \text{break} | \text{continue} \\
& \quad | \{s\} \\
& \quad | \sto \tau x := l? \ ; s \\
&t\text{ypedef} x := \tau \ ; s \\
& \quad | s_1 \ ; s_2 | x : s \\
& \quad | \text{while}(e) s \\
& \quad | \text{for}(e_1 ; e_2 ; e_3) s \\
& \quad | \text{do} s \text{ while}(e) \\
& \quad | \text{if} (e) s_1 \text{ else} s_2 \\
d \in \text{decl} & := \text{struct} \tau x | \text{union} \tau x \\
& \quad | \text{typedef} \tau \\
& \quad | \text{enum} x := e? : \tau_i \\
& \quad | \text{global} l? : \sto \tau \\
& \quad | \text{fun}(\tau x?) s? : \sto \tau \\
\Theta \in \text{decls} & := \text{list (string} \times \text{decl})
\end{align*}
\]
CH$_2$O 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.

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

- Expansion of `typedef` and `enum` declarations
- Translation of constants like `INT_MIN`
- Translation of compound literals, e.g.

  \[
  \text{(struct } S)\{\ .x=1,\ \{4,r\},\ .y[4+1]=0,\ q\ \}
  \]

**Theorem (Type soundness)**
The translator only produces well-typed CH$_2$O core programs
CH$_2$O 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
Consider:

```c
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:

```
op = (oS : struct S, struct S)0 union U0 signed char[2]0, 16)signed char>void
```

```
signed char: 10000100 01000100 EEEEEEEE EEEEEEEE
```

```
void*: (ptr p)0 (ptr p)1 . . . (ptr p)31
```

```
op = (oS : struct S, struct S)0 union U0 signed char[2]0, 16)signed char>void
```
Typing of \( \text{CH}_2\text{O} \) core C

**Expression judgment** \( \Gamma, \Gamma_f, \Delta, \vec{\tau} \vdash e : \tau_r \)

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

For example:

\[
\begin{align*}
\vec{\tau}(i) &= \tau \\
\vec{\tau} &\vdash e : \tau_l \\
\vec{\tau} &\vdash \Gamma_f(f) = (\vec{\tau}, \sigma) \\
\vec{\tau} &\vdash f(\vec{e}) : \sigma_r
\end{align*}
\]

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

- \( \vec{\tau} \vdash e : \tau_r \)
- \( \vec{\tau} \vdash \text{skip} : (\text{false}, \bot) \)
- \( \vec{\tau} \vdash \text{return } e : (\text{true}, \tau) \)
- \( \vec{\tau} \vdash \text{goto } l : (\text{true}, \bot) \)

**State judgment** \( \Gamma, \Gamma_f, \Delta \vdash S : g \) \quad (typically \( g = \text{main} \))
Typing of CH$_2$O core C

Type preservation

Lemma (Type preservation)
If $S_1 : g$ and $S_1 \to S_2$, then $S_2 : g$

Theorem (Weak type safety)
If $S_1$ initial for $g(\vec{v})$, then if $S_1 \to^* S_2$ we have either:

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

**Goal:** define \( \text{exec} : \text{state} \rightarrow \mathcal{P}_{\text{fin}}(\text{state}) \) and extract to OCaml

**Problems:**

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

\[
\mathcal{S}(\mathcal{P}, \mathcal{E}[e_1], m_1) \rightarrow \mathcal{S}(\mathcal{P}, \mathcal{E}[e_2], m_2) \text{ if } (e_1, m_1) \rightarrow_h (e_2, m_2)
\]

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

\[
\mathcal{S}(\mathcal{P}, (\llcorner, \text{local}_\tau s), m) \\
\rightarrow \mathcal{S}((\text{local}_{o:\tau} \square) \mathcal{P}, (\llcorner, s), \text{alloc}_\Gamma o \tau \text{ false } m) \text{ if } o \notin \text{dom } m
\]

**Solutions:**

1. Enumerate all possible decompositions \( \mathcal{E}[e_1] \)

2. Pick a canonical object identifier \( \text{fresh } m \) for \( o \) (makes completeness difficult!)
Executable semantics
Soundness and completeness

**Theorem (Soundness)**
If $S_2 \in \text{exec } S_1$, then $S_1 \rightarrow S_2$

**Definition (Permutation)**
We let $S_1 \sim_f S_2$, if $S_2$ is obtained by renaming $S_1$ with respect to $f : \text{index} \rightarrow \text{option index}$

**Theorem (Completeness)**
If $S_1 \rightarrow^* S_2$, then there exists an $f$ and $S'_2$ such that:

\[
S_1 \quad \text{exec} \quad * \quad S'_2
\]

\[
S_2
\]

\[
\sim_f
\]
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 \(\sim 40.000\) 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/