Isabelle_C: A Generic Front-End of C11 Supported in Isabelle/PIDE

Frédéric Tuong* and Burkhart Wolff**
(*) Trinity College Dublin, Ireland
(**) LRI, Université Paris-Saclay, France

Isabelle Workshop @ IJCAR 2020, Paris, France
• Code-Verification - A Solved Problem?
OVERVIEW

- Code-Verification - A Solved Problem?
- Using Isabelle as Code-Verification Framework
OVERVIEW

• Code-Verification - A Solved Problem ?

• Using Isabelle as Code-Verification Framework
  • the IDE (called PIDE)
  • Generic Front-End - Generic Annotations …
  • Re-using existing semantic Back-Ends
OVERVIEW

• Code-Verification - A Solved Problem ?

• Using Isabelle as Code-Verification Framework
  
  • the IDE (called PIDE)
  
  • Generic Front-End - Generic Annotations …
  
  • Re-using existing semantic Back-Ends

• DEMO: Semantic Backends: CLean, AutoCorres
CODE-VERIFICATION - A SOLVED PROBLEM ?
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”).

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus.
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in Frama-C

CODE-VERIFICATION - A SOLVED PROBLEM?
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in Frama-C

```c
int linearsearch(int x, int t[], int n) {
    int i = 0;
    /*@ loop invariant \forall integer k; 0 <= k < i ==> t[k] < x; @ loop assigns i; @ loop variant n-i; */
    while (i < n) {
        if (t[i] < x) { i++; }
        else { return (t[i] == x); }
    }
    return 0;
}
```

CODE-VERIFICATION - A SOLVED PROBLEM ?

/* @ requires n>= 0 && \valid(t+0..n-&) @ requires \forall integer k,l; 0 <= k <= l < n ==> t[k] <= t[l]; @ ensures \result <==> (\exists integer k; 0 <= k < n && t[k] == x); @ assigns \nothing; */
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in Frama-C

```c
/* @ requires n>= 0 && \valid(t+0..n-\&) 
@ requires \forall integer k,l; 0 <= k <= l < n ==> t[k] <= t[l]; 
@ ensures \result <==> (\exists integer k; 0 <= k < n && t[k] == x)); 
@ assigns \nothing; */

int linearsearch(int x, int t[], int n) {
  int i = 0;
  /*@ loop invariant 0 <= i <= n; 
  @ loop invariant (\forall integer k; 0 <= k < i ==> t[k] < x); 
  @ loop assigns i; 
  @ loop variant n-i; */
  while (i < n) {
    if (t[i] < x) { i++; } 
    else { return (t[i] == x); }
  }
  return 0;
}
```
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is "the real thing")

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in Frama-C

```c
int linearsearch(int x, int t[], int n) {
    int i = 0;
    /*@ loop invariant 0 <= i <= n; */
    while (i < n) {
        if (t[i] < x) { i++; }
        else { return (t[i] == x); }
    }
    return 0;
}
```

CODE-VERIFICATION - A SOLVED PROBLEM?
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in Frama-C

```c
int linearsearch(int x, int t[], int n) {
    int i = 0;
    /*@ loop invariant 0 <= i <= n; */
    @ loop invariant (\forall integer k; 0 <= k < i ==> t[k] < x);
    @ loop invariant (\forall integer k; 0 <= k < n && t[k] == x);
    @ assigns \nothing; */
    while (i < n) {
        if (t[i] < x) { i++; }
        else { return (t[i] == x); }
    }
    return 0;
}
```
CODE-VERIFICATION - A SOLVED PROBLEM ?
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”).

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus.

Example in VCC-3
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in VCC-3

```c
struct Handle {
    obj_t obj;
    invariant(obj->handles[this] && closed(obj))
};

struct Data {
    bool handles[Handle*];
    invariant(forall(Handle *h; closed(h) ==> handles[h] <=> h->obj == this))
    invariant(old(closed(this)) && !closed(this) ==> !exists(Handle *h; handles[h]))
    invariant(is_thread(owner(this)) || old(handles) == handles || inv2(owner(this)))
};
```
Annotating programming code with pre-post-conditions and invariants is a popular Formal Method (for some, it is “the real thing”)

Annotations were inserted in a verification condition generator (VCG) executing a Dijkstra wp calculus

Example in VCC-3

```c
struct Handle {
    obj_t obj;
    invariant(obj->handles[thist] && closed(obj))
};
struct Data {
    bool handles[Handle*];
    invariant(forall(Handle *h; closed(h) ==> handles[h] <=> h->obj == this))
    invariant(old(closed(this)) && !closed(this) ==> !exists(Handle *h; handles[h]))
    invariant(is_thread(owner(this)) || old(handles) == handles || inv2(owner(this)))
};
```
CODE-VERIFICATION - A SOLVED PROBLEM ?
• When it comes to REAL programming languages, VCG’s make assumptions over
When it comes to REAL programming languages, VCG’s make assumptions over

- the language fragment
When it comes to REAL programming languages, VCG’s make assumptions over:

- the language fragment
- the treatment of language underspecification (execution order, determinism, arithmetic, …)
When it comes to REAL programming languages, VCG’s make assumptions over

- the language fragment
- the treatment of language underspecification (execution order, determinism, arithmetic, … )
- the execution model (sequential ? sequentially consistent? concurrent? )
When it comes to REAL programming languages, VCG’s make assumptions over

- the language fragment

- the treatment of language underspecification (execution order, determinism, arithmetic, …)

- the execution model (sequential ? sequentially consistent? concurrent? )

- the data-types (arithmetic ? union-types ? floats ? …)
When it comes to REAL programming languages, VCG’s make assumptions over

- the language fragment
- the treatment of language underspecification (execution order, determinism, arithmetic, … )
- the execution model (sequential ? sequentially consistent? concurrent? )
- the data-types (arithmetic ? union-types ? floats ? … )
- the memory and architecture model (global ? local ? physical ? byte-level layout ? … )
CODE-VERIFICATION - A SOLVED PROBLEM ?
CODE-VERIFICATION - A SOLVED PROBLEM?

- ... all these assumptions were kept
CODE-VERIFICATION - A SOLVED PROBLEM?

- ... all these assumptions were kept
  - either implicit in the VCG algorithm
... all these assumptions were kept

- either implicit in the VCG algorithm
- or explicit in the “background theory”

(VCC(1) : 300 axioms ...)
CODE-VERIFICATION - A SOLVED PROBLEM?
CODE-VERIFICATION - A SOLVED PROBLEM ?

- Most Existing Tools
  (STEPS, STEP2, VCC 1 - 3, Daphne, SAL-Annotations, why, Frama-C, ...)
  follow this “axiomatic” approach
CODE-VERIFICATION - A SOLVED PROBLEM?

• Most Existing Tools
  (STEPS, STEP2, VCC 1 - 3, Daphne, SAL-Annotations, why, Frama-C, ...)
  follow this “axiomatic” approach

• Alternative: Using Logical Embeddings in an Interactive Theorem Prover like Coq or Isabelle.
CODE-VERIFICATION -
A SOLVED PROBLEM ?

• Most Existing Tools
  (STEPS, STEP2, VCC 1 - 3, Daphne, SAL-Annotations, why, Frama-C, …)
follow this “axiomatic” approach

• Alternative: Using Logical Embeddings in an Interactive Theorem Prover like Coq or Isabelle.
  • Derived VCG,
  • Derived, guaranteed consistent memory model
  • Clear Management of the involved Logical Contexts
CODE-VERIFICATION - A SOLVED PROBLEM?

• Most Existing Tools
  (STEPS, STEP2, VCC 1 - 3, Daphne, SAL-Annotations, why, Frama-C, …)
  follow this “axiomatic” approach

• Alternative: Using Logical Embeddings in an Interactive Theorem Prover like Coq or Isabelle.
  • Derived VCG,
  • Derived, guaranteed consistent memory model
  • Clear Management of the involved Logical Contexts
  • … but still no guarantee that the model meets reality ;-)

WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Generic Front-End for C-Semantics in Isabelle/HOL
WHY ISABELLE AS PLATFORM
THE "ECLIPSE OF FORMAL METHODS TOOLS"

• Generic Front-End for C-Semantics in Isabelle/HOL

• Isabelle as a System Framework offers an IDE (PIDE) allowing programmable commands in a generic document model
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

- Generic Front-End for C-Semantics in Isabelle/HOL
- Isabelle as a System Framework offers an IDE (PIDE) allowing programmable commands in a generic document model
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Generic Front-End for C-Semantics in Isabelle/HOL

• Isabelle as a System Framework offers an IDE (PIDE)
  allowing programmable commands in a generic document model
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Generic Front-End for C-Semantics in Isabelle/HOL
  • Isabelle as a System Framework offers an IDE (PIDE) allowing programmable commands in a generic document model
  • Several formally proven consistent semantic “back-ends” can give different semantic interpretations for C-programs
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Generic Front-End for C-Semantics in Isabelle/HOL
  • Isabelle as a System Framework offers an IDE (PIDE) allowing programmable commands in a generic document model
  • Several formally proven consistent semantic “back-ends” can give different semantic interpretations for C-programs
  • Semantic Annotations were interpreted “back-end-specific” with logical context, C-scope and C term-context
WHY ISABELLE AS PLATFORM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Generic Front-End for C-Semantics in Isabelle/HOL

  • Isabelle as a System Framework offers an IDE (PIDE) allowing programmable commands in a generic document model

  • Several **formally proven consistent** semantic “back-ends” can give different semantic interpretations for C-programs

  • Semantic Annotations were interpreted “back-end-specific” with logical context, C-scope and C term-context

  • Generic Front-End can create different applications based on symbolic execution, test-generation and interactive and automated proofs
OUR SOLUTION

ISABELLE/C - A FRAMEWORK FOR C-TOOLS
OUR SOLUTION

ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:
OUR SOLUTION
ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:

```c
#include <stdio.h>

int main()
{
    int array[100], n, c, d, position, swap;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++) scanf("%d", &a[c]);
    for (c = 0; c < (n - 1); c++)
    {
        position = c;
    }
```
OUR SOLUTION
ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:

• Fully editable, IDE support
OUR SOLUTION
ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:

  • Fully editable, IDE support
  • navigation

```c
#include <stdio.h>

int main()
{
    int array[100], n, c, d, position, swap;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (c = 0; c < n; c++)
    {
        scanf("%d", &array[c]);
    }

    for (c = 0; c < (n - 1); c++)
    {
        position = c;
    }
```
OUR SOLUTION

ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:

  • Fully editable, IDE support
  • navigation
  • C11 syntax
OUR SOLUTION
ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• A new set of commands, most notably the new C-command inside PIDE:

• Fully editable, IDE support

• navigation

• C11 syntax

• Generic, programmable Annotations
DEMO

Isabelle/C and Isabelle/C/AutoCorres
BACKENDS
• At present, the following Logical Embeddings are possible targets for C11:
BACKENDS

• At present, the following Logical Embeddings are possible targets for C11:
  
• SIMPL, IMP [N. Schirmer 2004, T. Nipkow 1999]
At present, the following Logical Embeddings are possible targets for C11:

- SIMPL, IMP [N. Schirmer 2004, T. Nipkow 1999]
- IMP2 [P. Lammich 2019]
BACKENDS

- At present, the following Logical Embeddings are possible targets for C11:
  - SIMPL, IMP [N. Schirmer 2004, T. Nipkow 1999]
  - IMP2 [P. Lammich 2019]
  - ORCA [Y. Nemouchi @ al, 2018]
• At present, the following Logical Embeddings are possible targets for C11:
  
  • SIMPL, IMP [N. Schirmer 2004, T. Nipkow 1999]
  
  • IMP2 [P. Lammich 2019]
  
  • ORCA [Y. Nemouchi @ al, 2018]
  
  • CLEAN [Keller @ al, 2018], for White-Box Testing
BACKENDS

At present, the following Logical Embeddings are possible targets for C11:

- SIMPL, IMP [N. Schirmer 2004, T. Nipkow 1999]
- IMP2 [P. Lammich 2019]
- ORCA [Y. Nemouchi @ al, 2018]
- CLEAN [Keller @ al, 2018], for White-Box Testing
- AutoCorres [Klein @ al, 2014]
• AutoCorres [Klein @ al, 2014] realistic C model for OS code verification used in seL4 project, decent automation support, but complex.
OUR SOLUTION
ISABELLE/C - A FRAMEWORK FOR C-TOOLS

• Construction by Compiler-Generators
  (and not general-purpose, inner-syntax “Early-Parser” for λ-terms)

• Efficient parsing, and Intellisense

• Generic Scope-Analysis

• parses entire seL4 sources
  (26 kLoC in 1s)

• markup in 20 s
HOOKING UP BACKENDS
• Semantics of a Command:
HOOKING UP BACKENDS

• Semantics of a Command:

• Problems:
HOOKING UP BACKENDS

- Semantics of a Command:

- Problems:
  - ambiguity
HOOKING UP BACKENDS

- Semantics of a Command:

- Problems:
  
  - ambiguity

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

• Semantics of a Command:

• Problems:

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

  • format flexibility
    
    /*@ annotation_begin */ ... /*@ annotation_end */
HOOKING UP BACKENDS

• Semantics of a Command:

• Problems:

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

  • format flexibility
    /*@ annotation_begin */ ... /*@ annotation_end */

  • reference to the syntactic C Context
    /*@ assert ⟨a > i⟩ */
HOOKING UP BACKENDS

• Semantics of a Command:

• Problems:

  • ambiguity
    
      for (int i = 0; i < n; i++) a+= a*i /*@ annotation */
    
  • format flexibility
    
      /*@ annotation_begin */ ... /*@ annotation_end */
    
  • reference to the syntactic C Context
    
      /*@ assert ⟨a > i⟩ */
    
  • transformation of the logical context
HOOKING UP BACKENDS

- Semantics of a Command:

- Problems:
  - ambiguity
    ```c
    for (int i = 0; i < n; i++) a+= a*i /*@ annotation */
    ```
  - format flexibility
    ```c
    /*@ annotation_begin */ ... /*@ annotation_end */
    ```
  - reference to the syntactic C Context
    ```c
   /*@ assert ⟨a > i⟩ */
    ```
  - transformation of the logical context

```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)*/
```
HOOKING UP BACKENDS

• Semantics of a Command:

• Problems:

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

  • format flexibility
    ```
    /*@ annotation_begin */ ... /*@ annotation_end */
    ```

  • reference to the syntactic C Context
    ```
   /*@ assert ⟨a > i⟩ */
    ```

  • transformation of the logical context

  • scheduling the evaluation order

```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)) */
```
HOOKING UP BACKENDS
HOOKING UP BACKENDS

• General Mechanism to register a PIDE “command”:

\[ \text{Outer\_Syntax.command'}: K_{cmd} \rightarrow (\sigma \rightarrow \sigma) \text{ parser } \rightarrow \sigma \rightarrow \sigma \]
HOOKING UP BACKENDS

• General Mechanism to register a PIDE “command”:

   Outer_Syntax.command’: $K_{cmd} \rightarrow (\sigma \rightarrow \sigma)$ parser \rightarrow \sigma \rightarrow \sigma

• Isabelle/Isar: “setup ⟨ some sml ⟩”
HOOKING UP BACKENDS

• General Mechanism to register a PIDE "command":

\[ \text{Outer\_Syntax.command'}: K_{cmd} \rightarrow (\sigma \rightarrow \sigma) \text{ parser} \rightarrow \sigma \rightarrow \sigma \]

• Isabelle/Isar: "setup \(\langle\text{some sml}\rangle\)"

• Analogously, Isabelle/C provides an infrastructure to define "Annotation Commands"
HOOKING UP BACKENDS

• General Mechanism to register a PIDE "command":

\[
\text{Outer\_Syntax.command': } K_{cmd} \rightarrow (\sigma \rightarrow \sigma) \text{ parser} \rightarrow \sigma \rightarrow \sigma
\]

• Isabelle/Isar: "setup \langle \text{some sml} \rangle"

• Analogously, Isabelle/C provides an infrastructure to define "Annotation Commands"

\[
\text{C\_Annotation.command : } K_{cmd} \rightarrow (<n\text{-expr}> \rightarrow (\sigma \rightarrow \sigma)c_{parser}) \rightarrow \text{unit}
\]
\[
\text{C\_Annotation.command': } K_{cmd} \rightarrow (<n\text{-expr}> \rightarrow (\sigma \rightarrow \sigma)c_{parser}) \rightarrow \sigma \rightarrow \sigma
\]
HOOKING UP BACKENDS

• General Mechanism to register a PIDE “command”:

  Outer_Syntax.command': \( K_{cmd} \rightarrow (\sigma \rightarrow \sigma) \) parser \rightarrow \sigma \rightarrow \sigma

• Isabelle/Isar: “setup \( \langle \text{some sml} \rangle \)”

• Analogously, Isabelle/C provides an infrastructure to define “Annotation Commands”

  C_Annotation.command : \( K_{cmd} \rightarrow (\langle n\text{-expr} \rangle \rightarrow (\sigma \rightarrow \sigma)c\_parser) \rightarrow \text{unit} \)

  C_Annotation.command' : \( K_{cmd} \rightarrow (\langle n\text{-expr} \rangle \rightarrow (\sigma \rightarrow \sigma)c\_parser) \rightarrow \sigma \rightarrow \sigma \)

• ... \( \sigma \) is the logical context of the Isabelle system
HOOKING UP BACKENDS

- General Mechanism to register a PIDE “command”:

\[
\text{Outer}\_\text{Syntax.command'}: K_{\text{cmd}} \rightarrow (\sigma \rightarrow \sigma) \text{ parser} \rightarrow \sigma \rightarrow \sigma
\]

- Isabelle/Isar: “setup \{ some sml \}”

- Analogously, Isabelle/C provides an infrastructure to define “Annotation Commands”

\[
\text{C Annotation.command}: K_{\text{cmd}} \rightarrow (\text{<n-expr>} \rightarrow (\sigma \rightarrow \sigma)c_{\text{parser}}) \rightarrow \text{unit}
\]

\[
\text{C Annotation.command'}: K_{\text{cmd}} \rightarrow (\text{<n-expr>} \rightarrow (\sigma \rightarrow \sigma)c_{\text{parser}}) \rightarrow \sigma \rightarrow \sigma
\]

- … \( \sigma \) is the logical context of the Isabelle system

- … comprising in Isabelle/C an environment \text{env} and the stack of S-R- AST’s
Hooking up Backends
Hooking up Backends

• Navigation:
Hooking up Backends

- Navigation:

```c
int sum1(int a)
{
    while (a < 10)
        a = a + 1;
    return a;
}
```

```c
int sum2(int a)
/*@ ++@ INV: ...
++@ highlight */
{
    while (a < 10)
        a = a + 1;
    return a;
}
```
Hooking up Backends

• Navigation:

```c
int sum1(int a)
{
    while (a < 10)
        /*@ @ INV: <...> @ highlight */
        { a = a + 1; }
    return a;
}
```

```c
int sum2(int a)
/*@ @@ INV: <...> @@ highlight */
{
    while (a < 10)
        { a = a + 1; }
    return a;
}
```

• Scheduling:
Hooking up Backends

- Navigation:

```c
int sum1(int a) {
    while (a < 10) {
        @ highlight */
        { a = a + 1; }
    }
    return a;
}
```

- Scheduling:

```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↓"
*/
```
HOOKING UP BACKENDS
HOOKING UP BACKENDS

- Example Language Clean (called “C Lean”)
HOOKING UP BACKENDS

• Example Language Clean (called “C Lean”)

• Built upon a shallow embedding into State-Exception Monads
HOOING UP BACKENDS

- Example Language Clean (called "C Lean")
- Built upon a shallow embedding into State-Exception Monads
- Minimalistic language with
HOOKING UP BACKENDS

- Example Language Clean (called “C Lean”)
- Built upon a shallow embedding into State-Exception Monads
- Minimalistic language with
  - $\text{skip}_C$, $\text{sequence } _{-};-_-$, $\text{if}_C$, $\text{while}_C$, 
HOOKING UP BACKENDS

• Example Language Clean (called “C Lean”)

• Built upon a shallow embedding into State-Exception Monads

• Minimalistic language with
  • $\text{skip}_C$, $\text{sequence} _{-};_-$, $\text{if}_C$, $\text{while}_C$,
  • $\text{C-like control operators} \text{break}_C$, $\text{return}_C$
    based on implicit global control variables
HOOKING UP BACKENDS

- Example Language Clean (called “C Lean”)
- Built upon a shallow embedding into State-Exception Monads
- Minimalistic language with
  - \texttt{skip}_C, \texttt{sequence} \_\_\_;\_\_\_, \texttt{if}_C, \texttt{while}_C,
  - C-like control operators \texttt{break}_C, \texttt{return}_C based on implicit global control variables
  - local variables as stacks of global variables
HOOKING UP BACKENDS

- Example Language Clean (called “C Lean”)
- Built upon a shallow embedding into State-Exception Monads
- Minimalistic language with
  - \texttt{skip}, \texttt{sequence \_;\_}, \texttt{if}_C, \texttt{while}_C,
  - C-like control operators \texttt{break}_C, \texttt{return}_C based on implicit global control variables
  - local variables as stacks of global variables
  - (direct recursive) function calls
HOOKING UP BACKENDS
HOOKING UP BACKENDS
HOOKING UP BACKENDS

• Example in Isabelle/C/Clean
• Example in Isabelle/C/Clean

```isabelle
theory Prime imports Isabelle_C_Clean.Backend
  — <Clean back-end is now on>
begin

C <

//@ definition <prime_HOL (p :: nat) = \n (1 < p ∧ (∀ n ∈ {2..<p}. ¬ n dvd p))>

# define SQRT_UINT_MAX 65536

unsigned int k = 0;

unsigned int prime_c(unsigned int n) {

//@ preClean <C<n> ≤ UINT_MAX>
//@ postClean <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;

```
HOOKING UP BACKENDS

- Example in Isabelle/C/Clean

```plaintext
c theory Prime imports Isabelle_C_Clean.Backend
  -- <Clean back-end is now on>
begin

  C <
  //@ definition <prime_HOL (p :: nat) = 
  \(1 < p \land (\forall n \in \{2..p\}. \neg n \text{ dvd } p)\)>
  # define SQRT_UINT_MAX 65536
  unsigned int k = 0;
  unsigned int prime_C(\unsigned int n) {
    //@ preClean \(C\langle n \rangle \leq \text{UINT_MAX}\)
    //@ postClean \(C\langle \text{prime}_C(n) \rangle \neq 0 \iff \text{prime}_HOL C\langle n \rangle\)
    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;
  }
```

- C command generates for \prime_HOL, SQRT_UINT_MAX, \prime_{C\_pre}, \prime_{C\_post}, and \prime_C a monadic Clean presentation
HOOKING UP BACKENDS

- Example in Isabelle/C/Clean

```plaintext
theory Prime imports Isabelle_C_Clean.Backend
  — Clean back-end is now on
begin
C <
/// definition <prime_HOL (p :: nat) = \n(1 < p \land (\forall n \in \{2..<p\}. \neg n dvd p))>
# define SQRT_UINT_MAX 65536
unsigned int k = 0;
unsigned int prime_C(unsigned int n) {
/// preClean <C<n> \leq UINT_MAX>
/// postClean <C<prime_C(n)> \neq 0 \leftrightarrow 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;
}
```

- C command generates for prime_HOL, SQRT_UINT_MAX, prime_C_pre, prime_C_post, and prime_C a monadic Clean presentation

- definitions in the logical context σ
HOOKING UP BACKENDS
HOOKING UP BACKENDS

• Example in Isabelle/C/Clean
HOOKING UP BACKENDS

- Example in Isabelle/C/Clean

```haskell
"isPrime_core n ≡
  ifc (λσ. n < 2) then (return result_value_updatee (λσ. False))
  else skipSE fi;-
i_update ::=L (λσ. 2);-
whilec (λσ. (hdσi)σ < SQRT_UINT_MAX ∧ (hdσi)σ * (hdσi)σ ≤ n)
do
  (ifc (λσ. n mod (hd o i) σ = 0)
    then (return result_value_updatee (λσ. False))
    else skipSE fi;-
i_update ::=L (λσ. (hd o i) σ + 1))
  return_result_value_updatee (λσ. True)"
```
HOOKING UP BACKENDS

- Example in Isabelle/C/Clean

```plaintext
"isPrime_core n ≡
  if c (λσ. n < 2) then (return result_value_update (λσ. False))
  else skipSE fi;

  i_update :=L (λσ. 2);
  while c (λσ. (hd i)σ < SQRT_UINT_MAX ∧ (hd i)σ * (hd i)σ ≤ n)
  do
    (if c (λσ. n mod (hd i)σ = 0)
      then (return result_value_update (λσ. False))
      else skipSE fi ;-
      i_update :=L (λσ. (hd i)σ + 1))
  od ;-
  return result_value_update (λσ. True)"

"isPrime n ≡ block c push_local_isPrime_state
  (isPrime_core n)
  pop_local_isPrime_state"
```
HOOKING UP BACKENDS

• Example in Isabelle/C/Clean

```plaintext
"isPrime_core n ≡
  ifc (λσ. n < 2) then (\text{return\_result\_value\_update} (λσ. False))
  else skipSE fi;

  i_update ::=L (λσ. 2);

  whilec (λσ. (hd\_i)σ < SQRT_UINT_MAX ∧ (hd\_i)σ * (hd\_i)σ ≤ n)
  do
    (ifc (λσ. n mod (hd \_ i) σ = 0)
      then (\text{return\_result\_value\_update} (λσ. False))
      else skipSE fi ;-
    i_update ::=L (λσ. (hd \_ i) σ + 1))
  od ;-

  \text{return\_result\_value\_update} (λσ. True)"
```

```plaintext
"isPrime n ≡ blockC push\_local\_isPrime\_state
  (isPrime_core n)
  pop\_local\_isPrime\_state"
```

• Hoare-Triple over Monad:
HOOKING UP BACKENDS

- Example in Isabelle/C/Clean

```
"isPrime_core n ≡
  ifc (λσ. n < 2) then (return_value_result_true (λσ. False))
  else skipSE fi;
  i_update :=L (λσ. 2);
whilec (λσ. (hd(σ)) < SQRT_UINT_MAX ∧ (hd(σ)) * (hd(σ)) ≤ n)
  do
    ifc (λσ. n mod (hd i) = 0)
    then (return_value_result_true (λσ. False))
    else skipSE fi;
    i_update :=L (λσ. (hd i) + 1))
  od;
return_value_result_true (λσ. True)"
```

```
"isPrime n ≡ blockC push_local_isPrime_state
  (isPrime_core n)
  pop_local_isPrime_state"
```

- Hoare-Triple over Monad:

```
"{λσ. σ ∈ isPrime_pre (n)(σ) ∧ σ = σ_pre ∨}
  isPrime n
  {λr σ. σ ∈ isPrime_post(n) (σ_pre)(σ)(r) ∨}"
```
HOOKING UP BACKENDS
HOOKING UP BACKENDS

• Example in Isabelle/C/AutoCorres
HOOKING UP BACKENDS

- Example in Isabelle/C/AutoCorres

```isar
theorem (in primec) primec'_correct:
  (\{ \lambda _. n \leq UINT_MAX \} primec' n
   \{ \lambda primec'_ _. primec' \neq 0 \iff prime_HOL n \}!)
proof (rule validNF_assume_pre)
  assume 1: \langle n \leq UINT_MAX \rangle
  have 2: \langle n = 0 \lor n = 1 \lor n > 1 \rangle by linarith
  show ?thesis
    proof (insert 2, elim disjE)
      assume \langle n = 0 \rangle
      then show ?thesis
        by (clarsimp simp: primec'_def, wp, auto)
    next
```
DEMO

Isabelle/C and Isabelle/C/AutoCorres
CONCLUSION

- Isabelle/C: a generic Front-End for C providing general IDE support
- Technology can construct other parsers (C18, Javascript, Rust,...)
- Follows idea of “Isabelle as Eclipse” and Integrated Documents
- Instantiatable with various semantic interpretations of C, and derived, conservative libraries in HOL
- Platform for verification “back-ends” in Test and Proof
- Strong mechanism for plugin-separation as well as plugin-collaboration
Abstract

We report on an integration of a novel C11 Frontend into Isabelle/HOL enabling different semantic backends (AutoCorres, Securify, IMP2, Clean, . . . ). We discuss the challenges of a Generic Framework ranging from IDE to Proof-Support, and show how a small semantic backend can be hooked into our Framework enabling both deductive program verification as well as program-based Test-Generation.
ISABELLE - THE SYSTEM
THE "ECLIPSE OF FORMAL METHODS TOOLS"
ISABELLE - THE SYSTEM
THE "ECLIPSE OF FORMAL METHODS TOOLS"

- Isabelle is
ISABELLE - THE SYSTEM
THE “ECLIPSE OF FORMAL METHODS TOOLS”

• Isabelle is
  • an extensible programming system (component framework)
Isabelle is

- an extensible programming system (component framework)
- based on a parallel functional programming language SML
ISABELLE - THE SYSTEM
THE "ECLIPSE OF FORMAL METHODS TOOLS"

• Isabelle is
  • an extensible programming system (component framework)
  • based on a parallel functional programming language SML
  • geared towards ITP, but strong ATP support
ISABELLE - THE SYSTEM

THE “ECLIPSE OF FORMAL METHODS TOOLS”

- Isabelle is
  - an extensible programming system (component framework)
  - based on a parallel functional programming language SML
  - geared towards ITP, but strong ATP support