A Framework for the Automatic Formal Verification of Refinement from COGENT to C

Christine Rizkallah[4], Japheth Lim[1], Yutaka Nagashima[1], Thomas Sewell[1,2], Zilin Chen[1,2], Liam O'Connor[1,2], Toby Murray[1,3], Gabriele Keller[1,2], and Gerwin Klein[1,2]

[1] Data61 (formerly NICTA), CSIRO, Sydney, Australia

[2] University of New South Wales, Sydney, Australia

[3] University of Melbourne, Australia

[4] University of Pennsylvania, Philadelphia, PA, USA

ITP, Nancy, France
24th of August 2016

Cogent Project: Overall Story

Motivation

File systems are too important to remain unverified

COGENT Project: Overall Story

Motivation

File systems are too important to remain unverified

Problem

There are many file systems and they are huge ($\approx$ seL4)

COGENT Project: Overall Story

Motivation
File systems are too important to remain unverified

Problem
There are many file systems and they are huge ($\approx$ seL4)

Aim
Create method for feasibly verifying file systems

Cogent Project: Overall Story

Motivation

File systems are too important to remain unverified

Problem

There are many file systems and they are huge ($\approx$ seL4)

Aim

Create method for feasibly verifying file systems

Idea

Automate large portions of the verification process

Cogent Project: Overall Story

Motivation

File systems are too important to remain unverified

Problem

There are many file systems and they are huge (≈ seL4)

Aim

Create method for feasibly verifying file systems

Idea

Automate large portions of the verification process

Sounds good, but **HOW**?

Languages, Type Systems, and Certifying Compilers

Languages, Type Systems, and Certifying Compilers

- Purely functional languages allow for an easier, more direct style of formal reasoning about code

Languages, Type Systems, and Certifying Compilers

- Purely functional languages allow for an easier, more direct style of formal reasoning about code
- Design a restricted purely functional language, expressive enough to describe most of the file system code

Languages, Type Systems, and Certifying Compilers

- Purely functional languages allow for an easier, more direct style of formal reasoning about code
- Design a restricted purely functional language, expressive enough to describe most of the file system code
- Use type systems techniques (linear types) to enforce properties such as memory safety at compile time

Languages, Type Systems, and Certifying Compilers

- Purely functional languages allow for an easier, more direct style of formal reasoning about code
- Design a restricted purely functional language, expressive enough to describe most of the file system code
- Use type systems techniques (linear types) to enforce properties such as memory safety at compile time
- Create a certifying compiler to automate the low level "boring" parts of the verification

Cogent Project: Results in a Nutshell

- Cogent is a functional language with a linear type system.

Cogent Project: Results in a Nutshell

- Cogent is a functional language with a linear type system.
- Cogent's certifying compiler automatically generates:
  1. efficient C code
  2. a formal model of the code that is easy to reason about
  3. a machine-checked proof linking the two (in Isabelle)

COGENT Project: Results in a Nutshell

- COGENT is a functional language with a linear type system.
- COGENT's certifying compiler automatically generates:
  1. efficient C code
  2. a formal model of the code that is easy to reason about
  3. a machine-checked proof linking the two (in Isabelle)
- implemented `BilbyFs` and `ext2` file systems in COGENT
- verified functional correctness of key `BilbyFs` operations

Cogent Project: Results in a Nutshell

- Cogent is a functional language with a linear type system.
- Cogent's certifying compiler automatically generates:
  1. efficient C code
  2. a formal model of the code that is easy to reason about
  3. a machine-checked proof linking the two (in Isabelle)
- implemented `BilbyFs` and `ext2` file systems in Cogent
- verified functional correctness of key `BilbyFs` operations
  - relied on automating large parts of the verification

How does the Certifying Compiler Automatically Generate Refinement Theorems and Proofs?

How does the Certifying Compiler Automatically Generate Refinement Theorems and Proofs?

But First Let's Get Familiar with COGENT

The COGENT Language

- linearly typed, restricted, polymorphic, higher-order, purely functional language
- has typical **let**, **if**, ..., for records **take**, **put** (linearity)
- COGENT programs rely on an external library of abstract data types (ADTs) for data structures like arrays, lists, red-black trees, etc.
- loops are implemented as iterators over ADTs

Linear Type System

- Variables with a linear type must be used exactly once.
- COGENT's linear type system:
  - allows generating efficient imperative code with in-place updates
  - assists memory management
  - prevent errors such as use-after-free, memory leaks, pointer mismanagement in error-handling, etc.

COGENT Example: Flip

$$
\begin{array}{l}
\mathit{flip} \;\; :: \{\mathrm{f} \;\; :: \texttt{U8}\}\ \texttt{w} \rightarrow \{\mathrm{f} \;\; :: \texttt{U8}\}\ \texttt{w} \\
\mathit{flip}\ x = \\
\quad \textbf{take}\ x'\ \{\mathrm{f} = y\} = x \\
\quad \textbf{in if}\ y == 0 \\
\qquad \textbf{then put}\ x'.\mathrm{f} := 1 \\
\qquad \textbf{else put}\ x'.\mathrm{f} := 0
\end{array}
$$

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\begin{array}{l}\textbf{let}\ x\ =\ \mathit{allocData}\\ \textbf{and}\ y\ =\ x\\ \textbf{and}\ \_\ =\ \mathit{free}\ x\\ \textbf{in}\ y\end{array}$$

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\begin{array}{l} \textbf{let}\ x\ =\ allocData \\ \textbf{and}\ y\ =\ \boldsymbol{x} \\ \textbf{and}\ \_\ =\ free\ \boldsymbol{x} \\ \textbf{in}\ y \end{array}$$

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\textbf{let}\ x\ =\ allocData$$
$$\textbf{in}\ ()$$

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\textbf{let } x \ = \ allocData$$
$$\textbf{in } ()$$

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\begin{array}{l}\textbf{let}\ x\ =\ \mathit{allocData}\\ \textbf{in if}\ \mathit{condition}\ \textbf{then}\ \mathit{Some}\ x\\ \qquad\qquad\qquad\quad \textbf{else}\ \mathit{None}\end{array}$$

Variables with a linear type must be freed or returned.

Linear Type System: Examples

Variables with a linear type must be used exactly once.

$$\begin{array}{l} \textbf{let } x \ = \ allocData \\ \textbf{in if } condition \textbf{ then } Some\ x \\ \qquad\qquad\qquad\qquad \textbf{else } \boldsymbol{None} \end{array}$$

Variables with a linear type must be freed or returned.

Linear Type System: Examples

The linear type system enables the conversion from purely functional to imperative code with in-place memory update.

$$\begin{array}{l}\textbf{let}\ x\ =\ allocData \\ \textbf{and}\ y\ =\ updateData\ x \\ \textbf{in}\ y\end{array}$$

Linear Type System: Examples

The linear type system enables the conversion from purely functional to imperative code with in-place memory update.

$$
\begin{array}{l}
\textbf{let}\ x\ =\ \mathit{allocData} \\
\textbf{and}\ y\ =\ \mathit{updateData}\ x \\
\textbf{in}\ y
\end{array}
$$

The C code uses the same variables in memory for x and y.

Linear Type System: Examples

The linear type system enables the conversion from purely functional to imperative code with in-place memory update.

$$\begin{array}{l} \textbf{let}\ x\ =\ \mathit{allocData} \\ \textbf{and}\ y\ =\ \mathit{updateData}\ x \\ \textbf{in}\ y \end{array}$$

The C code uses the same variables in memory for x and y.

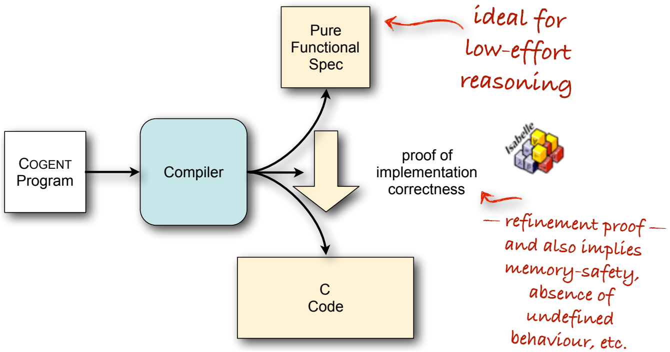
COGENT needs no garbage collector.

Why can we get away with type system and restrictions?

Idea

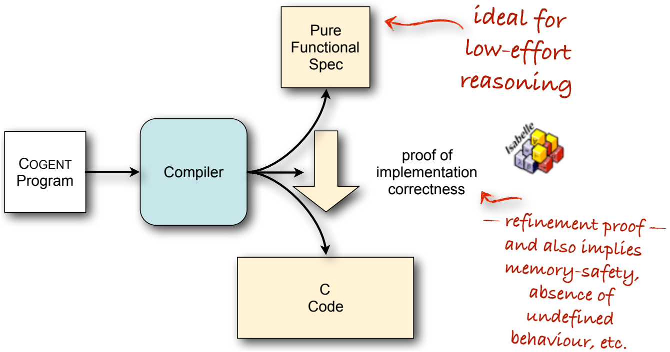
- Write most code in COGENT to simplify its verification
- Write a small part in C and verify it manually

Cogent: Certifying Compiler



- Certifying compiler: certifies correctness of its compilation.

COGENT: Certifying Compiler



- Certifying compiler: certifies correctness of its compilation.
- COGENT's certifying compiler automatically generates:
  1. efficient C code
  2. a formal HOL spec. of the code that is easy to reason about
  3. a machine-checked proof linking the two (in Isabelle)

COGENT: Certifying Compiler

Refinement

- The compiler uses an underlying routine to
  - discharge cumbersome C safety obligations and
  - provide a HOL emb. more amenable to verification.

Cogent: Certifying Compiler

Refinement

- The compiler uses an underlying routine to
  - discharge cumbersome C safety obligations and
  - provide a HOL emb. more amenable to verification.
- Theorem: When C executes to a value, HOL spec. evaluates similarly.

Cogent: Certifying Compiler

Refinement

- The compiler uses an underlying routine to
  - discharge cumbersome C safety obligations and
  - provide a HOL emb. more amenable to verification.
- Theorem: When C executes to a value, HOL spec. evaluates similarly.
- Hence, proofs about HOL spec. also hold for C code.

COGENT: Certifying Compiler

Refinement

- The compiler uses an underlying routine to
  - discharge cumbersome C safety obligations and
  - provide a HOL emb. more amenable to verification.
- Theorem: When C executes to a value, HOL spec. evaluates similarly.
- Hence, proofs about HOL spec. also hold for C code.
- This means we can now prove theorems using low-effort equational reasoning on HOL spec. rather than deal with tedious C, and the theorems also hold for the C code!

Cogent: Certifying Compiler

Refinement

- The compiler uses an underlying routine to
  - discharge cumbersome C safety obligations and
  - provide a HOL emb. more amenable to verification.
- Theorem: When C executes to a value, HOL spec. evaluates similarly.
- Hence, proofs about HOL spec. also hold for C code.
- This means we can now prove theorems using low-effort equational reasoning on HOL spec. rather than deal with tedious C, and the theorems also hold for the C code!

Performance

- The performance of the generated C is similar to that of hand written C.

Cogent Compiler uses Pre-Existing Tools

Simpl [Schirmer 2005]

- Imperative language embedded into Isabelle/HOL

C-Parser [Norrish 2012]

- straightforward translation from C to Simpl

AutoCorres [Greenaway et al. 2012]

- converts Simpl to monadic representation in Isabelle/HOL
- verified simplifications
- output meant for manual reasoning
- we tweak AutoCorres to make its output more predictable (for automation on top)

How does the Certifying Compiler Automatically Generate Refinement Theorems and Proofs?

Cogent Example: Back to Flip

```
flip :: {f :: U8} w → {f :: U8} w
flip x =
   take x′ {f = y} = x
   in if y == 0
      then put x′.f := 1
      else put x′.f := 0
```

Cogent Example: Generated HOL and C Flip

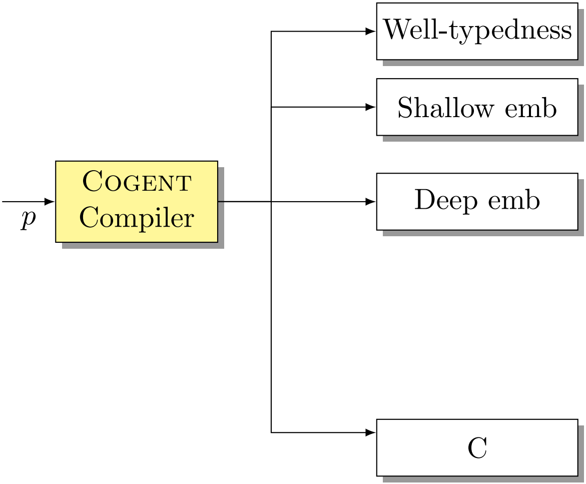
Functional Specification (HOL)

```
flip :: {f :: U8} → {f :: U8}
flip x =
  take x′ {f = y} = x
  in let tmp₁ = 0
     and tmp₂ = (y == tmp₁)
     in if tmp₂
        then let tmp₃ = 1
             and x″ = put x′.f := tmp₃
             in x″
        else let tmp₄ = 0
             and x″ = put x′.f := tmp₄
             in x″
```

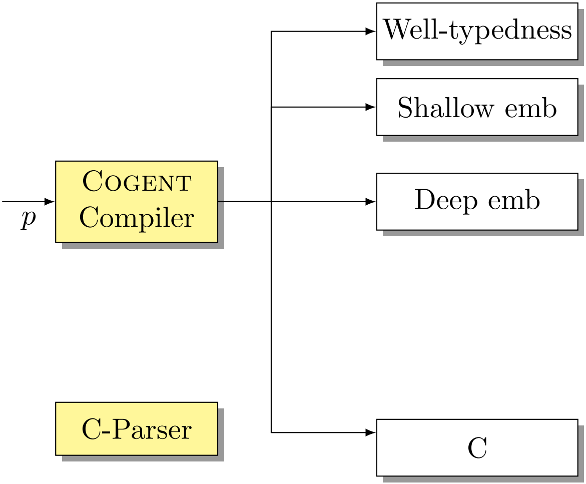
C code (monadic representation)

```
flip_C :: rec₁ ptr ⇒ (rec₁ ptr, σ) nondet_monad
flip_C x = do
  guard (λσ. is-valid σ x);
  y ← gets (λσ. σ[r].f);
  tmp₁ ← return 0;
  tmp₂ ← return bool (y = tmp₁);
  tmp_result ← condition (bool tmp₂ ≠ 0)
      (do tmp₃ ← return 1;
          guard (λσ. is-valid σ x);
          modify (λσ. σ[x].f := tmp₃);
          return x od)
      (do tmp₄ ← return 0;
          guard (λσ. is-valid σ x);
          modify (λσ. σ[x].f := tmp₄);
          return x od);
  return tmp_result
  od
```

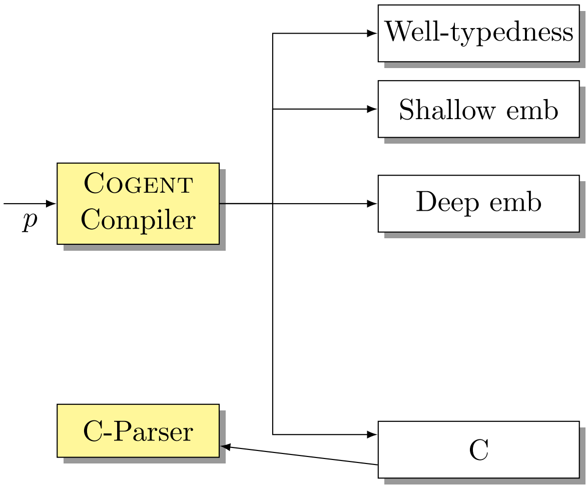
Overall Project: How did we structure refinement?



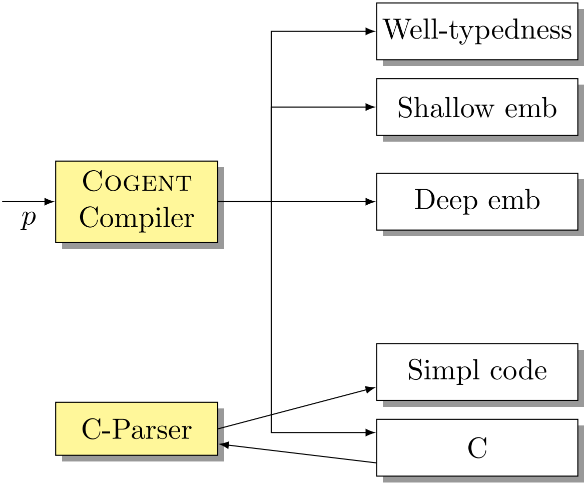
Overall Project: How did we structure refinement?



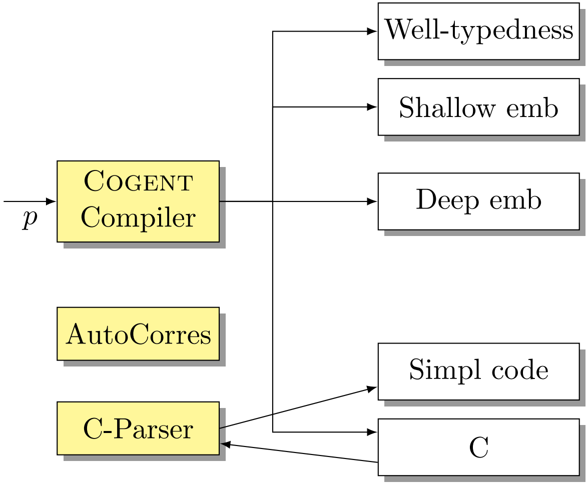
Overall Project: How did we structure refinement?



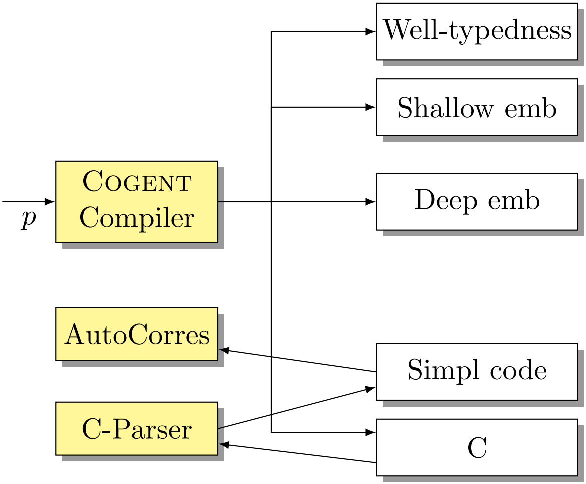
Overall Project: How did we structure refinement?



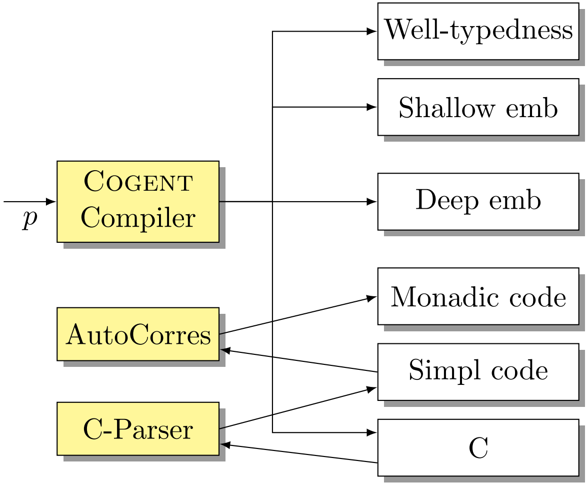
Overall Project: How did we structure refinement?



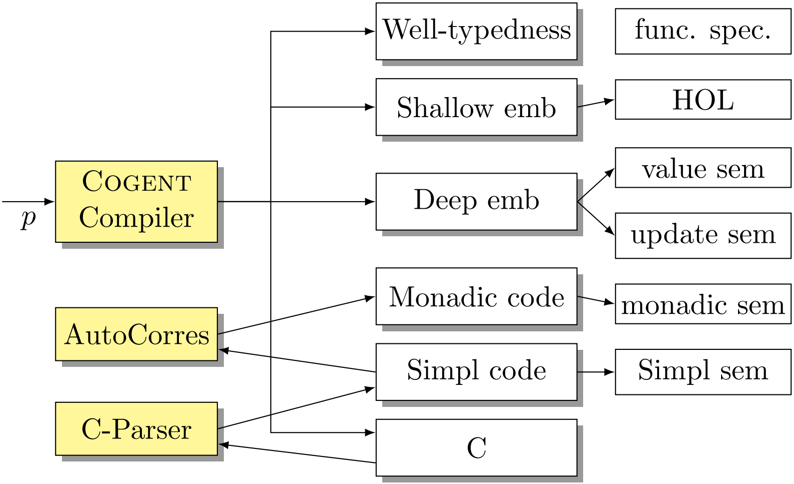
Overall Project: How did we structure refinement?



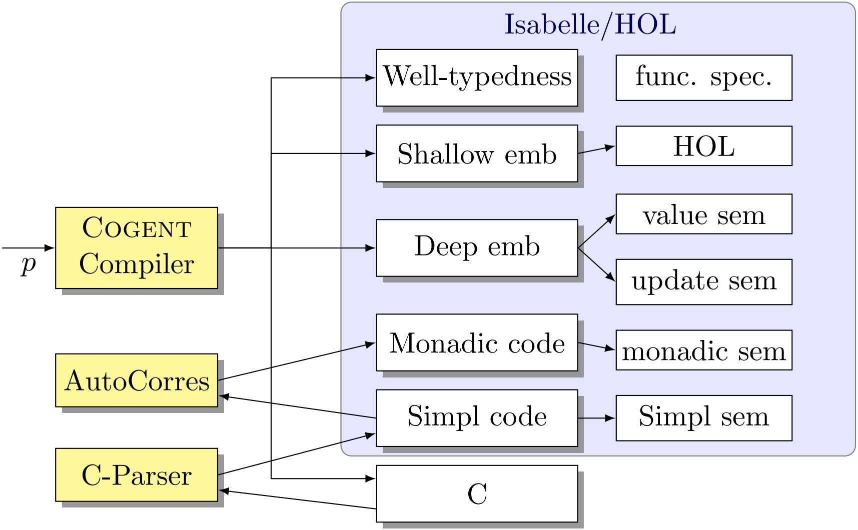
Overall Project: How did we structure refinement?



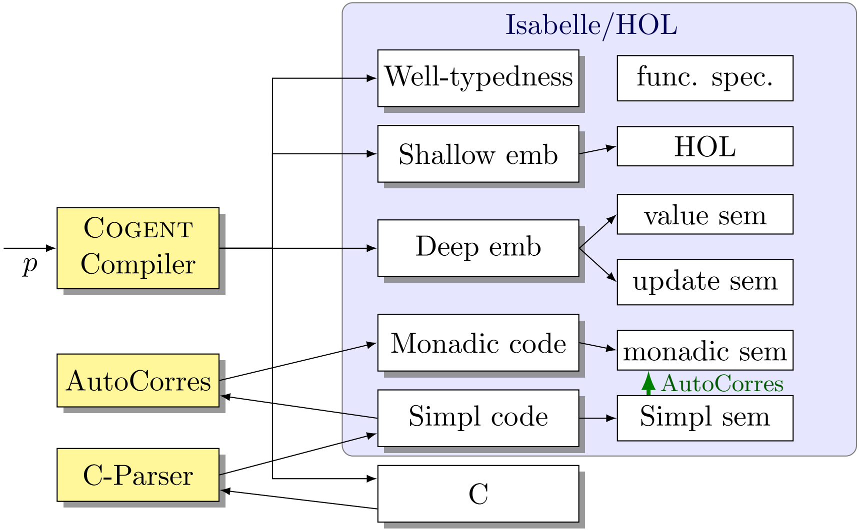
Overall Project: How did we structure refinement?



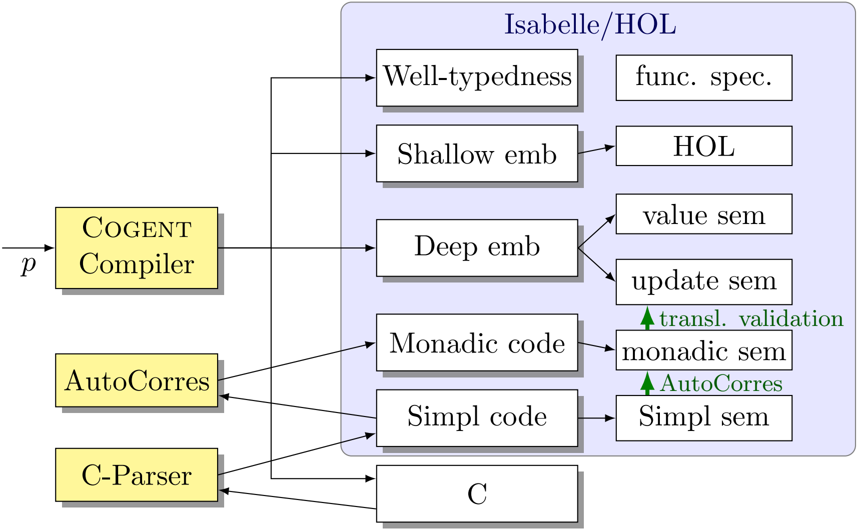
Overall Project: How did we structure refinement?



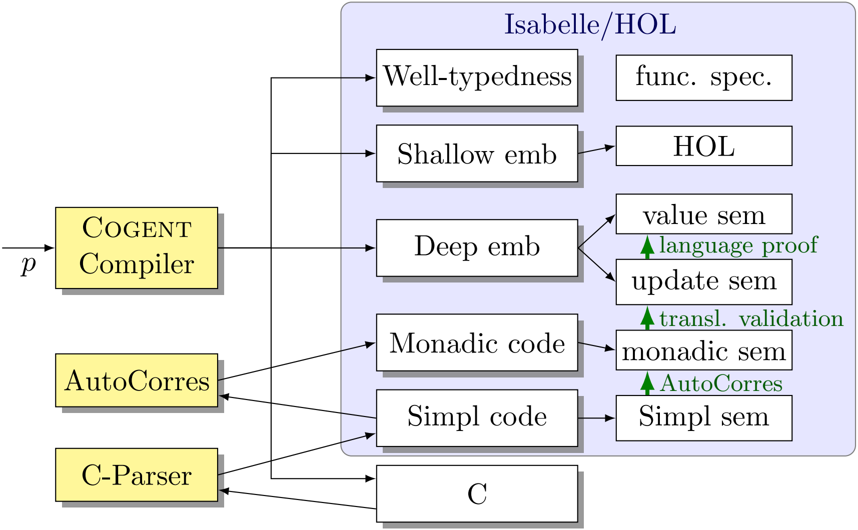
Overall Project: How did we structure refinement?



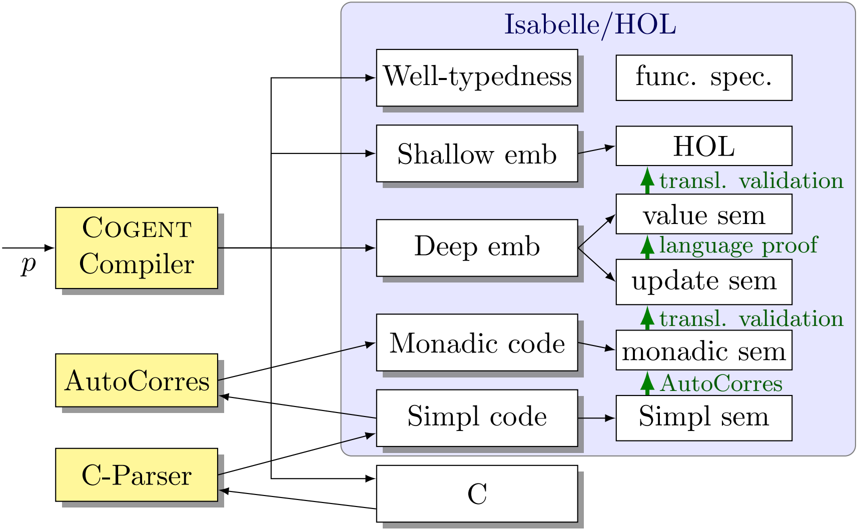
Overall Project: How did we structure refinement?



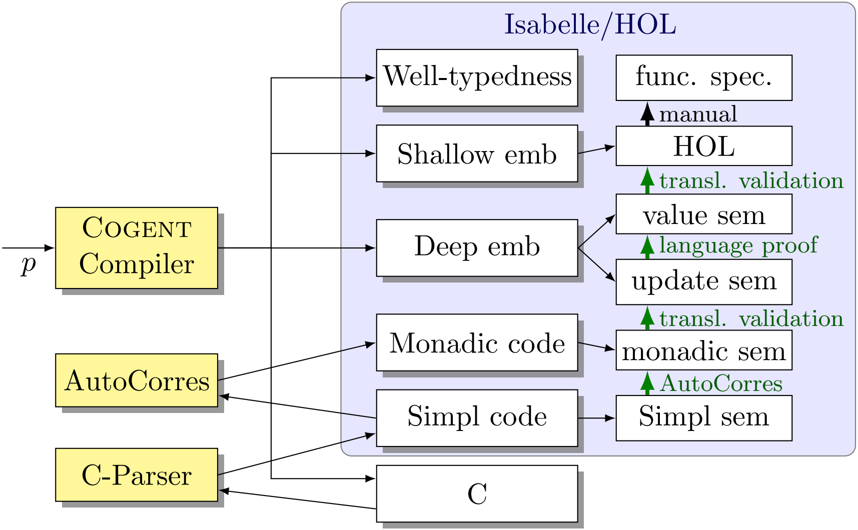
Overall Project: How did we structure refinement?



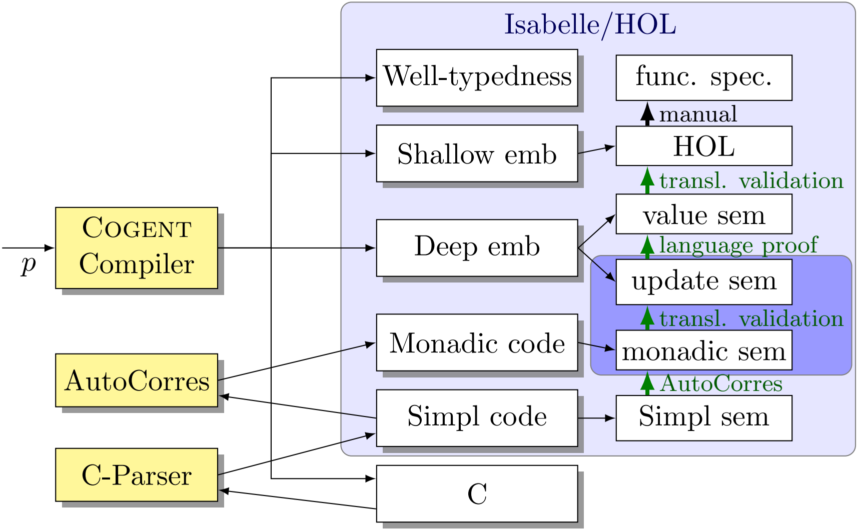
Overall Project: How did we structure refinement?



Overall Project: How did we structure refinement?



Overall Project: How did we structure refinement?



How did we prove refinement (update to monadic sem.)?

Data Relations and Refinement Predicate

- values
- types

- states
- statements

Theories and Automation

- refinement calculus
- well-typedness proof
- syntax directed proof automation

How did we prove refinement (update to monadic sem.)?

Data Relations and Refinement Predicate

- values
- types
  - several COGENT types correspond to the same C type
- states
- statements

Theories and Automation

- refinement calculus
- well-typedness proof
- syntax directed proof automation

How did we prove refinement (update to monadic sem.)?

Data Relations and Refinement Predicate

- values
- types
  - several COGENT types correspond to the same C type
  - partial type erasure removes linearity from COGENT type
- states
- statements (**corres** R e p_m U Γ μ σ)

Theories and Automation

- refinement calculus
- well-typedness proof
- syntax directed proof automation

Refinement Calculus: Var rule

$$\frac{(x \mapsto v_u) \in U \qquad \mathit{val\text{-}rel}\ v_u\ v_m}{\mathbf{corres}\ R\ x\ (\mathbf{return}\ v_m)\ U\ \Gamma\ \mu\ \sigma}\textsc{Var}$$

Refinement Calculus: If rule

$$\dfrac{\Gamma_1 \vdash c : \texttt{Bool} \qquad (\mathit{bool}\, c' = 0 \lor \mathit{bool}\, c' = 1) \\ c \text{ is a Cogent boolean equal to } (\mathit{bool}\ c' \neq 0) \\ \textbf{corres}\ R\ e_1\ e_1'\ U\ \Gamma_2\ \mu\ \sigma \qquad \textbf{corres}\ R\ e_2\ e_2'\ U\ \Gamma_2\ \mu\ \sigma}{\textbf{corres}\, R\, (\textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \\ (\textbf{do}\, x \leftarrow \textbf{condition}\ (\mathit{bool}\, c' \neq 0)\ e_1'\ e_2';\ \textbf{return}\, x\, \textbf{od})\, U\, (\Gamma_1\Gamma_2)\, \mu\, \sigma}\ \textsc{If}$$

Well-typedness

- Cogent compiler proves via an automated Isabelle tactic that the deep embedding of input program is well-typed

Well-typedness

- Cogent compiler proves via an automated Isabelle tactic that the deep embedding of input program is well-typed
- proving refinement requires access to typing judgements of subexpressions of the program

Well-typedness

- Cogent compiler proves via an automated Isabelle tactic that the deep embedding of input program is well-typed
- proving refinement requires access to typing judgements of subexpressions of the program
- due to linear types, it is not easy to statically infer that subexpressions of a well-typed program are well-typed

Well-typedness

- Cogent compiler proves via an automated Isabelle tactic that the deep embedding of input program is well-typed
- proving refinement requires access to typing judgements of subexpressions of the program
- due to linear types, it is not easy to statically infer that subexpressions of a well-typed program are well-typed
- our proof automation then uses well-typedness theorems to discharge typing assumptions in refinement calculus

Proof Automation (for concrete program)

- specialize refinement calculus
  - some of the rules in the calculus are pretty complicated
  - we specialize complex rules to one for each type (of record)

Proof Automation (for concrete program)

- specialize refinement calculus
  - some of the rules in the calculus are pretty complicated
  - we specialize complex rules to one for each type (of record)
- tactic compositionally applies syntax directed rules

Proof Automation (for concrete program)

- specialize refinement calculus
  - some of the rules in the calculus are pretty complicated
  - we specialize complex rules to one for each type (of record)
- tactic compositionally applies syntax directed rules
- tactic uses well-typedness theorems to discharge typing assumptions

Proof Automation (for concrete program)

- specialize refinement calculus
  - some of the rules in the calculus are pretty complicated
  - we specialize complex rules to one for each type (of record)
- tactic compositionally applies syntax directed rules
- tactic uses well-typedness theorems to discharge typing assumptions
- refinement for foreign functions is assumed

Conclusion

- We developed a compositional refinement calculus and proof rules to create a fully automatic refinement certificate from Cogent to C.
- Through co-generation of code and proofs our framework significantly reduces the cost of reasoning about efficient C.
- It does so by discharging cumbersome safety obligations and providing an embedding more amenable to verification.
- Our framework was applied to two real world file systems.

Future work

- Speed C-Parser and AutoCorres for our very large dev.
    - `BilbyFs` generated code is $\approx$ `1.8 x` the size of `seL4`
- Add optimizations (additional to ones we get by using `gcc`)
- Verify a library of abstract data types

Meet the other Verification/PL Folks

Japheth Lim

Yutaka Nagashima

Thomas Sewell

Zilin Chen

Liam O'Connor

Toby Murray

Gabi Keller

Gerwin Klein

Now that you know about COGENT,
consider trying it out!

COGENT: co-generation of code and proofs significantly reduces the cost of reasoning about efficient C.

Thanks for Listening, Questions?