Trustworthy Runtime Verification via Bisimulation (Experience Report)

Ryan Scott, Galois, Inc.
Mike Dodds, Galois, Inc.
Ivan Perez, KBR @ NASA Ames Research Center
Alwyn Goodloe, NASA Langley Research Center
Robert Dockins, Amazon

We wrote a compiler verification tool for proving the equivalence of high-level, stream code and low-level, translated C code.

We wrote a compiler verification tool for proving the equivalence of high-level, stream code and low-level, translated C code.

Both the stream language (Copilot) and the compiler (Copilot-C99) existed before we started this project (CopilotVerifier).

We wrote a compiler verification tool for proving the equivalence of high-level, stream code and low-level, translated C code.

Both the stream language (Copilot) and the compiler (Copilot-C99) existed before we started this project (CopilotVerifier).

We did this with a very limited time and engineering budget.

We wrote a compiler verification tool for proving the equivalence of high-level, stream code and low-level, translated C code.

Both the stream language (Copilot) and the compiler (Copilot-C99) existed before we started this project (CopilotVerifier).

We did this with a very limited time and engineering budget.

Our secret: building on off-the-shelf formal methods, tools and libraries.

CopilotVerifier uses bisimulation to prove program equivalence.

CopilotVerifier uses bisimulation to prove program equivalence.

Specifically, we show that the possible states of a stream program and the possible states of a C program are in bisimulation with each other.

CopilotVerifier uses bisimulation to prove program equivalence.

Specifically, we show that the possible states of a stream program and the possible states of a C program are in bisimulation with each other.

In effect, we show that the programs’ observable behaviors coincide.

CopilotVerifier uses bisimulation to prove program equivalence.

Specifically, we show that the possible states of a stream program and the possible states of a C program are in bisimulation with each other.

In effect, we show that the programs' observable behaviors coincide.

We reduce the key steps of a bisimulation proof to a set of goals that can be discharged with an SMT solver.

Results

We developed a fully working version of CopilotVerifier in just under one year with two engineers.

Results

We developed a fully working version of CopilotVerifier in just under one year with two engineers.

Plan to use CopilotVerifier as part of NASA missions in the future.

Results

We developed a fully working version of CopilotVerifier in just under one year with two engineers.

Plan to use CopilotVerifier as part of NASA missions in the future.

CopilotVerifier has already caught 10 bugs in Copilot.

Results

We developed a fully working version of CopilotVerifier in just under one year with two engineers.

Plan to use CopilotVerifier as part of NASA missions in the future.

CopilotVerifier has already caught 10 bugs in Copilot.

CopilotVerifier can verify all of the programs in the Copilot test suite, including an implementation of the *Well-Clear Violation* algorithm used in unmanned aircraft.

Copilot: a framework for writing monitors using runtime verification

Copilot

Copilot programs are written in a stream-based DSL in Haskell.

Copilot

Copilot programs are written in a stream-based DSL in Haskell.

Designed for monitoring systems at runtime using *runtime verification*.

Copilot

Copilot programs are written in a stream-based DSL in Haskell.

Designed for monitoring systems at runtime using *runtime verification*.

Originally developed by Galois and National Institute of Aerospace in 2010.

Copilot example

```
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)
```

Copilot example

```
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0
```

Copilot example

```
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0

spec :: Spec
spec = do
  trigger "even"
    (isEven fibs) [arg fibs]
  ...
```

Copilot example

```haskell
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0

spec :: Spec
spec = do
  trigger "even"
    (isEven fibs) [arg fibs]
  ...
```

Copilot-C99

Copilot example

```haskell
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0

spec :: Spec
spec = do
  trigger "even"
    (isEven fibs) [arg fibs]
  ...
```

Copilot-C99

```c
int fibs[2] = {1, 1};
size_t fibs_idx = 0;

bool even_guard(void) {
  return (fibs[fibs_idx % 2] % 2) == 0;
}

void step(void) {
  if (even_guard()) {
    even(fibs[fibs_idx % 2]);
  }
  ...
  fibs[idx] = fibs_gen();
  fibs_idx = (fibs_idx + 1) % 2;
}
```

Copilot example

```haskell
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0

spec :: Spec
spec = do
  trigger "even"
    (isEven fibs) [arg fibs]
  ...
```

Copilot-C99 →

```c
int fibs[2] = {1, 1};
size_t fibs_idx = 0;

bool even_guard(void) {
  return (fibs[fibs_idx % 2] % 2) == 0;
}

void step(void) {
  if (even_guard()) {
    even(fibs[fibs_idx % 2]);
  }
  ...
  fibs[idx] = fibs_gen();
  fibs_idx = (fibs_idx + 1) % 2;
}
```

Copilot example

```haskell
fibs :: Stream Int
fibs = [1, 1] ++ (fibs + drop 1 fibs)

isEven :: Stream Int -> Stream Bool
isEven n = (n `mod` 2) == 0

spec :: Spec
spec = do
  trigger "even"
    (isEven fibs) [arg fibs]
  ...
```

≈?

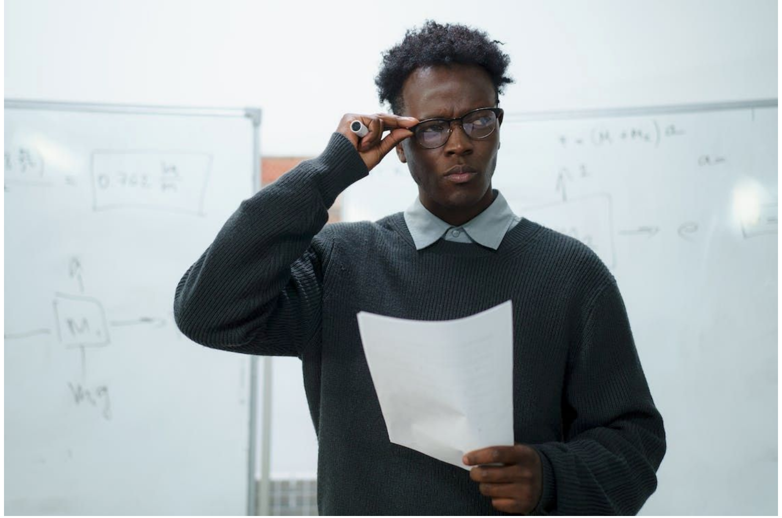
```c
int fibs[2] = {1, 1};
size_t fibs_idx = 0;

bool even_guard(void) {
  return (fibs[fibs_idx % 2] % 2) == 0;
}

void step(void) {
  if (even_guard()) {
    even(fibs[fibs_idx % 2]);
  }
  ...
  fibs[idx] = fibs_gen();
  fibs_idx = (fibs_idx + 1) % 2;
}
```

How do we know that Copilot-generated C code is *trustworthy*?

Option A: Audit the code by hand

Option A: Audit the code by hand

…but this is error-prone.



Free-to-use photo by cottonbro studio from Pexels

```
#include<stdio.h> #include<string.h>
main(){char*O,l[999]="'`acgo\177~|xp .
-\OR^8)NJ6%K4O+A2M(*OID57$3G1FBL";
while(O=fgets(l+45,954,stdin)){*l=O[
strlen(O)[O-1]=0,strspn(O,l+11)];
while(*O)switch((*l&&isalnum(*O))-!*l)
{case-1:{char*I=(O+=strspn(O,l+12)
+1)-2,O=34;while(*I&3&&(O=(O-16<<1)+
*I---'-')<80);putchar(O&93?*I
&8||!( I=memchr( l , O , 44 ) ) ?'?':
I-l+47:32); break; case 1: ;}*l=
(*O&31)[l-15+(*O>61)*32];while(putchar
(45+*l%2),(*l=*l+32>>1)>35); case 0:
putchar((++O ,32));}putchar(10);}}
```

Option B: Formally verify the Copilot compiler

Option B: Formally verify the Copilot compiler

…but this would require more time and budget than we had.

$$\forall C, \forall i, compile(C)(i) = semantics(C)(i)$$

Option C: Translation validation

Option C: Translation validation

…i.e., construct a proof of equivalence between the source and target programs each time the compiler runs.[1]

[1]Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. *Translation validation*.

Option C: Translation validation

…i.e., construct a proof of equivalence between the source and target programs each time the compiler runs.[1]

This is a weaker result than full compiler verification, but one that is more readily adaptable to existing compilers.

[1] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. *Translation validation*.

Option C: Translation validation ✓

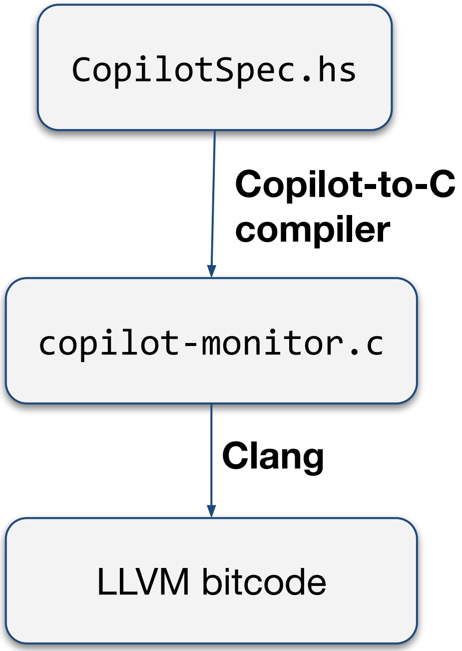
…i.e., construct a proof of equivalence between the source and target programs each time the compiler runs.[1]

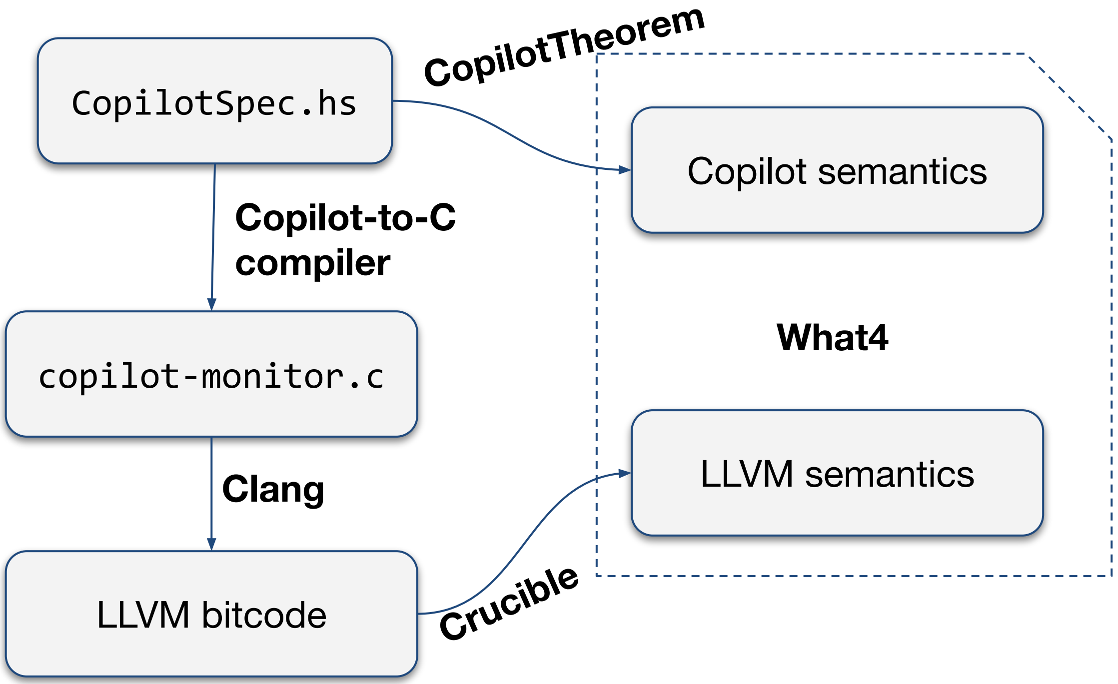
This is a weaker result than full compiler verification, but one that is more readily adaptable to existing compilers.

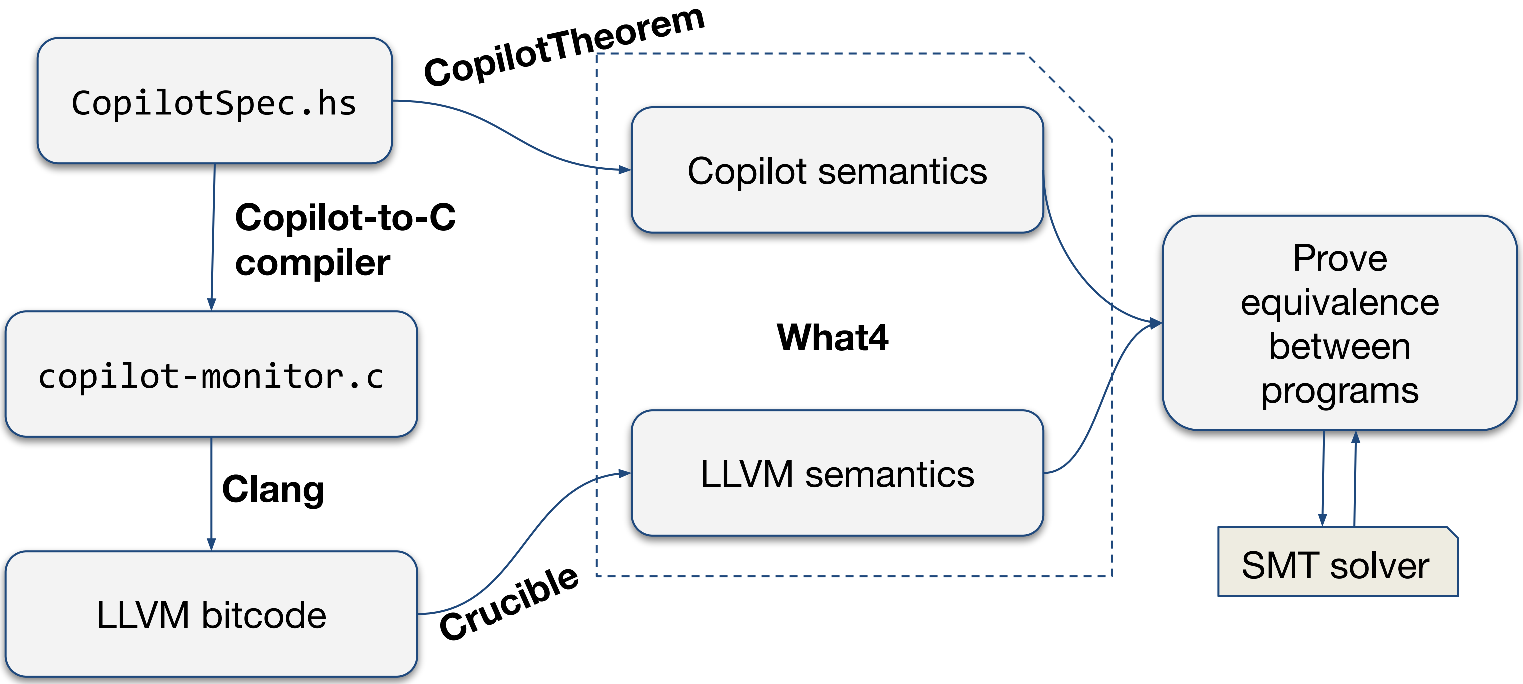
[1] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. *Translation validation*.

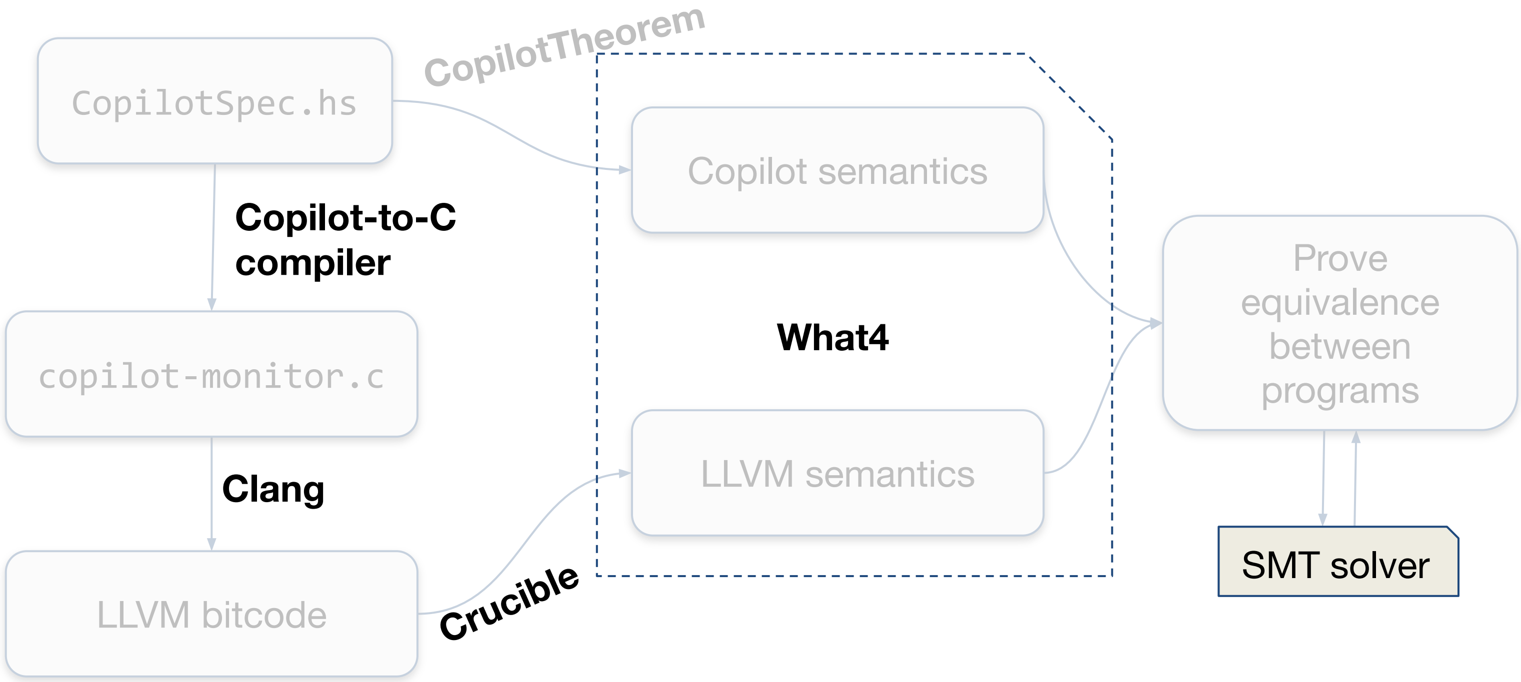
CopilotVerifier overview

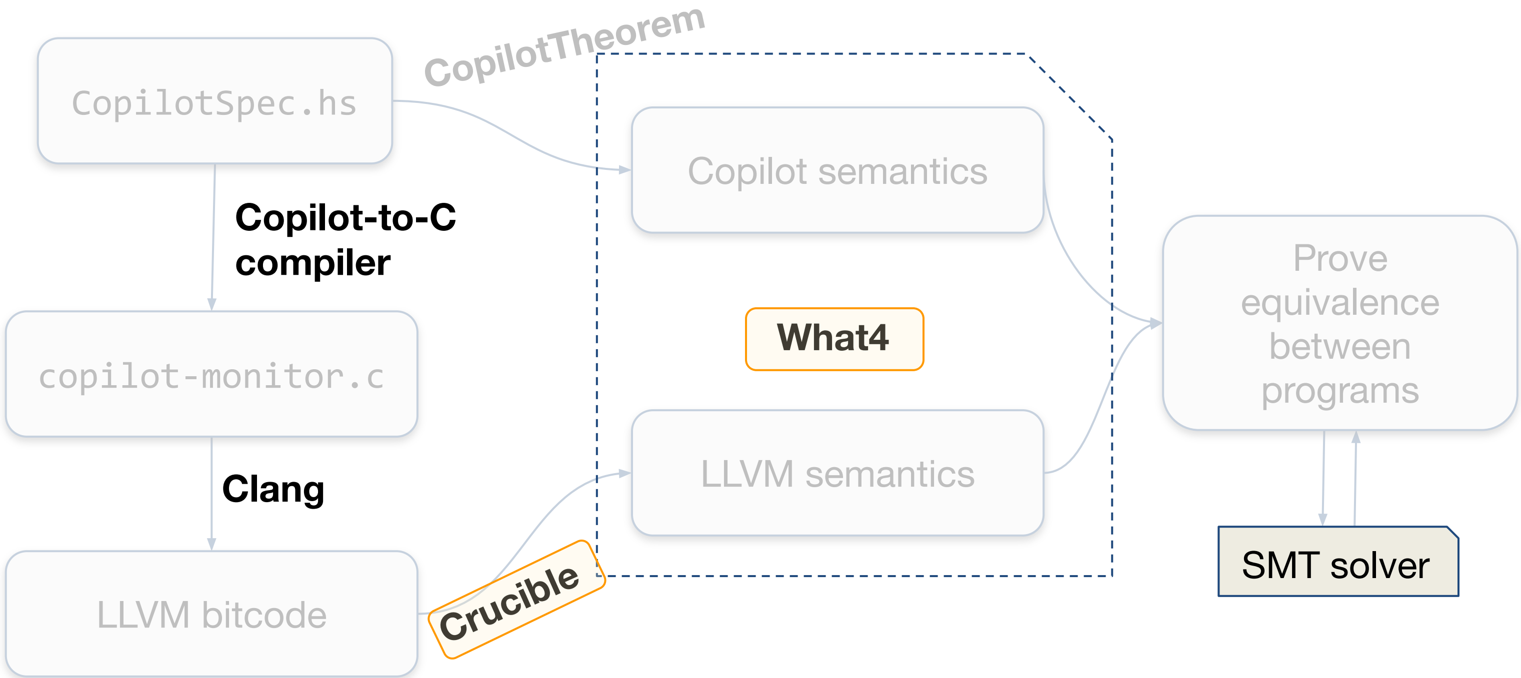
CopilotSpec.hs


CopilotSpec.hs
Copilot-to-C compiler
copilot-monitor.c
Clang
LLVM bitcode

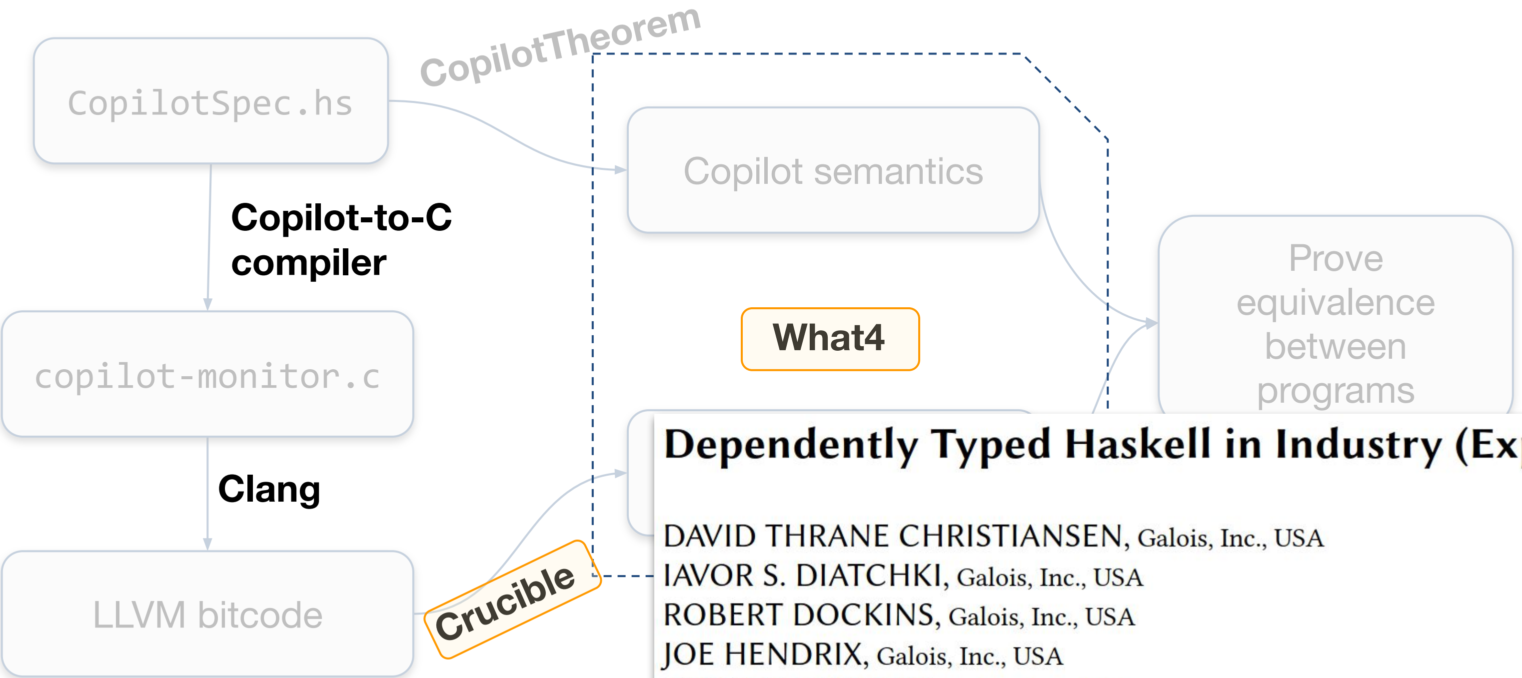

CopilotSpec.hs
CopilotTheorem
Copilot semantics
Copilot-to-C compiler
What4
copilot-monitor.c
Clang
LLVM semantics
LLVM bitcode
Crucible


CopilotSpec.hs
CopilotTheorem
Copilot semantics
Copilot-to-C
compiler
What4
Prove
equivalence
between
programs
copilot-monitor.c
Clang
LLVM semantics
SMT solver
LLVM bitcode
Crucible


CopilotSpec.hs
CopilotTheorem
Copilot semantics
Copilot-to-C
compiler
copilot-monitor.c
What4
Prove
equivalence
between
programs
Clang
LLVM semantics
LLVM bitcode
Crucible
SMT solver



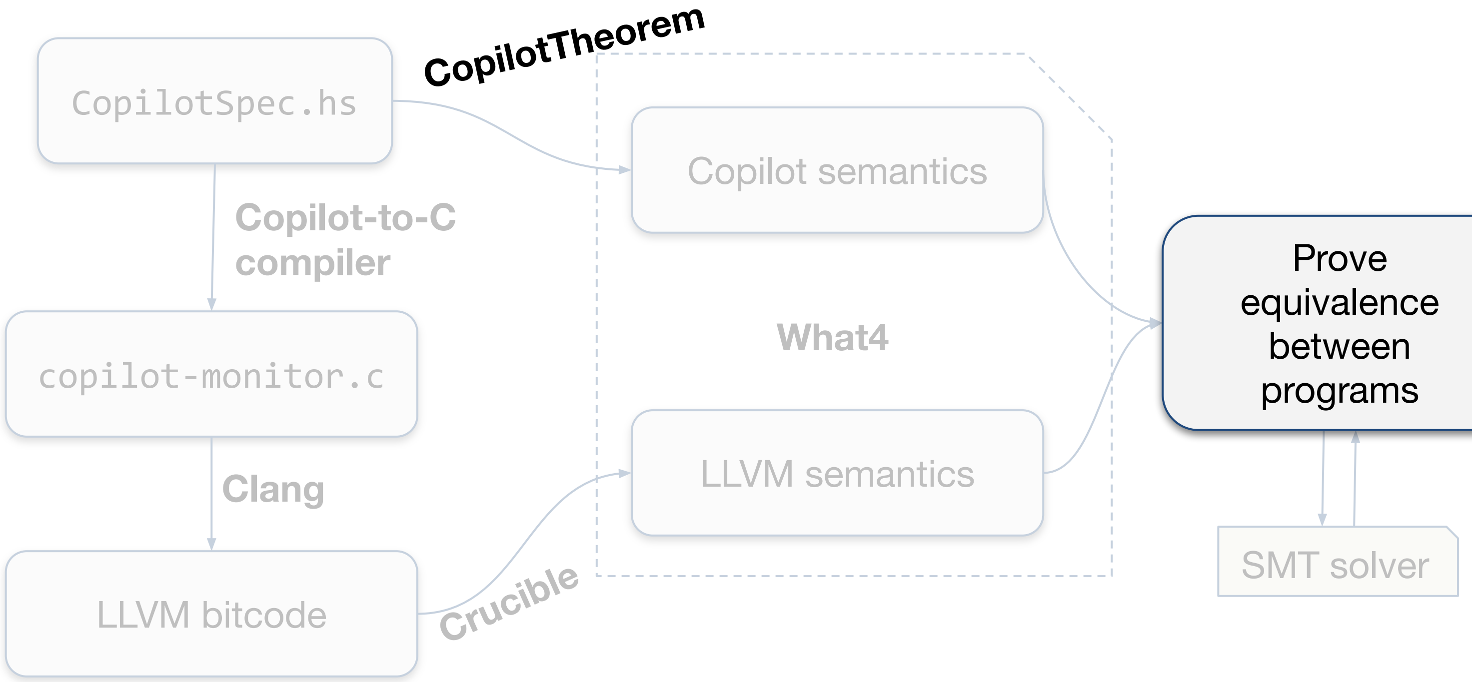
CopilotTheorem
CopilotSpec.hs
Copilot semantics
Copilot-to-C
compiler
What4
Prove
equivalence
between
programs
copilot-monitor.c
Clang
LLVM semantics
SMT solver
LLVM bitcode
Crucible



Dependently Typed Haskell in Industry (Experience Report)

DAVID THRANE CHRISTIANSEN, Galois, Inc., USA
IAVOR S. DIATCHKI, Galois, Inc., USA
ROBERT DOCKINS, Galois, Inc., USA
JOE HENDRIX, Galois, Inc., USA
TRISTAN RAVITCH, Galois, Inc., USA

Recent versions of the Haskell compiler GHC have a number of advanced features that allow many idioms from dependently typed programming to be encoded. We describe our experiences using this "dependently typed Haskell" to construct a performance-critical library that is a key component in a number of verification tools. We have discovered that it can be done, and it brings significant value, but also at a high cost. In this experience report, we describe the ways in which programming at the edge of what is expressible in Haskell's type system has brought us value, the difficulties that it has imposed, and some of the ways we coped with the difficulties.


CopilotTheorem
CopilotSpec.hs
Copilot-to-C
compiler
copilot-monitor.c
Clang
LLVM bitcode
Crucible
Copilot semantics
What4
LLVM semantics
Prove
equivalence
between
programs
SMT solver

Proving programs equivalent via bisimulation

We want to prove that a Copilot stream program and its corresponding C program are *extensionally equal*, i.e., at every time step:

- The same set of trigger functions are called in both programs with the same arguments
- The stream program crashes if and only if the C program crashes

Proving programs equivalent via bisimulation

We prove extensional equality by:

1. Representing each program as a labelled transition system (LTS)

Proving programs equivalent via bisimulation

We prove extensional equality by:

1. Representing each program as a labelled transition system (LTS)
2. Generating verification conditions to show the two LTSs are extensionally equal at a given time step

Goal 1: even trigger fires in both programs

```
(declare-fun s0_idx () (_ BitVec 64))
(define-fun x!0 () (_ BitVec 64) (bvadd s0_idx (_ bv4 64)))
(define-fun x!1 () (_ BitVec 64) (bvurem x!0 (_ bv5 64)))
(define-fun x!2 () (_ BitVec 64) (bvmul (_ bv4 64) x!1))
...
```

Goal 2: ...

...

Goal 3: ...

...

Proving programs equivalent via bisimulation

We prove extensional equality by:

1. Representing each program as a labelled transition system (LTS)
2. Generating verification conditions to show the two LTSs are extensionally equal at a given time step
3. Check verification conditions with SMT solver

Goal 1: even trigger fires in both programs

```
(declare-fun s0_idx () (_ BitVec 64))
(define-fun x!0 () (_ BitVec 64) (bvadd s0_idx (_ bv4 64)))
(define-fun x!1 () (_ BitVec 64) (bvurem x!0 (_ bv5 64)))
(define-fun x!2 () (_ BitVec 64) (bvmul (_ bv4 64) x!1))
...
```

Goal 2: ...

...

Goal 3: ...

...

More in the paper

- Handling floating-point operations (e.g., `sin`/`cos`) with SMT solvers
- How CopilotVerifier presents proof evidence for certification
  - Certification is a human-driven process, so we must produce evidence suitable for human auditors

Next steps

- Copilot has been released as Class D, open-source software at NASA

- Plan to use CopilotVerifier as part of safety cases for Class C NASA missions involving Copilot monitors

Next steps

- Copilot has been released as Class D, open-source software at NASA

- Plan to use CopilotVerifier as part of safety cases for Class C NASA missions involving Copilot monitors

- CopilotVerifier source: https://github.com/GaloisInc/copilot-verifier
  Copilot source: https://github.com/Copilot-Language/copilot

Next steps

- Copilot has been released as Class D, open-source software at NASA

- Plan to use CopilotVerifier as part of safety cases for Class C NASA missions involving Copilot monitors

- CopilotVerifier source: https://github.com/GaloisInc/copilot-verifier
  Copilot source: https://github.com/Copilot-Language/copilot

Thank you!

Backup slides

Handling floating-point ops with SMT solvers

- CopilotVerifier treats all floating-point operations (arithmetic, `sin`/`cos`, etc.) as uninterpreted functions at the SMT level
- This works, but it is brittle: the order of floating-point operations must be the exact same in both the stream and C programs
- For instance, these two stream expressions are *not* equivalent:

```
constantF :: Float -> Stream Float
```

```
constantF (150.0 / 255.0)
```

↓

```
0.5882353f
```

```
constantF 150.0 / constantF 255.0
```

↓

```
150.0f / 255.0f
```