Generating low-level code from high-level code for fast & verified programs

MFoCS seminar - 24/01/2023 - by Aron van Hof

Why is this relevant?

- Need for both safety and performance in e.g. cryptography
- Weighing safety of high-level code against performance of low-level code
- Solution: write (verified) high-level code and (high-performance) low-level code and manually prove they are equivalent
- However, this is tedious, an automated process would be preferred
- Today we will see 2 different approaches that try to bridge this gap by compiling high-level code to low-level code

Paper 1:

Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code.
By: Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen & Adam Chlipala

In: PLDI 2022

Paper 2:

Verified low-level programming embedded in F.*
By: Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet & Nikhil Swamy

In: ICFP 2017

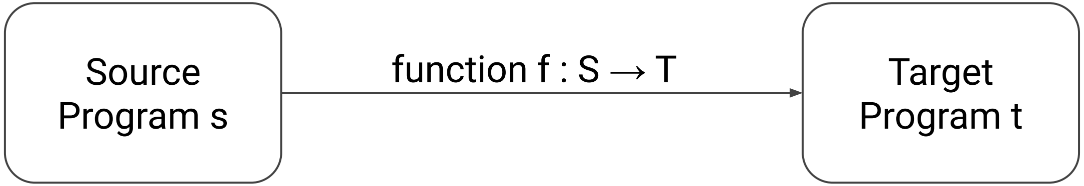
Paper 1: *Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code*

Outline paper 1

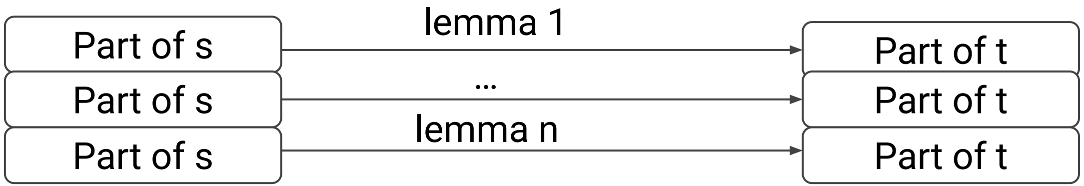
- Relational compilation
- Rupicola & the compilation pipeline
- Performance evaluation
- Limitations

Relational compilation (1)

- Given a source language S and target language T
- Traditional compilation: universal function f : S → T that preserves semantics



- Relational compilation: break f into separate lemmas to try to find semantically equivalent programs t and s, denoted t ~ s



Example

- Source language S describing arithmetic expressions:

```
Inductive S := SInt z | SAdd (s1 s2 : S).
```

- Target language T describing stack operations push & popadd:

```
Inductive T_Op := TPush z | TPopAdd.
Definition T := list T_Op.
```

Example

- Source language S describing arithmetic expressions:

```
Inductive S := SInt z | SAdd (s1 s2 : S).
```

- Target language T describing stack operations push & popadd:

```
Inductive T_Op := TPush z | TPopAdd.
Definition T := list T_Op.
```

- Define evaluation functions σS & σT:

$\sigma S : S \to \mathbb{Z}$

$\sigma T : T \to \text{list } \mathbb{Z} \to \text{list } \mathbb{Z}$

Example

- Source language S describing arithmetic expressions:

```
Inductive S := SInt z | SAdd (s1 s2 : S).
```

- Target language T describing stack operations push & popadd:

```
Inductive T_Op := TPush z | TPopAdd.
Definition T := list T_Op.
```

- Define evaluation functions σS & σT:

$\sigma S : S \to \mathbb{Z}$
$\sigma T : T \to \text{list } \mathbb{Z} \to \text{list } \mathbb{Z}$

- t ~ s if they evaluate to the same result for each initial stack:

$\forall\ zs, \sigma T\ t\ zs = \sigma S\ s :: zs$

Example: ordinary compilation

- Ordinary compilation as a single pass through the language instance, e.g.:

```
Fixpoint StoT (s : S) := match s with
  | SInt z        ⇒ [TPush z]
  | SAdd s1 s2 ⇒ StoT s1 ++ StoT s2 ++ [TPopAdd]
end.

Lemma StoT_ok : ∀ s, StoT s ~ s. Proof. … Qed.
```

Example: as lemmas

- Introduce a lemma for each relation:

```
Lemma StoT_SInt z :=  [TPush z] ~ SInt z.
Lemma StoT_Plus t1 s1 t2 s2:
  t1 ~ s1 → t2 ~ s2 →
  t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2.
```

Example: a simple source language program

- Example program s7 of language S, need to find t7:

```
Example s7 :=  SAdd (SInt 3) (SInt 4).
Example t7_rel: { t7 | t7 ~ s7 }.
```

- We use Compute to get the proof term after the proof has completed:

```
Lemma.
...
Defined.

Compute t7_rel.
```

Example: using lemmas to find target program

Proof steps:

Proof tree steps:

{ t7 | t7 ~ s7}

Example: using lemmas to find target program

Proof steps:

unfold s7.

Proof tree steps:

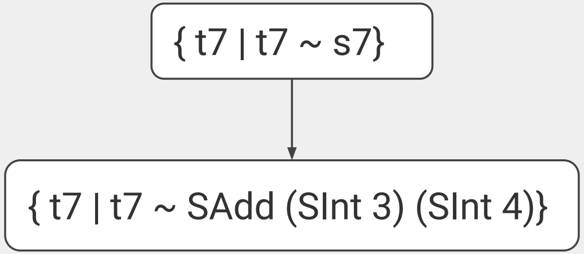
{ t7 | t7 ~ s7}

Example: using lemmas to find target program

Proof steps:

unfold s7.

Proof tree steps:



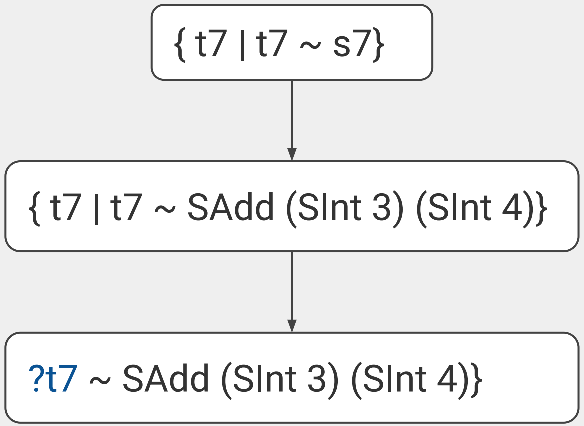
Example: using lemmas to find target program

Proof steps:

unfold s7.

eexists.

Proof tree steps:



Example: using lemmas to find target program

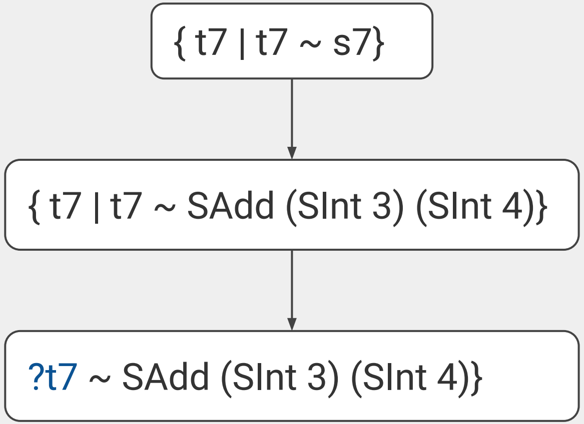
Proof steps:

unfold s7.

eexists.

?t7 ~ SAdd (SInt 3) (SInt 4)

Proof tree steps:



Example: using lemmas to find target program

Proof steps:

Proof tree steps:

?t7 ~ SAdd (SInt 3) (SInt 4)

Example: using lemmas to find target program

Proof steps:

Proof tree steps:

?t7 ~ SAdd (SInt 3) (SInt 4)

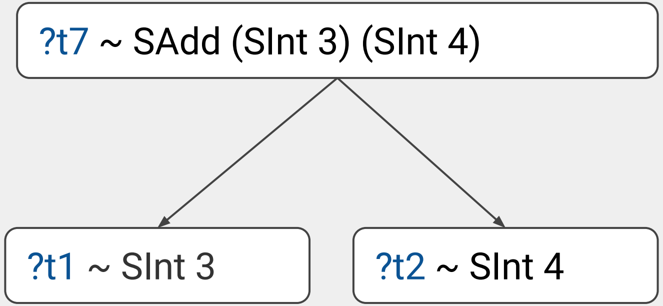
```
Lemma StoT_SInt z :=  [TPush z] ~ SInt z.
Lemma StoT_Plus t1 s1 t2 s2:
  t1 ~ s1 → t2 ~ s2 →
  t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2.
```

Example: using lemmas to find target program

Proof steps:

apply StoT_Plus.

Proof tree steps:



Example: using lemmas to find target program

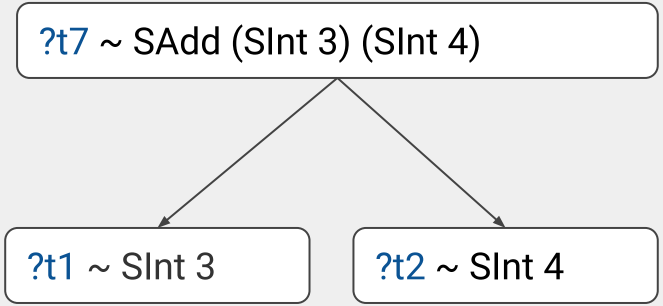
Proof steps:

apply StoT_Plus.

?t1 ~ SInt 3

?t2 ~ SInt 4

Proof tree steps:



Example: using lemmas to find target program

Proof steps:

Proof tree steps:

?t1 ~ SInt 3

?t2 ~ SInt 4

Example: using lemmas to find target program

Proof steps:

Proof tree steps:

?t1 ~ SInt 3

?t2 ~ SInt 4

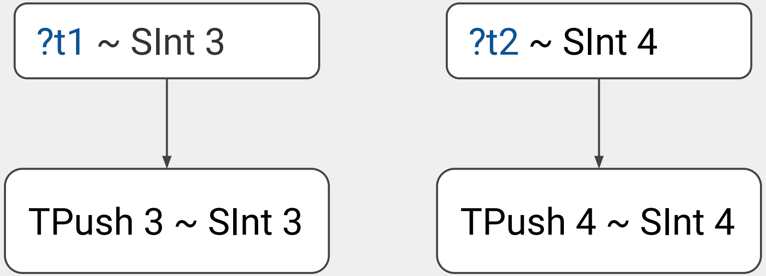
```
Lemma StoT_SInt z :=  [TPush z] ~ SInt z.
Lemma StoT_Plus t1 s1 t2 s2:
  t1 ~ s1 → t2 ~ s2 →
  t1 ++ t2 ++ [TPopAdd] ~ SAdd s1 s2.
```

Example: using lemmas to find target program

Proof steps:

```
all: apply StoT_SInt. Defined.
```

Proof tree steps:



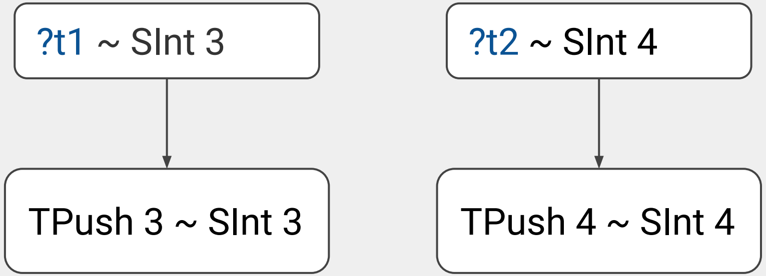
Example: using lemmas to find target program

Proof steps:

```
all: apply StoT_SInt. Defined.

Compute t7_rel.
= exist [TPush 3; TPush 4; TPopAdd]
```

Proof tree steps:



Relational compilation (2)

- We used lemmas to prove the existence of a target program
- Use Coq's automatic proof search for finding a program of the target language using the lemmas
- Soundness, but no completeness
- TL;DR: *a relational compiler is a collection of lemmas on semantic equivalences that can connect a source program to a target program*

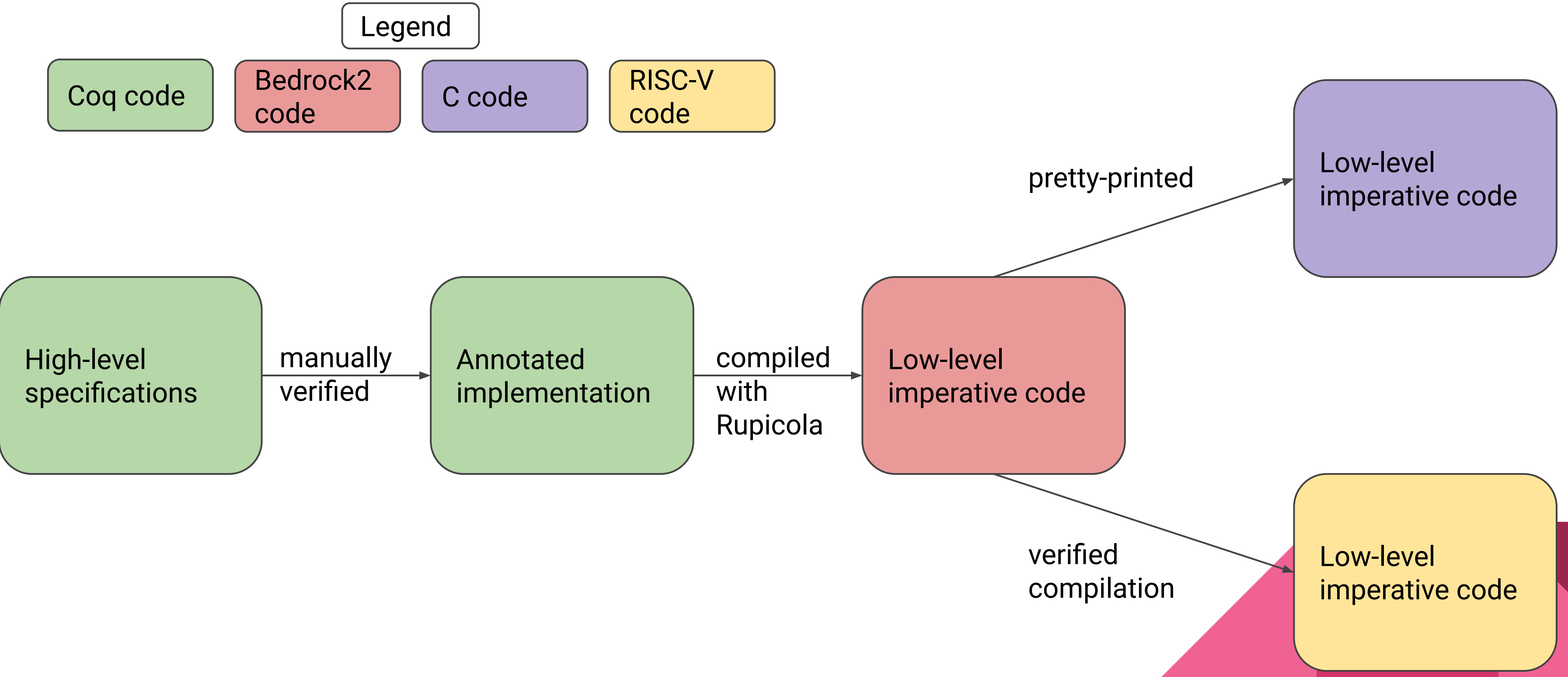
Relational compilation (3)

- What about compiling Coq code itself?
- A traditional compiler written in Coq cannot find an input type when the source language is Coq itself
- This is possible using relational compilation
- So relational compilation allows building a compiler for Coq within Coq itself!

Introducing Rupicola

- Rupicola: compiler-construction toolkit for compiling Coq to Bedrock2 (language similar to C)
- Implements idea of relational compilation by using proven lemmas to compile to the target language
- Users provide lemmas if a particular semantic equivalence has not yet been established
- Thus it is construction toolkit, rather than a general compiler itself

Compilation pipeline



Example: upstring

- Specification:

```
λs → String.map toupper s
```

- Annotated implementation:

```
λs → let/n s := ListArray.map(λb → a2b (toupper b2a b))) s in s
```

- Transformations:
  - In place mutation
  - For-loop rather than higher-order iteration
  - Different representation of strings

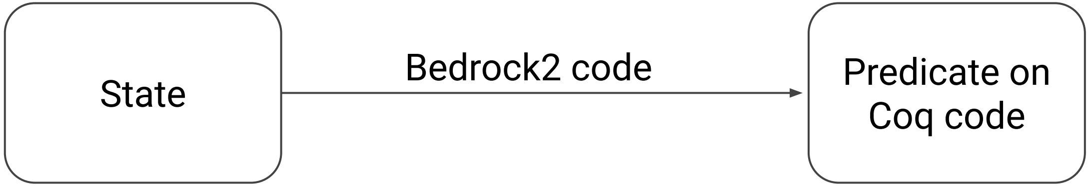
Example: upstring

- Generated C code:

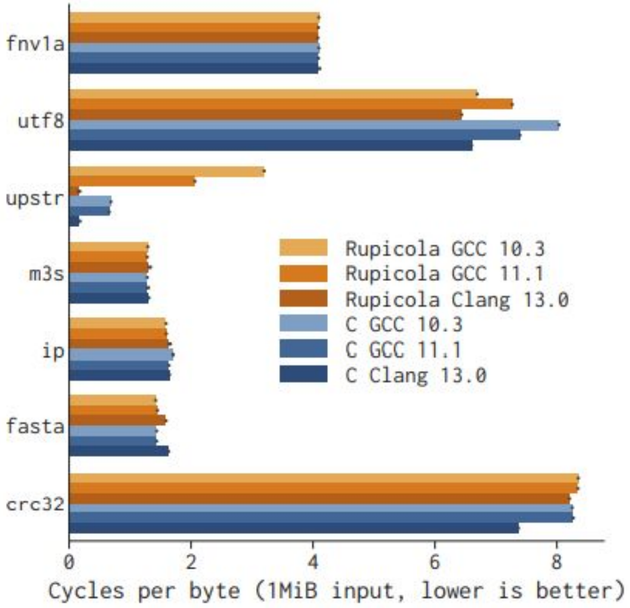
```
void upstr(uintptr_t as, uintptr_t len) {
  char *s = (char*s) as; int l = len;
  for (int pos = 0; pos < l; pos++) {
    s[pos] = (((unsigned)s[pos] - ‘a’) & 0xff) < 26 ? s[pos] & 0x5f : s[pos];
  }
}
```

Compiling with Rupicola

- Asks user to provide help when compilation initially fails
- A lemma can then be given, using a Hoare triple of the form:



Performance evaluation



Limitations

- Expertise in Coq, Bedrock2 and Rupicola needed
- Not all low-level patterns translate well to functional models
- There may still be bugs somewhere in the trusted computing base: Coq's proof checker & the pretty-printer from Bedrock2 to C

Paper 2: *Verified low-level programming embedded in F**

Outline paper 2

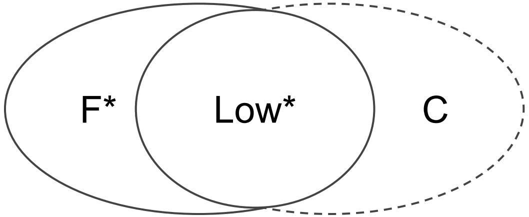
- F* & Low*
- The KaRaMeL compiler
- Modelling C in Low*
- Example: ChaCha20
- Performance evaluation
- Limitations

Introducing F*

- High-level functional language like Coq
- Supports dependent typing, user-defined monads & refined types
- Can also take the role of proof assistant through Coq-like tactics & automated proof search
- The F* ecosystem contains several Domain Specific Languages that each seek to fulfill a particular role, e.g. Low*, Steel, etc.
- This makes it more of a general purpose language than Coq

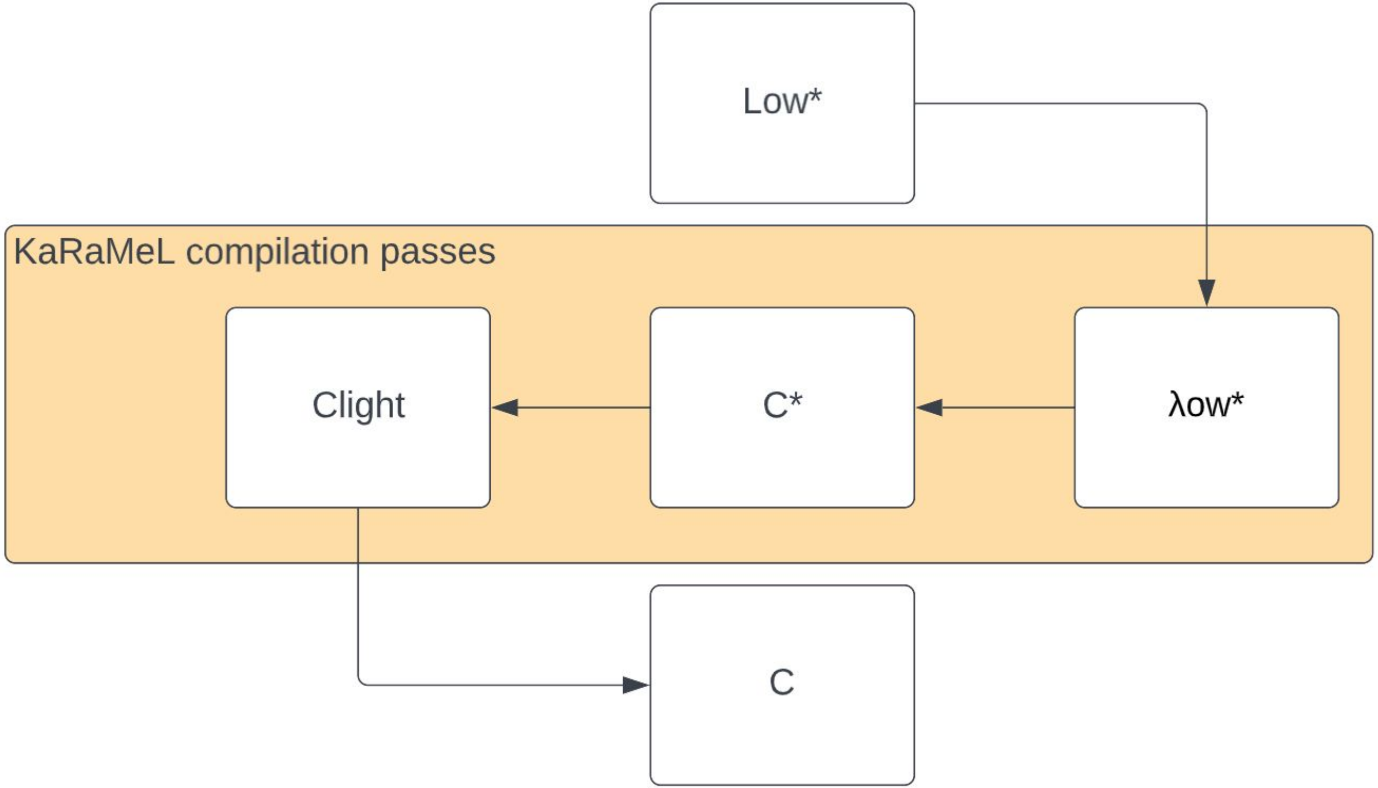
Introducing Low*

- Low* is a shallow embedding of (a subset of) C in F*
- Simulates C's memory model, arrays, etc.



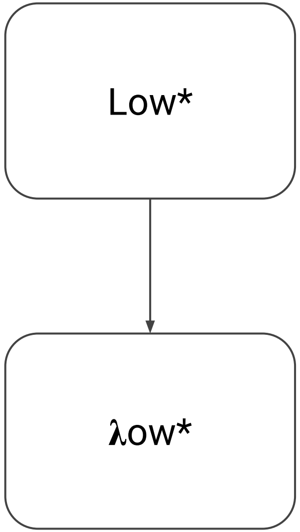
- Compiled to C using the KaRaMeL compiler
- Some type syntax: Tot & Ghost

The compilation process



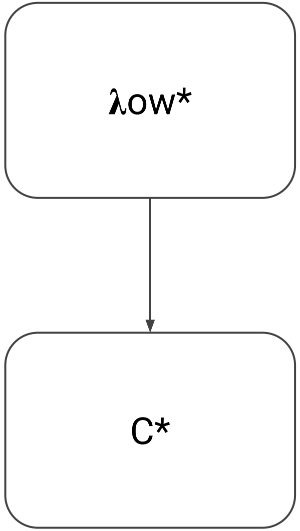
The compilation process: λow*

- λow*: establishes formal core of Low*
- Erases specifications & proofs



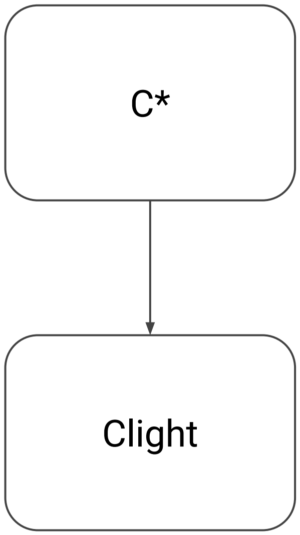
The compilation process: C*

- C*: intermediate language between λow* & Clight
- Syntax becomes more C-like
- Switches calling convention to explicitly push frame



The compilation process: Clight

- Clight: deterministic subset of C
- Hoists local variables
- Source language for CompCert
- Or use pretty-printer to C



Modeling C: memory model

- Start off by adding state through the F* state monad:

```
ST (a:Type) (requires pre: s → Type) (ensures post: s → a → s → Type)
```

- Essentially represents a function:

```
m0:s → (r:a, m1:s)
```

- Next we instantiate s with Hyperstack.mem

Memory model: hyper-stacks

- A hyper-stack partitions memory intro regions
- Each region has its own id and a predicate stating whether it is a stack or a heap region
- One stack region, root, outlives the other regions
- In code specification:

```
type rid
val is_stack_region: rid → Tot bool
type sid = r:rid{is_stack_region r}
type hid = r:rid{¬(is_stack_region r)}
val root: sid
```

Memory model: references

- Partial signature of the model:

```
type ref : Type → Type
val region_of: ref a → Ghost rid
val _∈_ : ref a → mem → Tot Type
val _ [_] : ref a → mem → Ghost a
val _ [_] ← _ : mem → ref a → a → Ghost mem
```

Example: ChaCha20

- Stream cipher for symmetric encryption;
- Computes pseudo-random block of bytes to encrypt
- We will see the Low* & C version

Example: ChaCha20

```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  …
  = …
```

```
void chacha20(
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)
  {
    …
  }
```

Example: ChaCha20

```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  …
  : Stack unit
    (requires (λm0 → output ∈ m0 ∧ key ∈ m0 ∧
     nonce ∈ m0))
    (ensures (λm0 _ m1 → modifies output m0 m1
     ∧ m1[output] ==
     Seq.prefix len(Spec.chacha20 m0[key]
     m0[nonce]) counter))) = …
```

```
void chacha20(
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)
  {
    …
  }
```

Example: ChaCha20

```
let chacha20
  (len: uint32{len ≤ blocklen})
  (output: bytes{len = output.length})
  …
  : Stack unit
    (requires (λm0 → output ∈ m0 ∧ key ∈ m0 ∧
     nonce ∈ m0))
    (ensures (λm0 _ m1 → modifies output m0 m1
     ∧ m1[output] ==
     Seq.prefix len(Spec.chacha20 m0[key]
     m0[nonce]) counter))) =
push_frame ();
let state = Buffer.create 0ul 32ul in
let block = Buffer.sub state 16ul 16ul in
chacha20_init block key nonce counter;
chacha20_update output state len;
pop_frame ()
```

```
void chacha20(
  uint32_t len,
  uint8_t *output,
  uint8_t *key,
  uint8_t *nonce,
  uint32_t counter)
  {





    uint32_t state[32] = { 0 };
    uint32_t *block = state + 16;
    chacha20_init(block, key, nonce, counter);
    chacha20_update(output, state, len);
  }
```

Performance evaluation

- High-assurance cryptographic library (HACL) for cryptographic primitives to test performance of C code generated by Low* & KaRaMeL in real-world setting
- Based on the NaCl API has characteristics like:
  - Only supports modern algorithms
  - Exposes general functions for certain functionality rather than specific algorithms

HACL* performance comparison

Algorithm	HACL*	Sodium	TweetNaCL	OpenSSL	eBACS fastest
ChaCha20	6.17 cy/B	6.97 cy/B	-	8.04 cy/B	1.23 cy/B
Salsa20	6.34 cy/B	8.44 cy/B	15.14 cy/B	-	1.39 cy/B
Poly1305	2.07 cy/B	2.48 cy/B	32.32 cy/B	2.16 cy/B	0.68 cy/B
Curve25519	157k cy/mul	162k cy/mul	1663k cy/mul	359k cy/mul	145 cy/mul
AEAD-ChaCha20-poly1305	8.37 cy/B	9.60 cy/B	-	8.53 cy/B	-
SecretBox	8.43 cy/B	11.03 cy/B	50.56 cy/B	-	-
Box	18.10 cy/B	20.97 cy/B	149.22 cy/B	-	-

Limitations

- Requires an understanding of F* and Low* languages as well as knowledge of low-level programming in C to utilize Low*
- Trusted Computing Base including F* type checking algorithm, the Z3 SMT solver used by F* and the KaRaMeL compiler

Conclusion

Summary of similarities & differences

	Rupicola	Low*
Programming to be done	Specification, annotated implementation and lemmas in high-level language Coq using Rupicola tool kit	Performance-critical parts in DSL Low* and proofs, specifications, etc. in high-level language F*
Compilation	Relational compilation from using Rupicola	Traditional compilation using separate program KaRaMeL
Correctness	Uses Coq proofs/typing, compilation is guaranteed to be sound	Uses F* proofs/typing
Trusted Computing Base	Coq and pretty-printer to C	F*, Z3 SMT solver & KaRaMeL

Q&A

Ask away!

Example: formal definitions

- Language definitions:

```
Inductive S := SInt z | SAdd (s1 s2 : S).
Inductive T_Op := TPush z | TPopAdd.
Definition T := list T_op.
```

- Example definition of σS:

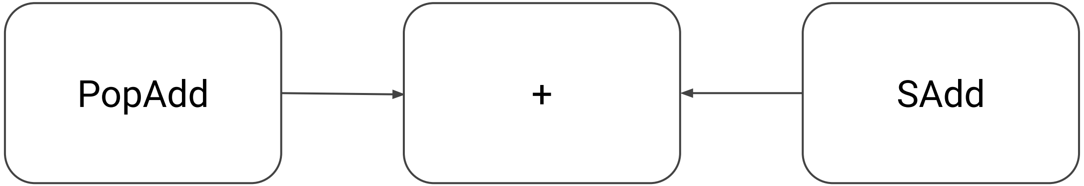
```
Fixpoint σS (s : S) :=
  match s with
  | SInt z         ⇒  z
  | SAdd s1 s2 ⇒  σS s1 + σS s2 end.
```

- Then t ~ s holds when:

```
∀ zs, σT t zs = σS s :: zs
```

Example

- Source language S describing arithmetic expressions
- Target language T describing stack operations push & popadd
- Define valuation functions σS & σT that map their operations to operations on $\mathbb{Z}$:



- t ~ s if they evaluate to the same result for each initial stack zs:

Low* restrictions

The code must:

- be first order to avoid allocating closures
- make heap allocations explicit
- not use recursive datatypes
- be monomorphic

Example: Dereferencing in heap

- Defining an operator ! for getting the value of the reference:

```
val (!): x:ref a → ST a (requires (λm → x∈m )) (ensures (λm0 y m1 → m0 = m1 ∧ y = m1[x]))
```

- Note how these F* features help guarantee correctness!

Modelling C: arrays

- Introduce a buffer type:

```
abstract type buffer a =
  | MkBuffer: max_length:uint32
    → content:ref (s:seq a{Seq.length s = max_length})
    → idx:uint32
    → length:uint32 {idx + length ≤ max_length} → buffer a
```