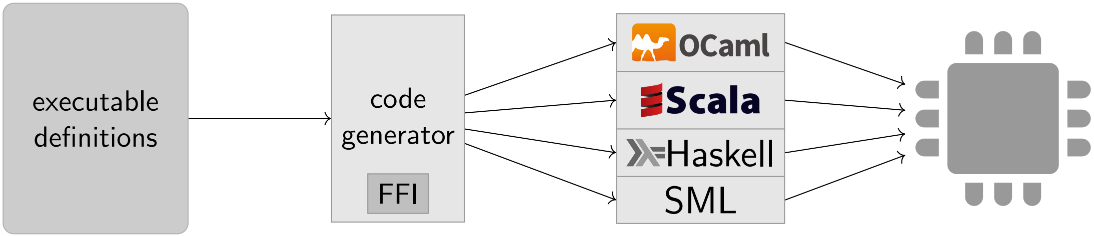
Fast machine words in

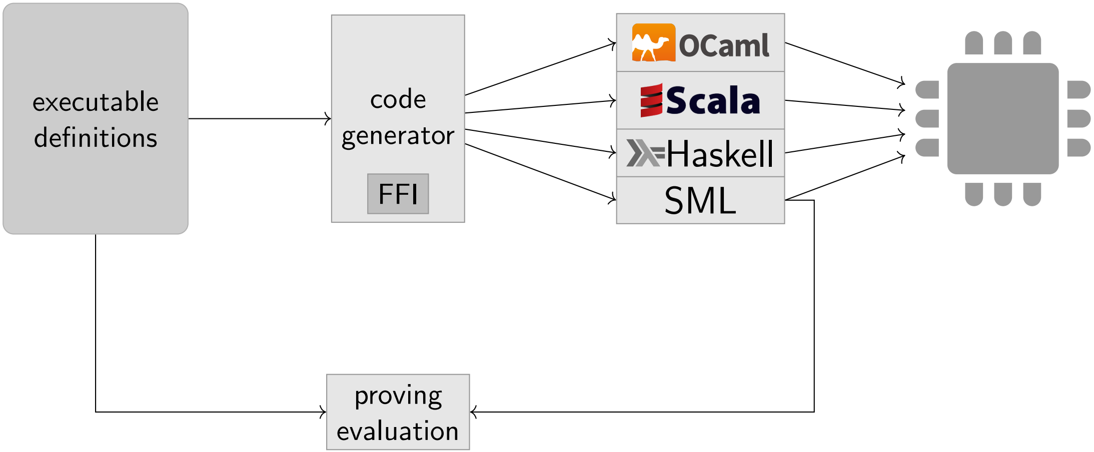
Andreas Lochbihler

Digital Asset (Switzerland) GmbH

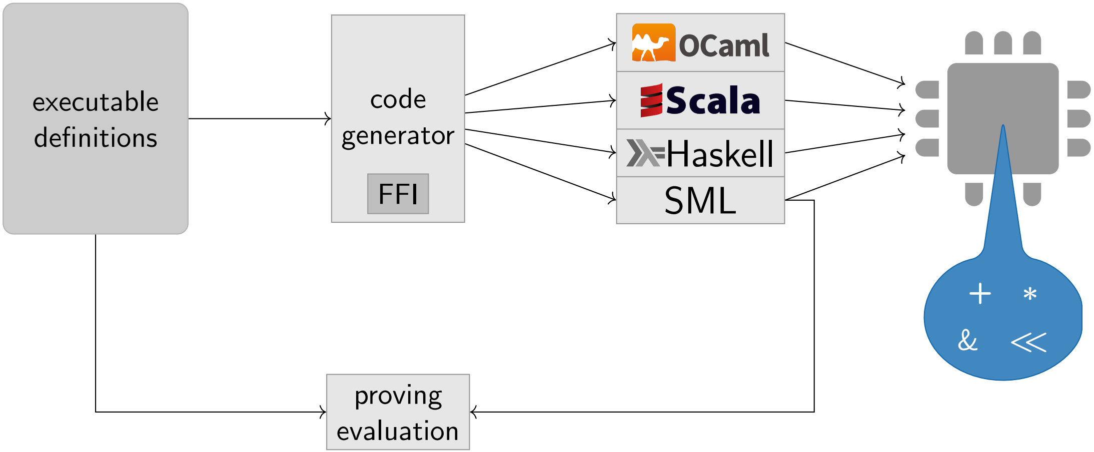
Code generation in Isabelle HOL



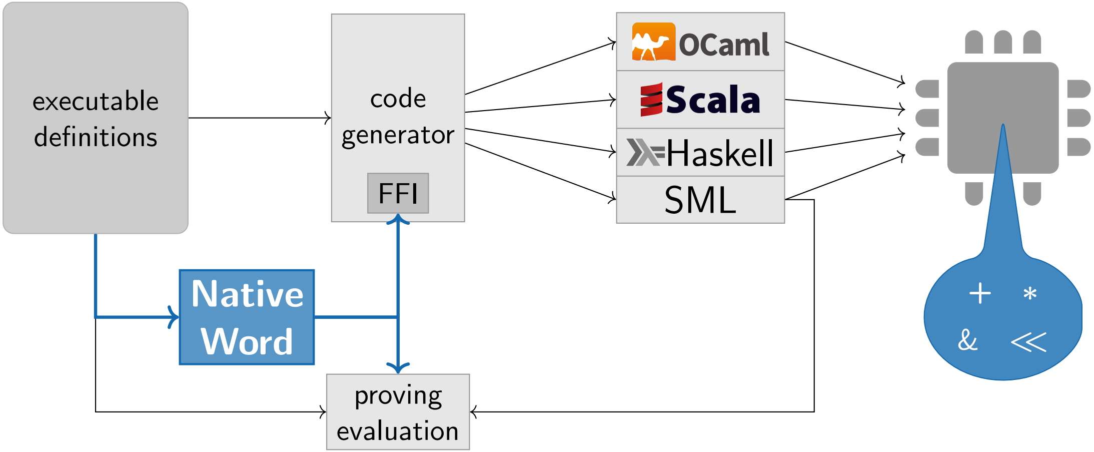
Code generation in Isabelle HOL



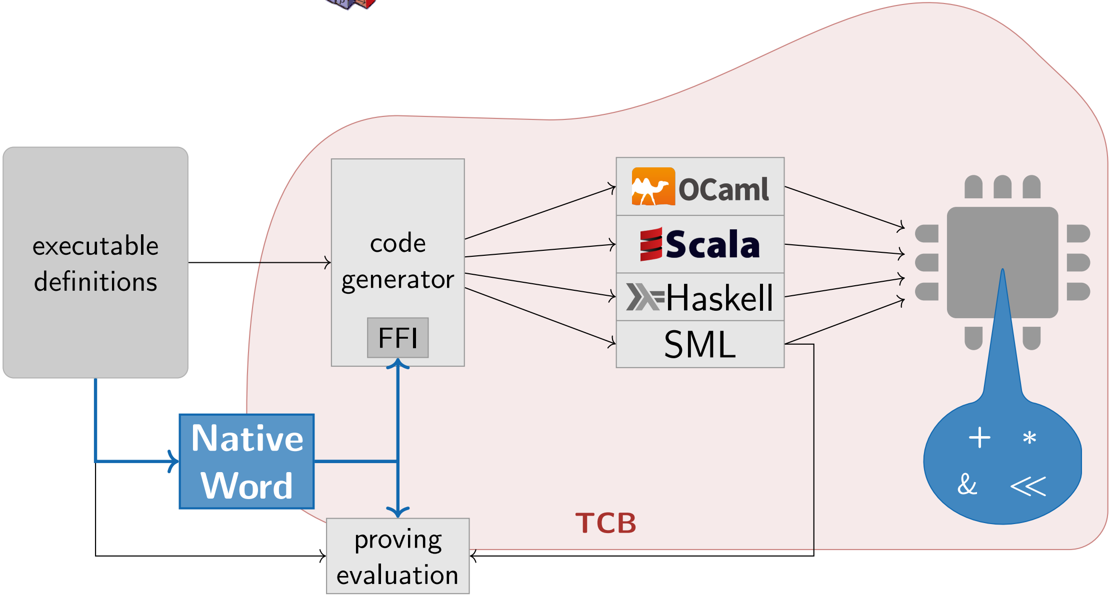
Code generation in Isabelle HOL



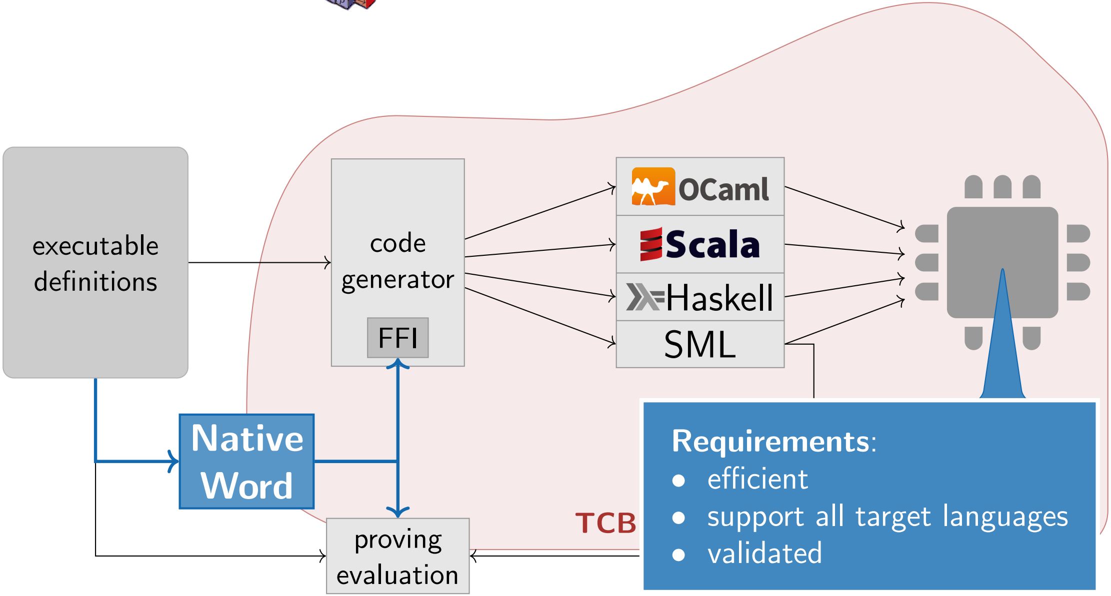
Code generation in Isabelle HOL



Code generation in Isabelle HOL

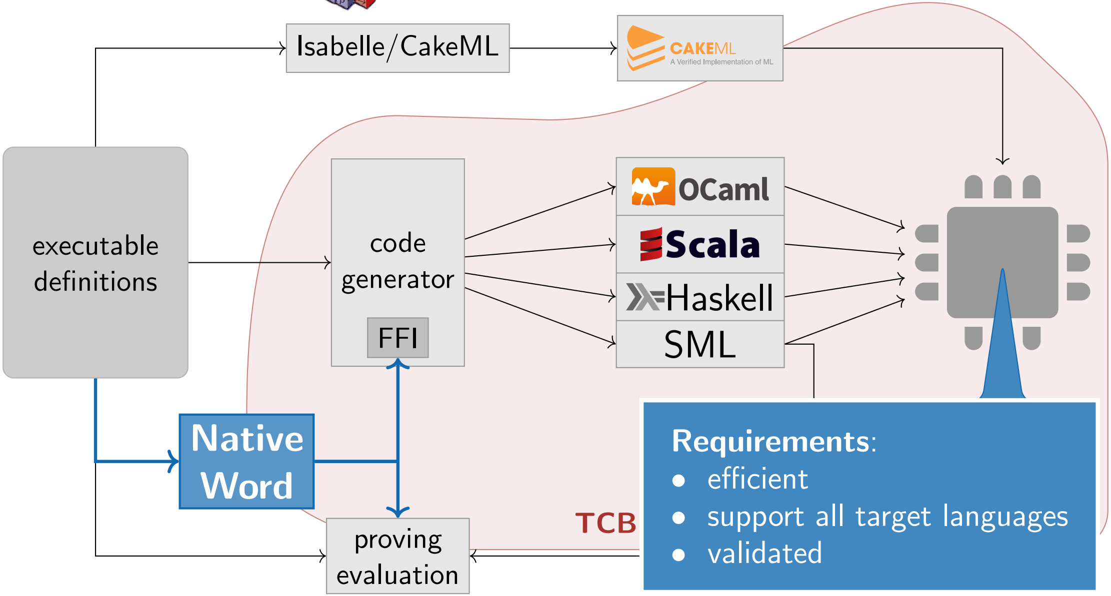


Code generation in



Requirements:
- efficient
- support all target languages
- validated

Code generation in Isabelle/HOL



Requirements:

- efficient
- support all target languages
- validated

What is the result on 32-bit words?

$$-5 \bmod 3 =$$

What is the result on 32-bit words?

$$-5 \bmod 3 = \begin{cases} \text{Isabelle} & 2 \\ \text{OCaml} & 1 \\ \text{Scala} & 1 \\ \text{Haskell} & 2 \\ \text{SML} & 2 \end{cases}$$

What is the result on 32-bit words?

$$-5 \bmod 3 = \begin{cases} \text{Isabelle} & 2 \\ \text{OCaml} & 1 \\ \text{Scala} & 1 \\ \text{Haskell} & 2 \\ \text{SML} & 2 \end{cases}$$

$$1 \ll (2^{31} + 1) =$$

What is the result on 32-bit words?

$$-5 \bmod 3 = \begin{cases} \text{Isabelle} & 2 \\ \text{OCaml} & 1 \\ \text{Scala} & 1 \\ \text{Haskell} & 2 \\ \text{SML} & 2 \end{cases}$$

$$1 \ll (2^{31} + 1) = \begin{cases} \text{Isabelle} & 0 \\ \text{OCaml} & \text{unspecified} \\ \text{Scala} & 2 \\ \text{Haskell} & \text{unspecified} \\ \text{SML} & \text{implementation-defined} \end{cases}$$

What is the result on 32-bit words?

$$-5 \bmod 3 = \begin{cases} \text{Isabelle} & 2 \\ \text{OCaml} & 1 \\ \text{Scala} & 1 \\ \text{Haskell} & 2 \\ \text{SML} & 2 \end{cases}$$

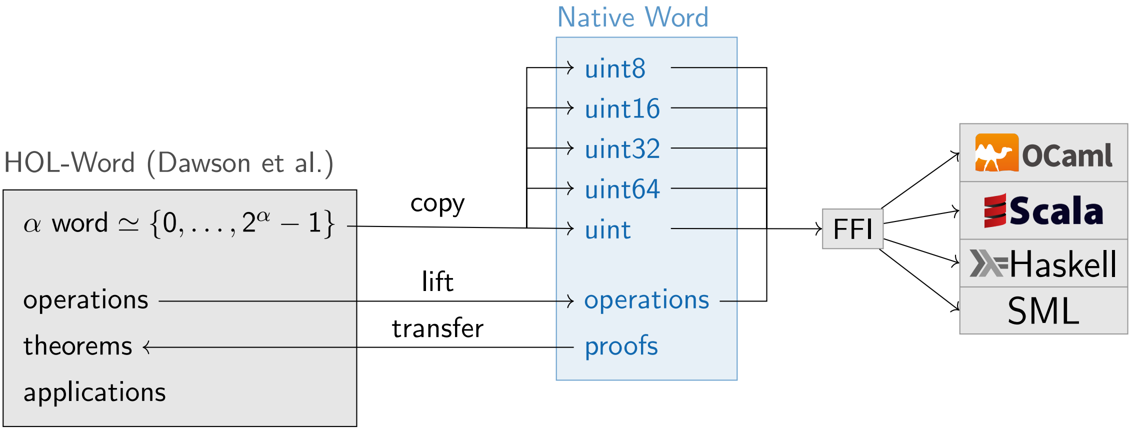
$$1 \ll (2^{31} + 1) = \begin{cases} \text{Isabelle} & 0 \\ \text{OCaml} & \text{unspecified} \\ \text{Scala} & 2 \\ \text{Haskell} & \text{unspecified} \\ \text{SML} & \text{implementation-defined} \begin{cases} \text{PolyML 32-bit} & 2 \\ \text{PolyML 64-bit} & 0 \end{cases} \end{cases}$$

Available bit-widths

bits	PolyML		SMLNJ	mlton	OCaml		GHC	Scala
	32	64			32	64		
8	√	√	√	√			√	√
16				√			√	√
32	√	√	√	√	√	√	√	√
64		√	√	√	√	√	√	√
default	31	63	31	32	31	63	≥ 30	32

(grey shading) = signed operations only

Let's abstract over these differences I



Let's abstract over these differences II

Conventional approach

1. Identify subset of common behaviour

```
definition divmod-abs x y =
  (|x| div |y| , |x| mod |y|)
```

2. Reduce to restricted behaviour

```
lemma [code]: divmod x y =
  ... if sgn x = sgn y then divmod-abs x y
      else ...
```

3. Common FFI for all languages

```
code-printing divmod-abs →
  (Haskell) divMod (abs _) (abs _)
  (OCaml) ...
  (Scala) ...
  (SML) ...
```

Let's abstract over these differences II

Conventional approach

1. Identify subset of common behaviour

```
definition divmod-abs x y =
   (|x| div |y| , |x| mod |y|)
```

2. Reduce to restricted behaviour

```
lemma [code]:  divmod x y =
   ... if sgn x = sgn y then divmod-abs x y
      else ...
```

3. Common FFI for all languages

```
code-printing divmod-abs →
   (Haskell) divMod (abs _) (abs _)
   (OCaml) ...
   (Scala) ...
   (SML) ...
```

2 case distinctions on the sign of each operand

PolyML: 2X slowdown

Let's abstract over these differences II

Conventional approach

1. Identify subset of common behaviour

```
definition divmod-abs x y =
  (|x| div |y| , |x| mod |y|)
```

2. Reduce to restricted behaviour

```
lemma [code]: divmod x y =
  ... if sgn x = sgn y then divmod-abs x y
      else ...
```

3. Common FFI for all languages

```
code-printing divmod-abs →
  (Haskell) divMod (abs _) (abs _)
  (OCaml) ...
  (Scala) ...
  (SML) ...
```

Cascading

1. Model behaviours of target languages

```
definition uint32-div x y = ...
definition uint32-sdiv x y = ...
```

2. Build cascade of models

```
lemma [code]:
  div x y = ... uint32-div ...
  uint32-div x y = ... uint32-sdiv ...
```

3. One FFI for each language

```
code-printing uint32-div →
  (Haskell) Prelude.div
code-printing uint32-sdiv →
  (OCaml) Int32.div
code-printing ... → ...
```

Let's abstract over these differences II

Conventional approach

1. Identify subset of common behaviour

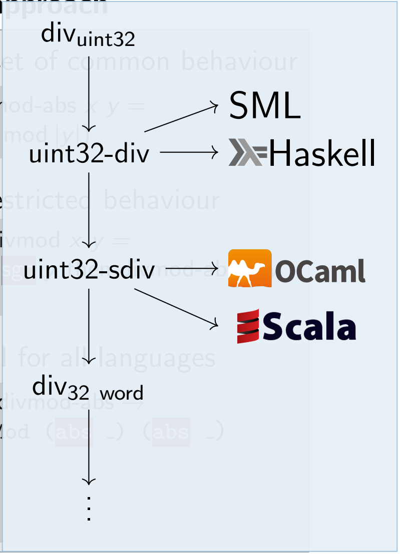
```
definition divmod-abs x y =
  (|x| div |y| , |x| mod |y|)
```

2. Reduce to restricted behaviour

```
lemma [code]: divmod x y =
  ... if sgn x = [illegible]
      else ...
```

3. Common FFI for all languages

```
code-printing divmod-abs →
  (Haskell) divMod (abs _) (abs _)
  (OCaml) ...
  (Scala) ...
  (SML) ...
```



Cascading

1. Model behaviours of target languages

```
definition uint32-div x y = ...
definition uint32-sdiv x y = ...
```

2. Build cascade of models

```
lemma [code]:
  div x y = ... uint32-div ...
  uint32-div x y = ... uint32-sdiv ...
```

3. One FFI for each language

```
code-printing uint32-div →
  (Haskell) Prelude.div
code-printing uint32-sdiv →
  (OCaml) Int32.div
code-printing ... → ...
```

What about unspecified behaviour?

Underspecification in OCaml

x << n is **undefined**
if $n > 32$

← `code-printing`

Underspecification in HOL

`definition` uint32-shiftl x n =
  if $n \leq 32$ then $x \ll n$
  else **undefined** $(\ll)$ x n

`lemma` [*code*]: $x \ll n =$
  if $n \leq 32$ then uint32-shiftl x n else 0

Underspecification leads to refinement

HOL axioms
definitions

Correctness w/o underspecification:
If code c terminates with result r,
then we can derive c = r.

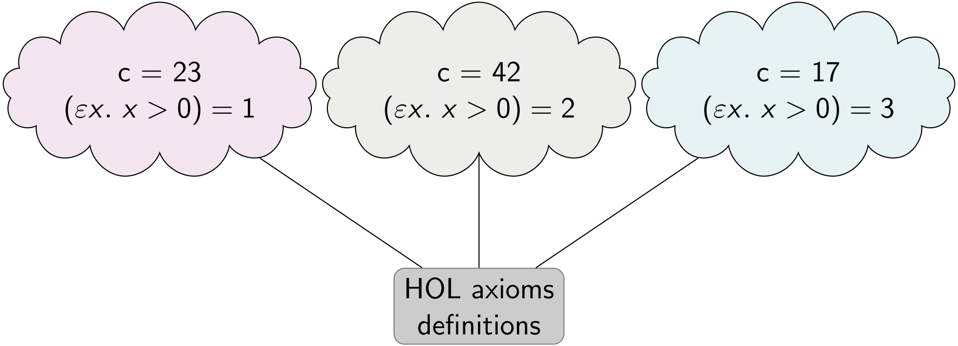
Underspecification leads to refinement

HOL axioms
definitions

Correctness w/o underspecification:
If code c terminates with result r,
then we can derive $c = r$.

Correctness with underspecification:
Every derivable property of the code c
applies to the result r.

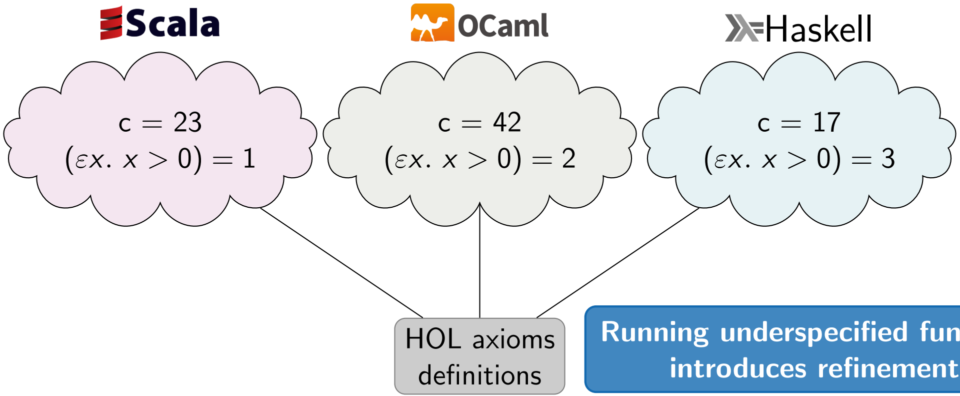
Underspecification leads to refinement



Correctness w/o underspecification:
If code c terminates with result r, then we can derive c = r.

Correctness with underspecification:
Every derivable property of the code c applies to the result r.

Underspecification leads to refinement



Running underspecified functions introduces refinement!

Correctness w/o underspecification:
If code c terminates with result r, then we can derive $c = r$.

Correctness with underspecification:
Every derivable property of the code c applies to the result r.

Underspecification leads to refinement

Scala

$c = 23$

$(\varepsilon x.\ x > 0) = 1$

OCaml

$c = 42$

$(\varepsilon x.\ x > 0) = 2$

Haskell

$c = 17$

$(\varepsilon x.\ x > 0) = 3$

HOL axioms definitions

Forbid underspecification for proofs!

Running underspecified functions introduces refinement!

Correctness w/o underspecification:
If code c terminates with result r, then we can derive $c = r$.

Correctness with underspecification:
Every derivable property of the code c applies to the result r.

Default word size with underspecified bit width

bits	PolyML		SMLNJ	mlton	OCaml		GHC	Scala
	32	64			32	64		
8	√	√	√	√			√	√
16				√			√	√
32	√	√	√	√	√	√	√	√
64		√	√	√	√	√	√	√
default	**31**	**63**	**31**	**32**	**31**	**63**	$>$ **30**	**32**

Default word size with underspecified bit width

bits	PolyML		SMLNJ	mlton	OCaml		GHC	Scala
	32	64			32	64		
8	√	√	√	√			√	√
16				√			√	√
32	√	√	√	√	√	√	√	√
64		√	√	√	√	√	√	√
uint → **default**	**31**	**63**	**31**	**32**	**31**	**63**	$>$ **30**	**32**

Unspecified bit size

Default word size with underspecified bit width

bits	PolyML 32	PolyML 64	SMLNJ	mlton	OCaml 32	OCaml 64	GHC	Scala
8	√	√	√	√			√	√
16				√			√	√
32	√	√	√	√	√	√	√	√
64		√	√	√	√	√	√	√
uint → **default**	**31**	**63**	**31**	**32**	**31**	**63**	$>$ **30**	**32**

Unspecified bit size

- hashing
- bit vectors
- dynamic implementation choices based on input size

Validation

- Framework to run test cases from within Isabelle/HOL

  `test-code` $\;251\ \mathsf{div}\ 3 = 83\;$ `in` Scala

Validation

- Framework to run test cases from within Isabelle/HOL

  `test-code` $251 \operatorname{div} 3 = 83$ `in` Scala SMLNJ MLton GHC PolyML

- Test cases for all operations on uint$*$

Validation

- Framework to run test cases from within Isabelle/HOL

  `test-code` $251 \operatorname{div} 3 = 83$ **in** Scala SMLNJ MLton GHC PolyML

- Test cases for all operations on uint$*$

- Revealed many errors in the FFI mapping – now fixed
- Found one error in PolyML 5.6 in 64-bit mode – fixed in 5.7

  $18446744073709551611 \operatorname{div} 3$ evaluates to 1431655763

Usage and Benchmarks

Usage:

- IsaFoR (Berlekamp-Zassenhaus)
- Fleury's verified SAT solver
- CAVA model checker
- Züst's TLS experiment

Usage and Benchmarks

Usage:

- IsaFoR (**Berlekamp-Zassenhaus**)
- Fleury's verified SAT solver
- CAVA model checker
- Züst's TLS experiment

Factor 400 polynomials over $\mathbb{Z}/p^k\mathbb{Z}$

Strategies:

1. Use unbounded GMP integers int.
2. If $p^k < 2^{16}$ use uint32.
   If $p^k < 2^{32}$ use uint64.
   Else use int
3. If $p^k < 2^{\text{default}/2}$ use uint.
   Else use int.

Usage and Benchmarks

Usage:

- IsaFoR (**Berlekamp-Zassenhaus**)
- Fleury's verified SAT solver
- CAVA model checker
- Züst's TLS experiment

Factor 400 polynomials over $\mathbb{Z}/p^k\mathbb{Z}$

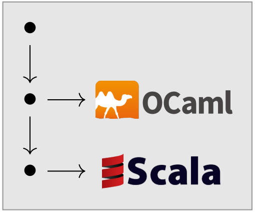
Strategies:

1. Use unbounded GMP integers int.
2. If $p^k < 2^{16}$ use uint32.
   If $p^k < 2^{32}$ use uint64.
   Else use int
3. If $p^k < 2^{\text{default}/2}$ use uint.
   Else use int.

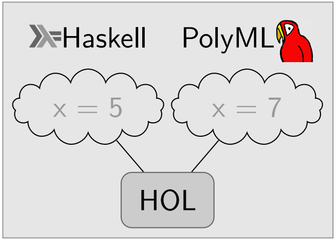
GHC	2 is **18 % faster** than 1.
PolyML	3 is **4 % faster** than 2.

Takeaways

1. Cascade pattern

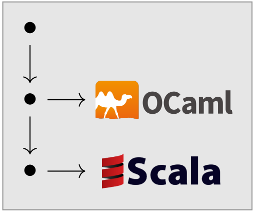


2. Model-theoretic underspecification

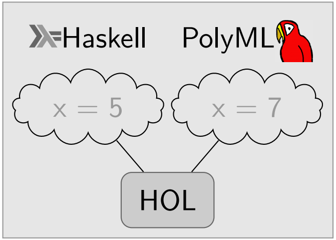


Takeaways

1. Cascade pattern



2. Model-theoretic underspecification



Try it out!

Native Word

in the Archive of Formal Proofs

www.isa-afp.org/entries/Native_Word.html

Testing framework

in the Isabelle distribution

HOL-Library.Code_Test