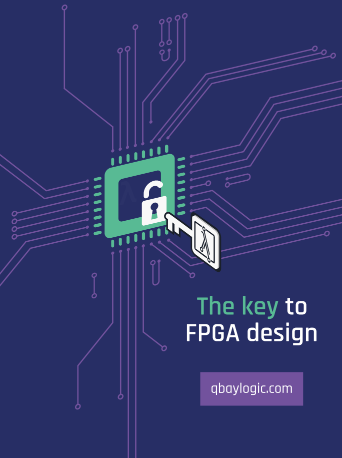
Creators of the
Clash compiler



ghc-typelits-proof-assist

A Haskell Plugin for Type-Nat Proofs

Felix Klein

HIW – June 6th, 2025

The key to FPGA-ASIC design

Problem

```haskell
import GHC.TypeNats

constrainedOp :: n ≤ m ⇒ SNat n → SNat m → ()
constrainedOp _ _ = ()

cOpApp :: n + 1 ≤ m ⇒ SNat n → SNat m → ()
cOpApp = constrainedOp
```

```
<interactive>:13:11: error: [GHC-64725]
  • Cannot satisfy: n <= m
  • In the expression: constrainedOp
  • In an equation for ‘cOpApp’: cOpApp = constrainedOp
```

Existing (partial) Solutions

- `ghc-typelits-natnormalise`
- `ghc-typelits-extra`
- `ghc-typelits-presburger`
- `ghc-typelits-sopsat`

They work well for "*simple*" expressions, but reach their limits as soon as the required deductions get too complex.

Any alternative solutions? 😈 "`unsafeCoerce`" 😈

Idea of this plugin: Use an external proof assistant to create the required evidence.

How does it work

```
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE UndecidableSuperClasses #-}

...

instance
  ( n + 1 ≤ m
  ) ⇒ Lemma n m
class -- =>
  ( n ≤ m
  ) ⇒ Lemma n m
{-/ Proof (Coq): Lemma /-}
```

1. Creates "`Lemma.v`", a Coq proof file with the signature of the proof.

How does it work

```
Require Import Coq.Init.Nat.
Require Import Coq.Init.Peano.

Require Import Coq.Arith.PeanoNat.

Lemma hsLemma: forall n m, n + 1 <= m -> n <= m.
```

2. Develop some proof in your favorite Coq IDE.

How does it work

```
...

Lemma hsLemma: forall n m, n + 1 <= m -> n <= m.
  intros.
  apply le_S in H.
  apply le_S_n.
  rewrite <- PeanoNat.Nat.add_0_l at 1.
  rewrite PeanoNat.Nat.add_comm.
  rewrite PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  rewrite <- PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  apply H.
Qed.
```

3. Copy back the proof into the Haskell "*proof comment*".

How does it work

```
...

instance
  ( n + 1 ≤ m
  ) ⇒ Lemma n m
class -- =>
  ( n ≤ m
  ) ⇒ Lemma n m
{-/ Proof (Coq): Lemma
  intros.
  apply le_S in H.
  apply le_S_n.
  rewrite <- PeanoNat.Nat.add_0_l at 1.
  rewrite PeanoNat.Nat.add_comm.
  rewrite PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  rewrite <- PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  apply H.
/-}
```

The QED Class

```
data Rewrite (c :: Constraint) = c ⇒ Rewrite

class c ⇒ QED (c :: Constraint) where
  apply :: (c ⇒ a) → a
  apply x = x

  using :: Rewrite c
  using = Rewrite
```

⇒ Enables the usage and distribution of the created evidence.

The QED Class

```
...

instance
  ( n + 1 ≤ m
  ) ⇒ Lemma n m
class -- =>
  ( n ≤ m
  ) ⇒ Lemma n m
{-/ Proof (Coq): Lemma
  intros.
  apply le_S in H.
  apply le_S_n.
  rewrite <- PeanoNat.Nat.add_0_l at 1.
  rewrite PeanoNat.Nat.add_comm.
  rewrite PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  rewrite <- PeanoNat.Nat.add_succ_l.
  rewrite PeanoNat.Nat.add_comm.
  apply H.
/-}
instance Lemma n m ⇒ QED (Lemma n m)
```

Using the Evidence

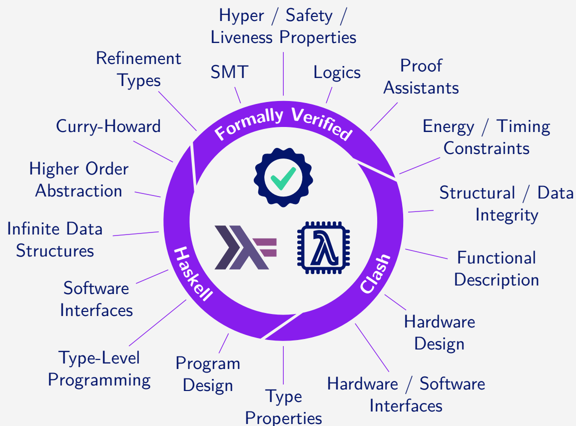
```
-- using 'Rewrite'
cOpApp :: forall n m. n + 1 ≤ m ⇒ SNat n → SNat m → ()
cOpApp | Rewrite ← using @(Lemma n m) = constrainedOp

-- using 'apply'
cOpApp :: forall n m. n + 1 ≤ m ⇒ SNat n → SNat m → ()
cOpApp = apply @(Lemma n m) constrainedOp

-- using the AutoProve feature of the plugin (limited)
usingAuto :: n + 1 ≤ m ⇒ SNat n → SNat m → ()
usingAuto = constrainedOp
```

Next Steps

- Custom type family support
- Support more proof assistants (currently: Agda, Coq)
- GHC 9.12 / 9.14 support (currently: only 9.10)
- Integrate with the other existing plugins

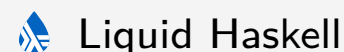
Clash Formal

Ecosystem formally verifiable IT



Proof Assistants

SMT / Refinement Types



Specification Frameworks

https://clash-formal.org