Conjecturing, Testing and Reasoning about Programs

Workshop on Verification and Program Transformation

27 March 2021

Moa Johansson, Chalmers University of Technology

CHALMERS
UNIVERSITY OF TECHNOLOGY

Introduction:

Theory Exploration

- **Discover interesting properties** about programs.
- **Build** type-correct **terms**. (start small, only non-redundant!)
- **Test & evaluate** on ground values.
- **Assemble equations.** (send to prover if you want)
- Use as lemmas, rewrite rules, equational specification…

Demo: Prove my homework

Or how to automate a CS undergraduate

```
fun sorted :: "nat list ⇒ bool"
  where "sorted [] = True"
  | "sorted [x] = True"
  | "sorted (x1#x2#xs) = ((x1 ≤ x2) ∧ sorted (x2#xs))"

fun ins :: "nat ⇒ nat list ⇒ nat list"
  where "ins x [] = [x]"
  | "ins x (y#ys) = (if (x ≤ y) then (x#y#ys) else y#(ins x ys))"

fun isort ::  "nat list ⇒ nat list"
  where "isort [] = []"
  | "isort (x#xs) = ins x (isort xs)"

theorem my_homework: "sorted (isort x)"
```

Demo: Prove my homework

First attempt: call Sledgehammer

```
theorem my_homework: "sorted (isort x)"
  sledgehammer
```

```
Sledgehammering...
"cvc4": Timed out
"vampire": Timed out
```

Sorted.thy (~/Desktop/)

```
fun sorted :: "nat list ⇒ bool"
  where "sorted [] = True"
  | "sorted [x] = True"
  | "sorted (x1#x2#xs) = ((x1 ≤ x2) ∧ sorted (x2#xs))"

fun ins :: "nat ⇒ nat list ⇒ nat list"
  where "ins x [] = [x]"
  | "ins x (y#ys) = (if (x ≤ y) then (x#y#ys) else y#(ins x ys))"

fun isort ::  "nat list ⇒ nat list"
  where "isort [] = []"
  | "isort (x#xs) = ins x (isort xs)"


theorem my_homework: "sorted (isort x)"
  sledgehammer

end
```

Proof state | Auto update | Update | Search: | 100%

```
Sledgehammering...
"cvc4": Timed out
"vampire": Timed out
```

Sidekick State Theories

Demo: Prove my homework

Six lemmas found and proved

lemma_a: sorted (ins x y) $\Longrightarrow$ sorted y

lemma_aa: sorted y $\Longrightarrow$ sorted (ins x y)

Key lemma

lemma_ab: ins y (ins x z) = ins x (ins y z)

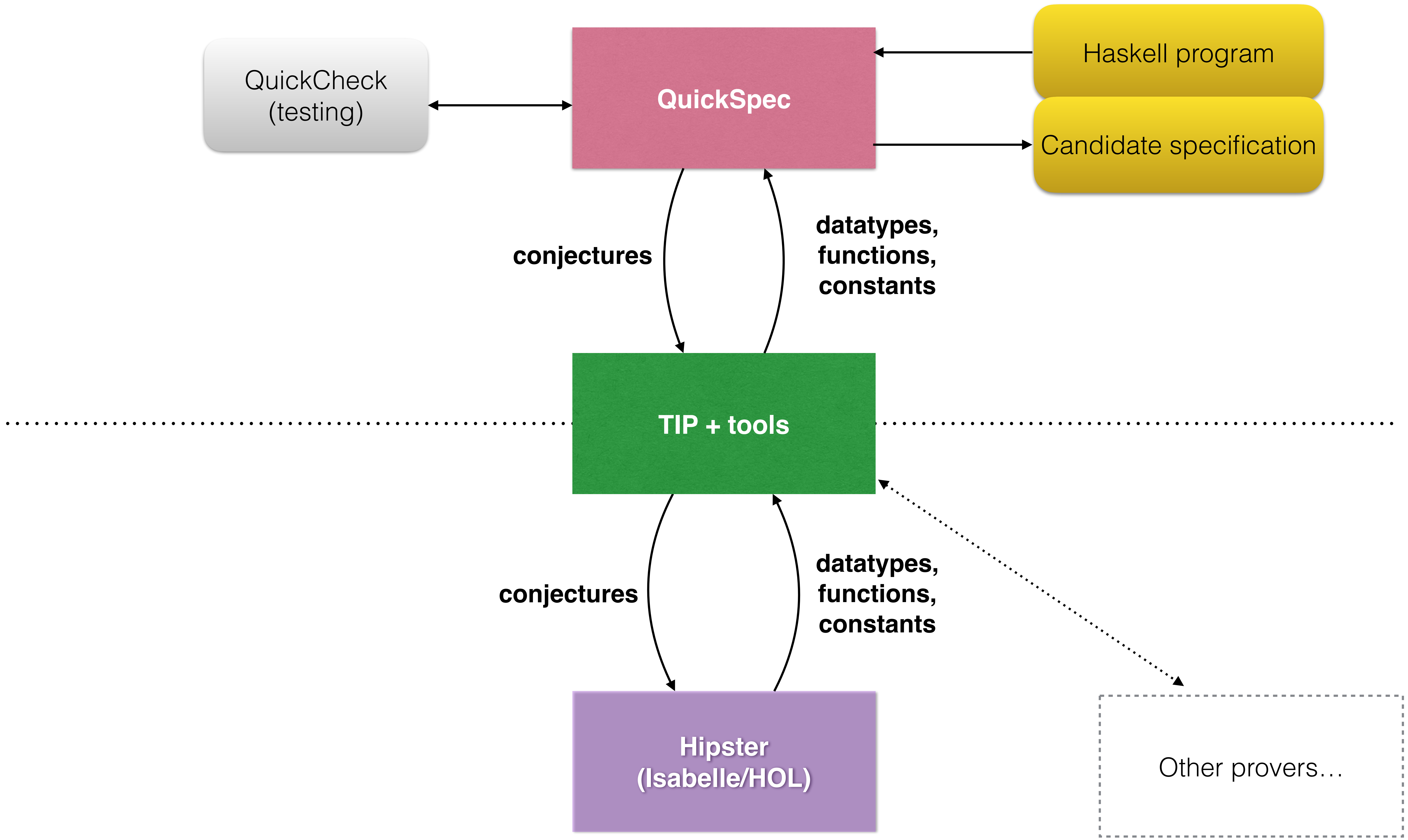
lemma_ac: sorted (isort x)
(* Needs lemma_aa *)

Our homework assignment

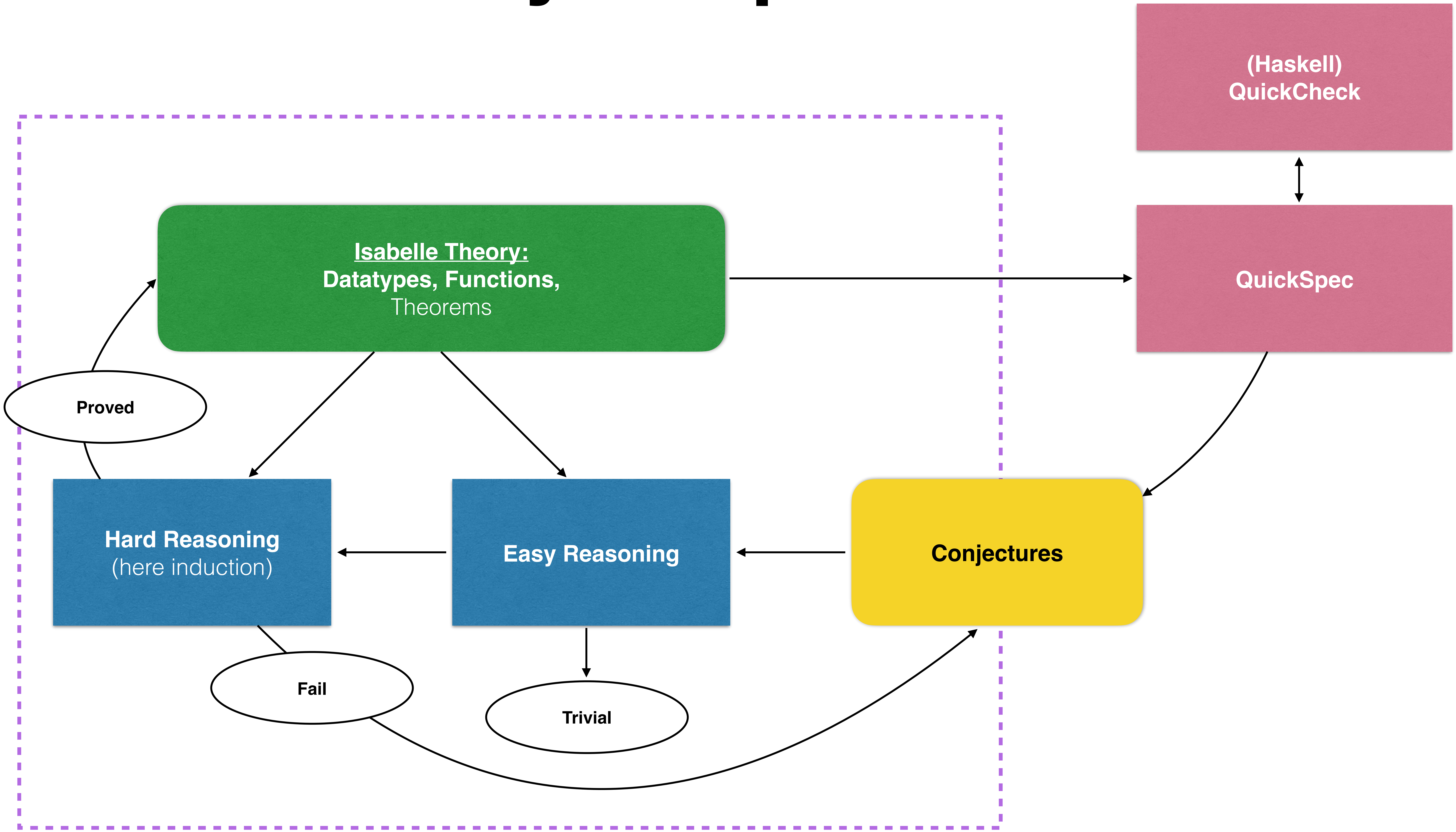
lemma_ad: "isort (ins x y) = ins x (isort y)"
(Needs lemma_ab *)*

lemma_ae: "isort (isort x) = isort x"
(Needs lemma_ad *)*

Architecture



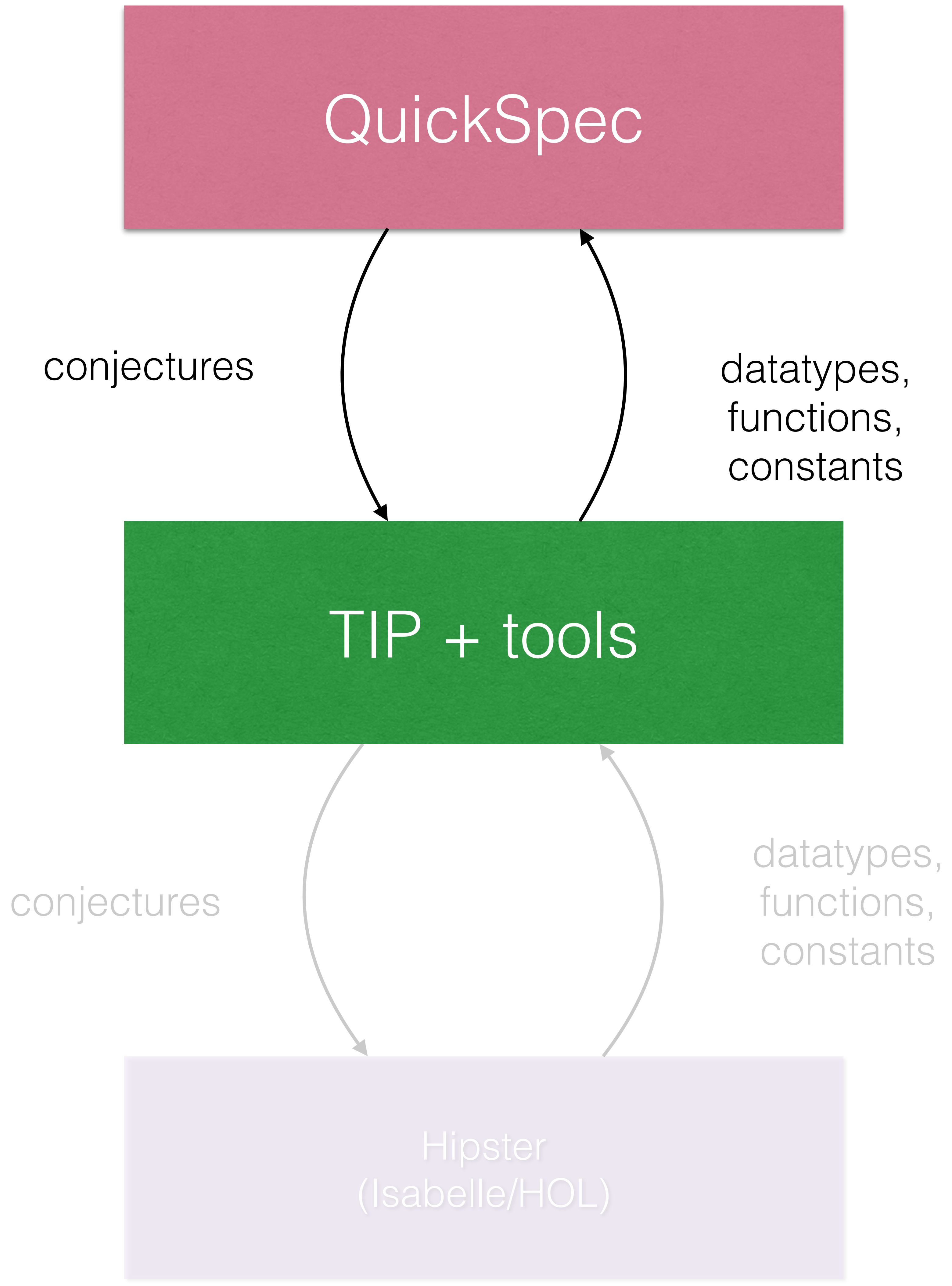
Theorem Discovery in Hipster

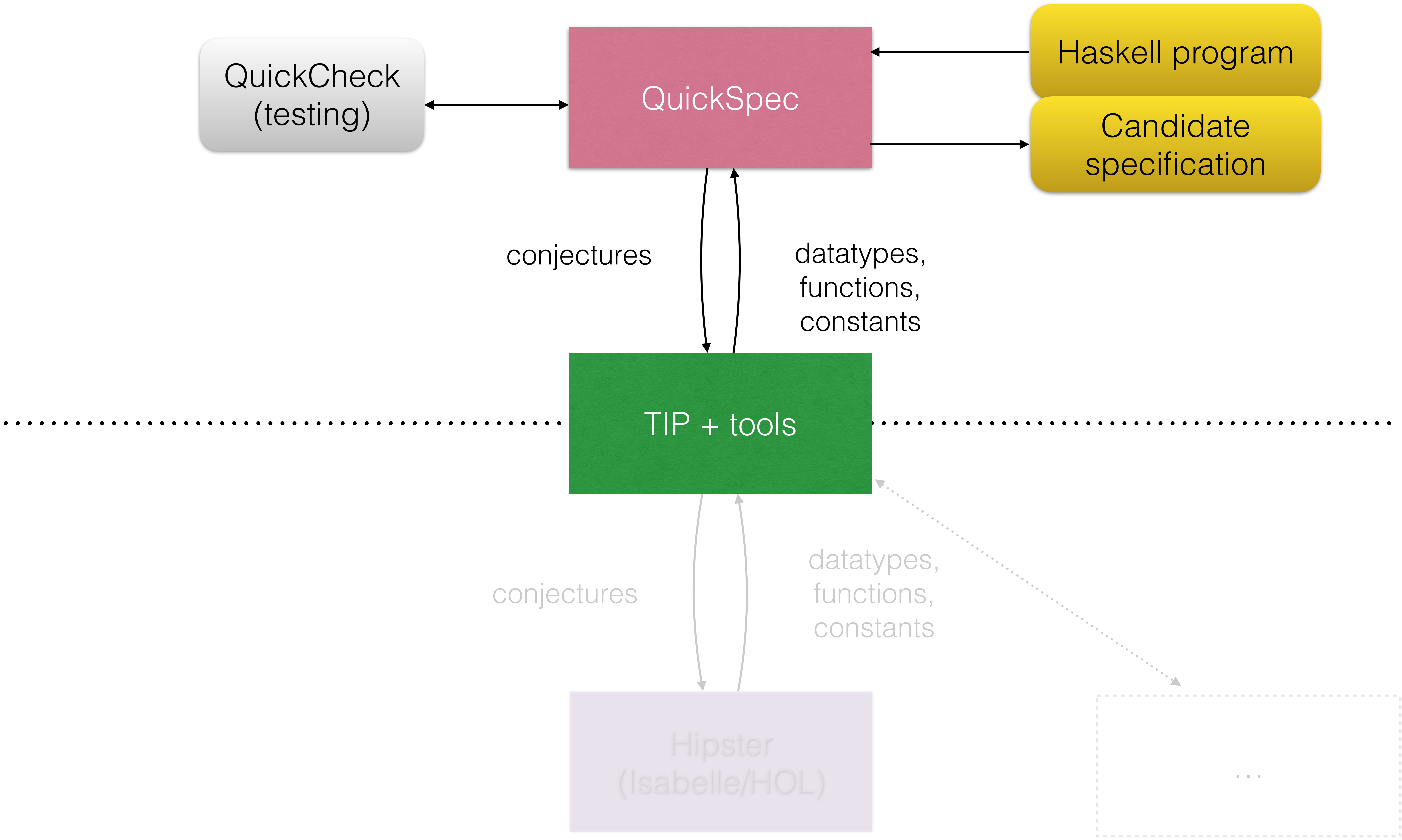


TIP

Middle layer

- Intermediate language based on SMT-LIB
  - Datatypes, recursive functions, pattern matching
- Translation tools to various formats, e.g. standard SMT-LIB.
- Libraries for writing pretty-printers and parsers




QuickCheck
(testing)
QuickSpec
Haskell program
Candidate
specification
conjectures
datatypes,
functions,
constants
TIP + tools
conjectures
datatypes,
functions,
constants
Hipster
(Isabelle/HOL)
...

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1]
ys —> []

---

xs
reverse (reverse xs)
sort xs
sort (reverse xs)
sort (xs++ys)

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1]
ys —> []

xs	[2, 1]
reverse (reverse xs)	reverse (reverse [2, 1])
sort xs	sort [2, 1]
sort (reverse xs)	sort (reverse [2, 1])
sort (xs++ys)	sort ([2, 1] ++ [])

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1]
ys —> []

xs	[2, 1]	[2, 1]
reverse (reverse xs)	reverse (reverse [2, 1])	[2, 1]
sort xs	sort [2, 1]	[1, 2]
sort (reverse xs)	sort (reverse [2, 1])	[1, 2]
sort (xs++ys)	sort ([2, 1] ++ [])	[1, 2]

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1, 3]
ys —> [3]

xs
reverse (reverse xs)
sort xs
sort (reverse xs)
sort (xs++ys)

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1, 3]
ys —> [3]

xs	[2, 1, 3]
reverse (reverse xs)	reverse (reverse [2, 1, 3])
sort xs	sort [2, 1, 3]
sort (reverse xs)	sort (reverse [2, 1, 3])
sort (xs++ys)	sort ([2, 1, 3] ++ [3])

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

xs —> [2, 1, 3]
ys —> [3]

xs	[2, 1, 3]	[2, 1, 3]
reverse (reverse xs)	reverse (reverse [2, 1, 3])	[2, 1, 3]
sort xs	sort [2, 1, 3]	[1, 2, 3]
sort (reverse xs)	sort (reverse [2, 1, 3])	[1, 2, 3]
sort (xs++ys)	sort ([2, 1, 3] ++ [3])	[1, 2, 3, 3]

How QuickSpec works

(General idea at least)

1. Term generation (small —> large)
2. Testing and evaluation (create equivalence classes)
3. Extract equations. Prune redundant using rewriting.

reverse (reverse xs) = xs
sort (reverse xs) = sort xs

~~reverse (reverse []) = []~~
~~reverse (reverse [xs++ys]) = [xs++ys]~~
~~sort (reverse (sort xs)) = sort (sort xs)~~
~~...~~

examples — -zsh — 107×45

```
jomoa@C20EPLE examples % ./Lists
```

Scalability?

A stress test

- 30+ functions, large chunk of Haskell's list library.
- Would hit exponential growth of search space…
- **But:**
    - Few equations contain near 30 symbols.
    - Interesting properties share similar structure.

```haskell
main = quickSpec [
  con "length" (length :: [A] -> Int),
  con "sort" (sort :: [Int] -> [Int]),
  con "scanr" (scanr :: (A -> B -> B) -> B -> [A] -> [B]),
  con "succ" (succ :: Int -> Int),
  con ">>=" ((>>=) :: [A] -> (A -> [B]) -> [B]),
  con "snd" (snd :: (A, B) -> B),
  con "reverse" (reverse :: [A] -> [A]),
  con "0" (0 :: Int),
  con "," ((,) :: A -> B -> (A, B)),
  con ">=>" ((>=>) :: (A -> [B]) -> (B -> [C]) -> A -> [C]),
  con ":" ((:) :: A -> [A] -> [A]),
  con "break" (break :: (A -> Bool) -> [A] -> ([A], [A])),
  con "filter" (filter :: (A -> Bool) -> [A] -> [A]),
  con "scanl" (scanl :: (B -> A -> B) -> B -> [A] -> [B]),
  con "zipWith" (zipWith :: (A -> B -> C) -> [A] -> [B] -> [C]),
  con "concat" (concat :: [[A]] -> [A]),
  con "zip" (zip :: [A] -> [B] -> [(A, B)]),
  con "usort" (usort :: [Int] -> [Int]),
  con "sum" (sum :: [Int] -> Int),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "map" (map :: (A -> B) -> [A] -> [B]),
  con "foldl" (foldl :: (B -> A -> B) -> B -> [A] -> B),
  con "takeWhile" (takeWhile :: (A -> Bool) -> [A] -> [A]),
  con "foldr" (foldr :: (A -> B -> B) -> B -> [A] -> B),
  con "drop" (drop :: Int -> [A] -> [A]),
  con "dropWhile" (dropWhile :: (A -> Bool) -> [A] -> [A]),
  con "span" (span :: (A -> Bool) -> [A] -> ([A], [A])),
  con "unzip" (unzip :: [(A, B)] -> ([A], [B])),
  con "+" ((+) :: Int -> Int -> Int),
  con "[]" ([] :: [A]),
  con "partition" (partition :: (A -> Bool) -> [A] -> ([A], [A])),
  con "fst" (fst :: (A, B) -> A),
  con "take" (take :: Int -> [A] -> [A])
  ]
```

Current work

Scaling exploration

- Limitations: theories with 30+ functions, large terms.
- Use **templates** to skip directly to interesting parts of search space.
- Search only small terms + instances of templates.
- Next: Template mining, applications.

Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing.
Sólrún Halla Einarsdóttir, Nicholas Smallbone and Moa Johansson, Proceedings of IFL, to appear 2021.

To conclude…

- Ongoing work — get in touch if you want to try it out!
- Hipster/QuickSpec are available for download.

Read more:

- ***Quick Specifications for the Busy Programmer.*** Nicholas Smallbone, Moa Johansson, Koen Claesson and Maximilian Algehed. Journal of Functional Programming, 2017.
- ***Automated Theory Exploration for Interactive Theorem Proving.*** Moa Johansson. Conference on Interactive Theorem Proving (ITP), p. 1-11, 2017.