Check Your (Students') Proofs — With Holes

Dennis Renz   Sibylle Schwarz   Johannes Waldmann
HTWK Leipzig, Germany

WFLP 2020

Programming by Proving (Exercise)

```
data N = Z | S N     -- unary (Peano) numbers
doubleN :: N -> N
doubleN Z = Z ; doubleN (S x) = S (S (doubleN x))

data B = Zero | Even B | Odd B    -- binary
value :: B -> N ; value Zero = Z
value (Even x) = doubleN (value x)
value (Odd x) = S (doubleN (value x))

-- implement succB and prove lemma:
succB :: B -> B ; succB Zero = _
succB (Even x) = _ ; succB (Odd x) = _
Lemma succ :
  forall b :: B : value (succB b) .=. S (value b)
Proof by induction on b :: B   ...   QED
```

Programming by Proving (partial Solution)

derive program (function `succB`) from specification (lemma `succ`) by writing the proof (replacing the dots "`...`") and filling holes (underscores) in the program to make the proof work.

```
                         S (value (Odd x))
(by def value)     .=. S (S (doubleN (value x)))
(by def doubleN) .=. doubleN (S (value x))
(by IH)            .=. doubleN (value (succB x))
(by def value)     .=. value (Even (succB x))
(by def succB)     .=. value (succB (Odd x))
```

E. W. Dijkstra: put the horse (proof) *before* the cart (program)!

This exercise is an example for the *Cyp* proof language (Durner and Noschinski 2013; Traytel 2019)

with our extensions: holes in programs and proofs;
also: types, integration of Cyp proof checker in auto-grader.

Cyp (Check Your Proofs)

programming language: subset of Haskell

- algebraic data types (`data`)
- function definitions with pattern matching and recursion
- no local names (no `let`, `where`, `case`, λ)
- higher-order types, but no type classes

proof language:

- by rewriting (equational reasoning)
- by extensionality (for equality of functions)
- by case analysis (on algebraic data types)
- by induction (on (recursive) algebraic data types)

original Cyp: separation of *theory* (program, axioms, goals) (given by instructor) from *proofs* (to be written by student)

What Cyp can do, and cannot do

can do:

- associativity of Peano-plus, List-append
  (induction on first argument)
- `map f . map g .=. map (f . g)`
  (extensionality, induction)

what about `merge :: Ord a => [a] -> [a] -> [a]`?

- no type classes, but can pass dictionary as extra argument
  `:: (a -> a -> Bool) -> [a] -> [a] -> [a]`
- cannot do induction on pair of arguments!

perhaps `insert::(a->a->Bool) -> a -> [a] -> [a]`?

- needs "if ($\leq$) is transitive, then ...", but have no implication!

still, equational reasoning and structural induction is plenty enough for our students (Bachelor Comp. Sci. 4th semester)

Holes

- hole $=$ missing sub-tree of program or proof
- motivation for introducing holes:
  - original Cyp: each goal (in the theory) acts as a proof-hole, there were no program-holes.
    Leads to “prove this program correct” exercises (that’s cart before horse!)
  - we can now give partial programs and partial proofs (e.g., one branch of a case analysis)
- Cyp handles submissions with holes gracefully:
  - assume hole can be filled,
  - continue checking other parts of proof
  - reject in the end.
- for step-wise development, cf. *typed holes* in Agda, GHC

Types

- original Cyp is untyped: if theory (given by instructor) is type-correct, proof (by student) cannot go wrong type-wise?
- Cyp accepted monomorphic proof for polymorphic lemma

  ```
  data U = U;      Lemma eek : x .=. y;
  Proof by case analysis on x :: U ... QED
  ... False (by eek) .=. True
  ```
- added Hindley-Milner typing for programs, lemmas, proofs,

  ```
  Lemma eek : forall x :: a, y :: a: x .=. y
  Proof by case analysis on x :: U -- rejected
  ```

  using *Typing Haskell in Haskell* (Jones, 2000)
- is needed for program-holes anyway
  (otherwise, student could write nonsense programs)

Summary/What else is in the paper

- we introduced holes in programs and in proofs, added a type checker, and integrated with Leipzig autotool
- we used Cyp/autotool for automated homework in a lecture recently (50 students, 4th semester Comp. Sci. Bachelor)
- examples: plain rewriting (no induction); Peano arithmetics; *lists*: length, append, map, fold; *trees*: mirror, inorder, size
- source code (GPL), documentation, examples: `https://gitlab.imn.htwk-leipzig.de/waldmann/cyp`

Appendix: remarks on implementation (methods, libraries used)

- ASTs: source location information in ASTs, and hiding them via GHC's pattern synonyms
- pretty-printing: avoid, print parts of original input instead
- matching for ASTs: short source code via generic traversals (Scrap Your Boilerplate, Lämmel and Peyton Jones 2003)

Discussion: Semantics of Cyp Programs

goal: provable property of Cyp program *P* should be observable when running *P* as a Haskell program

- note the similarity (it could be automated)

```
Lemma succ
  forall b::B : value(succB b) .=. S(value b)
leancheck $ \ (b :: B) ->
  value (succB b) == S (value b)
```

pattern matching: Haskell: top-down, Cyp: non-deterministically

- after `f Z = False ; f Z = True`, Cyp accepts
  `False (by def f) .=. f Z (by def f) .=. True`

possible future work:

- require naming of rule (`f.1`, `f.2`) in rewrite proof step
- enforce disjointness of patterns (reject this definition of `f`)

Discussion: overlapping clauses

This (and next slide) was asked in reviews.
Thanks for careful reading, will be helpful in paper's next version, didn't manage to update for pre-proceedings, but discuss now:

- Q: GHC's `-Woverlapping-patterns` does not detect
  `f (S x) y = _; f x (S y) = _`
  A: Indeed! To keep the paper correct, that option should be renamed (to `-Wredundant-patterns` :-) see `https://gitlab.haskell.org/ghc/ghc/-/issues/18643`
- Q: in Curry (Hanus et al., 1995), overlapping clauses define a non-deterministic function, and Cyp's statements about convertibility of expressions by rewriting are correct.

  A: Yes. So, "Cyp for Curry" next? Do it! (…and cite us.)

Discussion: termination of Cyp programs

- Q: …suggest to annotate programs with a function to project arguments to a simple well founded domain ($\mathbb{N}$, $\mathbb{N}^k$)
- A: we would then need a similar mechanism in proofs by induction? Otherwise, cannot prove properties of such functions?
  our suggestion (in the paper): require the student to mark the (structurally) decreasing argument
  reason (not stated in the paper): that argument likely is the induction variable.