Secrets of Type Driven Program Synthesis

Edwin Brady (ecb10@st-andrews.ac.uk)
University of St Andrews, Scotland
@edwinbrady

Lambda Days, 18th February 2021

Idris

Idris (`http://idris-lang.org/`) is a functional programming language with *first class types*. It supports *type-driven development* via interactive editing. Today's talk covers *type-driven program synthesis*

- *How* does it work?
- *What* can it do?

Idris

Introduction: Program Synthesis Examples

How Does It Work?

- *There is no magic!*
  - Essentially: Type-driven search, build programs incrementally, only exploring well-typed paths
  - Multiple results possible…
  - …ordered with a surprisingly simple heuristic

How Does It Work?

- *There is no magic!*
  - Essentially: Type-driven search, build programs incrementally, only exploring well-typed paths
  - Multiple results possible...
  - ...ordered with a surprisingly simple heuristic
- Some primitive operations/language features required
  - *Holes*, because partial search results are incomplete
  - *Unification*, for holes with only one possible solution
  - *Case splitting*, to refine function inputs

Outline: Expression Search

Given a hole `?f` of type `T`, try, in order:

1. Local variables
   - Refinement: use `fst` and `snd` to project elements from pairs
2. If `T` is a function type, (`a -> b`) solve with λ `x : a => ?f'`, then solve `?f'`.
3. If `T` is a data type, then for every constructor `C` of that type, try:
   - Solve with `C ?a1 ?a2 ...?an`
   - Unify solution with `T` (this might fail!)
   - Solve remaining holes
4. Solve with a recursive call, with a descending argument, to the function being defined

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = ?search

 0 m : Nat
 0 a : Type
   x : a
   xs : Vect k a
   ys : Vect m a
 0 n : Nat
------------------------------
search : Vect (S (plus k m)) a
```

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = ?a1 :: ?a2

 0 m : Nat
 0 a : Type
   x : a
   xs : Vect k a
   ys : Vect m a
 0 n : Nat
------------------------------
a1 : a
```

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: ?a2

 0 m : Nat
 0 a : Type
   x : a
   xs : Vect k a
   ys : Vect m a
 0 n : Nat
------------------------------
a2 : Vect (plus k m) a
```

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: append ?a3 ?a4

 0 m : Nat
 0 a : Type
   x : a
   xs : Vect k a
   ys : Vect m a
 0 n : Nat
------------------------------
a3 : Vect k a
```

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: append xs ?a4

 0 m : Nat
 0 a : Type
   x : a
   xs : Vect k a
   ys : Vect m a
 0 n : Nat
------------------------------
a4 : Vect m a
```

Example search problem

```
append : Vect n a -> Vect m a -> Vect (n + m) a
append [] ys = ys
append (x :: xs) ys = x :: append xs ys
```

```
No more holes!
```

Outline: Program Search

We have *Case splitting* and *Expression search*

- Program search is "just" the composition of these

Outline: Program Search

We have *Case splitting* and *Expression search*

- Program search is "just" the composition of these

For a function `f : T`

1. Generate an initial definition
   `f a1 a2 ... an = ?f_rhs`
   - Number of arguments calculated by looking at `T`
2. Apply expression search to `?f_rhs`
   - If that fails, choose an `a` to split, and repeat on the resulting pattern clauses
   - We choose the *leftmost* argument to split, and do not split to a depth greater than 1

Ordering Results

- Synthesis runs in a `Search` monad, which gives:
    - A search *result*
    - A *continuation*: what to do if either the current search action fails, or we are unsatisfied with the result
- Thus, a user can always ask for the *next result*
- In practice, we generate results in *batches*
    - Arbitrarily: 16 at a time
    - Order by *most local variables used*
        - Rationale: if a function has an argument, we probably wanted to use it
        - Suggested by Lennart Augustsson, who did this in Djinn

Refinement: Intermediate Definitions

Example: Run-length uncompression

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y)
    = ?search

 0 ty : Type
   x : ty
   y : RunLength more
   n : Nat
 0 xs : List ty
------------------------------
search : Singleton (rep n x ++ more)
```

Refinement: Intermediate Definitions

Example: Run-length uncompression

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y)
    = let Val ys = uncompress y in ?search

 0 ty : Type
   x : ty
   y : RunLength more
   n : Nat
 0 xs : List ty
------------------------------
search : Singleton (rep n x ++ more)
```

Refinement: Intermediate Definitions

Example: Run-length uncompression

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y)
    = let Val ys = uncompress y in Val ?search

 0 ty : Type
   x : ty
   y : RunLength more
   n : Nat
 0 xs : List ty
------------------------------
search : List ty
```

Refinement: Intermediate Definitions

Example: Run-length uncompression

```
uncompress : RunLength xs -> Singleton xs
uncompress Empty = Val []
uncompress (Run n x y)
    = let Val ys = uncompress y in Val (rep n x ++ ys)
```

`?search` solved by unification

Related Work

Lots of past and current work on program synthesis! Some suggestions, and some inspiration for Idris:

- *Djinn* (Haskell): `https://hackage.haskell.org/package/djinn`
- *Agsy* (Agda): `https://agda.readthedocs.io/en/v2.5.3/tools/auto.html`
- *Synquid*: "Program Synthesis from Polymorphic Refinement Types" Nadia Polikarpova et al, PLDI 2016
- "Type-and-Example-Directed Program Synthesis" Osera and Zdancewic, PLDI 2015

Summary, and Future Plans

- Given the right primitives, program search is surprisingly simple and often effective
  - Even without full dependent types!
- You can use it even more effectively if you know how it works
  - Especially, its strengths and limitations
  - "Be the machine"
- What about *domain-specific* synthesis?
  - Extend program search with special-purpose tactics, in a library
  - e.g. a session type library: "Please give me the next action in the protocol"
- Can *machine learning* help?
  - What would it learn from? Complete programs, sequences of editing actions? . . .