Programming and Proving with Guarded Recursion

Alexander Gryzlov
Research progammer
IMDEA Software Institute

PSSV-2023 workshop, 03/11/2023

Outline

- The history and concept of guarded recursion
- Stream programming
- Partiality monad
- Bird's algorithm
- Non-strictly positive datatypes

What is guarded recursion?

- A flavour of provability modality
- 1933 - Gödel's analysis of `S4`
- 1950s - Löb's axiom: $\Box(\Box A \to A) \to \Box A$
- 1960s - `GL` system
- 1970s - intuitionistic systems, fixed points
- 2000s - links to formal semantics and category theory

Nakano's approximation

- Nakano, [2000] "A modality for recursion"
- Initially denoted $\bullet$
- Continued by many authors, most notably:
- Atkey-McBride'2013
- A series of papers by Birkedal and coauthors in 2010s
- Nowadays symbolized by a right triangle $\triangleright$
- Viewed as a special type constructor in a type system

A thunk calculus

- $\triangleright$ `A` is "`A`, but available one step later"
- Essentially a thunk (delayed computation)
- Structure of an applicative
- `pure : A →` $\triangleright$ `A`
- `ap :` $\triangleright$ `(A → B) →` $\triangleright$ `A →` $\triangleright$ `B`

Ticked type theory

- We'll use Agda proof assistant for interactive examples
- Guarded modality is encoded as a function from `Tick`s of a modal type $\mathbb{T}$
- A proof of an elapsed time step
- Can encode `next` and `ap`

Ticked type theory

- Context variables right of a tick are available one step later
- Applying a tick to a term "consumes" it
- Prevents the temporal structure from collapsing
- `flatten : ▹ ▹ A → ▹ A`
- Not a monad

Clocked type theory

- We can work "inside of" thunks but not remove them
- Delayedness never decreases
- Weaker than proper conduction
- Can be extended by a constant $\Box$ modality or clock variables to allow forcing "completed" thunks

Functorial laws

- We can derive a functorial action
- `▹map : (A → B) → ▹ A → ▹ B`
- All the laws hold definitionally (by symbolic computation)

Cubical interaction

- Technically we're using the cubical mode of Agda
- No higher equalities / quotients / univalence
- Equality is encoded as a function from a continuous interval $\mathbb{I}$
- The interval is allowed to "time-travel"
- We can reason about the future in the present

Guarded recursion

- We can delay and combine computations, what now?
- Terminating → Productive
- Every recursive call is "guarded" by a thunk, giving back control
- Infinite / streaming computations, servers, OSs

Guarded recursion

- (Strong) Löb's axiom
- `fix: (▷ A → A) → A`
- A form of Y-combinator
- Postulated definition, unfolding made propositional
- Can be safely erased down to the usual fixpoint

Streams

- Classical infinite structure
- An inductive list with a delayed tail and no empty case
- Unique fixed point
- Mixing constructors and destructors

Stream reasoning

- Constant streams
- Unfolding the definition
- "Body" pattern

Stream reasoning

- Mapping streams
- Step inconsistency with induction
- Guarded induction
- Unfold → Apply → Fold pattern

Stream predicates

- `All` delayed predicate
- Mapping a pointwise function

Non-decreasing steps

- We can define `duplicate`
- But not `every-other`
- Running out of ticks
- Can be defined with clocks, however

Vs coinduction

- Coinductive mechanisms are more liberal
- Productivity checker is purely syntactic
- Spend a few hours on a proof, get shot down
- Guarded constructions are type-directed

Fixed point definition

- We can define streams as a fixed point in the universe
- Have to manually encode unrollings and constructors
- All the properties still hold

Folds and friends

- We can define other familiar functions
- `foldr`, `scan`, `zipWith`, `interleave`
- Numerical streams

Co-lists and left folds

- Co-lists can be empty
- Same operations
- Left fold is now possible but is partial
- Need a new datatype to express partiality

Partiality monad

- Many names: `L(ift)`, `Event`, `Delay`
- Essentially an arbitrary (even infinite) sequence of `▹`'s
- `now   : A → Part A`
- `later : ▹ Part A → Part A`
- Reassociating nested delays → a monad
- `▹▹▹(▹▹A) = ▹▹▹▹▹A`

Indexed partiality monad

- Can be made more graphic by indexing with steps
- ```
  mapᵈ : (A → B)
         → Delayed A n → Delayed B n
  ```
- ```
  apᵈ : Delayed (A → B) m
        → Delayed A n
        → Delayed B (max m n)
  ```
- runs "in parallel"
- ```
  _>>=ᵈ_ : Delayed A m
           → (A → Delayed B n)
           → Delayed B (m + n)
  ```
- runs sequentially
- Encoding applicative via bind changes complexity!
  ```

Left fold on Colists

- Now we can encode the left fold
-
```
foldl¹ : (B → A → B)
       → B → Colist A → Part B
```
- The result is delayed by a number of steps equal to the co-list length

Co-naturals

- "Delayed" unary numbers $\mathbb{N}\infty$
- Equal to `Part T`
- Useful for doing synthetic topology
- Sequential spaces

Bird's algorithm

- aka `replaceMin`
- Bird, [1984] "Using circular programs to eliminate multiple traversals of data"
- later generalized to `MonadFix` in Haskell
- given a binary tree with data in leaves, replaces all values with a minimum in a single pass

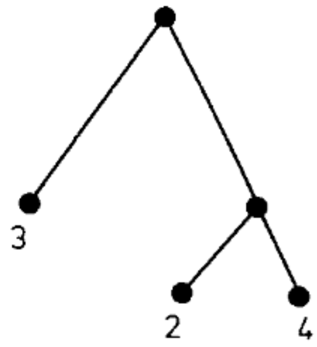


Fig. 1

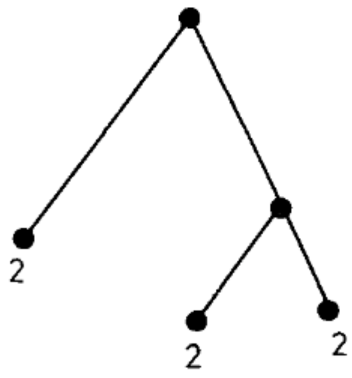


Fig. 2

Bird's algorithm

Classical form is somewhat weird

```
replaceMin :: Tree -> Tree
replaceMin t =
  let (r, m) = rmb (t, m) in r
  where
  rmb :: (Tree, Int) -> (Tree, Int)
  rmb (Leaf x, y) = (Leaf y, x)
  rmb (Node l r, y) =
    let (l',ml) = rmb (l, y)
        (r',mr) = rmb (r, y)
     in
   (Node l' r', min ml mr)
```

Guarded decomposition

- We can decompose this in two temporal phases
- Compute the minimum and construct the thunk
- Then run the thunk
- Uses the feedback combinator
- ```
  feedback : (▹ A → B × A) → B
  feedback f = fst (fix (f ∘ ▹map snd))
  ```
- Inserts intermediate data between steps
- Cannot run the thunk without clocks
  ```

Verification

- We can also verify the algorithm
- Result has the same shape
- All the elements are equal to the minimum
- Can be done with usual induction

Strict positivity

- Guarded recursion has two general areas of application:
    1. Working with potentially infinite data structures
    2. Working with non-strictly-positive recursive types

- **Strictly positive** type appears to the left of 0 arrows
- Another syntactic approximation

Strict positivity

- **Strictly positive** type appears to the left of 0 arrows
- Another syntactic approximation
- **Positive** type appears in even positions
- **Negative** type appears in odd positions

```
data Expr : Type -> Type where
  Foo : ((Expr a -> Expr a) -> Expr b) -> Expr (a -> b)
         ^^^^^^-------------------------- positive occurrence
                    ^^^^^^---------------- negative occurrence
                               ^^^^^^---- strictly positive occurrence
```

Guarded relaxation

- We can build a finer-grained approximation by guarding all positions
- Again, it must be encoded manually as a fixed point in the universe
- The positivity checker is still there
- Safe, but the price is potentially partial outputs

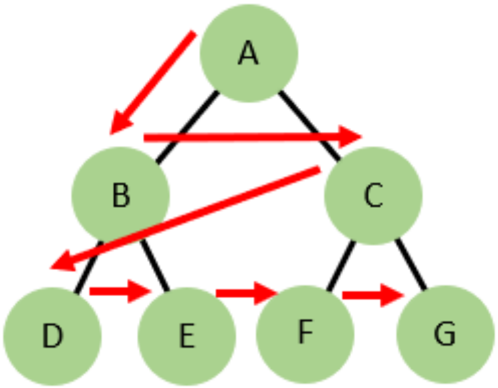
```
data Expr : Type -> Type where
  Foo : ((▹ Expr a -> ▹ Expr a) -> ▹ Expr b) -> Expr (a -> b)
```

Rec datatype

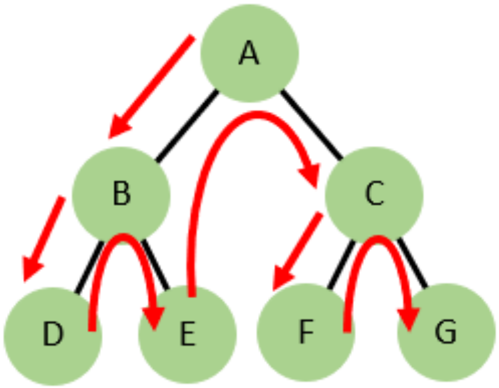
- ```
  data Rec : 𝒰 → 𝒰 where
    MkRec : (▹ Rec A → A) → Rec A
  ```
- Reified recursion
- `Rec ⊥` can be shown uninhabited
- `Rec ⊤` is isomorphic to `⊤`
  ```

Breadth-first traversal

- Another form of a binary tree, data on both leaves and nodes
- Compute a breadth-first traversal



breadth-first traversal



depth-first traversal

Hoffman's algorithm

- Typically done with queues
- Hoffman invented a purely functional continuation-based algorithm in 1993
- Requires a intermediate positive datatype
-
```
data Rou (A : 𝒰) : 𝒰 where
   overR : Rou A
   nextR : ((Rou A → List A) → List A)
         → Rou A
```

Guarded version

- As we're using a guarded approximation, we're forced to use co-lists
- Also, we need to provide all the infrastructure manually
- Un/rollings, constructors, recursor
- The algorithm recursively builds up a routine from a tree
- And uses guarded recursion to extract the value
- Initialized with an empty routine

Other uses

- The ability to work with non-strictly positive types is quite useful to do semantic work
- Logical relations are typically negative types
- `data R : Ty → Term → 𝒰 where`

```
  R𝟙 : ∅ ⊢ t ⦂ 𝟙
     → halts t
     → R 𝟙 t
  R⇒ : ∅ ⊢ t ⦂ (T₁ ⇒ T₂)
     → halts t
     → (∀ s → ▹ R T₁ s → Part (▹ R T₂ (t · s)))
     → R (T₁ ⇒ T₂) t
```

- The right hand side is delayed by N+1 steps
- Can construct denotational semantics for general recursion (PCF)

Conclusion

- A principled way to work with non-termination
- A common theme is overcoming syntactic approximations
- Thunks, streams, partiality
- Non-strictly positive datatypes
- Synthetic topology and domain theory
- Concurrency models (quotienting & cubical gizmos)

Working repos

- https://github.com/clayrat/guarded-cm/
- https://github.com/clayrat/logrel-guarded/

Literature

- Nakano, [2000] "A modality for recursion"
- Artemov, Beklemishev, [2004] "Provability logic"
- https://agda.readthedocs.io/en/latest/language/guarded.html
- Atkey, McBride, [2013] "Productive Coprogramming with Guarded Recursion"
- Bird, [1984] "Using circular programs to eliminate multiple traversals of data"
- Berger, Matthes, Setzer, [2019] "Martin Hofmann's Case for Non-Strictly Positive Data Types"
- Clouston, Bizjak, Grathwohl, Birkedal, [2016] "The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types"
- Paviotti, Mogelberg, Birkedal, [2015] "A model of PCF in Guarded Type Theory"

Contacts

- http://clayrat.github.io/
- https://software.imdea.org/~aliaksandr.hryzlou/
- https://www.linkedin.com/in/alexgryzlov/
- https://twitter.com/clayrat/