How to calculate with nondeterministic functions

Richard Bird and Florian Rabe

Computer Science, Oxford University resp. University Erlangen-Nürnberg

July 2019

Background

Calculate Functional Programs

- Bird–Meertens formalism (Squiggol)
  - derive functional programs from specifications
  - use equational reasoning to calculate correct programs
  - optimize along the way

  Example:

  $$h\,(f\;e\,xs) = F\,(h\,e)\,xs$$

  try to solve for F to get more efficient algorithm
- Richard's textbooks on functional programming
  - Introduction to Functional Programming, 1988
  - Introduction to Functional Programming using Haskell, 1998
  - Thinking Functionally with Haskell, 2014

History

My background

- not algorithms or functional programming
- formal systems (logics, type theories, foundations, DSLs, etc.)
- design, analysis, implementation
- applications to all STEM disciplines

This work

- Richard encountered problem with an elementary example
- He built bottom-up solution using non-deterministic functions
- I got involved in working out the formal details of the calculus

i.e., my contribution is arguably the less interesting part of this work :)

Overview

Summary

Our Approach

- Specifications tend to have non-deterministic flavor
  even when specifying deterministic functions
- Program calculation with deterministic λ-calculus can be limiting
- Our idea:
  - extend to λ-calculus with non-deterministic functions
  - in a way that preserves existing notations and theorems
    works well
  - mostly following the papers by Morris and Bunkenburg

Warning

- We calculate and execute only deterministic functions.
- We use non-deterministic functions only for specifications and intermediate values. calculus allows more but not explored here

Non-Determinism

Kinds of function

- Function $A \to B$ is relation on A and B that is
  - total (at least one output per input)
  - deterministic (at most one output per input)
- Partial functions = drop totality
  - very common in math and elementary CS
  - can be modeled as option-valued total functions

$$A \to \texttt{Option}\, B$$

- Non-deterministic functions = drop determinism
  - somewhat dual to partial functions, but much less commonly used
  - can be modeled as nonempty-set-valued deterministic functions

$$A \to \mathbb{P}^{\neq\varnothing}\, B$$

Motivation

A Common Optimization Problem

Two-step optimization process

1. generate list of candidate solutions (from some input)

$$\texttt{genCand} : \texttt{Input} \to \texttt{List}\,\texttt{Cand}$$

2. choose cheapest candidate from that list

$$\texttt{minCost} : \texttt{List}\,\texttt{Cand} \to \texttt{Cand}$$

$$\texttt{optimum}\,\mathit{input} = \texttt{minCost}\,(\texttt{genCand}\,\mathit{input})$$

`minCost` is where non-determinism will come in

- $\texttt{minCost}\,\mathit{cs}$ = some c with minimal cost among cs non-deterministic
- for now: $\texttt{minCost}\,\mathit{cs}$ = first such c deterministic

A More Specific Setting

$$\texttt{genCand} : \texttt{Input} \to \texttt{List}\,\texttt{Cand}$$

$$\texttt{minCost} : \texttt{List}\,\texttt{Cand} \to \texttt{Cand}$$

- *input* is some recursive data structure
- candidates for bigger input are built from candidates for smaller input
- our case: *input* is a list, and genCand is a fold over input

$$\texttt{extCand}\,x : \texttt{Cand} \to \texttt{List}\,\texttt{Cand}$$

extends candidate for *xs* to candidate list for $x :: xs$

$$\texttt{genCand}\,(x :: xs) = \texttt{extCand}\,x\,(\texttt{genCand}\,xs)$$

Idea to Derive Efficient Algorithm

- Fuse `minCost` and `genCand` into a single fold
- Greedy algorithm
  - don't build all candidates, apply `minCost` once at the end
  - apply `minCost` early on, extend only optimal candidates
- Not necessarily sound:

  non-optimal candidates for small input

  might extend to

  optimal candidates for large input

$$\mathtt{optimum}\ \mathit{input} = \mathtt{minCost}\ (\mathtt{genCand}\ \mathit{input})$$

$$\mathtt{genCand}\ (x :: xs) = \mathtt{extCand}\ x\ (\mathtt{genCand}\ xs)$$

$$\mathtt{genCand} : \mathtt{Input} \to \mathtt{List}\ \mathtt{Cand}$$

$$\mathtt{minCost} : \mathtt{List}\ \mathtt{Cand} \to \mathtt{Cand}$$

$$\mathtt{extCand}\ x : \mathit{Cand} \to \mathtt{List}\ \mathtt{Cand}$$

Solution through Program Calculation

Obtain a greedy algorithm from the specification

1. Assume

$$\texttt{optimum}\, \mathit{input} = F\, c_0\, \mathit{input}$$

(c_0 is base solution for empty input)

and try to solve for folding function F

Solution through Program Calculation

Obtain a greedy algorithm from the specification

1. Assume

$$\texttt{optimum}\,\mathit{input} = F\ c_0\ \mathit{input}$$

(c_0 is base solution for empty input)

and try to solve for folding function F

2. Routine equational reasoning yields
   - solution:

$$F\,x\,c = \texttt{minCost}\,(\texttt{extCand}\,x\,c)$$

   - soundness condition:

$$\texttt{optimum}\,(x :: xs) = F\,x\,(\texttt{optimum}\,xs)$$

   Intuition: solution $F\,x\,c$ for input $x :: xs$ is
   cheapest extension of solution c for input xs

A Subtle Problem

Soundness condition (from previous slide):

$$F\,x\,c = \mathtt{minCost}\,(\mathtt{extCand}\,x\,c)$$

$$\mathtt{optimum}\,(x :: xs) = F\,x\,(\mathtt{optimum}\,xs)$$

optimal candidate for $x :: xs$ must be
optimal extension of optimal candidate for xs

Soundness condition is intuitive and common
but subtly stronger than needed:

- `optimum` and F defined in terms of `minCost`
- Actually states:
  first optimal candidate for $x :: xs$ is
  first optimal extension of first optimal candidate for xs

rarely holds in practice

What went wrong?

What happens:

- Specification of `minCost` naturally non-deterministic
- Using standard λ-calculus forces
  artificial once-and-for-all choice to make `minCost` deterministic
- Program calculation uses only equality

artificial choices must be preserved

What should happen:

- Use λ-calculus with non-deterministic functions
- `minCost` returns some candidate with minimal cost
- Program calculation uses equality and refinement

gradual transition towards deterministic solution

Formal System: Syntax

Key Intuitions (Don't skip this slide)

Changes to standard λ-calculus

- $A \to B$ is type of **non-deterministic** functions
- Every term represents a **nonempty set** of possible values

Key Intuitions (Don't skip this slide)

Changes to standard λ-calculus

- $A \to B$ is type of **non-deterministic** functions
- Every term represents a **nonempty set** of possible values
- **Pure** terms <u>roughly</u> represent a single value

Key Intuitions (Don't skip this slide)

Changes to standard λ-calculus

- $A \to B$ is type of **non-deterministic** functions
- Every term represents a **nonempty set** of possible values
- **Pure** terms roughly represent a single value
- **Refinement** relation between terms of the same type:
  $s \overset{\text{ref}}{\longleftarrow} t$ iff s-values are also t-values

Key Intuitions (Don't skip this slide)

Changes to standard λ-calculus

- $A \to B$ is type of **non-deterministic** functions
- Every term represents a **nonempty set** of possible values
- **Pure** terms roughly represent a single value
- **Refinement** relation between terms of the same type:
  $s \overset{\text{ref}}{\leftarrow} t$ iff s-values are also t-values
- Refinement is an order at every type, in particular

$$s \overset{\text{ref}}{\leftarrow} t \quad \wedge \quad t \overset{\text{ref}}{\leftarrow} s \quad \Rightarrow \quad s \doteq t$$

$\doteq$ is the usual equality between terms

Key Intuitions (Don't skip this slide)

Changes to standard λ-calculus

- $A \to B$ is type of **non-deterministic** functions
- Every term represents a **nonempty set** of possible values
- **Pure** terms roughly represent a single value
- **Refinement** relation between terms of the same type:
  $s \overset{\text{ref}}{\leftarrow} t \quad$ iff $\quad s$-values are also t-values
- Refinement is an order at every type, in particular

$$s \overset{\text{ref}}{\leftarrow} t \quad \wedge \quad t \overset{\text{ref}}{\leftarrow} s \quad \Rightarrow \quad s \doteq t$$

$\doteq$ is the usual equality between terms

- Refinement for functions
  - point-wise: $f \overset{\text{ref}}{\leftarrow} g \quad$ iff $\quad f(x) \overset{\text{ref}}{\leftarrow} g(x)$ for all pure x
  - deterministic functions are minimal wrt refinement

Syntax: Type Theory

$$
\begin{array}{llll}
A, B & ::= & a & \text{base types (integers, lists, etc.)} \\
 & \mid & A \to B & \text{non-det. functions} \\
s, t & ::= & c & \text{base constants (addition, folding, etc.)} \\
 & \mid & x & \text{variables} \\
 & \mid & \lambda x : A.t & \text{function formation} \\
 & \mid & s\ t & \text{function application} \\
 & \mid & s \sqcap t & \text{non-deterministic choice}
\end{array}
$$

Typing rules as usual plus

$$\frac{\vdash s : A \qquad \vdash t : A}{\vdash s \sqcap t : A}$$

Syntax: Logic

Additional base types/constants:

- $\mathtt{bool} : \mathtt{type}$
- logical connectives and quantifiers as usual, e.g.,

$$\frac{\vdash s : A \qquad \vdash t : A}{\vdash s \doteq t : \mathtt{bool}}$$

- refinement predicate

$$\frac{\vdash s : A \qquad \vdash t : A}{\vdash s \stackrel{\text{ref}}{\leftarrow} t : \mathtt{bool}}$$

- purity predicate

$$\frac{\vdash t : A}{\vdash \mathit{pure}(t) : \mathtt{bool}}$$

Formal System: Semantics

Semantics: Overview

Syntax	Semantics
type A	set $[\![A]\!]$
context declaring $x : A$	environment mapping $\rho : x \mapsto [\![A]\!]$
term $t : A$	nonempty subset $[\![t]\!]_\rho \in \mathbb{P}^{\neq\varnothing}[\![A]\!]$
refinement $s \overset{\mathrm{ref}}{\leftarrow} t$	subset $[\![s]\!]_\rho \subseteq [\![t]\!]_\rho$
purity $pure(t)$ for $t : A$	$[\![t]\!]_\rho$ is generated by a single $v \in [\![A]\!]$
choice $s \sqcap t$	union $[\![s]\!]_\rho \cup [\![t]\!]_\rho$

Semantics: Functions

Functions are interpreted as set-valued semantic functions:

$$[\![A \to B]\!] = [\![A]\!] \Rightarrow \mathbb{P}^{\neq\varnothing}[\![B]\!]$$

using $\Rightarrow$ for the usual set-theoretical function space

Function application is monotonous wrt refinement:

$$[\![f\ t]\!]_\rho = \bigcup_{\varphi \in [\![f]\!]_\rho, \tau \in [\![t]\!]_\rho} \varphi(\tau)$$

Semantics: Functions

Functions are interpreted as set-valued semantic functions:

$$[\![A \to B]\!] = [\![A]\!] \Rightarrow \mathbb{P}^{\neq\varnothing}[\![B]\!]$$

using $\Rightarrow$ for the usual set-theoretical function space

Function application is monotonous wrt refinement:

$$[\![f\ t]\!]_\rho = \bigcup_{\varphi \in [\![f]\!]_\rho, \tau \in [\![t]\!]_\rho} \varphi(\tau)$$

The interpretation of a λ-abstractions is closed under refinements:

$$[\![\lambda x : A.t]\!]_\rho = \left\{\varphi \,\middle|\, \text{ for all } \xi \in [\![A]\!] : \ \varphi(\xi) \subseteq [\![t]\!]_{\rho, x \mapsto \xi}\right\}$$

contains all deterministic functions that return refinements of t

Semantics: Purity and Base Cases

For every type A, also define embedding $[\![A]\!] \ni \xi \mapsto \xi^{\leftarrow} \subseteq [\![A]\!]$

- for base types: $\xi^{\leftarrow} = \{\xi\}$
- for function types: closure under refinement

Pure terms are interpreted as embeddings of singletons:

$$[\![\mathit{pure}(t)]\!]_\rho = 1 \quad \text{iff} \quad [\![t]\!]_\rho = \tau^{\leftarrow} \text{ for some } \tau$$

- Variables

$$[\![x]\!]_\rho = \rho(x)^{\leftarrow}$$

note: $\rho(x) \in [\![A]\!]$, not $\rho(x) \subseteq [\![A]\!]$

- Base types: as usual
- Base constants c with usual semantics C:

$$[\![c]\!]_\rho = C^{\leftarrow}$$

straightforward if c is first-order

Formal System: Proof Theory

Overview

Akin to standard calculi for higher-order logic

- Judgment $\Gamma \vdash F$ for a context Γ and F : `bool`
- Usual axioms/rules for equality and propositional connectives

  modifications needed when variable binding is involved
- Intuitive axioms/rules for choice and refinement

  technical difficulty to get purity right

Multiple equivalent axiom systems

- In the sequel, no distinction between primitive and derivable rules
- Very subtle in practice to prove derivability of rules

  formalization in logical framework helps

Refinement and Choice

- General properties of refinement
  - $s \overset{\text{ref}}{\leftarrow} t$ is an order (wrt $\doteq$)
  - characteristic property:

$$s \overset{\text{ref}}{\leftarrow} t \quad \text{iff} \quad u \overset{\text{ref}}{\leftarrow} s \text{ implies } t \overset{\text{ref}}{\leftarrow} u \text{ for all } u$$

Refinement and Choice

- General properties of refinement
  - $s \overset{\text{ref}}{\leftarrow} t$ is an order (wrt $\dot{=}$)
  - characteristic property:

$$s \overset{\text{ref}}{\leftarrow} t \quad \text{iff} \quad u \overset{\text{ref}}{\leftarrow} s \text{ implies } t \overset{\text{ref}}{\leftarrow} u \text{ for all } u$$

- General properties of choice
  - $s \sqcap t$ is associative, commutative, idempotent (wrt $\dot{=}$)
  - no neutral element

    we do not have an undefined term with $[\![\perp]\!]_\rho = \varnothing$

Refinement and Choice

- General properties of refinement
  - $s \overset{\text{ref}}{\leftarrow} t$ is an order (wrt $\doteq$)
  - characteristic property:

$$s \overset{\text{ref}}{\leftarrow} t \quad \text{iff} \quad u \overset{\text{ref}}{\leftarrow} s \text{ implies } t \overset{\text{ref}}{\leftarrow} u \text{ for all } u$$

- General properties of choice
  - $s \sqcap t$ is associative, commutative, idempotent (wrt $\doteq$)
  - no neutral element

    we do not have an undefined term with $[\![\perp]\!]_\rho = \varnothing$
- Refinement of choice
  - $u \overset{\text{ref}}{\leftarrow} s \sqcap t$ refines to a pure term u iff s or t does
  - in particular, $t_i \overset{\text{ref}}{\leftarrow} (t_1 \sqcap t_2)$

Rules for Purity

- Purity predicate only present for technical reasons
- Pure are
  - base constants applied to any number of pure arguments
  - λ-abstractions

and thus all terms without $\sqcap$

- Syntactic vs. semantic approach
  - Semantic = use rule

$$\frac{\vdash \mathit{pure}(s) \qquad \vdash s \doteq t}{\vdash \mathit{pure}(t)}$$

thus $1 \sqcap 1$ is pure

  - literature uses syntactic rules like "variables are pure"

easier at first, trickier in the details

Rules for Function Application

- Distribution over choice:

$$\vdash f\,(s \sqcap t) \doteq (f\ s) \sqcap (f\ t)$$

$$\vdash (f \sqcap g)\ t \doteq (f\ t) \sqcap (g\ t)$$

- Monotonicity wrt refinement:

$$\frac{\vdash f' \overset{\text{ref}}{\leftarrow} f \qquad t' \overset{\text{ref}}{\leftarrow} t}{\vdash f'\ t' \overset{\text{ref}}{\leftarrow} f\ t}$$

- Characteristic property wrt refinement:

$$u \overset{\text{ref}}{\leftarrow} f\ t \qquad \text{iff} \qquad f' \overset{\text{ref}}{\leftarrow} f,\ t' \overset{\text{ref}}{\leftarrow} t,\ u \overset{\text{ref}}{\leftarrow} f'\ t'$$

Beta-Conversion

Intuition: bound variable is pure, so only substitute with pure terms

$$\frac{\vdash t : A \qquad pure(t)}{\vdash (\lambda x : A.t)\, s \doteq t[x/s]}$$

Counter-example if we omitted the purity condition

- Wrong:

$$(\lambda x : \mathbb{Z}.x + x)(1 \sqcap 2) \doteq (1 \sqcap 2) + (1 \sqcap 2) \doteq 2 \sqcap 3 \sqcap 4$$

- Correct:

$$(\lambda x : \mathbb{Z}.x{+}x)\,(1{\sqcap}2) \doteq ((\lambda x : \mathbb{Z}.x{+}x)\,1){\sqcap}((\lambda x : \mathbb{Z}.x{+}x)\,2) \doteq 2{\sqcap}4$$

Computational intuition: no lazy resolution of non-determinism

Xi-Conversion

- Equality conversion under a λ, congruence rule for binders
- Usual formulation

$$\frac{x : A \vdash f \doteq g}{\vdash \lambda x : A.f \doteq \lambda x : A.g}$$

- Adjusted: bound variable is pure, so add purity assumption when traversing into a binder

$$\frac{x : A,\ \mathit{pure}(x) \vdash f \doteq g}{\vdash \lambda x : A.f \doteq \lambda x : A.g}$$

needed to discharge purity conditions of the other rules

Computational intuition: functions can assume arguments to be pure

Eta-Conversion

Because λ-abstractions are pure, η can only hold for pure functions

$$\frac{\vdash f : A \to B \qquad \vdash pure(f)}{\vdash f \doteq \lambda x : A.(f\,x)}$$

Counter-example if we omitted the purity condition:

- Wrong:

$$f \sqcap g \doteq \lambda x : \mathbb{Z}.(f \sqcap g)\,x \doteq \lambda x : \mathbb{Z}.(f\,x) \sqcap (g\,x)$$

- Correct:

$$f \sqcap g \overset{\mathrm{ref}}{\leftarrow} \lambda x : \mathbb{Z}.(f\,x) \sqcap (g\,x)$$

  but not the other way around

Computational intuition: choices under a λ are resolved fresh each call

Formal System: Meta-Theorems

Overview

Soundness

- If $\vdash F$, then $[\![F]\!]_\rho = 1$
- In particular: if $\vdash s \stackrel{\text{ref}}{\leftarrow} t$, then $[\![s]\!]_\rho \subseteq [\![t]\!]_\rho$.

Consistency

- $\vdash F$ does not hold for all F

Completeness

- Not investigated at this point
- Presumably similar to usual higher-order logic

Conclusion

Revisiting the Motivating Example

- Applied to many examples in forthcoming textbook

  Algorithm Design using Haskell, Bird and Gibbons
- Two parts on greedy and thinning algorithms
- Based on two non-deterministic functions

$$\texttt{MinWith} : \texttt{List}\, A \to (A \to B) \to (B \to B \to \texttt{bool}) \to A$$

$$\texttt{ThinBy} : \texttt{List}\, A \to (A \to A \to \texttt{bool}) \to \texttt{List}\, A$$

- `minCost` from motivating example defined using `MinWith`
- Soundness conditions for greedy algorithms can be proved for many practical examples

Summary

- Program calculation can get awkward if non-deterministic specifications are around

  e.g., minimal wrt to cost, or thinning wrt order
- Elegant solution by allowing for non-deterministic functions
- Minimally invasive
  - little new syntax
  - old syntax/semantics embeddable
  - only minor changes to rules
  - some subtleties but manageable

  formalization in logical framework helps
- Many program calculation principles carry over

  deserves systematic attention