Proving the Correctness of Rewrite Rules in LIFT’s Rewrite-Based System

Xueying Qin (2335466Q)

Introduction

- Motivation
  - Ensuring the correctness of LIFT’s rewrite rules used for optimisation is important
  - Some rules and existing paper proofs are not well-structured or incorrect
- Aims
  - Designing concise and well-structured semantics for LIFT’s patterns in Agda
  - Verifying the correctness of LIFT’s rewrite rules
  - Revising incorrect and inaccurate rewrite rules

Background

- LIFT
  - High-level programming language which provides high performance and code portability
  - Primitive pattens: map, reduce, split, join, etc.
  - Rewrite rules encode optimisation strategies
- Curry-Howard Correspondence
  - Propositions as types
  - Proofs as programs
  - Simplification of proofs as evaluation of programs
- Agda
  - A dependently-typed programming language
  - Used as a proof assistant in this project

Semantics of LIFT in Agda - Data Types

- data -- The set of data types
  - Set in Agda
- nat -- Natural numbers
  - ℕ in Agda
- array -- An indexed collection
  - Vec in Agda

Semantics of LIFT in Agda -- Primitives

- map:

```
map : {n : ℕ} → {s : Set} → {t : Set} → (s → t) → Vec s n → Vec t n
map f [] = []
map f (x ∷ xs) = f x ∷ map f xs
```

- split:

```
split : (n : ℕ) → {m : ℕ} → {t : Set} → Vec t (m * n) → Vec (Vec t n) m
split n {zero} xs = []
split n {suc m} xs = take n {m * n} xs ∷ split n (drop n xs)
```

- join:

```
join : {n m : ℕ} → {t : Set} → Vec (Vec t n) m → Vec t (m * n)
join [] = []
join (xs ∷ xs₁) = xs ++ join xs₁
```

Equality Reasoning for Rewrite Rules - Split-Join

- A formal definition: $map\ f \rightarrow join \circ map\ (map\ f) \circ split\ n$
- Proof in Agda:

```
splitJoin : {m : ℕ} → {s : Set} → {t : Set} → (n : ℕ) → (f : s → t) → (xs : Vec s (m * n)) →
            (join ∘ map (map f) ∘ split n {m}) xs ≡ map f xs
splitJoin {m} n f xs =
  begin
    join (map (map f) (split n {m} xs))
  ≡⟨ cong join (splitBeforeMapMapF n {m} f xs) ⟩
    join (split n {m} (map f xs))
  ≡⟨ simplification n {m} (map f xs) ⟩
    map f xs
  ∎
```

Lemmas:

```
splitBeforeMapMapF : (n : ℕ) → {m : ℕ} → {s t : Set} →
                     (f : s → t) → (xs : Vec s (m * n)) →
                     map (map f) (split n {m} xs) ≡ split n {m} (map f xs)

simplification : (n : ℕ) → {m : ℕ} → {t : Set} → (xs : Vec t (m * n)) →
                 (join ∘ split n {m}) xs ≡ xs
```

Equality Reasoning for Rewrite Rules - Tiling

- A formal definition:

$$map\ f \circ slide\ size\ step \rightarrow join \circ map\ (\lambda\ tile.\ map\ f \circ (slide\ size\ step\ tile))\ slide\ u\ v$$

Choices of *u* and *v* are not specified in paper, we only know: *u - v = size - step*

- Slide is primitive defined as:

slide : {n : ℕ} → (sz : ℕ) → (sp : ℕ)→ {t : Set} → Vec t (sz + n * (suc sp)) → Vec (Vec t sz) (suc n)

- Proof in Agda - giving general restrictions to *u* and *v*:
  - *u = sz + n * suc sp, v = n + sp + n * sp*
  - Using *(suc sp)* and *(suc v)* to ensure they are larger than zero

Equality Reasoning for Rewrite Rules - Tiling (cont.)

- Proof in Agda - equality declaration:

```
tiling : {n m : ℕ} → {s t : Set} → (sz sp : ℕ) → (f : Vec s sz → Vec t sz) → (xs : Vec s (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
          join (map (λ (tile : Vec s (sz + n * (suc sp))) →
          map f (slide {n} sz sp tile)) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs)) ≡
          map f (slide {n + m * (suc n)} sz sp (cast (lem₁ n m sz sp) xs))
```

  - Changing the order of join in the expression
  - Proving the partitioning of slide
- Challenge:
  - The pattern matching on array’s size introduces complexity into the proof.

Research Outcomes

- Dependently typed pattern matching in machine verification is helpful for formalising and verifying these rewrite rules
- Induction is the core of the construction of semantics and proofs
- Breaking complex rewrite rules into reusable lemmas simplifies the process of developing proofs
- Using Agda REWRITE feature to improve flexibility of pattern matching
- Using heterogeneous equality to reason about equality between different types

Conclusion and Future Work

- Effective mechanical verification in Agda is developed for justifying the correctness of the rewrite rules in LIFT
- Most of the rules are proven to be correct
- Incorrect and inaccurate paper proofs and rules are revised
- We would like to generalise the verification on rules defined for n-dimensional arrays in the future