Proving the Correctness of Rewrite Rules in LIFT Rewite-Based System

Xueying Qin - 2335466Q

December 18, 2019

1 Status report

1.1 Proposal

1.1.1 Motivation

The rewrite system of LIFT systematically transforms high-level algorithmic patterns into low-level high performance OpenCL code with equivalent functionality. During this process, a set of rewrite rules are applied. Ensuring the correctness of these rules is important for ensuring the functionality is not altered during the rewrite process. Currently, the correctness of these rules is only proven correct on paper and only for a subset of the used rules. Thus, in this project I will develop mechanical proofs in Agda to show the correctness of the rewrite rules.

1.1.2 Aims

First of all, I would like to design effective semantics for primitive function to simplify the process of developing proofs. Secondly, developing effective lemmas to keep the modularity of proofs is also essential for this project. Thirdly, it is important to verify if the rules are correct and potentially propose strategies to fix issues if they occurs.

1.2 Progress

A set of primitives are defined and basic rewrite rules are proven. Details are recorded in the project repository: `https://github.com/XYUnknown/individual-project/blob/master/list.lagda.md`

- Primitives

```
id : {T : Set} → T → T
map : {n : ℕ} → {s : Set} → {t : Set} → (s → t) → Vec s n → Vec t n
take : (n : ℕ) → {m : ℕ} → {t : Set} → Vec t (n + m) → Vec t n
drop : (n : ℕ) → {m : ℕ} → {t : Set} → Vec t (n + m) → Vec t m
split : (n : ℕ) → {m : ℕ} → {t : Set} → Vec t (n * m) →
        Vec (Vec t n) m
join : {n m : ℕ} → {t : Set} → Vec (Vec t n) m → Vec t (n * m)
slide : {n : ℕ} → (sz : ℕ) → (sp : ℕ)→ {t : Set} →
        Vec t (sz + n * (suc sp)) → Vec (Vec t sz) (suc n)
reduceSeq : {n : ℕ} → {s t : Set} → (s → t → t) → t → Vec s n → t
```

```
reduce : {n : ℕ} → {t : Set} → (M : CommAssocMonoid t) → Vec t n → t
partRed : (n : ℕ) → {m : ℕ} → {t : Set} → (M : CommAssocMonoid t) →
          Vec t (suc m * n) → Vec t (suc m)
```

- Rewite rules
  - Identity rules
  - Fusion rules
  - Simplification rules
  - Split-join rule
  - Reduction rule
  - Partial reduction rules

1.3 Problems and risks

1.3.1 Problems

The dependent types and pattern matching in Agda overcomplicated defining primitives and developing proofs. For example, when reasoning about the size of a vector, it has been tedious to prove something like the size $m * suc\ n$ and $suc\ n * m$ are the same. We made use of the built-in `REWRITE` feature to override some pattern matching, which helped simplify the semantics and proofs.

The definitions of `reduce` and `partRed` require an arbitrary associtive and commutative operator with an identity element. It was difficult to solely define such an operator with the identity element in Agda. We resolved this issue by declaring a typeclass in Agda to enmulate this operator with certain properties. This ensured the modularity of proofs.

1.3.2 Risks

The built-in `REWRITE` is an unsafe feature in Agda, misusing it can potentially break the soundness of the type system in Agda. The mitigation of this risk is we must justify, i.e., provide proofs for the pattern matching to be rewritten into the system whenever we need to use this feature, to ensure the soundness of Agda type system.

1.4 Plan

In the first semester, I have finished the basic features proposed in the project proposal. In the first half of next semester, firstly, I will develop the semantics for more complicated primitive such as `iterate`, `reorder` and `transpose`. Sencondly, relevant rewrite rules will be proven. Moreover, we might look into the correctness of ELEVATE strategies. In the second half of semester, I will write the finial dissertation.

1.5 Ethics and data

This project does not involve human subjects or data. No approval required.