Proving the Correctness of Rewrite Rules in Rise Rewrite-Based System

Xueying Qin

Motivation

- The rewrite system of RISE (the successor of LIFT) transforms programs composed of high-level patterns into low-level code with equivalent functionality using rewrite rules
- Ensuring the correctness of these rules is important to ensure a program's functionality is not altered after optimisation
- **Therefore, we would like to develop mechanical proofs in Agda to show the correctness of these rules**

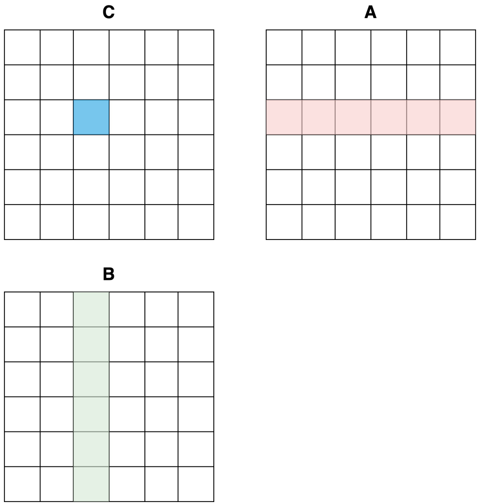
Background

- RISE
    - High-level programming language which provides high performance and code portability
    - Primitive patterns: 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

RISE Example - Matrix Multiplication

- Matrix multiplication expresses in RISE

```
matrixMultiplication A B = map fun (aRow =>
                             map fun (bCol =>
                               reduce add 0 (map mul (zip aRow bCol))
                             ) (transpose B)
                           ) A
```



- Rewrite rules can be applied for optimisation
  - $map\ f \longrightarrow join \circ map\ (map\ f) \circ split\ n$
  - $map\ (f \circ g) \longrightarrow map\ f \circ map\ g$
  - $reduce\ f\ id \circ map\ g \longrightarrow reduce\ (\lambda\ a\ b.\ f\ a\ (g\ b))\ id$
  - …

Semantics of RISE in Agda

- data -- The set of data types
  - `Set` in Agda
- nat -- Natural numbers
  - `ℕ` in Agda
- array -- An indexed collection
  - `Vec` in Agda
- function
  - The function type in Agda, written as `(x : A) → B` or `A → B`

Semantics of RISE in Agda - Natural Numbers

- Natural numbers:

```
-- The definition of natural numbers in Agda
data ℕ : Set where
zero : ℕ
suc : (n : ℕ) → ℕ

-- The definition of natural number addition in Agda
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)

-- The definition of natural number multiplication in Agda
_*_ : ℕ → ℕ → ℕ
zero * m = zero
suc n * m = m + n * m
```

Semantics of RISE in Agda - Indexed Collection

- An indexed collection:

```
-- Define an indexed collection
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : N} → A → Vec A n → Vec A (suc n)

-- The definition of vector concatenation
_++_ : {m n : ℕ} → Vec A m → Vec A n → Vec A (m + n)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Equality Reasoning for Rewrite Rules - Map-Fusion (1)

- A formal definition: $map\ f \circ map\ g \rightarrow map\ (f \circ g)$
- We first need to define the primitive map:
- $map : \{n : nat\} \rightarrow \{s\ t : data\} \rightarrow n.s \rightarrow n.t$

```
-- The definition of primitive map
map : {n : ℕ} → {S T : Set} → (S → T) → Vec S n → Vec T n
map f [] = []
map f (x :: xs) = f x :: map f xs
```

Equality Reasoning for Rewrite Rules - Map-Fusion (2)

- The map-fusion rule: $map\ f \circ map\ g \rightarrow map\ (f \circ g)$

```
fusion : {n : ℕ} → {S T R : Set} → (f : T → R) → (g : S → T) → (xs : Vec S n) →
         (map f ∘ map g) xs ≡ map (f ∘ g) xs

-- The proof of the map-fusion rule by induction
fusion f g [] = refl
fusion f g (x :: xs) = cong ((f ∘ g) x ::_) (fusion f ∘ g xs)
```

- `refl` is the reflexivity of equality
- Function `cong` is congruence, which is defined in Agda standard library as:

```
cong : {A B : Set} → ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
```

Equality Reasoning for Rewrite Rules - Split-Join (1)

- A formal definition: $map\ f \rightarrow join \circ map\ (map\ f) \circ split\ n$
- $split : (n : nat) \rightarrow \{m : nat\} \rightarrow \{t : data\} \rightarrow nm.t \rightarrow m.n.t$

```
-- The definition of primitive 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)
```

- $join : \{n\ m : nat\} \rightarrow \{t : data\} \rightarrow n.m.t \rightarrow nm.t$

```
-- The definition of primitive join
join : {n m : ℕ} → {T : Set} → Vec (Vec T n) m → Vec T (m * n)
join [] = []
join (xs :: xs₁) = xs ++ join xs₁
```

- Where `take` and `drop` are:

```
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
```

Equality Reasoning for Rewrite Rules - Split-Join (2)

- The split-join rule: $map\ f \rightarrow join \circ map\ (map\ f) \circ split\ n$

```
-- The proof of split-join rule
splitJoin : {m : ℕ} → {S 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
  ∎
```

Equality Reasoning for Rewrite Rules - Split-Join (3)

- 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
```

Proving `join` is Associative using Heterogeneous Equality (1)

- We have a rule stating `join` is associative: $join \circ join \to join \circ map\ join$
- When we tried to define the equality relation using propositional equality as:

```
joinBeforeJoin : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                 join (join xsss) ≡ join (map join xsss)
```

  The compiler complains:

```
n != n * m of type ℕ
when checking that the inferred type of an application
Vec T (n * (m * o))
matches the expected type
Vec T (n * m * o)
```

- `Vec T (n * (m * o))` and `Vec T (n * m * o)` are different types, even though the value of `(n * (m * o))` equals to `(n * m * o)` since multiplication is associative.
- We need an equality relation for different types, i.e., heterogeneous equality.

Proving `join` is Associative using Heterogeneous Equality (2)

- `join` is associative: $join \circ join \to join \circ map\ join$

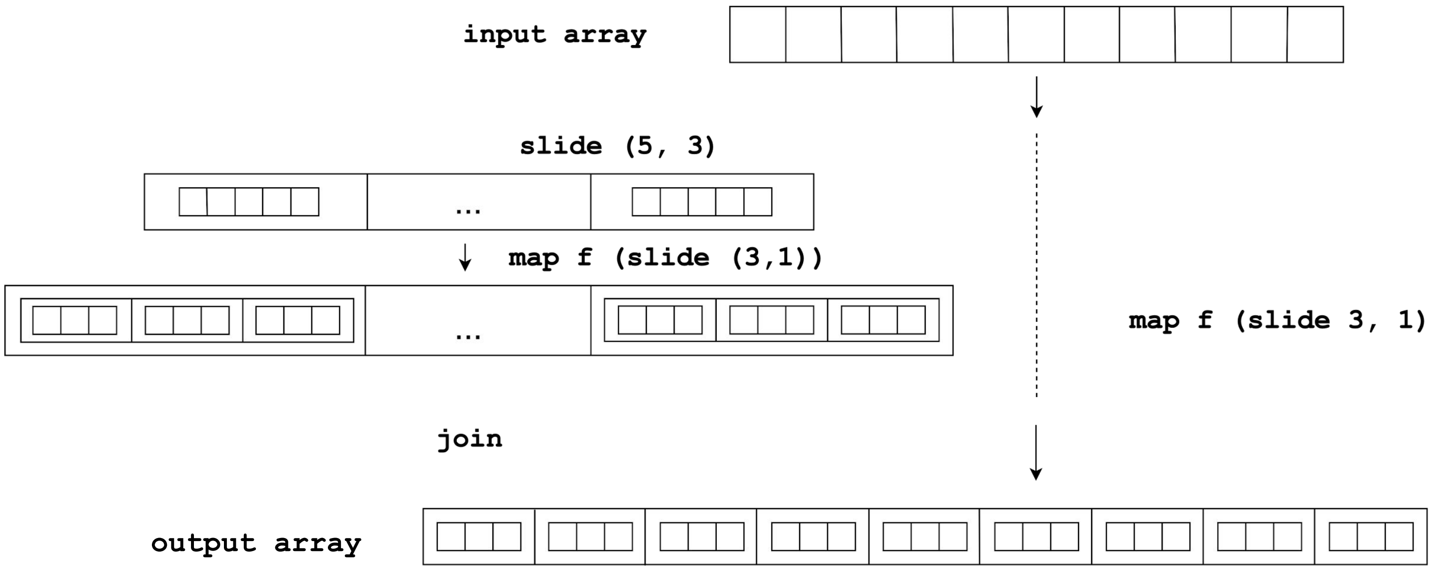
```
-- The proof of join is associative
joinBeforeJoin : {n m o : ℕ} → {T : Set} → (xsss : Vec (Vec (Vec T o) m) n) →
                 join (join xsss) ≅ join (map join xsss)
joinBeforeJoin [] = Heq.refl
joinBeforeJoin {suc n} {m} {o} {T} (xss :: xsss) =
  hbegin
    join (xss ++ join xsss)
  ≅⟨ join-++ xss (join xsss) ⟩
    join xss ++ join (join xsss)
  ≅⟨ hcong' (Vec T) (*-assoc n m o) (λ y → join xss ++ y) (joinBeforeJoin xsss) ⟩
    join xss ++ join (map join xsss)
  h∎
```

- Where `hcong'` is congruence in heterogeneous equality, `join-++` is a lemma:

```
join-++ : {n m o : ℕ} → {T : Set} → (xs1 : Vec (Vec T o) n) →
          (xs2 : Vec (Vec T o) m) → join (xs1 ++ xs2) ≅ join xs1 ++ join xs2
```

Equality Reasoning for Rewrite Rules - Tiling (1)

- A formal definition: $map\ f \circ slide\ size\ step \rightarrow join \circ map\ (\lambda\ tile.\ map\ f \circ (slide\ size\ step\ tile))\ slide\ u\ v$
- Example: size = 3, step = 1, u = 5, v = 3



Equality Reasoning for Rewrite Rules - Tiling (2)

- Issue: choices of *u* and *v* are not specified in paper, we only know:
  - $u - v = size - step$
- 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
- Let's define the primitive slide first:

Equality Reasoning for Rewrite Rules - Tiling (3)

- $slide : \{n : nat\} \to (sz\ sp : nat) \to \{t : data\} \to (sp * n + sz - sp).t \to n.sz.t$

```
-- The definition of primitive slide
slide : {n : ℕ} → (sz : ℕ) → (sp : ℕ) → {T : Set} → Vec T (sz + n * (suc sp)) →
        Vec (Vec T sz) (suc n)
slide {zero} sz sp xs = [ xs ]
slide {suc n} sz sp xs =
  take sz {(suc n) * (suc sp)} xs ::
  slide {n} sz sp (drop (suc sp) xs)
```

```
ERROR:
sz != sp of type ℕ
when checking that the expression xs has type
Vec T (suc sp + (sz + n * suc sp))
```

xs has type `Vec T (suc sz + (sp + n * suc sp))`

`drop (suc sp)` requires an argument with type `Vec T (suc sp + (sz + n * suc sp))`

`Vec T (suc sz + (sp + n * suc sp))` and `Vec T (suc sp + (sz + n * suc sp))` are not the same type, although we know the the sizes are equal and it's just the `xs` under this context.

Equality Reasoning for Rewrite Rules - Tiling (4)

- $slide : \{n : nat\} \to (sz\ sp : nat) \to \{t : data\} \to (sp * n + sz - sp).t \to n.sz.t$

```
-- The definition of primitive slide
slide : {n : ℕ} → (sz : ℕ) → (sp : ℕ) → {T : Set} → Vec T (sz + n * (suc sp)) →
        Vec (Vec T sz) (suc n)
slide {zero} sz sp xs = [ xs ]
slide {suc n} sz sp xs =
  take sz {(suc n) * (suc sp)} xs ::
  slide {n} sz sp (drop (suc sp) (cast (slide-lem n sz sp) xs))
```

- `cast` is used to cast the size of given array to satisfy pattern matching, defined as:

```
cast : {T : Set} → {m n : ℕ} → .(_ : m ≡ n) → Vec T m → Vec T n
cast {T} {zero} {zero} eq [] = []
cast {T} {suc m} {suc n} eq (x :: xs) = x :: cast {T} {m} {n} (cong pred eq) xs
```

Equality Reasoning for Rewrite Rules - Tiling (5)

- General ideas of developing proofs:
  - 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.
- Proof on the next slides:

Equality Reasoning for Rewrite Rules - Tiling (6)

```
-- the proof of the tiling rule
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))
tiling {n} {m} {s} {t} sz sp f xs =
  begin
    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))

  ≡⟨ cong join (map-λ {n} {m} sz sp f xs) ⟩ -- changing the order of the λ function
    join (map (map f) (map (slide {n} sz sp)
    (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)))

  ≡⟨ mapMapFBeforeJoin f (map (slide {n} sz sp)
    (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)) ⟩ -- changing the order of join
    map f (join (map (slide {n} sz sp) (slide {m} (sz + n * suc sp) (n + sp + n * sp) xs)))
  ≡⟨ cong (map f) (slideJoin {n} {m} sz sp xs) ⟩ -- the partitioning of slide
    refl
```

Equality Reasoning for Rewrite Rules - Tiling (7)

- Lemmas:

```
-- changing the order of the λ function
map-λ : {n m : ℕ} → {S T : Set} → (sz : N) → (sp :ℕ) → (f : Vec S sz → Vec T sz) →
  (xs : Vec S (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
  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 (map f) ((map (λ (tile : Vec S (sz + n * (suc sp))) →
  slide {n} sz sp tile)) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs))

-- changing the order of join
mapMapFBeforeJoin: {S T : Set} → {m n : ℕ} →
      (f : S → T) → (xs : Vec (Vec S n) m) →
      join (map (map f) xs) ≡ map f (join xs)
```

Equality Reasoning for Rewrite Rules - Tiling (8)

```
-- the partitioning of slide
slideJoin : {n m : ℕ} → {T : Set} → (sz : N) → (sp : N) →
     (xs : Vec T (sz + n * (suc sp) + m * suc (n + sp + n * sp))) →
     join (map (λ (tile : Vec T (sz + n * (suc sp))) →
     slide {n} sz sp tile) (slide {m} (sz + n * (suc sp)) (n + sp + n * sp) xs)) ≡
     slide {n + m * (suc n)} sz sp (cast (lem₁ n m sz sp) xs)
-- base case
slideJoin {n} {zero} sz sp xs =
  begin
    slide sz sp xs ++ []
  ≡⟨ ++-[] (slide sz sp xs) ⟩
    slide sz sp xs
  ≡⟨ cong (slide sz sp) (lem₂ {n} {sz} {sp} xs) ⟩
    refl
```

Equality Reasoning for Rewrite Rules - Tiling (9)

```
-- inductive case
slideJoin {n} {suc m} sz sp xs =
  begin
    slide {n} sz sp (take (sz + n * suc sp) xs) ++
    join (map (slide {n} sz sp) (slide {m} (sz + n * suc sp) (n + sp + n * sp)
    (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs))))
  ≡⟨ cong (slide {n} sz sp (take (sz + n * suc sp) xs) ++_)
     (slideJoin {n} {m} sz sp (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs))) ⟩
    slide {n} sz sp (take (sz + n * suc sp) xs) ++
    slide {n + m * suc n} sz sp (cast (lem₁ n m sz sp)
    (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs)))
  ≡⟨ lem₄ {n} {m} sz sp xs ⟩
    refl
```

Equality Reasoning for Rewrite Rules - Tiling (10)

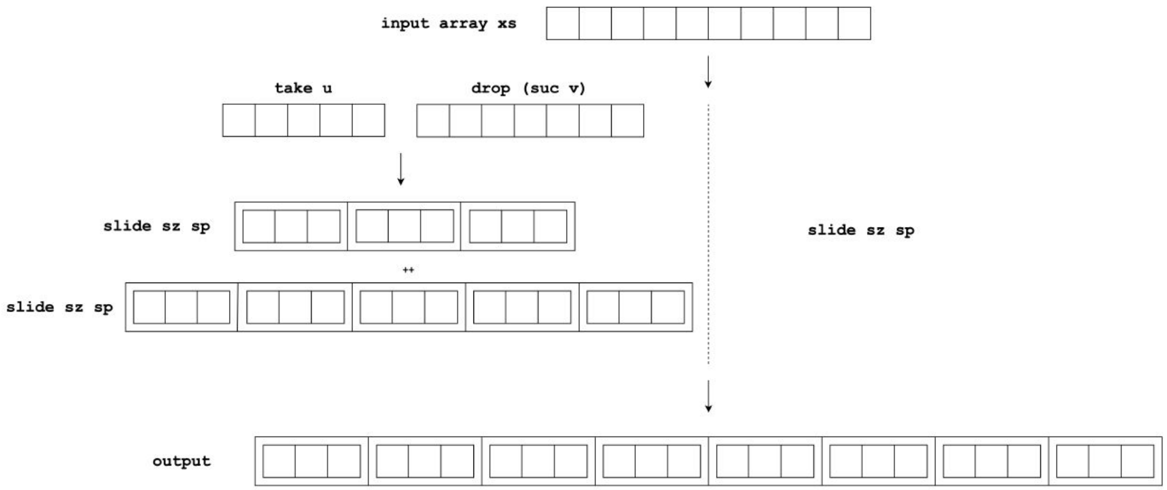
- Overcomplicated pattern matching in lem₄

```
postulate lem₄ : {n m : ℕ} → {T : Set} → (sz sp : ℕ) →
          (xs : Vec T (suc (sz + n * suc sp +
          (n + sp + n * sp + m * suc (n + sp + n * sp))))) →
          slide {n} sz sp (take (sz + n * suc sp)
          {suc (n + sp + n * sp + m * suc (n + sp + n * sp))} xs) ++
          slide {n + m * suc n} sz sp (cast (lem₁ n m sz sp)
          (drop (suc (n + sp + n * sp)) (cast (lem₃ n m sz sp) xs)))
          ≡
          take sz {suc (sp + (n + (n + m * suc n)) * suc sp)}
          (cast (lem₁ n (suc m) sz sp) xs) ::
          slide {n + (n + m * suc n)} sz sp
          (drop (suc sp) {sz + (n + (n + m * suc n)) * suc sp}
          (cast (slide-lem (n + (n + m * suc n)) sz sp )
          (cast (lem₁ n (suc m) sz sp) xs)))
```

It basically means: $\textit{slide}\ \textit{sz}\ \textit{sp} \circ \textit{take}\ u \mathbin{+\!\!+} \textit{slide}\ \textit{sz}\ \textit{sp} \circ \textit{drop}\ (\textit{suc}\ v) \longrightarrow \textit{slide}\ \textit{sz}\ \textit{sp}$

Equality Reasoning for Rewrite Rules - Tiling (11)

- We take *sz = 3*, *suc sp = 1*, *u = 5* and *suc v = 3* as an example:



- The RHS and LHS are obviously equal, however due to the overcomplicated pattern matching, we were not able to develop the proof.

Conclusion and Reflection

- Agda is helpful for formalising semantics and verifying rewrite rules
- The constraints on arrays' sizes in rewrite rules are specified and well maintained
- Reasoning about the equality between arrays' sizes can be complicated. We coped with this issue with some strategies:
  - Using `cast` to cast patterns at the constructor level
  - Using `REWRITE` to increase the flexibility of pattern matching
  - Using heterogeneous equality to reason about equality between two expression with different types
- However, sometimes the pattern matching is overcomplicated, causing some proofs not being able to be completed

Reference

A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In International Conference on Foundations of Software Science and Computational Structures, pages 57--71. Springer, 2011.

Agda Developer Team. Introduction to universes. URL https://agda.readthedocs.io/en/ latest/language/universe-levels.html. Accessed 2 Apr. 2020.

Agda Developer Team. The agda standard library, 2020. URL https://github.com/agda/ agda-stdlib. Accessed 2 Apr. 2020.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In Proceedings of the 2007 workshop on Programming languages meets program verification, pages 57--68, 2007.

R. Atkey, M. Steuwer, S. Lindley, and C. Dubach. Strategy preserving compilation for parallel functional code. CoRR, abs/1710.08332, 2017.

J. Cockx, N. Tabareau, and T.Winterhalter. How to tame your rewrite rules. Types for Proofs and Programs, TYPES, 2019.

H. B. Curry. Functionality in combinatory logic. Proceedings of the National Academy of Sciences of the United States of America, 20(11):584, 1934.

B. Hagdorn, M. Steuwer, R. Fu, and J. Lenfer. ELEVATE, 2020. URL https://github.com/elevate-lang/elevate/tree/master/src/main/scala/elevate/rise Accessed 2 Apr. 2020.

B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with Lift. In Proceedings of the 2018 International Symposium on Code Generation and Optimization, pages 100--112, 2018.

W. A. Howard. The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism, 44:479--490, 1980.

M. Steuwer. Improving programmability and performance portability on many-core processors. PhD thesis, University of Münster, 2015.

M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code. ACM SIGPLAN Notices, 50(9):205--217, 2015.

N. Ulf, A. D. Nils, and A. Andreas. Agda. URL https://wiki.portal.chalmers.se/agda/ pmwiki.php. Accessed 2 Apr. 2020.

P.Wadler. Propositions as types. Communications of the ACM, 58(12):75--84, 2015.

Thank you!

xueying.qin@ed.ac.uk
Project repository: https://github.com/XYUnknown/individual-project