Contents

1 Introduction

1.1 Motivation

1.2 Related work

1.3 Contributions

1.4 Overview

2 TTS System

2.1 Motivating Examples

2.1.1 Hughes' lists

One example of a type-changing program transformation is known as Hughes' lists [Hug86]. In his work, Hughes presents a method which reduces the computational overhead induced by the naive implementation of list concatenation. To see how this works, first consider the following implementation of list concatenation:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:xs ++ ys
infixr 5 ++
```

The running time of this function is dependent on the size of its first argument. Now let us see what calculations are being performed in the following examples.

```
s1,s2,s3,s4 :: [Char]
s1 = "aap" ++ ("noot" ++ "mies")
s2 = ("aap" ++ "noot") ++ "mies"
s3 = "aap" ++ "noot" ++ "mies"
s4 = (\x → x ++ "mies") ("aap" ++ "noot")
```

In the first example `"noot"` is traversed to create `"nootmies"`, and consecutively `"aap"` is traversed to create `"aapnootmies"`. The second example is almost identical, but first `"aapnoot"` is constructed by traversing `"aap"` and then `"aapnootmies"` is constructed after traversing `"aapnoot"`. Thus `"aap"` is traversed twice, a gross inefficiency! To partly counter this problem, (++) has been made right-associative, such that the third example produces the most optimal result. However, there are still many cases in which concatenation does not work optimal, as in the fourth example.

The Hughes' list transformation solves this by treating lists not as normal lists(`[a]`) but as functions over lists(`[a] → [a]`). Lists now become continuations of lists, where the continuation represents an unfinished list, for which the tail still has to be filled in. Lists and Hughes' lists can be transformed into each other by the functions **rep** and **abs**.

```
type HughesList a = [a] → [a]
rep :: [a] → HughesList a
rep ls = (ls ++)
```

```
abs :: HughesList a → [a]
abs c = c []
```

The speedup comes from the fact that, instead of normal concatenation, we can use function composition to concatenate two Hughes' lists.

```
s1, s2, s3, s4 :: [Char]
s1 = abs $ rep "aap" ∘ (rep "noot" ∘ rep "mies")
s2 = abs $ (rep "aap" ∘ rep "noot") ∘ rep "mies"
s3 = abs $ rep "aap" ∘ rep "noot" ∘ rep "mies"
s4 = abs $ (\x → x ∘ rep "mies") (rep "aap" ∘ rep "noot")
```

All examples now have the same, optimal running time because the continuation technique avoids building intermediate results: each list is only traversed at most once. Additionally, where the speed of normal concatenation depends on the size of its first argument, function composition has a constant running time.

2.1.2 Stream fusion

Another example of a type-changing program transformation is stream fusion, as found in Coutts et al. [CLS07, CSL07]. The goal of stream fusion is the same as Hughes' lists: optimizing operations on lists. Stream fusion does this using a technique called deforestation, which reduces the number of intermediate results constructed when doing operations on lists. Consider the following example:

```
map f ∘ filter c ∘ map g
```

When this example is compiled without optimization, an intermediate result will be constructed at the position of each composition. Modern compilers such as GHC can partly optimize this kind of overhead away, but not for all cases. A better solution is to use streams instead of lists. Streams are defined as follows:

```
data Step s a =
    Done
  | Yield a s
  | Skip s
data Stream a = some s. Stream {step :: s → Step s a, seed :: s}
```

Streams do not represent lists directly but store a seed and a function which can be used to obtain a new item from the list and a modified seed (Yield). When the list is empty Done will be returned and Skip will just modify the seed. This becomes more clear when we look at the function which converts a stream into a list:

```
abs :: Stream a → [a]
abs stream = extract $ seed stream
  where
```

```
    extract seed' =
      case step stream seed' of
        Done              → [ ]
        Skip newseed      → extract newseed
        Yield x newseed → x : extract newseed
```

In a similar fashion we can construct a stream from a list:

```
rep :: [a] → Stream a
rep ls = Stream next ls
  where
    next [ ] = Done
    next (x:xs) = Yield x xs
```

Here the list seed becomes the seed and the step function produces an item from the list at each step.

We first have to show that toStream and fromStream form a retraction pair. We show this by induction on the list argument.

Empty list:

```
  fromStream (toStream [ ])
≡ {Definition toStream}
  fromStream (Stream next [ ])
≡ {Definition fromStream}
  extract [ ]
≡ {Definition extract and next}
  [ ]
```

Cons case:

```
  fromStream (toStream (x : xs))
≡ {Definition toStream}
  fromStream (Stream next (x : xs))
≡ {Definition fromStream}
  extract (x : xs)
≡ {Definition extract and next}
  x : extract xs
≡ {Defintition fromStream}
  x : fromStream (Stream next xs)
≡ {Defintion toStream}
  x : fromStrean (toStream xs)
≡ {Induction hypothesis}
  x : xs
```

We can now define the retraction for the transformation system:

```
A a = [a]
R a = Stream a
rep = toStream
abs = fromStream
```

We need both the polymorphism and the parametrized types to make this example work. Not that this system also support nested application of transformation, so a term of type [[Int]] could be transformed to the type Stream (Stream Int).

The actual optimization in this transformation comes from the rewriting of functions which make use of the optimized stream structure. We give an example of map here with the proof:

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (x : xs) = f x : map f xs
mapS :: (a → b) → Stream a → Stream b
mapS f (Stream next seed) = Stream next' seed
  next' s =
    case next s of
      Done    → Done
      Skip s' → Skip s'
      Yield a s' → Yield (f a) s'
```

From this we can make the following transformation rule:

$\Phi = (\mathsf{a} \to \mathsf{b}) \to \mathsf{Id\ a} \to \mathsf{Id\ b}$
$\mathsf{map} \rightsquigarrow \mathsf{mapS} : \Phi$

However, we first have to show that mapS is a proper implementation for map, by showing:

$dimap_{\Phi}\ \mathbf{rep}\ \mathbf{abs}\ \mathsf{mapS\ f\ x} \equiv \mathsf{map\ f\ x}$

$dimap_{\Phi}\ \mathsf{toStream\ fromStream\ mapS\ f\ x}$
$\equiv \{\text{Definition } dimap_{\Phi}\}$
$(dimap_{\mathsf{Id\ a} \to \mathsf{Id\ b}}\ \mathbf{rep\ abs} \circ \mathsf{mapS} \circ dimap_{\mathsf{a} \to \mathsf{b}}\ \mathbf{abs\ rep})\ \mathsf{f\ x}$
$\equiv \{\text{Evaluation}\}$
$(dimap_{\mathsf{Id\ a} \to \mathsf{Id\ b}}\ \mathbf{rep\ abs}\ \$\ \mathsf{mapS}\ \$\ dimap_{\mathsf{a} \to \mathsf{b}}\ \mathbf{abs\ rep}\ \mathsf{f})\ \mathsf{x}$
$\equiv \{\text{Definition } dimap_{\mathsf{Id\ a} \to \mathsf{Id\ b}}\}$
$(dimap_{\mathsf{Id\ b}}\ \mathbf{rep\ abs} \circ (\mathsf{mapS}\ \$\ dimap_{\mathsf{a} \to \mathsf{b}}\ \mathbf{abs\ rep}\ \mathsf{f}) \circ dimap_{\mathsf{Id\ a}}\ \mathbf{abs\ rep})\ \mathsf{x}$
$\equiv \{\text{Evaluation}\}$
$dimap_{\mathsf{Id\ b}}\ \mathbf{rep\ abs}\ \$\ (\mathsf{mapS}\ \$\ dimap_{\mathsf{a} \to \mathsf{b}}\ \mathbf{abs\ rep}\ \mathsf{f})\ \$\ dimap_{\mathsf{Id\ a}}\ \mathbf{abs\ rep}\ \mathsf{x}$
$\equiv \{\text{Defintition } dimap.\}$
$dimap_{[\mathsf{b}]}\ \mathbf{rep\ abs}\ \$\ \mathbf{abs}\ \$\ (\mathsf{mapS}\ \$\ dimap_{\mathsf{a} \to \mathsf{b}}\ \mathbf{abs\ rep}\ \mathsf{f})\ \$\ dimap_{\mathsf{Stream\ a}}\ \mathbf{abs\ rep}\ \$\ \mathbf{rep}\ \mathsf{x}$

We now proof by induction on x

2.2 Type and Transform Systems

Looking closely at the previous two examples, we can see a transformation pattern emerge. In both examples a single type is changed into another, better optimized type. All functions on the original type are replaced by functions on the optimized type, while maintaining the same semantic behaviour. The functions **rep** and **abs** are used to perform conversions between both types.

Type and transform systems are a formalization of this transformation pattern.

2.2.1 Object Language

2.2.2 Typing functor

2.3 An STLC-based transformation system

A type and transform system (TTS) transforms a program which, when the transformation is successful, guarantees the following TTS properties:

- The source and result program are well-typed
- The source and result program are semantically equivalent

At the heart of a TTS is the TTS relation. The TTS relation specifies which well-typed source programs can be turned into a well-typed result program, as such:

$$\mathsf{e} : \tau \rightsquigarrow \mathsf{e'} : \tau'$$

Elements of this relation are defined using inference rules, much like inference rules in normal type systems. However, the TTS system validates and types the source and result terms simultaneously. The inference rules of the TTS system should also make sure that the TTS properties we defined for the system are maintained.

For a TTS system to be of any use, it should allow the user to specify transformations rules. Because the system still has to make sure the TTS properties are satisfied, the TTS can place restrictions on the transformations supplied by the user and the form of the source program. Thus the trick in creating a useful TTS is keeping the restrictions to a minimum while still being able to prove the TTS properties.

Thus far we have not defined what form the terms and types of the TTS system should be, nor have we specified what the user-created transformations should look like. A TTS is a general concept and could be defined for any terms and types as long as we can prove the desired TTS properties within the system we are defining.

The language for which we design the TTS is called the object language. We will now give an example of a simple TTS with as object language the simply typed lambda calculus.

2.4 A TTS for STLC

In this chapter we present a TTS for the simply typed lambda calculus. Although this is a simple example, it contains all the essential elements a TTS should have. A proof of correctness of this system can be found in Appendix A.

To recap, the terms and types of the simply typed lambda calculus are of the following form:

```
e ::= x | c | e e | \x. e
τ ::= T | τ → τ
```

2.4.1 The typing functor

Because we are building a TTS we want to allow the types of terms to change. However, allowing arbitrary type changes makes proving the TTS properties very hard. We want to maintain control over how and where the types have changed. To this end, we extend the normal STLC types with a 'hole' (Id) as follows.

```
Φ ::= Id | T | Φ → Φ
```

This hole is a special construct that can be filled in with a normal type to obtain a normal type again, as defined by the following interpretation function:

```
«F»ty → τ
«T»ty            = T
«Id»ty           = τ
«F_1 → F_2»ty = «F_1»ty → «F_2»ty
```

Thus Φ can be applied to a type to yield a new type. We call Φ a typing functor. We can now use this typing functor to express that we only want to change one type in the program, by constructing the TTS judgement in the following way:

```
e : «F»A ⇝ e' : «F»R
```

This enforces that only As are transformed into Rs, the rest of the type remains the same. The types types A and R play a special role. In the final implementation of the system the user can manually specify which types a transformation will transform. Thus A and R are 'global' in the TTS system and we implicitly assume them to be specified. Because of this, we rewrite the TTS judgement in a shorter form:

```
e ⇝ e' : Φ
```

where the properties typeOf (e) ≡ «F»A and typeOf (e') ≡ «F»R are left implicit.

STLC inference rules also contain a typing environment which assumes types for unbound variables. We want to allow changes in the types of unbound variables, but we also want to allow changing of the variables themselves to allow for rewriting. Thus we get the following rewrite environment:

$$\Gamma ::= \emptyset \mid \Gamma, \mathsf{x} \leadsto \mathsf{x'} : \Phi$$

Thus we have merged both the types and the environments of the source and result program into one, with the functor Φ accounting for the differences that may exist. With these building blocks in place, we and up with the following judgement for our STLC TTS system:

$$\Gamma \vdash \mathsf{e} \leadsto \mathsf{e'} : \Phi$$

The typing functor plays a crucial part in connecting the source and result programs. Before looking at user-supplied transformation rules, we will first introduce some theory behind functors.

2.4.2 Typing system

We now have the basic ingredients to define our TTS. The system is defined in Figure 2.1. The Var, Abs and App rule are very similar to the rules in STLC, except with an extra term and the functor instead of a type. These rules form the identity rules. If no rewrite would be applied these rules yield the identity transformation.

Shadowing on the rewrite environment Γ removes the rewrite rules which have a matching source and/or target term. This makes sure we do not apply rewrite rules to newly introduced variables, only to global definitions.

The RWVar rule rewrites a variable using a user-specified rule. The Rep and Abs rules can rewrite any term which is of the correct type. The Final rule in Figure 2.2 finalizes a transformation and concludes that both terms are semantically equal. This is only the case when there are no free variables and the type of the source and target terms are equal.

The next step is to turn these typing rules into an algorithm which will actually do a transformation. This will be done in the next section. We would also like to see proof that these rules only allow semantics preserving transformations. The proof of this can be found in appendix A.

2.5 A TTS for Hughes' lists

With the basic STLC TTS system in place, we can now define TTS system for Hughes' list transformation.

2.5.1 Example

2.6 A TTS for Stream fusion

$$\text{Id} \qquad \frac{}{\Gamma \vdash \mathsf{x} \rightsquigarrow \mathsf{x} : \Phi}$$

$$\text{Var} \qquad \frac{\mathsf{x} \rightsquigarrow \mathsf{x'} : \Phi \in \Gamma}{\Gamma \vdash \mathsf{x} \rightsquigarrow \mathsf{x'} : \Phi}$$

$$\text{Lambda} \qquad \frac{\Gamma^{x}, \mathsf{x} \rightsquigarrow \mathsf{x} : \Phi_a \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \Phi_r}{\Gamma \vdash \backslash\mathsf{x}.\, \mathsf{e} \rightsquigarrow \backslash\mathsf{x}.\, \mathsf{e'} : \Phi_a \rightarrow \Phi_r}$$

$$\text{App} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{f} \rightsquigarrow \mathsf{f'} : \Phi_a \rightarrow \Phi_r \\ \Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \Phi_a\end{array}}{\Gamma \vdash \mathsf{f}\,\mathsf{e} \rightsquigarrow \mathsf{f'}\,\mathsf{e'} : \Phi_r}$$

$$\text{I-Rep} \qquad \frac{\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{A}}{\Gamma \vdash \mathsf{e} \rightsquigarrow \mathbf{rep}\,\mathsf{e'} : \mathsf{Id}}$$

$$\text{I-Abs} \qquad \frac{\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{Id}}{\Gamma \vdash \mathsf{e} \rightsquigarrow \mathbf{abs}\,\mathsf{e'} : \mathsf{A}}$$

$$\text{Judgement} \qquad \boxed{\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \Phi}$$

Figure 2.1: Type checking rules for the propagation relation

$$\text{Final} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e'} : \Phi \\ \forall\, \mathsf{x} \rightsquigarrow \mathsf{x'} : \Phi_2 \in \Gamma, \mathit{dimap}_{\Phi 2}\ \mathbf{rep}\ \mathbf{abs}\ \mathsf{x'} \equiv \mathsf{x} \\ \text{«F»A} = \text{«F»R}\end{array}}{\mathsf{e} \equiv \mathsf{e'}}$$

Figure 2.2: Final rule to establish the equality between terms

$$\text{Abs}\ \dfrac{\text{App}\ \dfrac{\text{Abs}\ \dfrac{\text{Comp}\ \dfrac{\text{Var}\ \dfrac{}{\mathsf{x} : \mathsf{Id}, \emptyset \vdash \mathsf{x} \rightsquigarrow \mathsf{x} : \mathsf{Id}} \qquad \text{Rep}\ \dfrac{\text{Id}\ \dfrac{}{\emptyset \vdash \texttt{"mies"} \rightsquigarrow \texttt{"mies"} : \mathsf{String}}}{\mathsf{x} : \mathsf{Id}, \emptyset \vdash \texttt{"mies"} \rightsquigarrow \mathbf{rep}\ \texttt{"mies"} : \mathsf{Id}}}{\mathsf{x} : \mathsf{Id}, \emptyset \vdash \mathsf{x} \mathbin{+\!\!+} \texttt{"mies"} \rightsquigarrow \mathsf{x} \circ \mathbf{rep}\ \texttt{"mies"} : \mathsf{Id}}}{\emptyset \vdash \backslash\mathsf{x}.\, \mathsf{x} \mathbin{+\!\!+} \texttt{"mies"} \rightsquigarrow \backslash\mathsf{x}.\, \mathsf{x} \circ \mathbf{rep}\ \texttt{"mies"} : \mathsf{Id} \rightarrow \mathsf{Id}} \qquad \text{Rep}\ \dfrac{\text{Id}\ \dfrac{}{\emptyset \vdash \texttt{"aap"} \rightsquigarrow \texttt{"aap"} : \mathsf{String}}}{\emptyset \vdash \texttt{"aap"} \rightsquigarrow \mathbf{rep}\ \texttt{"aap"} : \mathsf{Id}}}{\emptyset \vdash (\backslash\mathsf{x}.\, \mathsf{x} \mathbin{+\!\!+} \texttt{"mies"})\ \texttt{"aap"} \rightsquigarrow (\backslash\mathsf{x}.\, \mathsf{x} \circ \mathbf{rep}\ \texttt{"mies"})\ (\mathbf{rep}\ \texttt{"aap"}) : \mathsf{Id}}}{\emptyset \vdash (\backslash\mathsf{x}.\, \mathsf{x} \mathbin{+\!\!+} \texttt{"mies"})\ \texttt{"aap"} \rightsquigarrow \mathbf{abs}\ \$\ (\backslash\mathsf{x}.\, \mathsf{x} \circ \mathbf{rep}\ \texttt{"mies"})\ (\mathbf{rep}\ \texttt{"aap"}) : \mathsf{String}}$$

3 Tools of the Trade

3.1 Functors

A functor can be seen as a function on types. It takes as parameter a type and yields a new type based on its argument. Associated with a type level functor is a term level functor which lifts functions on the type parameter to functions on the functor:

```
type Φ a                             –F unctor
fmap :: (a → b) → Φ a → Φ b –t erm level functor
```

For fmap to be a proper term-level functor it has to obey the functor laws:

```
fmap id = id                        –I dentity
fmap g ∘ fmap f = fmap (g ∘ f) –C omposition
```

A list is an example of a Functor in Haskell:

```
data List a = Nil | Cons a (List a)
fmap :: (a → b) → List a → List b
fmap _ Nil          = Nil
fmap f (Cons a l) = Cons (f a) (fmap f l)
```

What makes functors special, is that an implementation for the term level function fmap can be constructed from the functor type. This is what makes functors useful for our purpose. We can reason about the semantics of the terms by knowing the types.

For normal functors we can only construct a term-level functor when the argument type occurs in covariant positions within the datatype. This means we can construct fmap for all polynomial types (datatypes without functions), but not for all datatypes containing functions. However, our typing functor Φ includes a function space constructor, so we will need something more powerful.

Meijer and Hutton[**?**] showed that it is possible to define functors for function types (exponential types) when we use *dimap.*s. A *dimap.* is the same as as a normal fmap but with an extra function argument which can be used for occurrences of the type parameter at contra-variant positions. The functor laws also have an extended version for *dimap.*s:

$$dimap_{\Phi} :: (a \to b) \to (b \to a) \to \Phi\ b \to \Phi\ a$$

$$dimap_{\Phi}\ id\ id = id$$
$$dimap_{\Phi}\ g1\ g2 \circ dimap_{\Phi}\ f1\ f2 = dimap_{\Phi}\ (f1 \circ g1)\ (g2 \circ f2)$$

Note how the first function is contra-variant and the second covariant in the result type.

Based on this work, we can define how the term-level *dimap.* is derived from our typing functor using the following type-indexed function:

$$
\begin{array}{lll}
\mathit{dimap}_{\Phi} :: (\mathsf{a} \to \mathsf{b}) \to (\mathsf{b} \to \mathsf{a}) \to \text{«}\mathsf{F}\text{»}\mathsf{b} \to \text{«}\mathsf{F}\text{»}\mathsf{a} & & \\
\mathit{dimap}_{Id} & \mathbf{rep}\ \mathbf{abs}\ \mathsf{x} & = \mathbf{abs}\ \mathsf{x} \\
\mathit{dimap}_{T} & \mathbf{rep}\ \mathbf{abs}\ \mathsf{x} & = \mathsf{x} \\
\mathit{dimap}_{\Phi 1 \to \Phi 2} & \mathbf{rep}\ \mathbf{abs}\ \mathsf{f} & = \mathit{dimap}_{\Phi 2}\ \mathbf{rep}\ \mathbf{abs} \circ \mathsf{f} \circ \mathit{dimap}_{\Phi 1}\ \mathbf{abs}\ \mathbf{rep}
\end{array}
$$

3.2 Retractions

3.3 Equational Reasoning

4 Proof

4.1 Hoi

4.1.1 Inference rule properties

What is left is to proof that the property is preserved by all derivation rules. Of course, this assumes that the user has supplied a transformation which adheres to the required properties.

Var rule

$$\frac{\text{x} \rightsquigarrow \text{x'} : \Phi \in \Gamma}{\Gamma \vdash \text{x} \rightsquigarrow_\rho \text{x'} : \Phi}$$

We have to proof $\mathit{dimap}_\Phi\ \Gamma\ \mathbf{rep}\ \mathbf{abs}\ \text{x'} \equiv \text{x}$

$$\begin{aligned}
&\mathit{dimap}_\Phi\ \Gamma\ \mathbf{rep}\ \mathbf{abs}\ \text{x'}\\
\equiv\ &\{|\text{x} \rightsquigarrow \text{x'} : \Phi \in \text{G}|\}\\
&\mathit{dimap}_\Phi\ (\Gamma^x, \text{x} \rightsquigarrow \text{x'} : \Phi_a)\ \mathbf{rep}\ \mathbf{abs}\ \text{x'}\\
\equiv\ &\{\text{Definition } \mathit{dimap}_\Phi\ \Gamma\}\\
&\mathit{dimap}_\Phi\ \mathbf{rep}\ \mathbf{abs}\ \text{x'}\ @\ \text{mkSub}\ (\Gamma^x, \text{x} \rightsquigarrow \text{x'} : \Phi_a)\\
\equiv\ &\{\text{Definition mkSub}\}\\
&\mathit{dimap}_\Phi\ \mathbf{rep}\ \mathbf{abs}\ \text{x'}\ @\ [\text{x'/dimap\_F}\ \mathbf{abs}\ \mathbf{rep}\ \text{x}] \circ \text{mkSub}\ \Gamma^x\\
\equiv\ &\{\text{Apply substitution}\}\\
&\mathit{dimap}_\Phi\ \mathbf{rep}\ \mathbf{abs}\ \text{x'}\ @\ [\text{x'/dimap\_F}\ \mathbf{abs}\ \mathbf{rep}\ \text{x}] \circ \text{mkSub}\ \Gamma^x\\
\equiv\ &\{\text{Definition mkSub}\}\\
&\mathit{dimap}_\Phi\ \mathbf{rep}\ \mathbf{abs}\ (\mathit{dimap}_\Phi\ \mathbf{abs}\ \mathbf{rep}\ \text{x})\ @\ \text{mkSub}\ \Gamma^x\\
\equiv\ &\{\text{No eligible substitution } \textbf{in}\ \Gamma^x\}\\
&\mathit{dimap}_\Phi\ \mathbf{rep}\ \mathbf{abs}\ (\mathit{dimap}_\Phi\ \mathbf{abs}\ \mathbf{rep}\ \text{x})\\
\equiv\ &\{\text{Functor composition}\}\\
&\mathit{dimap}_\Phi\ (\mathbf{abs} \circ \mathbf{rep})\ (\mathbf{abs} \circ \mathbf{rep})\ \text{x}\\
\equiv\ &\{\text{Retraction identity}\}\\
&\mathit{dimap}_\Phi\ \text{id id x}\\
\equiv\ &\{\text{Functor identity}\}\\
&\text{x}
\end{aligned}$$

Abstraction rule

$$\frac{\Gamma^x; \text{x} \rightsquigarrow \text{x} : \Phi_a \vdash \text{e} \rightsquigarrow_\rho \text{e'} : \Phi_r}{\Gamma \vdash \backslash\text{x. e} \rightsquigarrow_\rho \backslash\text{x. e'} : \Phi_a \to \Phi_r}$$

We have to proof the following equivalence:

$$\textit{dimap}_{\Phi_a \to \Phi_r}\ \Gamma\ \textbf{rep}\ \textbf{abs}\ (\backslash \mathsf{x.\ e'}) \ \equiv\ \backslash \mathsf{x.\ e}$$

From the premises we know that:

$$\textit{dimap}_{\Phi r}\ (\Gamma^x, \mathsf{x} \rightsquigarrow \mathsf{x} : \Phi_a)\ \textbf{rep}\ \textbf{abs}\ \mathsf{e'} \ \equiv\ \mathsf{e}$$

Proof:

$\textit{dimap}_{\Phi_a \to \Phi_r}\ \Gamma\ \textbf{rep}\ \textbf{abs}\ (\backslash \mathsf{x.\ e'})$
$\equiv$ {Definition $\textit{dimap}_{\Phi_a \to \Phi_r}\ \Gamma$}
$\textit{dimap}_{\Phi_a \to \Phi_r}\ \textbf{rep}\ \textbf{abs}\ (\backslash \mathsf{x.\ e'})\ @\ \mathsf{mkSub}\ \Gamma$
$\equiv$ {Commute substitution over lambda}
$\textit{dimap}_{\Phi_a \to \Phi_r}\ \textbf{rep}\ \textbf{abs}\ (\backslash \mathsf{x.\ e'}\ @\ \mathsf{mkSub}\ \Gamma^x)$
$\equiv$ {Definition *dimap.*}
$\textit{dimap}_{\Phi r}\ \Gamma\ \textbf{rep}\ \textbf{abs} \circ (\backslash \mathsf{x.\ e'}\ @\ \mathsf{mkSub}\ \Gamma^x) \circ \textit{dimap}_{\Phi a}\ \Gamma\ \textbf{abs}\ \textbf{rep}$
$\equiv$ {Eta expansion}
$(\backslash \mathsf{x.}\ \textit{dimap}_{\Phi r}\ \Gamma\ \textbf{rep}\ \textbf{abs} \circ (\backslash \mathsf{x.\ e'}\ @\ \mathsf{mkSub}\ \Gamma^x) \circ \textit{dimap}_{\Phi a}\ \Gamma\ \textbf{abs}\ \textbf{rep}\ \$\ \mathsf{x})$
$\equiv$ {Evaluation}
$(\backslash \mathsf{x.}\ \textit{dimap}_{\Phi r}\ \textbf{rep}\ \textbf{abs}\ (\mathsf{e'}\ @\ [\mathsf{x/dimap\_Fa}\ \textbf{abs}\ \textbf{rep}\ \mathsf{x}] \circ \mathsf{mkSub}\ \Gamma^x)$
$\equiv$ {Definition mkSub}
$(\backslash \mathsf{x.}\ \textit{dimap}_{\Phi r}\ \textbf{rep}\ \textbf{abs}\ \mathsf{e'}\ @\ \mathsf{mkSub}\ (\Gamma^x, \mathsf{x} \rightsquigarrow \mathsf{x} : \Phi_a))$
$\equiv$ {Definition *dimap.*}
$(\backslash \mathsf{x.}\ \textit{dimap}_{\Phi r}\ (\Gamma^x, \mathsf{x} \rightsquigarrow \mathsf{x} : \Phi_a)\ \textbf{rep}\ \textbf{abs}\ \mathsf{e'})$
$\equiv$ {Premisse}
$(\backslash \mathsf{x.\ e})$

Application rule

$$\frac{\Gamma \vdash \mathsf{f} \rightsquigarrow_\rho \mathsf{f'} : \Phi_a \to \Phi_r \qquad \Gamma \vdash \mathsf{e} \rightsquigarrow_\rho \mathsf{e'} : \Phi_a}{\Gamma \vdash \mathsf{f\ e} \rightsquigarrow_\rho \mathsf{f'\ e'} : \Phi_r}$$

We have to proof the following equality:

$$\textit{dimap}_{\Phi r}\ \Gamma\ \textbf{rep}\ \textbf{abs}\ (\mathsf{f'\ e'}) \ \equiv\ \mathsf{f\ e}$$

From the premises we know that:

$\textit{dimap}_{\Phi}\ \Gamma\ \textbf{rep}\ \textbf{abs}\ \mathsf{f'} \ \equiv\ \mathsf{f}$
$\textit{dimap}_{\Phi a}\ \Gamma\ \textbf{rep}\ \textbf{abs}\ \mathsf{e'} \ \equiv\ \mathsf{e}$

Proof:

$\quad dimap_{\Phi r}\,\Gamma$ **rep abs** (f' e')
$\equiv$ {Definition $dimap_{\Phi r}\,\Gamma$}
$\quad dimap_{\Phi r}$ **rep abs** (f' e') @ mkSub Γ
$\equiv$ {Property below}
$\quad dimap_{\Phi}$ **rep abs** f' ($dimap_{\Phi A}$ **rep abs** e') @ mkSub Γ
$\equiv$ {Distribute substitution}
$\quad dimap_{\Phi}$ **rep abs** f' @ mkSub Γ ($dimap_{\Phi A}$ **rep abs** e' @ mkSub Γ)
$\equiv$ {Definition $dimap.$}
$\quad dimap_{\Phi}\,\Gamma$ **rep abs** f' ($dimap_{\Phi A}\,\Gamma$ **rep abs** e')
$\equiv$ {Induction hypotheses}
$\quad$ f e

Extra property $dimap_{\Phi r}$ **rep abs** (f' e') $\equiv$ $dimap_{\Phi_a \to \Phi_r}$ **rep abs** f' ($dimap_{\Phi a}$ **rep abs** e')

$\quad dimap_{\Phi_a \to \Phi_r}$ **rep abs** f' ($dimap_{\Phi a}$ **rep abs** e')
$\equiv$ {Definition of $dimap.$}
$\quad dimap_{\Phi r}$ **rep abs** \$ f' \$ $dimap_{\Phi a}$ **abs rep** \$ $dimap_{\Phi A}$ **rep abs** e'
$\equiv$ {Functor composition}
$\quad dimap_{\Phi R}$ **rep abs** \$ f' \$ $dimap_{\Phi a}$ (**rep** $\circ$ **abs**) (**rep** $\circ$ **abs**) e'
$\equiv$ {**rep** . **abs** :: i $\equiv$ id}
$\quad dimap_{\Phi r}$ **rep abs** \$ f' \$ $dimap_{\Phi a}$ id id e'
$\equiv$ {Functor identity}
$\quad dimap_{\Phi r}$ **rep abs** (f' e')

Rep rule

Applying rep to some source term results in a term which can be made equal to the source term by applying abs to it. This is reflected in the reasoning below.

$$\frac{\Gamma \vdash e \rightsquigarrow_{\rho} e' : A}{\Gamma \vdash e \rightsquigarrow_{\rho} \mathbf{rep}\ e' : Id}$$

We need to proof that $dimap_{Id}\,\Gamma$ **rep abs** (**rep** e') $\equiv$ e. From the premises we know that $dimap_{A}\,\Gamma$ **rep abs** e' $\equiv$ e.

$\quad dimap_{Id}\,\Gamma$ **rep abs** (**rep** e')
$\equiv$ {Definition $dimap_{Id}\,\Gamma$}
$\quad dimap_{Id}$ **rep abs** (**rep** e') @ mkSub Γ
$\equiv$ {Definition $dimap_{Id}$}
$\quad$ **abs** (**rep** e') @ mkSub Γ
$\equiv$ {Retraction}
$\quad$ e' @ mkSub Γ

$\equiv$ {Identity function}
id e' @ mkSub Γ
$\equiv$ {Definition $dimap_A$}
$dimap_A$ **rep abs** e' @ mkSub Γ
$\equiv$ {Definition $dimap_A$ Γ}
$dimap_A$ Γ **rep abs** e'
$\equiv$ {Premisse}
e

Abs rule

Applying abs to a suitable term equalizes the result and source term.

$$\frac{\Gamma \vdash \text{e} \leadsto_{\rho} \text{e'} : \text{Id}}{\Gamma \vdash \text{e} \leadsto_{\rho} \textbf{abs}\ \text{e'} : \text{A}}$$

We have to proof $dimap_A$ Γ **rep abs** (**abs** e') $\equiv$ e. From the premises we know that $dimap_{Id}$ Γ **rep abs** e' $\equiv$ e.

$dimap_A$ Γ **rep abs** (**abs** e')
$\equiv$ {Definition $dimap_A$ Γ}
$dimap_A$ **rep abs** (**abs** e') @ mkSub Γ
$\equiv$ {Definition $dimap_A$}
abs e' @ mkSub Γ
$\equiv$ {Definition $dimap_{Id}$}
$dimap_{Id}$ **rep abs** e' @ mkSub Γ
$\equiv$ {Defintion $dimap_{Id}$ Γ}
$dimap_{Id}$ Γ **rep abs** e'
$\equiv$ {Premise}
e

Transform rule

$$\frac{\begin{array}{c} \text{p1} \leadsto \text{p2} \in \rho \\ \Gamma;\Delta \vdash \text{e} \leadsto_{\rho} \text{e'} : \Phi_1 \\ \Gamma;\Delta \vdash_{rw} \text{p1} \leadsto \text{p2} \Rightarrow \text{e'} : \Phi_1\ \text{R} \leadsto \text{e''} : \Phi_2\ \text{R} \end{array}}{\Gamma;\Delta \vdash_{tr} \text{e} \leadsto_{\rho} \text{e''} : \Phi_2}$$

From the Rewrite premise we get the assertions that

Φ_1 A $=$ Φ_2 A
$dimap_{\Phi 1}$ **rep abs** θ_Θ (e) $=$ $dimap_{\Phi 2}$ **rep abs** θ_Θ (e')

From this we can see that the type of e is still valid after transformation ($\Phi_1\ \mathsf{A} \equiv \Phi_2\ \mathsf{A}$). For the terms the proof also follows easily.

$$\begin{aligned}
&\ \ \mathit{dimap}_{\Phi 2}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Gamma\ (\mathsf{e''})\\
&= \{\mathsf{Premisse} \vdash_{rw}\}\\
&\ \ \mathit{dimap}_{\Phi 1}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Gamma\ (\mathsf{e'})\\
&= \{\mathsf{Premisse} \vdash_{tr}\}\\
&\ \ \mathsf{e}
\end{aligned}$$

Rewrite rule

$$\frac{\begin{array}{l}\Gamma;\Delta \vdash_{mat} \mathsf{p1\ @\ e1} : \Phi_1\ \mathsf{R} \Rightarrow \mathsf{S}\\ \Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p2} \Rightarrow \mathsf{e2} : \Phi_2\ \mathsf{R}\\ \qquad\quad \Phi_1\ \mathsf{A} = \Phi_2\ \mathsf{A}\end{array}}{\Gamma;\Delta \vdash_{rw} \mathsf{p1} \rightsquigarrow \mathsf{p2\ @\ e1} : \Phi_1\ \mathsf{R} \Rightarrow \mathsf{e2} : \Phi_2\ \mathsf{R}}$$

First We have to proof that $\Phi_1\ \mathsf{A} = \Phi_2\ \mathsf{A}$, this follows easily from the premises.
The second thing we have to show is that:

$$\mathit{dimap}_{\Phi 1}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Theta\ (\mathsf{e}) = \mathit{dimap}_{\Phi 2}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Theta\ (\mathsf{e'})$$

We have restricted the user to only allow rewrite rules which abide the following law:

$$\frac{\begin{array}{c}\forall\ \mathsf{S},\Delta,\Gamma \qquad \Theta = \mathsf{mkSub}\,(\Delta)\\ \Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p1} \Rightarrow \mathsf{e1} : \Phi_1\ \mathsf{R}\\ \Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p2} \Rightarrow \mathsf{e2} : \Phi_2\ \mathsf{R}\\ \Phi_1\ \mathsf{A} \equiv \Phi_2\ \mathsf{A}\\ \rightarrow\\ \mathit{dimap}_{\Phi 1}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Theta\ (\mathsf{e1}) = \mathit{dimap}_{\Phi 2}\ \mathbf{rep}\ \mathbf{abs}\ \theta_\Theta\ (\mathsf{e2})\end{array}}{\mathsf{p1} \rightsquigarrow \mathsf{p2}}$$

Thus is we can proof the entire top side of the implication, we know that the desired law holds. $\Phi_1\ \mathsf{A} \equiv \Phi_2\ \mathsf{A}$ follows from the premises. $\Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p2} \Rightarrow \mathsf{e2} : \Phi_2\ \mathsf{R}$ also follows directly.
To proof $\Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p1} \Rightarrow \mathsf{e1} : \Phi_1\ \mathsf{R}$, we need the following lemma:

$$\begin{array}{l}\Gamma;\Delta \vdash_{mat} \mathsf{p1\ @\ e1} : \Phi_1\ \mathsf{R} \Rightarrow \mathsf{S}\\ \quad\rightarrow\\ \Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p1} \Rightarrow \mathsf{e1} : \Phi_1\ \mathsf{R}\end{array}$$

Eg. Mathing and then applying yields the same term. This is easy to see from the symmetric nature of matching and applying of patterns.
We can proof the left side of this implication, so we have our proof for $\Gamma;\Delta \vdash_{app} \mathsf{S\ @\ p1} \Rightarrow \mathsf{e1} : \Phi_1\ \mathsf{R}$

5 Mechanical Proof

Because a

5.1 STLC object language

```
infixr 6 _⇒_
data Ty : Set where
  ○ : Ty
  _⇒_ : Ty → Ty → Ty
infixl 5 _,_
data Con : Set where
  ϵ : Con
  _,_ : Con → Ty → Con
infix 4 _∋_
data _∋_ : Con → Ty → Set where
  vz : ∀ {Γ σ} → Γ,σ ∋ σ
  vs : ∀ {τ Γ σ} → Γ ∋ σ → Γ,τ ∋ σ


  -- Removing a variable from a context
infixl 5 _-_
_-_ : {σ : Ty} → (Γ : Con) → Γ ∋ σ → Con
ϵ - ()
(Γ,σ) - vz = Γ
(Γ,τ) - (vs x) = (Γ - x),τ


infix 2 _⊢_
infixl 10 _·_
data _⊢_ : Con → Ty → Set where
  var : ∀ {Γ σ} → Γ ∋ σ → Γ ⊢ σ
  Λ : ∀ {Γ σ τ} → Γ,σ ⊢ τ → Γ ⊢ σ ⇒ τ
  _·_ : ∀ {Γ σ τ} → Γ ⊢ σ ⇒ τ → Γ ⊢ σ → Γ ⊢ τ


wkTm : ∀ {σ Γ τ} → (x : Γ ∋ σ) → Γ - x ⊢ τ → Γ ⊢ τ
wkTm x (var v) = var (wkv x v)
wkTm x (Λ t) = Λ (wkTm (vs x) t)
wkTm x (t₁ · t₂) = wkTm x t₁ · wkTm x t₂
weaken : ∀ {Γ τ σ} → Γ ⊢ τ → Γ,σ ⊢ τ
weaken t = wkTm vz t
```

```
infix 1 _βη-≡_
data _βη-≡_ {Γ : Con} : {σ : Ty} → Tm Γ σ → Tm Γ σ → Set where
  brefl : ∀ {σ} → {t : Tm Γ σ} → t βη-≡ t
  bsym : ∀ {σ} → {t₁ t₂ : Tm Γ σ} → t₁ βη-≡ t₂ → t₂ βη-≡ t₁
  btrans : ∀ {σ} → {t₁ t₂ t₃ : Tm Γ σ} → t₁ βη-≡ t₂ → t₂ βη-≡ t₃ → t₁ βη-≡ t₃
  congΛ : ∀ {σ τ} → {t₁ t₂ : Tm (Γ,σ) τ} → (t₁ βη-≡ t₂) → Λ t₁ βη-≡ Λ t₂
  congApp : ∀ {σ τ} → {t₁ t₂ : Tm Γ (σ ⇒ τ)} → {u₁ u₂ : Tm Γ σ} → t₁ βη-≡ t₂ → u₁ βη-≡ u₂ → app
  β : ∀ {σ τ} → {t : Tm (Γ,σ) τ} → {u : Tm Γ σ} → app (Λ t) u βη-≡ (t / θ vz u)
  eta : ∀ {σ τ} → {t : Tm Γ (σ ⇒ τ)} → Λ (app (wkTm vz t) (var vz)) βη-≡ t
```

5.2 Tools for equational reasoning

5.2.1 Dealing with free variables

```
up : ∀ {Γ τ} → ε ⊢ τ → Γ ⊢ τ
up {ε} t = t
up {y, y'} t = weaken (up {y} t)
```

5.2.2 Structure traversal

5.2.3 Beta-equivalence tactic

5.3 Formalization

5.4 Proof

6 Conclusion

6.1 Related work

6.2 Future work

6.3 Acknowledgements

Bibliography

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP'07*, 2007.

[CSL07] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting haskell strings. In *In Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, pages 50–64. Springer-Verlag, 2007.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing hindley-milner type inference algorithms. Technical report, 2002.

[Hug86] R J M Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[KA10] Chantal Keller and Thorsten Altenkirch. Normalization by hereditary substitutions. 2010.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: extending fold and unfold to exponential types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 324–333, New York, NY, USA, 1995. ACM.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In *Logics of Programs*, pages 219–224. Springer-Verlag, 1985.