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' strings

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 string concatenation. Hughes' method does not only work for strings, but for lists in general, but in this explanation we will keep to strings for simplicity. To see how this works, first consider the standard implementation of string concatenation:

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

The running time of this function is dependent on the size of its first argument. Now let us analyze what calculations are being performed when executing 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' lists transformation solves this by treating string not as normal string (String) but as functions over strings (String → String). Strings now become continuations of strings, where the continuation represents an unfinished string, for which the tail still has to be filled in. Strings and Hughes' strings can be transformed into each other by the functions **rep** and **abs**.

```
rep_ss :: String → (String → String)
rep_ss ls = (ls ++)
```

```
abs_ss :: (String → String) → String
abs_ss c = c []
```

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

```
s1, s2, s3, s4 :: String
s1 = abs_ss $ rep_ss "aap" ∘ (rep_ss "noot" ∘ rep_ss "mies")
s2 = abs_ss $ (rep_ss "aap" ∘ rep_ss "noot") ∘ rep_ss "mies"
s3 = abs_ss $ rep_ss "aap" ∘ rep_ss "noot" ∘ rep_ss "mies"
s4 = abs_ss $ (\x → x ∘ rep_ss "mies") (rep_ss "aap" ∘ rep_ss "noot")
```

All examples now have the same, optimal running time because the continuation technique avoids building intermediate results: each string 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:

```
op :: [a] → [a]
op = 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. This function can be used to obtain a new item from the list and a modified seed (Yield). When the list is empty the function returns Done. Skip is returned when only the seed is modified. This becomes more clear when looking at the function which converts a stream into a list:

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

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

Conversely we can construct a stream from a list:

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

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

Many operations on lists can now be implemented by modifying the step function instead of traversing the entire stream. In this way intermediate constructions are avoided. An example is the map function:

```
mapS :: (a → b) → Stream a → Stream b
mapS f stream = Stream step' (seed stream)
  step' seed' =
    case step stream seed' of
      Done            → Done
      Skip newseed    → Skip newseed
      Yield a newseed → Yield (f a) newseed
```

Note that the mapS function is not recursive. This is what avoids the intermediate result from being constructed. We can now construct an optimized version of our example:

```
op :: [a] → [a]
op = absfs ∘ mapS f ∘ filterS c ∘ mapS g ∘ repfs
```

The only function constructing a list is **abs**, all other operations are 'fused' together.

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, more optimized type. All functions on the original type are replaced by functions on the optimized type, while maintaining the same overall semantics. The functions **rep** and **abs** are used to perform conversions between both types.

Type and transform systems are a formalization of this transformation pattern. In this section, the core concepts of this formalization will be introduced.

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

1. The source and result program have equal types
2. The source and result program are semantically equivalent

There are some important restrictions and liberties implied by these two properties that should be noted here. First off, property 1 implies that the source program has to be well-typed to be transformed, and that the result program will be well-typed. Also, while property 1 restricts transformations to preserve the type of the entire program, it does not restrict type changes to subterms of the program. In the same manner property 2 requires only the complete programs to be semantically equivalent but says nothing about the semantics of its subterms. This is best illustrated with a few examples of Hughes' strings transformations:

$$\texttt{"aap"} \rightsquigarrow \texttt{"aap"} \tag{2.1}$$
$$\texttt{"app"} \not\rightsquigarrow 42 \tag{2.2}$$
$$\texttt{"app"} \not\rightsquigarrow \texttt{"noot"} \tag{2.3}$$
$$\mathsf{upper}\ \texttt{"aap"} \not\rightsquigarrow \mathbf{rep}_{\mathsf{ss}}\ (\mathbf{abs}_{\mathsf{ss}}\ \mathsf{upper})\ \texttt{"aap"} \tag{2.4}$$
$$\texttt{"app"} +\!\!+ \texttt{"noot"} \not\rightsquigarrow \mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"noot"} \tag{2.5}$$
$$\texttt{"aap"} +\!\!+ \texttt{"noot"} \rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ (\mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"noot"}) \tag{2.6}$$
$$(\lambda\, \mathsf{x}.\ \mathsf{x} +\!\!+ \texttt{"noot"})\ \texttt{"aap"} \not\rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ (\lambda\, \mathsf{x}.\ \mathbf{rep}_{\mathsf{ss}}\ \mathsf{x} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"noot"})\ \texttt{"aap"} \tag{2.7}$$
$$(\lambda\, \mathsf{x}.\ \mathsf{x} +\!\!+ \texttt{"noot"})\ \texttt{"aap"} \rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ ((\lambda\, \mathsf{x}.\ \mathsf{x} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"noot"})\ (\mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"})) \tag{2.8}$$
$$\texttt{"aap"} \rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ (\mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"})) \tag{2.9}$$

Figure 2.1: Transformation examples

The $\rightsquigarrow$ symbol is used to denote a valid transformation from some program to a resulting program, $\not\rightsquigarrow$ represents transformations which do not adhere to the TTS transformation properties. The first two transformations are valid and invalid for obvious reasons. Example 2.3 and 2.4 are examples of when the types of a transformation are valid, but the semantics of the program have changed. Example 2.5 does not preserve the typing property and is thus not valid, the next example shows how this could become valid. Examples 2.7 and 2.8 are another example of wrong and right placement of $\mathbf{rep}_{\mathsf{ss}}$ and $\mathbf{abs}_{\mathsf{ss}}$. The last example 2.9 is correct and illustrates that in general the $\mathbf{abs}_{\mathsf{ss}}$ and $\mathbf{rep}_{\mathsf{ss}}$ function from Hughes' string transformation have the property that $\mathbf{abs} \circ \mathbf{rep}_{\mathsf{ss}} \equiv \mathsf{id}$. This property is key to the construction of the TTS system and will be elaborated upon in the next paragraph.

Restricted type changes Looking at the stream fusion and Hughes' strings examples we see that in both cases only one type is being changed. For Hughes' strings transformation, strings

are changed to string continuations and with stream fusion lists are changed to streams. We will restrict the TTS system to only changing one base type in the source program to one different type in the resulting program. The type within the source program is denoted by $\mathfrak{A}$, the type in the result program by $\mathfrak{R}$. These source and result types are required to form a retract, written as $\mathfrak{A} \lhd \mathfrak{R}$. Two types form a retract when there exists a pair of functions, **rep** $:: \mathfrak{A} \to \mathfrak{R}$ and **abs** $:: \mathfrak{R} \to \mathfrak{A}$, for which **abs** is the left inverse of **rep**, meaning that **abs** $\circ$ **rep** $\equiv$ id. Both Hughes' strings and the stream fusion functions **rep** and **abs** have this property.

Object Language A TTS can be built for many different programming languages. The language a TTS is designed for is called the object language. However, not every language is suitable as object language. The first TTS property requires the object language to be strongly typed. Also, in order to be able to relate the semantics of the source and result of a transformation, the object language should come with an equivalence relation for the semantics of its terms.

One of the most simple languages which is suitable as a TTS object language is the simply typed lambda calculus. We will use this language here to further introduce the core concepts of type and transform system, Chapter **??** handles TTS systems for object languages with more advanced features. The terms and types of the simply typed lambda calculus are of the following form:

$$\mathsf{e} ::= \mathsf{x} \mid \mathsf{c} \mid \mathsf{e}\,\mathsf{e} \mid \backslash \mathsf{x}.\,\mathsf{e}$$
$$\tau ::= \mathsf{T} \mid \tau \to \tau$$

As expressions we have variables, constants, application and abstraction. Types can either be base types or function space. Figure 2.2 gives the well-known typing rules for the simply typed lambda calculus. The type and transform system for the simply typed lambda calculus is called TTS_λ.

The judgement $\Gamma \vdash_\lambda \mathsf{e} : \tau$ can be seen as a 3-way relation between types, terms and type environments. The elements of this relation consist of the valid stlc type assignments, and membership to this relation is determined by the typing rules. This relation is the starting point for defining a type and transform system for the simply typed lambda calculus.

The TTS relation At the heart of each type and transform system there is a TTS relation. A TTS relation contains the valid transformations between source and result terms, together with the typing information for both these terms. A TTS relation can be systematically derived from the typing relation of the underlying object language. For STLC we derive the following relation:

$$\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$$

The resemblance with the original STLC typing judgement is obvious. Instead of one term, the relation now embodies two terms, the source and the result term. The $\mathring{\Gamma}$ and $\mathring{\tau}$ constructions are similar to normal types and contexts, except that they type the source and the result term simultaneously. Any variation in types between the two terms, is expressed in those constructions. We call such a modified type a typing functor and the context a functor context.

Con	$\dfrac{\mathsf{c} \text{ is a constant of type } \mathsf{T}}{\Gamma \vdash_\lambda \mathsf{c} : \mathsf{T}}$
Var	$\dfrac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash_\lambda \mathsf{x} : \tau}$
Lam	$\dfrac{\Gamma, \mathsf{x} : \tau_\mathsf{a} \vdash \mathsf{e} : \tau_\mathsf{r}}{\Gamma \vdash_\lambda \backslash \mathsf{x}.\, \mathsf{e} : \tau_\mathsf{a} \to \tau_\mathsf{r}}$
App	$\dfrac{\Gamma \vdash_\lambda \mathsf{f} : \tau_\mathsf{a} \to \tau_\mathsf{r} \quad \Gamma \vdash_\lambda \mathsf{e} : \tau_\mathsf{a}}{\Gamma \vdash_\lambda \mathsf{f}\,\mathsf{e} : \tau_\mathsf{r}}$
Judgement	$\boxed{\Gamma \vdash_\lambda \mathsf{e} : \tau}$

Figure 2.2: Typing rules for the simply typed lambda calculus

Typing functor The variation in types in a type and transform system is limited to only changing one base type $\mathfrak{A}$ from the source program into a base type $\mathfrak{R}$ in the transformation result. The typing functor allows such changes, while maintaining the invariant that both source and result terms are well-typed. The way this is achieved, is by extending the normal types of the object language with an 'hole' construction. This hole will represent the locations at which the types have changed during transformation. For STLC, the typing functor and functor context are defined as follows:

$$\mathring{\tau} := \iota \mid \mathsf{T} \mid \mathring{\tau} \to \mathring{\tau}$$
$$\mathring{\Gamma} := \emptyset \mid \mathring{\Gamma}, \mathsf{x} : \mathring{\tau}$$

Along with the typing functor and context we define a function which turns a functor into a base type of the object language. This is done by filling in the hole type with some type argument. This interpretation function for the typing functor and context are defined as follows:

$$\begin{aligned}
&[\![_]\!]__ : \mathring{\tau} \to \tau \to \tau \\
&[\![\tau]\!]_\tau = \tau \\
&[\![\mathsf{T}]\!]_\tau = \mathsf{T} \\
&[\![(\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r})]\!]_\tau = [\![(\mathring{\tau}_\mathsf{a})]\!]_\tau \to [\![(\mathring{\tau}_\mathsf{r})]\!]_\tau
\end{aligned}$$

$$\begin{aligned}
&[\![_]\!]__ : \mathring{\Gamma} \to \tau \to \tau \\
&[\![\emptyset]\!]_\tau = \emptyset \\
&[\![\mathring{\Gamma}, \mathsf{a} : \mathring{\tau}]\!]_\tau = [\![\mathring{\Gamma}]\!]_\tau, \mathsf{a} : [\![\mathring{\tau}]\!]_\tau
\end{aligned}$$

If a typing functor or context contains no holes we call it **complete**. Note that when a typing functor is complete, it is actually just an ordinary type in the object language. The predicate $\mathfrak{C}$

is used to decide completeness of a typing functor or context, such that if $\mathring{\tau}$ is complete, $\mathfrak{C}(\mathring{\tau})$ holds.

We can now construct some well-typed transformation examples for Hughes' strings as members of the TTS relation. The hole type is inserted at places where the type String in the source term is replaced by the type String $\rightarrow$ String in the result term. Thus the hole type reflects a change in types.

$$
\begin{array}{rll}
\emptyset \vdash_\lambda \texttt{"aap"} \rightsquigarrow \texttt{"aap"} & : \mathsf{String} \\
\emptyset \vdash_\lambda \texttt{"aap"} \rightsquigarrow \mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"} & : \iota \\
\emptyset \vdash_\lambda (+\!\!+) \rightsquigarrow \circ & : \iota \rightarrow \iota \rightarrow \iota \\
\emptyset \vdash_\lambda \backslash \mathsf{x}.\mathsf{x} +\!\!+ \texttt{"hoi"} \rightsquigarrow \backslash \mathsf{x}.\mathsf{x} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"hoi"} & : \iota \rightarrow \iota \\
\emptyset, \mathsf{x} : \iota \vdash_\lambda \mathsf{x} +\!\!+ \texttt{"hoi"} \rightsquigarrow \mathsf{x} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"hoi"} & : \iota \\
\emptyset \vdash_\lambda \texttt{"aap"} +\!\!+ \texttt{"hoi"} \rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ (\mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"} \circ \mathbf{rep}_{\mathsf{ss}}\ \texttt{"hoi"}) & : \mathsf{String} \\
\emptyset \vdash_\lambda \texttt{"aap"} \rightsquigarrow \mathbf{abs}_{\mathsf{ss}}\ (\mathbf{rep}_{\mathsf{ss}}\ \texttt{"aap"}) & : \mathsf{String}
\end{array}
$$

Figure 2.3: Transformation examples

Note that the functions involved in transformation ($\mathbf{rep}_{\mathsf{ss}}$, $\mathbf{abs}_{\mathsf{ss}}$, $(+\!\!+)$ and $\circ$) are not members of the functor context. These transformation terms are not introduced by the context but by separate rules in the TTS.

These examples also show that not every member of the typing relation is a valid TTS transformation, because, for a transformation to be a valid TTS transformation, the type of the source term has to be identical to the type of the result term. However, the typing relation explicitly allows changes in types. The elements of the typing relation which are also a valid TTS transformations are those for which the functor and functor context are complete. These contain no holes and thus no type changes. An element of the typing relation which has a complete typing functor and functor context is also called complete.

2.2.1 A TTS for STLC

The elements of a TTS relation are defined using inference rules. Figure 2.4 introduces the basic inference rules for TTS_λ. The first four rules (Tr-Con, Tr-Var, Tr-Lam and Tr-App) are modified versions of the STLC inference rules. An extra result term has been added and types have been changed to functors, just as the TTS relation. These rules do not perform any actual transformation but make sure transformation progresses over parts of the program which do not change. Consequently they are called the propagation rules.

Note that with the introduction of constants in Tr-Con, it is only allowed to introduce a constant with a base type, not a functor type. This base type can be treated as a functor type in the conclusion, because a functor type is an extensions of the base types: each base type is also a valid functor type.

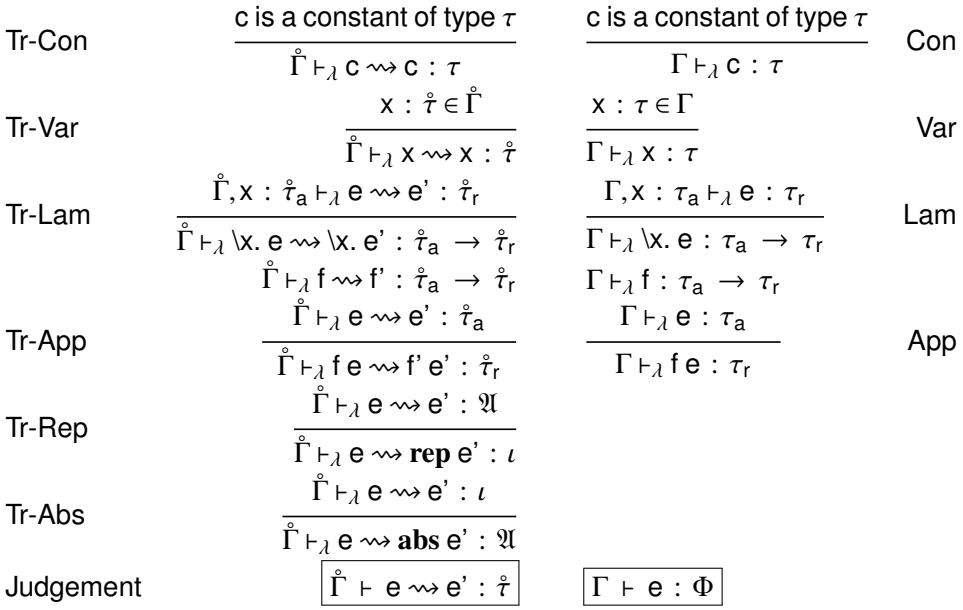

Figure 2.4: Type checking rules for the propagation relation

The rules Tr-Rep and Tr-Abs are called the introduction and elimination rules. These rules are used to convert a term of the source type to a term of the result type and back. Introduction using Tr-Rep introduces a hole in the typing functor, reflecting that the type of the term has changed at the position of this type. Note that the type $\mathfrak{A}$ and the functions **rep** and **abs** may be different for different instantiations of this basic STLC transformation system.

These rules form the basis for all STLC-based transformation systems. For specific transformations such as Hughes' strings this core is extended with extra rules to perform the required transformations.

2.3 A TTS for Hughes' strings

The Hughes' strings transformation constitutes of three core transformations: converting a string to a string continuation (**rep**), converting a string continuation back to a string (**abs**), and replacing string concatenation with function composition. For the STLC version of the Hughes' strings transformation, the TTS_λ base system is instantiated with the parameters specific to Hughes' strings. The source type $\mathfrak{A}$ is instantiated to String, $\mathfrak{R}$ is instantiated to String $\to$ String and the functions **rep** and **abs** in the Tr-Rep and Tr-Abs rules are taken to be the $\mathbf{rep}_{ss}$ and $\mathbf{abs}_{ss}$ functions from Hughes' strings. What is left is adding a rule for transforming string concatenation to function composition. This rule looks as follows:

Tr-Comp
$$\frac{}{\mathring{\Gamma} \vdash_\lambda (+\!\!+) \rightsquigarrow (\circ) : \iota \to \iota \to \iota}$$

The rule Tr-Comp introduces the functor type $\iota \to \iota \to \iota$, reflecting the type changes induced by this transformation.

Example Figure 2.5 shows an example derivation for a program transformation. This particular transformation derivation shows that the term (\x. x ++ "b") "a" can be transformed to the the term **abs** ((\x. x ∘ **rep** "b") (**rep** "a")) using the Hughes' lists transformation system. This example illustrates how both source and result terms are constructed and typed simultaneously when deriving a transformation. The result of the transformation yields a complete element of the TTS relation, and is thus a valid TTS transformation.

2.4 Performing a transformation

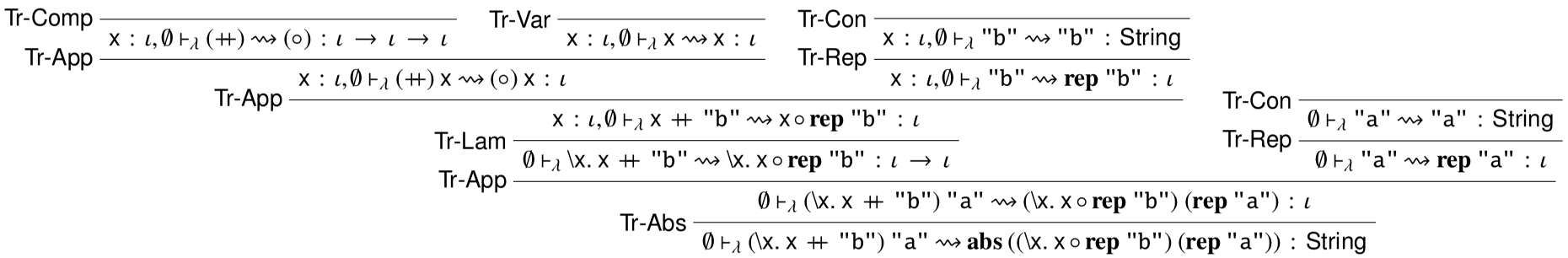

Figure 2.5: An example transformation derivation

3 Proof

The goal of a TTS relation is to enforce the TTS properties on a transformation. This chapter is dedicated to showing that this is indeed the case for the base system of TTS_λ. The main theorem shown here is formulated as follows:

Theorem 1 (Complete TTS_λ transformations ensure the TTS properties)
For all complete elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ from the relation TTS_λ, the transformation $\mathsf{e} \rightsquigarrow \mathsf{e'}$ adheres to the TTS properties.

This theorem is proven by showing that both TTS properties hold. The first section will show the first TTS property involving typing and type equality, the second section will show the second TTS property: transformation should be semantics preserving.

3.1 TTS_λ typing properties

The first TTS property implies that a TTS transformation should only allow correctly typed source and result programs. In other words, a TTS relation should be sound with respect to the underlying object language. This is formalized in the following way:

Lemma 1 (TTS_λ relation preserves typing)
The TTS_λ relation is said to be type preserving if for all elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ of the transformation relation we have that $[\![\mathring{\Gamma}]\!]_{\mathfrak{A}} \vdash_\lambda \mathsf{e} : [\![\mathring{\tau}]\!]_{\mathfrak{A}}$ and $[\![\mathring{\Gamma}]\!]_{\mathfrak{R}} \vdash_\lambda \mathsf{e'} : [\![\mathring{\tau}]\!]_{\mathfrak{R}}$

Proof 1
This proof follows from straightforward induction. The propagation rules of TTS_λ follow identical typing rules as the the underlying STLC object language. The Tr-Rep and Tr-Abs rules preserve the correctness of typing for both source and result terms. Thus, by induction all terms in the (TTS(stlc)) transformation relation have a typing assignment in the underlying STLC language.

It is now easy to formulate and show the first TTS property:

Theorem 2 (Complete TTS_λ transformations ensure the first TTS property)
For all complete elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ both terms e and $\mathsf{e'}$ have a valid typing derivation in STLC under the same environment Γ and τ.

Proof 2 (Lemma 1 and completeness imply Theorem ??)
From lemma 1 follows that for some element in the transformation relation $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$, we know that $[\![\mathring{\Gamma}]\!]_{\mathfrak{A}} \vdash_\lambda \mathsf{e} : [\![\mathring{\tau}]\!]_{\mathfrak{A}}$ and $[\![\mathring{\Gamma}]\!]_{\mathfrak{R}} \vdash_\lambda \mathsf{e'} : [\![\mathring{\tau}]\!]_{\mathfrak{R}}$ are valid typing derivations for e and $\mathsf{e'}$. From the definition of completeness follows that $[\![\mathring{\tau}]\!]_{\mathfrak{A}} = [\![\mathring{\tau}]\!]_{\mathfrak{R}}$ and $[\![\mathring{\Gamma}]\!]_{\mathfrak{A}} = [\![\mathring{\Gamma}]\!]_{\mathfrak{R}}$. Thus, e and $\mathsf{e'}$ have a valid typing assignment under the same type and type environment in STLC.

3.2 TTS_λ semantic properties

The second TTS property dictates that after transformation the semantics of the source and result program are identical. This is formulated by the following Theorem:

Theorem 3 (Complete TTS_λ transformations ensure the second TTS property)
For all complete elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ the equivalence $\mathsf{e} \equiv_{\beta\eta} \mathsf{e'}$ holds.

This is a much stronger property than mere type equivalence and is the cause of most of the restrictions that are laid upon the TTS_λ system.

Inductive formulation The type soundness Lemma 1 is easy to prove using induction because the TTS follows the structure of the typing rules. To formulate and prove an inductive hypothesis about the semantics of the typed terms is a bit more involved. What is needed is a way to connect the types in the typing rules with the semantics of the terms which are being typed. One way to relate terms and types is with the use of functors.

3.2.1 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 such a type-level functor is a term-level functor which lifts functions on the type parameter to functions on the functor:

$$\begin{aligned}&\mathsf{type}\ \Phi\ \mathsf{a}\\ &fmap :: (\mathsf{a} \to \mathsf{b}) \to \Phi\ \mathsf{a} \to \Phi\ \mathsf{b}\end{aligned}$$

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

$$\begin{aligned}&fmap\ \mathsf{id} = \mathsf{id}\\ &fmap\ \mathsf{g} \circ fmap\ \mathsf{f} = fmap\ (\mathsf{g} \circ \mathsf{f})\end{aligned}$$

A list is an example of a Functor in Haskell:

$$\mathbf{data}\ \mathsf{List\ a} = \mathsf{Nil} \mid \mathsf{Cons\ a\ (List\ a)}$$

$$\begin{aligned}&fmap_{\mathsf{list}} :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{List\ a} \to \mathsf{List\ b}\\ &fmap_{\mathsf{list}}\ \_\ \mathsf{Nil} = \mathsf{Nil}\\ &fmap_{\mathsf{list}}\ \mathsf{f}\ (\mathsf{Cons\ a\ l}) = \mathsf{Cons\ (f\ a)}\ (fmap_{\mathsf{list}}\ \mathsf{f\ l})\end{aligned}$$

What makes functors special, is that an implementation for the term level function $fmap$ can be unambiguously constructed from the functor type. In other words: from a given type a term can be derived which adheres to the functor properties. This is the way in which functors connect the term and type world. The rest of this section is dedicated to showing how such a term-level functor can be constructed for the typing functor $\mathring{\tau}$.

This does not work for any type, however. For normal functors, a term-level functor can only be constructed when the argument type occurs in covariant (result) positions within the datatype.

This means $fmap$ can be constructed for all polynomial types (datatypes without functions), but not for all datatypes containing functions. In the following case a functor can be constructed, because the functor parameter is in a covariant position:

```
type IntF a = Int → a
fmap_IntF :: (a → b) → IntF a → IntF b
fmap_IntF f intf = f ∘ intf
```

When the type parameter occurs at a covariant (or argument) position, a functor can not be constructed anymore:

```
type Func a = a → a
fmap_Func :: (a → b) → Func a → Func b
fmap_Func f func = f ∘ func ∘ ?
```

At the question mark a function is needed of the type `b → a` to convert the argument of type `b` to an argument of type `a`. The function argument is of type `a → b`, so this does not work. The typing functor $\mathring{\tau}$ includes a function space constructor, so to construct a term-level functor for the typing functor, something more powerful than a normal functor is needed.

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

```
dimap_Φ :: (b → a) → (a → b) → Φ a → Φ b
```

The Func example can now be completed with the use of a *dimap*.

```
dimap_Func :: (b → a) → (a → b) → Func a → Func b
dimap_Func g f func = f ∘ func ∘ g
```

The functor laws also generalize to the extended functor *dimap*:

```
dimap_Φ id id = id
dimap_Φ g1 g2 ∘ dimap_Φ f1 f2 = dimap_Φ (f1 ∘ g1) (g2 ∘ f2)
```

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

Based on this work, a term-level *dimap* can be defined for the typing functor $\mathring{\tau}$ using the following functor-indexed function:

```
dimap_τ̊ :: (a → b) → (b → a) → τ̊_b → τ̊_a
dimap_ι con cov x = cov x
dimap_T con cov x = x
dimap_τ̊a → τ̊r con cov f = dimap_τ̊r con cov ∘ f ∘ dimap_τ̊a con cov
```

At the hole position the covariant function is applied to manipulate the hole value. For constant types `T` the *dimap* is just the identity function. Most interesting is the case for function space, which recursively transforms the argument and result of the function, but with the argument functions `con` and `cov` switched for the contra-variant call. Switching these functions 'flips' the type signature of the function from $\mathring{\tau}_b \to \mathring{\tau}_a$ to $\mathring{\tau}_a \to \mathring{\tau}_b$.

Relating semantics To prove Theorem 3 we first prove a more general Lemma which also includes all incomplete terms. Note that for incomplete terms the types may not be equal and thus the semantics between the source and result term may be different. This Lemma makes use of *dimap* to relate the semantics of the result program to the source program and is formalized as follows:

Lemma 2 ($TTS_{\vdash_\lambda}$ elements have related semantics)
For all elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ the following equivalence holds:

$$dimap_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}}\ \mathsf{e'}) \equiv_{\beta\eta} \mathsf{e}$$

The difference between the semantics of e and $\mathsf{e'}$ is represented by $dimap_{\mathring{\tau}}$ **rep abs**, which is based on the difference in types (embodied in the typing functor). The **abs** and **rep** functions come form the retraction pair that the $TTS_{\vdash_\lambda}$ is instantiated with (see Section 2.2). The function $\theta_{\mathring{\Gamma}}$ substitutes all free variables in $\mathsf{e'}$ based on environment $\mathring{\Gamma}$, this is defined by the following function:

$$\begin{aligned}
&\theta_{\mathring{\Gamma}} :: [\![\mathring{\Gamma}]\!]_{\Re} \vdash_\lambda \mathsf{e} : \tau \rightarrow [\![\mathring{\Gamma}]\!]_{\mathfrak{U}} \vdash_\lambda \mathsf{e'} : \tau \\
&\theta_{\emptyset} \quad = \mathsf{id} \\
&\theta_{\mathring{\Gamma},\mathsf{x} : \mathring{\tau}} \ = \theta_{\mathring{\Gamma}} \circ [\mathsf{x} \mapsto dimap_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x}]
\end{aligned}$$

This substitution is needed to reason about the semantics of unbound variables and make the equivalence law of Lemma 2 type correct. This more general Lemma implies the original Theorem 3. This is easily shown by establishing the following two properties about complete typing functors and complete functor contexts:

Lemma 3 (*dimap* for a complete functor yields the identity function)
For all complete typing functors $\mathring{\tau}$ the equivalence $dimap_{\mathring{\tau}}\ \mathsf{con}\ \mathsf{cov} \equiv_{\beta\eta} \mathsf{id}$ holds for all functions con and cov.

Proof 3
This follows by induction on the typing functor $\mathring{\tau}$. The hole Id is non-existent because $\mathring{\tau}$ is complete. The case for base types T yields the identity function for *dimap* and thus holds. For function space the induction hypothesis gives the identity for the recursive calls in *dimap* and the composition of two identity functions is $\beta\eta$ equivalent to the identity function itself.

Lemma 4 (θ for a complete functor context yields the identity function)
For all complete functor contexts $\mathring{\Gamma}$ the equivalence $\theta_{\mathring{\tau}} \equiv_{\beta\eta} \mathsf{id}$ holds.

Proof 4
This follows from induction on $\mathring{\Gamma}$. The empty case is trivial. The extension case is easily shown using equational reasoning:

$$\begin{aligned}
&\theta_{\mathring{\Gamma},\mathsf{x} : \mathring{\tau}} \\
&\quad \equiv_{\beta\eta} \{\mathsf{Expand\ definition}\} \\
&\quad\ \ \theta_{\mathring{\Gamma}} \circ [\mathsf{x} \mapsto dimap_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x}] \\
&\quad \equiv_{\beta\eta} \{\mathsf{Induction\ hypothesis}\}
\end{aligned}$$

$\mathsf{id} \circ [\mathsf{x} \mapsto \mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x}]$
$\equiv_{\beta\eta}$ {Lemma|~\ref{lemma:dimap-complete} |}
$\mathsf{id} \circ [\mathsf{x} \mapsto \mathsf{x}]$
$\equiv_{\beta\eta}$ {Identity substitution}
$\mathsf{id} \circ \mathsf{id}$
$\equiv_{\beta\eta}$ {Identity composition}
id

Proof 5 (Lemma 2 implies Theorem 3)
This proof follows from the assumption of Lemma 2 and the completeness of the typing functor $\mathring{\tau}$ and typing context $\mathring{\Gamma}$.

e
$\equiv_{\beta\eta}$ {Assumption Lemma 2}
$\mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}}\ \mathsf{e'})$
$\equiv_{\beta\eta}$ {Lemma 3}
$\mathsf{id}\ \theta_{\mathring{\Gamma}}\ \mathsf{e'}$
$\equiv_{\beta\eta}$ {Def id}
$\theta_{\mathring{\Gamma}}\ \mathsf{e'}$
$\equiv_{\beta\eta}$ {Lemma 4}
$\mathsf{id}\ \mathsf{e'}$
$\equiv_{\beta\eta}$ {Def id}
e'

Before being able to prove Lemma 2, an auxiliary lemma is needed about the behavior of the retraction pair $\mathbf{rep} :: \mathfrak{A} \to \mathfrak{R}$ and $\mathbf{abs} :: \mathfrak{R} \to \mathfrak{A}$. The fact that $\mathfrak{A}$ and $\mathfrak{R}$ form a retract $\mathfrak{A} \lhd \mathfrak{R}$ assures the equivalence $\mathbf{abs}\ (\mathbf{rep}\ \mathsf{x}) \equiv_{\beta\eta} \mathsf{x}$. For values x produced by a TTS_λ transformation this law should also hold the other way around: $\mathbf{rep}\ (\mathbf{abs}\ \mathsf{x}) \equiv_{\beta\eta} \mathsf{x}$. A pair of functions which have both these property are called on isomorphism for the isomorphic types $\mathfrak{A}$ and $\mathfrak{R}$, denoted by $\mathfrak{A} \cong \mathfrak{R}$. This yields the following lemma:

Lemma 5 (A retract behaves as an isomorphism in TTS_λ)
For all elements of the TTS_λ relation $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{Id}$ the equivalence $\mathbf{rep}\ (\mathbf{abs}\ \mathsf{e'}) \equiv_{\beta\eta} \mathsf{e'}$ holds.

This lemma is proven by first showing a slightly different statement about what terms e' can evaluate to. Normalization for the simply typed lambda calculus is denoted by the function $\downarrow$

Lemma 6 (ι evaluates to rep)
For all elements of the TTS_λ relation $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{Id}$ the equivalence $\mathsf{e'}\downarrow \equiv_{\beta\eta} \mathbf{rep}\ \mathsf{x}$ holds for some x.

69 116 40

Proof 6
This follows easily

What is left is showing that Lemma 2 holds for the base $TTS_{\vdash_\lambda}$ relation. This will be proven by induction over the typing rules of TTS_λ.

Proof 7
Proving by induction over the typing rules means showing that for each typing rule, Lemma 2 holds for the conclusion of the typing rule assuming that the lemma holds for the premises of the typing rule. Each typing rule will be handled separately:

Tr-Con

$$\frac{\mathsf{c} \text{ is a constant of type } \tau}{\mathring{\Gamma} \vdash_\lambda \mathsf{c} \rightsquigarrow \mathsf{c} : \tau}$$

The constant rule follows easily because the the *dimap* of a base type (complete type) is the identity function and substitution over a constant equals to the identity substitution.

$$\begin{aligned}
&\mathit{dimap}_\tau \ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}}\ \mathsf{c}) \\
&\quad \equiv_{\beta\eta} \{\text{Lemma 3}\} \\
&\quad\ \ \theta_{\mathring{\Gamma}}\ \mathsf{c} \\
&\quad \equiv_{\beta\eta} \{\text{Substitution over a constant}\} \\
&\mathsf{c}
\end{aligned}$$

Tr-Var

$$\frac{\mathsf{x} : \mathring{\tau} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash_\lambda \mathsf{x} \rightsquigarrow \mathsf{x} : \mathring{\tau}}$$

The proof for the Tr-Var rule follows from the fact that the substitution $\theta_{\mathring{\Gamma}}$ and the conversion term $\mathit{dimap}_{\mathring{\tau}}$ **rep** **abs** evaluate to the identity.

$$\begin{aligned}
&\mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}}\ \mathsf{x}) \\
&\quad \equiv_{\beta\eta} \{\mathsf{x} : \mathring{\tau} \text{ is an element of } \mathring{\Gamma}\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}/\mathsf{x},\mathsf{x}:\mathring{\tau}}\ \mathsf{x}) \\
&\quad \equiv_{\beta\eta} \{\text{Definition } \theta\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}/\mathsf{x}} \circ [\mathsf{x} \mapsto \mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x}]\ \$\ \mathsf{x}) \\
&\quad \equiv_{\beta\eta} \{\text{Definition composition}\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}/\mathsf{x}}\ \mathsf{x}\ [\mathsf{x} \mapsto \mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x}]) \\
&\quad \equiv_{\beta\eta} \{\text{Apply substitution}\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs}\ (\theta_{\mathring{\Gamma}/\mathsf{x}}\ (\mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \mathsf{x})) \\
&\quad \equiv_{\beta\eta} \{\text{Definition composition}\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs} \circ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \$\ \theta_{\mathring{\Gamma}/\mathsf{x}}\ \mathsf{x} \\
&\quad \equiv_{\beta\eta} \{\text{No substitution for x}\} \\
&\quad\ \ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{rep}\ \mathbf{abs} \circ \mathit{dimap}_{\mathring{\tau}}\ \mathbf{abs}\ \mathbf{rep}\ \$\ \mathsf{x} \\
&\quad \equiv_{\beta\eta} \{\text{Functor composition}\}
\end{aligned}$$

$dimap_{\mathring{\tau}}$ (**abs** ∘ **rep**) (**abs** ∘ **rep**) x
≡$_{\beta\eta}$ {Retraction identity}
$dimap_{\mathring{\tau}}$ id id x
≡$_{\beta\eta}$ {Functor identity}
x

Tr-Lam

$$\frac{\mathring{\Gamma}, \mathsf{x} : \mathring{\tau}_\mathsf{a} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}_\mathsf{r}}{\mathring{\Gamma} \vdash_\lambda \backslash\mathsf{x}.\ \mathsf{e} \rightsquigarrow \backslash\mathsf{x}.\ \mathsf{e'} : \mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}$$

The Tr-Lam rule is proven by showing that taking a variable x from the environment and abstracting over it, also holds for the induction hypothesis. The induction hypothesis gives the following premisse to work with:

$dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}, \mathsf{x} : \mathring{\tau}_\mathsf{a}}$ e') ≡$_{\beta\eta}$ e

$dimap_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ (\x. e'))
≡$_{\beta\eta}$ {Commute capture-avoiding substitution over lambda}
$dimap_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}$ **rep abs** (\x. $\theta_{\mathring{\Gamma}/\mathsf{x}}$ e')
≡$_{\beta\eta}$ {Definition $dimap$}
$dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ∘ (\x. $\theta_{\mathring{\Gamma}/\mathsf{x}}$ e') ∘ $dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep**
≡$_{\beta\eta}$ {Eta expansion}
\x. $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ∘ (\x. $\theta_{\mathring{\Gamma}/\mathsf{x}}$ e') ∘ $dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** $ x
≡$_{\beta\eta}$ {Definition (∘)}
\x. ($dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ((\x. $\theta_{\mathring{\Gamma}/\mathsf{x}}$ e') ($dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** x))
≡$_{\beta\eta}$ {Evaluation}
\x. ($dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}/\mathsf{x}}$ e' [x ↦ $dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** x])
≡$_{\beta\eta}$ {Definition composition}
\x. ($dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}/\mathsf{x}}$ ∘ [x ↦ $dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** x] $ e')
≡$_{\beta\eta}$ {Definition θ}
\x. $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}, \mathsf{x} : \mathring{\tau}_\mathsf{a}}$ e')
≡$_{\beta\eta}$ {Premise}
\x. e

Tr-App

$$\frac{\mathring{\Gamma} \vdash_\lambda \mathsf{f} \rightsquigarrow \mathsf{f'} : \mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r} \quad \mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}_\mathsf{a}}{\mathring{\Gamma} \vdash_\lambda \mathsf{f}\ \mathsf{e} \rightsquigarrow \mathsf{f'}\ \mathsf{e'} : \mathring{\tau}_\mathsf{r}}$$

$dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ (f' e'))
  $\equiv_{\beta\eta}$ {Distribute substitution}
    $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ f' ($\theta_{\mathring{\Gamma}}$ e'))
  $\equiv_{\beta\eta}$ {$dimap$ identity}
    $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ f' ($dimap_{\mathring{\tau}_\mathsf{a}}$ id id ($\theta_{\mathring{\Gamma}}$ e')))
  $\equiv_{\beta\eta}$ {Lemma 5}
    $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ f' ($dimap_{\mathring{\tau}_\mathsf{a}}$ (**rep** $\circ$ **abs**) (**rep** $\circ$ **abs**) ($\theta_{\mathring{\Gamma}}$ e')))
  $\equiv_{\beta\eta}$ {Functor composition}
    $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ f' ($dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** $\circ$ $dimap_{\mathring{\tau}_\mathsf{a}}$ **rep abs** \$ $\theta_{\mathring{\Gamma}}$ e'))
  $\equiv_{\beta\eta}$ {Definition ($\circ$)}
    $dimap_{\mathring{\tau}_\mathsf{r}}$ **rep abs** $\circ$ ($\theta_{\mathring{\Gamma}}$ f') $\circ$ $dimap_{\mathring{\tau}_\mathsf{a}}$ **abs rep** \$ $dimap_{\mathring{\tau}_\mathsf{a}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ e'))
  $\equiv_{\beta\eta}$ {Definition $dimap_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}$}
    $dimap_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ f') ($dimap_{\mathring{\tau}_\mathsf{a}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ e')))
  $\equiv_{\beta\eta}$ {Premises}
f e

Tr-Rep

$$\frac{\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathfrak{U}}{\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathbf{rep}\ \mathsf{e'} : \iota}$$

$dimap_\iota$ **rep abs** ($\theta_{\mathring{\Gamma}}$ (**rep** e'))
  $\equiv_{\beta\eta}$ {Definition $dimap_\iota$}
    abs ($\theta_{\mathring{\Gamma}}$ (**rep** e'))
  $\equiv_{\beta\eta}$ {Commute substitution}
    abs (**rep** ($\theta_{\mathring{\Gamma}}$ e'))
  $\equiv_{\beta\eta}$ {Retraction}
    $\theta_{\mathring{\Gamma}}$ e'
  $\equiv_{\beta\eta}$ {Definition $dimap_{\mathfrak{U}}$}
    $dimap_{\mathfrak{U}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ e')
  $\equiv_{\beta\eta}$ {Premise}
e

Tr-Abs

$$\frac{\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \iota}{\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathbf{abs}\ \mathsf{e'} : \mathfrak{U}}$$

$dimap_{\mathfrak{U}}$ **rep abs** ($\theta_{\mathring{\Gamma}}$ (**abs** e'))
  $\equiv_{\beta\eta}$ {Definition $dimap_{\mathfrak{U}}$}
    $\theta_{\mathring{\Gamma}}$ (**abs** e')

$\equiv_{\beta\eta}$ {Commute substitution}
abs ($\theta_{\hat{\Gamma}}$ e')
$\equiv_{\beta\eta}$ {Definition ($dimap$ (ι))}
$dimap_{Id}$ **rep abs** ($\theta_{\hat{\Gamma}}$ e')
$\equiv_{\beta\eta}$ {Premise}
e

4 Mechanical Proof

5 Extensions to the TTS system

5.1 A TTS for Stream fusion

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.

[Keu11] Steven Keuchel. Generic programming with binders and scope, 2011.

[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.