A Type-Changing, Semantics-Preserving Program Transformation System

Bram Schuur *

February 11, 2013

*Msc. Thesis under supervision of Johan Jeuring and Sean Leather

Program transformations play a central role in the construction of compilers. Before a source program comes out as machine readable instructions, it has gone through a lot of optimization and simplification steps, preferably without changing the behaviour of the original program.

Traditionally, these transformations work by manipulating the syntactic constructs of a language, and do not alter the types of a program, or are applied to untyped sources. However, in recent years there has been an increasing interest in transformations in type-changing program transformations on typed sources: transformations in which it is possible to change the types together with the terms.

This work introduces an extensible transformation system based on the simply typed lambda calculus, which allows type-changing transformations between terms. The system ensures that a transformation preserves the semantics of the source term, independent of the concrete datatypes and evaluation strategy of the underlying lambda calculus. This statement is formally proven using logical relations and mechanically verified in the dependently typed programming language Agda. Central to the transformation system is the concept of typing functors, a construction which types the source and result of a transformation simultaneously, while allowing changes in types to occur.

Contents

1 Introduction

Program transformation is a central aspect of computer science, most importantly in the area of compiler construction. One expects from a compiler to transform human readable code into highly optimized machine code, while maintaining the expected behaviour of the program. During compilation, a compiler makes use of a wide range of program transformations to reach this goal. A program is simplified, variables are renamed and optimizations are applied while the program may be transformed into one or multiple intermediate languages.

Closely related areas in which program transformations play a role are program refactoring and program migration. Program transformations for refactoring and migration resemble those for compilation in that the behaviour of the resulting program should remain unchanged, but the intention of the transformations are different: refactoring is done to make a program better readable for humans and migration is done to adapt a program to a changed environment.

Program transformations A program transformation works by inspecting and manipulating the terms of a language to generate the desired result. However, programming languages such as Haskell or Java do not only consist of terms, but of types as well. Designing program transformations for such languages leaves the designer with a choice: either throw away the typing information and apply transformations to untyped terms, or somehow leave the types of the terms intact during transformation. Throwing away the typing information is sometimes unavoidable, for example when transforming typed Java to untyped assembler code. However, the transformations used for optimizing, refactoring and migration usually produce terms in the same language as their input language. For such transformations the choice between typed and untyped transformation is more difficult. Doing an untyped transformation gives a lot of flexibility, but can more easily result in untypeable terms. Furthermore, the types of a language are also a valuable source of information during when conducting program transformations.

For transformations on typed languages, on the other hand, the types themselves can be restrictive. One possibility is to leave the types of the source program intact during transformation and only substitute equally typed terms. This works, but there is a class of transformations which require changes in the types next to changes in the terms. The following is an example of such a transformation in the language Haskell, which replaces all occurrences of integral numbers by floating point numbers to reduce the rounding errors in a program:

```
percent :: Int → Int → Int
percent v v' = 100 * (v' `div` v)

percents :: [Int] → [Int]
percents vs =
  map (percent (sum vs)) vs
```

```
percent :: Float → Float → Float
percent v v' = 100 * (v' / v)

percents :: [Float] → [Float]
percents vs =
  map (percent (sum vs)) vs
```

Such a transformation requires a transformation of both the terms and the types. The types are changed from **Int**s to **Float**s, and the `div` operator is replaced by /. The type changes are not just local but propagate throughout the program: a program-wide type transformation. In recent years several of these type-changing program transformations have been developed, such as the worker/wrapper transformation by Gill and Hutton [GH09] or a continuation passing style transformation by Ahmed and Blume [AB11]. They both show that their transformations are equivalence preserving and type-preserving, two crucial properties for typed program transformations.

Transformation systems Creating such a program transformation is time-consuming and error-prone. It is time-consuming because a transformation has to either be built into an existing compiler, which is usually a complicated piece of software to work with, or one has to create a custom infrastructure to process terms of the language, which means dealing with detailed implementation issues. Because modern programming languages have many constructs which interact in many subtle ways, an error is easily made.

Because compilers are such critical parts of modern software systems, it is good practice to provide a proof of the fact that a program transformation works as expected. An error in a compiler can cause a lot of trouble. However, proving a program transformation correct is not easy and again time-consuming.

A way to counter these issues, is to look for possible abstractions. Instead of developing individual program transformations, look for parts that can be generalized and develop a generic transformation system in which transformations can be constructed. This makes developing individual transformations faster and easier, resulting in less errors.

1.1 Related Work

Several such systems for creating type-aware program transformation have been developed and implemented before in various forms.

- The Haskel GHC compiler allows user-defined transformation rules to transform terms, such as the following for map fusion:

  ```
  {-# RULES "map/map" forall f g xs. map f (map g xs) = map (f . g) xs #-}
  ```

  These rewrite rules are used by the compiler to simplify and optimize during compilation. The rules are type-checked using the internal type checker, which quickly catches typos

and simple errors. Although the individual rules cannot change the types in a program themselves, combinations of these rules can have much the same effect. This is more thoroughly discussed in Section 2.1.2

- Cunha and Visser [CV07] [COV06] developed a framework for type-changing program transformation as an EDSL in Haskell. Their system works on a point-free representation of terms to avoid having to deal with binders.

- Recently Erwig and Ren [ER07] proposed a generic transformation system for type-changing program transformation. In their approach they define an update calculus for program updates which can deal with both type-changes and scoping changes. They show that updates performed by their generic transformation system will result in well-typed programs and that the semantics between the source and result programs are preserved.

1.2 Motivation

Although all these systems facilitate type changes in their own way, none of these systems are suitable for program-wide transformation of types, which is a very real use-case for both program optimization and migration. We call such transformations *type-driven*: transformations in which the ultimate goal is to replace the types in a program. In such a transformation the terms are only changed to make a program work with a new type: term substitution is not a driving factor. We already introduced the transformation of integers to floats as a use-case for a type-driven transformation, but there are many more use-cases:

Migration A situation when program-wide modification of types is desired when migrating between libraries. In Haskell this could be migrating a program which uses Text as a representation for character data, to a representation using ByteStrings. This means changing the Text type inside the program as much as possible, and replacing the functions that work on Text by functions that work on ByteStrings. A similar migration occurs when moving between parsing libraries or when upgrading to a new version of a package.

This use-case was the primary motivation for the work of Erwig and Ren. However, their system has a rather ad-hoc way of specifying rules and type changes are very local, not in a program-wide manner.

Optimization Another use-case for type-changing program transformations is when the representation of a datatype is changed for optimization purposes. Hughes' strings transformation [Hug86] and the Stream fusion transformation [CLS07] are prominent examples of this and will be discussed in more detail in Section 2.1.

Refactoring Sometimes it is desired to change the representation of a datatype because this is more suitable for the programmer. For example, a point in space may be represented by its cartesian coordinates or by polar coordinates. When a user wants to switch between these representations, a transformation could be created to change the representation, and the way in which the program calculates with these representations.

Note that in most of these examples the same result could have been achieved by changing the underlying representation of a datatype instead of changing the program. However, this is not possible when the representation of a type is managed externally, for example when transforming base types or types which are part of a standard library. In these cases the only solution is to change the types in a program.

Also, when one wants to replace a type in a program, it is not always possible to find a suitable term-level replacement for all functions. In this case it is not possible to do a program-wide replacement of a type. For example in the case of the integer to floating point transformations, there is no floating point alternative for bit-shifting an **Int** representation. Thus there is no alternative but to keep the type unchanged and convert the floating point to an integer.

```
magic :: Int → Int → Int
magic a b = a .|. xor b
```

```
magic :: Float → Float → Float
magic a b = round a .|. xor (round b)
```

It is also possible that a type is not fully abstract and it can be inspected using pattern matching. In this case the internals of a type are exposed and one does not get around keeping the original type intact.

Situations like these make it beneficial to have a type-changing transformation system which works directly on program representations and which tries to replace a type as much as possible, but can locally keep the original types intact.

1.3 Contributions

This work introduces a generic formal system for such type-changing program transformations, which we call type-and-transform systems, TTS for short. This system is intended to be a basis for type-changing program-wide transformations and has the following features:

STLC The transformation system is designed for languages based on the simply typed lambda calculus. The core transformations are purely based on STLC, extensions are later discussed.

Generic The system is generic in the types and terms that are transformed. It can be parametrized with specific type and term transformations to obtain a specific program transformation. The type-and-transform system is also independent of the interpretation of the underlying type language.

Proven The system is proven to allow only transformations which yield type correct programs and which yield a semantically equivalent program after transformation. This claim is proven using logical relations and mechanically verified in the dependently type programming language Agda [Nor07], thus contributing to the POPLmark challenge [pop] of verified metatheory in programming languages.

Note that this system does not specify how the transformation system is implemented. The TTS system specifies the typing rules for type-changing program transformations, not how these transformations are actually performed. Section 2.4 discusses possibilities for implementation.

1.4 Overview

This section gives an overview of the upcoming chapters.

Chapter 2 introduces the general concepts and construction of the TTS system. First more extensive motivating examples of type-changing program transformations are introduced in the first two sections. Subsequently the basic type-and-transform system for the simply typed lambda calculus is presented, followed by an example application of the Hughes' strings in Section 2.3. The last section will briefly discuss the possibilities and problems when implementing an actual transformations system.

Chapter 3 establishes the core properties of the type-and-transform system. Section 3.1 and Section 3.2 establish the type soundness and productivity of the type-and-transform system. Section 3.3 introduces logical relations and uses these to prove the semantic correctness of program transformations. The last two sections discuss some of the theoretical background of the transformation system.

Chapter 4 discusses the formalization of the type-and-transform system in the programming language Agda in the first two sections and shows how the transformation system is mechanically proven correct in section 4.3. Section 4.4 introduces an example transformation based on monoids, the last section discusses the content of this chapter.

In Chapter 5 possible extensions of the type-and-transform system are discussed, to make the system ready for real-world languages. The last two chapters discuss future work and conclude.

2 Type and Transform Systems

This chapter introduces the core concepts of type and transform systems. The first section gives two motivating examples of type-changing program transformations which will be used throughout this work.

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 we will use strings for simplicity. To see what problem Hughes' strings solve, 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 depends on the size of its first argument. The problem with this definition becomes clear when analyzing necessary computations 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' 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 $\mathbf{rep}_{ss}$ and $\mathbf{abs}_{ss}$.

```
repss :: String → (String → String)
repss ls = (ls ++)
absss :: (String → String) → String
absss c = c ""
```

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

```
s1 , s2 , s3 , s4 :: String
s1 = absss $ repss "aap" ∘ (repss "noot" ∘ repss "mies")
s2 = absss $ (repss "aap" ∘ repss "noot") ∘ repss "mies"
s3 = absss $ repss "aap" ∘ repss "noot" ∘ repss "mies"
s4 = absss $ (\x → x ∘ repss "mies") (repss "aap" ∘ repss "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]. 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 data structures constructed during evaluation. Consider the following example:

```
op :: (Int → Int) → [Int] → [Int]
op f = map f ∘ filter isEven ∘ map f
```

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 by changing the internal representation of lists to **Stream**s

```
data Step s a =
    Done
  | Yield a s
  | Skip s
data Stream a = ∃ 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:

```
absfs :: Stream a → [a]
absfs 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. It only manipulates the function that works on lists, not the lists themselves. This is what avoids the intermediate result from being constructed. We can now construct an optimized version of our example:

```
op :: (Int → Int) → [Int] → [Int]
op f = absfs ∘ mapS f ∘ filterS isEven ∘ mapS g ∘ repfs
```

The only function constructing a list is $\mathbf{abs}_{fs}$, all other operations are 'fused' together.

Stream fusion in GHC The Haskell GHC compiler incorporates a system which can be used to create such optimizations. This system is based on local, type-checked rewrite rules to rewrite terms which can be optimized. This is done for the Stream Fusion optimization by redefining the basic list functions and adding an optimizing rewrite rule in the following way:

```
map :: (a → b) → [a] → [b]
map f = absfs ∘ mapS f ∘ repfs
```

```
filter :: (a → Bool) → [a] → [a]
filter f = abs_fs ∘ filterS f ∘ rep_fs
  {-# RULES "rep/abs" rep . abs = id #-}
```

The **op** example can now be optimized by the compiler, by first expanding and inlining all definitions and consecutively applying the optimization rewrite rule, getting rid of all abs-rep pairs. Thus this achieves a type-changing transformation by applying a non type-changing rewrite rule. However, this system is limited by how much the compiler can inline and replace the terms. It is primarily a syntactic method. To see where optimization may fail, consider the following example:

```
op :: (Int → Int) → [Int] → [Int]
op f = let applyMap = (\x → map f x)
       in applyMap ∘ filter isEven ∘ applyMap
```

When expanding these definitions, this yields the following term

```
op :: (Int → Int) → [Int] → [Int]
op f = let applyMap = (\x → abs_fs ∘ mapS f ∘ rep_fs $ x)
       in applyMap ∘ abs_fs ∘ filterS isEven ∘ rep_fs ∘ applyMap
```

Without proper inlining, the rep-abs rule can not fire. In this situation the inliner may solve this issue, but there are situations where inlining is not enough to make Stream Fusion work. Allowing direct type changes in the program however, yields the desired result:

```
op :: (Int → Int) → [Int] → [Int]
op f = let applyMap = (\x → mapS f x)
       in abs_fs ∘ applyMap ∘ filterS isEven ∘ applyMap ∘ rep_fs
```

2.2 Core Transformation System

Looking closely at the previous two examples, we can see a transformation pattern emerge. The transformations are *type-directed*: in both examples a single type is changed into another, more optimized type. The term transformations consequently follow the transformation of the types: all functions on the original type are replaced by functions on the optimized type, while maintaining the same overall semantics. In the process of transforming to a new type, the functions **rep** and **abs** have the special role of performing conversions between the original and the new type.

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

2.2.1 Transformation properties

Before developing a transformation system, it useful to formalize some overall properties that we want a transformation system to uphold. Type and transform systems are intended to maintain the following properties for its transformations:

Sound The source and result program have equal types

Equivalence The source and result program are semantically equivalent

Productive A transformation yields a result for all possible inputs

Note that property the soundness does not exclude changes of types within a program. The complete program after transformation should have the same type as the original, but types may have been changed within the subterms. In other words: the resulting types remain the same, but the *typing derivation* may differ. In the same manner the equivalence property requires only the complete programs to be semantically equivalent but says nothing about the semantics of its sub-terms. This is consistent with the example transformations: Locally the types and terms have been changed, but the overall type and semantics have remained the same.

Furthermore, this implies that the source program has to be well-typed to be eligible for transformation. The transformation should also provably produce a type-correct program as a result.

To get a feeling for what could be a valid transformation, consider the following example transformations:

$$\texttt{"aap"} \rightsquigarrow \texttt{"aap"} \tag{2.1}$$

$$\texttt{"aap"} \not\rightsquigarrow 42 \tag{2.2}$$

$$\texttt{"aap"} \not\rightsquigarrow \texttt{"noot"} \tag{2.3}$$

$$\textbf{upper}\ \texttt{"aap"} \not\rightsquigarrow \textbf{rep}_{\text{ss}}\ (\textbf{abs}_{\text{ss}}\ \textbf{upper})\ \texttt{"aap"} \tag{2.4}$$

$$\texttt{"app"} +\!\!+ \texttt{"noot"} \not\rightsquigarrow \textbf{rep}_{\text{ss}}\ \texttt{"aap"} \circ \textbf{rep}_{\text{ss}}\ \texttt{"noot"} \tag{2.5}$$

$$\texttt{"aap"} +\!\!+ \texttt{"noot"} \rightsquigarrow \textbf{abs}_{\text{ss}}\ (\textbf{rep}_{\text{ss}}\ \texttt{"aap"} \circ \textbf{rep}_{\text{ss}}\ \texttt{"noot"}) \tag{2.6}$$

$$(\lambda\, \texttt{x}.\ \texttt{x} +\!\!+ \texttt{"noot"})\ \texttt{"aap"} \rightsquigarrow \textbf{abs}_{\text{ss}}\ ((\lambda\, \texttt{x}.\ \texttt{x} \circ \textbf{rep}_{\text{ss}}\ \texttt{"noot"})\ (\textbf{rep}_{\text{ss}}\ \texttt{"aap"})) \tag{2.7}$$

$$(\lambda\, \texttt{x}.\ \texttt{x} +\!\!+ \texttt{"noot"})\ \texttt{"aap"} \rightsquigarrow \textbf{abs}_{\text{ss}}\ ((\lambda\, \texttt{x}.\ \textbf{rep}_{\text{ss}}\ \texttt{x} \circ \textbf{rep}_{\text{ss}}\ \texttt{"noot"})\ \texttt{"aap"}) \tag{2.8}$$

$$(\lambda\, \texttt{x}.\ \texttt{x} +\!\!+ \texttt{"noot"})\ \texttt{"aap"} \not\rightsquigarrow \textbf{rep}_{\text{ss}}\ (\textbf{abs}_{\text{ss}}\ (\lambda\, \texttt{x}.\ \texttt{x} +\!\!+ \texttt{"noot"}))\ \texttt{"aap"} \tag{2.9}$$

$$\texttt{"aap"} \rightsquigarrow \textbf{abs}_{\text{ss}}\ (\textbf{rep}_{\text{ss}}\ \texttt{"aap"}) \tag{2.10}$$

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 show transformations in which 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 show that there can be multiple valid results for the same source program. The next example shows that the wrong placement of $\textbf{abs}_{\text{ss}}$ and $\textbf{rep}_{\text{ss}}$ can result in a type-correct solution, but with the wrong semantics. I this case, `"noot"` and `"app"` will be reversed! The last example 2.10 is correct

and illustrates that the $\mathbf{abs}_{\mathsf{ss}}$ and $\mathbf{rep}_{\mathsf{ss}}$ function from Hughes' string transformation have the property that $\mathbf{abs} \circ \mathbf{rep}_{\mathsf{ss}} \equiv \mathbf{id}$. 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 $\mathcal{A}$, the type in the result program by $\mathcal{R}$. These source and result types are required to form a *retract*, written as $\mathcal{A} \lhd \mathcal{R}$. Two types form a retract when there exists a pair of functions, $\mathbf{rep} :: \mathcal{A} \to \mathcal{R}$ and $\mathbf{abs} :: \mathcal{R} \to \mathcal{A}$, for which **abs** is the left inverse of **rep**, meaning that $\mathbf{abs} \circ \mathbf{rep} \equiv \mathbf{id}$. Both Hughes' strings and the stream fusion functions **rep** and **abs** have this property.

2.2.2 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. To assure the type soundness transformation property, the object language should have a strong type system. 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 (stlc). This language will be used to further introduce the core concepts of type-and-transform systems. Chapter 5 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:

$$
\begin{aligned}
&\mathsf{e} ::= \mathsf{x} \mid \mathsf{c} \mid \mathsf{e}\,\mathsf{e} \mid \backslash \mathsf{x}.\,\mathsf{e} \\
&\tau ::= \mathsf{T} \mid \tau \to \tau \\
&\Gamma ::= \emptyset \mid \Gamma\,,\,\mathsf{x} : \tau
\end{aligned}
$$

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.

It is not necessary to specify a complete semantics for the simply typed lambda calculus. TTS_λ is valid for all semantics which admit $\beta\eta$-convertibility on terms, as will be shown in chapter 3.

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	$\dfrac{\Gamma, \mathsf{x} : \tau_\mathsf{a} \vdash \mathsf{e} : \tau_\mathsf{r}}{\Gamma \vdash_\lambda \backslash \mathsf{x}.\, \mathsf{e} : \tau_\mathsf{a} \rightarrow \tau_\mathsf{r}}$
App	$\dfrac{\Gamma \vdash_\lambda \mathsf{f} : \tau_\mathsf{a} \rightarrow \tau_\mathsf{r} \quad \Gamma \vdash_\lambda \mathsf{e} : \tau_\mathsf{a}}{\Gamma \vdash_\lambda \mathsf{f}\,\mathsf{e} : \tau_\mathsf{r}}$
Judgement	$\boxed{\Gamma \vdash_\lambda \mathsf{e} : \tau}$

Figure 2.2: Typing rules for the simply typed lambda calculus

The TTS_λ 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. Such a modified type is called a *typing functor* and the context is called a *functor context*. Any variation in types between the source and result terms is embodied in the typing functor and functor context.

Typing functor The variation in types in TTS_λ is limited to only changing one base type $\mathcal{A}$ from the source program into a type $\mathcal{R}$ 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, represented by a ι. 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:

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

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:

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

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

The typing functor and context are an extension of normal types. Thus, normal types naturally give rise to a functor, as is shown by the following lifting functions:

$$
\begin{aligned}
&\tau\uparrow : \mathring{\tau} \\
&\mathsf{T}\uparrow &&= \mathsf{T} \\
&(\tau_\mathsf{a} \to \tau_\mathsf{r})\uparrow &&= \mathring{\tau}_\mathsf{a}\uparrow \to \mathring{\tau}_\mathsf{r}\uparrow
\end{aligned}
$$

$$
\begin{aligned}
&\Gamma\uparrow : \mathring{\Gamma} \\
&\Gamma\uparrow &&= \emptyset \\
&\Gamma\,,\, \mathsf{a} : \tau\uparrow &&= \Gamma\uparrow\,,\, \mathsf{a} : \tau\uparrow
\end{aligned}
$$

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, lifted to a functor.

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** $\to$ **String** in the result term. Thus the hole type reflects a change in types.

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

Figure 2.3: Transformation examples

A member of the TTS_λ typing relation is said to be *complete* when it results in a base type and the typing context is *complete*. Because a complete functor type is just a lifted base type, we

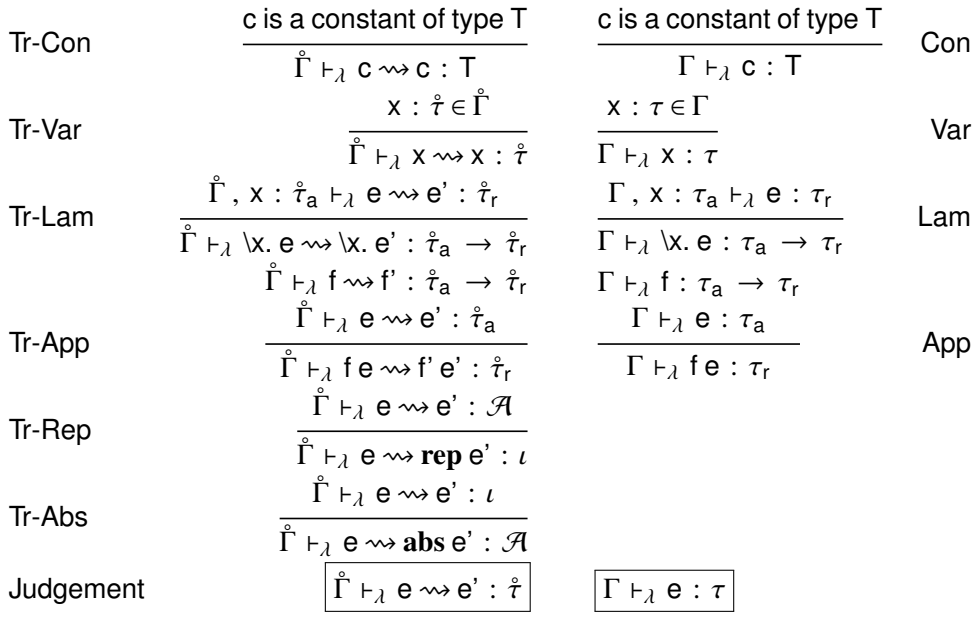

Figure 2.4: Type checking rules for the propagation relation

can write a complete transformation in the following form: $\Gamma \uparrow \vdash_\lambda e \rightsquigarrow e' : T$. Each complete element of the typing relation should preserve the TTS properties. This claim will be shown in chapter 3 for TTS_λ.

2.2.3 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. The existence of these rules makes that this transformation system adheres to the productivity property: there is always an identity transformation rule.

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 $\mathcal{A}$, $\mathcal{R}$ 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 ($\mathbf{rep}_{\mathsf{ss}}$), converting a **String** continuation back to a **String** ($\mathbf{abs}_{\mathsf{ss}}$), 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 $\mathcal{A}$ is instantiated to **String**, $\mathcal{R}$ is instantiated to **String** $\rightarrow$ **String** and the functions **rep** and **abs** in the Tr-Rep and Tr-Abs rules are taken to be the $\mathbf{rep}_{\mathsf{ss}}$ and $\mathbf{abs}_{\mathsf{ss}}$ functions from Hughes' strings. What is left is adding a rule for transforming string concatenation to function composition. This rule looks as follows:

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

The rule Tr-Comp introduces the functor type $\iota \rightarrow \iota \rightarrow \iota$, reflecting the type changes induced by this transformation. The Hughes' strings transformation is abbreviated as TTS_{ss}.

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 term **abs** ((\x. x $\circ$ **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

In the previous sections a basic transformation system is presented which defines what constitutes a valid transformation for TTS_λ. However, this deduction system does not show how to algorithmically perform an actual transformation. For example, the deduction system allows an infinite application of the **rep** and **abs** rule, such as the following:

e $\rightsquigarrow$ **abs** (**rep** (**abs** (**rep** e)))

This is hardly an optimizing program transformation! Even worse, the system allows endless chains of **abs** and **rep** applications, thus allowing infinite deduction trees. The reason that such endless chains can occur is because the rules of TTS_λ are not *syntax directed*. In a normal typing derivation for the simply typed lambda calculus, the derivation tree is uniquely determined by the term that is being typed. However, in the TTS_λ system this is not the case. Multiple results are possible for the same input.

Thus program transformation becomes a form of proof search: Given an input program, try to find a valid TTS_λ derivation. While one can always create an identity transformation (do nothing), this would not result in any improvements. One wants to find a transformation which gives as much optimization as possible, while avoiding unnecessary application of **abs** and **rep**.

How to actually do this is not part of this work, but is part of ongoing research by Sean Leather.

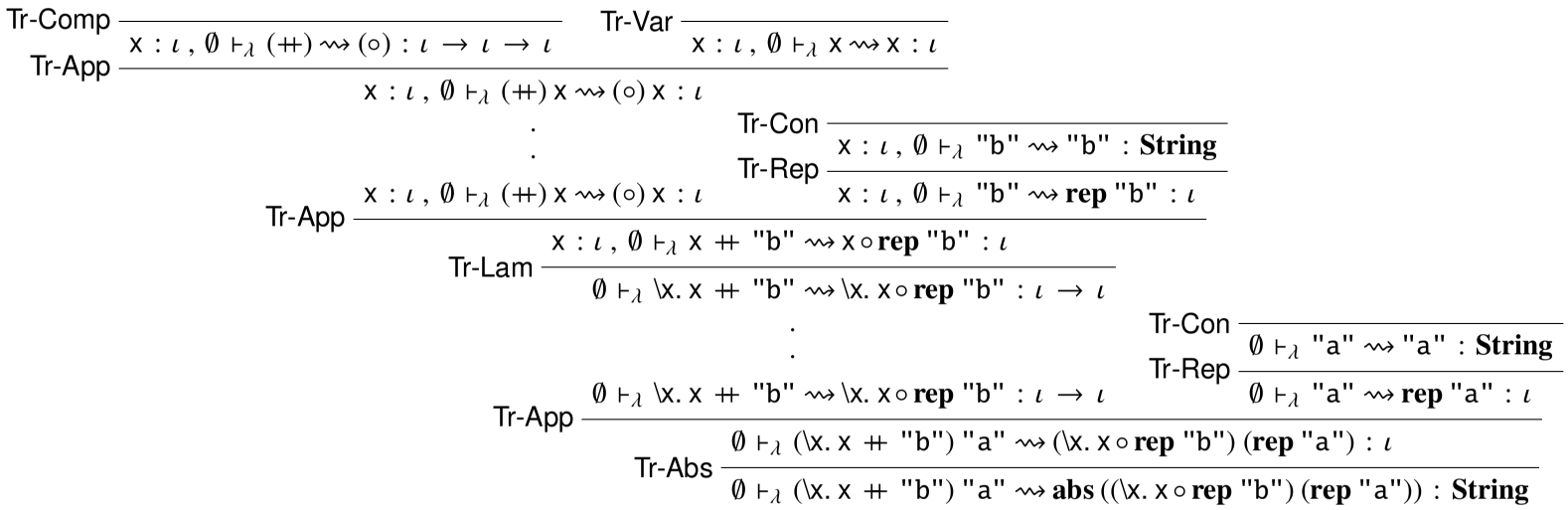

Figure 2.5: An example transformation derivation

3 TTS_λ Preserves the Transformation Properties

The goal of a TTS transformation relation is to enforce the transformation properties as defined at the beginning of section 2.2.1. The transformation system should preserve typing, should produce a resulting program which is equivalent to the source and should always be productive. 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 Transformation Properties)
For all complete elements $\Gamma \uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{T}$ from the relation TTS_λ, the transformation $\mathsf{e} \rightsquigarrow \mathsf{e'}$ adheres to the transformation properties.

The first three section of this chapter show the Productivity, Soundness and Equivalence property for TTS_λ. The last two sections reflect on the theoretical background of the type and transform system.

3.1 TTS_λ is Sound

A complete transformation is defined to be a transformation which is typed with a base type and for which the environment is complete (contains no holes). This means that, as long as TTS_λ does indeed type both the source and result term correctly, a complete transformation will result in two equally typed terms. This type soundness property is formalized by the following lemma:

Lemma 1 (TTS_λ are Correctly Typed)
The TTS_λ relation is sound if for all elements $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathring{\tau}$ of the transformation relation we have that $[\![\mathring{\Gamma}]\!]_{\mathcal{A}} \vdash_\lambda \mathsf{e} : [\![\mathring{\tau}]\!]_{\mathcal{A}}$ and $[\![\mathring{\Gamma}]\!]_{\mathcal{R}} \vdash_\lambda \mathsf{e'} : [\![\mathring{\tau}]\!]_{\mathcal{R}}$ have a valid typing derivation in stlc.

Proof 1
The evidence for this property follows from straightforward induction on the derivations of terms. The propagation rules of TTS_λ follow identical typing rules as the the underlying STLC object language, and thus preserve the typing of the transformed terms. 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_λ transformation relation have a typing assignment in the underlying STLC language.

It is now easy to state more formally that the soundness property holds for TTS_λ:

Theorem 2 (Complete Transformations ensure Soundness)
For all complete elements $\Gamma \uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{T}$ both terms e and $\mathsf{e'}$ have equal type assignments $\Gamma \vdash_\lambda \mathsf{e} : \mathsf{T}$ and $\Gamma \vdash_\lambda \mathsf{e'} : \mathsf{T}$ in the simply typed lambda calculus.

Proof 2 (Type Correctness and Completeness imply Soundness)
From lemma 1 follows that for a complete element in the transformation relation $\Gamma \uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{T}$, we know that here exist valid typing derivations for $[\![\Gamma \cdot \uparrow]\!]_{\mathcal{A}} \vdash_\lambda \mathsf{e} : \mathsf{T}$ and $[\![\Gamma \uparrow]\!]_{\mathcal{R}} \vdash_\lambda \mathsf{e'} : \mathsf{T}$. From the definition of a complete environment simply follows that $[\![\Gamma \uparrow]\!]_\tau \equiv \Gamma$ for all τ. Thus, $\Gamma \vdash_\lambda \mathsf{e} : \mathsf{T}$ and $\Gamma \vdash_\lambda \mathsf{e'} : \mathsf{T}$ are valid typing assignments in STLC with the same type and type environment.

3.2 TTS_λ is Productive

To show that a program transformation always produces a result, it is enough to show that the transformation system admits the identity transformation: A program can always be transformed into itself. More formally, the following theorem has to be shown:

Theorem 3 (TTS_λ admits the identity transformation)
For all well-typed terms $\Gamma \vdash_\lambda \mathsf{e} : \tau$ an identity transformation exists with the judgement $\Gamma \uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e} : \tau$.

Proof 3 (TTS_λ admits the identity transformation)
The existence of the propagation rules makes that the identity transformation can always be constructed. Each rule in the stlc typing derivation has a matching rule in TTS_λ which can always be applied.

3.3 TTS_λ Preserves Semantics

The third TTS property states that two terms should be semantically equivalent. Semantic equivalence is proven by showing $\beta\eta$-convertibility of the source and result term under all closing environments, as formulated in as the following property:

Lemma 2 (Complete TTS_λ Transformations Preserve Semantics)
For all complete elements $\Gamma \uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e'} : \mathsf{T}$ of the typing relation, $\mathsf{e} / \theta \equiv_{\beta\eta} \mathsf{e'} / \theta$ for all closing substitutions θ.

e / s represents capture-avoiding substitution of s in e. A closing substitution is a substitution which substitutes all free variables for closed terms. This is represented by the following construction:

$$\begin{aligned} \emptyset\downarrow &= \mathbf{id} \\ \theta, \mathsf{x}\downarrow &= [\mathsf{x} \mapsto \mathsf{e}] \circ \theta\downarrow \end{aligned}$$

Lemma 2 is proven with the use of a logical relations.

3.3.1 Logical Relations

A proof by logical relations is a proof methodology which has been very useful for proving properties about programming languages based on the simply typed lambda calculus. The fundamentals of logical relations are explained here, for further reading the reader is referred to Mitchell [Mit96], Hinze [Hin00] or Schurmann [SS08].

A logical relation represents a property over a term, or between multiple terms. Characteristic of a logical relation is that the represented property depends on the type of the term(s) over which the property is established. For base types this is often a simple judgement formulated in some logic. At function types the relation establishes that inputs related by the logical relation result in outputs related by the logical relation. Thus a typical unary logical relation for the simply typed lambda calculus looks as follows:

$$\begin{array}{lll} [\![\mathsf{e}]\!]_{\mathsf{T}} & = & \mathcal{J}(\mathsf{e}) \\ [\![\mathsf{e}]\!]_{\tau_a \to \tau_r} & = & \forall\, \mathsf{a} : [\![\mathsf{a}]\!]_{\tau_a} \supset [\![\mathsf{e\ a}]\!]_{\tau_r} \end{array}$$

Here $\mathcal{J}(\mathsf{e})$ represents some property of the term e. The case for function terms states that function application should preserve this property. The logical relation may be stated in any logic, but is often represented in set-theory. Another way to look at logical relations is as an induction hypothesis indexed by types.

3.3.2 A Logical Relation for TTS_λ

TTS_λ constructs a transformation between two terms e and $\mathsf{e'}$, typed by a typing functor $\mathring{\tau}$. Thus the logical relation becomes a binary logical relation over e and $\mathsf{e'}$, indexed by the typing functor $\mathring{\tau}$. To deal with free variables the relation is parametrized by closing substitutions to close the terms. This leads to the following logical relation for TTS_λ.

$$\begin{array}{lll} \mathcal{V}[\![\iota]\!] & \theta\,\theta'\ \mathsf{e}\ \mathsf{e'} & = \mathbf{rep}\ (\mathsf{e} \,/\, \theta) \equiv_{\beta\eta} \mathsf{e'} \,/\, \theta' \\ \mathcal{V}[\![\mathsf{T}]\!] & \theta\,\theta'\ \mathsf{e}\ \mathsf{e'} & = \mathsf{e} \,/\, \theta \equiv_{\beta\eta} \mathsf{e'} \,/\, \theta' \\ \mathcal{V}[\![\mathring{\tau}_a \to \mathring{\tau}_r]\!]\,\theta\,\theta'\ \mathsf{e}\ \mathsf{e'} & & = \forall\, \mathsf{a\ a'} : \mathcal{V}[\![\mathring{\tau}_a]\!]\,\theta\,\theta'\ \mathsf{a\ a'} \supset \mathcal{V}[\![\mathring{\tau}_r]\!]\,\theta\,\theta'\ (\mathsf{e\ a})\ (\mathsf{e'\ a'}) \end{array}$$

The basic property that is being established is $\beta\eta$-convertibility between the source and result term. The case for the hole type ι represents that the source term can be transformed into the result term by applying the function **rep**. For base types we expect both terms to be convertible. The function type establishes that related transformations result in related transformations when applied.

The logical relation extends naturally from single terms to closing substitutions. Two closing substitutions are related when they substitute related terms for equal variables. Note that equal variables may have different types in the source and result terms, but are indexed by the same variable in the functor context. The relation for closing substitutions is indexed by the types in the functor context, which it closes.

$$\begin{array}{lll} \mathcal{E}[\![\emptyset]\!] & \mathbf{id}\quad\quad\ \mathbf{id} & = \emptyset \\ \mathcal{E}[\![\mathring{\Gamma}\,,\ \mathring{\tau}]\!]\ ([\mathsf{x} \mapsto \mathsf{i}] \circ \theta)\ ([\mathsf{x} \mapsto \mathsf{i'}] \circ \theta') & & = \mathcal{E}[\![\mathring{\Gamma}]\!]\,\theta\,\theta' \wedge \mathcal{V}[\![\mathring{\tau}]\!]\,\theta\,\theta'\ \mathsf{i\ i'} \end{array}$$

3.3.3 Proof using Logical Relations

When constructing a proof using logical relations, the proof is usually constructed using two key theorems, occurring in roughly the following form:

1. Fundamental theorem: the relation $[\![\mathsf{e}]\!]_\tau$ can be established for all terms $\mathsf{e} : \tau$

2. Extraction theorem: terms related by $[\![\mathsf{e}]\!]_\tau$ give rise to some property $\mathcal{P}(\mathsf{e})$

The terms 'Fundamental Theorem' and 'Extraction Theorem' are the names used in [SS08]. Other sources refer to the Fundamental Theorem as the Basic or Main Lemma [Mit96] [Hin00]. It is easy to see that the combination of these two theorems can be used to prove a property $\mathcal{P}(\mathsf{e})$ for all $\mathsf{e} : \tau$. This same approach is taken here, leading to the following key theorems to show that a transformation in TTS_λ is semantics preserving.

Theorem 4 (Fundamental Theorem for TTS_λ)
For all term e and e' for which a transformation deduction $\mathring{\Gamma} \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathring{\tau}$ exists, the logical relation $\mathcal{V}[\![\mathring{\tau}]\!]\,\theta\,\theta'\,\mathsf{e}\,\mathsf{e}'$ holds under all related environments $\mathcal{E}[\![\mathring{\Gamma}]\!]\,\theta\,\theta'$.

Theorem 5 (Extraction Theorem for TTS_λ)
For all complete terms $\Gamma \vdash_\lambda \mathsf{e} : \mathsf{T}$, $\Gamma \vdash_\lambda \mathsf{e}' : \mathsf{T}$ and closing substitution $\theta : \Gamma\downarrow$, the relation $\mathcal{V}[\![\mathsf{T}]\!]\,\theta\,\theta\,\mathsf{e}\,\mathsf{e}'$ implies $\mathsf{e}\,/\,\theta \equiv_{\beta\eta} \mathsf{e}'\,/\,\theta$.

Corollary 1 (A Substitution Environment is logically related to Itself)
An important corollary of the Fundamental Theorem and Identity Transformation property, is that a substitution environment is related to itself. This follows from the fact that all terms in the environment have an identity transformation and according to the fundamental theorem each valid transformation is part of the logical relation.

Proof 4 (Extraction Theorem)
The proof for the extraction theorem is immediate, because the logical relation $\mathcal{V}[\![\mathsf{T}]\!]\,\theta\,\theta\,\mathsf{e}\,\mathsf{e}'$ on base types is defined as $\mathsf{e}\,/\,\theta \equiv_{\beta\eta} \mathsf{e}'\,/\,\theta$.

With these properties it is possible to show that TTS_λ is semantics preserving.

Proof 5 (The Fundamental and Extraction Theorem imply the Equivalence Property)
The Fundamental Theorem shows that all complete elements $\Gamma\uparrow \vdash_\lambda \mathsf{e} \rightsquigarrow \mathsf{e}' : \mathsf{T}$ are logically related, with aid of Corollary 1 which shows that substitution environments are always related to themselves. The Extraction Theorem shows that logical relations give rise to $\beta\eta$-equality for complete terms, thus $\mathsf{e}\,/\,\theta \equiv_{\beta\eta} \mathsf{e}'\,/\,\theta$ for all θ.

What is left is showing the fundamental theorem for TTS_λ.

Fundamental theorem Before showing proof of the fundamental theorem, two properties between logical relations have to be established. These properties are stated as 'logical equivalences' between two logical relations, meaning that either both relations hold, or they both do not hold. They always have the same truth value.

The first lemma establishes that $\mathcal{V}[\![\mathring{\tau}]\!]$ is closed under $\beta\eta$-equivalence of the related terms. The second lemma is the Crossing Lemma, which shows that substitutions can be commuted between the closing environments and the related terms. This lemma is the key to proving the logical relation under beta reduction.

Lemma 3 ($\mathcal{V}[\![\mathring{\tau}]\!]$ is Closed under $\beta\eta$-equality)
Two logical relations $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ e e' and $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ f f' are logically equivalent when e $\equiv_{\beta\eta}$ f and e' $\equiv_{\beta\eta}$ f'.

Lemma 4 (Crossing Lemma)
For any two terms i and i' in which x does not appear free, the relation $\mathcal{V}[\![\mathring{\tau}]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') e e' is logically equivalent to $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ e / [x ↦ i] e' / [x ↦ i'].

Both lemmas are proven at the end of this paragraph on page 29, first the fundamental theorem is proven.

Proof 6 (Proof of the Fundamental Theorem)
The fundamental theorem shows that all terms e and e' arising from a transformation derivation $\mathring{\Gamma} \vdash_\lambda$ e ⇝ e' : $\mathring{\tau}$ give rise to an element in the logical relation $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ e e', under all related environments $\mathcal{E}[\![\mathring{\Gamma}]\!]\ \theta\ \theta'$. This is shown by induction on the deduction rules of TTS_λ.

Tr-Con
$$\frac{\text{c is a constant of type T}}{\mathring{\Gamma} \vdash_\lambda \text{c} \rightsquigarrow \text{c} : \text{T}}$$

Here we have to show the logical relation for base types: c / $\theta \equiv_{\beta\eta}$ c / θ', which is trivially true for constants.

c / θ
$\equiv_{\beta\eta}$ {Substitution over constants}
c
$\equiv_{\beta\eta}$ {Substitution over constants}
c / θ'

Tr-Var
$$\frac{\text{x} : \mathring{\tau} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash_\lambda \text{x} \rightsquigarrow \text{x} : \mathring{\tau}}$$

Relatedness of variables is shown 'under related substitution environments', which contain two related terms i and i' for each free variable:

$\mathcal{E}[\![\mathring{\Gamma}$, x]] ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ')
⊃ {Extract related terms for variable x}
$\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ i i'
⊃ {Substitution}
$\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'$ (x / [x ↦ i]) (x / [x ↦ i'])
⊃ {Crossing lemma}
$\mathcal{V}[\![\mathring{\tau}]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') x x

Tr-Lam
$$\frac{\mathring{\Gamma}\text{, x} : \mathring{\tau}_\text{a} \vdash_\lambda \text{e} \rightsquigarrow \text{e'} : \mathring{\tau}_\text{r}}{\mathring{\Gamma} \vdash_\lambda \backslash\text{x. e} \rightsquigarrow \backslash\text{x. e'} : \mathring{\tau}_\text{a} \rightarrow \mathring{\tau}_\text{r}}$$

The lambda rule shows that related inputs result in related outputs under beta-reduction. The following statement is to be shown: $\mathcal{V}[\![\mathring{\tau}_a]\!]\,\theta\,\theta'$ a a' $\supset \mathcal{V}[\![\mathring{\tau}_r]\!]\,\theta\,\theta'$ ((\x. e) a) ((\x. e') a')

$\mathcal{V}[\![\mathring{\tau}_a]\!]\,\theta\,\theta'$ a a'
$\supset$ {Induction hypothesis on e and e' **with** related a and a' **in** the environment}
$\mathcal{V}[\![\mathring{\tau}_r]\!]\,([x \mapsto a] \circ \theta)\,([x \mapsto a'] \circ \theta')$ e e'
$\supset$ {Crossing lemma}
$\mathcal{V}[\![\mathring{\tau}_r]\!]\,\theta\,\theta'$ (e / [x $\mapsto$ a]) (e' / [x $\mapsto$ a'])
$\supset$ {Relation is closed under beta-eta equivalence}
$\mathcal{V}[\![\mathring{\tau}_r]\!]\,\theta\,\theta'$ ((\x. e) a) ((\x. e') a')

Tr-App
$$\frac{\mathring{\Gamma} \vdash_\lambda f \rightsquigarrow f' : \mathring{\tau}_a \to \mathring{\tau}_r \qquad \mathring{\Gamma} \vdash_\lambda e \rightsquigarrow e' : \mathring{\tau}_a}{\mathring{\Gamma} \vdash_\lambda f\ e \rightsquigarrow f'\ e' : \mathring{\tau}_r}$$

The logical relation already establishes that related inputs result in related outputs for related terms. This makes the application rule trivial to prove. The following needs to be shown: $\mathcal{V}[\![\tau_r]\!]\,\theta\,\theta'$ (f e) (f' e')

$\mathcal{V}[\![\tau_a]\!]\,\theta\,\theta'$ e e' {From induction on e and e'}
$\supset$ {From induction on f and f' and modus ponens}
$\mathcal{V}[\![\tau_r]\!]\,\theta\,\theta'$ (f e) (f' e')

Tr-Rep
$$\frac{\mathring{\Gamma} \vdash_\lambda e \rightsquigarrow e' : \mathcal{A}}{\mathring{\Gamma} \vdash_\lambda e \rightsquigarrow \mathbf{rep}\ e' : \iota}$$

The logical relation for the hole type ι dictates that **rep** applied to the source term should be $\beta\eta$-equivalent to the result term. Thus for the rep rule we simple need to show that **rep** e / $\theta \equiv_{\beta\eta}$ **rep** e' / θ'.

rep e / θ
$\equiv_{\beta\eta}$ {Induction hypothesis}
rep e' / θ'

Tr-Abs
$$\frac{\mathring{\Gamma} \vdash_\lambda e \rightsquigarrow e' : \iota}{\mathring{\Gamma} \vdash_\lambda e \rightsquigarrow \mathbf{abs}\ e' : \mathcal{A}}$$

To prove the Tr-Abs rule, the fact that **abs** and **rep** are a retraction pair is needed. The statement to prove is e / $\theta \equiv_{\beta\eta}$ **abs** e' / θ'.

e / θ
$\equiv_{\beta\eta}$ {Retraction pair **abs** and **rep**}
abs (**rep** e / θ)

$\equiv_{\beta\eta}$ {Induction hypothesis}
abs e' / θ'

What is left is showing the $\beta\eta$-closure and Crossing Lemma used in this proof.

Proof 7 (Proof of Crossing Lemma)
We need to show that the relation $\mathcal{V}[\![\mathring{\tau}]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') e e' is logically equivalent to $\mathcal{V}[\![\mathring{\tau}]\!]$ θ θ' (e / [x ↦ i]) (e' / [x ↦ i']). In other words: substitutions can be commuted between the environment and the terms.

This logical equivalence is proven by showing implication in both directions. In both directions the proof is carried out by induction on the typing functor $\mathring{\tau}$. The two implications directions are proven by mutual induction: both directions of the bi-implication make use of the other direction to prove themselves. While this may seem like cyclic reasoning, it is not, because the induction hypothesis is only used on smaller types.

Proof in the direction: $\mathcal{V}[\![\mathring{\tau}]\!]$ θ θ' (e / [x ↦ i]) (e' / [x ↦ i']) implies $\mathcal{V}[\![\mathring{\tau}]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') e e'.

ι case, showing **rep** e / [x ↦ i] ∘ θ $\equiv_{\beta\eta}$ e' / [x ↦ i'] ∘ θ'

rep e / [x ↦ i] ∘ θ
$\equiv_{\beta\eta}$ {Definition substitution and · / ·}
rep (e / [x ↦ i]) / θ
$\equiv_{\beta\eta}$ {Premise}
(e' / [x ↦ i']) / θ'
$\equiv_{\beta\eta}$ {Definition substitution and · / ·}
e' / [x ↦ i'] ∘ θ'

T case is analogous to the ι case.

$\tau_a \to \tau_r$ case, showing $\mathcal{V}[\![\mathring{\tau}_a]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') a a' ⊃ $\mathcal{V}[\![\mathring{\tau}_r]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') (e a) (e' a')

$\mathcal{V}[\![\mathring{\tau}_a]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') a a'
⊃ {Crossing lemma}
$\mathcal{V}[\![\mathring{\tau}_a]\!]$ θ θ' (a / [x ↦ i]) (a' / [x ↦ i'])
⊃ {Premise}
$\mathcal{V}[\![\mathring{\tau}_r]\!]$ θ θ' (e a / [x ↦ i]) (e a' / [x ↦ i'])
⊃ {Induction}
$\mathcal{V}[\![\mathring{\tau}_r]\!]$ ([x ↦ i] ∘ θ) ([x ↦ i'] ∘ θ') (e a) (e' a')

Proof in the direction: $\mathcal{V}[\![\mathring{\tau}]\!]\ ([\mathsf{x} \mapsto \mathsf{i}] \circ \theta)\ ([\mathsf{x} \mapsto \mathsf{i'}] \circ \theta')\ \mathsf{e}\ \mathsf{e'}$ implies $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'\ (\mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}])\ (\mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}])$.

ι case, showing $\mathbf{rep}\ \mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}] \circ \theta \equiv_{\beta\eta} \mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}] \circ \theta'$

$$
\begin{array}{l}
\quad \mathbf{rep}\ (\mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}]) / \theta \\
\equiv_{\beta\eta} \{\text{Definition substitution}\} \\
\quad \mathbf{rep}\ \mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}] \circ \theta \\
\equiv_{\beta\eta} \{\text{Premise}\} \\
\quad \mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}] \circ \theta' \\
\equiv_{\beta\eta} \{\text{Definition substitution}\} \\
\quad (\mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}]) / \theta'
\end{array}
$$

T case is analogous to the ι case.

$\tau_\mathsf{a} \to \tau_\mathsf{r}$ case, showing $\mathcal{V}[\![\mathring{\tau}_\mathsf{a}]\!]\ \theta\ \theta'\ \mathsf{a}\ \mathsf{a'} \supset \mathcal{V}[\![\mathring{\tau}_\mathsf{r}]\!]\ \theta\ \theta'\ ((\mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}]\ \mathsf{a}))\ ((\mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}]\ \mathsf{a'}))$

$$
\begin{array}{l}
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{a}]\!]\ \theta\ \theta'\ \mathsf{a}\ \mathsf{a'} \\
\supset \{\text{Substitution on unbound variable x } \mathbf{in} \text{ a}\} \\
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{a}]\!]\ \theta\ \theta'\ (\mathsf{a} / [\mathsf{x} \mapsto \mathsf{i}])\ (\mathsf{a'} / [\mathsf{x} \mapsto \mathsf{i'}]) \\
\supset \{\text{Crossing lemma}\} \\
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{a}]\!]\ ([\mathsf{x} \mapsto \mathsf{i}] \circ \theta)\ ([\mathsf{x} \mapsto \mathsf{i'}] \circ \theta')\ \mathsf{a}\ \mathsf{a'} \\
\supset \{\text{Premise}\} \\
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{r}]\!]\ ([\mathsf{x} \mapsto \mathsf{i}] \circ \theta)\ ([\mathsf{x} \mapsto \mathsf{i'}] \circ \theta')\ (\mathsf{e}\ \mathsf{a})\ (\mathsf{e}\ \mathsf{a'}) \\
\supset \{\text{Induction}\} \\
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{r}]\!]\ \theta\ \theta'\ (\mathsf{e}\ \mathsf{a} / [\mathsf{x} \mapsto \mathsf{i}])\ (\mathsf{e}\ \mathsf{a'} / [\mathsf{x} \mapsto \mathsf{i'}]) \\
\supset \{\text{x is unbound } \mathbf{in} \text{ a}\} \\
\quad \mathcal{V}[\![\mathring{\tau}_\mathsf{r}]\!]\ \theta\ \theta'\ (\mathsf{e} / [\mathsf{x} \mapsto \mathsf{i}]\ \mathsf{a})\ (\mathsf{e'} / [\mathsf{x} \mapsto \mathsf{i'}]\ \mathsf{a'})
\end{array}
$$

Proof 8 (Proof of $\beta\eta$-closure)
We have to show that $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'\ \mathsf{e}\ \mathsf{e'}$ and $\mathcal{V}[\![\mathring{\tau}]\!]\ \theta\ \theta'\ \mathsf{f}\ \mathsf{f'}$ are logically equivalent when $\mathsf{e} \equiv_{\beta\eta} \mathsf{f}$ and $\mathsf{e'} \equiv_{\beta\eta} \mathsf{f'}$. The proof is by induction on $\mathring{\tau}$. Due to the symmetric nature of this statement it is only necessary to prove this property in one direction.

ι case, proving: $\mathbf{rep}\ \mathsf{f} / \theta \equiv_{\beta\eta} \mathsf{f'} / \theta'$

$$
\begin{array}{l}
\quad \mathbf{rep}\ \mathsf{f} / \theta \\
\equiv_{\beta\eta} \{\text{Premise e} \equiv_{\beta\eta} \text{f}\} \\
\quad \mathbf{rep}\ \mathsf{e} / \theta \\
\equiv_{\beta\eta} \{\text{Premise}\} \\
\quad \mathsf{e'} / \theta' \\
\equiv_{\beta\eta} \{\text{Premise}\} \\
\quad \mathsf{f'} / \theta'
\end{array}
$$

T case is analogous to the ι case.

$\tau_a \to \tau_r$ case, showing for all a and a': $\mathcal{V}[\![\tau_a]\!]\,\theta\,\theta'$ a a' $\supset \mathcal{V}[\![\tau_r]\!]\,\theta\,\theta'$ (f a) (f' a') given the premises.

$\mathcal{V}[\![\tau_a]\!]\,\theta\,\theta'$ a a'
$\supset$ {Premise}
$\mathcal{V}[\![\tau_r]\!]\,\theta\,\theta'$ (e a) (e a')
$\supset$ {Induction hypothesis **with** extended premises: e a $\equiv_{\beta\eta}$ f a and e' a' $\equiv_{\beta\eta}$ f' a'}
$\mathcal{V}[\![\tau_r]\!]\,\theta\,\theta'$ (f a) (f a')

3.4 Hughes' Strings Transformation Preserves the Transformation Properties

We can now show that the Hughes' Strings instantiation of TTS_λ is a valid transformation. For this two additional properties have to be established:

- It needs to be shown that $\mathbf{abs}_{ss}$ and $\mathbf{rep}_{ss}$ form a retraction pair, because this is a premise of TTS_λ. A retraction pair has the property that $\mathbf{abs}_{ss} \circ \mathbf{rep}_{ss} \equiv \mathbf{id}$.
- An extra case has to be added to the Fundamental Theorem for the Tr-Comp rewrite rule.

The retraction is proven through straightforward equational reasoning.

Proof 9 ($\mathbf{abs}_{ss}$ and $\mathbf{rep}_{ss}$ form a retraction)
$\mathbf{abs}_{ss} \circ \mathbf{rep}_{ss}$
$\equiv_{\beta\eta}$ {Eta expansion}
\x $\to$ ($\mathbf{abs}_{ss} \circ \mathbf{rep}_{ss}$) x
$\equiv_{\beta\eta}$ {Definition composition}
\x $\to$ $\mathbf{abs}_{ss}$ ($\mathbf{rep}_{ss}$ x)
$\equiv_{\beta\eta}$ {Defintition of $\mathbf{rep}_{ss}$}
\x $\to$ $\mathbf{abs}_{ss}$ (x ++)
$\equiv_{\beta\eta}$ {Definition of $\mathbf{abs}_{ss}$}
\x $\to$ x ++ ""
$\equiv_{\beta\eta}$ {Definition of (++)}
\x $\to$ x
$\equiv_{\beta\eta}$ {Definition of identity}
id

The extra case for Tr-Comp for the Fundamental Theorem shows that the Tr-Comp rule adheres to the logical relation. This boils down to the following lemma:

Lemma 5 (Tr-Comp supports the Fundamental Theorem)
For all environments s and s' and related terms $\mathbf{rep}_{ss}$ a / s $\equiv_{\beta\eta}$ a' / s' and $\mathbf{rep}_{ss}$ b / s $\equiv_{\beta\eta}$ b' / s' we have that $\mathbf{rep}_{ss}$ (a ++ b) / s $\equiv_{\beta\eta}$ a' $\circ$ b' / s'.

Proof 10 (Tr-Comp support the Fundamental Theorem)

$(\mathbf{rep}_{ss}\ (a +\!\!+ b)) / s$
$\equiv_{\beta\eta}$ {Definition $\mathbf{rep}_{ss}$}
$((a +\!\!+ b) +\!\!+) / s$
$\equiv_{\beta\eta}$ {Commute substitution}
$((a / s +\!\!+ b / s) +\!\!+)$
$\equiv_{\beta\eta}$ {Eta expansion}
$\backslash x.\ (a / s +\!\!+ b / s) +\!\!+ x$
$\equiv_{\beta\eta}$ {Associativity $(+\!\!+)$}
$\backslash x.\ a / s +\!\!+ (b / s +\!\!+ x)$
$\equiv_{\beta\eta}$ {Definition $(\circ)$}
$\backslash x.\ (a / s +\!\!+) \circ (b / s +\!\!+)\ x$
$\equiv_{\beta\eta}$ {Eta reduction}
$(a / s +\!\!+) \circ (b / s +\!\!+)$
$\equiv_{\beta\eta}$ {Definition $\mathbf{rep}_{ss}$}
$\mathbf{rep}_{ss}\ a / s \circ \mathbf{rep}_{ss}\ b / s$
$\equiv_{\beta\eta}$ {Premises}
$a' / s' \circ b' / s'$
$\equiv_{\beta\eta}$ {Definition substitution}
$a' \circ b' / s'$

3.5 Discussion

Taking a step back, we can see that the ideas of the type and transform system are firmly rooted in the idea of type abstraction, or representation independence as it is called in Mitchell [Mit96]. The idea of type abstraction is the idea that the implementation of a datatype can be freely changed, as long as this results in the same external behavior. This makes it possible for the creators of compilers to change for example the representation of integers without breaking all software, as long as all functions on integers yield comparable results.

In our transformation system, the **abs** and **rep** functions make it possible to locally change the representation of a type, making it possible to switch to a different type in specific parts of the program. The requirement of the type and transform system that the transformed types form a retraction and that each transformed term is related to another transformed term can be seen as showing that the transformed type is just a different implementation for the original type. In the case of Hughes' strings the relatedness of Tr-Comp and the left inverse of $\mathbf{rep}_{\mathsf{ss}}$ and $\mathbf{abs}_{\mathsf{ss}}$ show that **String** $\rightarrow$ **String** can be a replacement implementation for **String** under the idea of type abstraction.

The idea that function space can preserve relations between types has been addressed before by Backhouse & Backhouse [BB02]. In their work they show that if there exists a Galois Connection between two types, there exists a Galois Connection between the functions on those types. As example they show that for functions which work on booleans, there is a function which works on integers which can be used as an implementation of the boolean function. They also prove this using logical relations.

Ahmed and Blume [AB11] used logical relations to show that a CPS transformation from the simply typed lambda calculus to System F will correctly execute in the target language. Their approach is very similar to ours in that they use a logical relation to relate the terms under transformation. However, where they use the logical relation to show that a transformed term can transformed back, we use a forward translation in our approach: the ι case of the logical relation shows that the result term can be constructed from the source term.

3.6 The Typing Functor

The typing functor is the heart of the type and transform system. The use of this functor gives rise to both the type-correctness as well as the semantic correctness of TTS_λ through the logical relation. The fact that this works out is because the typing functor is defined as an actual functor as known in the field category theory. It is purely out of theoretical curiosity that we show this here. Before showing that the typing functor is a proper functor, we introduce the concept of functors and difunctors.

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:

```
type F a
fmap :: (a → b) → F a → F b
```

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

```
fmap id = id
fmap g ∘ fmap f = fmap (g ∘ f)
```

A list is an example of a functor in Haskell:

```
data List a = Nil | Cons a (List a)


fmap_list :: (a → b) → List a → List b
fmap_list _ Nil         = Nil
fmap_list f (Cons a l) = Cons (f a) (fmap_list f l)
```

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 $\mathring{\tau}$.

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 contravariant (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[MH95] 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 contravariant positions.

$$\mathbf{dimap}_{\mathsf{F}} :: (\mathsf{b} \to \mathsf{a}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{F\ a} \to \mathsf{F\ b}$$

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

$$\begin{aligned}&\mathbf{dimap}_{\mathsf{Func}} :: (\mathsf{b} \to \mathsf{a}) \to (\mathsf{a} \to \mathsf{b}) \to \mathsf{Func\ a} \to \mathsf{Func\ b}\\ &\mathbf{dimap}_{\mathsf{Func}}\ \mathsf{g\ f\ func} = \mathsf{f} \circ \mathsf{func} \circ \mathsf{g}\end{aligned}$$

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

$$\begin{aligned}&\mathbf{dimap}_{\mathsf{F}}\ \mathbf{id}\ \mathbf{id} = \mathbf{id}\\ &\mathbf{dimap}_{\mathsf{F}}\ \mathsf{g1\ g2} \circ \mathbf{dimap}_{\mathsf{F}}\ \mathsf{f1\ f2} = \mathbf{dimap}_{\mathsf{F}}\ (\mathsf{f1} \circ \mathsf{g1})\ (\mathsf{g2} \circ \mathsf{f2})\end{aligned}$$

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

Such a **dimap** can be defined for the typing functor $\mathring{\tau}$ using the following functor-indexed function:

$$\begin{aligned}&\mathbf{dimap}_{\mathring{\tau}} :: (\mathsf{a} \to \mathsf{b}) \to (\mathsf{b} \to \mathsf{a}) \to \mathring{\tau}_{\mathsf{b}} \to \mathring{\tau}_{\mathsf{a}}\\ &\mathbf{dimap}_{\iota}\quad \mathsf{con\ cov\ x} = \mathsf{cov\ x}\\ &\mathbf{dimap}_{\mathsf{T}}\quad \mathsf{con\ cov\ x} = \mathsf{x}\\ &\mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{con\ cov\ f} = \mathbf{dimap}_{\mathring{\tau}_{\mathsf{r}}}\ \mathsf{con\ cov} \circ \mathsf{f} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}}}\ \mathsf{cov\ con}\end{aligned}$$

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 $\mathring{\tau}_{\mathsf{b}} \to \mathring{\tau}_{\mathsf{a}}$ to $\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{b}}$.

Typing Functor Showing that the typing functor is an actual functor can now be done by showing that the **dimap** constructed for the typing functor preserves the functor laws: identity and composition. The identity law follows readily from the definition of **dimap** and induction on the typing functor. The functor law for composition also follows from induction on the typing functor, the only non-trival case here is the function case, which can be shown using equational reasoning:

$$\begin{aligned}&\quad \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{g1\ g2} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{f1\ f2}\\ &\equiv_{\beta\eta}\ \{\mathsf{Eta\ expansion}\}\\ &\quad \backslash \mathsf{x}.\ (\mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{g1\ g2} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{f1\ f2})\ \mathsf{x}\\ &\equiv_{\beta\eta}\ \{\mathsf{Definition\ composition}\}\\ &\quad \backslash \mathsf{x}.\ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{g1\ g2}\ (\mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}} \to \mathring{\tau}_{\mathsf{r}}}\ \mathsf{f1\ f2\ x})\\ &\equiv_{\beta\eta}\ \{\mathsf{Definition}\ \mathbf{dimap}\}\\ &\quad \mathbf{dimap}_{\mathring{\tau}_{\mathsf{r}}}\ \mathsf{g1\ g2} \circ (\mathbf{dimap}_{\mathring{\tau}_{\mathsf{r}}}\ \mathsf{f1\ f2} \circ \mathsf{x} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}}}\ \mathsf{f2\ f1}) \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}}}\ \mathsf{g2\ g1}\\ &\equiv_{\beta\eta}\ \{\mathsf{Associativity\ composition}\}\\ &\quad (\mathbf{dimap}_{\mathring{\tau}_{\mathsf{r}}}\ \mathsf{g1\ g2} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{r}}}\ \mathsf{f1\ f2}) \circ \mathsf{x} \circ (\mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}}}\ \mathsf{f2\ f1} \circ \mathbf{dimap}_{\mathring{\tau}_{\mathsf{a}}}\ \mathsf{g2\ g1})\\ &\equiv_{\beta\eta}\ \{\mathsf{Induction\ hypothesis}\}\end{aligned}$$

$$
\begin{aligned}
&\mathbf{dimap}_{\mathring{\tau}_\mathsf{r}}\ (\mathsf{g2} \circ \mathsf{f2})\ (\mathsf{f1} \circ \mathsf{g1}) \circ \mathsf{x} \circ \mathbf{dimap}_{\mathring{\tau}_\mathsf{a}}\ (\mathsf{f1} \circ \mathsf{g1})\ (\mathsf{g2} \circ \mathsf{f2}) \\
\equiv_{\beta\eta}\ &\{\text{Definition } \mathbf{dimap}\} \\
&\mathbf{dimap}_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}\ (\mathsf{f1} \circ \mathsf{g1})\ (\mathsf{g2} \circ \mathsf{f2})\ \mathsf{x} \\
\equiv_{\beta\eta}\ &\{\text{Eta redutcion}\} \\
&\mathbf{dimap}_{\mathring{\tau}_\mathsf{a} \to \mathring{\tau}_\mathsf{r}}\ (\mathsf{f1} \circ \mathsf{g1})\ (\mathsf{g2} \circ \mathsf{f2})
\end{aligned}
$$

Thus the typing functor is a proper functor.

4 Mechanical Proof of the Transformation Properties

Proving properties about the programs we write increases our trust in the reliability of our software. Although a pen-and-paper proof usually suffices to prove properties, it is still possible to make errors. In recent years it has become more common to prove parts of software systems with the use of theorem provers. To take this one step further, the POPLmark [pop] challenge has set its goal to mechanically verify all meta-theory of programming languages.

In this light, we have verified the TTS_λ transformation system in the dependently type programming language Agda [Nor07]. Agda is a programming language which can be used for theorem proving and is based on constructive mathematics. We have chosen Agda as proof-assistant because it supports universe polymorphism, clean syntax through the use of mixfix operators and parametrized modules.

This chapter will give an overview of how the transformation system is represented in Agda and how the transformation properties are mechanically verified. The proof relies on the validity of the Curry-Howard correspondence, which will be introduced in Section 4.3. The source code can be found on GitHub [sou] for the interested reader.

4.1 STLC Object Language

Fundamental to a mechanical formalization of the TTS system is the representation of the object language, STLC. STLC can be represented in multiple ways, as described in [KJ12]. The essential choice is between a Higher Order Abstract Syntax and a de Bruijn representation, among others. Although HOAS terms can be constructed in Agda, Agda is not strong enough to reason about semantic equivalence of HOAS terms, unlike other languages such as Twelf [SS08]. The representation chosen here is a first-order representation using well-typed de Bruijn indices as found in Keller and Altenkirch [KA10]. A first-order formulation is mandatory because it allows full inductive reasoning over terms and types in the object language and thus reasoning about the semantics. Formulating using well-typed de Bruijn indices is useful because it asserts important properties about the terms by construction.

De Bruijn indices is a way to represent variables in languages based on the lambda calculus. Instead of naming a variable, a variable is given an index which denotes at which lambda the variable is bound. More precisely, the index denotes the number of lambdas that occur between the variable and its binding site. *Well-typed* de Bruijn indices are an extension of this idea: each variable now has a type associated with its binding site. When a term contains free variables, a context is used to assign each free variable a type and a binding place. Types and contexts are defined in the following way:

```
data Con : Set where
  ϵ     : Con
  _ , _ : Con → Ty → Con

data Ty : Set where
  C     : ℕ → Ty
  _⇒_ : Ty → Ty → Ty
```

Because variables are nameless in the de Bruijn representation, each item in the type context **Con** represents a consecutive binding site for free variables and the type that is associated with it. Variable indices are defined as indices into this surrounding context.

```
data _∋_ : Con → Ty → Set where
  vz : ∀ {Γ σ}   → Γ , σ ∋ σ
  vs : ∀ {τ Γ σ} → Γ ∋ σ → Γ , τ ∋ σ
```

Using these constructions we can construct a representation for the simply typed lambda calculus.

```
data _⊢_ (Γ : Con) : Ty → Set where
  var : ∀ {σ}   → Γ ∋ σ → Γ ⊢ σ
  Λ   : ∀ {σ τ} → Γ , σ ⊢ τ → Γ ⊢ σ ⇒ τ
  _·_ : ∀ {σ τ} → Γ ⊢ σ ⇒ τ → Γ ⊢ σ → Γ ⊢ τ
```

Terms are indexed by a context representing the free variables it may contain, along with the resulting type of the term itself. A variable is represented by an index into the context and produces a term with the type of that variable. A lambda binding binds a free variable by removing it from the context and introducing a function space. Function application combines a function and argument into a term of the result type.

This method of representing STLC terms has two very important benefits:

- The terms are well-typed *by construction*. In other words, it is impossible to construct an ill-typed stlc term using this data type.
- The terms are well-scoped *by construction*. Free variables are explicitly represented in the typing context. A term with an empty context is guaranteed to contain no free variables.

Note also that we do not specify a semantic interpretation for the simply typed lambda calculus and no typing language is specified. This is done intentionally to show that the correctness of TTS_λ is independent of the semantics of the simply typed lambda calculus and independent of the typing language. The only restriction on the semantics is that it allows $\beta\eta$-convertibility as defined in section 4.1.2.

4.1.1 Manipulating and Constructing Terms

The simply typed lambda calculus comes equipped with a set of functions to construct, and, most importantly, evaluate terms.

Context weakening When constructing STLC terms the need arises to introduce fresh variables. Creating room in a typing context for a variable is simple, but the typing context of a term can not be changed at will. A function is needed which changes a term to accept an extra free variable in the context.

Extending a typing context is done with help of the converse function, removal of a variable from the context:

```
_-_ : {σ : Ty} → (Γ : Con) → Γ ∋ σ → Con
ϵ     - ()
Γ , σ - vz    = Γ
Γ , τ - vs x  = (Γ - x) , τ
```

Using this function it is possible to express how a term *without* a certain variable, can be turned into a term *with* that variable. This is done by the following two functions:

```
wkv : ∀ {Γ σ τ} → (x : Γ ∋ σ) → Γ - x ∋ τ → Γ ∋ τ
wkv vz y          = vs y
wkv (vs x) vz     = vz
wkv (vs x) (vs y) = vs (wkv x y)


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 : ∀ {σ Γ τ} → Γ ⊢ τ → Γ , σ ⊢ τ
weaken t = wkTm vz t
```

This formulation gives the flexibility to introduce a free variable at any position in the context.

Simultaneous substitution Reductions in the simply typed lambda calculus are based upon substitution. A variant of substitutions that goes very well with the well-typed de Bruijn formulation of the lambda calulus are simultaneous substitutions. Simultaneous substitution is an operation in which all free variables in a term are substituted, at once, with terms belonging to an entirely new context. Such a substitution is defined as follows:

```
data _=>_ : Con → Con → Set where
  sz : ∀ {Δ}     → ϵ ⇒ Δ
  ss : ∀ {Γ Δ τ} → Γ ⇒ Δ → Δ ⊢ τ → Γ , τ ⇒ Δ
```

The substitution type is indexed by two typing contexts. The first type context represents the free variables that will be replaced and the second type context represents the new typing context after substitution. This new typing context is the typing context for all substituted terms.

Before defining the function which will perform the actual substitution on terms, it is necessary to extend to concept of term weakening (opening up a free variable) to simultaneous substitutions. This is done by the following functions:

```
wkS : ∀ {τ Γ Δ} → (x : Δ ∋ τ) → Γ ⇒ Δ - x → Γ ⇒ Δ
wkS v sz       = sz
wkS v (ss y y') = ss (wkS v y) (wkTm v y')

extS : ∀ {τ Γ Δ} → (x : Γ ∋ τ) → (t : Δ ⊢ τ) → Γ - x ⇒ Δ → Γ ⇒ Δ
extS vz t s             = ss s t
extS (vs y) t (ss y' y0) = ss (extS y t y') y0

wkExtS : {Γ Δ : Con} {τ : Ty} → (x : Γ ∋ τ) → (y : Δ ∋ τ)
  → Γ - x ⇒ Δ - y → Γ ⇒ Δ
wkExtS x y v = extS x (var y) (wkS y v)
```

The **wkS** function weakens the result context of a substitution, the **extS** function extends a substitution such that it replaces an extra free variable with a term. **wkExtS** is defined for convenience to create a 'gap' in a substitution. This function adds a new free variable to a substitution which will be replaced by another newly created variable.

The substitution function _/_ can now be defined in the following way, with use of helper function **lookup**. **lookup** retrieves the term at some variable index from a substitution.

```
lookup : ∀ {Γ Δ τ} → (v : Γ ∋ τ) → Γ ⇒ Δ → Δ ⊢ τ
lookup vz (ss y y')       = y'
lookup (vs y) (ss y' y0) = lookup y y'

_/_ : ∀ {Γ Δ τ} → Γ ⊢ τ → Γ ⇒ Δ → Δ ⊢ τ
_/_ (var y) s = lookup y s
_/_ (Λ y)   s = Λ (y / wkExtS vz vz s)
_/_ (y · y') s = (y / s) · (y' / s)
```

Using **lookup** the variable case becomes simple to implement. The interesting case is applying substitutions over lambda abstractions. Lambda abstraction binds a variable from the free variable context of its subterm. In order to be able to apply the given substitution to this subterm, we extend the given substitution with an identity substitution for the newly bound variable. The lambda bound variable will be replaced by itself. This is the desired working of capture-avoiding substitution: shadowed variables will not be changed.

Simultaneous substitutions can not only be applied to terms, but can be applied to other simultaneous substitutions as well. This is shown by the following function.

```
_/=>_ : ∀ {Γ Δ Δ'} → Γ ⇒ Δ → Δ ⇒ Δ' → Γ ⇒ Δ'
sz      /=> s' = sz
ss y y' /=> s' = ss (y /=> s') (y' / s')
```

Single substitution Single substitution can be implemented using simultaneous substitution. Single substitution is needed in order to be able to implement β-reduction. A single substitution is created by extending the identity substitution with a term for one variable.

```
I : ∀ {Γ} → Γ ⇒ Γ
I {ε}      = sz
```

```
I {y , y'} = wkExtS vz vz (I {y})
sub : ∀ {Γ τ} → (v : Γ ∋ τ) → (x : Γ - v ⊢ τ) → Γ ⇒ Γ - v
sub v x = extS v x I
```

The function **up** shows how a simultaneous substitution can be used to lift a closed term into an arbitrary context.

```
up : ∀ {Γ τ} → ϵ ⊢ τ → Γ ⊢ τ
up t = t / sz
```

Term closure The semantic equivalence property of TTS_λ is defined on terms 'under a closing environment'. Thus the object language is equipped with a closure construction, defined as follows:

```
data _↓ : (Γ : Con) → Set where
  ϵ   : ϵ ↓
  cls : ∀ {τ Γ} → Γ ↓ → Γ ⊢ τ → Γ , τ ↓
```

Such a closure carries a closing term for each variable in the environment. These terms should be substituted into a term *sequantially* to close the term. Such a closing operation can not be defined directly but a closure can be turned into an equivalent simultaneous substitution using the following function:

```
↓-=> : ∀ {Γ} → Γ ↓ → Γ ⇒ ϵ
↓-=> ϵ          = sz
↓-=> (cls y y') = ss (↓-=> y) (y' / ↓-=> y)
```

Defining a term closing function is now simple:

```
close : ∀ {Γ τ} → Γ ⊢ τ → Γ ↓ → ϵ ⊢ τ
close t s = t / ↓-=> s
```

Variable construction Variable indices are constructed in the sequential way of Peano numberings. While this simple representation is useful for defining functions and reasoning with indices, it is not very convenient to write variables as sequences of constructors when constructing terms. The helper function **i** makes this process a bit simpler by injecting a natural number into a stlc variable in the free value context.

```
size : Con → ℕ
size ϵ       = zero
size (y , y') = suc (size y)
ind : (Γ : Con) → Fin (size Γ) → Ty
ind ϵ ()
```

```
ind (y , y') zero    = y'
ind (y , y') (suc i) = ind y i
i : ∀ {Γ} → (i : Fin (size Γ)) → Γ ∋ ind Γ i
i {ϵ} ()
i {y , y'} zero    = vz
i {y , y'} (suc i) = vs (i i)
i : ∀ m {Γ} {m<n : True (suc m ⩽? size Γ)} → Γ ∋ ind Γ (#_ m {size Γ} {m<n})
i v = i (# v)
```

This only works as long as the natural number is within the bounds of the context. This is checked by an inferred predicate in the function **i**. If this predicate holds, a bounded (finite) number can be constructed using #, which can then be turned into a variable index using **i**.

Term construction is facilitated by the function **v**. Variables can now be written as normal numbers:

```
v : (m : ℕ) → {Γ : Con} → {m<n : True (suc m ⩽? size Γ)}
    → Γ ⊢ ind Γ (#_ m {size Γ} {m<n})
v x = var (i x)
```

Combined with Agda's mix-fix syntax, this yields a relatively clutter-free method of constructing stlc terms:

```
id : ∀ {a Γ} → Γ ⊢ a ⇒ a
id = Λ (v 0)
const : ∀ {a b Γ} → Γ ⊢ a ⇒ b ⇒ a
const = Λ (Λ (v 1))
comp : ∀ {Γ a b c} → Γ ⊢ (b ⇒ c) ⇒ (a ⇒ b) ⇒ a ⇒ c
comp = Λ (Λ (Λ (v 2 · (v 1 · v 0))))
infixl 2 _∘_
_∘_ : ∀ {a b c Γ} → Γ ⊢ b ⇒ c → Γ ⊢ a ⇒ b → Γ ⊢ a ⇒ c
t1 ∘ t2 = comp · t1 · t2
```

4.1.2 Equality

The semantics of stlc terms can be related using a convertibility relation. The $\beta\eta$-convertibility relation used here is based on work by Keller and Altenkirch [KA10].

```
data _βη-≡_ {Γ : Con} : {σ : Ty} → Γ ⊢ σ → Γ ⊢ σ → Set where
  □     : ∀ {σ} → {t : Γ ⊢ σ} → t βη-≡ t
  bsym : ∀ {σ} → {t₁ t₂ : Γ ⊢ σ} → t₁ βη-≡ t₂ → t₂ βη-≡ t₁
  _↔_ : ∀ {σ} → {t₁ t₂ t₃ : Γ ⊢ σ} → t₁ βη-≡ t₂ → t₂ βη-≡ t₃ → t₁ βη-≡ t₃
  %Λ_ : ∀ {σ τ} → {t₁ t₂ : Γ , σ ⊢ τ} → t₁ βη-≡ t₂ → Λ t₁ βη-≡ Λ t₂
  _%·_ : ∀ {σ τ} → {t₁ t₂ : Γ ⊢ σ ⇒ τ} → {u₁ u₂ : Γ ⊢ σ}
```

```
                → t₁ βη-≡ t₂ → u₁ βη-≡ u₂ → t₁ · u₁ βη-≡ t₂ · u₂
β               : ∀ {σ τ} → {t : Γ , σ ⊢ τ} → {u : Γ ⊢ σ} → Λ t · u βη-≡ t / sub vz u
η               : ∀ {σ τ} → {t : Γ ⊢ σ ⇒ τ} → Λ (weaken t · v 0) βη-≡ t
```

The first three constructors establish reflexivity, symmetry and transitivity of $\beta\eta$-equality. %Λ_ and _%·_ establishes that convertibility is a congruence relation. β and η represent the respective beta-redutcion and eta-expansion rules. Note that because of the use of well-typed de-Bruijn indices, the subject reduction property of the lambda calculus comes for free. β-reduction is guaranteed to preserve typing.

Because the $\beta\eta$-convertibility relation is reflexive, symmetric and transitive, it gives rise to a setoid. In Agda this makes the relation eligible for use with the equational reasoning module, which makes proving properties much more intuitive. As an example, the **ext** property shows that this formalization of the lambda calculus is extensional.

```
βηsetoid : {Γ : Con} {σ : Ty} → Setoid _ _
βηsetoid {Γ} {σ} =
  record {Carrier = Γ ⊢ σ
    ;_≈_ = _βη-≡_
    ;isEquivalence =
      record {refl = □
        ;sym   = bsym
        ;trans = _↔_
        }
    }

ext : ∀ {Γ σ τ} → (e e' : Γ ⊢ σ ⇒ τ)
  → weaken e · v 0 βη-≡ weaken e' · v 0 → e βη-≡ e'
ext e e' eq =
  let open Relation.Binary.EqReasoning βηsetoid
           renaming (_≈⟨_⟩_ to _↔⟨_⟩_)
  in begin
       e
     ↔⟨ bsym η ⟩
       Λ (weaken e · v 0)
     ↔⟨ %Λ eq ⟩
       Λ (weaken e' · v 0)
     ↔⟨ η ⟩
       e'
     ■
```

4.2 Formalization of TTS_λ

Typing Functor With the object language layed out, it is now possible to formalize the basic constructs of TTS_λ. Essential to the type and transform system is the typing functor. The typing

functor is defined as a straightforward inductive datatype as follows:

```
data Functor : Set where
  ι    : Functor
  C    : (𝔄 : ℕ) → Functor
  _⟶_ : (τ̊₁ τ̊₂ : Functor) → Functor
```

As expected, the functor datatype is the same as the stlc base type, with an extra constructor ι representing a hole in the type. Two functions establish the relation between functor types and normal types: The interpretation function $[\![\_]\!]\mathring{\tau}\_$ and the lifting function $\_\uparrow\mathring{\tau}$.

```
[[_]]τ̊_ : Functor → Ty → Ty
[[ ι          ]]τ̊ τ = τ
[[ C n        ]]τ̊ τ = C n
[[ τ̊₁ ⟶ τ̊₂ ]]τ̊ τ = [[ τ̊₁ ]]τ̊ τ ⇒ [[ τ̊₂ ]]τ̊ τ

_↑τ̊ : Ty → Functor
C n      ↑τ̊ = C n
τ₁ ⇒ τ₂ ↑τ̊ = τ₁ ↑τ̊ ⟶ τ₂ ↑τ̊
```

In the Agda world, the functor datatype can be seen as a universe, with $[\![\_]\!]\mathring{\tau}\_$ as universe interpretation onto the base types.

Functor Context A functor context is basically the same as a normal context, only containing functors. The accompanying interpretation and lifting functions are equally straightforward.

```
data Ftx : Set where
  ϵ     : Ftx
  _,_ : (Γ̊ : Ftx) → (τ̊ : Functor) → Ftx

[[_]]Γ̊_ : Ftx → Ty → Con
[[ ϵ      ]]Γ̊ n = ϵ
[[ Γ̊ , τ̊ ]]Γ̊ n = [[ Γ̊ ]]Γ̊ n , [[ τ̊ ]]τ̊ n

_↑Γ̊ : Con → Ftx
ϵ      ↑Γ̊ = ϵ
y , y' ↑Γ̊ = (y ↑Γ̊) , y' ↑τ̊
```

Also the concept of variables extends naturally to functors, with the accompanying interpretation function.

```
data _∋Γ̊_ : Ftx → Functor → Set where
  vz : ∀ {Γ̊ τ̊} → Γ̊ , τ̊ ∋Γ̊ τ̊
  vs : ∀ {Γ τ̊₁ τ̊₂} → Γ ∋Γ̊ τ̊₁ → Γ , τ̊₂ ∋Γ̊ τ̊₁

[[_]]∋_ : ∀ {Γ̊ τ̊} → (v : Γ̊ ∋Γ̊ τ̊) → (a : Ty) → [[ Γ̊ ]]Γ̊ a ∋ [[ τ̊ ]]τ̊ a
[[ vz   ]]∋ t = vz
[[ vs y ]]∋ t = vs ([[ y ]]∋ t)
```

TTS_λ is intended to be a base system for type-changing transformations, independent of which types and terms are changed. In other words, TTS_λ should be parametrizable with different types and rewriting rules. In Agda this can be expressed using parametrized modules. The base system is parametrized by the source and result type $\mathfrak{A}$ and $\mathfrak{R}$, together with the **abs** and **rep** functions to convert between them. Additionally a set of rules is expected which perform the actual transformations.

```
module TTS.Judgement.Base
  (𝔄 : ℕ) (ℜ : Ty)
  (rep : ϵ ⊢ C 𝔄 ⇒ ℜ) (abs : ϵ ⊢ ℜ ⇒ C 𝔄)
  (rules : Rules 𝔄 ℜ)
    where
```

The inductive family representing TTS_λ now looks as follows:

```
infix 1 _:_⊨_⇝_
data _:_⊨_⇝_ : (Γ̊ : Ftx) → (τ̊ : Functor) → (e : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄)
                 → (e' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ) → Set where
  var   : ∀ {Γ̊ τ̊} (v : Γ̊ ∋Γ̊ τ̊)
        → Γ̊ : τ̊ ⊨ var (⟦ v ⟧∋ C 𝔄) ⇝ var (⟦ v ⟧∋ ℜ)
  app   : ∀ {Γ̊ τ̊1 τ̊2 e1 e1' e2 e2'}
        → Γ̊ : τ̊1 ⟶ τ̊2 ⊨ e1 ⇝ e1' → Γ̊ : τ̊1 ⊨ e2 ⇝ e2'
        → Γ̊ : τ̊2 ⊨ e1 · e2 ⇝ e1' · e2'
  lam   : ∀ {Γ̊ τ̊1 τ̊2 e e'} → Γ̊ , τ̊1 : τ̊2 ⊨ e ⇝ e' → Γ̊ : τ̊1 ⟶ τ̊2 ⊨ Λ e ⇝ Λ e'
  i-rep : ∀ {Γ̊ e e'} → Γ̊ : C 𝔄 ⊨ e ⇝ e' → Γ̊ : ι ⊨ e ⇝ up rep · e'
  i-abs : ∀ {Γ̊ e e'} → Γ̊ : ι ⊨ e ⇝ e' → Γ̊ : C 𝔄 ⊨ e ⇝ up abs · e'
  rule  : ∀ {Γ̊ τ̊ e e'} → Rule {τ̊} e e' rules → Γ̊ : τ̊ ⊨ up e ⇝ up e'
```

The inductive family closely follows the specification from chapter 2. In the judgement the functor is moved in front of the converted terms, because scoping in Agda is from left to right.

This definition shows clearly how two terms are typed by one typing functor and functor context. The **var**, **app** and **lam** rules just propagate these type changes using the rules of the simply typed lambda calculus. The **i-rep**, **i-abs** and **rule** construction actually have the power to change the types and terms.

Transformation rules There is a bit of room in the design space as to how represent transformation rules. In this version of the type-and-transform system a simple dictionary of transformable terms is used.

```
data Rules (𝔄 : ℕ) (ℜ : Ty) : Set where
  ϵ       : Rules 𝔄 ℜ
  replace : ∀ {τ̊} → (r : Rules 𝔄 ℜ) → (e : ϵ ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄)
          → (e' : ϵ ⊢ ⟦ τ̊ ⟧τ̊ ℜ) → Rules 𝔄 ℜ
```

```
data Rule {τ̊ 𝔄 ℜ} (e : ϵ ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄) (e' : ϵ ⊢ ⟦ τ̊ ⟧τ̊ ℜ) : Rules 𝔄 ℜ → Set where
  rule : ∀ rs → Rule e e' (replace {𝔄} {ℜ} {τ̊} rs e e')
  skip : ∀ {rs τ̊' f f'} → (r : Rule {τ̊} e e' rs)
                        → Rule e e' (replace {𝔄} {ℜ} {τ̊'} rs f f')
```

The **Rules** datatype is a list of transformation terms. A **Rule** indexes a rule in this list. Note that only closed terms can be transformed using this formulation, this makes the transformations context-insensitive.

4.3 Properties

We will now mechanically show that the defined system adheres to the desired transformation properties of Section 2.2. This proof follows the same structure as the 'pen-and-paper' proof from Chapter 3 and relies on the Curry-Howard correspondence for its validity.

Curry-Howard Correspondence The Curry Howard correspondence is the notion that there exists a direct connection between types in a programming language and propositions in classical, predicate and intuitionistic logic. This correspondence makes it possible to construct proofs as programs: providing an implementation program for a given type corresponds to proving the corresponding logical property. According to the Curry-Howard correspondence, implication in logic has a direct relation to function space in programming languages, and universal quantification is related to dependent function space. Sum types and product types have a direct relation to disjunction and conjunction. In this way the Curry-Howard correspondence makes it possible to mechanically prove logical properties in a programming language, a feature that is put to good use in this section.

4.3.1 Typing Property

The TTS typing property does not need any further proving but is inherent in the construction of the whole system. Terms are type-correct by construction using the well-typed De Bruijn indices and the transformation relation is indexed by two terms which derive a valid type from the typing functor.

4.3.2 Identity Transformation

As a bare minimum, all transformation systems should allow the identity transformation to be productive. The following function shows this by showing that a valid transformation derivation exists for any term in the simply typed lambda calculus:

```
id⇝ : ∀ {Γ τ} → (e : Γ ⊢ τ) → Γ ↑Γ̊ : τ ↑τ̊ ⊨ e ⇝ e
```

The typing functor and functor context of the identity transformation are the lifted version of the base type and context.

This shows clearly how implication in logic is connected to function space in a programming language. The statement: the existence of a valid term implies the existence of a transformation

for that term, is transformed into showing that a function exists which constructs a transformation out of a term.

4.3.3 Semantic Equivalence

The Agda formalization of the equivalence proof strictly follows the proof structure as described in Chapter 3. The first challenge is to find a suitable Agda representation for the logical relation. The representation used here was inspired by the work of Swierstra [Swi12], who uses a unary logical relation to prove termination of the simply typed lambda calculus in Agda.

Logical relation To construct the logical relation we can make good use of dependent types: the relation constructs a different type for the different **Functor** constructors. For base types it constructs a $\beta\eta$ equivalence and for function space it constructs an implication in the form of a normal Agda function.

```
module TTS.Relation.Base (𝔄 : ℕ) (ℜ : Ty) (rep : ϵ ⊢ C 𝔄 ⇒ ℜ) where

rel : ∀ {Γ̊ τ̊} → (e : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄) → (e' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ)
  → (s : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓) → (s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓) → Set
rel {Γ̊} {ι}          e e' s s' = up rep · close e s βη-≡ close e' s'
rel {Γ̊} {C n}        e e' s s' = close e s βη-≡ close e' s'
rel {Γ̊} {τ̊₁ ⟶ τ̊₂} e e' s s' = ∀ a a' → rel {Γ̊} {τ̊₁} a a' s s'
                                  → rel {Γ̊} {τ̊₂} (e · a) (e' · a') s s'
```

Before proving the Fundamental Lemma we first establish the $\beta\eta$-closure and crossing lemma's over the logical relation. Because the type of the logical relation depends on the functor, there is no choice but to prove properties about the logical relation by induction on the typing functor.

```
rel-βη : ∀ {Γ̊ τ̊} {e e' : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄} {f f' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ}
       → (s : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓) → (s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓) → e βη-≡ e' → f βη-≡ f'
       → rel {Γ̊} {τ̊} e f s s' → rel {Γ̊} {τ̊} e' f' s s'
rel-βη {Γ̊} {ι}          s s' eq1 eq2 r = (□ %· %close s (bsym eq1))
                                         ↔ r
                                         ↔ %close s' eq2
rel-βη {Γ̊} {C n}        s s' eq1 eq2 r = %close s (bsym eq1)
                                         ↔ r
                                         ↔ %close s' eq2
rel-βη {Γ̊} {τ̊₁ ⟶ τ̊₂} s s' eq1 eq2 r =
  λ a a' ra → rel-βη {Γ̊} {τ̊₂} s s' (eq1 %· □) (eq2 %· □) (r a a' ra)
```

The $\beta\eta$ closure proof is another good example of the Curry-Howard correspondence in action. In the proof of Chapter 3 we used sequences of implication to show the case for function space. In the Curry-Howard correspondence implication elimination becomes function application.

For the Crossing Lemma we just show the types, which give an exact formal representation of the lemma. The logical equivalence of the crossing lemma is proven by mutual induction on the

two directions of the implication, in Agda we can model this by using a mutual block. Agda's termination checker will tell whether we have cyclic reasoning or not: if the construction of the proof term terminates, we have no cyclic reasoning, if it is proven using infinite recursion the reasoning was cyclic.

```
mutual
  cross-left  : ∀ {Γ̊ τ̊ τ̊'} → (e : ⟦ Γ̊ ⟧Γ̊ C 𝔄 , ⟦ τ̊' ⟧τ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄)
              → (e' : ⟦ Γ̊ ⟧Γ̊ ℜ , ⟦ τ̊' ⟧τ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ)
              → (s : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓) → (s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓)
              → (a : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊' ⟧τ̊ C 𝔄) → (a' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊' ⟧τ̊ ℜ)
              → rel {Γ̊} {τ̊} (e / sub vz a) (e' / sub vz a') s s'
              → rel {Γ̊ , τ̊'} {τ̊} e e' (cls s a) (cls s' a')
  cross-right : ∀ {Γ̊ τ̊ τ̊'} → (e : ⟦ Γ̊ ⟧Γ̊ C 𝔄 , ⟦ τ̊' ⟧τ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄)
              → (e' : ⟦ Γ̊ ⟧Γ̊ ℜ , ⟦ τ̊' ⟧τ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ)
              → (s : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓) → (s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓)
              → (a : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊' ⟧τ̊ C 𝔄) → (a' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊' ⟧τ̊ ℜ)
              → rel {Γ̊ , τ̊'} {τ̊} e e' (cls s a) (cls s' a')
              → rel {Γ̊} {τ̊} (e / sub vz a) (e' / sub vz a') s s'
```

The entire proof for this lemma can be found in the sources.

Relating environments Semantic equivalence is proven 'under related closing environments'. The fact that two closing environments are related is expressed by the following inductive datatype.

```
data Rel↓ : (Γ̊ : Ftx) → (⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓) → (⟦ Γ̊ ⟧Γ̊ ℜ ↓) → Set where
  ϵ   : Rel↓ ϵ ϵ ϵ
  cls : ∀ {Γ̊ τ̊} {e : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄} {e' : ⟦ Γ̊ ⟧Γ̊ ℜ ⊢ ⟦ τ̊ ⟧τ̊ ℜ}
      → {s : ⟦ Γ̊ ⟧Γ̊ C 𝔄 ↓} → {s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓} → Rel↓ Γ̊ s s'
      → (w : rel {Γ̊} {τ̊} e e' s s') → Rel↓ (Γ̊ , τ̊) (cls s e) (cls s' e')
```

Rel↓ is indexed by the closing environments it relates. For each two respective terms in the environment a witness w is evidence of the fact that both terms are related.

Relating transformation rules A type-and-transform transformation is only semantics preserving if the additional transformation rules preserve the transformation equivalence property. This is witnessed by the following construction, which establishes that each transformation rule in a rule set preserves the logical relation.

```
data RuleProofs : Rules 𝔄 ℜ → Set where
  ϵ     : RuleProofs ϵ
  proof : ∀ {rs τ̊} → RuleProofs rs → (e : ϵ ⊢ ⟦ τ̊ ⟧τ̊ C 𝔄)
        → (e' : ϵ ⊢ ⟦ τ̊ ⟧τ̊ ℜ) → (r : rel {ϵ} {τ̊} e e' ϵ ϵ)
        → RuleProofs (replace {𝔄} {ℜ} {τ̊} rs e e')
```

Equivalence proof We can now give give a formal proof of the Equivalence transformation property for TTS_λ. The proofs in this paragraph are slightly simplified by removing some of the book-keeping which is typical of proofs in intensional type theory. In intensional type theory every commutation between operators or equivalence has to be explicitly applied. The complete proofs can be found in the sources.

A program transformation is only semantics preserving for types which form a retraction and for transformation rules which are related. This is expressed by proving within a parametrized module:

```
module TransformationProof (abs-rep : abs ∘ rep βη-≡ id)
                           (proofs : RuleProofs rules) where
```

In the context of the Curry-Howard correspondence a module parameter can be seen as an assumption in logic. In this case we assume **abs** and **rep** to form a retraction and all the rules to adhere to the logical relation.

Within this parametrized module we can now show the fundamental theorem and extraction theorem. The fundamental theorem shows that a type and transform system transformation is logically related under related environments.

```
fundament : ∀ {Γ̊ τ̊ e e'} → (v : Γ̊ : τ̊ ⊨ e ⇝ e') → (s : ⟦ Γ̊ ⟧Γ̊ C ℜ ↓)
          → (s' : ⟦ Γ̊ ⟧Γ̊ ℜ ↓) → Rel↓ Γ̊ s s' → rel {Γ̊} {τ̊} e e' s s'
```

From this follows the corollary that a closing environment is related to itself. This follows from the fact that we can construct an identity transformation for each term in the closing environment. The fundamental theorem then shows that the transformation is related.

```
relΓ : ∀ {Γ} → (s : Γ ↓) → Rel↓ (Γ ↑Γ̊) s s
relΓ ϵ          = ϵ
relΓ (cls y y') = cls (relΓ y) (fundament (id⇝ y') _ _ (relΓ y))
```

The extraction lemma establishes that for base types the logical relation implies that the two related terms are $\beta\eta$-equivalent. This follows immediately form the definition of the logical relation. For base types the logical relation defines the $\beta\eta$-equivalence we are trying to prove.

```
extract : ∀ {Γ n} → (e e' : Γ ⊢ C n) → (s s' : Γ ↓)
        → rel {Γ ↑Γ̊} {C n} e e' s s' → close e s βη-≡ close e' s'
extract {Γ} {n} e e' s s' r = r
```

Using these three lemmas we can prove the final equivalence Theorem: proving that a complete transformation yields equivalent terms under any closing environment.

```
equivalence : ∀ {Γ n} → (e e' : Γ ⊢ C n) → (s : Γ ↓)
            → Γ ↑Γ̊ : C n ⊨ e ⇝ e' → close e s βη-≡ close e' s
equivalence e e' v s = extract e e' s s (fundament v _ _ (relΓ s))
```

The fact that the transformation is complete is enforced by the base type functor (**C** n) and the fact that the functor context should be lifted from some normal context Γ. This mechanically proves that the type and transform system is semantics preserving.

4.4 Monoid Transformation

Because the object language used for mechanically proving TTS_λ does not specify how types can be inhibited, it is not possible to express Hughes' strings transformation in this framework. However, it is possible to prove a more general transformation of which Hughes' strings is an instantiation: the monoid transformation.

A monoid is a mathematical structure consisting of a type **S** and a binary operation $\_\bullet\_ :: \mathbf{S} \to \mathbf{S} \to \mathbf{S}$ and an identity *zero* **::** **S**, which adhere to the following laws:

Associativity: $\forall\ a\ b\ c.\ a \bullet (b \bullet c) \equiv_{\beta\eta} (a \bullet b) \bullet c$
Identity: $\forall\ a.\ a \bullet \mathit{zero} \equiv_{\beta\eta} \mathit{zero} \bullet a \equiv_{\beta\eta} a$

It is not difficult to see that $(+\!\!+)$ and "" form a monoid for **String**s. We can construct a transformation for monoids which will make sure that all applications of the $\_\bullet\_$ function are *right-associative*, exactly what is being done in Hughes' string transformation.

The monoid transformation is parametrized by a monoidal structure: A type, the two operations and the accompanying laws. For this transformation we only need the right-identity of the $\_\bullet\_$ operation, so it is a bit more liberal than a normal monoid.

```
module TTS.Monoid (𝔄 : ℕ) (zero : ϵ ⊢ C 𝔄)
                  (_•_ : ϵ ⊢ C 𝔄 ⇒ C 𝔄 ⇒ C 𝔄)
                  (assoc :      ∀ {Γ} (a₁ a₂ a₃ : Γ ⊢ C 𝔄)
                           →    up _•_ · (up _•_ · a₁ · a₂) · a₃
                           βη-≡ up _•_ · a₁ · (up _•_ · a₂ · a₃))
                  (right-identity :  ∀ {Γ} (e : Γ ⊢ C 𝔄)
                                   → up _•_ · e · up zero βη-≡ e) where
```

Based on this monoid the ingredients for the transformation can be defined. The representation type $\mathfrak{R}$ is constructed as a function over the monoidal type, analogous to Hughes' strings, the **abs** and **rep** function are constructed accordingly. The transformation has just one rule which converts the binary operator into function composition.

```
ℜ : Ty
ℜ = C 𝔄 ⇒ C 𝔄

abs : ϵ ⊢ ℜ ⇒ C 𝔄
abs = Λ (v 0 · up zero)

rep : ϵ ⊢ C 𝔄 ⇒ ℜ
rep = Λ (up _•_ · v 0)

rules : Rules 𝔄 ℜ
rules = replace {𝔄} {ℜ} {ι ⟶ ι ⟶ ι} ϵ _•_ comp

import TTS.Judgement.Base
open TTS.Judgement.Base 𝔄 ℜ rep abs rules
```

This gives rise to a basic type and transform system for monoids. To show that this transformation preserves the semantics, we have to show that the **abs** and **rep** function form a retraction, and that the $\_\bullet\_$ to $\circ$ transformation preserves the logical relation.

```
abs-rep : abs ∘ rep βη-≡ id
abs-rep =
  let open Relation.Binary.EqReasoning βηsetoid
        renaming (_≈⟨_⟩_ to _↔⟨_⟩_)
  in begin
    _ ↔⟨ eval-comp _ _ ⟩
    _ ↔⟨ %Λ (%Λ (□ %· %≡ wk-up (vs vz) zero)
          %· (%Λ (%≡ wk-up (vs vz) _•_ %· □) %· □)) ⟩
    _ ↔⟨ %Λ (□ %· β ↔ (%≡ up-id _•_ _ %· □)) ⟩
    _ ↔⟨ %Λ β ⟩
    _ ↔⟨ %Λ (□ %· %≡ up-id zero _) ⟩
    _ ↔⟨ %Λ right-identity (var vz) ⟩
    _ ■
```

The rule for transforming _•_ to **comp** has the functor type $\iota \to \iota \to \iota$. The logical relation for this type expands to a function expecting and resulting in $\beta\eta$-equivalences. Proving can thus be done using equational reasoning.

```
m-rel : rel {ϵ} {ι ⟶ ι ⟶ ι} _•_ comp ϵ ϵ
m-rel a a' r a0 a1 r1 =
  let open Relation.Binary.EqReasoning βηsetoid
        renaming (_≈⟨_⟩_ to _↔⟨_⟩_)
  in begin
    _ ↔⟨ %Λ (%≡ merge-/ _•_ _ _ %· □) %· □ ⟩
    _ ↔⟨ β ⟩
    _ ↔⟨ %≡ merge-/ _•_ _ _ %· □ ⟩
    _ ↔⟨ bsym η ↔ %Λ (%≡ wk-up vz (_•_ · (_•_ · a · a0)) %· □) ⟩
    _ ↔⟨ %Λ assoc (up a) (up a0) (v 0) ⟩
    _ ↔⟨ %Λ (%≡ split-/ _•_ _ _ %· □ %· (%≡ split-/ _•_ _ _ %· □ %· □)) ⟩
    _ ↔⟨ %Λ (bsym β %· (bsym β %· □)) ⟩
    _ ↔⟨ %Λ (%Λ (bsym (%≡ wk-up _ _•_) %· □) %· bsym (%≡ wk-up _ a)
          %· (%Λ (bsym (%≡ wk-up _ _•_) %· □) %· bsym (%≡ wk-up _ a0) %· □))
          ⟩
    _ ↔⟨ bsym (eval-comp _ _) ⟩
    _ ↔⟨ %Λ (%≡ split-/ _•_ _ _ %· □)
          %· □ %∘ %Λ (%≡ split-/ _•_ _ _ %· □) %· □ ⟩
    _ ↔⟨ r %∘ r1 ⟩
    _ ■
proofs : RuleProofs rules
proofs = proof ϵ _•_ comp m-rel


open TransformationProof abs-rep proofs
```

We can use this to directly show the equivalence of transformed terms.

$$\mathbf{trans}_1 : (\epsilon\ ,\ \mathbf{C}\ \mathfrak{A}) : \mathbf{C}\ \mathfrak{A} \models \mathbf{up}\ \_ \bullet \_ \cdot \mathbf{v}\ 0 \cdot \mathbf{v}\ 0 \rightsquigarrow (\mathbf{up}\ \mathbf{abs} \cdot (\mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0 \circ \mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0))$$

$$\mathbf{trans}_1 = \mathbf{i\text{-}abs}\ (\mathbf{app}\ (\mathbf{app}\ (\mathbf{rule}\ (\mathbf{rule}\ \epsilon))\ (\mathbf{i\text{-}rep}\ (\mathbf{var}\ \mathbf{vz})))\ (\mathbf{i\text{-}rep}\ (\mathbf{var}\ \mathbf{vz})))$$

$$\mathbf{proof}_1 : (\mathsf{s} : (\epsilon\ ,\ \mathbf{C}\ \mathfrak{A})\downarrow) \rightarrow \mathbf{close}\ (\mathbf{up}\ \_ \bullet \_ \cdot \mathbf{v}\ 0 \cdot \mathbf{v}\ 0)\ \mathsf{s} \beta\eta\text{-}\equiv \mathbf{close}\ (\mathbf{up}\ \mathbf{abs} \cdot (\mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0 \circ \mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0))\ \mathsf{s}$$

$$\begin{aligned}
&\mathbf{proof}_1 = \mathbf{equivalence}\ \{\epsilon\ ,\ \mathbf{C}\ \mathfrak{A}\}\ \{\mathfrak{A}\} \\
&\quad (\mathbf{up}\ _ \bullet _ \cdot \mathbf{v}\ 0 \cdot \mathbf{v}\ 0) \\
&\quad (\mathbf{up}\ \mathbf{abs} \cdot (\mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0 \circ \mathbf{up}\ \mathbf{rep} \cdot \mathbf{v}\ 0)) \\
&\quad \mathbf{trans}_1
\end{aligned}$$

4.5 Discussion

Extensionality The equivalence proof shows that the source and result term from a complete type-and-transform deduction are $\beta\eta$-equivalent *for all possible substitutions*. Earlier we have seen that the simply typed lambda calculus is extensional: if two terms are equivalent for all possible inputs, we can treat those two terms as equal. Thus we should not only be able to deduce that the terms are equal 'up to substitution', but we should be able to deduce a direct equivalence:

$$\begin{aligned}\textbf{strengthen} : \forall\,\{\Gamma\ \mathsf{n}\} \to (\mathsf{e}\ \mathsf{e'} : \Gamma \vdash \mathbf{C}\ \mathsf{n}) \to (\mathsf{s} : \Gamma\downarrow)\\ \to \textbf{close}\ \mathsf{e}\ \mathsf{s}\ \beta\eta\text{-}\equiv \textbf{close}\ \mathsf{e'}\ \mathsf{s} \to \mathsf{e}\ \beta\eta\text{-}\equiv\ \mathsf{e'}\end{aligned}$$

Although this is a valid statement, it is not possible to prove this directly Agda. Agda does not support a general extensionality lemma, and thus we can not simply deduce the equivalence, although it is correct in this particular case of the simply typed lambda calculus.

Structural Logical Relations One way out of this would be to construct a separate, more restrictive logic in which extensionality does hold. This is the approach taken by Schürmann and Sarnat [SS08]. They use logical relations to prove properties about the simply typed lambda lambda calculus in Twelf. To overcome the limits of Twelf's logic they express the logical relation in a *separate logic* defined within Twelf. This separate logic is carefully constructed to hold properties which are not true in Twelf in general. They call this approach structural logical relations.

This same approach can be taken here. Instead of expressing the logical in Agda itself, it can be expressed in a stronger (limited) logic defined *within* Agda, which preserves extensionality.

Extended types This approach may also be used prove or disprove a bit more liberal version of the type-and-transform system. We strongly suspect that the transformation equivalence property does not only hold for terms with *base types* but for hole-free types in general, including function types. However, the logical relation uses Agda's function space to relate function types and it is not possible to eliminate function space without an argument in a language which has no extensionality lemma. The approach of Structural logical relations might be a solution to this.

5 Extensions to the TTS System

Although the simply typed lambda calculus is a convenient language with which to explain ideas and prove general concepts, it is too simple to serve as a basis for real-world transformation systems. Most notably, STLC does not support recursion or polymorphism, two constructs which are present in almost all real-world functional languages.

This chapter introduces such fundamental extensions for the basic TTS_λ along several axis. The soundness of these extensions is not proven here but is informally shown and provability is briefly discussed.

5.1 Extending to Let-Polymorphism

One of the most common extensions to the simply typed lambda calculus is let-polymorphism with Hindley-Milner typing as found in ML and Haskell. This extension introduces the concept of type variables into the language and a way to quantify over them, as in the following definitions:

$$\begin{aligned}
\tau &::= \mathbf{T} \mid \mathsf{a} \mid \tau \to \tau \\
\sigma &::= \forall\, \mathsf{a}.\, \sigma \mid \tau \\
\Gamma &::= \Gamma\,,\, \sigma \mid \emptyset \\
\theta &::= [\mathsf{a} \mapsto \tau] \circ \theta \mid \mathbf{id}
\end{aligned}$$

The typing functor of TTS_λ is essentially an extension of the base types, and thus works equally well in the presence of type variables:

$$\begin{aligned}
\mathring{\tau} &::= \mathbf{T} \mid \iota \mid \mathsf{a} \mid \mathring{\tau} \to \mathring{\tau} \\
\mathring{\sigma} &::= \forall\, \mathsf{a}.\, \mathring{\sigma} \mid \mathring{\tau} \\
\mathring{\Gamma} &::= \mathring{\Gamma}\,,\, \mathsf{x} : \mathring{\sigma} \mid \emptyset \\
\mathring{\theta} &::= [\mathsf{a} \mapsto \mathring{\tau}] \circ \mathring{\theta} \mid \mathbf{id}
\end{aligned}$$

Hindley-Milner typing prescribes two functions which can be used to introduce and eliminate quantification, $\mathsf{gen} :: \Gamma \to \tau \to \sigma$ and $\mathsf{inst} :: \sigma \to \theta \to \tau$. gen quantifies over all type variables in τ except the variables which occur free in the environment Γ. inst creates a normal type from a schema and a substitution θ for all quantified variables. These function can be straightforwardly extended to typing functors by treating the ι construct as a normal base type in both functions.

The terms in the let-polymorphic lambda calculus are extended with a **let** construct which guides generalization and instantiation, this gives the following term definitions:

$$\mathsf{e} ::= \mathsf{x} \mid \mathsf{c} \mid \mathsf{e}\ \mathsf{e} \mid \backslash \mathsf{x}.\ \mathsf{e} \mid \mathbf{let}\ \mathsf{x} = \mathsf{e}\ \mathbf{in}\ \mathsf{e'}$$

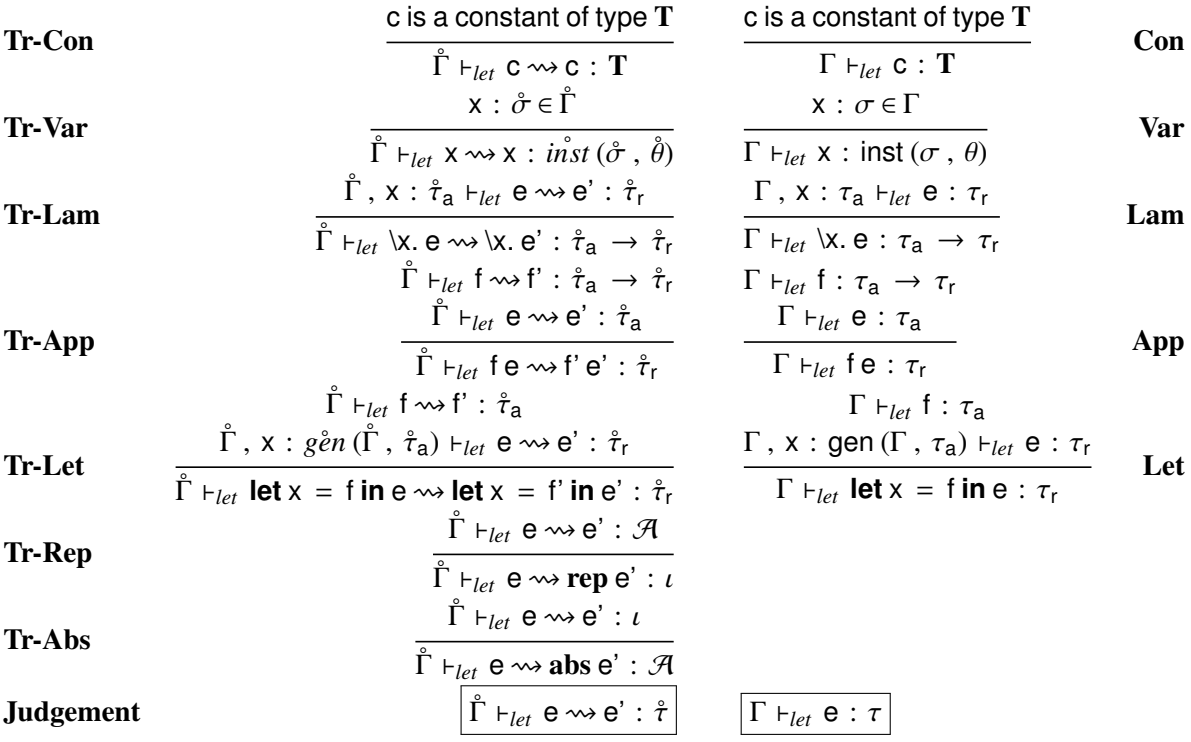

Figure 5.1: Type checking rules for the propagation relation of TTS **let**

Figure 5.1 shows the typing rules for the let-polymorphic lambda calculus and the corresponding transformation rules for TTS **let**, the core transformation system. This extension is again a straightforward extension of the basic Hindley-Milner typing rules. Proving this transformation correct however is more difficult.

Informally we reason that, although the language now contains polymorphism, the type-and-transform system only makes changes to the base types. Thus, type variables will be left alone during transformation. Type instantiations can now also instantiate to the hole type, but this is no problem because a polymorphic function can be instantiated with any type.

5.2 Including a Fixpoint

Although the simply typed lambda calculus serves well as a foundation for a typed program transformation system, it has one important shortcoming: it is not Turing complete. A language needs some form of unbridled recursion to become Turing complete. A well-known construct for recursion is the fixpoint. A fixpoint works by taking a function and turning it into a possibly infinite self-application. An example of a fixpoint is the Y combinator, but other variants exist as well.

```
fix :: (a → a) → a
fix f = (\x → f (x x)) (\x → f (x x))
```

Note that this expression does not type-check in Haskell. There are ways to make such a primitive recursion operator type-check but usually the fixpoint is an internal part of a language. Programs containing general recursive functions are compiled into a version containing only fixpoints as recursion primitive. A typing rule and propagation rule for a primitive **fix** can be constructed as follows:

$$\textbf{Tr-Fix}\quad \frac{\mathring{\Gamma} \vdash_\lambda \mathsf{f} \rightsquigarrow \mathsf{f'} : \mathbf{T} \to \mathbf{T}}{\mathring{\Gamma} \vdash_\lambda \mathbf{fix}\,\mathsf{f} \rightsquigarrow \mathbf{fix}\,\mathsf{f'} : \mathbf{T}} \qquad \frac{\Gamma \vdash_\lambda \mathsf{f} : \mathbf{T} \to \mathbf{T}}{\Gamma \vdash_\lambda \mathbf{fix}\,\mathsf{f} : \mathbf{T}}\quad \textbf{Fix}$$

Because **fix** is a primitive function, it has a separate convertibility rule, μ, which states that one of the function applications can be peeled off the infinite chain of self-applications.

$$\begin{aligned}\mu\quad &: \forall\,\{\tau\} \to \{\mathsf{f} : \Gamma \vdash \tau \Rightarrow \tau\} \to \mathbf{fix}\,\mathsf{f}\,\beta\eta\text{-}\equiv\,\mathsf{f}\cdot\mathbf{fix}\,\mathsf{f}\\ \%\mathsf{fix}\;&: \forall\,\{\tau\} \to \{\mathsf{f}\,\mathsf{f'} : \Gamma \vdash \tau \Rightarrow \tau\} \to \mathsf{f}\,\beta\eta\text{-}\equiv\,\mathsf{f'} \to \mathbf{fix}\,\mathsf{f}\,\beta\eta\text{-}\equiv\,\mathbf{fix}\,\mathsf{f'}\end{aligned}$$

While this seems like a straightforward enough extension to the basic TTS_λ system, the fixpoint has some implications on the semantics of a language, and thus on the way proofs are conducted for that language.

Impact on evaluation The main problem is that a fixpoint introduces possibly infinite recursion: divergent terms. This means that, next to the normal values which a term can produce, a term can also produce $\bot$: no value at all! In such a language, the order of evaluation becomes important.

The lambda calculus without fixpoint is *strongly normalizing*, meaning that however evaluation is performed, the answer will always be the same. However, with the presence of $\bot$, the order of evaluation can have an impact on the result of evaluation. This is best illustrated with an example:

$$\mathbf{v} = \mathbf{take}\;10\;(\mathbf{fix}\;(1:))$$

fix (1 :) produces an infinite list of 1's. The outcome of this function now really depends on what is evaluated first. If the arguments to a function are evaluated before a function is called, this program will loop forever. If, however, functions are being reduced before their arguments, this program will produce a list of 10 1's. The first type of evaluation is called call-by-value or strict evaluation, the second is called call-by-need or lazy evaluation.

This example shows that lazy evaluation can sometimes produce an answer when strict evaluation can not. This is because lazy evaluation has a very nice property which strict evaluation lacks: if a term can produce a value (has a *normal form*), lazy evaluation will compute this value.

Convertibility The fact that the evaluation order impacts the final outcome of an expression also impacts equational reasoning and the convertibility relation. Take for example the following equality:

$$\mathbf{const}\;1\;(\mathbf{fix}\;(\mathbf{:1})) \equiv_{\beta\eta} 1$$

In a lazy semantics this would be a valid equality. In a strict semantics this is unequal: **fix** (**:1**) has to be evaluated be evaluated before **const** and will thus result in $\bot$ instead of 1. In general, the convertibility relation is only sound for semantics which use *lazy evaluation*. This also means that the proofs in the type-and-transform system are only valid for a language with lazy semantics.

Totality Convertibility can be made sound in a strict language by restricting the **fix** function to functions which provably never produce $\bot$: they never loop. A provably finite recursive function is called a *total* function. Agda is an example of a language which allows equational reasoning because, although it has strict semantics, it requires totality on its functions. Note that limitation again removes the turing completeness from a language.

6 Future Work

The research in this work presents a formal system for program transformation on a very high level, but a lot of work still needs to be done before this idea can be applied in the real world. The work that still has to be done can roughly be separated into three main areas: extensions for the object language, extensions of the transformation system itself and working toward an implementation. The upcoming sections will highlight some questions that may be beneficial to each of these areas.

6.1 Language Extensions

Although the most vital language extensions of recursion and polymorphism are covered in this work, there are many more features which exist in real-world programming language. How each of these features can be adapted to work with type-and-transform systems is a big source of future work.

Parametrized datatypes Most functional programming languages allow types to be parametrized by other types. This poses two important challenges. First off, types may be changed within the parameters of a parametrized data type. Furthermore, one would want to change parametrizable types, such as the **List** a and **Stream** a in the stream fusion example. In this case the **List** should be changed to a **Stream** regardless of the type parameter.

How transformations for such datatypes can be done and for what datatypes the semantics can be preserved is still an open question. Data types can have many features in themselves, such as nested data types and GADTs and it is not clear how these features interact with the type-and-transform system. It could be that simply maintaining the retraction property is enough to guarantee the equivalence properties on such datatypes, but this is not yet proven.

Type Classes Type classes language feature allow the implementation of polymorphic functions to be determined by the type with which they are used. Changing a type here would change behaviour of the polymorphic function, something which does usually not happen. How to account for this in the context of type-and-transform systems has yet to be researched.

Let-polymorphism proof Although we did a proof by handwaving for the let-polymorphic lambda calculus, this proof should also be formalized. Johann and Voigtländer [JV09] give a logical relation for Haskell's underlying base language Core, which is based on the polymorphic lambda calculus. Based on this work it should be possible to prove or disprove the correctness of the let-polymorphic type-and-transform system.

6.2 Transformation System Extensions

Generic Transformation System We have seen that a type-and-transform system can readily be derived from the object language, as long as it has a typing system and has well-defined deduction rules. It may be possible to mathematically formulate the mechanism with which a type-and-transform system can be constructed. This would require a structural characterization of the eligible object languages and a procedure with which the type-and-transform system can be derived. This might even lead to a generic proof of correctness for all derivable programming languages.

Transforming Multiple Types The type-and-transform system presented in this work is limited to the transformation of one type only. It would be interesting to research the possibility of changing multiple types at once. One way this extension could be done is by allowing multiple source types for one result type. This would allow a wider class of transformations, in which multiple types can be transformed into one type. For example, unifying all string types in a program to one representation.

Another way to allow multiple types to be transformed is by having multiple source-result type pairs. This could be beneficial for composing multiple independent transformations together into one simultaneous transformation pass, or rewriting functions involving multiple transformed types.

In both these extensions the interaction between type changes and transformation has to be carefully researched. The mayor question here is if, and how, the concept of retracts can be generalized to multiple types.

6.3 Implementation

When looking at the implementation of type-and-transform systems there are two main topics to be researched: developing efficient algorithms for transformation and integrating the transformation system into existing infrastructure in a user-friendly and maintainable way.

Efficient implementation Naively generating all possible transformation results will result in a slow transformation system. Developing efficient algorithms and heuristics is of essential importance for a real-world application. Not only should transformation be fast, but transformation should also produce the 'best' of all possible results. How to perform good and fast transformations is ongoing research by Sean Leather.

GHC Core-To-Core Transformations Recent version of the GHC Haskell compiler allow the user to specify transformation passes. These transformation passes are performed on a typed intermediate language called Core. The Core language is designed to be a simple, desugared functional language upon which optimizations and transformations can be performed, which makes it an ideal candidate for implementation of a type-and-transform system. The Core language has some specific characteristics such as type coercions which should first be researched in the context of type-and-transform systems.

Monoid Transformation The monoid transformation could implemented as a separate transformation to optimize the evaluation of monoidal structures. The monoidal transformation can take an arbitrary binary operation and transform a program such that the binary operator is only applied to its left or right argument. A simple user annotation could specify in which way the application nesting should be transformed, such as a fixity declaration:

infixl 5 $(+\!\!+)$

7 Conclusion

In this work we have presented a formal system for type-changing program transformations. The transformation system is build on top of the simply typed lambda calculus and allows the transformation of a single type into another type while rewriting the operations involving those types. Keeping track of type changes is done by typing the source and result simultaneously using the typing functor, which represents type changes in the programs with a special 'hole' construction. The system has been mechanically proven in Agda to preserve the semantics of the source program after transformation.

Although the simply typed lambda calculus may seem too simple to represent a real-world language, it shows that the type-and-transform system can deal with essentials of functional programming: abstraction and application. We are confident that type-and-transform systems and the simultaneous typing technique using typing functors can be readily adapted to more advanced language features, of which we have shown fixed-point recursion and polymorphism.

We argue that type-and-transform systems form a solid yet simple paradigm to construct and prove program transformations. Because the transformation rules and typing functor are simply derived from the underlying object language the transformation system can be easily understood.

What the performance of real-world implementations of the type-and-transform systems will be has yet to be seen. The system will ultimately rely on some form of proof search which can be slow to compute. Whether this will be a problem can only be found out by making an implementation.

Bibliography

[AB11] Amal Ahmed and Matthias Blume. An equivalence-preserving cps translation via multi-language semantics. In *ICFP*, pages 431–444, 2011.

[BB02] Kevin Backhouse and Roland Backhouse. Logical relations and galois connections. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, MPC '02, pages 23–39, London, UK, UK, 2002. Springer-Verlag.

[CH07] Karl Crary and Robert Harper. Syntactic logical relations for polymorphic and recursive types. *Electron. Notes Theor. Comput. Sci.*, 172:259–299, April 2007.

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

[COV06] Alcino Cunha, José Nuno Oliveira, and Joost Visser. Type-safe two-level data transformation. In *Proceedings of the 14th international conference on Formal Methods*, FM'06, pages 284–299, Berlin, Heidelberg, 2006. Springer-Verlag.

[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.

[CV07] Alcino Cunha and Joost Visser. Strongly typed rewriting for coupled software transformation. *Electron. Notes Theor. Comput. Sci.*, 174(1):17–34, April 2007.

[ER07] Martin Erwig and Deling Ren. An update calculus for expressing type-safe program updates. *Sci. Comput. Program.*, 67(2-3):199–222, 2007.

[GH09] Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner Type Inference Algorithms. Technical report, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2002.

[Hin00] Ralf Hinze. *Generic Programs and Proofs*. PhD thesis, Bonn University, 2000.

[HS99] Furio Honsell and Donald Sannella. Pre-logical relations. In *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*, CSL '99, pages 546–561, London, UK, UK, 1999. Springer-Verlag.

[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.

[JV04] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. *SIGPLAN Not.*, 39(1):99–110, January 2004.

[JV09] Patricia Johann and Janis Voigtländer. A family of syntactic logical relations for the semantics of haskell-like languages. *Inf. Comput.*, 207(2):341–368, 2009.

[KA10] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[KJ12] Steven Keuchel and Johan T. Jeuring. Generic conversions of abstract syntax representations. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming*, WGP '12, pages 57–68, New York, NY, USA, 2012. ACM.

[Kuc97] Jakov Kucan. Metatheorems about convertibility in typed lambda calculi: Applications to cps transform and "free theorems". Technical report, 1997.

[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.

[Mit96] John C. Mitchell. *Foundations for programming languages*. Foundation of computing series. MIT Press, 1996.

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

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[pop] The poplmark challenge. `http://www.seas.upenn.edu/~plclub/poplmark/`.

[pro] Program-transformation.org. `http://http://www.program-transformation.org/`.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[sou] Source code. `https://github.com/craffit/thesis`.

[SS08] Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In *LICS*, pages 69–80, 2008.

[str] Stratego/xt. `http://strategoxt.org/`.

[Swi12] Wouter Swierstra. From mathematics to abstract machine: A formal derivation of an executable krivine machine. In *MSFP*, pages 163–177, 2012.

[VeaB98] Eelco Visser and Zine el-abidine Benaissa. A core language for rewriting. In *Electronic Notes in Theoretical Computer Science*, pages 1–4. Elsevier, 1998.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.