Type-Changing Rewriting and Semantics-Preserving Transformation

Sean Leather

Johan Jeuring

Andres Löh

Bram Schuur

Technical Report UU-CS-2013-017
October 2013

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Type-Changing Rewriting and Semantics-Preserving Transformation

Sean Leather
Utrecht University
s.p.leather@uu.nl

Johan Jeuring
Utrecht University
Open University Netherlands
j.t.jeuring@uu.nl

Andres Löh
Well-Typed LLP
andres@well-typed.com

Bram Schuur
Utrecht University
b.schuur@uu.nl

Abstract

We have identified a class of regular, whole-program transformations that cannot be safely performed with typical transformation techniques because transformation requires changing the types of terms. In these transformations, we want to change typically large parts of a program from using one type to using another type while simultaneously preserving the original program semantics after transformation.

In this paper, we present type-and-transform systems, an automated approach to the whole-program transformation of terms involving type A to terms involving the isomorphic type B using type-changing rewrite rules. Type-and-transform systems establish typing and semantics relations between all source and target subprograms such that a complete transformation can guarantee the equivalent semantics of a whole program. We describe the type-and-transform system for the lambda calculus with let-polymorphism and general recursion, including several examples from the literature and properties of the system.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory

Keywords type-and-transform systems, type-changing rewriting, semantics-preserving program transformation

1. Introduction

Program improvement sometimes involves large, homogeneous changes that are not intended to modify program functionality (other than, perhaps, performance). For example, a programmer might rename variables, reorganize code, or update code to use a new library API. Of course, these changes can still introduce unwanted errors into a program. Consequently, programmers often use tools to help automate common patterns of change such as refactoring [8]. Compilers or interpreters may also be employed for large changes such as optimization without necessitating programmer intervention. In functional programming, term rewriting [1] can be used to safely change programs with simple rewrite rules.

Many approaches to automated semantics-preserving program improvement only allow type-preserving updates to code. This is only natural: in a statically typed programming language, type safety is a prerequisite for a working program. Replacing one term with another of a different type challenges the effort of guaranteeing the preservation of semantics between the terms. Some type-changing rewrites may be straightforward: adding a parameter to a function, for example. Other changes are not obvious: changing one string type to a different string type, in which the APIs of the two types are not equivalent. A completely transformed program should work as before, i.e. the strings are still strings. However, the evaluation may now be more efficient. Or, for example, the program now supports Unicode characters whereas before the encoding was ASCII. Our focus is the class of transformations between isomorphic types with possibly different APIs.

In this paper, we discuss a foundation for certain automated semantics-preserving and type-changing program transformations. We use purely functional programming languages with strong, static type systems. Such languages allow us to utilize the type system for safety as well as driving change throughout the program. By disallowing or isolating side effects, such languages also simplify the proof of semantics preservation. Our object language is the lambda calculus with let-polymorphism and general recursion (a.k.a. the Hindley-Milner type system).

A *type-and-transform system* defines, for a given language, how to relate two programs such that all "unresolved" term and type changes are identified and can (eventually) be resolved resulting in the programs being semantically equivalent. A type-and-transform system specifies the structure of a *transformation*[1] that relates one typed program (the source) to another (the target). The target is actually the (possibly) modified source. A type-and-transform system also specifies how a program can be modified with a *typed*

[1] To avoid confusing readers from different backgrounds, we point out that we have hijacked the terms "transformation" and "rewrite," among others, and substituted specialized meanings.

$$\begin{tikzcd}[column sep=large]
\text{source} & \text{target} \\
e_s : \tau_s \arrow[r, squiggly, "\text{transformation}"] \arrow[rd, squiggly] & e_l : \tau_l \arrow[d, squiggly, "\text{rewriting}"] \\
 & e_r : \tau_r
\end{tikzcd}$$

Figure 1. Diagram of transformation and rewriting

rewrite rule, an extension of the usual rewrite rule that can, under certain conditions, impose a change of type between its left-hand side (lhs) and right-hand side (rhs) patterns.

The structure of a transformation mirrors the structure of the source term, preserving both the syntactic relation of corresponding source and target subterms and the typing relations of each subterm. A transformation also records any modifications that arise when a typed rewrite rule is applied to a target. If the source and initial target are related by a transformation, then we can construct a new transformation relating the source and new target. A *complete transformation*[2] relates two programs with the same type and equivalent semantics, even though the programs may differ syntactically.

Figure 1 provides a visualization of the connections between transformation and rewriting. Here, e_s is the source term, and τ_s is the type of e_s. The subscripts l and r indicate the target terms and types matching the respective lhs and rhs of a rewrite rule. The lower frequency oscillating vertical arrow represents rewriting, while the higher frequency arrows represent transformations.

A set of typed rewrite rules describes all the allowed changes between the terms of two types, $\mathcal{A}$ and $\mathcal{R}$.[3] The conversion between types is given by the functions $rep : \mathcal{A} \to \mathcal{R}$ and $abs : \mathcal{R} \to \mathcal{A}$.

In this paper, we focus on types $\mathcal{A}$ and $\mathcal{R}$ that are isomorphic. That is, both of the following equivalences hold:

$$rep \circ abs \equiv id_{\mathcal{R}\to\mathcal{R}} \qquad (\textit{rep-abs})$$

$$abs \circ rep \equiv id_{\mathcal{A}\to\mathcal{A}} \qquad (\textit{abs-rep})$$

This simplifies the proof of semantics, but it also means many type pairs are not supported.

We believe the conversion requirement can be weakened to a retract (i.e. only (*abs-rep*) is necessary). This would, for example, allow transformations between the types $\mathcal{A} = String$ and $\mathcal{R} = String \to String$. One of the authors has already shown this to some extent. In his master's thesis, Schuur [26] demonstrated a type-and-transform system for the simply typed lambda calculus using a logical relation as proof technique. There is precedence [24] for using logical relations with more interesting languages such as ours, which has general recursion and polymorphism. We will explore this in future work, but we feel that this paper stands well on its own as a basis for type-and-transform systems.

1.1 An Application of Type-and-Transform Systems

To motivate type-and-transform systems, we present an application that will also serve as a running example. Pat, the programmer, will guide us through the general scenario presenting the problem and possible solutions. For our code examples, we use Haskell notation.

Scenario Pat writes a program using type A. The operations involving A (A-terms) are convenient for programming, but programming with A can produce inefficient programs, and Pat discovers the program has this problem. Fortunately for Pat, another type B is isomorphic to A and more efficient but not as convenient (e.g. the set of B-terms is smaller or the code is more verbose). Unfortunately, replacing A-terms with B-terms or inserting conversions at all the right places is time-consuming and error-prone.

[2] This is not related to "completeness" but rather to a subset of transformations obeying certain properties described in Section 6.

[3] The abbreviations $\mathcal{A}$/*abs* and $\mathcal{R}$/*rep* refer to abstraction and representation, respectively, in deference to Hughes [17].

```
newtype Z = Z (S → S)          type S = String   -- for brevity
(◇) :: Z → Z → Z                rep :: S → Z
Z f ◇ Z g = Z (f ∘ g)           rep xs = Z (xs++)
ϵ :: Z                          abs :: Z → S
ϵ = Z id                        abs (Z f) = f ""
```

Figure 2. Difference strings library

Pat can attempt to solve the problem using a type-and-transform system to automatically transform the program with one of two potential approaches:

1. Pat uses a compiler flag. The compiler "knows" about the A-to-B transformation and converts A-terms to B-terms, safely and completely. In the meantime, Pat will continue to utilize A-terms, comfortable in the knowledge that the compiler will optimize[4] them to B-terms.
2. Pat uses an IDE component. After the operation – which could be considered a form of refactoring – Pat uses the newly transformed code with B-terms instead of A-terms.

Typical Examples The canonical example is transforming lists to an alternative representation (sometimes called difference lists) as first-class functions on lists [17]. In a similar vein, cons-lists can be replaced by join-lists [27, 33] or finger trees [16]. There are also multiple string types, each with a different application: people in the Haskell community often encounter problems using *String* (a synonym for $[Char]$) when they should be using *ByteString* [3] or *Text* [14].

Example: Difference Strings Substantial use of the standard Haskell "append" operation $+\!\!+$ on lists can be problematic since left-bracketed associations, such as $(xs +\!\!+ ys) +\!\!+ zs$, are inefficient: xs is traversed twice during evaluation. Even though $+\!\!+$ is right-associative, we cannot always easily guarantee that $+\!\!+$ is used in a right-bracketed way, especially when abstraction is used, e.g. as in **let** $as = xs +\!\!+ ys$ **in** $as +\!\!+ zs$.

We paraphrase the Hughes [17] solution to the append problem with two adaptations. First, we specialize the code from lists to lists of chracters (i.e. *String*). This simplifies our initial presentation of type-and-transform systems. We revisit the example with lists in Section 7.4. Second, we use a Haskell **newtype** for the difference list string type Z, and we assume that Z is abstract outside this library. This ensures that we have an isomorphism. The code for the library is in Figure 2. The isomorphism with the standard *String* type is implemented by *rep* and *abs*.

With the Z library, we can now give some example transformation targets. The simple $(xs +\!\!+ ys) +\!\!+ zs$ becomes $abs\ ((rep\ xs \diamond rep\ ys) \diamond rep\ zs)$ – xs is no longer traversed twice. A transformation can push change through bindings: the above **let** example can be transformed to abs (**let** $as = rep\ xs \diamond rep\ ys$ **in** $as \diamond rep\ zs$) – note that as has a new type. Lastly, borrowing an example from Hughes [17], we can transform the reverse function (specialized to strings):

```
rev :: S → S
rev ""       = ""
rev (x : xs) = rev xs ++ (x : "")
```

[4] To clarify, transformation does not guarantee improvement, but it does expedite comparing transfomed and untransformed programs.

Terms	e, f	$::= x \mid f\ e \mid \lambda x.e \mid \mathbf{fix}\ e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$
Types	τ, υ	$::= \alpha \mid B \mid \tau \to \upsilon$
Type Schemes	ς	$::= \forall\bar{\alpha}.\tau$
Environments	Γ	$::= \varepsilon \mid \Gamma, \nu : \varsigma$
Variables	ν	$::= x \mid m$

Figure 3. Object language syntax

There are multiple possible targets (and we discuss how to choose one in Section 7.2), but, assuming *rev* is found in a larger program, the most likely tarket is:

$$\begin{aligned}
&rev' :: S \to Z\\
&rev'\ \texttt{""} \quad\ = \epsilon\\
&rev'\ (x : xs) = rev'\ xs \diamond rep\ (x : \texttt{""})
\end{aligned}$$

This version (renamed for presentation) allows the Z type to propagate further through the uses of rev'. Alternatively, every $rev\ e$ can be transformed to $abs\ (rev'\ e)$.

As a programming abstraction, the difference string representation is clearly not as convenient as the string representation. Optimization may not be a concern early in the development cycle, and the simplicity of strings can be a strong motivating factor. But later, inefficiency can become a significant problem, and the transformation to difference strings becomes very useful.

Gill and Hutton [13] also use *rev* to demonstrate the worker/wrapper transformation; however, their work differs from ours in at least two respects. First, we describe fully automated transformations, while the worker/wrapper transformation is a manual proof technique. Second, the scope of a worker/wrapper transformation is a recursive function, while the scope of our transformation is determined by an arbitrary binding, i.e. anything from a local definition to a module or even the whole program.

1.2 Contributions

The contributions of this paper are the following:

- We describe the type-and-transform system for the lambda calculus with let-polymorphism and general recursion.
- We establish and prove the properties necessary for type safety and preserving semantics.
- We discuss the transformation algorithm, choice heuristics, and extending to parameterized types.

We have also developed a monadic Haskell implementation[5] of the algorithm. Note that our primary focus is on the theory, and we do not discuss many practical aspects. We plan to extend the theory to Haskell and investigate the real-world efficiency and effectiveness of type-and-transform systems.

1.3 Overview

The remainder of this paper is organized as follows. We begin in Section 2 with a discussion of the object language, including the type system and semantics. In Section 3, we look at a few example transformations to develop an intuitive understanding. We dive into type-and-transform systems by introducing the typing and semantics in Sections 4 and 5, respectively. In Section 6, we present the formal definitions and correctness proofs of the important concepts. Section 7 follows up with discussions of a transformation algorithm, transformation choice heuristics, parameterized type constructors, and the transformation of lists to difference lists. We describe two more applications of type-and-transform systems in Section 8. Finally, we examine related work in Section 9 and conclude with our future plans in Section 10.

[5] A `cabal` package is available at `http://www.staff.science.uu.nl/~leath101/code/tts-0.3.2.tar.gz`.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\tau \prec \Gamma(x)}{\Gamma \vdash x : \tau}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ e : \tau}\ (\textsc{Fix})$$

$$\frac{\Gamma \vdash f : \tau \to \upsilon \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f\ e : \upsilon}\ (\textsc{App})$$

$$\frac{\Gamma, x : \tau \vdash e : \upsilon}{\Gamma \vdash \lambda x.e : \tau \to \upsilon}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \mathcal{G}_\Gamma(\tau) \vdash e_2 : \upsilon}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \upsilon}\ (\textsc{Let})$$

Figure 4. Object language type system

2. Object Language

Our object language is the lambda calculus with general recursion (a **fix** primitive) and polymorphic **let**-bindings with the Hindley-Milner type system [5, 22]. It is a small language but interesting enough for useful examples.

Figure 3 gives the grammar for the language. The term syntax is standard. For readability, we borrow features from Haskell such as infix binary operators and list notation, but all examples can easily be translated to the core language.

A type τ is either a type variable α, a base type B (e.g. integer or string), or a function type. We use $^{\alpha}/_{B}$ as a shortcut for either a type variable or a base type later in the paper. A type scheme ς quantifies over a vector $\bar{\alpha}$ of type variables in a type. If $\bar{\alpha}$ is empty, we write the type scheme as a type.

Type environments are finite maps from variables to type schemes. A type environment is either empty or the union of an environment Γ with $\{\nu : \varsigma\}$, where ν does not occur free in Γ. We use α-renaming where necessary to avoid shadowing. In later sections, we distinguish metavariables (m) from object variables (x), but the environment domain is the sum of the two. The notation $\varsigma = \Gamma(x)$ indicates that $x : \varsigma \in \Gamma$.

A type substitution[6] σ is a finite map from type variables to types. A substitution that replaces $\alpha_1, \ldots, \alpha_n$ with $\tau_1, \ldots, \tau_n$ is written as $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$. The empty substitution is written as *id*, and the composition of two substitutions σ_1 and σ_2 is $\sigma_1 \circ \sigma_2$. We indicate the application of a substitution σ to a type τ by juxtaposition: $\sigma\tau$. Substitution uses α-renaming where necessary to avoid capture.

Instantiation and generalization are defined as follows:

- A type τ is an instance of a type scheme $\varsigma = \forall\bar{\alpha}.\tau'$ if there exists a substitution σ, whose domain is a subset of $\bar{\alpha}$, such that $\tau = \sigma\tau'$. We write instantiation as $\tau \prec \varsigma$.
- The closure $\mathcal{G}_\Gamma(\tau)$ of the type τ under the environment Γ is defined as (where $fv(x)$ means the free variables of x):

$$\mathcal{G}_\Gamma(\tau) = \forall\bar{\alpha}.\tau \ \text{ where }\ \bar{\alpha} = fv(\tau) \setminus fv(\Gamma)$$

The typing judgment $\Gamma \vdash e : \tau$ says that the term e, closed by the type environment Γ, has the type τ. The inference rules are given in Figure 4.

For the language semantics, we use the following equivalences:

[6] We later use forms of substitution for mapping things other than types, but the notation remains the same.

Source	Target	
$\texttt{"a"} : S$	$\mathit{rep}\ \texttt{"a"} : Z$	(1)
$+\!\!+ : S \to S \to S$	$\diamond : Z \to Z \to Z$	(2)
$x +\!\!+ \texttt{"b"} : S$	$x \diamond \mathit{rep}\ \texttt{"b"} : Z$	(3)
$(\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} : S$	$\mathit{abs}\ ((\lambda x.x \diamond \mathit{rep}\ \texttt{"b"})\ (\mathit{rep}\ \texttt{"a"})) : S$	(4)
$(\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} : S$	$\mathit{abs}\ ((\lambda x.\mathit{rep}\ x \diamond \mathit{rep}\ \texttt{"b"})\ \texttt{"a"}) : S$	(5)
$(\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} : S$	$(\lambda x.\mathit{abs}\ (\mathit{rep}\ x \diamond \mathit{rep}\ \texttt{"b"}))\ \texttt{"a"} : S$	(6)

Table 1. Examples of transformations

$$(\lambda x.e_2)\ e_1 \equiv [x \mapsto e_1]e_2 \qquad \text{(RED-LAM)}$$

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \equiv [x \mapsto e_1]e_2 \qquad \text{(RED-LET)}$$

$$\textbf{fix}\ (g \circ f) \equiv g\ (\textbf{fix}\ (f \circ g)) \qquad \text{(RED-ROLLING)}$$

The first two are standard reduction rules for a call-by-name semantics. The last equivalence, (RED-ROLLING), is the *rolling rule* for the least fixed point [2, 13].

3. A Brief Look at Transformation

In this section, we look at a few transformations in our object language[7] to expand on the description of the running example in Section 1.1 and to develop an intuitive understanding of transformation.

The simplest transformation is one that relates a typed term to itself; that is, transformation is reflexive. Consider the small but interesting examples listed in Table 1.

The first two are examples of the most basic transformations because each involves a single rewrite rule. In (1), a string is rewritten to a difference string by applying *rep* to the string. The transformation of (2) is a simple renaming operation.

Each of the examples (1) and (2) changes the type of the term, but note that the type changes have a regular pattern: every S becomes a Z and the type's functional structure is preserved. Using this pattern, we can intuitively "resolve" the changes and return the targets to terms that are semantically equivalent to the sources by applying *abs* and *rep* (and eta-reduction) in certains ways:

$$\texttt{"a"} \equiv \mathit{abs}\ (\mathit{rep}\ \texttt{"a"}) \qquad +\!\!+ \equiv \lambda x.\lambda y.\mathit{abs}\ (\mathit{rep}\ x \diamond \mathit{rep}\ y)$$

In (3), the source x has type S, but the target x has type Z. A transformation allows free variables to have different types in the source and target by relating the type environments, which may have different ranges but must have the same domains.

A transformation can relate different targets to one source, as demonstrated by (4), (5), and (6). The relation is left-total (a.k.a. a multivalued function) because the identity transformation is always allowed. We discuss the practical problem of choosing a preferred target in Section 7.2.

The transformation relation relates a source to a target and not necessarily vice versa. That is, we cannot guarantee that it is a symmetric relation. For example, *abs* and *rep* are only introduced and never eliminated; so, we cannot define a transformation relating a rewritten target to a source.

Examples (4), (5), and (6), in which the types are equal, are complete transformations. These transformations allow us to substitute the target for the source. Incomplete transformations such as (1), (2), and (3), can be subtransformations of complete transformations, but they are not complete themselves.

In the next section, we describe the typing infrastructure for rewriting and transformation.

[7] For simplicity, we consider any datatypes or **newtype**s defined in Haskell code to be base types in the object language.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathcal{T}(\tau,\upsilon) = \textbf{let}\ (\sigma,\mathring{\tau}) = \mathcal{T}'(\tau,\upsilon)\ \textbf{in}\ \mathring{\tau}} \\
\mathcal{T}'(B_s, & B_t) & = (id, B_s)\ \textbf{if}\ B_s \equiv B_t \\
\mathcal{T}'(\tau, & \alpha) & = ([\alpha \mapsto \tau], \tau) \\
\mathcal{T}'(\alpha, & \upsilon) & = ([\alpha \mapsto \upsilon], \upsilon) \\
\multicolumn{2}{l}{\mathcal{T}'(\tau_1 \to \tau_2, \upsilon_1 \to \upsilon_2)} & = \textbf{let}\ (\sigma_1, \mathring{\tau}_1) = \mathcal{T}'(\tau_1, \upsilon_1) \\
 & & \qquad\ (\sigma_2, \mathring{\tau}_2) = \mathcal{T}'(\sigma_1 \tau_2, \sigma_1 \upsilon_2) \\
 & & \quad\ \textbf{in}\ \ (\sigma_2 \circ \sigma_1, \sigma_2 \mathring{\tau}_1 \to \mathring{\tau}_2) \\
\mathcal{T}'(\mathcal{A}, & \mathcal{R}) & = (id, \iota)
\end{array}
$$

Figure 5. Definition of $\mathcal{T}$ and $\mathcal{T}'$ on types

4. The Typing of Type-and-Transform Systems

A key feature of type-and-transform systems is the support for transformations that allow for type-changing rewrites but enforce the discipline of type safety. We discuss the balance in this section by first describing type functors, a basic but important underlying concept in type-and-transform systems. Then, we present typed rewrite rules and transformations, especially the type-related aspects. We discuss the semantics-related aspects in Section 5.

4.1 Type Functors

The types of the two terms in a transformation are related by a *type functor*, which has the following syntax:

$$\mathring{\tau}, \mathring{\upsilon}\ ::=\ \alpha \mid B \mid \mathring{\tau} \to \mathring{\upsilon} \mid \iota$$

A type functor indicates the difference between two types (which we call $\mathcal{A}$ and $\mathcal{R}$) with the distinguished element ι. In the running example, wherever S ($\mathcal{A}$) is found in the source type and Z ($\mathcal{R}$) is found in the target type, the type functor has ι. Otherwise, the type functor mirrors the common structure of the two types.

The type functor $\mathring{\tau}$ of a transformation from source type τ_s to a target type τ_t is given by $\mathcal{T}(\tau_s, \tau_t)$, defined in Figure 5. The definition[8] of $\mathcal{T}$ uses $\mathcal{T}'$. The first component of $\mathcal{T}'(\tau,\upsilon)$ is the most general unifier, $\mathcal{U}(\tau,\upsilon)$, if it exists. That is, if $\sigma = \mathcal{U}(\tau,\upsilon)$, then $(\sigma,\mathring{\tau}) = \mathcal{T}'(\tau,\upsilon)$, and $\mathring{\tau}$ is isomorphic to τ and υ. The more interesting case occurs where $\mathcal{U}(\tau,\upsilon)$ is not defined but $\mathring{\tau} = \mathcal{T}(\tau,\upsilon)$ is. In that case, there will be a ι in $\mathring{\tau}$ wherever τ has $\mathcal{A}$ and υ has $\mathcal{R}$.

The type projection of a type functor $\mathring{\tau}$ is $\mathring{\tau}\langle\upsilon\rangle$, where every ι in $\mathring{\tau}$ is replaced by υ:

$$
\begin{array}{r}
{}^{\alpha}\!/\!{}_{B}\langle\upsilon\rangle = {}^{\alpha}\!/\!{}_{B} \\
(\mathring{\tau} \to \mathring{\upsilon})\langle\upsilon\rangle = \mathring{\tau}\langle\upsilon\rangle \to \mathring{\upsilon}\langle\upsilon\rangle \\
\iota\langle\upsilon\rangle = \upsilon
\end{array}
$$

Note that $\_\langle\tau\rangle$ is surjective, e.g. $\iota\langle S\rangle \equiv S\langle S\rangle$; this property will be useful later. With the definitions of $\mathcal{T}$ and $\_\langle\_\rangle$, we can state, for any $\mathring{\tau}$, the following inversion property:

$$\mathring{\tau} \equiv \mathcal{T}(\mathring{\tau}\langle\mathcal{A}\rangle, \mathring{\tau}\langle\mathcal{R}\rangle) \qquad (\mathring{\tau}\text{-INV})$$

For a transformation type functor $\mathring{\tau}$, the source type is $\mathring{\tau}\langle\mathcal{A}\rangle$, and the target type is $\mathring{\tau}\langle\mathcal{R}\rangle$.

To close transformations where free variables can change types, we use a *type functor environment* $\mathring{\Gamma}$, a slight adaptation of a type environment that maps variables to *type functor schemes*:

$$
\begin{array}{lll}
\mathring{\Gamma} & ::= & \varepsilon \mid \mathring{\Gamma}, x : \mathring{\varsigma} \\
\mathring{\varsigma} & ::= & \forall \bar{\alpha}.\mathring{\tau}
\end{array}
$$

Instantiation ($\prec$) and generalization ($\mathcal{G}$) work as expected. $\mathcal{T}$ can also be lifted to type functor schemes and environments:

$$\mathcal{T}_{\mathring{\Gamma}}(\varsigma_1, \varsigma_2) = \mathcal{G}_{\mathring{\Gamma}}(\mathcal{T}(\tau_1, \tau_2))\ \textbf{where}\ \tau_1 \prec \varsigma_1, \tau_2 \prec \varsigma_2$$

[8] Figure 5 is simplified in two ways for clarity. (1) The types $\mathcal{A}$ and $\mathcal{R}$ are implicit parameters. (2) To be more general, $\mathcal{A}$ and $\mathcal{R}$ should not be treated as patterns but checked for unification, e.g. with $\mathcal{U}(\tau,\mathcal{A})$.

$$\boxed{\Gamma \vdash p : \tau}$$

$$\frac{\tau \prec \Gamma(v)}{\Gamma \vdash v : \tau}\ \text{(P-VAR)} \qquad \frac{\Gamma \vdash p_1 : \tau \to \upsilon \qquad \Gamma \vdash p_2 : \tau}{\Gamma \vdash p_1\, p_2 : \upsilon}\ \text{(P-APP)}$$

$$\boxed{\Gamma \vdash \rho}$$

$$\frac{\Gamma \cup \mathring{\Gamma}\langle\mathcal{R}\rangle \vdash p_l : \mathring{\tau}_l\langle\mathcal{R}\rangle \qquad \Gamma \cup \mathring{\Gamma}\langle\mathcal{R}\rangle \vdash p_r : \mathring{\tau}_r\langle\mathcal{R}\rangle \\ \mathring{\tau}_l\langle\mathcal{A}\rangle \equiv \mathring{\tau}_r\langle\mathcal{A}\rangle}{\Gamma \vdash (\mathring{\Gamma} \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r)}\ \text{(R)}$$

$$\boxed{\Gamma \vdash \mathcal{R}}$$

$$\frac{\rho \in \mathcal{R} \qquad \Gamma \vdash \rho}{\Gamma \vdash \mathcal{R}}\ \text{(RS)}$$

Figure 6. Pattern, rule, and rule set typing

$$\begin{aligned}
&\mathcal{T}(\varepsilon, \qquad\qquad \varepsilon) \qquad\qquad = \varepsilon \\
&\mathcal{T}((\Gamma_1, v_1 : \varsigma_1), (\Gamma_2, v_2 : \varsigma_2)) = \\
&\quad \textbf{let}\ \mathring{\Gamma} = \mathcal{T}(\Gamma_1, \Gamma_2)\ \textbf{in}\ \mathring{\Gamma}, v_1 : \mathcal{T}_{\mathring{\Gamma}}(\varsigma_1, \varsigma_2)\ \textbf{if}\ v_1 \equiv v_2
\end{aligned}$$

We can likewise define lifted versions of $\_\langle\_\rangle$:

$$\begin{aligned}
\mathring{\varsigma}\langle\upsilon\rangle_{\mathring{\Gamma}} &= \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}\langle\upsilon\rangle)\ \textbf{where}\ \mathring{\tau} \prec \mathring{\varsigma} \\
\varepsilon\langle\upsilon\rangle &= \varepsilon \\
(\mathring{\Gamma}, v : \mathring{\varsigma})\langle\upsilon\rangle &= \mathring{\Gamma}\langle\upsilon\rangle, v : \mathring{\varsigma}\langle\upsilon\rangle_{\mathring{\Gamma}}
\end{aligned}$$

These lead to the following inversion properties:

$$\mathring{\varsigma} \equiv \mathcal{T}_{\mathring{\Gamma}}(\mathring{\varsigma}\langle\mathcal{A}\rangle_{\mathring{\Gamma}}, \mathring{\varsigma}\langle\mathcal{R}\rangle_{\mathring{\Gamma}}) \qquad (\mathring{\varsigma}\text{-INV})$$

$$\mathring{\Gamma} \equiv \mathcal{T}(\mathring{\Gamma}\langle\mathcal{A}\rangle, \mathring{\Gamma}\langle\mathcal{R}\rangle) \qquad (\mathring{\Gamma}\text{-INV})$$

From example (3) of Table 1, we infer the source and target type environments to be $\Gamma_s = \{x : S, \ldots\}$ and $\Gamma_t = \{x : Z, \ldots\}$, respectively. Thus, the type functor environment of the transformation is $\mathcal{T}(\Gamma_s, \Gamma_t) = \{x : \iota, \ldots\}$.

In the next section, we look at the other important component of the system, rewriting, and how type functors play a role there.

4.2 Typed Rewrite Rules

The typed rewrite rule is the basic unit of change. In standard term rewriting systems, the rule appears as a pair of patterns, $p_l \rightsquigarrow p_r$, where p_l is the lhs, p_r is the rhs, and p has the following syntax:

$$p \ ::= \ v \ \mid \ p_1\, p_2$$

A pattern is either a variable or the application of two patterns. Object variables (x), which are syntactically distinct from metavariables (m), are constant symbols. A term e is an instance of p if a substitution θ (mapping metavariables to terms) exists such that $\theta p \equiv e$. A *redex* is an instance of the lhs, θp_l, and *contracting* the redex means replacing it with the corresponding instance of the rhs, θp_r.

In type-and-transform systems, we extend the notation for a rule (ρ) by annotating the patterns with type functors and annotating the rule itself with a type functor environment:

$$\rho \ ::= \ \mathring{\Gamma} \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r$$

A rule set $\mathcal{R}$ is a finite set of typed rewrite rules.

The type functor environment $\mathring{\Gamma}$ of a rule closes over the metavariables of the two patterns; it does not include the object variables. The object variables are bound in the transformation type functor environment (described in Section 4.3).

Consider the rule set for the running example:

$$\begin{tikzcd}
\mathring{\tau}_l\langle\mathcal{A}\rangle \equiv \mathring{\tau}_r\langle\mathcal{A}\rangle \arrow[r, squiggly, "\mathring{\tau}_l"] \arrow[dr, squiggly, "\mathring{\tau}_r"'] & \mathring{\tau}_l\langle\mathcal{R}\rangle \arrow[d, squiggly] \\
& \mathring{\tau}_r\langle\mathcal{R}\rangle
\end{tikzcd}$$

Figure 7. Diagram of transformation type functors and types

$$\{m : S\} \triangleright m \ : S \qquad \rightsquigarrow rep\ m : \iota \qquad \text{(SZ-1)}$$

$$\{m : \iota\} \triangleright m \ : \iota \qquad \rightsquigarrow abs\ m : S \qquad \text{(SZ-2)}$$

$$\varepsilon \triangleright +\!\!+ : S \to S \to S \rightsquigarrow \diamond \qquad : \iota \to \iota \to \iota \qquad \text{(SZ-3)}$$

There is a simple intuition behind the type functors in the above rules: if a pattern or metavariable has type Z (which is the target type of the transformation), that type is replaced with the type functor “placeholder” ι.

We can view ι as a viral infection that spreads throughout the program via rewriting. The infection is introduced with (SZ-1), propagated by (SZ-3), and eliminated by (SZ-2). In the end, we need the $\mathcal{A}$ type (e.g. S) to preserve the source type, and the $\mathcal{R}$ type (Z) is an underlying type used only during transformation.

A rule set $\mathcal{R}$ must be well-typed under some type environment Γ (closing over the object variables) according to the inference rules for the judgment $\Gamma \vdash \mathcal{R}$ in Figure 6. The inference rules for the prerequisite judgment on typing patterns ($\Gamma \vdash p : \tau$) are standard, but the inference rule for typing a rewrite rule ($\Gamma \vdash \rho$) needs explanation.

In Figure 1, we showed that rewriting applies a rule to a target term to produce a new target term and that a source term is related by transformations to both target terms. Consider the type functors $\mathring{\tau}_l$ and $\mathring{\tau}_r$ for the two transformations. By adapting Figure 1 to type functors, as discussed in Section 4.1, Figure 7 shows the relationships between the type functors and the source and target types .

The premises of the inference rule (R) of the judgment $\Gamma \vdash \rho$ include two relationships from Figure 7. First, the patterns must be typed with the target type (e.g. $\mathring{\tau}_l\langle\mathcal{R}\rangle$ for the lhs) and type environment ($\mathring{\Gamma}_l\langle\mathcal{R}\rangle$). Second, the lhs and rhs source types must be equivalent:[9]

$$\mathring{\tau}_l\langle\mathcal{A}\rangle \equiv \mathring{\tau}_r\langle\mathcal{A}\rangle \qquad (\mathring{\tau}\text{-REW})$$

These conditions ensure that a typed rewriting rule will preserve the typing relationships of a transformation. The reader may wish to verify that the rule set of (SZ-1), (SZ-2), and (SZ-3) is well-typed.

4.3 Transformation

A transformation is given by a derivation of the following judgment:

$$\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$$

The relation can be interpreted as: given a type functor environment $\mathring{\Gamma}$ and a typed rewrite rule set $\mathcal{R}$, a source e transforms to a target e' with the type functor $\mathring{\tau}$.

The inference rules for the transformation judgment are given in Figure 8. Most of the rules correspond directly to typing rules in Figure 4. They enforce the structural mirroring of the source and target as well as the typing of the terms. Type functors and type functor environments are treated simply as types and type environments.

The one inference rule that does not correspond to typing, (T-REW), is for type-changing rewriting. Its premises are:

[9] This is where the surjectivity of $\_\langle\tau\rangle$ plays a role.

$$\boxed{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}}$$

$$\frac{\mathring{\tau} \prec \mathring{\Gamma}(x)}{\mathring{\Gamma} \vdash x \overset{\mathcal{R}}{\rightsquigarrow} x : \mathring{\tau}}\ \text{(T-VAR)} \qquad \frac{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau} \to \mathring{\tau}}{\mathring{\Gamma} \vdash \mathbf{fix}\ e \overset{\mathcal{R}}{\rightsquigarrow} \mathbf{fix}\ e' : \mathring{\tau}}\ \text{(T-FIX)}$$

$$\frac{\mathring{\Gamma} \vdash e_1 \overset{\mathcal{R}}{\rightsquigarrow} e'_1 : \mathring{\tau} \to \mathring{\upsilon} \qquad \mathring{\Gamma} \vdash e_2 \overset{\mathcal{R}}{\rightsquigarrow} e'_2 : \mathring{\tau}}{\mathring{\Gamma} \vdash e_1\ e_2 \overset{\mathcal{R}}{\rightsquigarrow} e'_1\ e'_2 : \mathring{\upsilon}}\ \text{(T-APP)}$$

$$\frac{\mathring{\Gamma}, x : \mathring{\tau} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\upsilon}}{\mathring{\Gamma} \vdash \lambda x.e \overset{\mathcal{R}}{\rightsquigarrow} \lambda x.e' : \mathring{\tau} \to \mathring{\upsilon}}\ \text{(T-LAM)}$$

$$\frac{\mathring{\Gamma} \vdash e_1 \overset{\mathcal{R}}{\rightsquigarrow} e'_1 : \mathring{\tau} \qquad \mathring{\Gamma}, x : \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}) \vdash e_2 \overset{\mathcal{R}}{\rightsquigarrow} e'_2 : \mathring{\upsilon}}{\mathring{\Gamma} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \overset{\mathcal{R}}{\rightsquigarrow} \mathbf{let}\ x = e'_1\ \mathbf{in}\ e'_2 : \mathring{\upsilon}}\ \text{(T-LET)}$$

$$\frac{\begin{array}{c}(\mathring{\Gamma}_m \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r) \in \mathcal{R} \\ \mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}_l \qquad \mathring{\Gamma};\mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\rightsquigarrow} p_l @ e' \Rightarrow \theta\end{array}}{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} \theta p_r : \mathring{\tau}_r}\ \text{(T-REW)}$$

$$\boxed{\mathring{\Gamma};\mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\rightsquigarrow} p @ e' \Rightarrow \theta}$$

$$\frac{}{\mathring{\Gamma};\mathring{\Gamma}_m \vdash x \overset{\mathcal{R}}{\rightsquigarrow} x @ x \Rightarrow id}\ \text{(M-VAR)}$$

$$\frac{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau} \qquad \mathring{\tau} \prec \mathring{\Gamma}_m(m)}{\mathring{\Gamma};\mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\rightsquigarrow} m @ e' \Rightarrow [m \mapsto e']}\ \text{(M-MVAR)}$$

$$\frac{\begin{array}{c}\mathring{\Gamma};\mathring{\Gamma}_m \vdash e_1 \overset{\mathcal{R}}{\rightsquigarrow} p_1 @ e'_1 \Rightarrow \theta_1 \\ \mathring{\Gamma};\mathring{\Gamma}_m \vdash e_2 \overset{\mathcal{R}}{\rightsquigarrow} p_2 @ e'_2 \Rightarrow \theta_2\end{array}}{\mathring{\Gamma};\mathring{\Gamma}_m \vdash e_1\ e_2 \overset{\mathcal{R}}{\rightsquigarrow} p_1\ p_2 @ e'_1\ e'_2 \Rightarrow \theta_2 \circ \theta_1}\ \text{(M-APP)}$$

Figure 8. Transformation and pattern matching

1. There is a typed rewrite rule ρ in the rule set $\mathcal{R}$.
2. There is a transformation with the lhs type functor of ρ.
3. The transformation target e' is a redex (θp_l) of ρ.

The consequence of these obligations is a transformation with the source of the transformation in premise 2, a target that is the contraction (θp_r) of ρ, and a type functor $(\mathring{\tau}_r)$ from the rhs of ρ.

The third premise, describing pattern matching for a typed rewrite rule, is given by the following judgment:

$$\mathring{\Gamma};\mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\rightsquigarrow} p @ e' \Rightarrow \theta$$

The interpretation is that, given an object variable environment $\mathring{\Gamma}$ and a metavariable environment $\mathring{\Gamma}_m$, a pattern p matches a target e' and produces a substitution θ. Of the inference rules shown in Figure 8, (M-VAR) and (M-APP) are straightforward structural matches. In (M-MVAR), we see that the source e is used to ensure that, when a metavariable is found, the corresponding source and target terms are components of a subtransformation. The metavariable type functor from $\mathring{\Gamma}_m$ must match the subtransformation's type functor.

We have presented the purely type-related aspects of type-and-transform systems: type functors and the inference systems for typing rewrite rules and describing transformations. However, rewriting and transformation also establish relations between terms. We discuss this in the next section.

5. The Semantics of Type-and-Transform Systems

In this section, we describe the semantics relations of rewriting and transformation. We begin with a description of a type functor as a difunctor, linking types to terms. Then, we discuss the difunctor properties required for typed rewrite rules and transformation.

5.1 Difunctors

A *difunctor* [9, 21] is a mixed-variant binary type constructor F with the function:

$$dimap : \forall a\ a'\ b\ b'.(a' \to b') \to (b \to a) \to F\ b'\ b \to F\ a'\ a$$

The first parameter of F is contravariant, and the second is covariant. The function *dimap* must obey the following laws of identity and distribution over composition:

$$dimap\ id\ id \equiv id \qquad \text{(D-ID)}$$

$$dimap\ (g \circ h)\ (i \circ j) \equiv dimap\ h\ i \circ dimap\ g\ j \qquad \text{(D-COMP)}$$

A type functor is a difunctor with the same parameter in the covariant and contravariant positions. That is, if F is a type functor, then $\mathring{\tau}\langle a \rangle = F\ a\ a$, and *dimap* can be simplified:

$$dimap : \forall a\ b.(a \to b) \to (b \to a) \to F\ b\ b \to F\ a\ a$$

For brevity, we write the *dimap* for type functors as $\mathcal{D}_{\mathring{\tau}}$:

$$\begin{array}{ll}
\mathcal{D}_{\mathring{\tau}} : \forall a\ b.(a \to b) \to (b \to a) \to \mathring{\tau}\langle b \rangle \to \mathring{\tau}\langle a \rangle \\
\mathcal{D}_{\alpha/B} & f\ g = id \\
\mathcal{D}_{\mathring{\tau} \to \mathring{\upsilon}}\, f\ g = \lambda x \to \mathcal{D}_{\mathring{\upsilon}}\, f\ g \circ x \circ \mathcal{D}_{\mathring{\tau}}\ g\, f \\
\mathcal{D}_{\iota} & f\ g = g
\end{array}$$

Note the argument reversal in the contravariant usage of the $\mathring{\tau} \to \mathring{\upsilon}$ case.

As with previous functions on types, we can lift *dimap* to type functor schemes and environments. Schemes are straightforward:

$$\begin{array}{l}
\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}} : \forall a\ b.(a \to b) \to (b \to a) \to \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}\langle b \rangle) \to \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}\langle a \rangle) \\
\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}} = \mathcal{D}_{\mathring{\tau}}\ \mathbf{where}\ \mathring{\tau} \prec \mathring{\varsigma}
\end{array}$$

Lifting *dimap* to type functor environments requires a slight twist. We give $\mathcal{D}_{\mathring{\Gamma}}\, f\ g$ the type $\mathring{\Gamma}\langle b \rangle \to \mathring{\Gamma}\langle a \rangle$ and define it as a substitution on terms:

$$\begin{array}{ll}
\mathcal{D}_{\mathring{\Gamma}} : \forall a\ b.(a \to b) \to (b \to a) \to \mathring{\Gamma}\langle b \rangle \to \mathring{\Gamma}\langle a \rangle \\
\mathcal{D}_{\varepsilon} & f\ g = id \\
\mathcal{D}_{\mathring{\Gamma}, \nu : \mathring{\varsigma}}\, f\ g = \mathcal{D}_{\mathring{\Gamma}}\, f\ g \circ [\nu \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ g\, f\ \nu]
\end{array}$$

Note that the use of $\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}$ in the second case is contravariant.

Here is an example of applying the various *dimap*s:

$$\begin{aligned}
(\mathcal{D}_{\{x:\iota\}}\ rep\ abs)x &= (id \circ [x \mapsto \mathcal{D}_{\iota,\varepsilon}\ abs\ rep\ x])x \\
&= [x \mapsto \mathcal{D}_{\iota}\ abs\ rep\ x]x \\
&= rep\ x
\end{aligned}$$

We do not use $\mathcal{D}_{\mathring{\Gamma}}\ rep\ abs$ in any other form, so, for conciseness, we omit the arguments *rep* and *abs*. To reduce the number of brackets, substitution application has a higher precedence than function application.

5.2 Typed Rewrite Rules

A rewrite rule $\mathring{\Gamma} \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r$ is typed by the inference rule (R), but this condition is not sufficient to prevent rewriting from breaking a program. (It is trivial to come up with an example rewrite rule that changes terms but not types.) Our intention is ultimately to preserve the semantics of the source term in the target (for a complete transformation), so we must establish a relation between the rule patterns that connects them to the source term.

From the source type equivalence $\mathring{\tau}_l\langle\mathcal{A}\rangle \equiv \mathring{\tau}_r\langle\mathcal{A}\rangle$ (Section 4.2), we derive the following equivalence on patterns:

$$\mathcal{D}_{\mathring{\tau}_l}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} p_l \equiv \mathcal{D}_{\mathring{\tau}_r}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} p_r \qquad \text{(D-REW)}$$

Given an isomorphism for $\mathcal{A}$ and $\mathcal{R}$, we map the patterns (as terms) to equivalent forms using the *dimap* of each pattern's type functor (i.e. $\mathcal{D}_{\mathring{\tau}}\ rep\ abs : \mathring{\tau}\langle\mathcal{R}\rangle \to \mathring{\tau}\langle\mathcal{A}\rangle$). Before applying $\mathcal{D}_{\mathring{\tau}}$ to a pattern, we apply the substitution $\mathcal{D}_{\mathring{\Gamma}}$, which applies $\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}$ to each metavariable in the pattern.

The difunctor ensures that we can map both a term (or pattern) and a type functor, thus preserving the equivalence of both. To demonstrate (D-REW), we give the following properties for the rewrite rules of the running example (see Section 4.2):

$$\mathcal{D}_S\ rep\ abs\ \mathcal{D}_{\{m:S\}} m \equiv \mathcal{D}_\iota\ rep\ abs\ \mathcal{D}_{\{m:S\}} (rep\ m) \qquad \text{(SZ-D-1)}$$

$$\mathcal{D}_\iota\ rep\ abs\ \mathcal{D}_{\{m:\iota\}} m \equiv \mathcal{D}_S\ rep\ abs\ \mathcal{D}_{\{m:\iota\}} (abs\ m) \qquad \text{(SZ-D-2)}$$

$$\mathcal{D}_{S\to S\to S}\ rep\ abs\ \mathcal{D}_\varepsilon (+\!\!+) \equiv \mathcal{D}_{\iota\to\iota\to\iota}\ rep\ abs\ \mathcal{D}_\varepsilon (\diamond) \qquad \text{(SZ-D-3)}$$

We leave the proof of these properties as a simple exercise. It is worth noting that these rules could be proven even if $\mathcal{D}_{\mathring{\Gamma}} = id$ (for any $\mathring{\Gamma}$); however, the substitution is necessary when both patterns have both metavariables and object variables, as in the following:

$$\{m : \iota\} \rhd (abs\ m+\!\!+) : S \to S \rightsquigarrow (m\diamond) : \iota \to \iota$$

We leave the (D-REW) property and proof of this rule as exercises for the reader. The solution can be found in Appendix A.

5.3 Transformation

In Section 4.3, we established a transformation $\mathring{\Gamma} \vdash e_s \overset{\mathcal{R}}{\rightsquigarrow} e_t : \mathring{\tau}$ as a relation between a source e_s and a target e_t whose types may differ as specified by the type functor $\mathring{\tau}$. As with typed rewrite rules, we can relate the semantics of the terms using the difunctor aspect of the type functor. We apply a *dimap* to the target term to equate it to the source term:

$$e_s \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_t \qquad \text{(D-TRANS)}$$

In a transformation with the type functor $\mathring{\tau}$, the source does not change, so we map the target $e_t : \mathring{\tau}\langle\mathcal{R}\rangle$ to a term equivalent to the source $e_s : \mathring{\tau}\langle\mathcal{A}\rangle$ with $\mathcal{D}_{\mathring{\tau}}\ rep\ abs : \mathring{\tau}\langle\mathcal{R}\rangle \to \mathring{\tau}\langle\mathcal{A}\rangle$ and $\mathcal{D}_{\mathring{\Gamma}}$.

For an example of (D-TRANS) in action, we prove the semantics relation of example (3). The proposition is:

$$x +\!\!+ \texttt{"b"} \equiv \mathcal{D}_\iota\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} (x \diamond rep\ \texttt{"b"})$$

The subscript ι in $\mathcal{D}_\iota$ comes from rewriting with (SZ-3). The environment $\mathring{\Gamma}$ contains a mapping for at least all the variables mentioned, including $\{x : \iota\}$. The proof follows:

$$\begin{aligned}
& \mathcal{D}_\iota\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} (x \diamond rep\ \texttt{"b"}) \\
\equiv\ & \{\text{ apply } \mathcal{D}_{\mathring{\Gamma}} \} \\
& \mathcal{D}_\iota\ rep\ abs\ (\mathcal{D}_\iota\ abs\ rep\ x \diamond rep\ \texttt{"b"}) \\
\equiv\ & \{\text{ apply } \mathcal{D}_\iota \} \\
& abs\ (rep\ x \diamond rep\ \texttt{"b"}) \\
\equiv\ & \{\text{ apply } rep \} \\
& abs\ (Z\ (x+\!\!+) \diamond Z\ (\texttt{"b"}+\!\!+)) \\
\equiv\ & \{\text{ apply } \diamond \} \\
& abs\ (Z\ ((x+\!\!+) \circ (\texttt{"b"}+\!\!+))) \\
\equiv\ & \{\text{ apply } abs \} \\
& ((x+\!\!+) \circ (\texttt{"b"}+\!\!+))\ \texttt{""} \\
\equiv\ & \{\text{ apply } \circ \} \\
& (\lambda y \to x +\!\!+ \texttt{"b"} +\!\!+ y)\ \texttt{""} \\
\equiv\ & \{\text{ (RED-LAM) } \} \\
& x +\!\!+ \texttt{"b"} +\!\!+ \texttt{""} \\
\equiv\ & \{\ \texttt{""} \text{ is the unit of } +\!\!+ \} \\
& x +\!\!+ \texttt{"b"}
\end{aligned}$$

In the next section, we discuss the formal definitions and properties of the concepts introduced in Sections 4 and 5.

6. Definitions and Properties

We have introduced the typing and semantics relations of typed rewrite rules. For type-and-transform systems, we require the rules to be *valid*:

Definition 1 (Typed rewrite rule validity). *Given a type environment Γ and an $\mathcal{A}/\mathcal{R}$ isomorphism, a typed rewrite rule*

$$\rho = \mathring{\Gamma} \rhd p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r$$

is valid if it satisfies:

1. *$\Gamma \vdash \rho$ (Section 4.2)*
2. *$\mathcal{D}_{\mathring{\tau}_l}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} p_l \equiv \mathcal{D}_{\mathring{\tau}_r}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} p_r$ (Section 5.2)*
3. *Left-linearity*
4. *$fv(p_l) \supseteq fv(p_r)$* •

Property 3 (i.e. no metavariable occurs twice in p_l) keeps the system simple by avoiding the need for equality on terms. Property 4 (where $fv(p)$ is the set of metavariables in p) prevents unbound metavariables. To avoid nontermination, some term-rewriting systems disallow a lone metavariable in the lhs [1]; however, they are essential to the expressiveness of typed rewrite rules. To ensure termination, we use a particular rewriting strategy (see Section 7.1).

A rule set is valid if every rule in the set is valid for the same environment and isomorphism.

We now formally define a transformation:

Definition 2 (Transformation). *Given an $\mathcal{A}/\mathcal{R}$ isomorphism, a transformation is a tuple*[10] *$(\mathring{\Gamma}, \mathcal{R}, e, e', \mathring{\tau})$, where $\mathcal{R}$ is valid for $\mathring{\Gamma}\langle\mathcal{R}\rangle$ and the isomorphism, that satisfies $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$ (Section 4.3).* •

The most basic property that transformations have is that the source and target terms are well-typed:

Theorem 1 (Typing of transformation terms). *The terms of a transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$ are typed by:*

1. *$\mathring{\Gamma}\langle\mathcal{A}\rangle \vdash e : \mathring{\tau}\langle\mathcal{A}\rangle$*
2. *$\mathring{\Gamma}\langle\mathcal{R}\rangle \vdash e' : \mathring{\tau}\langle\mathcal{R}\rangle$* □

PROOF By straightforward rule induction on the derivations. In the (T-REW) case, the rewrite rule validity ensures the rhs and thus the contraction will be appropriately typed. ■

A transformation also allows us to relate the semantics of the source and target:

Theorem 2 (Semantics of transformation terms). *A transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$ satisfies $e \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e'$ (Section 5.3).* □

PROOF By rule induction on the derivations of $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$. In the (T-FIX) case, we use (RED-ROLLING). In the (T-REW) case, we use the (D-REW) property for each rewrite rule. The full proof is given in Appendix B. ■

The final property is that of a complete transformation, in which $\mathring{\tau}\langle\mathcal{A}\rangle \equiv \mathring{\tau}\langle\mathcal{R}\rangle$. Complete transformations have the special property that the semantics of the terms are also equivalent. First, we need to explain how to determine the same-type property. Recall that ι indicates where the type changes in a type functor (Figure 5). We simply check that $\mathring{\tau}$ "does not have" any ιs:

[10] To be precise, a transformation is a tuple that satisfies the transformation judgment, but we normally use the judgment to refer to a transformation.

$$\begin{aligned}
\bar{\iota}(\alpha/B) &= \mathit{true} \\
\bar{\iota}(\iota) &= \mathit{false} \\
\bar{\iota}(\mathring{\tau} \to \mathring{\upsilon}) &= \bar{\iota}(\mathring{\tau}) \wedge \bar{\iota}(\mathring{\upsilon})
\end{aligned}$$

These lemmas follow from the definition of $\bar{\iota}(\_)$:

Lemma 1. *If $\bar{\iota}(\mathring{\tau})$, then $\mathring{\tau}\langle\tau\rangle \equiv \mathring{\tau}\langle\upsilon\rangle$ for any τ and υ.* □

PROOF By straightforward induction on $\mathring{\tau}$. ■

Lemma 2. *If $\bar{\iota}(\mathring{\tau})$, then $\mathcal{D}_{\mathring{\tau}}\, f\, g \equiv id$ for any f and g.* □

PROOF By straightforward induction on $\mathring{\tau}$. ■

It is straightforward to lift the function $\bar{\iota}(\_)$ to – and thus prove the above lemmas for – type functor schemes and environments.

We can now define complete transformations:

Definition 3 (Complete transformation). *A transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}$ is complete if $\bar{\iota}(\mathring{\Gamma})$ and $\bar{\iota}(\mathring{\tau})$.* ●

The expected properties follow:

Theorem 3 (Typing of complete transformation terms). *If a transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}$ is complete, then the following hold:*

1. $\Gamma \equiv \mathring{\Gamma}\langle\mathcal{A}\rangle \equiv \mathring{\Gamma}\langle\mathcal{R}\rangle$ *and* $\tau \equiv \mathring{\tau}\langle\mathcal{A}\rangle \equiv \mathring{\tau}\langle\mathcal{R}\rangle$
2. $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$ □

PROOF Follows from Theorem 1 and Lemma 1. ■

Theorem 4 (Semantics of complete transformation terms). *If a transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}$ is complete, then $e \equiv e'$.* □

PROOF Follows from Theorem 2 and Lemma 2. ■

This completes the formal description of type-and-transform systems. In the next section, we discuss other aspects.

7. Discussion

There are a number of points for further discussion on type-and-transform systems.

7.1 Algorithm

Here, we present a summary of the features of a transformation algorithm using a valid typed rewrite rule set. We will report on the algorithm in greater detail in a separate paper.

Our algorithm is a direct adaptation of algorithm $\mathcal{W}$ [22], a standard algorithm for the object language type system. Given the correspondence of transformation (Figure 8) to typing (Figure 4), the similarity is not surprising. The arguments to our algorithm are a rewrite rule set, a type environment, and a term. The result is a list of recursively annotated terms – every subterm is annotated with a type substitution, a type functor, and a weight. The recursive structure of the algorithm is identical to $\mathcal{W}$; the primary difference is that we rewrite the subterm after every recursive call. The substitution serves the same purpose as in $\mathcal{W}$, and the type functor is treated as a simple type. We use weights to choose the preferred transformed term. This is described more in Section 7.2.

Rewriting works in the usual way – pattern matching and substitution on the annotated terms – however, matching also includes type functor annotations. This can be seen in (M-MVAR) in Figure 8, which needs the type functor from a subtransformation.

In (T-REW) (Figure 8), we pick one of the rewrite rules from the rule set, but in the algorithm, we apply every rule. This is why the algorithm returns a list of results: multiple rules (including identity) may by successful. Type-incorrect rewrite results are discarded.

A transformation according to Figure 8 may not terminate. That is, a derivation of $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}$ may be infinitely long because the inference rule (T-REW) can be instantiated an unlimited number of times. (Unlike the other inference rules, (T-REW) is not syntax-directed.) In the transformation algorithm, we apply each rewrite rule once at each subterm. This strategy guarantees termination, but it also does not produce all possible results. In our experience with type-and-transform systems, this is not an issue; however, we plan to explore it further with an empirical comparison of algorithmic variations.

Soundness The transformation algorithm is sound. It implements the transformation of Definition 2. Soundness follows from the correspondence between transformation and typing and rewrite rule validity.

Completeness It is trivial to show that the algorithm does not satisfy completeness. In a separate paper, we plan to describe restricted transformations for which we can prove completeness.

Source	Target	Weight	
`"a"` $+\!\!+$ `"b"` $+\!\!+$ `"c"`			(7)
	$\mathit{abs}\ (\mathit{rep}$ `"a"` $\diamond\, \mathit{rep}$ `"b"` $\diamond\, \mathit{rep}$ `"c"`$)$	$(-4,4)$	
	`"a"` $+\!\!+\ \mathit{abs}\ (\mathit{rep}$ `"b"` $\diamond\, \mathit{rep}$ `"c"`$)$	$(-2,3)$	
$(\lambda x.x +\!\!+ x)$ `"a"`			(8)
	$\mathit{abs}\ ((\lambda x.x \diamond x)\ (\mathit{rep}$ `"a"`$))$	$(-2,2)$	
	$\mathit{abs}\ ((\lambda x.\mathit{rep}\ x \diamond \mathit{rep}\ x)$ `"a"`$)$	$(-2,3)$	

Table 2. Examples of comparable transformations

7.2 Choosing the "Best" Transformation

Given a program P, a well-scoped subprogram T, and a metric $|\_|$ (e.g. tokens), the *transformation coverage* is the ratio $|T|/|P|$ where T has undergone a complete transformation and the context of P excluding T has not been transformed.

One might think there is an optimal transformation for the desired coverage, but we have found no useful strict ordering on transformations. Instead, we have often seen multiple transformation targets that are equally "good." Consider the example transformations (4), (5), and (6) in Table 1. It is not obvious which is better; in the context of a larger program, any one of them may prove more useful.

Some transformations, on the other hand, are clearly better than others. Consider the two example sources in Table 2 (ignoring the weight column for now), each with two possible targets. In (7), we prefer to have more $+\!\!+$ replaced by $\diamond$. In (8), we prefer to have fewer *rep* introduced. In general, the preferences may be contradictory.

Recall the viral infection analogy. The basic idea is that we infect as early as possible and spread the infection as far as possible. Rewrite rules such as (SZ-1) introduce the infection, and rules such as (SZ-2) eliminate the infection. We therefore wish to minimize the occurrence of these rules, which we call *repair rules* since they repair types. The rule (SZ-3) is a "vector" for the infection, transmitting it from one term to another. We wish to maximize the application of these rules, which we call *rename rules* since they usually rename functions.

We assign a score to each rename rule depending on how "valuable" the rule is. The score is a measure of how effective the rule is in maximizing the transformation coverage. Consider the following rules for some types $\mathcal{A}$ and $\mathcal{R}$:

$$\varepsilon \rhd f_0 : \mathcal{A} \leadsto g_0 \quad : \iota \qquad (\rho_0)$$

$$\varepsilon \rhd f_1 : \mathcal{A} \to \mathcal{A} \leadsto g_1 \quad : \iota \to \iota \qquad (\rho_1)$$

$$\{m : \iota\} \rhd m : \iota \leadsto \mathit{revert}\ m : \mathcal{A} \qquad (\mathit{revert})$$

Rules (ρ_0) and (ρ_1) are rename rules, and rule (revert) is a repair rule.

To determine the score of rule (ρ_0), we observe that it does not necessarily improve the program more than the identity rewrite, the "default" if no rules are applied. For example, rewriting f_0 to $\mathit{revert}\ g_0$ is probably not preferable over leaving f_0 alone (though an argument could be made otherwise, depending on the system). Alternatively, rewriting $f_1\,f_0$ to $\mathit{revert}\ (g_1\ g_0)$ is likely preferred. To indicate this ambivalence for (ρ_0), we assign it a score of 0, the same score as the identity rewrite (or not rewriting at all).

Rule (ρ_1), on the other hand, always improves the coverage of the transformation (over the identity) because it requires rewriting the argument to f_1. Its score is -1 (as in golf, a lower score is better). If we rewrite $f_1\ f_0$ to $g_1\ g_0$, the rename score is the sum of the rename rule scores, -1.

In general, the rename score of a rule that rewrites an n-arity function f_n to g_n is $-n$. For simplicity, we include in the arity only the arguments of type $\mathcal{A}$; other types either do not affect the score (as in *Int*) or have a non-obvious effect (as in function types).

After maximizing the renames, we minimize the repairs, which means applying as few repair rules as possible.

To rank targets, we assign each a weight – a pair (m,n) where m is the rename score and n is the number of repair rules – such that we can order the targets lexicographically by weight. The winning target is the program with the smallest weight. Alternatively stated:

1. We choose the targets with the lowest rename score.
2. If multiple targets have the same rename score, we choose the targets with fewer repairs.
3. We (arbitrarily) choose the first of the remaining targets.

In the running example, we assign the rename score -2 to the rule (SZ-3) since the function $+\!\!+$ has an arity of 2. The weights in Table 2 show how our approach leads to our preferred choices.

7.3 Parameterized Type Constructors

Up to this point, we have used only simple (nullary) types to simplify explanation. We can also support parameterized type constructors.

The adapted syntax of types and type functors follows:

$$\begin{array}{lcl}
\varphi & ::= & c,d \mid C \\
\mathring{\varphi} & ::= & \varphi \mid \iota \\
\tau,\upsilon & ::= & \alpha \mid B \mid \tau \to \upsilon \mid \varphi\ \tau \\
\mathring{\tau},\mathring{\upsilon} & ::= & \alpha \mid B \mid \mathring{\tau} \to \mathring{\upsilon} \mid \mathring{\varphi}\ \mathring{\tau}
\end{array}$$

A type constructor is either a type variable (c,d) or a base type constructor (C), and we now use ι as a type functor constructor. We modify type projection, $\mathring{\tau}\langle\varphi\rangle$, for constructors and extend it with new cases:

$$\begin{aligned}
{}^{\alpha}\!/_{B}\langle\varphi\rangle &= {}^{\alpha}\!/_{B} \\
(\mathring{\tau} \to \mathring{\upsilon})\langle\varphi\rangle &= \mathring{\tau}\langle\varphi\rangle \to \mathring{\upsilon}\langle\varphi\rangle \\
(C\ \mathring{\tau})\langle\varphi\rangle &= C\ \mathring{\tau}\langle\varphi\rangle \\
(\iota\ \mathring{\tau})\langle\varphi\rangle &= \varphi\ \mathring{\tau}\langle\varphi\rangle
\end{aligned}$$

In the last two cases, C and φ are difunctors, as we can see more clearly in the definition of $\mathcal{D}_{\mathring{\tau}}$:

$$\begin{array}{ll}
\mathcal{D}_{\mathring{\tau}} : \forall c\ d.(\forall a\ b.(a \to b) \to (b \to a) \to c\ b \to c\ a) \to & \\
\qquad\qquad (\forall a\ b.(a \to b) \to (b \to a) \to d\ b \to d\ a) \to & \\
\qquad\qquad \forall a.(c\ a \to d\ a) \to (d\ a \to c\ a) \to \mathring{\tau}\langle d\rangle \to \mathring{\tau}\langle c\rangle & \\
\mathcal{D}_{\alpha/B} & d_c\ d_d\ f\ g = \mathit{id} \\
\mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}} & d_c\ d_d\ f\ g = \lambda x \to \mathcal{D}_{\mathring{\upsilon}}\ d_c\ d_d\ f\ g \circ x \circ \mathcal{D}_{\mathring{\tau}}\ d_d\ d_c\ g\,f \\
\mathcal{D}_{C\ \mathring{\tau}} & d_c\ d_d\ f\ g = \mathit{dimap}_C\ (\mathcal{D}_{\mathring{\tau}}\ d_d\ d_c\ g\,f)\ (\mathcal{D}_{\mathring{\tau}}\ d_c\ d_d\,f\ g) \\
\mathcal{D}_{\iota\ \mathring{\tau}} & d_c\ d_d\ f\ g = g \circ d_d\ (\mathcal{D}_{\mathring{\tau}}\ d_d\ d_c\ g\,f)\ (\mathcal{D}_{\mathring{\tau}}\ d_c\ d_d\,f\ g)
\end{array}$$

Here, dimap_C is the *dimap* for the base constructor C, and the function arguments d_c and d_d are the *dimap*s for the relevant type constructors. Note that we do not define $\mathcal{D}_{\mathring{\tau}}$ for type constructor variables because we do not have a *dimap* for those.

As an aside, if the parameter of a type constructor φ is not used in contravariant positions, then $\mathit{dimap}_{\varphi}\,f\ g \equiv \mathit{map}_{\varphi}\ g$, where map_{φ} is the covariant functor of φ.

The type-and-transform systems work of Sections 4, 5, and 6 can be developed in a straightforward manner for unary type constructors. It is also possible to define $\mathcal{D}_{\mathring{\tau}}$ for type constructors of arbitrary arity using kind-indexed types [15].

7.4 Difference Lists

With support for parameterized type constructors, we can describe the transformation for Hughes' lists or difference lists, mentioned in Section 1.1.

Difference lists are trivially different from difference strings (Figure 2). We present only the **newtype** and the type signatures:

$$\begin{array}{l}
\textbf{newtype}\ H\ a = H\ ([a] \to [a]) \\
(\diamond) :: H\ a \to H\ a \to H\ a \\
\epsilon\quad :: H\ a \\
\mathit{rep} :: [a] \to H\ a \\
\mathit{abs} :: H\ a \to [a]
\end{array}$$

To describe the transformation of lists to difference lists, the following inputs are needed for the type-and-transform system:

1. Type (constructor) pair and functions to witness the isomorphism
2. Typed rewrite rules, including both repair rules (such as the isomorphism functions) and rename rules
3. Proof that the rewrite rules are valid (according to Definition 1)

As far as what is necessary for a practical system, only the rewrite rules are needed. The isomorphism is implied by the rules, and the proof is an external obligation for correctness. We leave the proof as an exercise for the reader. In general, these proofs are not very difficult. They follow the style of the example proof in Section 5.3.

The typed rewrite rules for the list-to-difference-list transformation are:

$$\{m:[a]\} \triangleright m\ :[a] \rightsquigarrow \mathit{rep}\ m : \iota\ a \tag{9}$$
$$\{m:\iota\ a\} \triangleright m\ :\iota\ a \rightsquigarrow \mathit{abs}\ m : [a] \tag{10}$$
$$\varepsilon \triangleright +\!\!+ : [a] \to [a] \to [a] \rightsquigarrow \diamond \quad : \iota\ a \to \iota\ a \to \iota\ a \tag{11}$$
$$\varepsilon \triangleright [\,]\ : [a] \rightsquigarrow \epsilon \quad : \iota\ a \tag{12}$$

Only (12) is new compared to the difference string rules (Sections 4.2 and 7.2), and we assign it a score of 0.

There are a few interesting transformations that we can perform. The first is the reverse example (for reference, r is the initial reverse function and r' is the transformed function):

$$\begin{array}{l}
\textbf{let}\ r\ = \textbf{fix}\ (\lambda f.\mathit{list}\ [\,]\ (\lambda x\ xs.f\ xs +\!\!+ \quad [x]))\ \textbf{in} \qquad r\ [1,2] \\
\textbf{let}\ r' = \textbf{fix}\ (\lambda f.\mathit{list}\ \epsilon\ \ (\lambda x\ xs.f\ xs\ \diamond\ \mathit{rep}\ [x]))\ \textbf{in}\ \mathit{abs}\ (r'\ [1,2])
\end{array}$$

In lieu of pattern matching (i.e. with **case** in Haskell), we use the list eliminator:

$$\mathit{list} : \forall a\ b.(a \to [a] \to b) \to [a] \to b$$

Note, as we mentioned in Section 1.1, how the transformation in r' extends beyond the function definition. An example similar to reverse is the concat function (c):

$$\begin{array}{l}
\textbf{let}\ c\ = \textbf{fix}\ (\lambda f.\mathit{list}\ [\,]\ (\lambda x\ xs. \qquad x +\!\!+ f\ xs))\ \textbf{in} \qquad c\ [[0],[1]] \\
\textbf{let}\ c' = \textbf{fix}\ (\lambda f.\mathit{list}\ \epsilon\ \ (\lambda x\ xs.\mathit{rep}\ x\ \diamond\ f\ xs))\ \textbf{in}\ \mathit{abs}\ (c'\ [[0],[1]])
\end{array}$$

We have run the examples from this section through our implementation and confirmed the results, but these examples just touch the surface of how much a transformation can change a program. For example, by changing the function *list* to a difference list eliminator *dlist*, we can also change the types of the inputs to these functions. In a related paper, van Eekelen et al. [31] explore the options on transforming patterns and constructors.

8. Other Applications

In this section, we describe two more applications of type-and-transform systems in the fashion of Section 1.1. With each concrete example, we give the rewrite rules for the transformation as in Section 7.4.

8.1 Generalization

Software reuse means writing code that can used more than once. One technique for doing this is generalizing the code: abstracting over the details to create code that can be instantiated in more places.

Scenario Pat writes a program using type A. It solves the problem for the moment, but Pat realizes that it would be useful to have a type $B\ T$, where B is some parameterized type and T is the argument that would instantiate a type isomorphic to A. This would would be useful for using functions defined on B and even for instantiated B with another argument.

Pat instructs the IDE (or command-line tool) to transform A-terms to $B\ T$-terms using the type-and-transform system. Now, Pat can begin using the benefits of B.

Typical Examples Trivial transformations include changing a specialized *IntList* to $[Int]$ or $[Int]$ to *Tree Int* (a rose tree of *Int*s). A more interesting example is transforming a datatype to a type class, e.g. *String* to (roughly) $StringLike\ a \Rightarrow a$, assuming there is an instance of *StringLike* for *String*. In other words, the methods of the type class *StringLike* are smart constructors, and we are not changing the type so much as changing the terms that construct and use the type. After transformation, *String* can be substituted with another type that has an instance of *StringLike*.

Transforming specialized code to datatype-generic code is an example of this scenario. In datatype-generic programming (DGP), the structure of a datatype is represented by a collection of other types, isomorphic to the original datatype [10]. (In the scenario, T is the structure representation in $B\ T$.)

Many generic functions are available with DGP libraries. Some libraries hide their representation from the user but some require users to program with it, often using smart constructors [19, 29]. We present a simplified example as a case study.

Example: Fixed-Point of Base Functors A regular datatype in Haskell can be represented as the fixed point of a base functor. For example, the datatype Exp_F is the base functor of *Exp*:

$$\begin{aligned}&\textbf{data}\ Exp \quad\ \ = Val\ \ Int \mid Add\ \ Exp\ Exp\\ &\textbf{data}\ Exp_F\ r = Val_F\ Int \mid Add_F\ r\ \ \ r\end{aligned}$$

Exp_F is a simple copy of the datatype with every recursive position replaced by a fresh type parameter r. The fixed point of Exp_F is defined using a datatype *Fix* that embodies recursion in the type:

$$\begin{aligned}&\textbf{newtype}\ Fix\ f = In\ \{out :: f\ (Fix\ f)\}\\ &\textbf{type}\ FExp = Fix\ Exp_F\end{aligned}$$

Given a *Functor* instance of Exp_F, natural recursion on regular datatypes is defined by the fold (or catamorphism):

$$\begin{aligned}&fold :: Functor\ f \Rightarrow (f\ a \to a) \to Fix\ f \to a\\ &fold\ alg = alg \circ fmap\ (fold\ alg) \circ out\end{aligned}$$

The types *Exp* and *FExp* are isomorphic (modulo undefined values):

$$\begin{aligned}&from :: Exp \to FExp\\ &from\ (Val\ i) \qquad\ \ = val\ i\\ &from\ (Add\ e_1\ e_2) = add\ (from\ e_1)\ (from\ e_2)\end{aligned}$$

$$\begin{aligned}&to :: FExp \to Exp\\ &to\ (In\ (Val_F\ i)) \qquad\ = Val\ i\\ &to\ (In\ (Add_F\ e_1\ e_2)) = Add\ (to\ e_1)\ (to\ e_2)\end{aligned}$$

Rather than construct *FExp* terms directly, as in:

$$three = In\ (Add_F\ (In\ (Val_F\ 1))\ (In\ (Val_F\ 2)))$$

we use smart constructors:

$$\begin{aligned}&val :: Int \to FExp\\ &val\ i = In\ (Val_F\ i)\end{aligned}$$

$$\begin{aligned}&add :: FExp \to FExp \to FExp\\ &add\ e_1\ e_2 = In\ (Add_F\ e_1\ e_2)\end{aligned}$$

As an additional convenience, we define a specialized fold for *FExp*:[11]

$$\begin{aligned}&foldFExp :: (Int \to r) \to (r \to r \to r) \to FExp \to r\\ &foldFExp\ v\ a = fold\ alg\\ &\quad \textbf{where}\ alg\ (Val_F\ i) \qquad = v\ i\\ &\qquad\qquad\ \ alg\ (Add_F\ r_1\ r_2) = a\ r_1\ r_2\end{aligned}$$

To contrast the recursion styles of *Exp* and *FExp*, we show the evaluation function for each:

$$\begin{aligned}&eval :: Exp \to Int\\ &eval\ (Val\ i) \qquad\ = i\\ &eval\ (Add\ e_1\ e_2) = eval\ e_1 + eval\ e_2\end{aligned}$$

$$\begin{aligned}&eval_F :: FExp \to Int\\ &eval_F = foldFExp\ id\ (+)\end{aligned}$$

Transformation The *Exp*-to-*FExp* transformation involves two operation classes:

1. Rewriting built-in constructors to their smart-constructor analogs: *Val* becomes *val* and *Add* becomes *add*
2. Inserting conversions where necessary: *to* and *from* are used to avoid type mismatches

At first glance, it may seem that the conversions are unnecessary. That is, it is a simple matter of rewriting the constructors without needing *to* or *from*. But consider other *Exp*-functions that are not generic: e.g. $isVal :: Exp \to Bool$ or $five :: Exp$. These cannot be transformed or discarded, and the conversions are necessary to use them. The example in Section 8.2 has more of these sorts of functions.

The type *FExp* gives us generic functions to use, and the construction is now hidden behind smart constructors. This provides the opportunity to implement more generic functionality such as our previous work on incrementalization [19].

In the object language, the typed rewrite rules are:

$$\{m : Exp\} \rhd m \quad : Exp \qquad\qquad \rightsquigarrow from\ m : \iota \tag{13}$$

$$\{m : \iota\} \rhd m \quad : \iota \qquad\qquad\quad \rightsquigarrow to\ m \quad : Exp \tag{14}$$

$$\varepsilon \rhd Val\ : Int \to Exp \qquad\quad \rightsquigarrow val \qquad : Int \to \iota \tag{15}$$

$$\varepsilon \rhd Add : Exp \to Exp \to Exp \rightsquigarrow add \qquad : \iota \to \iota \to \iota \tag{16}$$

To the rename rules (15) and (16), we assign the scores 0 and -2, respectively.

[11] We can, of course, define *foldExp* just as easily, but there are other approaches, e.g. pattern functors [35], that can provide convenient folds for free. For the sake of simplicity, we present only the base-functor approach.

`complex-rect`	$rect$	$:: Float \to Float \to Rect$
	$real$	$:: Rect \to Float$
	$imag$	$:: Rect \to Float$
	$+_R, \times_R$	$:: Rect \to Rect \to Rect$
`complex-polar`	$polar$	$:: Float \to Float \to Pol$
	mag	$:: Pol \to Float$
	$phase$	$:: Pol \to Float$
	$+_P, \times_P$	$:: Pol \to Pol \to Pol$

Figure 9. Interfaces of two libraries for complex numbers

8.2 Integration

Software development sometimes requires using multiple libraries with variations on the same concepts. Type-and-transform systems can assist in integrating these libraries.

Scenario Pat has two libraries with the respective types A and B that denote the "same" idea but serve different purposes (e.g. by having different APIs). Pat prefers type A in one part of the code and type B in a different part, but Pat still needs to translate As to Bs and vice versa, so that the parts stay connected.

To transform a part of a program, Pat selects a well-scoped subprogram, such as one or more modules, and directs a type-and-transform tool to transform that subprogram. This leaves the rest of the program untouched.

Typical Examples Time is often implemented in different ways: Unix system time, clock time, timestamps (e.g. for NTP), etc. Calendar dates are defined with numerous standards: Gregorian, Hijri, Gujarati, etc. Multiple data representations are common: consider the various representations of XML, JSON, and other serialization formats.

Example: Complex Numbers As a simple example, we consider integrating two libraries, presented in Figure 9, for representing complex numbers [11, 33]. The library `complex-rect` uses the rectangular (Cartesian) coordinate system with the *Rect* type, and the library `complex-polar` uses the polar coordinate system with the *Pol* type.

Each library has a function (*rect* or *polar*) for constructing a value of its type from *Float*s, though the arguments naturally have different meanings. The components of the *Rect* representation are provided by *real* and *imag*, while the components of *Pol* are provided by *mag* and *phase*. Both libraries have analogous functions for performing addition and multiplication. If the libraries do not provide conversion functions, we must write them:

$$asPol :: Rect \to Pol$$
$$asRec :: Pol \to Rect$$

Transformation Suppose that we need a transformation to change *Rect*-terms to *Pol*-terms. The typed rewrite rules are:

$$\{m : Rect\} \rhd m \;: Rect \leadsto asPol\ m : \iota \tag{17}$$
$$\{m : \iota\} \rhd m \;: \iota \leadsto asRec\ m : Rect \tag{18}$$
$$\varepsilon \rhd +_R : Rect \to Rect \to Rect \leadsto +_P \;: \iota \to \iota \to \iota \tag{19}$$
$$\varepsilon \rhd \times_R : Rect \to Rect \to Rect \leadsto \times_P \;: \iota \to \iota \to \iota \tag{20}$$

Some functions do not have analogs. In the transformed program, they may end up using the isomorphism functions: e.g. *rect* becomes $asPol \circ rect$ and *real* becomes $real \circ asRec$. As with previous examples, we assign the score -2 to each of the rename rules (19) and (20).

9. Related Work

Program transformation is studied in many contexts, and there is a vast amount of related work. In this section, we identify a subset of the work that is most relevant and compare it to type-and-transform systems.

Term rewriting is a technique that has been extensively applied to program transformation. Stratego [32] is a well-known language and tool set for program transformation using rewriting. It is representative of strategy languages in which many transformations can be specified. With standard term rewriting, it appears to be difficult to support type-changing rewrite rules while preserving type safety and semantics. Type-and-transform systems can perhaps be viewed as an adaptation of term rewriting.

Some applications of type-and-transform systems can be considered refactoring or interactive program transformation. HaRe [20] is a Haskell refactoring tool that supports a number of of automatic refactorings; however, it does not provide type-changing rewriting for whole-program transformations. Other tool-supported equational reasoning approaches include PATH [30] and HERMIT [7], both of which do not appear to facilitate type-changing rewriting. Nonetheless, it may be possible to build a type-and-transform system onto one of the above systems.

Erwig and Ren [6] define an update calculus, whose capabilities include rewrites and scope changes as well as update composition, alternation, and recursion. Their type-change system ensures that an update preserves type correctness for many type-changing transformations. The update calculus is intended for some type-changing updates; however, it does not have a mechanism for propagating type changes through bound variables. We were unable to specify any of our examples in the update calculus. On the other hand, a key feature of the update calculus is its support for scope changes, something that type-and-transform systems do not allow. It appears that type-and-transform systems and the update calculus complement each other.

One might see our approach as a type-and-effect system [12] if one views the transformation as a side effect of an extended type system. However, that analogy is stretched rather thin. We do not modify how the type system works, but instead derive from the type a type functor that relates programs using the underlying type system.

Cunha and Visser [4] describe a strongly typed rewriting system for calculating transformations that change both the structure of types and terms. They use a point-free program calculus with one constructor for pointwise functions over which no transformation is done. We do not distinguish different forms of syntax: all functions in the lambda calculus can be transformed. Type-and-transform systems, on the other hand, do not provide strategies for rewriting: the type changes drive the rewriting.

Coercions are functions inserted into a program to change terms from one type to a subsuming type. Kießling and Luo [18] define coercions in a Hindley-Milner type system using subtyping instead of an isomorphism between types. Their coercions serve a similar purpose to our rewrite rules, though the latter are slightly more general. Our notion of a complete transformation is loosely related to their idea of completion. Swamy et al. [28] describe type-directed coercion insertion in simply-typed lambda calculus with a focus on non-ambiguity. Our work takes advantage of ambiguity (via multiple rewrites) to find the "best" transformation. One primary difference between coercions and type-and-transform systems is that the latter allow for type changes to propagate through bindings while the former restrict type changes to function application.

10. Conclusions and Future Work

This paper introduces type-and-transform systems: automatic program transformation with type-changing rewriting that is type-safe and semantics-preserving. The type-and-transform system of a programming language is the specification of transformations, derived from the language's type system, and typed rewrite rules, which change terms and types in a regular fashion. We described the type-and-transform system for the lambda calculus with let-polymorphism and general recursion, and we proved that a complete transformation preserves typing and semantics.

We continue to investigate and refine type-and-transform systems. As stated in Section 1, we are working on improving our proof technique. There are connections from type-and-transform systems to abstraction [25], representation independence [23], and parametricity [34]. For example, we might consider ι as a special free variable and treat the type as a relation on types that instantiate ι differently. The connection to parametricity is not immediate, however. In parametricity, the type relation $\forall a.[a] \to [a]$ holds for *any* type relation instantiated for a. In type-and-transform systems, the same type relation holds only if the instantiating types are isomorphic. We will explore these connections in more depth in future work.

We plan to expand the model of type-and-transform systems to allow for transformation between a larger variety of types. We also want to describe transformation sequences and transformations with multiple types.

Type-and-transform systems may also be applicable to compilers, e.g. for whole-program optimization. We have done preliminary work with System F, and we will look into System FC, the core language of GHC.

This paper used a toy language to explain the theory and prove properties. We plan to build on this foundation by developing the theory for larger object languages such as Haskell and writing tools to experiment with real-world programs and investigate practical aspects of type-and-transform systems such as transformation effectiveness, algorithm performance, and choice heuristics.

Acknowledgments

We are grateful for the many constructive conversations with Joeri van Eekelen, Jurriaan Hage, José Pedro Magalhães, Bastiaan Heeren, Stefan Holdermans, Patrik Jansson, Jan Rochel, and Doaitse Swierstra. We also thank the Dutch Haskell Users Group and the anonymous reviewers for their very helpful suggestions.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] R. C. Backhouse, M. Bijsterveld, R. v. Geldrop, and J. v. d. Woude. Categorical Fixed Point Calculus. In *Proc. of CTCS*, pages 159–179, 1995.

[3] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell Strings. In *Proc. of PADL*, pages 50–64, 2007.

[4] A. Cunha and J. Visser. Strongly Typed Rewriting For Coupled Software Transformation. In *Proc. of RULE*, pages 17–34, 2006.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of POPL*, pages 207–212, 1982.

[6] M. Erwig and D. Ren. An update calculus for expressing type-safe program updates. *Sci. Comput. Program.*, 67:199–222, 2007.

[7] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Proc. of Haskell*, pages 1–12. ACM, 2012.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] P. Freyd. Recursive Types Reduced to Inductive Types. In *Proc. of LICS*, pages 498–507, 1990.

[10] J. Gibbons. Datatype-Generic Programming. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Proc. of Spring School on Datatype-Generic Programming*, pages 1–71, 2007.

[11] J. Gibbons. Unfolding Abstract Datatypes. In *Proc. of MPC*, pages 110–133, 2008.

[12] D. K. Gifford and J. M. Lucassen. Integrating Functional and Imperative Programming. In *Proc. of LFP*, pages 28–38, 1986.

[13] A. Gill and G. Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19(2):227–251, 2009.

[14] T. Harper. Stream Fusion on Haskell Unicode Strings. In *Proc. of IFL*, pages 125–140, 2011.

[15] R. Hinze. Polytypic Values Possess Polykinded Types. In *Proc. of MPC*, pages 2–27, 2000.

[16] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, Mar. 2006.

[17] R. J. M. Hughes. A Novel Representation of Lists and Its Application to the Function "reverse". *Inf. Process. Lett.*, 22(3):141–144, 1986.

[18] R. Kießling and Z. Luo. Coercions in Hindley-Milner Systems. In *Proc. of TYPES*, pages 259–275, 2003.

[19] S. Leather, A. Löh, and J. Jeuring. Pull-Ups, Push-Downs, and Passing It Around: Exercises in Functional Incrementalization. In *Proc. of IFL*, pages 159–178, 2011.

[20] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In *Proc. of Haskell*, pages 27–38, 2003.

[21] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proc. of FPCA*, pages 324–333, 1995.

[22] R. Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[23] J. C. Mitchell. Representation independence and data abstraction. In *Proc. of POPL*, pages 263–276, 1986.

[24] A. M. Pitts. Parametric Polymorphism and Operational Equivalence. *Math. Struct. in Comp. Science*, 10:321–359, 2000.

[25] J. C. Reynolds. Types, Abstraction and Parametric Polymorphism. *Information Processing*, pages 513–523, 1983.

[26] B. Schuur. A Type-Changing, Semantics-Preserving Program Transformation System. Master's thesis, Department of Information and Computing Sciences, Utrecht University, February 2013.

[27] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Softw: Pract. Exper.*, 12(11):1082–1084, 1982.

[28] N. Swamy, M. Hicks, and G. M. Bierman. A Theory of Typed Coercions and Its Applications. In *Proc. of ICFP*, pages 329–340, 2009.

[29] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[30] M. Tullsen. *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, 2002.

[31] J. van Eekelen, S. Leather, and J. Jeuring. Type-Changing Program Transformations with Pattern Matching. In *Proc. of the Workshop on Haskell And Rewriting Techniques (HART) 2013*, 2013.

[32] E. Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *J. Symb. Comput.*, 40(1):831–873, 2005.

[33] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. of POPL*, pages 307–313, 1987.

[34] P. Wadler. Theorems for free! In *Proc. of FPCA*, pages 347–359, 1989.

[35] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proc. of ICFP*, pages 233–244, 2009.

A. Example Proofs

In Section 5.2, we mentioned the following typed rewrite rule:

$$\{m : \iota\} \triangleright (\mathit{abs}\ m{+\!\!+}) : S \to S \leadsto (m\diamond) : \iota \to \iota$$

The corresponding (D-REW) property is:

$$\mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\mathit{abs}\ m{+\!\!+}) \equiv \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(m\diamond)$$

The proof of this property follows:

$$\begin{array}{ll}
 & \mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\mathit{abs}\ m{+\!\!+}) \\
\equiv & \{\ \text{apply } \mathcal{D}_{\{m:\iota\}}\ \} \\
 & \mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ (\mathit{abs}\ (\mathit{rep}\ m){+\!\!+}) \\
\equiv & \{\ (\mathit{abs\text{-}rep})\ \} \\
 & \mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ (m{+\!\!+}) \\
\equiv & \{\ \text{apply } \mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ \} \\
 & (m{+\!\!+}) \\
\equiv & \{\ \eta\text{-conversion}\ \} \\
 & \lambda y.m + \!\!+ y \\
\equiv & \{\ \text{section}\ \} \\
 & \lambda y.(m{+\!\!+})\ y \\
\equiv & \{\ \texttt{""} \text{ is the unit for } {+\!\!+}\ \} \\
 & \lambda y.(m{+\!\!+})\ (y + \!\!+ \texttt{""}) \\
\equiv & \{\ \beta\text{-conversion, section}\ \} \\
 & \lambda y.(\lambda x.(m{+\!\!+})\ ((y{+\!\!+})\ x))\ \texttt{""} \\
\equiv & \{\ \text{apply } \circ\ \} \\
 & \lambda y.((m{+\!\!+}) \circ (y{+\!\!+}))\ \texttt{""} \\
\equiv & \{\ \text{apply } \mathit{abs}\ \} \\
 & \lambda y.\mathit{abs}\ (Z\ ((m{+\!\!+}) \circ (y{+\!\!+}))) \\
\equiv & \{\ \text{apply } \diamond\ \} \\
 & \lambda y.\mathit{abs}\ (Z\ (m{+\!\!+}) \diamond Z\ (y{+\!\!+})) \\
\equiv & \{\ \text{apply } \mathit{rep}\ \} \\
 & \lambda y.\mathit{abs}\ (\mathit{rep}\ m \diamond \mathit{rep}\ y) \\
\equiv & \{\ \beta\text{-conversion}\ \} \\
 & \lambda y.\mathit{abs}\ ((\lambda x.\mathit{rep}\ m \diamond \mathit{rep}\ x)\ y) \\
\equiv & \{\ \eta\text{-conversion}\ \} \\
 & \mathit{abs} \circ (\lambda x.\mathit{rep}\ m \diamond \mathit{rep}\ x) \\
\equiv & \{\ \text{section}\ \} \\
 & \mathit{abs} \circ (\lambda x.(\mathit{rep}\ m\diamond)\ (\mathit{rep}\ x)) \\
\equiv & \{\ \text{apply } \circ\ \} \\
 & \mathit{abs} \circ (\mathit{rep}\ m\diamond) \circ \mathit{rep} \\
\equiv & \{\ \text{apply } \mathcal{D}_{\iota}\ \mathit{rep}\ \mathit{abs} \text{ and } \mathcal{D}_{\iota}\ \mathit{abs}\ \mathit{rep}\ \} \\
 & \mathcal{D}_{\iota}\ \mathit{rep}\ \mathit{abs} \circ (\mathit{rep}\ m\diamond) \circ \mathcal{D}_{\iota}\ \mathit{abs}\ \mathit{rep} \\
\equiv & \{\ \text{apply } \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \} \\
 & \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ (\mathit{rep}\ m\diamond) \\
\equiv & \{\ \text{apply } \mathcal{D}_{\{m:\iota\}}\ \} \\
 & \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(m\diamond)
\end{array}$$

B. Proof of Transformation Semantics

This appendix gives the full proof of Theorem 2: a transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}$ satisfies (D-TRANS), repeated here:

$$e \equiv \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e'$$

The proof proceeds by inference rule induction on the derivations. We include the relevant rule with each case for convenient reference.

CASE (T-VAR) $x \equiv \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}x$

$$\frac{\mathring{\tau} \prec \mathring{\Gamma}(x)}{\mathring{\Gamma} \vdash x \overset{\mathcal{R}}{\leadsto} x : \mathring{\tau}}\ \text{(T-VAR)}$$

$$\begin{array}{ll}
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}x \\
\equiv & \{\ \text{expand } \mathcal{D}_{\mathring{\Gamma}},\ x : \mathring{\varsigma} \in \mathring{\Gamma}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}',x:\mathring{\varsigma}}x \\
\equiv & \{\ \text{apply } \mathcal{D}_{\mathring{\Gamma}',x:\mathring{\varsigma}}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}'} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}'}\ \mathit{abs}\ \mathit{rep}\ x])x \\
\equiv & \{\ \text{apply substitution, } x : \mathring{\varsigma} \notin \mathring{\Gamma}'\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}'}\ \mathit{abs}\ \mathit{rep}\ x) \\
\equiv & \{\ \text{apply } \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}'},\ \mathring{\tau} \prec \mathring{\varsigma}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x) \\
\equiv & \{\ \text{(D-COMP)}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ (\mathit{abs} \circ \mathit{rep})\ (\mathit{abs} \circ \mathit{rep})\ x \\
\equiv & \{\ (\mathit{abs\text{-}rep})\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{id}\ \mathit{id}\ x \\
\equiv & \{\ \text{(D-ID)}\ \} \\
 & x
\end{array}$$

■

CASE (T-FIX) $\mathbf{fix}\ e \equiv \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{fix}\ e')$

$$\frac{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau} \to \mathring{\tau}}{\mathring{\Gamma} \vdash \mathbf{fix}\ e \overset{\mathcal{R}}{\leadsto} \mathbf{fix}\ e' : \mathring{\tau}}\ \text{(T-FIX)}$$

$$\begin{array}{ll}
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{fix}\ e') \\
\equiv & \{\ \mathcal{D}_{\mathring{\Gamma}} \text{ distributes over } \mathbf{fix}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ \mathcal{D}_{\mathring{\Gamma}}e') \\
\equiv & \{\ \mathit{id} \text{ is the unit for } \circ\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}}e' \circ \mathit{id})) \\
\equiv & \{\ \text{(D-ID)}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}}e' \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{id}\ \mathit{id})) \\
\equiv & \{\ (\mathit{rep\text{-}abs})\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}}e' \circ \mathcal{D}_{\mathring{\tau}}\ (\mathit{rep} \circ \mathit{abs})\ (\mathit{rep} \circ \mathit{abs}))) \\
\equiv & \{\ \text{(D-COMP)}\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}}e' \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep} \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs})) \\
\equiv & \{\ \text{associativity of } \circ\ \} \\
 & \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{fix}\ ((\mathcal{D}_{\mathring{\Gamma}}e' \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}) \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs})) \\
\equiv & \{\ \text{(RED-ROLLING)}\ \} \\
 & \mathbf{fix}\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs} \circ \mathcal{D}_{\mathring{\Gamma}}e' \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}) \\
\equiv & \{\ \text{apply } \mathcal{D}_{\mathring{\tau}\to\mathring{\tau}}\ \} \\
 & \mathbf{fix}\ (\mathcal{D}_{\mathring{\tau}\to\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e') \\
\equiv & \{\ e \equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e'\ \} \\
 & \mathbf{fix}\ e
\end{array}$$

■

CASE (T-APP) $e_1\ e_2 \equiv \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(e'_1\ e'_2)$

$$\frac{\mathring{\Gamma} \vdash e_1 \overset{\mathcal{R}}{\leadsto} e'_1 : \mathring{\tau} \to \mathring{\upsilon} \qquad \mathring{\Gamma} \vdash e_2 \overset{\mathcal{R}}{\leadsto} e'_2 : \mathring{\tau}}{\mathring{\Gamma} \vdash e_1\ e_2 \overset{\mathcal{R}}{\leadsto} e'_1\ e'_2 : \mathring{\upsilon}}\ \text{(T-APP)}$$

$$\begin{array}{ll}
 & \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(e'_1\ e'_2) \\
\equiv & \{\ \mathcal{D}_{\mathring{\Gamma}} \text{ distributes over application}\ \} \\
 & \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}}e'_1\ \mathcal{D}_{\mathring{\Gamma}}e'_2) \\
\equiv & \{\ \text{(D-ID)}\ \} \\
 & \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}}e'_1\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{id}\ \mathit{id}\ \mathcal{D}_{\mathring{\Gamma}}e'_2)) \\
\equiv & \{\ (\mathit{rep\text{-}abs})\ \} \\
 & \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}}e'_1\ (\mathcal{D}_{\mathring{\tau}}\ (\mathit{rep} \circ \mathit{abs})\ (\mathit{rep} \circ \mathit{abs})\ \mathcal{D}_{\mathring{\Gamma}}e'_2)) \\
\equiv & \{\ \text{(D-COMP)}\ \} \\
 & \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}}e'_1\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e'_2))) \\
\equiv & \{\ \text{apply } \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \} \\
 & \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e'_1\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}e'_2)
\end{array}$$

$\equiv\quad \{\ e_1 \equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1,\ e_2 \equiv \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_2\ \}$

$e_1\ e_2$

■

CASE (T-LAM) $\lambda x.e \equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\lambda x.e')$

$$\frac{\mathring{\Gamma}, x:\mathring{\tau} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\upsilon}}{\mathring{\Gamma} \vdash \lambda x.e \overset{\mathcal{R}}{\leadsto} \lambda x.e' : \mathring{\tau} \to \mathring{\upsilon}}\ \text{(T-LAM)}$$

$\mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\lambda x.e')$

$\equiv\quad \{\ \mathcal{D}_{\mathring{\Gamma}}$ distributes over λ since $x:\mathring{\tau} \notin \mathring{\Gamma}\ \}$

$\mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\lambda x.\mathcal{D}_{\mathring{\Gamma}} e')$

$\equiv\quad \{$ apply $\mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs} \circ (\lambda x.\mathcal{D}_{\mathring{\Gamma}} e') \circ \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}$

$\equiv\quad \{$ apply $\circ\ \}$

$\lambda x.\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ ((\lambda x.\mathcal{D}_{\mathring{\Gamma}} e')\ (\mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x))$

$\equiv\quad \{$ (RED-LAM) $\}$

$\lambda x.\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ [x \mapsto \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x]\mathcal{D}_{\mathring{\Gamma}} e'$

$\equiv\quad \{$ substitution composition $\}$

$\lambda x.\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ ([x \mapsto \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x] \circ \mathcal{D}_{\mathring{\Gamma}})e'$

$\equiv\quad \{$ commute $\circ$ since $x:\mathring{\tau} \notin \mathring{\Gamma}\ \}$

$\lambda x.\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x])e'$

$\equiv\quad \{$ apply $\mathcal{D}_{\mathring{\Gamma},x:\mathring{\tau}}\ \}$

$\lambda x.\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma},x:\mathring{\tau}} e'$

$\equiv\quad \{\ e \equiv \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma},x:\mathring{\tau}} e'\ \}$

$\lambda x.e$

■

CASE (T-LET) $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \equiv \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{let}\ x = e'_1\ \mathbf{in}\ e'_2)$

$$\frac{\mathring{\Gamma} \vdash e_1 \overset{\mathcal{R}}{\leadsto} e'_1 : \mathring{\tau} \qquad \mathring{\Gamma}, x:\mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}) \vdash e_2 \overset{\mathcal{R}}{\leadsto} e'_2 : \mathring{\upsilon}}{\mathring{\Gamma} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \overset{\mathcal{R}}{\leadsto} \mathbf{let}\ x = e'_1\ \mathbf{in}\ e'_2 : \mathring{\upsilon}}\ \text{(T-LET)}$$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{let}\ x = e'_1\ \mathbf{in}\ e'_2)$

$\equiv\quad \{\ \mathcal{D}_{\mathring{\Gamma}}$ distributes over **let** since $x:\mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau}) \notin \mathring{\Gamma}\ \}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathbf{let}\ x = \mathcal{D}_{\mathring{\Gamma}} e'_1\ \mathbf{in}\ \mathcal{D}_{\mathring{\Gamma}} e'_2)$

$\equiv\quad \{$ (RED-LET) $\}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ [x \mapsto \mathcal{D}_{\mathring{\Gamma}} e'_1]\mathcal{D}_{\mathring{\Gamma}} e'_2$

$\equiv\quad \{$ substitution composition $\}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ ([x \mapsto \mathcal{D}_{\mathring{\Gamma}} e'_1] \circ \mathcal{D}_{\mathring{\Gamma}})e'_2$

$\equiv\quad \{$ (D-ID), $\mathring{\varsigma} = \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau})\ \}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{id}\ \mathit{id}\ \mathcal{D}_{\mathring{\Gamma}} e'_1] \circ \mathcal{D}_{\mathring{\Gamma}})e'_2$

$\equiv\quad \{$ (*rep-abs*) $\}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}$

$\quad ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ (\mathit{rep} \circ \mathit{abs})\ (\mathit{rep} \circ \mathit{abs})\ \mathcal{D}_{\mathring{\Gamma}} e'_1] \circ \mathcal{D}_{\mathring{\Gamma}})e'_2$

$\equiv\quad \{$ (D-COMP) $\}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}$

$\quad ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ (\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1)] \circ \mathcal{D}_{\mathring{\Gamma}})e'_2$

$\equiv\quad \{$ split substitution $\}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}$

$\quad ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1] \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ x] \circ \mathcal{D}_{\mathring{\Gamma}})e'_2$

$\equiv\quad \{$ commute $\circ$ since $x:\mathring{\varsigma} \notin \mathring{\Gamma}\ \}$

$\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}$

$\quad ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1] \circ \mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ x])e'_2$

$\equiv\quad \{\ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1]$ distributes over application $\}$

$[x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1]$

$\quad (\mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ x])e'_2)$

$\equiv\quad \{$ (RED-LET) $\}$

$\mathbf{let}\ x = \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1$

$\mathbf{in}\ \ \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ x])e'_2$

$\equiv\quad \{$ apply $\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \}$

$\mathbf{let}\ x = \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1$

$\mathbf{in}\ \ \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ \mathit{abs}\ \mathit{rep}\ x])e'_2$

$\equiv\quad \{$ apply $\mathcal{D}_{\mathring{\Gamma}}\ \}$

$\mathbf{let}\ x = \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1\ \mathbf{in}\ \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma},x:\mathring{\varsigma}} e'_2$

$\equiv\quad \{\ e_1 \equiv \mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'_1,\ e_2 \equiv \mathcal{D}_{\mathring{\upsilon}}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma},x:\mathring{\varsigma}} e'_2\ \}$

$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$

■

CASE (T-REW) $e \equiv \mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} \theta p_r$

$$\frac{\begin{array}{c}(\mathring{\Gamma}_m \rhd p_l : \mathring{\tau}_l \leadsto p_r : \mathring{\tau}_r) \in \mathcal{R} \\ \mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} e' : \mathring{\tau}_l \qquad \mathring{\Gamma};\mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\leadsto} p_l @ e' \Rightarrow \theta\end{array}}{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\leadsto} \theta p_r : \mathring{\tau}_r}\ \text{(T-REW)}$$

$\mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} \theta p_r$

$\equiv\quad \{\ \theta$ commutes with $\mathcal{D}_{\mathring{\Gamma}}$ and distributes over application $\}$

$\theta(\mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} p_r)$

$\equiv\quad \{$ (D-REW) $\}$

$\theta(\mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} p_l)$

$\equiv\quad \{\ \theta$ commutes with $\mathcal{D}_{\mathring{\Gamma}}$ and distributes over application $\}$

$\mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} \theta p_l$

$\equiv\quad \{\ e' \equiv \theta p_l\ \}$

$\mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'$

$\equiv\quad \{\ e \equiv \mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}} e'\ \}$

e

■