Type-Changing Rewriting and Semantics-Preserving Transformation

Sean Leather

Johan Jeuring

Andres Löh

Bram Schuur

Technical Report UU-CS-2015-012
September 2015

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[a], Johan Jeuring[a,b], Andres Löh[c], Bram Schuur[a]

[a] *Utrecht University*
[b] *Open University Netherlands*
[c] *Well-Typed LLP*

Abstract

We have identified a class of whole-program transformations that are regular in structure and require changing the types of terms throughout a program while simultaneously preserving the initial semantics after transformation. This class of transformations cannot be safely performed with typical term rewriting techniques, which do not allow for changing the types of terms.

In this paper, we present a formalization of type-and-transform systems, an automated approach to the whole-program transformation of terms of one type to terms of a different, isomorphic type using type-changing rewrite rules. A type-and-transform system defines typing and semantics relations between all corresponding source and target subprograms such that a complete transformation guarantees that the whole programs have equivalent types and semantics. 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.

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

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.

Email addresses: s.p.leather@uu.nl (Sean Leather), j.t.jeuring@uu.nl (Johan Jeuring), andres@well-typed.com (Andres Löh), bramschuur@gmail.com (Bram Schuur)

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 *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). A target is actually the possibly modified source. A type-and-transform system also specifies how a program can be modified with a *typed rewrite rule*, an extension of the usual term rewrite rule that can, under certain conditions, impose a change of type between its left-hand side (lhs) and right-hand side (rhs) patterns.

A transformation reflects the structure of the source term, preserving both the syntactic relation of corresponding subterms in the source and target and the typing relation of those subterms. A transformation also records all rewritings to the target by an associated set of typed rewrite rules. A *complete transformation*[2] is a transformation with the same source and target types and equivalent semantics, even though the programs may differ syntactically.

Figure 1 provides a visualization of the connections between transformation and rewriting. The diagram is split vertically to position the parts relevant to the source program on the left and the target program on the right. A program such as $e_s : \tau_s$ represents the term e_s – in this case, the source term – with its type τ_s. Transformations are horizontal, indicating the relation between source and target, and use an $\rightsquigarrow$ arrow. Applying a typed rewrite rule is a vertical step from one transformation to another with

[1] With apologies for the abuse of terminology, we have borrowed the terms "transformation" and "rewrite," among others, and given them specific meanings that differ from those in other contexts.

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

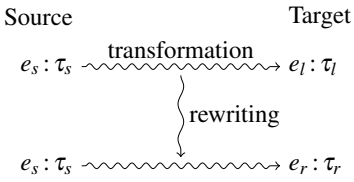


Figure 1: Diagram of the relationship between transformation and rewriting

an $\rightsquigarrow$ arrow.[3] With typed rewriting, the target term and type can change; however, the transformations "before" and "after" rewriting must each preserve a relation between its respective target and the same source. It is in this sense that typed rewriting relates two transformations rather than two terms, as is typical for term rewriting. In future sections, we will revisit the diagrammatic technique of Figure 1 to help elucidate the relationships between the components of transformation and rewriting.

The associated set of typed rewrite rules describes all the allowed term and type changes for a transformation. We use two metavariables, $\mathcal{A}$ and $\mathcal{R}$, to indicate the abstraction and representation types, respectively, which are the only types that can be changed. The basic conversion between these types is given by the functions $rep : \mathcal{A} \to \mathcal{R}$ and $abs : \mathcal{R} \to \mathcal{A}$.[4]

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

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

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

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

As an aside, we believe that the isomorphism requirement can be weakened to a retract – that is, only (ABS-REP) would be necessary. A retract would allow transformations between, for example, the types $\mathcal{A} = String$ and $\mathcal{R} = String \to String$, which do not have an isomorphism (see Section 1.1 for why). One of the authors has already shown the retract requirement to some extent. In a master's thesis, Schuur [29] demonstrated a type-and-transform system for the simply typed lambda calculus using a logical relation as proof technique. There is a precedence [27] for using logical relations for 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 an introduction to and foundation 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 serve as a running example. Pat, the programmer, will be our guide through various scenar-

[3] The intuition behind the arrows is that, where rewriting is a change or a "bump in the road" ($\rightsquigarrow$), a transformation may include a sequence of rewrites or multiple bumps ($\leadsto$).

[4] We adopted the use of $\mathcal{A}$/*abs* and $\mathcal{R}$/*rep* from Hughes [17].

ios describing problems and our solutions. For code examples, we use a Haskell-like language.

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.

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[5] 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 [32, 38] 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 applications such as $(xs +\!\!+ ys) +\!\!+ zs$ are inefficient: the structure of xs is effectively traversed twice during evaluation. Even though $+\!\!+$ is right-associative, we cannot easily guarantee that $+\!\!+$ is always used in a right-bracketed way, especially when abstraction is used, as in **let** $as = xs +\!\!+ ys$ **in** $as +\!\!+ zs$.

We present a "difference strings" library in Figure 2. Using this library, we would write the example $ex_1 = (xs +\!\!+ ys) +\!\!+ zs$ as $(rep\ xs \diamond rep\ ys) \diamond rep\ zs$, which reduces to $Z\,(\lambda as.xs +\!\!+ ys +\!\!+ zs +\!\!+ as)$. To that, we apply abs to get $xs +\!\!+ ys +\!\!+ zs +\!\!+$ "", a term equivalent (by the associative and unit properties of $+\!\!+$) to ex_1 but without the unnecessary extra work involved in the left-bracketed $+\!\!+$ chain.

This library is an adaptation of Hughes [17] with two modifications. First, we specialize to lists of characters (i.e. *String*). This allows us to simplify the initial presentation of type-and-transform systems. We extend the language with parameterized type

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

$$
\begin{array}{lll}
 & \mathit{rep} :: S \to Z & (\diamond) :: Z \to Z \to Z \\
\textbf{type } S = \mathit{String} & \mathit{rep}\ \mathit{xs} = Z\ (\mathit{xs} +\!\!+) & Z\,f \diamond Z\,g = Z\ (f \circ g) \\
\textbf{newtype } Z = Z\ (S \to S) & \mathit{abs} :: Z \to S & \epsilon :: Z \\
 & \mathit{abs}\ (Z\,f) = f\ \texttt{""} & \epsilon = Z\ \mathit{id}
\end{array}
$$

Figure 2: Difference strings library

constructors in Section 8 and discuss difference lists in Section 8.1. Second, we use a Haskell **newtype** Z for the difference string type, and we assume that the constructor is not visible outside the library (i.e. Z is an abstract type). Without this approach, we do not have an isomorphism. Note, for example, that $(\mathit{rep} \circ \mathit{abs})\ (Z\ (\lambda x.\texttt{"a"})) \equiv Z\ (\texttt{"a"} +\!\!+)$ but $(\lambda x.\texttt{"a"}) \not\equiv (\texttt{"a"} +\!\!+)$. There is no corresponding *String* value for every $\mathit{String} \to \mathit{String}$ function, so we cannot allow arbitrary functions in Z values.

With a type-and-transform system, we can automatically transform string code to difference string code. Not only can $(\mathit{xs} +\!\!+ \mathit{ys}) +\!\!+ \mathit{zs}$ be transformed to $\mathit{abs}\ ((\mathit{rep}\ \mathit{xs} \diamond \mathit{rep}\ \mathit{ys}) \diamond \mathit{rep}\ \mathit{zs})$, but transformation can be "pushed" through bindings. The example **let** $\mathit{as} = \mathit{xs} +\!\!+ \mathit{ys}$ **in** $\mathit{as} +\!\!+ \mathit{zs}$ can be transformed to abs (**let** $\mathit{as} = \mathit{rep}\ \mathit{xs} \diamond \mathit{rep}\ \mathit{ys}$ **in** $\mathit{as} \diamond \mathit{rep}\ \mathit{zs}$). Note that the type of as is changed by the transformation: from *String* to $\mathit{String} \to \mathit{String}$. The feature of bound variables with changing types is an important motivation for using a type-and-transform system instead of a term rewriting system.

For a larger example, we adopt the *reverse* function from Hughes [17]:

$$
\begin{array}{l}
\mathit{reverse} :: S \to S \\
\mathit{reverse}\ \texttt{""} \quad = \texttt{""} \\
\mathit{reverse}\ (x : \mathit{xs}) = \mathit{reverse}\ \mathit{xs} +\!\!+ (x : \texttt{""})
\end{array}
$$

There are many potential transformations of *reverse*. Here are two:[6]

$$
\begin{array}{l}
\mathit{reverse}_1 :: S \to Z \\
\mathit{reverse}_1\ \texttt{""} \quad = \epsilon \\
\mathit{reverse}_1\ (x : \mathit{xs}) = \mathit{reverse}_1\ \mathit{xs} \diamond \mathit{rep}\ (x : \texttt{""}) \\
\mathit{reverse}_2 :: S \to S \\
\mathit{reverse}_2\ \texttt{""} \quad = \texttt{""} \\
\mathit{reverse}_2\ (x : \mathit{xs}) = \mathit{abs}\ (\mathit{rep}\ (\mathit{reverse}_2\ \mathit{xs}) \diamond \mathit{rep}\ (x : \texttt{""}))
\end{array}
$$

We prefer $\mathit{reverse}_1$ over $\mathit{reverse}_2$ because (1) it has fewer uses of *abs* and *rep* and (2) it allows us to "propagate" the type Z further throughout the program by changing the type of *reverse* everywhere it is used. Even if there is only one use of *reverse*, the transformation would need to add only one use of *abs*, as in $\mathit{abs}\ (\mathit{reverse}\ e)$. We discuss the choice of one transformation over others in Section 7.2.

The above transformations are expressed using typed rewrite rules, which can be very simple and yet quite expressive. With one typed rewrite rule – e.g. rewrite $+\!\!+$ to

[6] These functions are named for reference. The transformation does not actually rename functions.

$\diamond$ – we describe a minimal change that is powerful because the types of the lhs and rhs are different. Term rewrite rules, by contrast, only include patterns with the same types – e.g. rewrite $xs +\!\!+ ys$ to $abs\ (rep\ xs +\!\!+ rep\ ys)$. For ex_1, this rewrite would produce a result, $abs\ (rep\ (abs\ (rep\ xs \diamond rep\ ys)) \diamond rep\ zs)$, that is rather more verbose than the aforementioned transformation of ex_1. (Of course, we can apply other rules to rewrite the term again, but that is unnecessary in type-and-transform systems.) Also, just the one typed rewrite rule allows us to transform $map\ (+\!\!+)$ to $map\ (\diamond)$ without mentioning *map*: type-and-transform systems take advantage of polymorphism and higher-order functions to propagate change.

As a programming abstraction, the difference string representation is clearly not as convenient as the string representation: it requires inserting *abs* and *rep* at strategic points. 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 automatic transformation to difference strings becomes very useful.

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 also extend the language to include parameterized type constructors.
- We establish and prove the properties necessary for preserving type safety and semantics through type-changing rewriting.
- We provide several examples demonstrating the application of type-and-transform systems.

We have developed a Haskell implementation [20] of the transformation algorithm for experimentation. But the primary focus of this paper is on formalization, and we only summarize the algorithm and practical considerations. Our ultimate goal is to extend the theory to Haskell, implement full support for the language, and investigate the real-world effectiveness of type-and-transform systems.

This article is an update of another paper published with the same name [22]. We have made a number of changes since that first paper. All previously omitted proofs have been included, and we give a detailed discussion of them. The diagrams have been improved, and new examples were added to visualize the relationship between transformation and rewriting. Finally, we expanded the analysis on the nuances of typed rewriting and incorporated an example of a transformation derivation tree.

1.3. Overview

The remainder of this paper is organized as follows. We begin in Section 2 with a discussion of the basic object language 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 in Section 4 and the semantics in Section 5. In Section 6, we present the formal definitions and correctness

Terms	e,f	$::= x \mid f\ e \mid \lambda x.e \mid \textbf{fix}\ e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$
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

proofs of the important concepts. Section 7 includes brief discussions of the transformation algorithm and practical aspects. We extend the language with parameterized type constructors in Section 8 and use that extension for a difference list transformation. In Section 9, we describe two more applications of type-and-transform systems. Finally, we examine related work in Section 10 and conclude with our future plans in Section 11.

2. Lambda Calculus with General Recursion and let-Polymorphism

Our object language is the lambda calculus with general recursion (a **fix** primitive) and polymorphic **let**-bindings with the Damas-Hindley-Milner type system [25, 5]. 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.

A type environment is a finite map from variables to type schemes. A variable is either an object variable x and or a syntactically distinct metavariable m. Metavariables appear only in rewrite rules for pattern matching on object terms (Section 4.2). 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. The notation $\varsigma = \Gamma(x)$ indicates that $x : \varsigma \in \Gamma$.

A type substitution[7] 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

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

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

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

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

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

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

Figure 4: Object language type system

$\tau \prec \varsigma$.

- 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:

$$(\lambda x.e_2)\ e_1 \equiv [x \mapsto e_1]e_2 \qquad (\textsc{Red-Lam})$$

$$\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \equiv [x \mapsto e_1]e_2 \qquad (\textsc{Red-Let})$$

$$\lambda x f\ x \equiv f \quad \text{where } x \notin fv(f) \qquad (\textsc{Red-Eta})$$

$$\mathbf{fix}\ (g \circ f) \equiv g\ (\mathbf{fix}\ (f \circ g)) \qquad (\textsc{Red-Rolling})$$

The first three[8] are reduction rules for a call-by-name semantics. The last equivalence, (RED-ROLLING), is the rolling rule, discussed in Section 6, for the least fixed point.

3. A Brief Look at Transformation

In this section, we look at a few transformations in our object language[9] 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.

[8] Not all of these rules are used in the main text. See also the proofs in Appendix A and Appendix B.

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

Table 1: Examples of transformations

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)

Consider the example transformations in Table 1. The first two involve 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 these changes the type of the term, but note that the type changes have a regular pattern. Every S becomes a Z, and the type function structure is preserved, i.e. the number of arrows and the relationships between them are the same in the source and target.

At this point, the reader might try finding a combination of *abs* and *rep* (and (RED-ETA)) for each example that changes the target to be semantically equivalent to the source. For example, (1) and (2) can have their targets rewritten such that the following equations hold:

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

Developing this intuition will help with understanding later concepts.

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

Multiple transformations can have the same source but different targets, 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 transformation in Section 7.2.

A transformation relates a source e_s to a target e_t but not necessarily the source e_t to the target e_s. That is, the relation is not symmetric. For example, *abs* and *rep* are only introduced and never eliminated; so, we cannot define a transformation relating a changed target to a source.

Examples (4), (5), and (6) are complete transformations, and the source and target have equal types. Complete transformations allow the target to be substituted for the source. Incomplete transformations such as (1), (2), and (3) can be subtransformations (i.e. transformations of subterms) of complete transformations, but they are not complete themselves.

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

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

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[10]. The definition 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 syntactically equal to $\sigma\tau$ and $\sigma\upsilon$. The more interesting case occurs where $\mathcal{U}(\tau,\upsilon)$ is not defined but $(\sigma,\mathring{\tau}) = \mathcal{T}'(\tau,\upsilon)$ is defined. In that case, an ι is found at every position in $\mathring{\tau}$ where $\mathcal{A}$ occurs at the corresponding position in $\sigma\tau$ and $\mathcal{R}$ occurs at the corresponding position in $\sigma\upsilon$.

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:[11]

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

[10] Figure 5 is simplified for clarity. The types $\mathcal{A}$ and $\mathcal{R}$ are implicit parameters and, to be more general, should not be patterns but checked for unification, i.e. with $\mathcal{U}(\tau,\mathcal{A})$.

[11] Note that $\_\langle\tau\rangle$ is surjective but not injective. For example, $\iota\langle S\rangle \equiv S\langle S\rangle$. This property will be important later.

Given the definitions of $\mathcal{T}$ and $\_\langle\_\rangle$, we can state, for any $\mathring{\tau}$, the following inversion property:

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

To prove this, we first show that $(id, \mathring{\tau}) \equiv \mathcal{T}'(\mathring{\tau}\langle\mathcal{A}\rangle, \mathring{\tau}\langle\mathcal{R}\rangle)$. The proof is by straightforward induction on the structure of the type functor $\mathring{\tau}$. Note that the substitutions in the proof are all *id*. This is because the types are equivalent except when $\mathring{\tau}$ is ι. In that case, $\mathcal{T}'(\mathring{\tau}\langle\mathcal{A}\rangle, \mathring{\tau}\langle\mathcal{R}\rangle) \equiv \mathcal{T}'(\iota\langle\mathcal{A}\rangle, \iota\langle\mathcal{R}\rangle) \equiv \mathcal{T}'(\mathcal{A}, \mathcal{R}) \equiv (id, \iota) \equiv (id, \mathring{\tau})$.

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*:

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

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

$$\begin{aligned} &\mathcal{T}_{\mathring{\Gamma}}(\varsigma_1, \varsigma_2) = \mathcal{G}_{\mathring{\Gamma}}(\mathcal{T}(\tau_1, \tau_2)) \textbf{ where } \tau_1 \prec \varsigma_1, \tau_2 \prec \varsigma_2 \\ &\mathcal{T}(\varepsilon \quad\quad\quad\quad, \varepsilon \quad\quad\quad\quad) = \varepsilon \\ &\mathcal{T}((\Gamma_1, v_1 : \varsigma_1), (\Gamma_2, v_2 : \varsigma_2)) = \textbf{let } \mathring{\Gamma} = \mathcal{T}(\Gamma_1, \Gamma_2) \textbf{ in } \mathring{\Gamma}, v_1 : \mathcal{T}_{\mathring{\Gamma}}(\varsigma_1, \varsigma_2) \textbf{ if } v_1 \equiv v_2 \end{aligned}$$

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

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

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 $\mathcal{T}(\Gamma_s, \Gamma_t) = \{x : \iota, \ldots\}$.

The inversion properties establish the source and target types and environments of a transformation. These are necessary for the typing of transformations as formalized by Theorem 1 in Section 6.

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 expression patterns, $p_l \rightsquigarrow p_r$, where p_l is the lhs, p_r is the rhs, and p has the following syntax:

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

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

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

$$\frac{\begin{array}{c}\Gamma \cup \mathring{\Gamma}\langle \mathcal{R} \rangle \vdash p_l : \mathring{\tau}_l\langle \mathcal{R} \rangle \\ \Gamma \cup \mathring{\Gamma}\langle \mathcal{R} \rangle \vdash p_r : \mathring{\tau}_r\langle \mathcal{R} \rangle \\ \mathring{\tau}_l\langle \mathcal{A} \rangle \equiv \mathring{\tau}_r\langle \mathcal{A} \rangle\end{array}}{\Gamma \vdash (\mathring{\Gamma} \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r)}\ \text{(RR)} \qquad \frac{\forall \rho \in \mathcal{R}.\Gamma \vdash \rho}{\Gamma \vdash \mathcal{R}}\ \text{(RS)}$$

Figure 6: Pattern (p), rule (ρ), and rule set ($\mathcal{R}$) typing

$$p \ ::= \ \nu \mid p_1\ p_2$$

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 a rewrite rule ρ by annotating patterns with type functors and annotating the rule itself with a type functor environment for the metavariables in the patterns:

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

We use $\mathcal{R}$ to denote a finite set of typed rewrite rules.

Consider the following typed rewrite rule set for our running example:

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

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

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

There is a simple intuition behind the type functors in the above rules: if a pattern or metavariable would have type Z (the representation type) under normal typing rules, that type is replaced with the "placeholder" ι.

One can view the use of ι as a "viral infection" that spreads throughout the program via rewriting. First, the infection is introduced by rewriting with (SZ-1), which has ι only in a basic (non-function) target type. Next, the infection is transmitted to other parts of the program by (SZ-3). Its target has a function type that has ι in both argument and result positions. Finally, we eliminate the infection with (SZ-2), which has ι only in a non-function source type and not in the target type.

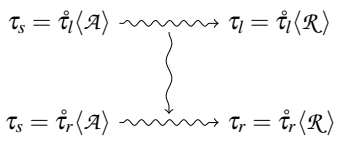


Figure 7: Diagram relating types to type functors for rewriting

Figure 6 presents typing rules for patterns, rewrite rules, and rule sets. The typing of patterns with $\Gamma \vdash p : \tau$ is standard, though it is worth noting that object variables (i.e. constants) and metavariables are treated equally in (P-VAR). The typing of a rule set $\mathcal{R}$ with $\Gamma \vdash \mathcal{R}$ is also straightforward: a rule set is well-typed if all of its rules are well-typed. However, the typing of a rule ρ with $\Gamma \vdash \rho$ needs some explanation.

The premises of the inference rule (RR) effectively define two conditions for typing rewrite rules. The first condition is that each pattern must be typed with its target type under the target type environment. Alternatively stated, a pattern p_l has the target type τ_l, which is equivalent to $\mathring{\tau}_l\langle\mathcal{R}\rangle$ (the type functor $\mathring{\tau}_l$ with ι substituted for the representation type $\mathcal{R}$). Note that the type functor environment $\mathring{\Gamma}$ of a rule closes only over the metavariables of the patterns; the object variables must be bound in the type environment Γ. The second condition is that the lhs and rhs source types must be equivalent:[12]

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

To understand ($\mathring{\tau}$-REW), recall the relationships between the terms and types of rewriting and transformation as depicted in Figure 1. We adapt that diagram in Figure 7 to show only the types and their equivalent type functor projections. Note how the type functors $\mathring{\tau}_l$ and $\mathring{\tau}_r$ are associated with the before and after transformations (recalling the inversion property ($\mathring{\tau}$-INV)). But the source types of each transformation must be equivalent; therefore, a typed rewrite rule must have the ($\mathring{\tau}$-REW) property.

The conditions for typing a rewrite rule ensure that the rule will preserve the typing of a transformation, which we describe in detail in the next section. Before continuing, however, the reader may wish to prove that each of the rules (SZ-1), (SZ-2), and (SZ-3) is well-typed according to $\Gamma \vdash \rho$ using an appropriate Γ derived from Figure 2.

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

[12] This is where the aforementioned non-injectivity of $\_\langle\tau\rangle$ is important.

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

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

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

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

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

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

Figure 8: Transformation

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 exception to the typing correspondence rule is (T-REW) for type-changing rewriting.

We would prefer to have a simple formulation of type-changing rewriting such as:

$$\frac{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} \theta p_l : \mathring{\tau}_l \qquad (\mathring{\Gamma}_m \rhd p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r) \in \mathcal{R}}{\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} \theta p_r : \mathring{\tau}_r}$$

Given a typed rewrite rule from the rule set $\mathcal{R}$ and a transformation whose target term is the instance of the lhs pattern p_l for some substitution θ, we can derive a transformation whose target term is the instance of the rhs pattern for the same substitution. However, this inference rule has a problem. Consider one of the example typed rewrite rules we have seen, (SZ-1):

$$\{m : S\} \rhd m : S \rightsquigarrow \mathit{rep}\ m : \iota$$

In an untyped term rewriting system, this would be:

$$m \rightsquigarrow \mathit{rep}\ m$$

This rewrite rule does not seem very useful because it can be applied to every term. But in type-and-transform systems, (SZ-1) is useful in a very practical sense because it allows us to apply the *rep* conversion to as many subterms as possible, increasing the probability of having a useful transformation. Of course, we need to restrict rewriting to preserve typing (and later semantics) when this rule is applied. Upon closer inspection

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

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

$$\frac{\mathring{\Gamma};\mathring{\Gamma}_m \vdash e_1 \overset{\mathcal{R}}{\rightsquigarrow} p_1@e_1' \Rightarrow \theta_1 \qquad \mathring{\Gamma};\mathring{\Gamma}_m \vdash e_2 \overset{\mathcal{R}}{\rightsquigarrow} p_2@e_2' \Rightarrow \theta_2}{\mathring{\Gamma};\mathring{\Gamma}_m \vdash e_1\ e_2 \overset{\mathcal{R}}{\rightsquigarrow} p_1\ p_2@e_1'\ e_2' \Rightarrow \theta_2 \circ \theta_1}\ \text{(M-APP)}$$

Figure 9: Typed pattern matching

of our simple formulation, we see that the substitution here is, in fact, only partially typed. We are not checking that the type of each metavariable is an instance of the type of the subterm it matches. This leaves us with ill-typed terms for some rewrite rule applications.

Rather than relinquish the useful typed rewrite rules, we define a typed pattern matching that gives us well-typed rewriting during transformation:

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

The interpretation is that, given an object variable environment $\mathring{\Gamma}$ and a metavariable environment[13] $\mathring{\Gamma}_m$, a pattern p matches a target e' and produces a substitution θ such that $\theta p \equiv e'$ (see Lemma 1). Of the inference rules shown in Figure 9, (M-VAR) and (M-APP) are straightforward structural rules. In (M-MVAR), we see why the source e is needed in the judgment. When a metavariable pattern is found, the corresponding source and target terms must be components of a transformation whose type functor $\mathring{\tau}$ is an instance of the metavariable type functor $\mathring{\Gamma}_m(m)$. This ensures that the redex is well-typed.

Returning to the inference rule (T-REW), it says that if we have a well-typed redex θp_l from typed pattern matching along with a transformation with that redex as the target term, then we have a transformation with the contraction θp_r as the target. Also, the transformation above the line has the type functor $\mathring{\tau}_l$, and the transformation below the line has $\mathring{\tau}_r$.

For another illustration of (T-REW), see Figure 10, which is the diagram of Figure 7 extended with the terms of rewriting. Assume we have the typed rewrite rule $\mathring{\Gamma}_m \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r$. If we have a transformation $\mathring{\Gamma} \vdash e_s \overset{\mathcal{R}}{\rightsquigarrow} \theta p_l : \mathring{\tau}_l$ (where the rule (M-MVAR) necessarily holds), we can apply the rewrite rule. The result is the transformation $\mathring{\Gamma} \vdash e_s \overset{\mathcal{R}}{\rightsquigarrow} \theta p_r : \mathring{\tau}_r$. As before, the ($\mathring{\tau}$-REW) property, $\mathring{\tau}_l\langle\mathcal{A}\rangle \equiv \mathring{\tau}_r\langle\mathcal{A}\rangle$, holds if we have a well-typed rewrite rule.

To conclude our introduction to transformation typing, we describe the transformation derivation of example (3) from Table 1:

[13] This m is an indicator of the environment kind. It is not a metavariable.

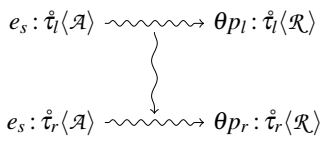


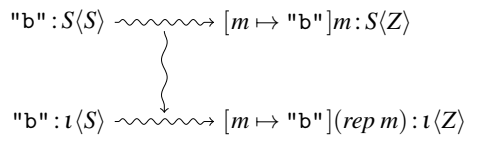
Figure 10: Diagram of rewriting with $\mathring{\Gamma}_m \triangleright p_l : \mathring{\tau}_l \rightsquigarrow p_r : \mathring{\tau}_r$

$$\mathring{\Gamma}_1 \vdash x +\!\!+ \texttt{"b"} \overset{\mathcal{R}}{\rightsquigarrow} x \diamond \mathit{rep}\ \texttt{"b"} : \iota$$

Here, $\mathring{\Gamma}_1 = \{x : \iota, \texttt{"b"} : S, \mathit{abs} : Z \to S, \mathit{rep} : S \to Z, Z : (S \to S) \to Z, (+\!\!+) : S \to S \to S, (\diamond) : Z \to Z \to Z\}$ and $\mathcal{R} = \{\text{(SZ-1)}, \text{(SZ-2)}, \text{(SZ-3)}\}$. A partial derivation tree follows:

$$\dfrac{\dfrac{\mathring{\Gamma}_1 \vdash (+\!\!+) \overset{\mathcal{R}}{\rightsquigarrow} (\diamond) : \iota \to \iota \to \iota \qquad \dfrac{\iota \prec \mathring{\Gamma}_1(x)}{\mathring{\Gamma}_1 \vdash x \overset{\mathcal{R}}{\rightsquigarrow} x : \iota}\ \text{(T-VAR)}}{\mathring{\Gamma}_1 \vdash (+\!\!+)\, x \overset{\mathcal{R}}{\rightsquigarrow} (\diamond)\, x : \iota \to \iota}\ \text{(T-APP)} \quad \vdots \quad \dfrac{\begin{array}{c}(\{m : S\} \triangleright m : S \rightsquigarrow \mathit{rep}\, m : \iota) \in \mathcal{R} \\ \mathring{\Gamma}_1 \vdash \texttt{"b"} \overset{\mathcal{R}}{\rightsquigarrow} \texttt{"b"} : S \\ \dfrac{\mathring{\Gamma}_1; \{m : S\} \vdash \texttt{"b"} \overset{\mathcal{R}}{\rightsquigarrow} m@\texttt{"b"} \Rightarrow [m \mapsto \texttt{"b"}]}{\mathring{\Gamma}_1 \vdash \texttt{"b"} \overset{\mathcal{R}}{\rightsquigarrow} [m \mapsto \texttt{"b"}](\mathit{rep}\, m) : \iota}\ \text{(T-REW)}\end{array}}{\mathring{\Gamma}_1 \vdash x +\!\!+ \texttt{"b"} \overset{\mathcal{R}}{\rightsquigarrow} x \diamond \mathit{rep}\ \texttt{"b"} : \iota}\ \text{(T-APP)}$$

The derivation tree includes several inference rules from Figure 8 – eliding one (T-VAR) and one (T-REW) instance – but our focus is the (T-REW) instance using the rewrite rule (SZ-1). First, the reader may wish to write the derivation of typed pattern matching to confirm that it holds according to the rules in Figure 9. Second, note that $\mathring{\tau}_l = S$ and $\mathring{\tau}_r = \iota$ and that these type functors match the transformation type functors in the premise and conclusion. We can instantiate the diagram in Figure 10 to clearly depict the concrete relationships in this instance:



In Section 5.3, we revisit this example for our discussion of transformation semantics.

We have presented the 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 semantic 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, 24] is a mixed-variant binary type constructor F with the F-indexed *dimap* function:[14]

$$\mathcal{D}_F : \forall a\ a'\ b\ b'.(b' \to a') \to (a \to b) \to F\ a'\ a \to F\ b'\ b$$

The first type parameter of F is contravariant, and the second is covariant. The function $\mathcal{D}_F$ must obey the following laws of identity and distribution over composition:

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

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

For a unary type constructor F, the equivalent of a binary constructor $F\ a\ a$ in which the same type parameter appears in both co- and contravariant positions, we write $\mathcal{D}_F$ as:

$$\mathcal{D}_F : \forall a\ b.(b \to a) \to (a \to b) \to F\ a \to F\ b$$

A type functor $\mathring{\tau}$ is a unary difunctor F. We write the parameterized constructor as $\mathring{\tau}\langle a \rangle = F\ a$ and define the dimap as:

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

Note the f and g argument reversal due to contravariance in the function type case.

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

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

Lifting the 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:

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

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 dimaps:

[14] We use $\mathcal{D}$, not dimap, for the function name since it appears a substantial number of times in this article.

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

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 (RR), 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:

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

To each pattern, we apply first the environment difunctor $\mathcal{D}_{\mathring{\Gamma}}$ for the rewrite rule environment $\mathring{\Gamma}$, which applies a type scheme difunctor $\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}$ to each metavariable with the scheme $\mathring{\varsigma}$. Then, we apply the pattern type functor $\mathcal{D}_{\mathring{\tau}}\ rep\ abs : \mathring{\tau}\langle\mathcal{R}\rangle \to \mathring{\tau}\langle\mathcal{A}\rangle$ for the pattern's type functor $\mathring{\tau}$.

The property (D-REW) must be proven for each typed rewrite rule in a transformation rule set. These are the respective properties of the rules (SZ-1), (SZ-2), and (SZ-3) in Section 4.2:

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

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

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

For the proof of an equation, we use equational reasoning with the definitions in Section 5.1 and Figure 2. Consider the following, simplified proof of (SZ-D-3):

$$\begin{aligned}&\mathcal{D}_{\iota\to\iota\to\iota}\ rep\ abs\ \mathcal{D}_{\varepsilon}(\diamond) \\ &\quad\equiv \mathcal{D}_{\iota\to\iota}\ rep\ abs \circ (\diamond) \circ \mathcal{D}_{\iota}\ abs\ rep && \mathcal{D}_{\iota\to\iota\to\iota},\ \mathcal{D}_{\varepsilon}\text{ definitions} \\ &\quad\equiv (\lambda y.\mathcal{D}_{\iota}\ rep\ abs \circ y \circ \mathcal{D}_{\iota}\ abs\ rep) \circ (\diamond) \circ \mathcal{D}_{\iota}\ abs\ rep && \mathcal{D}_{\iota\to\iota}\text{ definition} \\ &\quad\equiv (\lambda y.abs \circ y \circ rep) \circ (\diamond) \circ rep && \mathcal{D}_{\iota}\text{ definition} \\ &\quad\equiv \lambda x.\lambda y.abs\ (rep\ x \diamond rep\ y) && \text{simplification} \\ &\quad\equiv \lambda x.\lambda y.abs\ (Z\ (x+\!\!+) \diamond Z\ (y+\!\!+)) && rep\text{ definition} \\ &\quad\equiv \lambda x.\lambda y.abs\ (Z\ ((x+\!\!+) \circ (y+\!\!+))) && \diamond\text{ definition} \\ &\quad\equiv \lambda x.\lambda y.((x+\!\!+) \circ (y+\!\!+))\ ""\ "" && abs\text{ definition} \\ &\quad\equiv \lambda x.\lambda y.x +\!\!+ y +\!\!+ ""\ "" && \text{simplification} \\ &\quad\equiv (+\!\!+) && +\!\!+\text{ unit, simplification} \\ &\quad\equiv \mathcal{D}_{S\to S\to S}\ rep\ abs\ \mathcal{D}_{\varepsilon}(+\!\!+) && \mathcal{D}_{S\to S\to S},\ \mathcal{D}_{\varepsilon}\text{ definitions}\end{aligned}$$

The proof for (SZ-D-2) uses a non-empty type functor environment:

$$
\begin{aligned}
&\mathcal{D}_\iota\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}} m \\
&\quad \equiv \mathit{abs}\ \mathcal{D}_{\{m:\iota\}} m && \mathcal{D}_\iota \text{ definition} \\
&\quad \equiv \mathcal{D}_{\{m:\iota\}} (\mathit{abs}\ m) && \text{substitution distributes over } \mathit{abs} \text{ application} \\
&\quad \equiv \mathcal{D}_S\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}} (\mathit{abs}\ m) && \mathcal{D}_S \text{ definition}
\end{aligned}
$$

The proof for (SZ-D-1) is similar.

It is worth noting that, for the equations (SZ-D-1), (SZ-D-2), and (SZ-D-3), we could use an alternative property that is simpler than (D-REW):

$$\mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep}\ \mathit{abs}\ p_l \equiv \mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep}\ \mathit{abs}\ p_r$$

In other words, none of the proofs actually require $\mathcal{D}_{\mathring{\Gamma}}$. However, this property does not support more interesting rewrite rules in which both the lhs and rhs patterns mix metavariables with object variables, such as:

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

Defining the (D-REW) property for this rule and proving it is an interesting exercise. The reader may wish to attempt it and then refer to Appendix A for the solution.

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:

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

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 $\mathcal{D}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs} : \mathring{\tau}\langle \mathcal{R} \rangle \to \mathring{\tau}\langle \mathcal{A} \rangle$ and $\mathcal{D}_{\mathring{\Gamma}}$.

For an example of (D-TRANS) in use, we present the proof of example (3). In Section 4.3, we gave the typing judgment and derivation tree and defined $\mathring{\Gamma}_1$. The equation is:

$$x +\!\!+ \texttt{"b"} \equiv \mathcal{D}_\iota\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}_1} (x \diamond \mathit{rep}\ \texttt{"b"})$$

The proof follows:

$$
\begin{aligned}
&\mathcal{D}_\iota\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\mathring{\Gamma}_1} (x \diamond \mathit{rep}\ \texttt{"b"}) \\
&\quad \equiv \mathcal{D}_\iota\ \mathit{rep}\ \mathit{abs}\ (\mathcal{D}_\iota\ \mathit{abs}\ \mathit{rep}\ x \diamond \mathit{rep}\ \texttt{"b"}) && \mathcal{D}_{\mathring{\Gamma}_1} \text{ definition} \\
&\quad \equiv \mathit{abs}\ (\mathit{rep}\ x \diamond \mathit{rep}\ \texttt{"b"}) && \mathcal{D}_\iota \text{ definition} \\
&\quad \equiv (\lambda y.\lambda z.\mathit{abs}\ (\mathit{rep}\ y \diamond \mathit{rep}\ z))\ x\ \texttt{"b"} && \text{(RED-ETA)} \\
&\quad \equiv \mathcal{D}_{\iota\to\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_\varepsilon(\diamond)\ x\ \texttt{"b"} && \mathcal{D}_{\iota\to\iota\to\iota},\ \mathcal{D}_\varepsilon \text{ definitions} \\
&\quad \equiv \mathcal{D}_{S\to S\to S}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_\varepsilon(+\!\!+)\ x\ \texttt{"b"} && \text{(SZ-D-3)} \\
&\quad \equiv x +\!\!+ \texttt{"b"} && \mathcal{D}_{S\to S\to S},\ \mathcal{D}_\varepsilon \text{ definitions}
\end{aligned}
$$

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

A rule set $\mathcal{R}$ is valid if every rule $\rho \in \mathcal{R}$ is valid for Γ and $\mathcal{A}/\mathcal{R}$. •

We can now formally define a transformation:

Definition 2 (Transformation). *Given an $\mathcal{A}/\mathcal{R}$ isomorphism, a transformation is a tuple*[15] *$(\mathring{\Gamma}, \mathcal{R}, e, e', \mathring{\tau})$, where $\mathcal{R}$ is valid for $\mathring{\Gamma}\langle\mathcal{R}\rangle$ and $\mathcal{A}/\mathcal{R}$, that satisfies $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$ (Section 4.3).* •

The first basic property of a transformation is that the source and target terms are well-typed according to the source and target types and environments given by the inversion properties ($\mathring{\tau}$-INV), ($\mathring{\varsigma}$-INV), and ($\mathring{\Gamma}$-INV) (Section 4.1):

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. ■

The second basic property is the semantic relation between the source and target terms:

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).* □

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

Proof By rule induction on the derivations. We discuss some of the cases here. The remaining cases can be found in Appendix B.

The simplest case is the one for variables. The rule

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

must satisfy $x \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} x$.

Case (T-VAR)

$$\begin{aligned}
&\mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} x \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}', x:\mathring{\tau}} x && \mathring{\tau} \prec \mathring{\Gamma}(x),\ \mathring{\Gamma} = \mathring{\Gamma}', x:\mathring{\tau} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}'} \circ [x \mapsto \mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x])x && \mathcal{D}_{\mathring{\Gamma}', x:\mathring{\tau}} \text{ definition} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x) && \text{substitution} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ (abs \circ rep)\ (abs \circ rep)\ x && \text{(D-COMP)} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ id\ id\ x && \text{(ABS-REP)} \\
&\quad \equiv x && \text{(D-ID)}
\end{aligned}$$

In the proof, we use the difunctor laws and unfold the type functor definition to clarify what happens when applying the substitution. Most importantly, we need the (ABS-REP) direction of the isomorphism to show that the type functor is the identity operation.

The case for **fix** is more interesting. The rule

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

must satisfy $\mathbf{fix}\ e \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{fix}\ e')$.

Case (T-FIX)

$$\begin{aligned}
&\mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(\mathbf{fix}\ e') \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ \mathcal{D}_{\mathring{\Gamma}} e') && \text{substitution distributes over } \mathbf{fix} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}} e' \circ id)) && \circ \text{ unit} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}} e' \circ \mathcal{D}_{\mathring{\tau}}\ id\ id)) && \text{(D-ID)} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}} e' \circ \mathcal{D}_{\mathring{\tau}}\ (rep \circ abs)\ (rep \circ abs))) && \text{(REP-ABS)} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ (\mathcal{D}_{\mathring{\Gamma}} e' \circ \mathcal{D}_{\mathring{\tau}}\ abs\ rep \circ \mathcal{D}_{\mathring{\tau}}\ rep\ abs)) && \text{(D-COMP)} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ (\mathbf{fix}\ ((\mathcal{D}_{\mathring{\Gamma}} e' \circ \mathcal{D}_{\mathring{\tau}}\ abs\ rep) \circ \mathcal{D}_{\mathring{\tau}}\ rep\ abs)) && \circ \text{ associativity} \\
&\quad \equiv \mathbf{fix}\ (\mathcal{D}_{\mathring{\tau}}\ rep\ abs \circ \mathcal{D}_{\mathring{\Gamma}} e' \circ \mathcal{D}_{\mathring{\tau}}\ abs\ rep) && \text{(RED-ROLLING)} \\
&\quad \equiv \mathbf{fix}\ (\mathcal{D}_{\mathring{\tau}\to\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e') && \mathcal{D}_{\mathring{\tau}\to\mathring{\tau}} \text{ definition} \\
&\quad \equiv \mathbf{fix}\ e && \text{IH}
\end{aligned}$$

Here, we use the difunctor laws again plus some standard properties of substitution and composition. Since **fix** is syntactically recursive, the last rule is the inductive hypothesis (IH), which says that the premise $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau} \to \mathring{\tau}$ satisfies $e \equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\tau}}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}} e'$.

One noteworthy point is the use of (REP-ABS), which is due to the function argument to **fix**. Having a function type means that not only must we convert the result type (as we did with (ABS-REP) in the (T-VAR) case), we must also convert the argument type, which leads to an inversion of the conversion and the need for both directions of the $\mathcal{A}/\mathcal{R}$ isomorphism. For the same reason, we find (REP-ABS) in the proof cases for (T-APP) and (T-LET). The latter has an implicit function that can be seen in its semantic equivalence to a lambda application: $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \equiv (\lambda x.e_2)\ e_1$.

To rewrite the fixed point, we use the *rolling rule*, (RED-ROLLING), which was first described by Backhouse et al. [2] for a fixed point calculus with the least fixed point.[16] The formalization of the worker/wrapper transformation [13] uses the rolling rule and provides a nice description of it.

The last case we discuss is the (T-REW) rule

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

which should satisfy $e \equiv \mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}}(\theta p_r)$.

Case (T-REW)

$$\begin{aligned}
&\mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}}(\theta p_r) \\
&\quad \equiv \theta(\mathcal{D}_{\mathring{\tau}_r}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}} p_r) && \text{See } \textit{Note on } \theta. \\
&\quad \equiv \theta(\mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}} p_l) && \text{(D-REW)} \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}}(\theta p_l) && \text{See } \textit{Note on } \theta. \\
&\quad \equiv \mathcal{D}_{\mathring{\tau}_l}\ \mathit{rep\ abs}\ \mathcal{D}_{\mathring{\Gamma}} e' && \text{Lemma 1} \\
&\quad \equiv e && \text{IH}
\end{aligned}$$

In this proof case, we use (D-REW) to rewrite the rhs to the lhs, and Lemma 1, provided below, shows that $e' \equiv \theta p_l$.

Note on θ. In each of the referenced steps, we commute θ with $\mathcal{D}_{\mathring{\Gamma}}$ and distribute it over application. Recall that the domain of θ consists only of metavariables (Figure 9). The difunctors only apply to object variables or terms, so the domain of θ does not conflict, and we can pass θ around without concern.

This concludes our discussion of the proof. The remaining cases can be found in Appendix B. ∎

Lemma 1. *If* $\mathring{\Gamma}; \mathring{\Gamma}_m \vdash e \overset{\mathcal{R}}{\rightsquigarrow} p @ e' \Rightarrow \theta$, *then* $e' \equiv \theta p$. □

[16] According to Backhouse et al. [2], this form of the rolling rule was first derived by Lambert Meertens in "unpublished discussion notes."

Proof By straightforward induction on the structure of p and e' and the standard definitions of substitution, *id*, and $\circ$. ∎

The last 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. Before giving the definition, however, we need to explain this "same-type" property.

Recall that ι indicates where the type changes in a type functor (Figure 5). We then define a function that determines if $\mathring{\tau}$ "has" any ιs:

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

That is, $\bar{\iota}(\mathring{\tau})$ holds iff no ι appears anywhere in $\mathring{\tau}$. Lemma 2 and Lemma 3 give us other properties of $\mathring{\tau}$:

Lemma 2. *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 3. *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 prove the lemmas for) type functor schemes and environments.

We can now define complete transformations:

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

The properties follow:

Theorem 3 (Typing of complete transformation terms). *If a transformation $\mathring{\Gamma} \vdash e \overset{\mathcal{R}}{\rightsquigarrow} e' : \mathring{\tau}$ 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 2. ∎

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

Proof Follows from Theorem 2 and Lemma 3. ∎

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

7. Discussion

In this section, we discuss the algorithmic and practical considerations of type-and-transform systems.

7.1. Algorithm

Type-and-transform systems can be implemented in algorithmic form, and we have developed a Haskell implementation,[17] which we have used to experiment with all of the examples in this paper. The code is available for download [20].

The transformation algorithm is a type inference algorithm that implements transformation (Figure 8) in the same way that algorithm $\mathcal{W}$ [25] implements typing (Figure 4). The primary difference is the addition of rewriting, including typed pattern matching (Figure 9).

The rewriting inference rule (T-REW), unlike the other rules of Figure 8, is not syntax-directed, which means a derivation may not terminate. To guarantee termination in the algorithm, we restrict it to rewriting a subterm only a finite number of times. The algorithm is still inherently nondeterministic because any one subterm can be matched by multiple rewrite rules. As we discuss in the next section, nondeterminism allows the algorithm to try different transformations "locally" in the effort of producing a better transformation "globally."

We plan to present the details of the transformation algorithm in another article. In particular, we will show that the algorithm is sound – it implements the transformation of Definition 2 – which follows from the correspondence to algorithm $\mathcal{W}$. It is trivial to show that a basic algorithm is not complete due to the flexibility of rewriting, but we believe we can prove completeness for a restricted formulation of rewriting.

7.2. Practical Aspects

Experiments with our implementation have led us to the choices of supporting very general typed rewrite rules (per the discussion in Section 4.3) and of simplifying rewriting by applying each rewrite rule at most once to the same subterm. The latter implies that we do not produce all possible results. In practice, however, we have found that to be less of a problem than longer transformation times required for performing more rewriting.

The nondeterminism due to rewriting leads to the problem of selecting one of many possible transformations. It is tempting to think that there is an optimal choice, but we have found no useful strict ordering on transformations. Instead, we have seen multiple transformation targets that are equally "good." Consider the three different targets for the same source in the examples (4), (5), and (6) from Table 1. It is not immediately 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, do seem clearly better than others. Consider the examples in Table 2: two sources, each with two of the many possible targets.

[17] The implementation was simplified by using the UNBOUND [40] library for substitution and fresh name generation, and the performance was improved by using the *LogicT* monad [19] for efficient nondeterminism.

Table 2: Examples of transformations with preference

Source	Target	Preferred	
$\texttt{"a"} +\!\!+ \texttt{"b"} +\!\!+ \texttt{"c"}$			(7)
	$abs\ (rep\ \texttt{"a"} \diamond rep\ \texttt{"b"} \diamond rep\ \texttt{"c"})$	✓	
	$\texttt{"a"} +\!\!+ abs\ (rep\ \texttt{"b"} \diamond rep\ \texttt{"c"})$		
$(\lambda x.x +\!\!+ x)\ \texttt{"a"}$			(8)
	$abs\ ((\lambda x.x \diamond x)\ (rep\ \texttt{"a"}))$	✓	
	$abs\ ((\lambda x.rep\ x \diamond rep\ x)\ \texttt{"a"})$		

We have marked which targets we prefer for this particular type-and-transform system (i.e. this rewrite rule set).

The motivations for the preferred transformations are based on what we see as the desired outcome for each of the rewrite rules in the rule set. For example, we prefer to replace as many $+\!\!+$ terms with $\diamond$ terms as possible (i.e. apply (SZ-3) as much as possible) because $\diamond$ is, in general, more efficient, as discussed in Section 1.1. This manifests as a preference for the first target in (7).

If there are an equal number of $+\!\!+ \rightsquigarrow \diamond$ rewrites, then we prefer having as few *rep* terms introduced as possible. Conversion is not free, so we should only convert when needed. This results in the preference indicated for (8).

Recall the viral infection analogy of Section 4.2. Ironically, perhaps, we prioritize the following:

1. maximize the transmission (spread the virus as far as possible),
2. minimize the introduction (infect early), and
3. minimize the elimination (stop late).

In our implementation, we designate each rewrite rule with a weight that indicates the priorities above and use a simple heuristic to score transformations, choosing one with the "best" score. If there are multiple results with the same top score, we pick an arbitrary one.

All of the type-and-transform systems with which we have experimented exhibit the same characteristics of preference that we have described. Type-and-transform systems appear ideally suited to these sorts of changes that are whole-program and viral.

8. Parameterized Type Constructors

Up to this point, we have used only simple (nullary) types to simplify explanation. In this section, we extend our work to support parameterized type constructors and demonstrate it with difference lists.

The adapted syntax of types and type functors follows:

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

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:

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

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

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

Here, $\mathcal{D}_C$ is the dimap for the base constructor C, and the function arguments $\mathring{d}_c$ and $\mathring{d}_d$ are the dimaps 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 $\mathcal{D}_{\varphi}\ f\ g \equiv map_{\varphi}\ g$, 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].

8.1. 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). The function definitions are the same; only the type and the type signatures differ:

$$
\textbf{newtype}\ H\ a = H\ ([a] \to [a]) \qquad \begin{array}{ll} rep :: [a] \to H\ a & (\diamond) :: H\ a \to H\ a \to H\ a \\ abs :: H\ a \to [a] & \epsilon\ \ :: H\ a \end{array}
$$

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 rules with the conversion functions

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 5.2.

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

$$\{m : [a]\} \triangleright m : [a] \quad \rightsquigarrow rep\ m : \iota\ a \tag{9}$$

$$\{m : \iota\ a\} \triangleright m : \iota\ a \quad \rightsquigarrow abs\ m : [a] \tag{10}$$

$$\varepsilon \triangleright +\!\!+ : [a] \to [a] \to [a] \rightsquigarrow \diamond \quad : \iota\ a \to \iota\ a \to \iota\ a \tag{11}$$

$$\varepsilon \triangleright [\,] : [a] \quad \rightsquigarrow \epsilon \quad : \iota\ a \tag{12}$$

In addition to rewrite rules adapted from the difference string rules of Sections 4.2, we add the rewrite rule (12) to cover the case of rewriting empty lists. This rule is not strictly necessary; it is merely an optimization of (9) when the term is an empty list. But (12) gives us nicer transformations and demonstrates that we can extend a rewrite rule set with overlapping rules as well as perform some basic compiler optimizations.

There are some interesting transformations that we can demonstrate. The first is the *reverse* example (shown as source above target):

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

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

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

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

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

These examples barely touch the surface of how much of a program can be changed by a transformation. For example, by changing the function *list* to the difference list eliminator *dlist*, we can also change the types of the inputs to these functions. In a related paper, van Eekelen et al. [36] explore the options for transforming data constructors and patterns, which allow for even more parts of a program to be transformed.

9. Other Applications

In this section, we describe two more applications of type-and-transform systems in the style of Section 1.1. With each concrete example, we give the rewrite rules for the transformation as in Section 8.1.

9.1. 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 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 *Seq Int* (a finger tree [16] 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 [34, 21]. 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*:

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

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, we get natural recursion on *FExp* using a 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):

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

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

$$\mathit{three} = \mathit{In}\ (\mathit{Add}_F\ (\mathit{In}\ (\mathit{Val}_F\ 1))\ (\mathit{In}\ (\mathit{Val}_F\ 2)))$$

we use smart constructors:

$$
\begin{aligned}
&\mathit{val} :: \mathit{Int} \to \mathit{FExp} \\
&\mathit{val}\ i = \mathit{In}\ (\mathit{Val}_F\ i) \\
&\mathit{add} :: \mathit{FExp} \to \mathit{FExp} \to \mathit{FExp} \\
&\mathit{add}\ e_1\ e_2 = \mathit{In}\ (\mathit{Add}_F\ e_1\ e_2)
\end{aligned}
$$

As an additional convenience, we define a specialized fold for *FExp*:[18]

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

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

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Exp} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ i) \quad\ = i \\
&\mathit{eval}\ (\mathit{Add}\ e_1\ e_2) = \mathit{eval}\ e_1 + \mathit{eval}\ e_2 \\
&\mathit{eval}_F :: \mathit{FExp} \to \mathit{Int} \\
&\mathit{eval}_F = \mathit{foldFExp}\ \mathit{id}\ (+)
\end{aligned}
$$

Transformation. The *Exp*-to-*FExp* transformation can be split into two logical classes of rewrite rules:

1. Rewriting built-in constructors to their smart-constructor analogs:

$$\varepsilon \triangleright \mathit{Val} : \mathit{Int} \to \mathit{Exp} \quad \rightsquigarrow \mathit{val} : \mathit{Int} \to \iota \tag{13}$$

$$\varepsilon \triangleright \mathit{Add} : \mathit{Exp} \to \mathit{Exp} \to \mathit{Exp} \rightsquigarrow \mathit{add} : \iota \to \iota \to \iota \tag{14}$$

   Note that these are the transmission rules of Section 7.2. Rule (14) is more preferred because it can infect via both of its arguments.

[18] We can, of course, define *foldExp* just as easily, but there are other approaches, e.g. pattern functors [41], that can provide convenient folds for free. For the sake of simplicity, we present only the base-functor approach.

2. Inserting conversions where necessary:

$$\{m : Exp\} \triangleright m : Exp \rightsquigarrow from\ m : \iota \tag{15}$$

$$\{m : \iota\} \triangleright m : \iota \quad \rightsquigarrow to\ m \quad : Exp \tag{16}$$

If the entire programming interface consisted only of *Val* and *Add*, then it may seem like the conversion class of rules would be unnecessary. In other words, the problem of transformation becomes a simple matter of term rewriting, not typed rewriting. However, once a function such as $isVal :: Exp \to Bool$ is included in the interface without including a rule rewriting it, then the simple term rewriting approach breaks down: it will change the type of the argument to *isVal* without changing the type of *isVal*. Conversion is necessary for these cases, which we believe to be common in real-world situations.

One may instead take the contrary view and suggest that the smart-constructor class of rules is unnecessary and that the conversions are enough. Recall that the goal of this scenario is to perform a sort of refactoring to change as much of the code as possible, allowing Pat the programmer to begin using the new generic interface. If the code was riddled with applications of *to* and *from*, this would arguably not be considered an improvement to the program. Code readability could be significantly impaired, and Pat would need to do more work to find the right places to use generic functions. Lastly, generic programming approaches such as incrementalization [21] benefit more from the continuous use of a generic representation rather than repeated conversion between representations.

9.2. 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 *A*s to *B*s 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, time stamps (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.

`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 11: Interfaces of two libraries for complex numbers

Example: Complex Numbers. As a simple example, we consider integrating two libraries, presented in Figure 11, for representing complex numbers [11, 38]. 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:

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

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 prioritize the transmission rules (19) and (20) in our implementation.

10. 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 [37] 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 [23] is a Haskell refactoring tool that supports a number of automatic refactorings; however, it does not provide type-changing rewriting for whole-program transformations. Other tool-supported equational reasoning approaches include PATH [35] and HERMIT [7], both of which do not appear to directly implement the sort of automatic, whole-program, type-changing rewriting that type-and-transform systems employs. Nonetheless, it may be possible to build a type-and-transform system with one of these 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 Damas-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. [33] 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.

One closely related line of research is the worker/wrapper transformation [13, 30], a proof technique for systematically transforming a recursive computation with the aim of improving its performance. The two approaches share similar tools, such as changing the types of parts of a program, and some of the transformations are similar – Gill and Hutton [13] also use the *reverse* example. However, their work differs from

ours in several respects.

First, the focus of a worker/wrapper transformation is a single recursive function or a group of mutually recursive functions. The focus of a type-and-transform system, on the other hand, is restricted only by the scope given to it, which may be a function, a module, or an entire program. A worker/wrapper transformation is primarily intended for improving performance, while a type-and-transform system may be used for other applications (see Section 9).

Second, a worker/wrapper transformation requires manual steps to change a function into a "worker" component, which may have a new type, and a "wrapper" component that has the same type as the original function and massages the input and output for the worker. Sculthorpe et al. [31] show that a worker/wrapper transformation can be mechanized to run in the GHC compiler; however, the transformation steps must still be defined and ordered. A type-and-transform system is a fully automatic system that requires defining a set of typed rewrite rules. We believe these rules are reasonably simple, but there may be a large number of them, depending on how many different symbols one wants to rewrite. The type-and-transform system rewrite rules are naturally restricted to the general constraints of the system and the lack of context, since they should be applicable anywhere, while the rewrite operations of a worker/wrapper transformation can be sequenced and use context to select the appropriate time and place of application.

Whether one wants to use a worker/wrapper transformation or a type-and-transform system depends on the situation. Our impressions are that neither approach subsumes the other and that each demonstrates an interesting and useful approach to improving programs.

11. 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 [28], representation independence [26], and parametricity [39]. 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.

This work has been partially funded by the Netherlands Organization for Scientific Research (NWO) through the project "Real-Life Datatype-Generic Programming" (612.063.613).

[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 *Proceedings of Category Theory and Computer Science (CTCS)*, pages 159–179, 1995.

[3] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell Strings. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 50–64, 2007.

[4] A. Cunha and J. Visser. Strongly Typed Rewriting For Coupled Software Transformation. In *Proceedings of the International Workshop on Rule Based Programming (RULE)*, pages 17–34. Elsevier, 2006.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 207–212. ACM, 1982.

[6] M. Erwig and D. Ren. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 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 *Proceedings of the Workshop on Haskell*, pages 1–12, 2012.

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

[9] P. Freyd. Recursive Types Reduced to Inductive Types. In *Proceedings of Logic in Computer Science (LICS)*, pages 498–507, 1990.

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

[11] J. Gibbons. Unfolding Abstract Datatypes. In *Proceedings of Mathematics of Program Construction (MPC)*, pages 110–133, 2008.

[12] D. K. Gifford and J. M. Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the Conference on LISP and Functional Programming (LFP)*, pages 28–38. ACM, 1986.

[13] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

[14] T. Harper. Stream Fusion on Haskell Unicode Strings. In *Proceedings of the International Conference on Implementation and Application of Functional Languages (IFL)*, pages 125–140, 2011.

[15] R. Hinze. Polytypic Values Possess Polykinded Types. In *Proceedings of Mathematics of Program Construction (MPC)*, pages 2–27, 2000.

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

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

[18] R. Kießling and Z. Luo. Coercions in Hindley-Milner Systems. In *Proceedings of the International Workshop on Types for Proofs and Programs (TYPES)*, pages 259–275, 2003.

[19] O. Kiselyov, C. C. Shan, D. P. Friedman, and A. Sabry. Backtracking, Interleaving, and Terminating Monad Transformers. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 192–203. ACM, 2005.

[20] S. Leather. `tts-0.3.3`, 2014. URL `http://www.staff.science.uu.nl/~leath101/code/tts-0.3.3.tar.gz`.

[21] S. Leather, A. Löh, and J. Jeuring. Pull-Ups, Push-Downs, and Passing It Around: Exercises in Functional Incrementalization. In *Proceedings of the International Conference on Implementation and Application of Functional Languages (IFL)*, pages 159–178, 2011.

[22] S. Leather, J. Jeuring, A. Löh, and B. Schuur. Type-Changing Rewriting and Semantics-Preserving Transformation. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 109–120. ACM, 2014.

[23] H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In *Proceedings of the Workshop on Haskell*, pages 27–38, 2003.

[24] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Proceedings of the International Conference on Functional Programming and Computer Architecture (FPCA)*, pages 324–333, 1995.

[25] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[26] J. C. Mitchell. Representation independence and data abstraction. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 263–276. ACM, 1986.

[27] A. M. Pitts. Typed Operational Reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. MIT Press, 2005.

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

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

[30] N. Sculthorpe and G. Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014.

[31] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Proceedings of the International Conference on Implementation and Application of Functional Languages (IFL)*, 2012.

[32] M. R. Sleep and S. Holmström. A short note concerning lazy reduction rules for append. *Software: Practice and Experience*, 12(11):1082–1084, 1982.

[33] N. Swamy, M. Hicks, and G. M. Bierman. A Theory of Typed Coercions and Its Applications. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 329–340. ACM, 2009.

[34] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4): 423–436, July 2008.

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

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

[37] E. Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[38] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 307–313. ACM, 1987.

[39] P. Wadler. Theorems for free! In *Proceedings of the International Conference on Functional Programming and Computer Architecture (FPCA)*, pages 347–359, 1989.

[40] S. Weirich, B. A. Yorgey, and T. Sheard. Binders Unbound. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 333–345. ACM, 2011.

[41] A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 233–244. ACM, 2009.

Appendix A. Proof of (D-REW) Example

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

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

To avoid confusing syntactic complications of infix operators, we use $\underline{\mathit{append}} = (+\!\!+)$ and $\underline{\mathit{compose}} = (\diamond)$ in strictly prefix positions:

$$\{\, m : \iota \,\} \triangleright \underline{\mathit{append}}\ (\mathit{abs}\ m) : S \to S \rightsquigarrow \underline{\mathit{compose}}\ m : \iota \to \iota$$

The corresponding (D-REW) property is:

$$\mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\underline{\mathit{append}}\ (\mathit{abs}\ m)) \equiv \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\underline{\mathit{compose}}\ m)$$

The proof of this property follows:

$$\begin{aligned}
&\mathcal{D}_{S\to S}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\underline{\mathit{append}}\ (\mathit{abs}\ m)) \\
&\quad\equiv \mathcal{D}_{\{m:\iota\}}(\underline{\mathit{append}}\ (\mathit{abs}\ m)) && \mathcal{D}_{S\to S}\text{ definition} \\
&\quad\equiv \underline{\mathit{append}}\ (\mathit{abs}\ (\mathit{rep}\ m)) && \mathcal{D}_{\{m:\iota\}}\text{ definition} \\
&\quad\equiv \underline{\mathit{append}}\ m && \text{(ABS-REP)} \\
&\quad\equiv \lambda y.\underline{\mathit{append}}\ m\ y && \text{(RED-ETA)} \\
&\quad\equiv \lambda y.\underline{\mathit{append}}\ m\ (\underline{\mathit{append}}\ y\ \texttt{""}) && +\!\!+\text{ unit} \\
&\quad\equiv \lambda y.(\underline{\mathit{append}}\ m \circ \underline{\mathit{append}}\ y)\ \texttt{""} && \circ\text{ definition, (RED-LAM)} \\
&\quad\equiv \lambda y.\mathit{abs}\ (Z\ (\underline{\mathit{append}}\ m \circ \underline{\mathit{append}}\ y)) && \mathit{abs}\text{ definition} \\
&\quad\equiv \lambda y.\mathit{abs}\ (\underline{\mathit{compose}}\ (Z\ (\underline{\mathit{append}}\ m))\ (Z\ (\underline{\mathit{append}}\ y))) && \diamond\text{ definition} \\
&\quad\equiv \lambda y.\mathit{abs}\ (\underline{\mathit{compose}}\ (\mathit{rep}\ m)\ (\mathit{rep}\ y)) && \mathit{rep}\text{ definition} \\
&\quad\equiv \lambda y.\mathit{abs}\ ((\underline{\mathit{compose}}\ (\mathit{rep}\ m) \circ \mathit{rep})\ y) && \circ\text{ definition, (RED-LAM)} \\
&\quad\equiv \mathit{abs} \circ \underline{\mathit{compose}}\ (\mathit{rep}\ m) \circ \mathit{rep} && \circ\text{ definition} \\
&\quad\equiv \mathcal{D}_\iota\ \mathit{rep}\ \mathit{abs} \circ \underline{\mathit{compose}}\ (\mathit{rep}\ m) \circ \mathcal{D}_\iota\ \mathit{abs}\ \mathit{rep} && \mathcal{D}_\iota\text{ definition} \\
&\quad\equiv \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ (\underline{\mathit{compose}}\ (\mathit{rep}\ m)) && \mathcal{D}_{\iota\to\iota}\text{ definition} \\
&\quad\equiv \mathcal{D}_{\iota\to\iota}\ \mathit{rep}\ \mathit{abs}\ \mathcal{D}_{\{m:\iota\}}(\underline{\mathit{compose}}\ m) && \mathcal{D}_{\{m:\iota\}}\text{ definition}
\end{aligned}$$

Appendix B. Proof of Transformation Semantics

This appendix gives the cases omitted from the proof of Theorem 2 in Section 6. For each proof case, we include the relevant rule for a convenient reference.

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

Case (T-App) $e_1\ e_2 \equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(e_1'\ e_2')$

$$\begin{aligned}
&\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(e_1'\ e_2') \\
&\quad\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} e_1'\ \mathcal{D}_{\mathring{\Gamma}} e_2') && \text{substitution distributes over application} \\
&\quad\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} e_1'\ (\mathcal{D}_{\mathring{\tau}}\ id\ id\ \mathcal{D}_{\mathring{\Gamma}} e_2')) && \text{(D-ID)} \\
&\quad\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} e_1'\ (\mathcal{D}_{\mathring{\tau}}\ (rep \circ abs)\ (rep \circ abs)\ \mathcal{D}_{\mathring{\Gamma}} e_2')) && \text{(REP-ABS)} \\
&\quad\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} e_1'\ (\mathcal{D}_{\mathring{\tau}}\ abs\ rep\ (\mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_2'))) && \text{(D-COMP)} \\
&\quad\equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'\ (\mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_2') && \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \text{definition} \\
&\quad\equiv e_1\ e_2 && \text{IH}
\end{aligned}$$

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

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

$$\begin{aligned}
&\mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(\lambda x.e') \\
&\quad\equiv \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ rep\ abs\ (\lambda x.\mathcal{D}_{\mathring{\Gamma}} e') && \mathcal{D}_{\mathring{\Gamma}}\ \text{distributes over}\ \lambda\ \text{since}\ x : \mathring{\tau} \notin \mathring{\Gamma} \\
&\quad\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs \circ (\lambda x.\mathcal{D}_{\mathring{\Gamma}} e') \circ \mathcal{D}_{\mathring{\tau}}\ abs\ rep && \mathcal{D}_{\mathring{\tau}\to\mathring{\upsilon}}\ \text{definition} \\
&\quad\equiv \lambda x.\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ((\lambda x.\mathcal{D}_{\mathring{\Gamma}} e')\ (\mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x)) && \circ\ \text{definition} \\
&\quad\equiv \lambda x.\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ [x \mapsto \mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x]\mathcal{D}_{\mathring{\Gamma}} e' && \text{(RED-LAM)} \\
&\quad\equiv \lambda x.\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x] \circ \mathcal{D}_{\mathring{\Gamma}})e' && \text{substitution composition} \\
&\quad\equiv \lambda x.\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\tau}}\ abs\ rep\ x])e' && \text{commute}\ \circ\ \text{since}\ x : \mathring{\tau} \notin \mathring{\Gamma} \\
&\quad\equiv \lambda x.\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}, x:\mathring{\tau}} e' && \mathcal{D}_{\mathring{\Gamma}, x:\mathring{\tau}}\ \text{definition} \\
&\quad\equiv \lambda x.e && \text{IH}
\end{aligned}$$

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

Case (T-LET) $\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(\textbf{let}\ x = e_1'\ \textbf{in}\ e_2')$

$\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}}(\textbf{let}\ x = e_1'\ \textbf{in}\ e_2')$

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\textbf{let}\ x = \mathcal{D}_{\mathring{\Gamma}} e_1'\ \textbf{in}\ \mathcal{D}_{\mathring{\Gamma}} e_2')$

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

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ [x \mapsto \mathcal{D}_{\mathring{\Gamma}} e_1']\mathcal{D}_{\mathring{\Gamma}} e_2'$ (RED-LET)

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\Gamma}} e_1'] \circ \mathcal{D}_{\mathring{\Gamma}}) e_2'$ substitution composition

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ id\ id\ \mathcal{D}_{\mathring{\Gamma}} e_1'] \circ \mathcal{D}_{\mathring{\Gamma}}) e_2'$ (D-ID), $\mathring{\varsigma} = \mathcal{G}_{\mathring{\Gamma}}(\mathring{\tau})$

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ (rep \circ abs)\ (rep \circ abs)\ \mathcal{D}_{\mathring{\Gamma}} e_1'] \circ \mathcal{D}_{\mathring{\Gamma}}) e_2'$ (REP-ABS)

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ (\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1')] \circ \mathcal{D}_{\mathring{\Gamma}}) e_2'$ (D-COMP)

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'] \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ x] \circ \mathcal{D}_{\mathring{\Gamma}}) e_2'$

substitution composition

$\equiv \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ ([x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'] \circ \mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ x]) e_2'$

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

$\equiv [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1']\ (\mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ x]) e_2')$

substitution distributes over application

$\equiv \textbf{let}\ x = \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'\ \textbf{in}\ \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ x]) e_2'$

(RED-LET)

$\equiv \textbf{let}\ x = \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'\ \textbf{in}\ \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ (\mathcal{D}_{\mathring{\Gamma}} \circ [x \mapsto \mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}\ abs\ rep\ x]) e_2'$

$\mathcal{D}_{\mathring{\varsigma},\mathring{\Gamma}}$ definition

$\equiv \textbf{let}\ x = \mathcal{D}_{\mathring{\tau}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}} e_1'\ \textbf{in}\ \mathcal{D}_{\mathring{\upsilon}}\ rep\ abs\ \mathcal{D}_{\mathring{\Gamma}, x:\mathring{\varsigma}} e_2'$ $\mathcal{D}_{\mathring{\Gamma}}$ definition

$\equiv \textbf{let}\ x = e_1\ \textbf{in}\ e_2$ IH