Type-and-Transform Systems

Sean Leather

Johan Jeuring

Andres Löh

Bram Schuur

Technical Report UU-CS-2012-004
March 2012

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

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

Keywords type-and-transform systems, constraint-based type inference, program transformation

Abstract

We present type-and-transform systems, an approach to type-safe, semantics-preserving, automatic program transformation. A type-and-transform system maps a source program using one type to a target program using another type. The core of the system, propagation, is derived directly from the object language type system. The transformation itself is defined with simple typed rewrite rules.

In this paper, we describe the theory of type-and-transform systems and give an implementation. We illustrate the concept with several realistic examples from the literature, and we establish the correctness properties of type-and-transform systems.

1. Introduction

Program improvement often involves changing a program in a regular and methodical way, for example when updating a program to use a new version of a library. This monotonous work is often very boring—and thus error-prone—and should be automated.

As programmers in functional languages such as Haskell, ML, F#, Scala, and Typed Racket, we rely on strong, static, and expressive type systems to give a program an automatically checked specification. Types are proof that programs do not have certain errors or "go wrong" [21].

Some automatic program improvement techniques give no guarantees that type correctness is preserved. For example, HLint, a tool for suggesting possible improvements to Haskell code, has included the following warning in its manual [22]:

> ***Disclaimer:*** *While these hints are meant to be correct, they aren't guaranteed to be. Please report any non-equivalent hints [...]*

HLint is a very useful and well-tested tool. It supports a significant number of hints and can be customized with more hints, though it does not transform a program directly.

In this paper, we present a technique for automatic program transformation that preserves both the type-correctness and the semantics of a program. We focus on transformations that map a source program using one type to a target program using another type, where there is a clear relationship between the source and target types. A common example of this is a change in a library's API between releases. This well-known problem has prompted various solutions. Recently, Google created a special tool, `Gofix` [4], for their programming language Go. `Gofix` automatically updates Go programs with the API changes for each release of Go, allowing the standard library and compiler developers more freedom to evolve user-visible features while reducing the upgrade difficulty faced by programmers.

Interesting potential applications of our transformation system often arise in the Haskell community. One recent example that garnered some discussion was described by Johan Tibell as [1]:

> *Friends don't let friends use String.*

It is often better to use the `text` [12] and `bytestring` [3] libraries, even though *String* is more common in the standard libraries. In order to make Haskell programs more efficient, Tibell and others promote migration away from *String* to these other libraries.

In general, suppose we have a program that uses a library with functions involving some type $\mathcal{A}$, and we wish to migrate our program to another library using some type $\mathcal{R}$ instead of $\mathcal{A}$. Perhaps the library evolved, meaning $\mathcal{A}$ and $\mathcal{R}$ are the same type, but the available functions are different. Or perhaps the new library has a more efficient implementation than the old one. Either way, we do not want to break our program during migration, whether that happens by introducing a type error or by changing the program's semantics.

We present *type-and-transform systems* as a solution to type-changing program migration. A type-and-transform system describes a type-safe, semantics-preserving, automatic program transformation. Type-and-transform systems infer program transformations through a combination of propagating type changes and typed rewriting to transform terms. Transformations do not change the semantics of a program, though they can alter the types and semantics of subterms within the program.

We see type-and-transform systems as tools for refactoring or code generation. In refactoring, a programmer transforms her own code and continues to work with the result. In a compiler, a type-and-transform system serves as an optimization phase, perhaps performing some of the inlining duties.

1.1 Contributions

The contributions of this paper are the following:

- We introduce type-and-transform systems for the simply typed lambda calculus and the polymorphic lambda calculus.
- We describe several practical examples from the literature to motivate the need for a system that safely and automatically migrates programs from using one type to using another. We define type-and-transform systems for these examples.
- We define the correctness properties of type-and-transform systems.
- We give an implementation of a type-and-transform system, including how we deal with nondeterminism and potential nontermination.

1.2 Overview

This paper is organized as follows. In Section 2, we describe two examples of transformations that motivate type-and-transform systems. We then introduce type-and-transform systems for the simply typed lambda calculus in Section 3. In Section 4, we present an implementation of a type-and-transform system, including background on constraint-based type inference and descriptions of typed rewriting and the propagation algorithm. In Section 5, we explain the modifications needed for type-and-transform systems with let-polymorphism and type constructors. In Section 6, we revisit the examples from Section 2 and formulate them as type-and-transform systems. We discuss related work in Section 7, share some future work plans in Section 8, and conclude in Section 9.

2. Examples

To get an intuition of what type-and-transform systems are capable of, we introduce several examples.

2.1 Hughes' Lists

Hughes [15] represents lists as first-class functions. A Haskell expression $e :: [a]$ is represented by an expression $e' :: [a] \to [a]$.

[1] `https://plus.google.com/115504368969270249241/posts/PNoyWzwJJ9y`

The benefit of this representation is that it improves the overall efficiency of using the "append" operation:

$$(++) :: [a] \to [a] \to [a]$$
$$(++)\ [\,] \quad\ ys = ys$$
$$(++)\ (x : xs)\ ys = x : xs ++ ys$$

infixr 5 ++

Rewriting one's code to make use of functions instead of strings amounts to replacing occurrences of ++ by function composition and turning lists into functions. In the end, however, a programmer usually needs a list and not a function on lists, so we use the following two functions to convert between the *abstract* type (used in programs) and the *representation* type (desired, but perhaps not as easy to use):

$$rep :: [a] \to ([a] \to [a])$$
$$rep = (++)$$
$$abs :: ([a] \to [a]) \to [a]$$
$$abs\ f = f\ [\,]$$

Note that $abs\ (rep\ x) \equiv x ++ [\,] \equiv x$ (Hughes' first law) and hence $abs \circ rep \equiv id :: [a] \to [a]$. A simple example of using *abs* and *rep* is the following rewrite[2]:

$$[0,1,1] ++ [2,4,9] \rightsquigarrow abs\ (rep\ [0,1,1] \circ rep\ [2,4,9])$$

We use the following three transformations at particular points in the expressions to arrive at the target from the source.

1. Apply *rep* to list expressions
2. Apply *abs* to a result of an application of *rep*
3. Rename ++ to ∘

Note that we can apply these transformations at different places and in a different order. We later discuss how we arrive at this particular combination of applications of transformation rules.

The functions ++ and ∘ are related. The type of composition, $(b \to c) \to (a \to b) \to (a \to c)$, specialized to functions on lists (of some type a), juxtaposed with the type of ++, looks as follows:

$$(++) :: [a] \to [a] \to [a]$$
$$(\circ)\ \ :: ([a] \to [a]) \to ([a] \to [a]) \to ([a] \to [a])$$

For transformation 3 above, we need to show that ∘ correctly "implements" ++ in the representation. That is, $abs\ (e_1 \circ e_2) \equiv abs\ e_1 ++ abs\ e_2$ (Hughes' second law). The proof is given by Hughes. Returning to the example, we can see that the transformation produces exactly the desired result:

$$\begin{aligned} &abs\ (rep\ [0,1,1] \circ rep\ [2,4,9]) \\ &\quad \equiv ((++)\ [0,1,1] \circ (++)\ [2,4,9])\ [\,] \\ &\quad \equiv [0,1,1] ++ [2,4,9] ++ [\,] \\ &\quad \equiv [0,1,1] ++ [2,4,9] \end{aligned}$$

2.2 Streams

Coutts et al. [2, 3] describe stream fusion, a deforestation technique that eliminates superfluous intermediate values. The constructor of the datatype *Stream a* embeds both a seed value (initially, the list) and a nonrecursive continuation that represents one step in the processing of the value. To convert lists to and from streams, we use the functions:

$$stream\ \ :: [a] \to Stream\ a$$
$$unstream :: Stream\ a \to [a]$$

[2] For now, we use $\rightsquigarrow$ informally to mean the source program on the left is transformed to the target program on the right. Later, we describe a formal mechanism of transformation.

The *stream* function (along with every other *Stream*-producing function) is nonrecursive. The *unstream* function unfolds the stream by recursively applying the continuation. The continuation used in *unstream* effectively becomes the composition of all continuations in the *Stream* functions between *stream* and *unstream*.

Consider this simple composition of list functions (given some functions f, c, and g):

$$map\ f \circ filter\ c \circ map\ g$$

When compiled without optimization, evaluation of this expression would produce two intermediate structures, one for each composition. For stream fusion, we define the following functions:

$$map_s\ \ :: (a \to b) \to Stream\ a \to Stream\ b$$
$$filter_s :: (a \to Bool) \to Stream\ a \to Stream\ a$$

We add these GHC rewrite rules [16] to the stream library:

$$map\ f \rightsquigarrow unstream \circ map_s\ f \circ stream$$
$$filter\ f \rightsquigarrow unstream \circ filter_s\ f \circ stream$$

Then, the above example transforms to:

$$\begin{aligned} &unstream \circ map_s\ f\ \ \circ stream \circ unstream \\ &\qquad \circ filter_s\ c \circ stream \circ unstream \\ &\qquad \circ map_s\ g\ \ \circ stream \end{aligned}$$

We also define a rule for *stream* and *unstream* that removes these intermediate functions.

$$stream\ (unstream\ e) \rightsquigarrow e$$

This leaves us with:

$$unstream \circ map_s\ f \circ filter_s\ c \circ map_s\ g \circ stream$$

The only recursive function left is *unstream*, and the remaining intermediate structures can be fused by compiler optimizations such as inlining.

Consider the slightly more complicated example:

```
let (h,j) =  {- complex code that produces (map f, map g) -}
in  h ∘ filter c ∘ j
```

If GHC decides to not inline h and j, then we might end up with the following:

```
let (h,j) =  {- complex code that produces (map f, map g) -}
in  h ∘ unstream ∘ filter_s c ∘ stream ∘ j
```

At this point, we cannot fuse anything. However, with a transformation system that can pass type changes through locally bound variables and arbitrary code (regardless of complexity), we end up with this:

```
let (h,j) =  {- complex code that produces (map_s f, map_s g) -}
in  unstream ∘ h ∘ filter_s c ∘ j ∘ stream
```

The idea of stream transformations is the same as described in Section 2.1: *rep* is analogous to *stream*, and *abs* is analogous to *unstream*. Our approach differs from stream fusion in that we are not rewriting certain combinations of functions. We are instead creating a boundary between types (here, between $[a]$ and *Stream a*), and we are using transformations such as $map \rightsquigarrow map_s$ and $filter \rightsquigarrow filter_s$ to spread the *Stream* throughout the program until we reach the limits of all possible transformations.

3. A Type-and-Transform System

A type-and-transform system transforms a well-typed source program to a well-typed target program, ensuring that the whole-program semantics remains unchanged. In the following sections, we introduce the underlying language of programs, the concepts

$$(a)\quad \tau ::= b \mid \tau \to \tau \qquad\qquad (b)\quad \mathring{\tau} ::= b \mid \mathring{\tau} \to \mathring{\tau} \mid \iota$$
$$\Gamma ::= \varepsilon \mid \Gamma, x:\tau \qquad\qquad \mathring{\Gamma} ::= \varepsilon \mid \mathring{\Gamma}, x:\mathring{\tau}$$

Figure 1. Grammar for the STLC (a) types and typing environment and (b) type functors and type functor environment

needed for type-and-transform systems, the relation that forms a type-and-transform system, and the properties that determine correctness of a transformation.

3.1 The Object Language

Type-and-transform systems can be built for many programming languages based on the lambda calculus. We call the underlying language the *object language*. We describe some requirements necessary for an object language to support a type-and-transform system.

The language should have a strong type system. We derive a type-and-transform system from the object language's type system. The type system of the object language should satisfy subject reduction: evaluating a well-typed program results in a well-typed program. The safety of a transformation depends on the safety of the type system.

To relate the semantics of the source and target programs in a transformation, we use equational reasoning and $\beta\eta$-equivalence. This implies that imperative languages and other languages which present difficulties for term rewriting may require a different approach.

We introduce type-and-transform systems using the simply typed lambda calculus (STLC) as our object language. We discuss the admittedly more complicated type-and-transform systems for the lambda calculus with let-polymorphism in Section 5 and show how the examples from Section 2, which require the lambda calculus with let-polymorphism and type constructors, are implemented in Section 6.

We use the following expression syntax:

$$e ::= x \mid e\ e \mid \lambda x.e \mid \textbf{fix}\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

Expressions include variables, function application, lambda abstractions, the recursion primitive **fix**, and **let**-bindings.

The syntax for types is given in Figure 1(a). Types include base types such as *Int* and *String* and the arrow type for functions. The type system is defined by the standard judgment $\Gamma \vdash e : \tau$ with the inference rules in Figure 2(a). The typing environment Γ is either empty or extended by a binding $x : \tau$ in which any previous x-mapping in Γ is hidden in the usual way.

As a running example in the following sections, we use a version of the Hughes' lists transformation simplified to strings. We delimit string literals with `""`. We also assume the availability of functions such as $+\!\!+$, $\circ$, and *showString* (defined as $+\!\!+$) for strings. For visual appeal, we make liberal use of infix binary operators (as found in Haskell). All examples can easily be translated to prefix notation.

3.2 Transformations

To motivate the design of type-and-transform systems, let us first consider some potential transformations. Not all of them are desired, and we discuss the validity of each. Consider the following, where $abs = \lambda f.f$ `""` and $rep = (+\!\!+)$:

$$42 \rightsquigarrow 42 \tag{1}$$
$$\texttt{"right"} \rightsquigarrow \texttt{"wrong"} \tag{2}$$
$$upper\ \texttt{"a"} \rightsquigarrow rep\ (abs\ upper)\ \texttt{"a"} \tag{3}$$
$$\texttt{"a"} +\!\!+ \texttt{"b"} \rightsquigarrow rep\ \texttt{"a"} \circ rep\ \texttt{"b"} \tag{4}$$
$$length\ (\texttt{"a"} +\!\!+ \texttt{"b"}) \rightsquigarrow length\ (abs\ (rep\ \texttt{"a"} \circ rep\ \texttt{"b"})) \tag{5}$$
$$(\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} \rightsquigarrow abs\ (\lambda x.rep\ x \circ rep\ \texttt{"b"})\ \texttt{"a"} \tag{6}$$
$$(\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} \rightsquigarrow abs\ ((\lambda x.x \circ rep\ \texttt{"b"})\ (rep\ \texttt{"a"})) \tag{7}$$

A transformation that does not change the source term, such as (1), is trivially correct. A transformation such as (2), which changes the meaning of a program, is trivially incorrect. In (3), *abs* is applied to the function $upper : String \to String$ that changes every character in a string to its uppercase form. The source reduces to `"A"` while the target reduces to `"a"`, and thus the transformation is incorrect. In (4), we see appropriate applications of *rep* and $\circ$ (according to the transformations described in Section 2.1); however, the transformation is not yet "complete," since the types are different. If this source term were embedded in a program, then (4) would be correct: this is shown in (5). Another way to complete the transformation of (4) is to apply *abs* to the target. At a glance, (6) appears correct, but the source reduces to `"ab"` and the target to `"ba"`. A correct transformation for this source program is (7).

A type-and-transform system cannot allow the transformations in (2), (3), (4), and (6) but should allow (1), (5), and (7). From these examples we observe:

- A transformation cannot change the type of a program, but it can change the type of a term within the program. We distinguish the notion of *term transformation*, a correct but possibly incomplete transformation, from that of *program transformation*, the transformation of a whole program. Note that a term transformation may still involve multiple rewriting steps.
- A transformation cannot arbitrarily change the meaning of an expression. There should be a relation between every transformed source and target term establishing that the semantics is preserved (even if the type changes).
- The types of locally bound variables can change through a transformation.
- If we use the inferred types of *rep* and *abs*, we can get transformations such as (6). We specialize the types of these functions, e.g. $abs : (String \to String) \to String$, to avoid these incorrect transformations.
- While $abs \circ rep \equiv id$, we see in (3) that $rep \circ abs \equiv id$ does not hold in general.

In the next section, we introduce the concept of a type functor, which expresses many of the above requirements.

3.3 Type Functors

The pair of types mentioned in previous sections have a special relationship. These types, $\mathcal{A}$ and $\mathcal{R}$, form a retract, $\mathcal{A} \triangleleft \mathcal{R}$, with the pair of functions $abs : \mathcal{R} \to \mathcal{A}$ and $rep : \mathcal{A} \to \mathcal{R}$. A retract has the property that *abs* is a left-inverse of *rep* or:

$$abs \circ rep \equiv id : \mathcal{A} \to \mathcal{A}$$

In Sections 2.1 and 2.2, we saw the retracts $[a] \triangleleft [a] \to [a]$ and $[a] \triangleleft Stream\ a$. Our running example is the retract $String \triangleleft String \to String$.

In Section 3.2, we saw that a term transformation can result in a target term with a different type from the source term. The relationship between the types is defined precisely with the *type functor* $\mathring{\tau}$ defined in Figure 1(b). We define a distinguished type ι not found in the object language types. A transformation is typed

$$(a)\quad \boxed{\Gamma \vdash e : \tau}$$

$$\text{T-Var}\quad \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\text{T-App}\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{T-Lam}\quad \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{T-Fix}\quad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathbf{fix}\ e : \tau}$$

$$\text{T-Let}\quad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x:\tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$(b)\quad \boxed{\mathring{\Gamma} \vdash e \rightsquigarrow e : \mathring{\tau}}$$

$$\text{TT-Var}\quad \frac{x:\mathring{\tau} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash x \rightsquigarrow x : \mathring{\tau}}$$

$$\text{TT-App}\quad \frac{\mathring{\Gamma} \vdash e_1 \rightsquigarrow e_1' : \mathring{\tau}_1 \to \mathring{\tau}_2 \qquad \mathring{\Gamma} \vdash e_2 \rightsquigarrow e_2' : \mathring{\tau}_1}{\mathring{\Gamma} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \mathring{\tau}_2}$$

$$\text{TT-Lam}\quad \frac{\mathring{\Gamma}, x:\mathring{\tau}_1 \vdash e \rightsquigarrow e' : \mathring{\tau}_2}{\mathring{\Gamma} \vdash \lambda x.e \rightsquigarrow \lambda x.e' : \mathring{\tau}_1 \to \mathring{\tau}_2}$$

$$\text{TT-Fix}\quad \frac{\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau} \to \mathring{\tau}}{\mathring{\Gamma} \vdash \mathbf{fix}\ e \rightsquigarrow \mathbf{fix}\ e' : \mathring{\tau}}$$

$$\text{TT-Let}\quad \frac{\mathring{\Gamma} \vdash e_1 \rightsquigarrow e_1' : \mathring{\tau}_1 \qquad \mathring{\Gamma}, x:\mathring{\tau}_1 \vdash e_2 \rightsquigarrow e_2' : \mathring{\tau}}{\mathring{\Gamma} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2' : \mathring{\tau}}$$

Figure 2. (a) Inference rules for the type system. (b) Propagation inference rules for the type-and-transform relation.

by a type functor, where the differences between the source and target types are replaced by the "hole" ι. Here are a few examples:

$$\begin{aligned} \texttt{"a"} &\rightsquigarrow \mathit{rep}\ \texttt{"a"} && : \iota \\ (+\!+) &\rightsquigarrow (\circ) && : \iota \to \iota \to \iota \\ \lambda x.x +\!+ \texttt{"b"} &\rightsquigarrow \lambda x.x \circ \mathit{rep}\ \texttt{"b"} && : \iota \to \iota \\ \texttt{"a"} &\rightsquigarrow \mathit{abs}\ (\mathit{rep}\ \texttt{"a"}) && : \mathit{String} \end{aligned} \tag{8}$$

The holes occur only in the positions where the source has the type $\mathcal{A}$ (*String*) and the target has the type $\mathcal{R}$ (*String* → *String*).

The type functor is determined from the source and target types as follows:

$$\begin{aligned} &\mathit{mgtf}\ \mathcal{A} && \mathcal{R} && = \iota \\ &\mathit{mgtf}\ b_1 && b_2 \mid b_1 \equiv b_2 && = b_1 \\ &\mathit{mgtf}\ (\tau_1 \to \tau_2)\ (\tau_3 \to \tau_4) && && = \mathit{mgtf}\ \tau_1\ \tau_3 \to \mathit{mgtf}\ \tau_2\ \tau_4 \end{aligned}$$

mgtf is a partial function: any transformation of which the types do not determine a type functor is not well-typed.

We recover the types by interpreting the type functor:

$$\begin{aligned} [\![b]\!]_\tau &= b \\ [\![\iota]\!]_\tau &= \tau \\ [\![\mathring{\tau}_1 \to \mathring{\tau}_2]\!]_\tau &= [\![\mathring{\tau}_1]\!]_\tau \to [\![\mathring{\tau}_1]\!]_\tau \end{aligned}$$

The interpretation replaces every ι with the type argument. For the source, the argument is $\mathcal{A}$ and for the target, $\mathcal{R}$. For example, the types of the source and target in (8) are $\mathcal{A} \to \mathcal{A}$ and $\mathcal{R} \to \mathcal{R}$, respectively.

For working with type functors, we also need the environment defined in Figure 1(b). We use the following interpretation of the type functor environment to retrieve the typing environment for a given source or target:

$$\begin{aligned} [\![\varepsilon]\!]_\tau &= \varepsilon \\ \left[\!\!\left[\mathring{\Gamma}, x:\mathring{\tau}\right]\!\!\right]_\tau &= \left[\!\!\left[\mathring{\Gamma}\right]\!\!\right]_\tau, x:[\![\mathring{\tau}]\!]_\tau \end{aligned}$$

In future sections, we use type and type functor interchangeably. It should be clear from the context which is meant.

3.4 The Type-and-Transform Relation

Typing proves the judgment $\Gamma \vdash e : \tau$ with a derivation using inference rules to relate a term e to a type τ within the context Γ. Similarly, transforming proves the judgment $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ to relate a source term e to a target term e' and a type functor $\mathring{\tau}$ within the context $\mathring{\Gamma}$.

The (core) type-and-transform inference rules, given in Figure 2(b), show a clear analogy with the type inference rules. Every rule is immediately derived from its counterpart typing rule. This system is actually a slight generalization of the type system where terms are duplicated and typed with type functors. If no ι types are used, derivations using the left and right systems of Figure 2 are isomorphic.

The inference rules relate source and target terms with a type functor. They also map variables to type functors instead of types. This allows a variable's type to change during transformation.

Figure 2(b) gives inference rules that are necessary for typing but not sufficient for transformation. We extend the system with the following two rules:

$$\text{TT-Rep}\quad \frac{\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathcal{A} \qquad \mathit{rep} : \mathcal{A} \to \mathcal{R} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash e \rightsquigarrow \mathit{rep}\ e' : \iota}$$

$$\text{TT-Abs}\quad \frac{\mathring{\Gamma} \vdash e \rightsquigarrow e' : \iota \qquad \mathit{abs} : \mathcal{R} \to \mathcal{A} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash e \rightsquigarrow \mathit{abs}\ e' : \mathcal{A}}$$

These rules define two particular transformations. In TT-Rep, we assume a transformation with the type functor $\mathcal{A}$. This implies that both source and target have the type $\mathcal{A}$ since $[\![\mathcal{A}]\!]_\mathcal{A} \equiv [\![\mathcal{A}]\!]_\mathcal{R} \equiv \mathcal{A}$. We transform the target term by applying *rep* to it. We assign the transformation the type functor $\mathit{mgtf}\ \mathcal{A}\ \mathcal{R} \equiv \iota$. In TT-Abs, we assume a transformation typed ι (thus $e : \mathcal{A}$ and $e' : \mathcal{R}$). We apply *abs* to the target. Since the source and target types are now both $\mathcal{A}$, we assign the transformation the type $\mathit{mgtf}\ \mathcal{A}\ \mathcal{A} \equiv \mathcal{A}$.

The system now relates a well-typed source program to a well-typed, transformed target program. We call TT-Rep and TT-Abs *transformation rules*, and we refer to the rules in Figure 2(b) as *propagation rules* since they propagate type changes through but do not transform a program. Propagation rules are derived from the typing rules and are thus language-specific. Transformation rules describe a particular transformation. The *type-and-transform relation* is the union of these two rule sets.

The location of ι in TT-Rep and TT-Abs is significant. It determines the ordering of application for *rep* and *abs*. The function *rep* is the only term producing "values" of the type ι, and the only consumer of ι values is *abs*. Consequently, *abs* cannot be applied to *upper* in example 3 from Section 3.2.

The type-and-transform system as described is of limited value, since *abs* and *rep* can only be applied in limited ways. We increase the utility of a type-and-transform system by extending it with

more transformation rules. Every added rule gives the system more possibilities to transform a program.

Continuing with the strings example, we extend our system with the following rule:

$$\text{TT-Comp}\quad \frac{(+\!+) : \mathcal{A} \to \mathcal{A} \to \mathcal{A} \in \mathring{\Gamma} \qquad (\circ) : \mathcal{R} \to \mathcal{R} \to \mathcal{R} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash (+\!+) \rightsquigarrow (\circ) : \iota \to \iota \to \iota}$$

As with the previous transformation rules, the type functor of the transformation and the source and target types are clearly related. Again, the location of ι is significant. In this case, we are only renaming a variable, but the type functor must indicate exactly where the types change, both in the parameters and the results.

This brings us to our first property:

Theorem 1 (Transformation preserves typing) If $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ is a term transformation and $\mathcal{A} \triangleleft \mathcal{R}$ is the retract, then the source and target terms are well-typed when the type functors are interpreted under the types of the retract:

$$\left[\!\left[\mathring{\Gamma}\right]\!\right]_{\mathcal{A}} \vdash e : [\![\mathring{\tau}]\!]_{\mathcal{A}} \qquad\qquad \left[\!\left[\mathring{\Gamma}\right]\!\right]_{\mathcal{R}} \vdash e' : [\![\mathring{\tau}]\!]_{\mathcal{R}}$$

The proof is by straightforward induction on the inference rules.

In the next section, we describe how a program transformation is well-typed.

3.5 Completion

As indicated in Section 3.2, allowing for transformations to change the type of a whole program is too permissive. A successful program transformation should produce a target that is, for all intents and purposes, no different from the source. Among other things, its type should not change.

We describe a transformation as having the property of *completion* (not "completeness") if it returns a target program with the same type as the source program. To formally formulate the property, we need a way to indicate that a type functor has no holes. We do this by turning the type functor interpretation from a total function into a partial function, that is, when a hole is encountered, interpretation fails (returns $\bot$): $[\![\iota]\!]_{\bot} \equiv \bot$.

Theorem 2 (Completion) If $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ is a program transformation, then the source and target programs are well-typed and have the same type when the type functors have no holes:

$$\left[\!\left[\mathring{\Gamma}\right]\!\right]_{\bot} \vdash e : [\![\mathring{\tau}]\!]_{\bot} \qquad\qquad \left[\!\left[\mathring{\Gamma}\right]\!\right]_{\bot} \vdash e' : [\![\mathring{\tau}]\!]_{\bot}$$

The proof is by straightforward induction on the inference rules.

In the next section, we describe how a transformation preserves the semantics of the source program in the target program.

3.6 Correctness

We would like to make the following statement: if $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ is a program transformation, then $e \equiv e'$, where $\equiv$ is $\beta\eta$-equivalence on terms. This property does not hold for every term transformation (e.g. with $\texttt{"a"} \rightsquigarrow \mathit{rep}\ \texttt{"a"}$, $\texttt{"a"} \not\equiv \mathit{rep}\ \texttt{"a"}$), but we can describe a more general property for all term transformations and then use a variant of that property for program transformations.

We use type functors to type transformations and indicate where types change. If we consider type functors to be the type-level mapping of objects, we can also define a term-level mapping of morphisms. However, we cannot use normal (covariant) functors because function types have negative occurrences. We need mixed-variant functors or *difunctors* [20]. Whereas the typical difunctor might look like $F\ a\ b$ with a in the contravariant position and b in the covariant position, we need only one parameter a in both positions: $[\![\mathring{\tau}]\!]_a$. Together with the type constructor, the function *dimap* implements a difunctor:

$$\begin{aligned}
&\mathit{dimap}_{\mathring{\tau}} : (a \to b) \to (b \to a) \to [\![\mathring{\tau}]\!]_b \to [\![\mathring{\tau}]\!]_a \\
&\mathit{dimap}_b \qquad f\ g = \mathit{id} \\
&\mathit{dimap}_{\mathring{\tau}_1 \to \mathring{\tau}_2}\ f\ g = \lambda x \to \mathit{dimap}_{\mathring{\tau}_2}\ f\ g \circ x \circ \mathit{dimap}_{\mathring{\tau}_1}\ g\ f \\
&\mathit{dimap}_{\iota} \qquad f\ g = g
\end{aligned}$$

We define *dimap* as a type-indexed function. At base types, *dimap* is the identity. At function types, *dimap* is the composition of the parameter *dimap*, the function itself, and the result *dimap*. Note that the arguments are swapped in the parameter *dimap*. At the hole type, *dimap* is the covariant argument. Additionally, *dimap* obeys the following laws of identity and distribution over composition:

$$\begin{aligned}
&\mathit{dimap}_{\mathring{\tau}}\ \mathit{id} \qquad \mathit{id} \quad \equiv\ \mathit{id} \\
&\mathit{dimap}_{\mathring{\tau}}\ (g \circ h)\ (i \circ j) \ \equiv\ \mathit{dimap}_{\mathring{\tau}}\ h\ i \circ \mathit{dimap}_{\mathring{\tau}}\ g\ j
\end{aligned}$$

When we apply $\mathit{dimap}_{\iota}\ \mathit{rep}\ \mathit{abs}$ to a term, we get the reverse transformation of that term. For example, $\mathit{dimap}_{\iota}\ \mathit{rep}\ \mathit{abs}\ (\mathit{rep}\ \texttt{"a"}) \equiv \mathit{abs}\ (\mathit{rep}\ \texttt{"a"}) \equiv \texttt{"a"}$.

In addition to difunctors, we need capture-avoiding substitution. A substitution maps variables to terms (or types as we will need later), $[x \mapsto e]$, and can be the identity, *id*, or the composition of other substitutions, $\theta_1 \circ \theta_2$. Substitutions are applied by juxtaposition, θe, which has higher precedence than function application. A term transformation may have free variables bound in a type functor environment. To accurately describe a mapping between source and target terms, we derive a substitution from the environment:

$$\begin{aligned}
&\theta_{\mathcal{E}} \quad = \mathit{id} \\
&\theta_{\mathring{\Gamma}, x : \mathring{\tau}} = \theta_{\mathring{\Gamma}} \circ [x \mapsto \mathit{dimap}_{\mathring{\tau}}\ \mathit{abs}\ \mathit{rep}\ x]
\end{aligned}$$

Now, we can state the correctness property for term transformations:

Theorem 3 (Difunctors encode transformation) If $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ is a term transformation and $\mathcal{A} \triangleleft \mathcal{R}$ is the retract with functions *rep* and *abs*, then the source term is $\beta\eta$-equivalent to the *dimap* of the target term:

$$e \equiv \mathit{dimap}_{\mathring{\tau}}\ \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}} e'$$

The proof is by induction on the inference rules. We show the TT-App rule, repeated here for convenience:

$$\text{TT-App}\quad \frac{\mathring{\Gamma} \vdash e_1 \rightsquigarrow e_1' : \mathring{\tau}_1 \to \mathring{\tau}_2 \qquad \mathring{\Gamma} \vdash e_2 \rightsquigarrow e_2' : \mathring{\tau}_1}{\mathring{\Gamma} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \mathring{\tau}_2}$$

The goal of the proof is:

$$\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}}(e_1'\ e_2') \equiv e_1\ e_2$$

From the premises, we know:

$$\begin{aligned}
&\mathit{dimap}_{\mathring{\tau}_1 \to \mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}} e_1' \equiv e_1 \\
&\mathit{dimap}_{\mathring{\tau}_1} \qquad \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}} e_2' \equiv e_2
\end{aligned}$$

The proof:

$$\begin{aligned}
&\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}}(e_1'\ e_2') \\
\equiv\ &\{\ \text{Distribute substitution}\ \} \\
&\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ (\theta_{\mathring{\Gamma}} e_1'\ \theta_{\mathring{\Gamma}} e_2') \\
\equiv\ &\{\ \text{Difunctor identity}\ \} \\
&\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ (\theta_{\mathring{\Gamma}} e_1'\ (\mathit{dimap}_{\mathring{\tau}_1}\ \mathit{id}\ \mathit{id}\ \theta_{\mathring{\Gamma}} e_2')) \\
\equiv\ &\{\ \mathit{rep} \circ \mathit{abs} \equiv \mathit{id}\ \} \\
&\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ (\theta_{\mathring{\Gamma}} e_1'\ (\mathit{dimap}_{\mathring{\tau}_1}\ (\mathit{rep} \circ \mathit{abs})\ (\mathit{rep} \circ \mathit{abs})\ \theta_{\mathring{\Gamma}} e_2')) \\
\equiv\ &\{\ \text{Difunctor composition}\ \} \\
&\mathit{dimap}_{\mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs} \\
&\quad (\theta_{\mathring{\Gamma}} e_1'\ ((\mathit{dimap}_{\mathring{\tau}_1}\ \mathit{abs}\ \mathit{rep} \circ \mathit{dimap}_{\mathring{\tau}_1}\ \mathit{rep}\ \mathit{abs})\ \theta_{\mathring{\Gamma}} e_2')) \\
\equiv\ &\{\ \text{Definition of } \mathit{dimap}_{\mathring{\tau}_1 \to \mathring{\tau}_2}\ \} \\
&\mathit{dimap}_{\mathring{\tau}_1 \to \mathring{\tau}_2}\ \mathit{rep}\ \mathit{abs}\ (\theta_{\mathring{\Gamma}} e_1'\ (\mathit{dimap}_{\mathring{\tau}_1}\ \mathit{rep}\ \mathit{abs}\ \theta_{\mathring{\Gamma}} e_2')) \\
\equiv\ &\{\ \text{Premises}\ \} \\
&e_1\ e_2
\end{aligned}$$

It turns out that we need an additional property for the proof: the inverse identity $rep \circ abs \equiv id : \iota \to \iota$. As we pointed out in Section 3.2, this does not hold in general. However, it does hold during transformation thanks to type functors.

The basic idea is that terms of type ι can only be constructed by transformations rules. In the rules given, that includes TT-Rep and TT-Comp. So, we only have to show that $rep \circ abs \equiv id$ for terms constructed from these rules. This is the lemma:

Lemma 1 (Normal form of ι) For every term transformation $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \iota$, the target term is $\beta\eta$-equivalent to *rep* applied to the source term:

$$e' \equiv rep\ e$$

After $\beta\eta$-reduction of e' to one of the normal forms of ι terms (either $rep\ e$ or $e_1 \circ e_2$), the proof is by induction on those forms. For *rep*, this is trivial. The proof for $\circ$ is found in Appendix A.2.

Now, the proof for $rep \circ abs \equiv id$ is straightforward:

$$\begin{array}{ll} & rep\ (abs\ e') \\ \equiv & \{\ \text{Lemma 1}\ \} \\ & rep\ (abs\ (rep\ e)) \\ \equiv & \{\ abs \circ rep \equiv id\ \} \\ & rep\ e \\ \equiv & \{\ \text{Lemma 1}\ \} \\ & e' \end{array}$$

The remaining inference rule proofs are given in Appendix A.1.

The following property states how semantics is preserved in a program transformation:

Theorem 4 (Transformation preserves semantics) If $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau}$ is a program transformation and $\mathcal{A} \triangleleft \mathcal{R}$ is the retract with functions *rep* and *abs*, then the source term is $\beta\eta$-equivalent to the target term:

$$e \equiv e'$$

For Theorem 4, we need the following lemmas:

Lemma 2 (*dimap* without holes) If $[\![\mathring{\tau}]\!]_\perp$ is a valid type, then $dimap_{\mathring{\tau}}\ f\ g \equiv id$.

The proof is by straightforward induction on the structure of $\mathring{\tau}$.

Lemma 3 ($\mathring{\Gamma}$ without holes) If $\left[\!\left[\mathring{\Gamma}\right]\!\right]_\perp$ is a valid environment, then $\theta_{\mathring{\Gamma}} \equiv id$.

The proof is by induction on the structure of $\mathring{\Gamma}$. In the $\mathring{\Gamma}, x : \mathring{\tau}$ case, we use Lemma 2 for $dimap_{\mathring{\tau}}\ abs\ rep \equiv id$. Here is the proof of Theorem 4:

$$\begin{array}{ll} & e' \\ \equiv & \{\ \text{Theorem 3}\ \} \\ & dimap_{\mathring{\tau}}\ rep\ abs\ \theta_{\mathring{\Gamma}} e \\ \equiv & \{\ \text{Theorem 2 and Lemma 2}\ \} \\ & \theta_{\mathring{\Gamma}} e \\ \equiv & \{\ \text{Theorem 2 and Lemma 3}\ \} \\ & e \end{array}$$

This concludes the discussion on correctness. In the next section, we describe an implementation of a type-and-transform system.

4. Implementation

We describe an implementation of the type-and-transform system described in the previous section. The implementation here uses inference rules. For an executable implementation, download the Haskell source [3].

One issue for type-and-transform systems is the nondeterminism of transformation rules. In Figure 2, as with the type inference rules, the propagation rules are syntax-directed. But some transformation rules such as TT-Abs and TT-Rep, can be applied to expressions of almost arbitrary form. For a single source program, there are numerous targets:

$$\begin{array}{l} (\lambda x.x)\ \texttt{"a"} \rightsquigarrow (\lambda x.x)\ (abs\ (rep\ \texttt{"a"})) \\ \qquad\qquad \rightsquigarrow (\lambda x.abs\ x)\ (rep\ \texttt{"a"}) \\ \qquad\qquad \rightsquigarrow abs\ (rep\ (\lambda x.x)\ \texttt{"a"}) \\ \qquad\qquad \rightsquigarrow abs\ (rep\ \cdots\ (abs\ (rep\ (\lambda x.x)\ \texttt{"a"}))\ \cdots) \end{array}$$

As the last example implies, the possibilities are infinite. A naive type-and-transform algorithm would not terminate!

A second issue is that type changes "flow" through a program (via the propagation rules and especially locally bound variables) in unpredictable ways. Consider the following transformations:

$$\begin{array}{ll} (\lambda x.x +\!\!+ \texttt{"b"})\ \texttt{"a"} \rightsquigarrow & (\lambda x.rep\ x \circ rep\ \texttt{"b"})\quad \texttt{"a"} \\ \rightsquigarrow & (\lambda x.abs\ (x \circ rep\ \texttt{"b"}))\ (rep\ \texttt{"a"}) \end{array}$$

In the first case, we find a function of type $String \to \iota$ and an argument of type *String*. We have transformed the function without changing its parameter type, so the argument does not need to change. In the second case, the function has the type $\iota \to String$ and the argument has the type ι. Thus, we have transformed both the function and the argument such that the parameter type unifies with the argument type. At the application, we cannot necessarily decide whether the function or the argument should be transformed. The transformation "driver" (that which changes types) can come from either (or neither), and the choice may be arbitrary. We could always makes the same choice; however, we think this approach rules out many desired transformations. In our experiments, we often observed that a locally optimal choice does not produce a global optimum. As a result, we prefer to leave the algorithm as flexible as possible.

To solve the above issues, we give an algorithm in Section 4.3. It is nondeterministic (produces multiple results) but can pick a "best" result using simple heuristics. We avoid termination problems by only applying a single transformation rule at each node (e.g. application, **let**, etc.) in the program. Thus, we never have transformations such as $\texttt{"a"} \rightsquigarrow abs\ (rep\ \texttt{"a"})$ (or any other application of *abs* directly after *rep*). To avoid flow issues, we use a bottom-up algorithm derived from constraint-based type inference, which we introduce in the next section.

4.1 Constraint-Based Type Inference

Many inference algorithms traverse the children in an application node in a top-down way: visit one child (e.g. the function) first and visit the sibling (e.g. the argument) second. In type-and-transform systems, transformations can occur nondeterministically, and types can change as a result. We wish to propagate the type change throughout a program, and it can come from a function or an argument. It is unclear how to do this by visiting the application node children in the above fashion, because the type change of one can result in the transformation of another, which may feed back to the first (e.g. through a locally bound variable). A far simpler solution is to transform both children in all possible ways and find which combinations are successful. For this, we use a bottom-up algorithm based on constraint-based type inference.

In constraint-based type inference [13], we collect type-equality constraints from the children of each node and solve them with

[3] `http://www.staff.science.uu.nl/~leath101/publications/icfp2012/tts-0.3.tar.gz`

unification to determine types. For example, a function type τ_1 and argument type τ_2 must be related by the constraint $\tau_1 \equiv \tau_2 \to \beta$, where β is a fresh type variable[4]. The solution to this constraint is the standard most general unifier, $mgu\ \tau_1\ (\tau_2 \to \beta)$. The unifier is a substitution θ that, when applied to one of the arguments, $\theta\tau_1$, gives the principal type. When there is a set of constraints C to be solved, we use the following function:

$$\begin{aligned} &solve\ \varepsilon &&= id \\ &solve\ (\{\tau_1 \equiv \tau_2\} \cup C) &&= \textbf{let}\ \theta = mgu\ \tau_1\ \tau_2\ \textbf{in}\ \theta \circ solve\ \theta C \end{aligned}$$

Since the algorithm is bottom-up, the types of variables may not initially be known. We use an assumption environment, A, whose form is the same as Γ, to track the known type of every variable whose binding is still unknown. When a binding is found for a variable v, we remove all mappings from A: $A \backslash v = \{x : \mathring{\tau} \mid x : \mathring{\tau} \in A \wedge x \not\equiv v\}$.

The judgment for our constraint-based type inference algorithm is $\Gamma, A \vdash e : \tau$, and the algorithm is shown in Figure 3(a). This algorithm is adapted from [13] with a few minor differences. First, we do not pass any constraints up from one node to the next. We solve all constraints at the "earliest" possible point. This ensures that the types are the most specific, a boon to efficiency in the type-and-transform algorithm as we will see. Second, we include the top-down environment Γ along with the bottom-up assumptions A. As a consequence, we have two alternatives for the C-Var rule, one when a variable is mapped by Γ and one when we assume it to be typed by a fresh variable. The Γ is not strictly necessary, since we can always ensure that the free variables of A map correctly to some initial environment. Again, however, it is useful to have the most specific types.

Note that both $\Gamma \vdash e : \tau$ and $\Gamma, A \vdash e : \tau$ define the same type system [13]. Also, for an environment Γ and closed term e, both $\Gamma \vdash e : \tau$ and $\Gamma, \varepsilon \vdash e : \tau$ give the same type τ.

In the next section, we describe how to do constraint-based typed rewriting.

4.2 Typed Rewriting

Previously, we described both propagation and transformation rules in the same relation. In the implementation, we have different roles for these two components. Propagation serves as the algorithm for constructing target terms: we discuss it in the next section. Transformation rules are provided as input to the system. The retract type pair $\mathcal{A} \triangleleft \mathcal{R}$ is also an input, and we assume it implicitly throughout.

Recall the running example of Section 3. It involved the retract $String \triangleleft String \to String$ and the three transformation rules. We now write these rules as follows:

$$m \rightsquigarrow showString\ (m : String) : \iota \tag{9}$$

$$m \rightsquigarrow (m : \iota)\ "" \quad : String \tag{10}$$

$$(++) \rightsquigarrow (\circ) \quad : \iota \to \iota \to \iota \tag{11}$$

Rather than define *abs* and *rep*, we directly use their definitions: *rep* is *showString* and *abs* is application to the empty string. Rule 9 implements TT-Rep: the metavariable m has the same type as the premise, *String* (equivalently: $\mathcal{A}$), and the target type is ι as it is in the conclusion of the rule. Rules 10 and 11 implement TT-Abs and TT-Comp, respectively, in a similar fashion.

Transformation rules are given with the following syntax:

$$\begin{aligned} r &::= p \rightsquigarrow \pi : \mathring{\tau} \\ p &::= x \mid m \mid p\ p \\ \pi &::= x \mid m : \mathring{\tau} \mid \pi\ \pi \end{aligned}$$

[4] We now extend types with type variables α. We use β to mean a type variable that is fresh for a given context.

A rule r is a triple with a source pattern, a target pattern, and a type functor. A pattern may include variables, metavariables, and application. In target patterns, each metavariable is annotated with a type functor.

A rule set ρ is the set of transformation rules for one type-and-transform system. Every type functor in each rule of a rule set is interpreted using a retract for the rule set.

Rules need to satisfy a number of properties. First, they must be well-typed according to Theorem 1. Typing is straightforward, and we do not give the details here. Second, they must satisfy the *dimap* equivalence from Theorem 3. Third, at most one unique metavariable may appear in the rule. If a rule contains a metavariable, it must appear once in both the source and the target. The metavariable restriction exists because the propagation prevents the arbitrary combination of terms in a transformation. For example, the following rule will never work because the types of the function and argument will not unify during propagation

$$m_1\ m_2 \rightsquigarrow (m_2 : Int \to String)\ (m_1 : Int) : \iota$$

To transform a term, we use *typed rewriting*. Typed rewriting is an adaptation of term rewriting with pattern matching on typed terms and typing the application of substitutions. We discuss each of these in turn.

Typed pattern matching requires a special substitution θ^τ, that maps each metavariable to a triple of an assumption set, a expression, and a type. The judgment for typed pattern matching takes an environment, source pattern, and source term as input and produces a substitution:

$$\mathring{\Gamma} \vdash p @ e \Rightarrow \theta^\tau$$

Typed pattern matching is implemented with the following rules:

$$\text{M-Var} \quad \frac{}{\mathring{\Gamma} \vdash x @ x \Rightarrow id}$$

$$\text{M-MVar} \quad \frac{\mathring{\Gamma}, A \vdash e : \mathring{\tau}}{\mathring{\Gamma} \vdash m @ e \Rightarrow [m \mapsto (A, e, \mathring{\tau})]}$$

$$\text{M-App} \quad \frac{\mathring{\Gamma} \vdash p_1 @ e_1 \Rightarrow \theta_1^\tau \qquad \mathring{\Gamma} \vdash p_2 @ e_2 \Rightarrow \theta_2^\tau}{\mathring{\Gamma} \vdash p_1\ p_2 @ e_1\ e_2 \Rightarrow \theta_1^\tau \circ \theta_2^\tau}$$

Object variables (M-Var) match if the pattern variable and term variable are equal. Metavariables (M-MVar) match arbitrary well-typed terms and produces a singleton substitution. A pattern application (M-App) matches a term application if the subpatterns match the corresponding subterms.

In typed substitution application, we apply the special substitution to a pattern, resulting in a well-typed term:

$$\mathring{\Gamma}, A \vdash \theta^\tau @ \pi \Rightarrow e : \mathring{\tau}$$

The inputs are the environment, the substitution, the target pattern, and the outputs are the target term, its type, and its assumption set. Typed substitution application is defined by the following rules:

$$\text{A-Var} \quad \frac{x : \mathring{\tau} \in \mathring{\Gamma}}{\mathring{\Gamma}, \varepsilon \vdash \theta^\tau @ x \Rightarrow x : \mathring{\tau}}$$

$$\text{A-MVar} \quad \frac{(A, e, \mathring{\tau}') = \theta^\tau m \qquad \theta = solve\ \{\mathring{\tau} \equiv \mathring{\tau}'\}}{\mathring{\Gamma}, \theta A \vdash \theta^\tau @ m : \mathring{\tau} \Rightarrow e : \theta\mathring{\tau}'}$$

$$\text{A-App} \quad \frac{\mathring{\Gamma}, A_1 \vdash \theta^\tau @ \pi_1 \Rightarrow e_1 : \mathring{\tau}_1 \qquad \mathring{\Gamma}, A_2 \vdash \theta^\tau @ \pi_2 \Rightarrow e_2 : \mathring{\tau}_2 \qquad \theta = solve\ \{\mathring{\tau}_1 \equiv \mathring{\tau}_2 \to \beta\}}{\mathring{\Gamma}, \theta(A_1 \cup A_2) \vdash \theta^\tau @ \pi_1\ \pi_2 \Rightarrow e_1\ e_2 : \theta\beta}$$

Object variables (A-Var) are not affected by substitution. Substitution on a metavariable (A-MVar) produces a term, its assumptions, and its type. Due to the rule properties mentioned earlier, the substitution will always apply. The metavariable type annotation is used

(*a*) $\boxed{\Gamma, A \vdash e : \tau}$

$$\textsc{C-Var}\quad \frac{x:\tau \in \Gamma}{\Gamma, \varepsilon \vdash x:\tau} \qquad \frac{x:\tau \notin \Gamma}{\Gamma, \{x:\beta\} \vdash x:\beta}$$

$$\textsc{C-App}\quad \frac{\Gamma, A_1 \vdash e_1 : \tau_1 \qquad \Gamma, A_2 \vdash e_2 : \tau_2 \qquad \theta = solve\,\{\tau_1 \equiv \tau_2 \to \beta\}}{\Gamma, \theta(A_1 \cup A_2) \vdash e_1\ e_2 : \theta\beta}$$

$$\textsc{C-Lam}\quad \frac{\Gamma \backslash x, A \vdash e : \tau \qquad \theta = solve\,\{\beta \equiv \tau_x \mid x:\tau_x \in A\}}{\Gamma, \theta(A \backslash x) \vdash \lambda x.e : \theta(\beta \to \tau)}$$

$$\textsc{C-Let}\quad \frac{\Gamma, A_1 \vdash e_1 : \tau_1 \qquad \Gamma \backslash x, A_2 \vdash e_2 : \tau_2 \qquad \theta = solve\,\{\tau_1 \equiv \tau_x \mid x:\tau_x \in A_2\}}{\Gamma, \theta(A_1 \cup A_2 \backslash x) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \theta\tau_2}$$

$$\textsc{C-Fix}\quad \frac{\Gamma, A \vdash e : \tau \qquad \theta = solve\,\{\tau \equiv \beta \to \beta\}}{\Gamma, \theta A \vdash \textbf{fix}\ e : \theta\beta}$$

(*b*) $\boxed{\mathring{\Gamma}, [A_i]^n \vdash_{\text{pr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n}$

$$\textsc{P-Var}\quad \frac{x:\mathring{\tau} \in \mathring{\Gamma}}{\mathring{\Gamma}, \varepsilon \vdash_{\text{pr}} x \rightsquigarrow_\rho x : \mathring{\tau}} \qquad \frac{x:\mathring{\tau} \notin \mathring{\Gamma}}{\mathring{\Gamma}, \{x:\beta\} \vdash_{\text{pr}} x \rightsquigarrow_\rho x : \beta}$$

$$\textsc{P-App}\quad \frac{\mathring{\Gamma}, [A_i]^m \vdash_{\text{tr}} e_1 \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^m \qquad \mathring{\Gamma}, [A_j]^n \vdash_{\text{tr}} e_2 \rightsquigarrow_\rho [e_j : \mathring{\tau}_j]^n \qquad [\theta_{ij} = solve\,\{\mathring{\tau}_i \equiv \mathring{\tau}_j \to \beta_{ij}\}]^{mn}}{\mathring{\Gamma}, [\theta_{ij}(A_i \cup A_j)]^{mn} \vdash_{\text{pr}} e_1\ e_2 \rightsquigarrow_\rho [e_i\ e_j : \theta_{ij}\beta_{ij}]^{mn}}$$

$$\textsc{P-Lam}\quad \frac{\mathring{\Gamma} \backslash x, [A_i]^n \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n \qquad [\theta_i = solve\,\{\beta_i \equiv \mathring{\tau}_x \mid x:\mathring{\tau}_x \in A_i\}]^n}{\mathring{\Gamma}, [\theta_i(A_i \backslash x)]^n \vdash_{\text{pr}} \lambda x.e \rightsquigarrow_\rho [\lambda x.e_i]^n : \theta_i(\beta_i \to \mathring{\tau}_i)}$$

$$\textsc{P-Let}\quad \frac{\mathring{\Gamma}, [A_i]^m \vdash_{\text{tr}} e_1 \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^m \qquad \mathring{\Gamma} \backslash x, [A_j]^n \vdash_{\text{tr}} e_2 \rightsquigarrow_\rho [e_j : \mathring{\tau}_j]^n \qquad [\theta_{ij} = solve\,\{\mathring{\tau}_i \equiv \mathring{\tau}_x \mid x:\mathring{\tau}_x \in A_j\}]^{mn}}{\mathring{\Gamma}, [\theta_{ij}(A_i \cup A_j \backslash x)]^{mn} \vdash_{\text{pr}} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightsquigarrow_\rho [\textbf{let}\ x = e_i\ \textbf{in}\ e_j : \theta_{ij}\mathring{\tau}_j]^{mn}}$$

$$\textsc{P-Fix}\quad \frac{\mathring{\Gamma}, [A_i]^n \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n \qquad [\theta_i = solve\,\{\mathring{\tau}_i \equiv \beta_i \to \beta_i\}]^n}{\mathring{\Gamma}, [\theta_i A_i]^n \vdash_{\text{pr}} \textbf{fix}\ e \rightsquigarrow_\rho [\textbf{fix}\ e_i : \theta_i\beta_i]^n}$$

Figure 3. (a) Constraint-based type inference rules. (b) Propagation inference rules.

to update any unknown types. (Without this, we cannot transform $(\lambda x.x)$ `"a"` to $(\lambda x.x$ `""`$)$ $(showString$ `"a"`$)$.) Substitution on a pattern application (A-App) produces a term application if substitution on the components succeeds and the application is well-typed.

Typed rewriting describes the relation between a source term and a target term via the transformation rule:

$$\mathring{\Gamma}, A \vdash p \rightsquigarrow \pi : \mathring{\tau} @ e \Rightarrow e : \mathring{\tau}$$

Given an environment $\mathring{\Gamma}$, a rule $p \rightsquigarrow \pi : \mathring{\tau}$, and a source term e, produce the target term e', target type $\mathring{\tau}'$, and assumption set A. Typed rewriting is implemented by the Rew rule:

$$\textsc{Rew}\quad \frac{\mathring{\Gamma} \vdash p @ e \Rightarrow \theta^\tau \qquad \mathring{\Gamma}, A \vdash \theta^\tau @ \pi \Rightarrow e' : \mathring{\tau} \qquad \theta = solve\,\{[\![\mathring{\tau}]\!]_{\mathcal{R}} \equiv [\![\mathring{\tau}']\!]_{\mathcal{R}}\}}{\mathring{\Gamma}, A \vdash p \rightsquigarrow \pi : \mathring{\tau}' @ e \Rightarrow e' : \theta\mathring{\tau}'}$$

In the premises, we have typed pattern matching and typed substitution application with the special substitution. The (interpreted) substitution application type functor must unify with the (interpreted) rule's type functor. We use the latter in the conclusion since it has the necessary holes. Rule 9 is an example where this is required.

In the next section, we build on typed rewriting of single terms to transform terms into multiple targets and traverse those terms with propagation.

4.3 Propagation

The algorithm for STLC type-and-transform systems takes as inputs a retract (a pair of types), a transformation rule set, a typing environment and a source program. Its result is a set of triples: a target term, its type, and an assumption set. The assumption rule set must be empty (i.e. the term must closed). To choose the best result, we can include a weight along with each output set (thus making it a quadruple); however, to avoid cluttering the presentation, we omit the weight and discuss it in the next section.

One issue we need to solve is nontermination: transformation rules might be applied ad infinitum. We avoid this by only allowing one transformation per node of the syntax tree. Alternatively stated, we alternate propagation and transformation. Propagation is the mechanism for propagating type changes (and for preventing bad transformations). Transformation is the mechanism for rewriting terms. Since multiple transformations may apply at any one node, there can be multiple possible results. In this sense, we are mapping one source to many targets.

To transform a single source term to multiple target terms, we first need a notation for expressing the one-to-many relationship: $e \rightsquigarrow_\rho [e_i]^n$. This says that the source e can be transformed with the rule set ρ into at most n targets, where a target is e_i for $i \in 1 \ldots n$[5]. We use a similar notation for a premise iterated at most n times. In $[r_j @ e_i \Rightarrow e_{ij} : \tau_{ij}]^{n|\rho|}$, we are drawing from n terms (indexed by i) and $|\rho|$ rules (indexed by j) and finding the Cartesian product (whose results are indexed by ij). Since this property may not hold true for all $n|\rho|$ combinations, we can only present the maximum cardinality in the abstract. If the ijth premise fails, the ijth conclusion naturally does not hold. Lastly, note that any index i or cardinality n in an inference is the same i or n everywhere in the rule.

Transformation with a rule set ρ is expressed as:

$$\mathring{\Gamma}, [A_i]^n \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n$$

This reads as: under an environment $\mathring{\Gamma}$, a rule set ρ transforms a source e into n targets with each target e_i having a type $\mathring{\tau}_i$ and an assumption set A_i. We define transformation inference with the following rule:

$$\textsc{Tra}\quad \frac{\mathring{\Gamma}, [A_i]^n \vdash_{\text{pr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n \qquad [r_j \in \rho]^{|\rho|} \qquad [\mathring{\Gamma}, A_{ij} \vdash r_j @ e_i \Rightarrow e_{ij} : \mathring{\tau}_{ij}]^{n|\rho|}}{\mathring{\Gamma}, [A_{ij}]^{n|\rho|} \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_{ij} : \mathring{\tau}_{ij}]^{n|\rho|}}$$

We extract each rule r_j from the rule set ρ and instantiate typed rewriting with r_j and a term e_i. If rewriting succeeds, the target triple $(A_{ij}, e_{ij}, \mathring{\tau}_{ij})$ is the result that is carried through to the conclusion.

In Tra, the rewriting source e_i comes from propagation of the original source e. Propagation is expressed with an identical judgment as transformation:

$$\mathring{\Gamma}, [A_i]^n \vdash_{\text{pr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n$$

However, the role of propagation is different. Propagation does not invoke the rule set ρ but merely passes it on to other transformations. Propagation's main purpose is to check the types of

[5] For indexes, we use i and j. For target sizes, we use m, n, and the rule set cardinality $|\rho|$

transformed targets, keeping the well-typed terms and discarding the ill-typed ones. The propagation rules in Figure 3(b) are immediately derived from the constraint-based type inference rules in Figure 3(a). In typing, the premises are naturally recursive, but in propagation, the premises refer to transformation. This mutually recursive inference is the key to termination. Termination of the type-and-transform system relies solely on termination of the propagation algorithm, which, given its syntax-directed nature, is straightforward to prove.

Since transformation inference produces multiple targets and these multiple targets are combined in Cartesian products at P-App and P-Let, propagation takes polynomial time, $O(|\rho|^{|e|})$, where $|e|$ is the size of the program e. To avoid this target program explosion, we prune all results with type failures as early and often as possible. Consequently, the use of *solve* in almost every propagation rule and the use of the the environment $\mathring{\Gamma}$ are the primary differences with constraint-based type inference in [13]

Programs are transformed using transformation inference:

$$\mathring{\Gamma}, [\varepsilon]^n \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n$$

Only if a target program has an empty assumption set (thus, having no free variables) is that program considered.

We can describe the soundness of our implementation:

Theorem 5 (Soundness) For an environment $\mathring{\Gamma}$, rule set ρ, source term e, and n target pairs $(e_i, \mathring{\tau}_i)$, transformation inference is sound with respect to the type-and-transform relation:

$$\mathring{\Gamma}, [\varepsilon]^n \vdash_{\text{tr}} e \rightsquigarrow_\rho [e_i : \mathring{\tau}_i]^n \Rightarrow [\mathring{\Gamma} \vdash e \rightsquigarrow e_i : \mathring{\tau}_i]^n$$

Transformation inference inherits soundness from [13] by deriving propagation from constraint-based type inference.

The implementation clearly does not satisfy completeness. In other words, we cannot prove this property:

$$\mathring{\Gamma} \vdash e \rightsquigarrow e' : \mathring{\tau} \Rightarrow \mathring{\Gamma}, [\varepsilon]^1 \vdash_{\text{tr}} e \rightsquigarrow_\rho [e' : \mathring{\tau}']^1$$

The transformation rules can relate an infinite number of terms, and we explicitly avoid this in propagation.

In the next section, we discuss how to find the best transformation.

4.4 Targeting the Best

When transforming programs, we generally prefer a single target, not many different possibilities There are various approaches to selecting a target, and we describe one that proved successful in our experience.

The idea is that target programs can be sorted by the presence of or absence of certain transformations. Some rules should fire as much as possible and some as little as possible. For example, consider the following transformations, each shown with two valid targets:

$$\texttt{"a"} +\!\!+ \texttt{"b"} +\!\!+ \texttt{"c"}$$

$$\rightsquigarrow (\mathit{showString}\ \texttt{"a"} \circ \mathit{showString}\ \texttt{"b"} \circ \mathit{showString}\ \texttt{"c"})\ \texttt{""} \quad (12)$$

$$\rightsquigarrow (\mathit{showString}\ \texttt{"a"} \circ \mathit{showString}\ \texttt{"b"})\ \texttt{""} +\!\!+ \texttt{"c"} \quad (13)$$

$$(\lambda x.x +\!\!+ x)\ \texttt{"a"}$$

$$\rightsquigarrow (\lambda x.x \circ x)\ (\mathit{showString}\ \texttt{"a"})\ \texttt{""} \quad (14)$$

$$\rightsquigarrow (\lambda x.\mathit{showString}\ x \circ \mathit{showString}\ x)\ \texttt{"a"}\ \texttt{""} \quad (15)$$

We prefer target 12 over 13 to maximize the use of rule 11, and we prefer target 14 over 15 to minimize the use of rules 9 and 10.

We classify rules into two groups, the *rename* group and the *repair* group. Rename rules (e.g. 11) involve renaming functions from the $\mathcal{A}$ domain to equivalent functions in the $\mathcal{R}$ domain. These rules often do not involve metavariables. They are aligned with the goal of the transformation and thus their use is encouraged. Repair rules (e.g. 9 and 10) lie at the barrier between the $\mathcal{A}$ and $\mathcal{R}$ domains. They tend to use metavariables and have the form of $m \rightsquigarrow f\ m$ or $f\ m \rightsquigarrow m$ for some function f. We want the barrier to be as "thin" as possible, so we discourage their use.

Rename rules get a positive weight, depending on how desired they are. For examples, we weight 11 with a 1. We might also add the rules:

$$\texttt{""} \rightsquigarrow \mathit{id} \quad :\iota \quad (16)$$

$$m \circ \mathit{id} \rightsquigarrow (m : \iota) : \iota \quad (17)$$

Rule 17 actually requires two other rename rules to precede it in order to fire. We weight rule 16 with a 1 and 17 with a 3, indicating that it is more desirable than both of the two other rename rules alone.

We assign every target term a weight, a pair of numbers (m,n). m is the negated sum of the weights of rename rules that have been applied, and n is the count of repair rules. Sorting lexicographically, the target program with the smallest weight is the best. It turns out that $(0,0)$ is the unaltered source. To reduce the number of programs under consideration, we can filter out all programs with a weight greater than $(0,0)$.

As an aside, one might wonder if we can optimize the weighting approach and pick the best target terms at each node instead of waiting to the end of transformation to pick the best. It only takes a few experiments to realize, however, that such a local solution will result in globally poor transformations. What appears poor locally can result in a global best and vice versa.

5. Polymorphism and Type Constructors

In our presentation of type-and-transform systems, we have leaned on the simply typed lambda calculus for our object language. This approach allowed us to focus more on our contribution and less on the mechanics of the object language type system. A slightly more complex language, the lambda calculus with let-polymorphism (a.k.a. the Damas-Hindley-Milner [6, 21] type system), would detract from the explanation with complications that are not fundamental to the problem. However, such a language is fundamentally more useful, so it is important that we discuss type-and-transform systems in its context.

The syntax for types and type functors is given below:

$$\begin{array}{llll} \tau & ::= b \mid \alpha \mid \tau \to \tau \mid c\ \tau & \mathring{\tau} & ::= b \mid \alpha \mid \mathring{\tau} \to \mathring{\tau} \mid c\ \mathring{\tau} \mid \iota_c\ \mathring{\tau} \\ \Gamma & ::= \varepsilon \mid \Gamma, x : \forall \overline{\alpha}.\tau & \mathring{\Gamma} & ::= \varepsilon \mid \mathring{\Gamma}, x : \forall \overline{\alpha}.\mathring{\tau} \end{array}$$

We extend types and type functors with the application of base type constructors c. We designate ι_c as a type constructor hole. (We might have holes for any kind of type, but the differences are not fundamental.) The typing and type functor environments map variables to type schemes, types with their variables quantified.

The changes to the inference rules of Figure 2 are standard and well-studied. We generalize the type of the definition in TT-Let, and we instantiate a variable's type scheme in TT-Var. The correlation between the type system and the type-and-transform system for let-polymorphism is preserved.

The conversion to type functors becomes somewhat more complicated, now that we deal with variables (*mgtf* is applied to instantiated schemes):

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{mgtf}\ \tau_1\ \tau_2 = \textbf{let}\ (\theta, \mathring{\tau}) = \mathit{mgtf}'\ \tau_1\ \tau_2\ \textbf{in}\ \theta\mathring{\tau}} \\ \mathit{mgtf}'\ b_1 \quad b_2 & \mid b_1 \equiv b_2 & = (\mathit{id}, b_1) \\ \mathit{mgtf}'\ \alpha \quad \tau & & = ([\alpha \mapsto \tau], \tau) \\ \mathit{mgtf}'\ \tau \quad \alpha & & = ([\alpha \mapsto \tau], \tau) \\ \mathit{mgtf}'\ (\tau_1 \to \tau_2)\ (\tau_3 \to \tau_4) & & = \textbf{let}\ (\theta_1, \mathring{\tau}_1) = \mathit{mgtf}'\ \tau_1\ \tau_3 \end{array}$$

$$
\begin{array}{llll}
 & & & (\theta_2, \mathring{\tau}_2) = mgtf'\ \tau_2\ \tau_4 \\
 & & & \textbf{in}\ (\theta_1 \circ \theta_2, \mathring{\tau}_1 \to \mathring{\tau}_2) \\
mgtf'\ (c_1\ \tau_1) & (c_2\ \tau_2) & \mid c_1 \equiv c_2 & = \textbf{let}\ (\theta, \mathring{\tau}) = mgtf'\ \tau_1\ \tau_2 \\
 & & & \textbf{in}\ (\theta, c_1\ \mathring{\tau}) \\
mgtf'\ (\mathcal{A}_c\ \tau_1) & (\mathcal{R}_c\ \tau_2) & & = \textbf{let}\ (\theta, \mathring{\tau}) = mgtf'\ \tau_1\ \tau_2 \\
 & & & \textbf{in}\ (\theta, \iota_c\ \mathring{\tau})
\end{array}
$$

Note that $\mathcal{A}_c$ and $\mathcal{R}_c$ are type constructors. The interpretation of a type functor into a type is still rather simple, except that the argument is a type constructor c' instead of a type:

$$
\begin{array}{ll}
[\![b]\!]_{c'} & = b \\
[\![\alpha]\!]_{c'} & = \alpha \\
[\![\mathring{\tau}_1 \to \mathring{\tau}_2]\!]_{c'} & = [\![\mathring{\tau}_1]\!]_{c'} \to [\![\mathring{\tau}_1]\!]_{c'} \\
[\![c\ \mathring{\tau}]\!]_{c'} & = c\ [\![\mathring{\tau}]\!]_{c'} \\
[\![\iota_c\ \mathring{\tau}]\!]_{c'} & = c'\ [\![\mathring{\tau}]\!]_{c'}
\end{array}
$$

The proof of correctness is very similar to Section 3.6 with one key difference: $dimap_{\mathring{\tau}}$ is indexed on type applications:

$$
\begin{array}{ll}
dimap_{\mathring{\tau}} : (c_1\ a \to c_2\ a) \to (c_2\ a \to c_1\ a) \to [\![\mathring{\tau}]\!]_{c_2} \to [\![\mathring{\tau}]\!]_{c_1} & \\
dimap_b & f\ g = id \\
dimap_\alpha & f\ g = id \\
dimap_{\mathring{\tau}_1 \to \mathring{\tau}_2} & f\ g = \lambda x.dimap_{\mathring{\tau}_2}\ f\ g \circ x \circ dimap_{\mathring{\tau}_1}\ g\ f \\
dimap_{c\ \mathring{\tau}} & f\ g = dimap'_c\ (dimap_{\mathring{\tau}}\ g\ f)\ (dimap_{\mathring{\tau}}\ f\ g) \\
dimap_{\iota_c\ \mathring{\tau}} & f\ g = dimap'_{c_1}\ (dimap_{\mathring{\tau}}\ g\ f)\ (dimap_{\mathring{\tau}}\ f\ g) \circ g
\end{array}
$$

The auxiliary function $dimap'_c$ must be defined for each type constructor c. If the type constructor is a covariant functor F, then $dimap'_F\ f\ g \equiv fmap_F\ g$. For example, $dimap'_{[\,]}\ f\ g \equiv map\ g$.

The implementation is very similar to Section 4; however, constraint-based type inference for let-polymorphism requires a bit more work. Primarily, the constraints are more extensive. In STLC, we only needed equality, $\tau_1 \equiv \tau_2$. With let-polymorphism, we also need a constraint indicating that a type is an explicit instance of a type scheme, $\tau_1 \preceq \forall \overline{\alpha}.\tau_2$ and a constraint indicating that one type is an implicit instance of another, $\tau_1 \preccurlyeq_M \tau_2$. An implicit instance constraint reads as: τ_1 should be an instance of the type scheme obtained by generalizing τ_2 without quantifying the type variables that are free in M. The set of monomorphic type variables M is extended downwards through inference rules by adding a fresh variable for the parameter type at abstractions. The bottom-up traversal cannot always solve implicit constraints immediately, so a constraint set C must be passed upwards (as with the assumption set). We refer the reader to [13] for the details on polymorphic constraint-based type inference.

Our constraint-based typing judgment for the polymorphic lambda calculus is:

$$\Gamma, M, A, C \vdash e : \tau$$

Naturally, this means the propagation judgment is:

$$\mathring{\Gamma}, M, [A_i]^n, [C_i]^n \vdash_{\text{pr}} e \rightsquigarrow_{\rho} [e_i : \mathring{\tau}_i]^n$$

For each target term e_i, we have an associated type $\mathring{\tau}_i$, assumption set A_i and constraint set C_i. The environment $\mathring{\Gamma}$ and monomorphic type variables M are passed downwards and hold for all results at a given node. We inherit soundness for transformation inference in let-polymorphism from the constraint-based type inference.

In the next section, we describe several example transformations implemented in the language described here.

6. Examples, Revisited

Now that we have explained type-and-transform systems, let us revisit the examples presented in Section 2 and how to implement them.

6.1 Hughes' Lists

In Section 3, we used the running example of strings to demonstrate a type-and-transform system. This is, of course, a specialization of Hughes' lists. In the more general case, the retract is $[a] \triangleleft DL\ a$ where $DL\ a = [a] \to [a]$ is the type also referred to as difference lists. We might directly use the functions available to us such as as we did in Section 4.2, but we prefer an abstract API such as that of the Haskell library `dlist`[6]. Using a similar API (adapted to distinguish names), we can implement the basic transformation rules as follows:

$$
\begin{array}{ll}
m \rightsquigarrow fromList\ (m : [a]) & : \iota\ a \\
m \rightsquigarrow toList\ (m : \iota\ a) & : [a] \\
(+\!\!+) \rightsquigarrow (\diamond) & : \iota\ a \to \iota\ a \to \iota\ a
\end{array}
$$

The functions are defined as:

$$
\begin{array}{ll}
fromList & = (+\!\!+) \\
toList & = \lambda f.f\ [\,] \\
(\diamond) & = (\circ)
\end{array}
$$

The last is also the binary *Monoid* operator on *DL*.

There are plenty of other rules that would make this transformation more useful. Ideally, all operations are mapped from one domain (lists) to the other (dlists). Here are a few more rules:

$$
\begin{array}{ll}
[\,] \rightsquigarrow [\,]_{DL} & : \iota\ a \\
foldr \rightsquigarrow foldr_{DL} & : (a \to b \to b) \to b \to \iota\ a \to b
\end{array}
\tag{18}
$$

With 18 and the list elimination function $list : b \to (a \to [a] \to b) \to [a] \to b$, we can perform are few more interesting transformations :

$$
\begin{array}{l}
\textbf{let}\ reverse = \textbf{fix}\ (\lambda r.list\ [\,]\ (\lambda x.\lambda xs.r\ xs +\!\!+ [x])) \\
\textbf{in}\ \ reverse\ [1,2] \\
\rightsquigarrow \textbf{let}\ reverse = \textbf{fix}\ (\lambda r.list\ [\,]_{DL}\ (\lambda x.\lambda xs.r\ xs \diamond fromList\ [x])) \\
\quad \textbf{in}\ \ toList\ (reverse\ [1,2]) \\
\textbf{let}\ concat = \textbf{fix}\ (\lambda r.list\ [\,]\ (\lambda x.\lambda xs.x +\!\!+ r\ xs)) \\
\textbf{in}\ \ concat\ [[0]] \\
\rightsquigarrow \textbf{let}\ concat = \textbf{fix}\ (\lambda r.list\ [\,]_{DL}\ (\lambda x.\lambda xs.fromList\ x \diamond r\ xs)) \\
\quad \textbf{in}\ \ toList\ (concat\ [[0]])
\end{array}
$$

These examples (using Haskell list notation for brevity) are similar to the worker/wrapper transformation [11].

6.2 Streams

The transformation rules for streams are straightforward:

$$
\begin{array}{ll}
m \rightsquigarrow stream\ (m : [a]) & : \iota\ a \\
m \rightsquigarrow unstream\ (m : \iota\ a) & : [a] \\
map \rightsquigarrow map_s & : (a \to b) \to \iota\ a \to \iota\ b \\
filter \rightsquigarrow filter_s & : (a \to Bool) \to \iota\ a \to \iota\ a
\end{array}
$$

Again, a "production" transformation should exhaustively cover the stream library functions.

As an aside, we note that the implementations of the *Stream* datatype typically involve existential quantification. Here, we assume *Stream* is an abstract type constructor, so it does not affect our example. It does appear in the correctness proof, but it does not invalidate the streams type-and-transform system.

7. Related Work

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

[6] `http://hackage.haskell.org/package/dlist`

Term rewriting [1] is a technique that has been extensively applied to program transformation. Stratego [24] is a well-known language and toolset for program transformation using rewriting. It is representative of strategy languages in which basically any transformation can be specified. One of our early approaches involved term rewriting; however, we found that we could not define a type-safe strategy that involved type changes coming from arbitrary directions (e.g. from the function or the argument or both in application). The combination of flexibility and safety of type-and-transform systems is nontrivial in term rewriting systems. Of course, our typed rewriting (Section 4.2) is inspired by term rewriting.

Another common technique for program transformation is refactoring [9]. HaRe [18] is a Haskell refactoring tool that supports a number of of automatic refactorings. The kind of pervasive, type-changing transformations we want are not currently provided by HaRe, and it is not clear how to modify HaRe to do such transformations (without re-implementing the type-and-transform infrastructure). Additionally, HaRe and other refactoring tools require transformations to be expressed with operations on the abstract syntax tree, which can complicate the definition of transformations. Our transformation rules are simple and defined in the concrete object syntax. Type-and-transform systems do not support many of the refactorings of HaRe, most prominently the ones involving scope updates.

One might see our approach as a sort of type-and-effect system [10] if one views the transformation as a side effect of an extended type system. However, that analogy is stretched rather thin since we do not modify how the type system works. Instead, we derive from the type system a new relation between programs and their types.

In the type-and-transform relation, transformation and propagation rules are distinguishable only by their purpose. In the implementation, we describe how to define transformation rules and integrate them into the propagation system. Heeren, Hage, and Swierstra [14] describe a similar idea for improving the quality of type error messages, particularly for embedded domain-specific languages. They define specialized type rules and verify that such rules do not change the type system. We also verify that transformation rules do not change the type-and-transform system, which means they should neither introduce type errors nor change the semantics of a source program.

Cunha and Visser [5] developed a strongly typed rewriting system for coupled software transformation. They calculate type-safe, type-changing transformations with a strategic rewrite system. They use a point-free program calculus and a type representation for their embedded object language and transformations. It may be possible to implement our examples with their approach, but it would be limited to the point-free calculus. In contrast, our work uses the syntax and types of a typed object language, including locally bound variables. We also believe our transformation rules are simpler to design. Further work is necessary to determine what transformations can be implemented with each system and how large the overlap is.

Erwig and Den [8] define a program update calculus for defining well-typed updates. The capabilities of the update calculus include simple rewrites, scope changes, composing updates, alternating updates, and recursive updates. Their type-change system ensures that an update preserves well-typedness even for type-changing transformations. However, the update calculus does not encode knowledge of type changes or of previous rewrites (in a first-class sense). Beyond basic rewrites, the update calculus is more scope-driven than type-driven. We were unable to specify any of our examples in the update calculus. In addition, we found the update calculus type system to be complex, necessarily so of course, due to the nature of preserving the type correctness with updates. On the other hand, a type-and-transform system is not much more complex than the type system from which it is derived. In an email [7], Erwig stated that they have moved beyond the idea of update programs that guarantee type correctness because it affords a class of updates too small to be relevant in practice. We hope to increase the the size of that class with type-and-transform systems.

Coercions [17, 19, 23] extend the polymorphic type system wth the notion that data of one type can be coerced to a subsuming type by inserting functions between those types. The subtyping is similar to our retract, and the coercions are related to our transformation rules. Coercion insertion is limited to only certain nodes in the syntax (e.g. function application) while type-and-transform systems can rewrite any subterm. Coercions seem to require an extended term language. In our case, we extend the object language types to type functors while leaving the object language untouched. Also, while coercions serve a similar purpose to transformation rules, the latter are more general.

In the next section, we discuss some future possibilities for our work.

8. Future Work

We are currently expanding the system's support for language features. Being Haskell programmers ourselves, we wish to support all of Haskell. With Haskell support, we can explore how well the system works on large programs. Toward this end, we will investigate a tool, such as HLint or HaRe, that can transform real-world programs. Programmers can define their own transformations on their programs.

We believe type-and-transform systems are applicable to code generation in compilers. We have done successful preliminary experiments with System F, and we think supporting GHC's System FC is possible.

Supporting real languages allows for the study of more examples. We can look at how useful the different type-and-transform systems are on real programs. We can, for example, compare our streams transformation to stream fusion. There are many other examples we can try: **newtype** introduction, monad introduction, and datatype-generic programming (DGP).

We are currently looking into how we can transform a program to use a DGP library. For example, we transform a program from lists to a generic representation type. After transformation, the programmer can easily refactor the program to work with trees or another datatype.

We are considering variations to the algorithm to improve efficiency. In practice, the worst case does not occur, but the time can still be improved.

We plan to explore how transformations can be combined. Can we sequence and interleave transformations safely? How do we deal with more types?

9. Conclusion

We have described type-and-transform systems, an approach to type-safe, semantics-preserving, automatic program transformation. A system is specified by the type-and-transform relation, which is the union of propagation rules and transformation rules. A system is implemented by the propagation algorithm and typed rewriting with a simple syntax for defining transformation rules. We have shown that type-and-transform systems preserve types in the simply typed and polymorphic lambda calculi. We have also shown that our implementation is sound and naturally not complete.

We have just touched the surface of type-and-transform systems. We plan on developing the theory to explore more expres-

siveness and the implementation to validate our ideas in practical settings.

Acknowledgments

We greatly appreciate discussion with our colleagues, including Joeri van Eekelen, Jurriaan Hage, Bastiaan Heeren, Stefan Holdermans, Patrik Jansson, Jan Rochel, and Doaitse Swierstra, as well as with members of the Dutch Haskell Users Group. We are also grateful to the anonymous reviewers for their helpful suggestions.

References

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

[2] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *ICFP 2007*, 2007.

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

[4] R. Cox. Introducing Gofix, April 2011. `http://blog.golang.org/2011/04/introducing-gofix.html`.

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

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

[7] M. Erwig. Personal email. September 2010.

[8] M. Erwig and D. Ren. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67:199–222, 2007.

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

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

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

[12] T. Harper. Stream Fusion on Haskell Unicode Strings. In M. T. Morazán and S.-B. Scholz, editors, *IFL 2009*, pages 125–140. Springer, 2011.

[13] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, September 2005.

[14] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the Type Inference Process. In *ICFP 2003*, pages 3–13. ACM, 2003.

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

[16] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *Haskell 2001*. ACM, 2001.

[17] R. Kießling and Z. Luo. Coercions in Hindley-Milner Systems. volume 3085 of *Lecture Notes in Computer Science*, chapter 17, pages 259–275. Springer, Berlin, Heidelberg, 2004.

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

[19] Z. Luo. Coercions in a polymorphic type system. *Math. Struct. in Comp. Science*, 18:729–751, 2008.

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

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

[22] N. Mitchell. HLint Manual, February 2010. `http://community.haskell.org/~ndm/darcs/hlint/hlint.htm`.

[23] N. Swamy, M. Hicks, and G. M. Bierman. A Theory of Typed Coercions and its Applications. In *ICFP 2009*, pages 329–340. ACM, August 2009.

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

A. Proof of Correctness

This appendix contains the proofs for Theorem 3 and Lemma 1.

A.1 Difunctors Encode Transformation

In Theorem 3, we relate the source and target terms of each type-and-transform inference rule via $dimap_{\mathring{\tau}}$. The proof of TT-App is given in Section 3.4. The proofs of TT-Let and TT-Fix can be implemented using TT-Lam and TT-App. In this section, we prove TT-Var, TT-Lam, TT-Rep, TT-Abs, and TT-Comp.

$$\text{TT-Var}\quad \frac{x:\mathring{\tau}\in\mathring{\Gamma}}{\mathring{\Gamma}\vdash x\rightsquigarrow x:\mathring{\tau}}$$

Proof of: $dimap_{\mathring{\tau}}\ rep\ abs\ \theta_{\mathring{\Gamma}}x \equiv x$

$dimap_{\mathring{\tau}}\ rep\ abs\ \theta_{\mathring{\Gamma}}x$
≡ { Premise: $x:\mathring{\tau}\in\mathring{\Gamma}$ so $\theta_{\mathring{\Gamma}} = \theta_{\mathring{\Gamma},} \circ [x\mapsto dimap_{\mathring{\tau}}\ abs\ rep\ x]$ }
$dimap_{\mathring{\tau}}\ rep\ abs\ [x\mapsto dimap_{\mathring{\tau}}\ abs\ rep\ x]x$
≡ { Substitution }
$dimap_{\mathring{\tau}}\ rep\ abs\ (dimap_{\mathring{\tau}}\ abs\ rep\ x)$
≡ { Difunctor composition }
$dimap_{\mathring{\tau}}\ (abs\circ rep)\ (abs\circ rep)\ x$
≡ { Retract }
$dimap_{\mathring{\tau}}\ id\ id\ x$
≡ { Difunctor identity }
x

$$\text{TT-Lam}\quad \frac{\mathring{\Gamma},x:\mathring{\tau}_1\vdash e\rightsquigarrow e':\mathring{\tau}_2}{\mathring{\Gamma}\vdash \lambda x.e\rightsquigarrow \lambda x.e':\mathring{\tau}_1\to\mathring{\tau}_2}$$

Proof of: $dimap_{\mathring{\tau}_1\to\mathring{\tau}_2}\ rep\ abs\ \theta_{\mathring{\Gamma}}(\lambda x.e') \equiv \lambda x.e$

$dimap_{\mathring{\tau}_1\to\mathring{\tau}_2}\ rep\ abs\ \theta_{\mathring{\Gamma}}(\lambda x.e')$
≡ { Commute substitution: $\theta_{\mathring{\Gamma}} = \theta_{\mathring{\Gamma},} \circ [x\mapsto dimap_{\mathring{\tau}}\ abs\ rep\ x]$ }
$dimap_{\mathring{\tau}_1\to\mathring{\tau}_2}\ rep\ abs\ (\lambda x.\theta_{\mathring{\Gamma}'}e')$
≡ { Definition of $dimap_{\mathring{\tau}_1\to\mathring{\tau}_2}$ }
$dimap_{\mathring{\tau}_2}\ rep\ abs\circ(\lambda x.\theta_{\mathring{\Gamma}'}e')\circ dimap_{\mathring{\tau}_1}\ abs\ rep$
≡ { η-expansion }
$\lambda x.(dimap_{\mathring{\tau}_2}\ rep\ abs\circ(\lambda x.\theta_{\mathring{\Gamma}'}e')\circ dimap_{\mathring{\tau}_1}\ abs\ rep)\ x$
≡ { β-reduction }
$\lambda x.dimap_{\mathring{\tau}_2}\ rep\ abs\ (\theta_{\mathring{\Gamma}'}\circ[x\mapsto dimap_{\mathring{\tau}_1}\ abs\ rep\ x])e'$
≡ { Definition of $\theta_{\mathring{\Gamma}}$ }
$\lambda x.dimap_{\mathring{\tau}_2}\ rep\ abs\ \theta_{\mathring{\Gamma}}e'$
≡ { Shadowing }
$\lambda x.dimap_{\mathring{\tau}_2}\ rep\ abs\ (\theta_{\mathring{\Gamma},x:\mathring{\tau}_1}e')$
≡ { Premise }
$\lambda x.e$

$$\text{TT-Rep}\quad \frac{\mathring{\Gamma}\vdash e\rightsquigarrow e':\mathcal{A}\qquad rep:\mathcal{A}\to\mathcal{R}\in\mathring{\Gamma}}{\mathring{\Gamma}\vdash e\rightsquigarrow rep\ e':\iota}$$

Proof of: $dimap_{\iota}\ rep\ abs\ \theta_{\mathring{\Gamma}}(rep\ e') \equiv e$

$dimap_{\iota}\ rep\ abs\ \theta_{\mathring{\Gamma}}(rep\ e')$
≡ { Definition of $dimap_{\iota}$ }
$abs\ \theta_{\mathring{\Gamma}}(rep\ e')$
≡ { Commute substitution }
$\theta_{\mathring{\Gamma}}(abs\ (rep\ e'))$
≡ { Retract }

$\theta_{\mathring{\Gamma}} e'$
$\equiv$ { Introduce *id* }
$id\ \theta_{\mathring{\Gamma}} e'$
$\equiv$ { Definition of $dimap_{\mathcal{A}}$ }
$dimap_{\mathcal{A}}\ rep\ abs\ (\theta_{\mathring{\Gamma}} e')$
$\equiv$ { Premise }
e

$$\text{TT-Abs}\quad \frac{\mathring{\Gamma} \vdash e \rightsquigarrow e' : \iota \qquad abs : \mathcal{R} \to \mathcal{A} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash e \rightsquigarrow abs\ e' : \mathcal{A}}$$

Proof of: $dimap_{\mathcal{A}}\ rep\ abs\ \theta_{\mathring{\Gamma}}(abs\ e') \equiv e$

$dimap_{\mathcal{A}}\ rep\ abs\ \theta_{\mathring{\Gamma}}(abs\ e')$
$\equiv$ { Definition of $dimap_{\mathcal{A}}$ }
$\theta_{\mathring{\Gamma}}(abs\ e')$
$\equiv$ { Commute substitution }
$abs\ \theta_{\mathring{\Gamma}} e'$
$\equiv$ { Definition of $dimap_{\iota}$ }
$dimap_{\iota}\ rep\ abs\ \theta_{\mathring{\Gamma}} e'$
$\equiv$ { Premise }
e

$$\text{TT-Comp}\quad \frac{(+\!+) : \mathcal{A} \to \mathcal{A} \to \mathcal{A} \in \mathring{\Gamma} \qquad (\circ) : \mathcal{R} \to \mathcal{R} \to \mathcal{R} \in \mathring{\Gamma}}{\mathring{\Gamma} \vdash (+\!+) \rightsquigarrow (\circ) : \iota \to \iota \to \iota}$$

Proof of: $dimap_{\iota\to\iota\to\iota}\ rep\ abs\ \theta_{\mathring{\Gamma}}(\circ) \equiv (+\!+)$

$dimap_{\iota\to\iota\to\iota}\ rep\ abs\ \theta_{\mathring{\Gamma}}(\circ)$
$\equiv$ { Definition of $\theta_{\mathring{\Gamma}}$ and substitution }
$dimap_{\iota\to\iota\to\iota}\ rep\ abs\ (\circ)$
$\equiv$ { Definition of $dimap_{\iota\to\iota\to\iota}$ and η-expansion }
$\lambda x.\lambda y.abs\ (rep\ x \circ rep\ y)$
$\equiv$ { Definitions and η-reduction }
$(+\!+)$

A.2 Normal Form of ι

In Section 3.6, we also established normal forms for the type functor ι in Lemma 1. For this, we need to show that e' in $\mathring{\Gamma} \vdash e \rightsquigarrow e' : \iota$ can be evaluated to any normal form established by the transformation rules. These include all rules with ι in the result position of the type functor: TT-Rep and TT-Comp. In the former, *rep e* is the target term, so the proof is trivial. In the latter, we need to prove that $e'_1 \circ e'_2$ can be reduced to *rep e*:

$e'_1 \circ e'_2$
$\equiv$ { Induction and η-expansion }
$\lambda x.rep\ e_1\ (rep\ e_2\ x)$
$\equiv$ { Definitions }
$\lambda x.e_1 +\!+ (e_2 +\!+ x)$
$\equiv$ { Associativity of $+\!+$ }
$\lambda x.(e_1 +\!+ e_2) +\!+ x$
$\equiv$ { Definition of *rep* and η-reduction }
$rep\ (e_1 +\!+ e_2)$

As an aside, we note that *rep* is a homomorphism on the algebras of the source and target transformation rules:

$$rep\ (x +\!+ y) \equiv rep\ x \circ rep\ y$$

This property is more general than the above and would also be sufficient for the proof.