# Correctness of a Higher-Order Removal Transformation through a Relational Reasoning*

Susumu Nishimura

Department of Mathematics, Faculty of Science, Kyoto University
susumu@math.kyoto-u.ac.jp

**Abstract.** The syntactic logical relations developed by Pitts are applied to show the correctness of a higher-order removal program transformation algorithm. The convenient proof method that resorts to the induction on the structure of programs does not apply because of the circular references to be introduced by the transformation. Using a variant of the syntactic logical relations, in which every pair of the transformation source and target are related, one can break the circularity and make an inductive proof go through. This demonstrates that the syntactic logical relations provide a basis of another general proof method for showing the correctness of program transformations.

## 1 Introduction

The aim of program transformation is to improve the run-time efficiency, keeping the semantics of the source program. The correctness proof on a program transformation algorithm is usually done by an inductive argument on the program structure, where each induction step is proved by an equational reasoning on the general syntactic pattern of the corresponding local transformation site. This inductive proof principle applies to a large class of program transformation algorithms, but there are certain ones that require a stronger one.

In this paper, we demonstrate that the *syntactic logical relations* developed by Pitts [14] can serve as another powerful device for proving the correctness of program transformations. The logical relations are a type-indexed family of relations $\{\Delta_\tau\}_\tau$ between program expressions, where each $\Delta_\tau$ is inductively defined on the structure of type $\tau$ [9]. Pitts developed an operational technique to construct logical relations over a syntactic domain and called them 'syntactic' logical relations. He also showed that the syntactic logical relations coincide with the observational equivalence relation, *i. e.*, $(M, M') \in \Delta_\tau$ for some $\tau$ iff $M$ and $M'$ show the same evaluation behavior when they are embedded in any context of a larger program.

The syntactic logical relations provide a firm basis for arguing the observational equivalence between the transformation source and target. Given a transformation algorithm, we alter the original definition of the syntactic logical relations so that they relate every pair of a source program and its transformation

* Most of this work was done while the author was at Research Institute for Mathematical Sciences, Kyoto University.

result. That is, we construct the family of relations satisfying

$$(M, \mathcal{W}[\![M]\!]) \in \Delta_\tau,$$

where $M$ is an arbitrary source expression of type $\tau$ and $\mathcal{W}[\![M]\!]$ stands for the transformation result. Then the correctness of the transformation (*i. e.*, the source program and its transformation result behave similarly) follows from the coincidence between the logical relations and the observational equivalence.

There are a few attempts to apply a syntactically formulated relational framework to the correctness proof of program transformation. An example is a boxing transformation on polymorphic values [7, 8]. However, the language considered does not have fixpoint recursion. The proof technique in the present paper has a more powerful ability to reason about the observational equivalence in the presence of fixpoint operators (and hence it is sensitive to termination behavior too). Johann [6] exploited the parametricity result induced from the syntactic logical relations to give a formal justification on the shortcut fusion transformation [4], which is a popular transformation algorithm for removing intermediate data structures that are produced between producer/consumer function pairs. The present paper demonstrates that not only the parametricity but also the framework of the syntactic logical relations itself can be a formal basis for arguing the correctness of program transformations.

We show how the syntactic logical relations can effectively constitute a formal correctness proof, through a non-trivial example of program transformation, namely, a *higher-order removal transformation*, which was introduced by the author as a transformation step for removing higher-order functions in a particular class of programs [11]. The major difficulty in giving a correctness proof for this transformation is raised by a *circularity* to be introduced into the structure of the target program. The presence of circularity makes it difficult to show the total correctness of transformation (*i. e.*, the semantics of the whole program is preserved), since there is no base case to induct on the program structure. Using logical relations, however, we can break the circularity and make the induction go through.

Though we employed Pitts' syntactic logical relations, it would be also possible to give the correctness proof using logical relations that are, usually done as such, defined over semantic domains, as Nielsen [10] did in the correctness proof of the general higher-order removal algorithm (*a.k.a.* defunctionalization [15]). However, we prefer Pitts' syntactic approach, since we can avoid technical complications that arise from the domain theoretic approach.

The rest of the paper is organized as follows. Section 2 informally presents the higher-order removal transformation algorithm mentioned above. Section 3 introduces a relational framework for reasoning about observational equivalence of programs based on the syntactic logical relations. Section 4 gives a formal correctness proof of the higher-order removal transformation. Finally Section 5 concludes the paper with discussions on future research topics.

## 2 The Higher-Order Removal Transformation by Example

*Deforestation* [19] is a program transformation which removes intermediate data structures that are produced by a function and are immediately consumed by another function. Though notable refinements and improvements on the deforestation algorithm have been proposed [4, 16, 1, 17], it had been long undiscovered how to deforest functions that are defined by means of *accumulating parameters*, which are function arguments passed around throughout a series of recursive calls, as popularly seen in the tail recursive definition of list reverse function. A solution, which is based on the framework of macro tree transducers [2], to this problem was recently proposed by Voigtländer [18]. The author independently developed another one, which was inspired from the composition method for attribute grammars [3].

The program transformation considered in this paper is a higher-order removal transformation, which is a vital step in the latter deforestation algorithm. Let us explain the deforestation process by applying it to an example given in Figure 1. We use Haskell like syntax [5] extended with record syntax: (``$l_1$=$M_1$ , ... , $l_n$=$M_n$``) expresses a record with labels $l_1, \ldots, l_n$ ($n \geq 0$) associated with expressions $M_1, \ldots, M_n$, *resp.*, `#`$l\,M$ selects the value associated with the field $l$ of record $M$. (`` $l_1$::$\tau_1$ , ... , $l_n$::$\tau_n$ ``) represents the type of a record whose label $l_i$ is each associated with a value of type $\tau_i$.

The source program to which deforestation process is applied is the function `revflat` given in Figure 1 (a). It is a composition of a producer function `flat` and a consumer function `rev`, which compute the preorder traversal list of leaves of a binary tree and the reversal of list elements, respectively. Though the definition may look verbose, the intention is made clear by removing all the record labels. For example, the definition of `flat` is equivalent to the following program:

```
flat :: T a → [a] → [a]
flat (Node(t,t')) x0 = flat t (flat t' x0)
flat (Leaf  a)    x0 = a:x0
```

We put distinct record labels as the "tags" to uniquely identify the accumulating parameter and the computation result of each different function; `h` and `l` are used for tagging the accumulating parameter and the computation result of the function `flat`, *resp.*; similarly, `k` and `j` for those of the function `rev`. The named record labels are more useful for the purpose of unique identification than the unlabeled tuples.

Figure 2 illustrates the whole deforestation process, which comprises of four subtransformation steps. First, it applies shortcut fusion [4], which is a standard deforestation technique, in order to remove intermediate data structures and to obtain a single recursive function definition. The result is, however, a higher-order function, and we would like to remove this higher-orderedness. For this, we further apply two transformations, which perform type conversions on the higher-order function. They are intended to translate the program into a form

```
data T a = Node (T a,T a) | Leaf a

flat :: T a → (h::[a]) → (l::[a])
flat (Node(t,t')) x0 = (l=#l (flat t (h=#l (flat t' (h=#h x0)))))
flat (Leaf a)     x0 = (l=a:(#h x0))

rev :: [a] → (k::[a]) → (j::[a])
rev (a:t) x0 = (j=#j (rev t (k=a:(#k x0))))
rev []    x0 = (j=#k x0)

revflat :: T a→ [a]
revflat x = #j (rev (#l (flat x (h=[]))) (k=[]))
```

Figure 1(a): Source program

```
rf :: T a → ((hk::[a]) → (hj::[a])) → (lk::[a]) → (lj::[a])
rf (Node(t,t')) x0 = let x2 = rf t' (\y2 -> (hj=#hj (x0 (hk=#hk y2)))) in
                     let x1 = rf t  (\y1 -> (hj=#lj (x2 (lk=#hk y1)))) in
                     \y0 -> (lj=#lj (x1 (lk=#lk y0)))
rf (Leaf a)     x0 = \y0 -> (lj=#hj (x0 (hk=a:(#lk y0))))

revflat :: T a → [a]
revflat x = #lj (rf x (\y -> (hj=#hk y)) (lk=[]))
```

Figure 1(b): Intermediate transformation result

```
rf' :: T a → (hj ::[a], lk::[a]) → (hk::[a], lj ::[a])
rf' (Node(t,t')) x0 = let x2 = rf' t' (hj=#hj x0, lk=#hk x1)
                          x1 = rf' t  (hj=#lj x2, lk=#lk x0)
                      in (lj=#lj x1, hk=#hk x2)
rf' (Leaf a)     x0 = (lj=#hj x0, hk=a:(#lk x0))

revflat :: T a → [a]
revflat x = #lj (rf' x (hj=#hk (rf' x (hj=Ω, lk=[])), lk=[]))
```

Figure 1(c): The result of higher-order removal transformation

**Fig. 1.** Transformation example

```
flat :: T a → (h::[a]) → (l::[a]),   rev :: [a] → (k::[a]) → (j::[a])
                                  ⇓   shortcut fusion
rf :: a T → (h::(k::[a])→(j::[a])) → (l::(k::[a])→(j::[a]))
                                  ⇓
rf :: a T → ((h::(k::[a]))→(h::(j::[a]))) → ((l::(k::[a]))→(l::(j::[a])))
                                  ⇓
rf :: a T → ((hk::[a])→(hj::[a])) → ((lk::[a])→(lj::[a]))
                                  ⇓   higher-order removal
rf' :: a T → (hj::[a],lk::[a]) → (hk::[a],lj::[a])
```

**Fig. 2.** The whole deforestation process

that is suitable as the input to the final higher-order removal transformation step. The intermediate result by these transformations is given in Figure 1(b). (Since the main concern of this paper is in the application of the syntactic logical relations to the correctness proof of a program transformation, we do not go any further on the details of these preceding transformations. The transformation rules for them can be found in [11].)

We notice that, in Figure 1(b), the result of every recursive call of `rf` $t\ \cdots$ (where $t$ is either the variable `t` or `t'`) is bound by a let construct to a variable $\mathtt{x}_i$ ($i > 0$) and also that every let bound variable $\mathtt{x}_i$ appears, per each constructor case of the definition of `rf`, once and only once in the form `#lj (`$\mathtt{x}_i$ `(lk=`$M$`))`; We write $M_{\mathtt{x}_i}$ for such expression $M$. The occurrence of the formal function argument $\mathtt{x}_0$ of `rf` is also linear and it appears in the form `#hj (`$\mathtt{x}_0$ `(hk=`$M_{\mathtt{x}_0}$`))`.

The result of applying the higher-order removal transformation is given in Figure 1(c). At the type level, this transformation is understood as a type conversion

$$\begin{array}{ll}\text{from} & \sigma_1 = \mathtt{((hk::[a])} \to \mathtt{(hj::[a]))} \to \mathtt{((lk::[a])} \to \mathtt{(lj::[a]))} \\ \text{to} & \sigma_2 = \mathtt{(hj::[a],lk::[a])} \to \mathtt{(hk::[a],lj::[a])}.\end{array}$$

Notice that `hj` and `lk`, which are the labels of record types occurring at negative positions in the higher-order function type $\sigma_1$, constitute the argument type `(hj::[a],lk::[a])` of the first-order function type $\sigma_2$ and also that `hk` and `lj`, which are the labels of records occurring at positive positions in $\sigma_1$, constitute the result type `(hk::[a],lj::[a])` of $\sigma_2$.

$$\begin{array}{lcll}
\mathcal{W}'[\![H :: T]\!] & = & \mathcal{W}'[\![H]\!] :: \mathcal{W}'[\![T]\!] & \\
\mathcal{W}'[\![\#lk\ y_0]\!] & = & \#lk\ x_0 & \\
\mathcal{W}'[\![\#hk\ y_i]\!] & = & \#hk\ x_i & (1 \le i \le n) \\
\mathcal{W}'[\![\#hj\ (x_0\ (hk = M_{x_0}))]\!] & = & \#hj\ x_0 & \\
\mathcal{W}'[\![\#lj\ (x_i\ (lk = M_{x_i}))]\!] & = & \#lj\ x_i & (1 \le i \le n) \\
\mathcal{W}'[\![M]\!] & = & M & (\text{otherwise})
\end{array}$$

**Fig. 3.** Transformation rules for higher-order removal

The higher-order removal transformation on the function `rf` that respects the above type conversion is formulated as follows. (The transformation on `revflat` is a bit technical, and is discussed later in Section 4.2.) Every `let` binding site $\mathtt{x}_i$ `= rf` $t$ `(\y`$_i$ `-> (hj=`$N_i$`))` is rewritten to

$$\mathtt{x}_i \texttt{ = rf' } t \texttt{ (hj=}\mathcal{W}'[\![N_i]\!]\texttt{, lk=}\mathcal{W}'[\![M_{\mathtt{x}_i}]\!]\texttt{)} \tag{1}$$

where $\mathcal{W}'[\![M]\!]$ denotes a transformation on $M$, which is inductively defined on the structure of $M$ by the rules given in Figure 3. (We only consider a special class of expressions and hence the rules in the figure are sufficient for the transformation.) The linearity in the use of each variable $\mathtt{x}_i$ is crucial here for uniquely identifying the expression $M_{\mathtt{x}_i}$. The function value `\y`$_0$ `-> (lj=`$N_0$`)` returned

by the function `rf` in Figure 1(b) is transformed similarly but to `(lj=`$\mathcal{W}'[\![N_0]\!]$`, hk=`$\mathcal{W}'[\![M_{\mathtt{x}_0}]\!]$`)`. To complete the transformation, the sequential let bindings are replaced with a single circular let. The final transformation result is given in Figure 1(c).

The above transformation process (1) is explained as follows. Every recursive call `rf` $t$ $\cdots$, whose result is denoted by a variable $\mathtt{x}_i$ ($i > 0$), is transformed into `rf'` $t$ $R$, where $R$ is a record of type `(hj::[a],lk::[a])` and the value of $R$ at each record label is drawn from the source program as follows: `hj` value is set to $N_i$, which is taken from the `hj` field of the record expression returned by the function $\backslash\mathtt{y}_i \rightarrow$ `(hj=`$N_i$`)` (*i. e.*, the second argument to the recursive call); Since every variable $\mathtt{x}_i$ is bound to the result of a recursive call `rf'` $t$ $\cdots$ (of type `(hk::[a],lj::[a])`) in the target program, every application site `#lj (`$\mathtt{x}_i$ `(lk=`$M$`))` is rewritten to `#lj` $\mathtt{x}_i$, which just refers to the `lj` value returned by the recursive call. Also, every occurrence of expression `#hk` $\mathtt{y}_i$ is rewritten to `#hk` $\mathtt{x}_i$, which just takes the `hk` value returned by the recursive call. The transformation for the case $i = 0$ is similarly explained.

The above explanation only alludes to the validity of the transformation from the viewpoint of the type correctness and it tells little about the semantic correctness. It is indeed difficult to give a complete justification on the transformation in a descriptive way. The source of difficulty is that the transformation we consider is a *non-local transformation*, where the result of transformation on a program expression depends on that of another transformation site (*e. g.*, in the transformation rule (1), the transformation result refers to that of another site, namely $\mathcal{W}'[\![M_{\mathtt{x}_i}]\!]$). Even worse, the target program involves a *circularity*, which is introduced by the circular let construct, in its program structure. Due to this circularity, it is even not obvious if the transformation preserves the termination behavior of the source program.

In order to obtain a complete justification, we would need to resort to a formal argument based upon a firm mathematical basis. We notice that the usual inductive argument on the program structure does not work effectively, since there is no base case to induct on the circular structure. In [18], Voigtländer faced this problem in the correctness proof of his deforestation algorithm too. He argued the correctness of each local transformation step only, leaving that of the entire transformation not formally proved.

In the present paper, we employ the syntactic logical relations [14] as the mathematical basis for reasoning about the correctness of the higher-order removal program transformation. For this, we reformulate the syntatic logical relations $\{\Delta_\tau\}_\tau$ so that, every transformation source $M$ and its transformation result, denoted by $\mathcal{W}[\![M]\!]$, are related, *i. e.*, $(M, \mathcal{W}[\![M]\!]) \in \Delta_\tau$. This enables a simultaneous reasoning on different transformation sites: we can induce the relation between the pair of a transformation source and its result (which includes references to the results of transformation on other sites) from the type structure, not resorting to the induction hypothesis on the program structure.

We note that, unlike the usual logical relations, the logical relations formulated as above induce relations between expressions of different types. That is,

$\Delta_\tau$ is a relation between expressions of type $\tau$ and $\mathcal{T}[\![\tau]\!]$, where $\mathcal{T}[\![\cdot]\!]$ is a type conversion function that replaces every type subexpression $\sigma_1$ in $\tau$ with $\sigma_2$. We will discuss the observational equivalence of expressions of different types up to certain contexts that subsume this type mismatch. (See Section 4.2 for the details.)

## 3 Relational Framework for a Functional Language

This section constructs a relational framework for reasoning about observational equivalence of programs, following the technique developed in [14]. The considered programming language is a simply typed call-by-name functional language.[1] In order to avoid syntactic verbosity, we assume the only recursive data structure is lists. However, the following results are easily generalized to include other data structures such as binary trees.

### 3.1 The simply typed functional language

The set of types and language expressions are defined as follows.

$$
\begin{array}{lll}
\textbf{Types} & \tau & ::= \tau \to \tau \mid (l_1 : \tau_1, \ldots, l_n : \tau_n) \mid \tau \; \mathit{list} \\
\textbf{Expressions} & M & ::= x \mid \lambda x : \tau . M \mid M M \mid \textbf{fix}\, M \\
 & & \quad \mid (l_1 = M_1, \ldots, l_n = M_n) \\
 & & \quad \mid \textbf{match}\, M\, \textbf{of}\, (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M \\
 & & \quad \mid \textbf{nil}_\tau \mid M :: M \mid \textbf{case}\, M\, \textbf{of nil} \Rightarrow M, x :: y \Rightarrow M
\end{array}
$$

Types are either a function type, a record type, or a list type. The set of type expressions are defined inductively, with *unit type* () (the record type which has no record labels) being the base case of induction. We say a type $\tau$ is $\tau'$*-free*, if $\tau$ has no occurrence of $\tau'$ as a type subexpression.

The syntax of the language is an extension of the simply typed $\lambda$-calculus with fixpoint operator $\textbf{fix}\, M$, records $(l_1 = M_1, \ldots, l_n = M_n)$ $(n \geq 0)$, **match**-expression on records, empty list $\textbf{nil}_\tau$, list constructor $M :: M$, and **case**-branch on lists. Records are labeled products, where the order of labels is insignificant. Expression $\textbf{match}\, M\, \textbf{of}\, (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M'$ matches a record $M$ against a pattern $(l_1 = x_1, \ldots, l_n = x_n)$ and evaluates $M'$ with binding each $x_i$ to the corresponding element identified by the label $l_i$ (the set of labels $l_1, \ldots, l_n$ is a subset of the labels of the record $M$).

As usual, a variable occurrence of $x$ in an expression $M$ is said to be *free*, if $x$ is not bound by any binding of $\lambda$-, **match**-, and **case**-constructs in $M$. We write $fv(M)$ to denote the set of free variables in $M$. An expression $M$ is called *closed* if $fv(M) = \emptyset$. We write $M[x_1/N_1, \ldots, x_n/N_n]$ to represent a *substitution* to expression $M$ whose free occurrences of variables $x_1, \ldots, x_n$ are simultaneously replaced with $N_1, \ldots, N_n$, *resp.* We assume any substitution is capture-free, *i. e.*, any free variable in $N_i$'s is not bound by the substitution.

[1] We consider a monomorphically typed language since parametricity is not the concern of the present paper. The results in the paper can be easily generalized to a polymorphically typed language, however.

$$\frac{}{V \Downarrow V} \qquad \frac{F \Downarrow \lambda x:\tau.M \quad M[x/A] \Downarrow V}{FA \Downarrow V} \qquad \frac{F(\mathbf{fix}\, F) \Downarrow V}{\mathbf{fix}\, F \Downarrow V}$$

$$\frac{M \Downarrow (l_1 = M_1, \ldots, l_n = M_n, \ldots) \quad M'[x_1/M_1, \ldots, x_n/M_n] \Downarrow V}{\mathbf{match}\, M\, \mathbf{of}\, (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M' \Downarrow V}$$

$$\frac{M \Downarrow \mathbf{nil}_\tau \quad M_1 \Downarrow V}{\mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2 \Downarrow V} \qquad \frac{M \Downarrow H :: T \quad M_2[x/H, y/T] \Downarrow V}{\mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2 \Downarrow V}$$

**Fig. 4.** Operational Semantics

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x:\tau \vdash M : \tau'}{\Gamma \vdash \lambda x:\tau.M : \tau \to \tau'} \qquad \frac{\Gamma \vdash F : \tau \to \tau' \quad \Gamma \vdash A : \tau}{\Gamma \vdash FA : \tau'}$$

$$\frac{\Gamma \vdash F : \tau \to \tau}{\Gamma \vdash \mathbf{fix}\, F : \tau} \qquad \frac{\Gamma \vdash M_1 : \tau_1 \quad \cdots \quad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash (l_1 = M_1, \ldots, l_n = M_n) : (l_1 : \tau_1, \ldots, l_n : \tau_n)}$$

$$\frac{\Gamma \vdash M : (l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots) \quad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M' : \tau}{\Gamma \vdash \mathbf{match}\, M\, \mathbf{of}\, (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M' : \tau}$$

$$\frac{}{\Gamma \vdash \mathbf{nil}_\tau : \tau\ list} \qquad \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash M' : \tau\ list}{\Gamma \vdash M :: M' : \tau\ list}$$

$$\frac{\Gamma \vdash M : \tau\ list \quad \Gamma \vdash M_1 : \tau' \quad \Gamma, x : \tau, y : \tau\ list \vdash M_2 : \tau'}{\Gamma \vdash \mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M_1,\ x :: y \Rightarrow M_2 : \tau'}$$

**Fig. 5.** Typing Rules

The formal (call-by-name) operational semantics is given in Figure 4. The *evaluation relation* is expressed by a binary relation $M \Downarrow V$, which reads the expression $M$ is evaluated to $V$, where $V$ is a *value* that belongs to a subset of expressions defined below.

$$V \ ::= \ \lambda x:\tau.M \ \mid \ (l_1 = M_1, \ldots, l_n = M_n) \ \mid \ \mathbf{nil}_\tau \ \mid \ M :: M.$$

The typing rules are defined in the usual way, as given in Figure 5. A *typing judgment* is a ternary relation $\Gamma \vdash M : \tau$, where $\Gamma$ is a finite map from variables to types, called *type environment*, $M$ is an expression, and $\tau$ is a type. We conventionally write a type environment as $x_1 : \tau_1, \ldots, x_n : \tau_n$ $(n \geq 0)$. We write $Exp\,(\tau)$ for the set of closed expressions of type $\tau$, that is, $Exp\,(\tau) = \{M \mid \vdash M : \tau\}$.

In what follows, we assume a few syntactic conventions for brevity.

We write $\Omega_\tau$ for $\mathbf{fix}\, (\lambda x:\tau.x)$, which is an expression of type $\tau$ that diverges under any context. The subscript may be omitted occasionally.

The *field selection* operator $\#l\ M$ is a shorthand for

$$\#l\ M \quad = \quad \mathbf{match}\, M\, \mathbf{of}\, (l = z) \Rightarrow z \qquad (z \text{ is a fresh variable}).$$

Circular let definitions are expressed by combining fixpoint and record operators:

$$\begin{aligned} &\mathbf{let}\ x_1 = M_1; \ldots; x_n = M_n\ \mathbf{in}\ M \\ = \ &\mathbf{match}\, (\mathbf{fix}\ (\lambda r : (\overline{x_1} : \tau_1, \ldots, \overline{x_n} : \tau_n). \\ &\qquad \mathbf{match}\, r\, \mathbf{of}\, (\overline{x_1}{=}x_1, \ldots, \overline{x_n}{=}x_n) \Rightarrow (\overline{x_1} = M_1, \ldots, \overline{x_n} = M_n))) \\ &\mathbf{of}\, (\overline{x_1} = x_1, \ldots, \overline{x_n} = x_n) \Rightarrow M \end{aligned}$$

where each $\tau_i$ is the type of $M_i$ and the variable $r$ and the labels $\overline{x_i}'s$ are fresh.

### 3.2 The syntactic logical relation

We construct the syntactic logical relations for the simply typed functional language defined above. Due to the page limitation, we omitted proofs and some of the technical details, for which readers are deferred to Pitts' original paper [14].

**$\top\top$-closed relations** A *frame stack* is a list of *frames*, written $Id \circ F_n \circ \cdots \circ F_1$ $(n \geq 0)$, where $Id$ represents the empty context, and frames $F_i$'s are either in the form $(-M)$, $(\mathbf{match} - \mathbf{of}\ (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M)$, or $(\mathbf{case} - \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2)$. We are interested in only frame stacks $S \in \mathit{Stack}(\tau)$, where $\mathit{Stack}(\tau)$ denotes the set of well-typed frame stacks representing an evaluation context whose argument (*i. e.*, the expression to be filled in the context) has type $\tau$ and overall result of computation has type $\tau'$ *list* for some type $\tau'$. For the lack of space, we omit the formal typing rules on frame stacks.

$$\frac{S \top M[x/A]}{S \circ (-A) \top \lambda x : \tau.M} \qquad \frac{S \circ (-A) \top F}{S \top FA} \qquad \frac{S \circ (-(\mathbf{fix}\ F)) \top F}{S \top \mathbf{fix}\ F}$$

$$\frac{}{Id \top \mathbf{nil}_\tau} \qquad \frac{S \circ (\mathbf{match} - \mathbf{of}\ (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M') \top M}{S \top \mathbf{match}\, M\, \mathbf{of}\ (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M'}$$

$$\frac{S \top M'[x_1/M_1, \ldots, x_n/M_n]}{S \circ (\mathbf{match} - \mathbf{of}\ (l_1 = x_1, \ldots, l_n = x_n) \Rightarrow M') \top (l_1 = M_1, \ldots, l_n = M_n, \ldots)}$$

$$\frac{}{Id \top \mathbf{nil}_\tau} \qquad \frac{S \circ (\mathbf{case} - \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2) \top M}{S \top \mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2}$$

$$\frac{S \top M_1}{S \circ (\mathbf{case} - \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2) \top \mathbf{nil}_\tau}$$

$$\frac{S \top M_2[x/H, y/T]}{S \circ (\mathbf{case} - \mathbf{of\ nil} \Rightarrow M_1, x :: y \Rightarrow M_2) \top H :: T}$$

**Fig. 6.** Rules for termination relation

A binary relation $S \top M$ is defined inductively, for every $S \in \mathit{Stack}(\tau)$ and $M \in \mathit{Exp}(\tau)$, by the rules given in Figure 6. The relation $S \top M$ reads: the evaluation of $M$ under context $S$ terminates and yields $\mathbf{nil}_{\tau'}$ for some $\tau'$.

Let us write $\mathit{Rel}(\tau, \tau')$ to denote the set of all relations between well-typed expressions that are subsets of $\mathit{Exp}(\tau) \times \mathit{Exp}(\tau')$. Similarly, we write $\mathit{StRel}(\tau, \tau')$ to denote the set of all relations between stack frames that are subsets of $\mathit{Stack}(\tau) \times \mathit{Stack}(\tau')$. We define a pair of complement operators $(-)^\top$ between these two relational domains.

**Definition 1.** *For any* $r \in \mathit{Rel}(\tau, \tau')$, *we define* $r^\top = \{(S, S') \mid \forall (M, M') \in r.\ (S \top M \Leftrightarrow S' \top M')\}$. *Also, for any* $s \in \mathit{StRel}(\tau, \tau')$, *we define* $s^\top = \{(M, M') \mid \forall (S, S') \in s.\ (S \top M \Leftrightarrow S' \top M')\}$.

The two $(-)^{\top}$ operators together define a Galois connection [13] with respect to inclusion (*i. e.*, $r \subseteq s^{\top} \Leftrightarrow s \subseteq r^{\top}$). An important property derived from Galois connections is that the double-negation operators $(-)^{\top\top}$ are a closure operator. That is, they are monotone ($r \subseteq r' \Rightarrow r^{\top\top} \subseteq r'^{\top\top}$), inflationary ($r \subseteq r^{\top\top}$), and idempotent ($r^{\top\top} = (r^{\top\top})^{\top\top}$).

**Definition 2.** *A relation $r$ is called* $\top\top$-closed, *if $r = r^{\top\top}$.*

**Defining logical relation** For each type constructor, we define a relation constructor, called *action*.

**Definition 3 (Actions).** *Let $r, r', r_1, \ldots, r_n$ be relations such that $r \in Rel\,(\tau, \tau')$, $r' \in Rel\,(\sigma, \sigma')$, $r_i \in Rel\,(\tau_i, \tau_i')$ $(i = 1, \ldots, n)$.*

*We define actions $(-) \to (-) : Rel\,(\tau, \tau') \times Rel\,(\sigma, \sigma') \to Rel\,(\tau \to \sigma, \tau' \to \sigma')$, $(l_1 : (-), \ldots, l_n : (-)) : Rel\,(\tau_1, \tau_1') \times \cdots \times Rel\,(\tau_n, \tau_n') \to Rel((l_1 : \tau_1, \ldots, l_n : \tau_n), (l_1 : \tau_1', \ldots, l_n : \tau_n'))$, and $(-)list : Rel\,(\tau, \tau') \to Rel\,(\tau\ list, \tau'\ list)$ as follows.*

$$
\begin{aligned}
r \to r' &= \{(F, F') \mid (FA, F'A') \in r' \text{ for all } (A, A') \in r\} \\
(l_1 : r_1, \ldots, l_n : r_n) &= \{((l_1 = M_1, \ldots, l_n = M_n), (l_1 = M_1', \ldots, l_n = M_n')) \mid \\
&\qquad (M_1, M_1') \in r_1, \ldots, (M_n, M_n') \in r_n\}^{\top\top} \\
(r)list &= \nu\gamma.\Phi_r(\gamma)
\end{aligned}
$$

*In the definition of $(r)list$, $\Phi_r : Rel\,(\tau\ list, \tau'\ list) \to Rel\,(\tau\ list, \tau'\ list)$ is a function such that $\Phi_r(\gamma) = (\{(\mathbf{nil}_\tau, \mathbf{nil}_{\tau'})\} \cup \{(H, T), (H', T') \mid (H, H') \in r \text{ and } (T, T') \in \gamma\})^{\top\top}$, and $\nu\gamma.F(\gamma)$ denotes a greatest fixpoint of a function $F$.*

The action for list types is well-defined (*i. e.*, the greatest fixpoint exists), since $\top\top$-closure is monotone and hence so is $\Phi_r$.

**Definition 4.** *The syntactic logical relations $\{\Delta_\tau\}_\tau$ are inductively defined on the structure of type $\tau$ by the following construction rules:*

$$
\begin{aligned}
\Delta_{\tau \to \tau'} &= \Delta_\tau \to \Delta_{\tau'} \\
\Delta_{(l_1:\tau_1, \ldots, l_n:\tau_n)} &= (l_1 : \Delta_{\tau_1}, \ldots, l_n : \Delta_{\tau_n}) \\
\Delta_{\tau\ list} &= (\Delta_\tau)list
\end{aligned}
$$

The above definition is carefully designed so that each relation $\Delta_\tau$ is $\top\top$-closed at any type index $\tau$. This, together with the following theorem, justifies the application of the fixpoint operator to expression $M$ of any type $\tau \to \tau$.

**Theorem 1.** [14, Theorem 3.11] *Let $r \in Rel\,(\tau, \tau')$ be a $\top\top$-closed relation, and let $F \in Exp\,(\tau \to \tau)$ and $F' \in Exp\,(\tau' \to \tau')$ be expressions such that $(FA, F'A') \in r$ for all $(A, A') \in r$. Then, it holds that $(\mathbf{fix}\, F, \mathbf{fix}\, F') \in r$.*

This theorem also gives a justification on the relational reasoning of programs that involve circular let expressions and recursive function definitions, which are a derivative of fixpoint operator.

### 3.3 Equational congruence relation and its properties

Let us write $M \sim M'$ to indicate that closed expressions $M$ and $M'$ are related w.r.t. the above logical relations, *i. e.*, $(M, M') \in \Delta_\tau$ for some type $\tau$. The next proposition holds for $\sim$-relation.

**Proposition 1.** [14, Proposition 4.6] *The relation $\sim$ is a congruence relation,* i. e., *it is reflexive, symmetric, transitive, and substitutive (*i. e., *if $fv(M) = \{x_1, \ldots, x_n\}$ and $M_1 \sim M'_1, \ldots, M_n \sim M'_n$ then $M[x_1/M_1, \ldots, x_n/M_n] \sim M[x_1/M'_1, \ldots, x_n/M'_n]$).*

In order for this proposition to hold for the present language, we need to verify some properties concerning record types. The formal proof is given in a separate paper [12].

In addition to the congruence, several equational properties on $\sim$ can be derived from the so-called *Kleene equivalence*.

**Proposition 2.** [14, Corollary 3.15] *Let $M, M' \in Exp(\tau)$. $M$ and $M'$ are called* Kleene equivalent, *written $M =_{\mathrm{kl}} M'$, if it holds that $\forall V.(M \Downarrow V \Leftrightarrow M' \Downarrow V)$.*

*Kleene equivalence respects $\sim$ relation, that is, if $M_1 =_{\mathrm{kl}} M'_1$, $M_2 =_{\mathrm{kl}} M'_2$, and $M_1 \sim M_2$, then $M'_1 \sim M'_2$.*

$$(\lambda x : \tau.M)N \sim M[x/N] \qquad \mathbf{fix}\, F \sim F(\mathbf{fix}\, F) \qquad \mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M \sim M\theta$$
$$\#l\, (l = M, \ldots) \sim M \qquad \mathbf{case}\, \mathbf{nil}_\tau\, \mathbf{of}\, \mathbf{nil} \Rightarrow M_1, h :: t \Rightarrow M_2 \sim M_1$$
$$\mathbf{case}\, H :: T\, \mathbf{of}\, \mathbf{nil} \Rightarrow M_1, h :: t \Rightarrow M_2 \sim M_2[h/H, t/T]$$
$$\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M \sim M\theta \qquad \mathbf{let}\, \mathcal{B}\, \mathbf{in}\, (\lambda x : \tau.M) \sim \lambda x : \tau.(\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M)$$
$$\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, MN \sim (\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M)(\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, N) \qquad \mathbf{let}\, \mathcal{B}\, \mathbf{in}\, \#l\, M \sim \#l\, (\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M)$$
$$\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, (l_1 = N_1, \ldots, l_m = N_m) \sim (l_1 = \mathbf{let}\, \mathcal{B}\, \mathbf{in}\, N_1, \ldots, l_m = \mathbf{let}\, \mathcal{B}\, \mathbf{in}\, N_m)$$
$$\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, H :: T \sim (\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, H) :: (\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, T)$$

where $\mathcal{B} = x_1 = M_1; \cdots; x_n = M_n$ and $\theta = [x_1/\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M_1, \ldots, x_n/\mathbf{let}\, \mathcal{B}\, \mathbf{in}\, M_n]$.

**Fig. 7.** Derived $\sim$-relations

Figure 7 gives several $\sim$-relations derived from Kleene equivalence; $\sim$-relation is stable up to $\beta$-reductions, fixpoint unfoldings (and therefore circular let unfoldings), and destructor/constructor pair cancellations (for records and lists); Also, the circular let constructs commute with several other constructs.

Since it has been proved that the syntactic logical relations coincide with the observational equivalence [14, Theorem 4.15], we hitherto reason about observational equivalence of expressions up to $\sim$, using the laws in Proposition 1 and Figure 7.

We notice that, though the present formalization considers list types only, the above results are easily generalized to any polynomial data types, *e. g.* binary

trees. The relations for any polynomial data types can be defined as a greatest fixpoint of a monotone function, likewise in the definition for list types.

## 4 The Correctness of Higher-Order Removal Transformation

Throughout this section, we write $U, X, Y$ to denote disjoint sets of variables $U = \{u_1, \ldots, u_m\}$, $X = \{x_0, \ldots, x_n\}$, and $Y = \{y_0, \ldots, y_n\}$ $(m, n \geq 0)$. We also fix a list type $\tau\ list$.

We omitted some of the technical details and proofs from this section, due to the page limitation. They can be found in a separate paper [12].

### 4.1 Definition of the higher-order removal transformation

Our transformation algorithm applies to a class of expressions of the following form (which is referred to as $I$, in the rest of the paper):

$$\begin{array}{rcl} I & = & \lambda x_0 : (hk : \tau\ list) \to (hj : \tau\ list). \\ & & \quad \textbf{let}\ x_n = f_n\ (\lambda y_n : (hk : \tau\ list).(hj = N_n))\ \textbf{in} \\ & & \quad \cdots \\ & & \quad \textbf{let}\ x_1 = f_1\ (\lambda y_1 : (hk : \tau\ list).(hj = N_1))\ \textbf{in} \\ & & \quad \lambda y_0 : (lk : \tau\ list).(lj = N_0) \end{array} \tag{2}$$

where $f_1, \ldots, f_n$ $(n \geq 0)$ are variables of type $((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list))$, and $N_1, \ldots, N_n$ are expressions that belong to a particular syntactic class. That is, $N_0 \in Exp_i^{\Gamma_U, X, Y, \pi}((lj : \tau\ list))$ and $N_i \in Exp_i^{\Gamma_U, X, Y, \pi}(hj : \tau\ list)$ $(1 \leq i \leq n)$, where the syntactic class designated by $Exp_i^{\Gamma_U, X, Y, \pi}(\tau)$ is defined as below.

**Definition 5.** *Let $\Gamma_U = u_1 : \tau_1, \ldots, u_m : \tau_m$ be a type environment and $\pi$ be a function $\{0, \ldots, n\} \mapsto \{0, \ldots, n\}$ such that $i > \pi(i)$ for every $i = 1, \ldots, n$.*

*We define a family of set of expressions $\{Exp_i^{\Gamma_U, X, Y, \pi}(\tau)\}_{0 \leq i \leq n}$ inductively by the following rules (for brevity, we write $Exp_i(\tau)$ for $Exp_i^{\Gamma_U, X, Y, \pi}(\tau)$):*

***h1*** *$M \in Exp_i(\tau)$ whenever $\Gamma_U \vdash M : \tau$,*
***h2*** *$H :: T \in Exp_i(\tau\ list)$ if $H \in Exp_i(\tau)$ and $T \in Exp_i(\tau\ list)$,*
***h3*** *$\#lk\ y_0 \in Exp_0(\tau\ list)$,*
***h4*** *$\#hk\ y_i \in Exp_i(\tau\ list)$ $(1 \leq i \leq n)$,*
***h5*** *$\#hj\ (x_0\ (hk = M_{x_0})) \in Exp_{\pi(0)}(\tau\ list)$, where $fv(M_{x_0}) \cap X = \emptyset$ and $M_{x_0} \in Exp_{\pi(0)}(\tau\ list)$, and*
***h6*** *$\#lj\ (x_i\ (lk = M_{x_i})) \in Exp_{\pi(i)}(\tau\ list)$ $(1 \leq i \leq n)$, where $fv(M_{x_i}) \cap X = \emptyset$ and $M_{x_i} \in Exp_{\pi(i)}(\tau\ list)$.*

For each $i$, $Exp_i(\tau)$ represents a syntactically restricted set of expressions of type $\tau$, which possibly contains free variables from $U$, $X$, and $Y$. Each variable $x_i \in X$ has to appear in a function application form (Rules **h5** and **h6**. $\pi(i)$

refers to the index $j$ of expression $N_j$ from which the variable $x_i$ is referenced); Each variable $y_i \in Y$ must appear with a field selection operator (**h3** and **h4**). The above expressions can be combined by the list constructor (**h2**).

The expression $I$ is further assumed to meet a *linearity condition*: Each variable $x_i \in X$ occurs once and only once throughout the expressions $N_0, \ldots, N_n$. (The side condition $i > \pi(i)$ $(i = 1, \ldots, n)$ guarantees that every use of $x_i$ is preceded by the corresponding let binding **let** $x_i = \cdots$.) We intendedly excluded $\lambda$-abstractions from the above rules, in order to prevent each unique occurrence of a variable $x_i$ from being applied to the argument that receives varying denotations. This may happen when a $\lambda$-expression that abstracts the $x_i$'s application is used in different contexts, *e. g.*, $(\lambda f. \cdots f\,A_1 \cdots f\,A_2)(\lambda z.\#lj\,(x_i(lk = \#lk\,z)))$.

$$
\begin{array}{l}
\mathcal{W}[\![\lambda x_0 : (hk : \tau\ list) \to (hj : \tau\ list).\\
\quad \textbf{let}\ x_n = f_n\ (\lambda y_n : (hk : \tau\ list).(hj = N_n))\ \textbf{in}\\
\quad\quad \cdots\\
\quad \textbf{let}\ x_1 = f_1\ (\lambda y_1 : (hk : \tau\ list).(hj = N_1))\ \textbf{in}\ \lambda y_0 : (lk : \tau\ list).(lj = N_0)]\!]\\
=\ \lambda x_0 : (hj : \tau\ list, lk : \tau\ list).\\
\quad\quad \textbf{let}\ x_n = f'_n\ (hj = \mathcal{W}'[\![N_n]\!], lk = \mathcal{W}'[\![M_{x_n}]\!]);\\
\quad\quad\quad\quad \cdots\\
\quad\quad\quad x_1 = f'_1\ (hj = \mathcal{W}'[\![N_1]\!], lk = \mathcal{W}'[\![M_{x_1}]\!])\\
\quad\quad \textbf{in}\ (lj = \mathcal{W}'[\![N_0]\!], hk = \mathcal{W}'[\![M_{x_0}]\!])
\end{array}
$$

where $\mathcal{W}'$ is the transformation defined in Figure 3, and variables $f_1, \ldots, f_n$ and $f'_1, \ldots, f'_n$ denote recursive function calls to a substructure of the input data type.

**Fig. 8.** The higher-order removal transformation

The formal definition of the higher-order removal transformation $\mathcal{W}$ is given in Figure 8. It is easy to verify that this transformation is type correct.

**Theorem 2.** *Let us write* $\sigma_1 = ((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list))$ *and* $\sigma_2 = (hj : \tau\ list, lk : \tau\ list) \to (hk : \tau\ list, lj : \tau\ list)$. *If* $\Gamma_U, f_1 : \sigma_1, \ldots, f_n : \sigma_1 \vdash I : \sigma_1$, *then* $\Gamma_U, f'_1 : \sigma_2, \ldots, f'_n : \sigma_2 \vdash \mathcal{W}[\![I]\!] : \sigma_2$.

### 4.2 Correctness of higher-order removal transformation

For the purpose of proving the correctness of the transformation, we reformulate the syntactic logical relations $\{\Delta_\tau\}_\tau$, as we discussed in Section 2. We replace only the relation indexed by $\sigma_1 = ((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list))$ with the relation $\Theta$ defined below.

**Definition 6.** $\Theta \in Rel(((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list)), (hj : \tau\ list, lk : \tau\ list) \to (hk : \tau\ list, lj : \tau\ list))$ *is a relation defined by:*

$(I, J) \in \Theta$ *iff*

*(a)* $(\#hk\ (J\ (hj = H, lk = L)), \#hk\ (J\ (hj = H', lk = L'))) \in \Delta_{\tau\ list}$ *for all* $(L, L') \in \Delta_{\tau\ list}$ *and* $H, H' \in Exp\ (\tau\ list)$, *and*

*(b)* $\left(\begin{array}{cc} \#lj\ (I\ G\ (lk = L)), & \#lj\ (J\ (hj = \#hj\ (G'\ (hk = \#hk\ (J\ (hj = \Omega_{\tau\ list}, lk = L'))))), \\ & lk = L')) \end{array}\right) \in \Delta_{\tau\ list}$

*for all* $(G, G') \in (hk : \Delta_{\tau\ list}) \to (hj : \Delta_{\tau\ list})$ *and* $(L, L') \in \Delta_{\tau\ list}$.

We note that the above modification does retain the original relation $\Delta_\tau$ whenever $\tau$ is $\sigma_1$-free, *i. e.*, $\tau$ is free from occurrences of $\sigma_1$. This implies that the observational equivalence of programs of $\sigma_1$-free types can be safely reasoned up to $\sim$-relation introduced in Section 3.3. In what follows, we assume $\tau$ ranges over the set of $\sigma_1$-free types.

We prove that $\Theta$ relates the transformation source $I$ and the target $\mathcal{W}[\![I]\!]$.

**Proposition 3.** *For every expression* $I$ *of the form* (2), *it holds that*

$$(I\theta_0, \mathcal{W}[\![I]\!]\theta_0') \in \Theta$$

*for any* $\theta_0 = [u_1/U_1, \ldots, u_m/U_m, f_1/I_1, \ldots, f_n/I_n]$ *and* $\theta_0' = [u_1/U_1', \ldots, u_m/U_m', f_1'/J_1, \ldots, f_n'/J_n]$ *satisfying* $(U_k, U_k') \in \Delta_{\Gamma_U(u_k)}$ $(1 \leq k \leq m)$ *and* $(I_i, J_i) \in \Theta$ $(1 \leq i \leq n)$.

We can prove this proposition basically by induction on the structure of expression, referring to the definition of the relation $\Theta$ when we encounter let bound variables. Instead of giving the lengthy proof of this proposition, we explain what are the intentions behind the definition of $\Theta$ and how they contribute to establishing the formal proof. (See [12] for the full proof.)

The property (a) of the definition of $\Theta$ indicates that the $hk$ value of the record computed by the transformed program $J = \mathcal{W}[\![I]\!]$ depends only on the $lk$ value of the input record and is not affected at all by the input $hj$ value. This means the expression $H'$ for the $hj$ field can be substituted with any other well-typed expression. This property is crucially exploited in the proof in order to break the circularity introduced by the circular let construct: We can suppress the circular references in the transformed program by virtually replacing certain variable references with expressions that do not cause circularity.

The property (b) implies that the $lj$ value, which carries the final answer computed by the original producer/consumer function pair, of the record computed by a single call to $I$ is obtained by using $J$ twice. Computing an $lj$ value by a call to $I$ with $G$ being the initial accumulating parameter is equal to computing an $lj$ value by first calling $J$ to obtain the $hk$ value from the input $lk$ value $L'$, and then by applying $J$ to the pair of the $hj$ value and the input $lk$ value $L'$, where the $hj$ value is obtained by applying $G'$ (an observationally equivalent copy of $G$) to the $hk$ value computed by the first call of $J$. This justifies the transformation on $\#lj\ (x_i\ (lk = M_{x_i}))$ by $\mathcal{W}'$.

The total correctness of the higher-order removal transformations is derived from the Proposition 3. Suppose we are given the following expressions $\mathcal{I}$ and $\mathcal{J}$:

$$\begin{aligned}
\mathcal{I} &= \lambda f : \tau\ list \to ((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list)).\\
&\quad (\lambda x : \tau\ list.\ \mathbf{case}\ x\ \mathbf{of\ nil} \Rightarrow I_{\mathbf{nil}}, h :: t \Rightarrow I_{h::t}[f_1/f\ t, \ldots, f_n/f\ t])\\
\mathcal{J} &= \lambda f' : \tau\ list \to (hj : \tau\ list, lk : \tau\ list) \to (hk : \tau\ list, lj : \tau\ list)\\
&\quad (\lambda x : \tau\ list.\ \mathbf{case}\ x\ \mathbf{of\ nil} \Rightarrow \mathcal{W}[\![I_{\mathbf{nil}}]\!],\\
&\qquad\qquad h :: t \Rightarrow \mathcal{W}[\![I_{h::t}]\!][f'_1/f'\ t, \ldots, f'_n/f'\ t])
\end{aligned}$$

where

- $I_{\mathbf{nil}}$ is an expression of the form (2) with $U = \emptyset$ and $n = 0$ (*i. e.*, no **let**-bindings), and
- $I_{h::t}$ is an expression of the form (2) with $U = \{h\}$.

In the above, $f\ t$ and $f'\ t$ express recursive calls to the list substructure $t$. This is generalized to any polynomial data types: *E. g.*, for the case of binary trees instead of lists, the variables $f_i$'s ($f'_i$'s, *resp.*) would be substituted with either $f\,t$ or $f\ t'$ ($f'\ t$ or $f'\ t'$, *resp.*), where $t$ and $t'$ are variables representing the two different subtrees of a tree node.

In the present paper, we only prove the observational equivalence between $\mathbf{fix}\,\mathcal{I}$ and $\mathbf{fix}\,\mathcal{J}$, when they are applied to lists of finite length. We notice that the observationally equivalent pairs of expressions of lists of length less than $n$ is characterized by the relation $\Phi^n_{\Delta_\tau}(\emptyset)$, where $\Phi_r$ is the function given in Definition 3. This characterization can be also generalized to any polynomial data types, with an appropriate definition of $\Phi_r$.

**Theorem 3 (Correctness of transformation).** *Let $\mathcal{I}$ and $\mathcal{J}$ be expressions defined as above. It holds that*

$$((\mathbf{fix}\,\mathcal{I})L, (\mathbf{fix}\,\mathcal{J})L') \in \Theta \qquad \textit{for every } (L, L') \in \Phi^n_{\Delta_\tau}(\emptyset)\ (n \geq 1).$$

We prove this theorem by induction on the length of lists, where the induction principle is justified by the following lemma.

**Lemma 1.** *If $(L, L') \in \Phi^n_{\Delta_\tau}(\emptyset)$ $(n \geq 1)$, then the following properties hold.*

*(i)* $L \Downarrow \mathbf{nil}_\tau$ *iff* $L' \Downarrow \mathbf{nil}_\tau$, *and*
*(ii) if* $L \Downarrow H :: T$ *then* $L' \Downarrow H' :: T'$ *for some* $H'$ *and* $T'$ *such that* $(H, H') \in \Delta_\tau$ *and* $(T, T') \in \Phi^{n-1}_{\Delta_\tau}(\emptyset)$.
*(iii) if* $L' \Downarrow H' :: T'$ *then* $L \Downarrow H :: T$ *for some* $H$ *and* $T$ *such that* $(H, H') \in \Delta_\tau$ *and* $(T, T') \in \Phi^{n-1}_{\Delta_\tau}(\emptyset)$.

*In particular when $n = 1$, if $L \Downarrow V$ and $L' \Downarrow V'$ then $V = V' = \mathbf{nil}_\tau$.*

*Proof.* (of Theorem 3) Proof is by induction on $n$. If both $L$ and $L'$ diverge, then so do both $(\mathbf{fix}\,\mathcal{I})L$ and $(\mathbf{fix}\,\mathcal{J})L'$. Hence $((\mathbf{fix}\,\mathcal{I})L, (\mathbf{fix}\,\mathcal{J})L') \in \Theta$.

Now we suppose that both $L$ and $L'$ do not diverge. (Non-divergence of one implies that of the other, by lemma 1.) If $n = 1$, then by lemma 1 both $L$ and $L'$ evaluate to $\mathbf{nil}_\tau$. Then we have $(\mathbf{fix}\,\mathcal{I})L \sim I_{\mathbf{nil}}$ and $(\mathbf{fix}\,\mathcal{J})L' \sim \mathcal{W}[\![I_{\mathbf{nil}}]\!]$. Since $(I_{\mathbf{nil}}, \mathcal{W}[\![I_{\mathbf{nil}}]\!]) \in \Theta$ by proposition 3, it follows that $(\mathbf{fix}\,\mathcal{I})L \sim (\mathbf{fix}\,\mathcal{J})L'$.

Suppose $n > 1$. By lemma 1, we have $L \Downarrow H :: T$ and $L' \Downarrow H' :: T'$ for some $(H, H') \in \Delta_\tau$ and $(T, T') \in \Phi_{\Delta_\tau}^{n-1}(\emptyset)$. Then, it holds that

$$(\mathbf{fix}\,\mathcal{I})L \sim I_{h::t}[h/H, f_1/(\mathbf{fix}\,\mathcal{I})T, \ldots, f_n/(\mathbf{fix}\,\mathcal{I})T] \quad \text{and}$$
$$(\mathbf{fix}\,\mathcal{J})L' \sim \mathcal{W}[\![I_{h::t}]\!][h/H', f_1'/(\mathbf{fix}\,\mathcal{J})T', \ldots, f_n'/(\mathbf{fix}\,\mathcal{J})T'].$$

Since $((\mathbf{fix}\,\mathcal{I})T, (\mathbf{fix}\,\mathcal{J})T') \in \Theta$, by induction hypothesis and proposition 3, we have $(\mathbf{fix}\,\mathcal{I})L \sim (\mathbf{fix}\,\mathcal{J})L'$. □

Now we show the total correctness of the higher-order removal transformation using the results in this section. Suppose we are given a recursive definition of function $\mathbf{fix}\,\mathcal{I}$, a function $G$ of type $((hk : \tau\ list) \to (hj : \tau\ list)) \to ((lk : \tau\ list) \to (lj : \tau\ list))$, and lists $L$ and $L'$ of type $\tau\ list$ where $L$ is of finite length. Then, it follows from the property (b) of the relation $\Theta$ (Definition 6) that

$$\#lj\ ((\mathbf{fix}\,\mathcal{I})\ L\ G\ (lk = L'))$$

is observationally equivalent to

$$\#lj\ ((\mathbf{fix}\,\mathcal{J})(hj = \#hj\ (G\ ((\mathbf{fix}\,\mathcal{J})(hj = \Omega, lk = L'))), lk = L')).$$

Since the two expressions are related up to $\Delta_{\tau\ list}$, they can substitute for each other under any context.

Applying this to the example in Figure 1 (with regarding $\mathbf{fix}\,\mathcal{I}$ as the recursive function definition of `rf` and also $\mathbf{fix}\,\mathcal{J}$ as that of `rf'`), we obtain

```
revflat x = #lj (rf' x (hj=#hj ((\y -> (hj=#hk y))
                                (rf' x (hj=Ω, lk=[])), lk=[])))
```

The final result in Figure 1(c) follows by a simple calculation. Note that the diverging expression $\Omega$ can be substituted with any well-typed list expression, since it does not affect the computation at all (as indicated by the property (a) of Definition 6).

## 5 Conclusion

We have shown that the framework of syntactic logical relations developed by Pitts [14] can be a powerful device in proving the correctness of program transformation. This was evidenced through a concrete example, a higher-order removal transformation, which is a part of a deforestation algorithm for functions with accumulating parameters.

We believe that the general proof method presented in this paper can apply to the correctness proof of other program transformations. In fact, the two

auxiliary transformation processes between the shortcut fusion and the higher-order removal transformation (in Figure 2) can be justified by a similar proof technique. This fact constitutes the complete correctness proof for the deforestation algorithm that was proposed by the author [11], together with a few more technical elaborations on the syntactic condition to be met by the two transformations. The author would like to report this result in a near future.

In Theorem 3, we assumed that the inputs are only finite lists (or finite polynomial data structures, in general). We leave the proof for the inputs of infinite lists open, as we were not able to prove the relation $\Theta$ is $\top\top$-closed: If $\Theta$ were proved $\top\top$-closed, we would have it as a direct consequence of Theorem 1 without resorting to the induction principle on finite lists. This implies that the $\top\top$-closedness is a key to the question if the composition algorithms for functions with accumulating parameters are valid for infinite data structures. It is folklore in the community of macro tree transducers and attribute grammars that the answer is positive, but there is no formal justification.

The open problem mentioned above raises some technical issues which may be worth investigating. For the relation $\Theta$ to be $\top\top$-closed, so must be both of the two properties in Definition 6. Though each property has its own difficulty in showing its $\top\top$-closedness, that of the property (b) seems more fundamental: The problem amounts to showing the $\top\top$-closedness of a relation $\Delta_{\tau\to\tau}$, which is defined for some particular type $\tau$ by $\Delta_{\tau\to\tau} = \{(F, F') \mid (FA,\ F'(F'A')) \in \Delta_\tau \text{ for all } (A, A') \in \Delta_\tau\}$. An attempt to show this $\top\top$-closedness would be stuck, since we only consider contexts with a single hole whereas in this case we need to reason about contexts with two different holes. It would be interesting to investigate a refinement of Pitts' proof technique for proving (or disproving) this $\top\top$-closedness.

**Acknowledgment** I thank Jacques Garrigue for his valuable comments on a draft version. I am also grateful to anonymous referees whose suggestions are helpful for improving the presentation.

## References

1. W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994.
2. J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.
3. H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19(6) of *SIGPLAN Notices*, pages 157–170, June 1984.
4. A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, June 1993.
5. The Haskell home page. `http://www.haskell.org/`.
6. P. Johann. Short cut fusion: Proved and improved. In *Semantics, Applications, and Implementation of Program Generation, Second International Workshop: SAIG*

*2001*, volume 2196 of *Lecture Notes in Computer Science*, pages 47–71. Springer Verlag, 2001.
7. X. Leroy. Unboxed objects and polymorphic typing. In *Conference record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*, pages 177–188. ACM Press, 1992.
8. Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP98)*, pages 1–12. ACM Press, 1998.
9. J. C. Mitchell. *Foundations for Programming Languages*. Foundation of Computing Series. The MIT Press, 1996.
10. L. R. Nielsen. A denotational investigation of defunctionalization. Technical Report RS-00-47, BRICS, 2000.
11. S. Nishimura. Deforesting in accumulating parameters via type-directed transformations. In *Informal proceedings of Asian Workshop on Programming Languages and Systems 2002 (APLAS'02)*, 2002. electronic version: `http://www.math.kyoto-u.ac.jp/~susumu/papers/aplas02.ps.gz`.
12. S. Nishimura. Correctness of a higher-order removal transformation through a relational reasoning. Technical Report Kyoto-Math 2003-06, Kyoto University, 2003. electronic version: `http://www.math.kyoto-u.ac.jp/~susumu/papers/aplas03-long.ps.gz`.
13. O. Ore. Galois connexions. *Transactions of American Mathematical Society*, 55:493–513, 1944.
14. A. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.
15. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order Symbolic Computation*, 11(4):363–397, 1998. Reprint from the Proc. of the 25th ACM National Conference (1972).
16. T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings $6^{th}$ ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, 1993.
17. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the $7^{th}$ International Conference on Functional Programming Languages and Computer Architecture*, pages 306–313, La Jolla, California, June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.
18. J. Voigtländer. Using circular programs to deforest in accumulating parameters. In K. Asai and W.-N. Chin, editors, *ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 126–137. ACM Press, 2002.
19. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.