Verifying Relational Properties of Functional Programs by First-Order Refinement

Kazuyuki Asada   Ryosuke Sato   Naoki Kobayashi

University of Tokyo

{asada,ryosuke,koba}@kb.is.s.u-tokyo.ac.jp

Abstract

Much progress has been made recently on fully automated verification of higher-order functional programs, based on refinement types and higher-order model checking. Most of those verification techniques are, however, based on *first-order* refinement types, hence unable to verify certain properties of functions (such as the equality of two recursive functions and the monotonicity of a function, which we call *relational properties*). To relax this limitation, we introduce a restricted form of higher-order refinement types where refinement predicates can refer to functions, and formalize a systematic program transformation to reduce type checking/inference for higher-order refinement types to that for first-order refinement types, so that the latter can be automatically solved by using an existing software model checker. We also prove the soundness of the transformation, and report on preliminary implementation and experiments.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Automated verification, Higher-order functional language, Refinement types

1. Introduction

There has been much progress in automated verification techniques for higher-order functional programs [10, 12–14, 17, 18, 20].[1] Most of those techniques abstract programs by using *first-order* predicates on base values (such as integers), due to the limitation of underlying theorem provers and predicate discovery procedures. For example, consider the program:

```
let rec sum n =
  if n<0 then 0 else n+sum(n-1).
```

Using the existing techniques [10, 13, 17, 18], one can verify that `sum` has the first-order refinement type: $(n : \textbf{int}) \to \{m : \textbf{int} \mid m \geq n\}$, which means that `sum n` returns a value no less than `n`. Here, $\{m : \textbf{int} \mid P(m)\}$ is the (refinement) type of integers m that satisfy $P(m)$.

Due to the restriction to the first-order predicates, however, it is difficult to reason about what we call *relational properties*, such as the relationship between two functions, and the relationship between two invocations of a function. For example, consider another version of the sum function:

```
let rec sumacc n m =
  if n<0 then m else sumacc (n-1) (m+n)
and sum2 n = sumacc n 0
```

Suppose we wish to check that $\texttt{sum2}(n)$ equals $\texttt{sum}(n)$ for every integer n. With general refinement types [7], that would amount to checking that `sumacc` and `sum2` have the following types:[2]

$$\texttt{sumacc} : (n : \textbf{int}) \to (m : \textbf{int}) \to \{r : \textbf{int} \mid r = m + \texttt{sum}(n)\}$$
$$\texttt{sum2} : (n : \textbf{int}) \to \{r : \textbf{int} \mid r = \texttt{sum}(n)\}$$

The type of `sum2` means that `sum2` takes an integer as an argument and returns an integer r that equals the value of $\texttt{sum}(n)$. With the first-order refinement types, however, `sum` cannot be used in predicates, so the only way to prove that $\texttt{sum2}(n)$ equals $\texttt{sum}(n)$ would be to verify precise input/output behaviors of the functions:

$$\texttt{sum}, \texttt{sum2} : (n : \textbf{int}) \to \{r : \textbf{int} \mid (n \geq 0 \land r = n(n+1)/2) \lor (n < 0 \land r = 0)\}.$$

Since this involves non-linear and disjunctive predicates, automated verification (which involves automated synthesis of the predicates above) is difficult. In fact, most of the recent automated verification tools do not deal with non-linear arithmetic.

Actually, with the first-order refinement types, there is a difficulty even with the "trivial" property that `sum` satisfies $\texttt{sum}\ x = x + \texttt{sum}\ (x-1)$ for every $x \geq 0$. This is almost the definition of the `sum` function, and it can be expressed and verified using the general refinement type:

$$\texttt{sum} : \{f : \textbf{int} \to \textbf{int} \mid \forall x.\, x \geq 0 \Rightarrow f(x) = x + f(x-1)\}.$$

Yet, with the restriction to first-order refinement types, one would need to infer the precise input/output behavior of `sum` (i.e., that $\texttt{sum}(x)$ returns $x(x+1)/2$).[3]

We face even more difficulties when dealing with higher-order functions. Consider the program in Figure 1. Here, a list is encoded as a function that maps each index to the corresponding element (or `None` if the index is out of bounds) [14], and the append function is defined. Suppose that we wish to verify that `append xs nil = xs`. With general refinement types, the property would be expressed by:

$$\begin{aligned}\texttt{append} : (x : \textbf{int} \to \textbf{int option}) &\to \\ \{y : \textbf{int} \to \textbf{int option} \mid y(0) = \texttt{None}\} &\to \\ \{r : \textbf{int} \to \textbf{int option} \mid r = x\}&\end{aligned}$$

[1] In the present paper, by *automated* verification, we mean (almost) fully automated one, where a tool can automatically verify a given program satisfies a given specification (expressed either in the form of assertions or refinement type declarations), without requiring invariant annotations (such as pre/post conditions for each function). It should be contrasted with refinement type checkers [4, 21] where a user must declare refinement types for *all* recursive functions including auxiliary functions. Some of the automated verification techniques above require a hint [20], however.

[2] As defined later, a formula $t_1 = t_2$ in a refinement type means that *if both t_1 and t_2 evaluate to (base) values*, then the values are equivalent.

[3] Another way would be to use uninterpreted function symbols, but for that purpose, one would first need to check that `sum` is total.

```
let nil i = None in
let tl xs = fun i-> xs(i+1) in
let cons x xs =
 fun i -> if i=0 then Some(x) else xs(i-1) in
let rec append xs ys =
 match xs(0) with None -> ys
   | Some(x) -> let xs' = tl xs in
                 cons x (append xs' ys)
```

Figure 1. Append function for functional encoding of lists

(where $r = x$ means the extensional equality of functions r and x) but one cannot directly express and verify the same property using first-order refinement types.

To overcome the problems above, we allow[4] programmers to specify (a restricted form of) general refinement types in source programs. For example, they can declare

$$\begin{array}{ll} \texttt{sum2:} & (n:\mathbf{int}) \to \{r:\mathbf{int} \mid r = \mathtt{sum}(n)\} \\ \texttt{append:} & (x:\mathbf{int} \to \mathbf{int\ option}) \to \\ & \{y:\mathbf{int} \to \mathbf{int\ option} \mid y(0) = \mathtt{None}\} \to \\ & \{r:\mathbf{int} \to \mathbf{int\ option} \mid \forall i. r(i) = x(i)\}. \end{array}$$

To take advantage of the recent advance of verification techniques based on first-order refinement types, however, we employ automated program transformation, so that the resulting program can be verified by using only first-order refinement types. The key idea of the transformation is to apply a kind of tupling transformation [5] to capture the relationship between two (or more) function calls at the level of first-order refinement. For example, for the `sum` program above, one can apply the standard tupling transformation (to combine two functions `sum` and `sumacc` into one) and obtain:

```
let rec sum_sumacc (n, m) =
  if n<0 then (0,m) else
  let (r1,r2)=sum_sumacc (n-1, m+n) in
    (r1+n, r2)
```

Checking the equivalence of `sum` and `sum2` then amounts to checking that `sum_sumacc` has the following first-order refinement type:

$$((n, m) : \mathbf{int} \times \mathbf{int}) \to \{(r_1, r_2) : \mathbf{int} \times \mathbf{int} \mid r_2 = r_1 + m\}.$$

The transformation for `append` is more involved: because the return type of the append function refers to the first argument, the append function is modified so that it returns a pair consisting of *the first argument and* the result:

```
let append2 xs ys = (xs, append xs ys).
```

Then, `append2` is further transformed to `append3` below, obtained by replacing `(xs, append xs ys)` with its tupled version.

```
let append3 xs ys (i,j) =
               (xs(i), append xs ys j).
```

The required property `append xs nil = xs` is then verified by checking that `append3` has the following first-order refinement type $\tau_{\mathtt{append3}}$:

$$\begin{array}{l} (x:\mathbf{int} \to \mathbf{int\ option}) \to \\ (y:((x:\mathbf{int}) \to \{r:\mathbf{int\ option} \mid x = 0 \Rightarrow r = \mathtt{None}\})) \to \\ ((i,j):\mathbf{int} \times \mathbf{int}) \to \{(r_1, r_2):\mathbf{int} \times \mathbf{int} \mid i = j \Rightarrow r_1 = r_2\}. \end{array}$$

The transformation sketched above has allowed us to express the *external* behavior of the append function by using first-order refinement types. With the transformation alone, however, the first-order refinement type checking does not succeed: For reasoning about the *internal* behavior of `append`, we need information about the relation between the two function calls `xs(i)` and `append xs ys j`, which cannot be expressed by first order refinement types. As already mentioned, with the restriction to first-order refinement types, the relationship between the return values of the two calls can only be obtained by relating the input/output relations of functions `xs` and `append`. To avoid that limitation, we further transform the program, by inlining `append` and tupling the two calls of the body of `append3`:

```
let append4 xs ys (i,j) =
   match xs(0) with None -> nil2 (i,j)
     | Some(x) ->
       let xs' = tl xs in
       let xszs' = append4 xs' ys in
       let xszs'' = cons2 x x xszs' in
         xszs'' (i,j)
```

Here, `nil2` and `cons2 x x xszs'` are respectively tupled versions of `(nil,nil)` and `(cons x xs', cons x zs')`, where `xszs'` is a tupled one of `xs'` and `zs'`.

At last, it can automatically be proved that `append4` has type $\tau_{\mathtt{append3}}$. (To clarify the ideas, we have over-simplified the transformation above. The actual output of the automatic transformation formalized later is more complicated.)

We formalize the idea sketched above and prove the soundness of the transformation. We also report on a prototype implementation of the approach as an extension to the software model checker MoCHi [10, 14] for a subset of OCaml. The implementation takes a program and its specification (in the form of refinement types) as input, and verifies them automatically (without invariant annotations for auxiliary functions) by applying the above transformations and calling MoCHi as a backend.

The rest of the paper is organized as follows. Section 2 introduces the source language. Section 3 presents the basic transformation for reducing the (restricted form of) general refinement type checking problem to the first-order refinement type checking problem. Roughly, this transformation corresponds to the one from `append` to `append3` above. As mentioned above, the basic transformation alone is not sufficient for automated verification via first-order refinement types; we therefore improve the transformation in Section 4 (which roughly corresponds to the transformation from `append3` to `append4` above). Section 5 reports on experiments and Section 6 discusses related work. We conclude the paper in Section 7.

2. Source Language

This section formalizes the source language and the verification problem.

2.1 Source Language

The source language, used as the target of our verification method, is a simply-typed, call-by-value, higher-order functional language with recursion. The syntax of *terms* is given by:

$$\begin{array}{rl} t \text{ (terms)} ::= & x \mid n \mid \mathtt{op}(t_1, \ldots, t_n) \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\ & \mid \mathbf{fix}(f, \lambda x.\, t) \mid t_1\, t_2 \mid (t_1, \ldots, t_n) \mid \mathbf{pr}_i t \mid \mathbf{fail} \end{array}$$

We use meta-variables $x, y, z, \ldots, f, g, h, \ldots$, and ν for variables. We have only integers as base values, which are denoted by the meta-variable n. The term $\mathtt{op}(\widetilde{t})$ (where $\widetilde{t}$ denotes a sequence of expressions) applies the primitive operation `op` on integers to $\widetilde{t}$. We assume that we have the equality operator $=$ as a primitive operation. We express Booleans by integers, and write $\mathbf{true}$ for

[4] But programmers are not obliged to specify types for all functions. In fact, for the example of `sum2`, no declaration is required for the function `sum`.

$$
\begin{array}{lll}
V \text{ (value)} & ::= n \mid \mathbf{fix}(f, \lambda x.\, t) \mid (V_1, \ldots, V_n) \\
A \text{ (answer)} & ::= V \mid \mathit{fail} \\
E \text{ (eval. ctx.)} & ::= [\,] \mid \mathtt{op}(\widetilde{V}, E, \widetilde{t}) \mid \mathbf{if}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
& \quad \mid E\, t \mid V\, E \mid (\widetilde{V}, E, \widetilde{t}) \mid \mathbf{pr}_i E
\end{array}
$$

$$E[\mathtt{op}(n_1, \ldots, n_k)] \longrightarrow E[[\![\mathtt{op}]\!](n_1, \ldots, n_k)]$$
$$E[\mathbf{fail}] \longrightarrow \mathit{fail}$$
$$E[\mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow E[t_1]$$
$$E[\mathbf{if}\ V\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow E[t_2] (V \neq \mathbf{true})$$
$$E[\mathbf{fix}(f, \lambda x.\, t) V] \longrightarrow E[t[f \mapsto \mathbf{fix}(f, \lambda x.\, t)][x \mapsto V]]$$
$$E[\mathbf{pr}_i(V_1, \ldots, V_n)] \longrightarrow E[V_i]$$

Figure 2. Operational semantics of the source language

1, and $\mathbf{false}$ for 0. The term $\mathbf{fix}(f, \lambda x.\, t)$ denotes the recursive function defined by $f = \lambda x.t$. When f does not occur in t, we write $\lambda x.\, t$ for $\mathbf{fix}(f, \lambda x.\, t)$. The term $t_1 t_2$ applies the function t_1 to t_2. We write $\mathbf{let}\ x = t\ \mathbf{in}\ t'$ for $(\lambda x.t')t$, and write also $t; t'$ for it when x does not occur in t'. The terms $(t_1, \ldots, t_n)$ and $\mathbf{pr}_i t$ respectively construct and destruct tuples. The special term $\mathbf{fail}$ aborts the execution. It is typically used to express assertions; $\mathbf{assert}(t)$, which asserts that t should evaluate to $\mathbf{true}$, is expressed by $\mathbf{if}\ t\ \mathbf{then\ true\ else\ fail}$. We call a closed term a *program*. We often write $\widetilde{t}$ for a sequence $t_1, \ldots, t_n$.

For the sake of simplicity, we assume that tuple constructors occur only in the outermost position or in the argument positions of function calls in source programs. We also assume that all the programs are simply-typed below (where **fail** can have every type).

The small-step semantics is shown in Figure 2. In the figure, $[\![\mathtt{op}]\!]$ is the integer operation denoted by $\mathtt{op}$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$, and $t \longrightarrow^k t'$ if t is reduced to t' in k steps. We write $t \uparrow$ if there is an infinite reduction sequence $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \cdots$. By the assumption that a program is simply-typed, for every program t, either t evaluates to an answer (i.e., $t \longrightarrow^* V$ or $t \longrightarrow^* \mathit{fail}$) or diverges (i.e., $t \uparrow$).

We express the specification of a program by using refinement types. The syntax of refinement types is given by:

$$
\begin{array}{l}
\tau \text{ (types)} ::= \rho \mid \left\{\nu : \prod_{i=1}^n (x_i{:}\, \rho_i) \;\middle|\; P\right\} \\
\rho \text{ (non-tuple types)} ::= \{\nu : \mathbf{int} \mid P\} \mid \{\nu : (x : \tau_1) \to \tau_2 \mid P\} \\
P \text{ (predicates)} ::= t \mid P \wedge P \mid \forall x.P
\end{array}
$$

where we have used a notational convention $\prod_{i=1}^n (x_i{:}\, \rho_i)$ to denote $(x_1{:}\, \rho_1) \times \cdots \times (x_{n-1}{:}\, \rho_{n-1}) \times \rho_n$ (thus, the variable x_n actually does not occur). The type $(x{:}\, \rho_1) \times \rho_2$ is a dependent sum type, where x may occur in ρ_2, and $(x{:}\, \tau_1) \to \tau_2$ is a dependent product type, where x may occur in τ_2. We use a metavariable σ to denote $\mathbf{int}$, $(x{:}\, \tau_1) \to \tau_2$, or $\prod_{i=1}^n (x_i{:}\, \rho_i)$. Intuitively, a *refinement type* $\{\nu : \sigma \mid P\}$ describes a value ν of type σ that satisfies the *refinement predicate* P. For example, $\{\nu : \mathbf{int} \mid \nu > 0\}$ describes a positive integer. The type $\{f : \mathbf{int} \to \mathbf{int} \mid \forall x, y.\, x \leq y \Rightarrow f(x) \leq f(y)\}$ describes a monotonic function on integers.

A refinement predicate P can be constructed from expressions and top-level logical connectives $\forall x$ and $\wedge$, where x ranges over integers. The other logical connectives can be expressed by using expression-level Boolean primitives, but their semantics is subtle due to the presence of effects (non-termination and abort) of expressions, as discussed later in Section 2.2.

We often write just σ for $\{x : \sigma \mid \mathbf{true}\}$; $\tau_1 \to \tau_2$ for $(x : \tau_1) \to \tau_2$, and $\rho_1 \times \rho_2$ for $(x{:}\, \rho_1) \times \rho_2$ if x is not important; τ^m for $\tau \times \cdots \times \tau$ (the m-th power); $\left\{(\nu_i)_{i \in n} : \prod_{i=1}^n (x_i{:}\, \rho_i) \;\middle|\; P\right\}$ for $\left\{\nu : \prod_{i=1}^n (x_i{:}\, \rho_i) \;\middle|\; P[\nu_1 \mapsto \mathbf{pr}_1 \nu, \ldots, \nu_n \mapsto \mathbf{pr}_n \nu]\right\}$; and $\forall \widetilde{x}.P$ for $\forall x_1, \ldots, x_n.P$.

$\boxed{\text{(Predicate)} \models_{\mathrm{p}}^n \subseteq \{P : \text{closed}\}}$

- $\models_{\mathrm{p}}^n \forall x.\, P \stackrel{\text{def}}{\iff} \models_{\mathrm{p}}^n P[x \mapsto m]$ for all integers m
- $\models_{\mathrm{p}}^n P_1 \wedge P_2 \stackrel{\text{def}}{\iff} \models_{\mathrm{p}}^n P_1$ and $\models_{\mathrm{p}}^n P_1$
- $\models_{\mathrm{p}}^n t \stackrel{\text{def}}{\iff} A = \mathbf{true}$ for all A and $k \leq n$ s.t. $t \longrightarrow^k A$

$\boxed{\text{(Value)} \models_{\mathrm{v}}^n, \models_{\mathrm{v}} \subseteq \{V : \text{closed}\} \times \{\tau : \text{closed}\}}$

- $\models_{\mathrm{v}}^n V : \{\nu : \sigma \mid P\} \stackrel{\text{def}}{\iff} \models_{\mathrm{v}}^n V : \sigma$ and $\models_{\mathrm{p}}^n P[\nu \mapsto V]$
- $\models_{\mathrm{v}}^n V : \mathbf{int} \stackrel{\text{def}}{\iff} V = m$ for some integer m
- $\models_{\mathrm{v}}^n V : (x_1 : \tau_1) \to \tau_2 \stackrel{\text{def}}{\iff}$ for all $n' \leq n$ and V_1, $\models_{\mathrm{v}}^{n'} V_1 : \tau_1$ implies $\models_{\mathrm{c}}^{n'} V V_1 : \tau_2[x_1 \mapsto V_1]$
- $\models_{\mathrm{v}}^n (V_1, \ldots, V_n) : \prod_{i=1}^n (x_i : \rho_i) \stackrel{\text{def}}{\iff}$ $\models_{\mathrm{v}}^n V_i : \rho_i[x_1 \mapsto V_1, \ldots, x_{i-1} \mapsto V_{i-1}]$ for all $i \leq n$
- $\models_{\mathrm{v}} V : \tau \stackrel{\text{def}}{\iff} \models_{\mathrm{v}}^n V : \tau$ for all n

$\boxed{\text{(Term)} \models_{\mathrm{c}}, \models_{\mathrm{c}}^n \subseteq \{t : \text{closed}\} \times \{\tau : \text{closed}\}}$

- $\models_{\mathrm{c}}^n t : \tau \stackrel{\text{def}}{\iff} \models_{\mathrm{v}}^{n-k} A : \tau$ for all A and $k \leq n$ s.t. $t \longrightarrow^k A$
- $\models t : \tau \stackrel{\text{def}}{\iff} \models_{\mathrm{c}}^n t : \tau$ for all n
  ($\iff \models_{\mathrm{v}} A : \tau$ for all A s.t. $t \longrightarrow^* A$)

Figure 3. Semantics of types

For a type τ we define the *simple type* $\mathtt{ST}(\tau)$ *of* τ as follows:

$$\mathtt{ST}(\{\nu : \sigma \mid P\}) = \mathtt{ST}(\sigma) \qquad \mathtt{ST}(\mathbf{int}) = \mathbf{int}$$
$$\mathtt{ST}((x : \tau_1) \to \tau_2) = \mathtt{ST}(\tau_1) \to \mathtt{ST}(\tau_2)$$
$$\mathtt{ST}((x : \tau_1) \times \tau_2) = \mathtt{ST}(\tau_1) \times \mathtt{ST}(\tau_2)$$

Also, we define the *order of* τ by:

$$\mathit{order}(\{\nu : \sigma \mid P\}) = \mathit{order}(\sigma) \qquad \mathit{order}(\mathbf{int}) = 0$$
$$\mathit{order}((x : \tau_1) \to \tau_2) = \max(\mathit{order}(\tau_1) + 1, \mathit{order}(\tau_2))$$
$$\mathit{order}((x : \tau_1) \times \tau_2) = \max(\mathit{order}(\tau_1), \mathit{order}(\tau_2)).$$

The syntax of types is subject to the usual scope rule; in $(x{:}\, \rho_1) \times \rho_2$ and $(x{:}\, \tau_1) \to \tau_2$, the scope of x is ρ_2 and τ_2 respectively. Furthermore, we require that every refinement predicate is well-typed and have type $\mathbf{int}$. See Appendix C for the details. To enable the reduction to first-order refinement type checking, we shall further restrict the syntax of types later in Section 2.3.

2.2 Semantics of Refinement Types

The semantics of types is defined in Figure 3, using step-indexed logical relations [1, 2, 6]. Roughly speaking, $\models_{\mathrm{c}}^n t : \tau$ means that t behaves like a term of type τ within n steps computation. For example, $\models_{\mathrm{c}}^n t : \mathbf{int}$ means that if t evaluates to an answer within n steps, then the answer is not *fail* but an integer, (otherwise if t needs more than n steps to evaluate, t may diverge or fail). Also, the condition $\models_{\mathrm{c}}^n t : \mathbf{int} \to \mathbf{int}$ means that if t evaluates to an answer A within n steps, say at k-step ($k \leq n$), then A must be a function, and $\models_{\mathrm{c}}^{n-k} A\, m : \mathbf{int}$ must hold for every integer m, i.e., if $A\, m$ converges to an answer within $n - k$ steps, then the answer is not *fail* but an integer. The connectives $\forall$ and $\wedge$ have genuine logical meaning, and especially they are commutative, so we often use the prenex normal form.

Notice that, by the definition, $\models_{\mathrm{p}}^n t$ holds for every n if t diverges. We write & and || for (expression-level) Boolean conjunction and disjunction. Notice that the semantics of $t_1 \wedge t_2$ and

$$\frac{\Gamma \mid \emptyset \vdash_{\mathtt{WF}} P}{\Gamma \vdash_{\mathtt{WF}} P}\ (\text{WF-PREDINIT}) \qquad \frac{\Gamma \mid \emptyset \vdash_{\mathtt{WF}} \tau}{\Gamma \vdash_{\mathtt{WF}} \tau}\ (\text{WF-INIT})$$

$$\frac{\mathtt{ST}(\mathtt{ElimHO}_0(\Gamma), \mathtt{ElimHO1}_1(\Delta)) \vdash_{\mathtt{S}} t : \mathbf{int}}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} t}\ (\text{WF-PREDTERM})$$

$$\frac{\Gamma \mid \Delta \vdash_{\mathtt{WF}} P_1 \qquad \Gamma \mid \Delta \vdash_{\mathtt{WF}} P_2}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} P_1 \wedge P_2}\ (\text{WF-PREDAND})$$

$$\frac{\Gamma \mid \Delta, y_1 : \mathbf{int}, \dots, y_n : \mathbf{int} \vdash_{\mathtt{WF}} P}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} \forall y_1, \dots, y_n.\, P}\ (\text{WF-PREDFORALL})$$

$$\frac{\Gamma, \Delta \mid \emptyset \vdash_{\mathtt{WF}} \sigma \qquad \Gamma \mid \Delta, x : \sigma \vdash_{\mathtt{WF}} P}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} \{x : \sigma \mid P\}}\ (\text{WF-REFINE}) \qquad \frac{}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} \mathbf{int}}\ (\text{WF-INT})$$

$$\frac{\Gamma, \Delta \mid \emptyset \vdash_{\mathtt{WF}} \tau_1 \qquad \Gamma, \Delta \mid x : \tau_1 \vdash_{\mathtt{WF}} \tau_2}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} (x : \tau_1) \to \tau_2}\ (\text{WF-FUN})$$

$$\frac{\Gamma \mid \Delta \vdash_{\mathtt{WF}} \tau_1 \qquad \Gamma \mid \Delta, x : \tau_1 \vdash_{\mathtt{WF}} \tau_2}{\Gamma \mid \Delta \vdash_{\mathtt{WF}} (x : \tau_1) \times \tau_2}\ (\text{WF-PAIR})$$

$$\frac{}{\vdash_{\mathtt{WF}} \emptyset}\ (\text{WF-ENIL}) \qquad \frac{\vdash_{\mathtt{WF}} \Gamma \qquad \Gamma \vdash_{\mathtt{WF}} \tau \qquad (x : \_) \notin \Gamma}{\vdash_{\mathtt{WF}} \Gamma, x : \tau}\ (\text{WF-ECONS})$$

$$\mathtt{ElimHO}_n(\Gamma) \stackrel{\text{def}}{=} \{(x : \tau) \in \Gamma \mid \mathit{depth}(\tau) <= n\}$$

$$\Gamma, \Delta ::= \emptyset \mid \Gamma, x : \tau$$

Figure 4. Well-formedness of types

t_1 & t_2 are different. For example, let Ω be a divergent term. Then $\models 1 : \{x : \mathbf{int} \mid \Omega \wedge x = 0\}$ does NOT hold, but $\models 1 : \{x : \mathbf{int} \mid \Omega \,\&\, x = 0\}$ DOES hold, since $\Omega \,\&\, x = 0$ diverges.

The goal of our verification is to check whether $\models t : \tau$ holds, given a program t and a type τ. Since the verification problem is undecidable, we aim to develop a sound but incomplete method below. As explained in Section 1, our approach is to use program transformation to reduce the (semantic) type checking problem $\models t : \tau$ to the first-order refinement type checking problem $\models t' : \tau'$ where τ' does not contain any function variables in refinement predicates, and to check $\models t' : \tau'$ using an automated verification tool such as MoCHi [10, 14, 19], which combines higher-order model checking [9] and predicate abstraction.

2.3 Restriction on Refinement Types

To enable the reduction of the refinement type checking problem $\models t : \tau$ to the first-order one $\models t' : \tau'$, we have to impose some restrictions on the type τ. The most important restriction is that only first-order function variables (i.e., functions whose simple types are of the form $\mathbf{int} \times \cdots \times \mathbf{int} \to \mathbf{int} \times \cdots \times \mathbf{int}$) may be used in refinement predicates. The other restrictions are rather technical. We describe below the details of the restrictions, but they may be skipped for the first reading.

1. We assume that every closed type τ satisfies the well-formedness condition $\emptyset \vdash_{\mathtt{WF}} \tau$ defined in Figure 4. In the figure, $\mathtt{ElimHO}_n(\Gamma)$ filters out all the bindings of types whose *depth* are greater than n, where the depth of a type is defined by:

$$\begin{aligned}&\mathit{depth}(\{\nu : \sigma \mid P\}) = \mathit{depth}(\sigma) \qquad \mathit{depth}(\mathbf{int}) = 0\\ &\mathit{depth}((x : \tau_1) \to \tau_2) = 1 + \max\{\mathit{depth}(\tau_1), \mathit{depth}(\tau_2)\}\\ &\mathit{depth}((x : \tau_1) \times \tau_2) = \max\{\mathit{depth}(\tau_1), \mathit{depth}(\tau_2)\}.\end{aligned}$$

In addition to the usual scope rules and well-typedness conditions of refinement predicates (that have been explained already in Section 2.1), the rules ensure that (i) only depth-1 function variables (i.e., variables of types whose depth is 1) may occur in refinement predicates, (ii) in a type of the form $(x : \tau_1) \to \{\nu : \sigma \mid P\}$ where τ_1 is a depth-1 function type, x may occur in P but not in σ (there is no such restriction if τ_1 is a depth-0 type), and (iii) in a type of the form $(f_1 : \tau_1) \times \{f_2 : \sigma_2 \mid P_2\} \times \cdots \times \{f_n : \sigma_n \mid P_n\}$, f_1 may occur in $P_2, \dots, P_n$ but not in $\sigma_2, \dots, \sigma_n$.

2. In a refinement predicate $\forall x_1, \dots, x_n.\, \wedge_j\, t_j$, for every t_j, if x_i occurs in t_j, there must be an occurrence of application of the form $f(\dots, x_i, \dots)$. Also, for every t_j, if a function variable f occurs, every occurrence must be as an application $f t$.

3. The special primitive **fail** must not occur in any refinement predicate. Also, in every application $t_1 t_2$ in a refinement predicate, t_2 must not contain function applications nor **fail**. (In other words, t_2 must be *effect-free*, in the sense that it neither diverges nor fail.)

4. Abstractions (i.e., $\mathbf{fix}(f, \lambda x.\, t)$) must not occur in refinement predicates, except in the form $\mathbf{let}\ x = t\ \mathbf{in}\ t'$.

5. In refinement predicates, usual if-expressions are not allowed; instead we allow "branch-strict" if-expression $\mathbf{ifs}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$ where t_1 and t_2 are both evaluated before the evaluation of t. This is equivalent to $t_1; t_2; \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$; hence, in other words, we allow if-expressions only in this form.

Please note that the above restrictions are essential only for the refinement predicates that occur in σ of a given type checking problem $\stackrel{?}{\models} t : \{\nu : \sigma \mid P\}$ rather than the top level refinement P; since given

$$\stackrel{?}{\models} t : \{\nu : \sigma \mid \forall \widetilde{x}.\, \wedge_i\, t_i\}$$

where $\forall \widetilde{x}.\, \wedge_i\, t_i$ does not satisfy the restrictions above, we can replace it by an equivalent problem

$$\stackrel{?}{\models} \mathbf{let}\ \nu = t\ \mathbf{in}\ (\nu, (\lambda \widetilde{x}.\, t_i)_i) : \sigma \times \prod_i (\mathbf{int}^n \to \{r : \mathbf{int} \mid r\}).$$

Remark 1. As in the case above, there is often a way to avoid the restrictions 1–5 listed above. A more fundamental restriction (besides the restriction that only first-order function variables may be used in refinement predicates), which is imposed by the syntax of refinement predicates defined in Section 2.1, is that existential quantifiers cannot be used. Due to the restriction, we cannot express the type:

$$\begin{aligned}&n : \mathbf{int} \to \{f : \mathbf{int} \to \mathbf{int} \mid \exists x. 1 \le x \le n \wedge f(x) = 0\}\\ &\to \{\nu : \mathbf{int} \mid \nu = 1\},\end{aligned}$$

which describes a higher-order function that takes an integer n and a function f, and returns 1 if there exists a value x such that $1 \le x \le n \wedge f(x) = 0$. This is a typical specification for a search function.

3. Encoding Functional Refinement

In this section, we present a transformation $(-)^\sharp$ for reducing a general refinement type checking problem to the first-order refinement type checking problem. In the rest of the paper, we use the assumptions explained in Section 2.1.

We first explain the ideas of the transformation $(-)^\sharp$ informally in Section 3.1. We give the formal definition of the transformation in Section 3.2. Finally in Section 3.3, we show the soundness of our verification method that uses $(-)^\sharp$.

3.1 Idea of the Transformation

The transformation $(-)^\sharp$ is in fact the composition of four transformations: $((((-)^{\sharp_1})^{\sharp_2})^{\sharp_3})^{\sharp_4}$. We explain the idea of each transformation from $(-)^{\sharp_4}$ to $(-)^{\sharp_1}$ in the reverse order of the applications, since $(-)^{\sharp_4}$ is the key step and the other ones perform preprocessing to enable the transformation $(-)^{\sharp_4}$.

$\sharp_4$: *Elimination of universal quantifiers and function symbols from a refinement predicate*
We first discuss a simple case, where there occur only one universal quantifier and one function symbol in a refinement predicate. Consider a refinement type of the form

$$\{f : \mathbf{int} \to \mathbf{int} \mid \forall x.\ P[f\,x]\}$$

where $P[f\,x]$ contains just one occurrence of $f\,x$ and no other occurrences of function variables. It can be encoded into the first-order refinement type

$$(x : \mathbf{int}) \to \{r : \mathbf{int} \mid P[r]\}.$$

By the semantics of types, the latter type means that, *for all argument* x, its "return value" r (i.e., $f\,x$) satisfies $P[r]$. The application $f\,x$ in the former type is expressed by the refinement variable r of the return value type, and the original quantifier $\forall x$ is encoded by the function type, or more precisely, "for all" in the semantics of the function type.

Now, let us consider a more general case where multiple function symbols occur. Given the type checking problem

$$\overset{?}{\models} (t_1, t_2)\ :\ \big\{(f, g) : \big(\tau_1 \to \tau_1'\big) \times \big(\tau_2 \to \tau_2'\big) \,\big|\, \forall x_1, x_2.\ P[f\,x_1, g\,x_2]\big\}$$

where each of the two different function variables occurs once in $P[f\,x_1, g\,x_2]$, we can transform it to:

$$\mathbf{let}\ f = t_1\ \mathbf{in}\ \mathbf{let}\ g = t_2\ \mathbf{in}\ \lambda(x_1, x_2).\,(f\,x_1, g\,x_2)\ :\ ((x_1, x_2) : \tau_1 \times \tau_2) \to \big\{(r_1, r_2) : \tau_1' \times \tau_2' \,\big|\, P[r_1, r_2]\big\}.$$

As in the case above for a single function occurrence, the transformation preserves the validity of the judgment.

To apply the transformation above, the following conditions on the refinement predicate (the part $\forall x_1, x_2.\ P[f\,x_1, g\,x_2]$ above) are required. (i) all the occurrences of function variables (f and g) are distinct from each other (ii) function arguments (x_1 and x_2 above) are variables rather than arbitrary terms, and they are distinct from each other, and universally quantified (iii) function variables f and g in a predicate P in $\{\nu : \sigma \mid P\}$ are declared at the position of ν. Those conditions are achieved by the preprocessing $(-)^{\sharp_3}$, $(-)^{\sharp_2}$, and $(-)^{\sharp_1}$ explained below.

$\sharp_3$ *Replication of functions*
If a function variable occurs n (> 1) times in a refinement predicate, we replicate the function and make a tuple consisting of n copies of the function. For example, for a typing

$$t : \{f : \mathbf{int} \to \mathbf{int} \mid P[f\,x, f\,y]\}$$

where f occurs exactly twice, we transform this to

$$\mathbf{let}\ f = t\ \mathbf{in}\ (f, f) :\ \big\{(f_1, f_2) : (\mathbf{int} \to \mathbf{int})^2 \,\big|\, P[f_1\,x, f_2\,y]\big\}\,,$$

so that each of the function variables f_1 and f_2 now occurs just once in the refinement predicate.

$\sharp_2$ *Normalization of function arguments in refinement predicates*
In this step, we ensure that all the function arguments in refinement predicates are variables, different from each other, and quantified universally.

Given a type of the form:

$$\{f : \mathbf{int} \to \mathbf{int} \mid \forall \widetilde{x}.\ P[f\,t]\}$$

where $P[-]$ is a context with one occurrence of the hole $[\,]$ and t is either a non-variable, or a quantified variable $x_i \in \{\widetilde{x}\}$ but there is another occurrence of x_i, we transform this to

$$\{f : \mathbf{int} \to \mathbf{int} \mid \forall \widetilde{x}, y.\ y = t \texttt{ => } P[f\,y]\}$$

where y is a fresh variable.

Recall that `=>` is an expression-level Boolean primitive. Thus, the transformation above preserves the semantics of types only if t is effect-free; this is guaranteed by Assumption (iv) in Section 2.3.

$\sharp_1$ *Removal of dependencies between functional arguments and return types*
In Step $\sharp_4$ above, we assumed "(iii) function variables ... in a predicate P in $\{\nu : \sigma \mid P\}$ are declared at the position of ν"; this can be relaxed so that a function variable in P may be bound at the position of f in $(f\!:\tau) \to \{\nu : \sigma \mid P\}$ as described below. A judgment

$$\overset{?}{\models} t : (f : \tau_1 \to \tau_2) \to \{\nu : \tau \mid P\}$$

can be transformed to

$$\overset{?}{\models} \mathbf{let}\ g = t\ \mathbf{in}\ \lambda f'.\,(f',\ g\,f') :\ (f\!:\tau_1 \!\to\! \tau_2) \to \big\{(f', \nu)\!: \big(f'\!:\tau_1 \to \tau_2\big) \times \tau \mid P[f \mapsto f']\big\}$$

where the function variable f' is fresh. Here, the function argument has been copied and attached to the return value, so that P may refer to the original argument.

In Section 1, $(-)^{\sharp_1}$ has been used for the example of `append2`. We now demonstrate uses of $(-)^{\sharp_2}$ and $(-)^{\sharp_4}$ with the other example in Section 1:

$$\overset{?}{\models} (\texttt{sum}, \texttt{sum2})\ :\ (f : \mathbf{int} \to \mathbf{int}) \times \{g : \mathbf{int} \to \mathbf{int} \mid \forall n.\, g(n) = f(n)\}\,.$$

The refinement predicate is transformed by $(-)^{\sharp_2}$ to

$$\forall n, n_1, n_2.\ n_1 = n \texttt{ => } n_2 = n \texttt{ => } g(n_1) = f(n_2),$$

which is equivalent to

$$\forall n_1, n_2.\ n_1 = n_2 \texttt{ => } g(n_1) = f(n_2).$$

By $(-)^{\sharp_4}$, the above type checking problem is reduced to the following one:

$$\overset{?}{\models} \lambda(n_1, n_2).\,(\texttt{sum}\,n_1, \texttt{sum2}\,n_2)\ :\ \big((n_1, n_2) : \mathbf{int}^2\big) \to \big\{(r_1, r_2) : \mathbf{int}^2 \,\big|\, n_1 = n_2 \texttt{ => } r_2 = r_1\big\}.$$

One may notice that the result of the transformation above is different from that of `sum` and `sumacc` in Section 1, which is obtained by applying a further transformation explained in Section 4.

3.2 Transformations

We give formal definitions of the transformations $(-)^{\sharp_1}$, $(-)^{\sharp_2}$, $(-)^{\sharp_3}$, and $(-)^{\sharp_4}$ in this order.

For the sake of simplicity, w.l.o.g., we assume that every term has a type of the following form:

$$\tau ::= \Big\{\ \nu : \prod_{i=1}^{n} x_i\!: \mathbf{int} \times \prod_{j=1}^{m} \big(f_j\!:\ (y_j\!: \tau_j) \to \tau_j'\big)\ \Big|\ P\ \Big\}.$$

In fact, any type (and accordingly terms of that type) can be transformed to the above form: e.g.,

$$\big\{(f, x) : \big(f\!:\ \big\{f : \tau \!\to\! \tau' \,\big|\, P_1\big\}\big) \times \{x : \mathbf{int} \mid P_2\} \,\big|\, P\big\}$$

$$\Big(\Big\{\, \nu : \prod_{i=1}^{n} x_i\colon \mathbf{int} \times \prod_{j=1}^{m} f_j\colon ((y_j{:}\tau_j) \to \tau'_j) \;\Big|\; P \,\Big\}\Big)^{\sharp_1} \stackrel{\text{def}}{=}$$

$$\Big\{\nu : \prod_{i=1}^{n} x_i\colon \mathbf{int} \times \prod_{j=1}^{m} \big(f_j\colon (y_j{:}\tau_j) \to \tau'_j\big)^{\sharp_1} \Big|\; P\Big\}$$

$$\big(((y_k)_k{:}\tau) \to \tau'\big)^{\sharp_1} \stackrel{\text{def}}{=}$$

$$\big((y_k)_k\colon (\tau)^{\sharp_1}\,\big) \to \big((y'_k)_{k \in D(1)} : \tau^{(1)}\big) \times \big(\,(\tau')^{\sharp_1}\, [y_k \mapsto y'_k]_{k \in D(1)}\big)$$

where, for the type $\tau = \big\{(y_k)_k : \prod_k (y_k\colon \rho_k) \;\big|\; P\big\}$,

$$D(1) \stackrel{\text{def}}{=} \{k \mid \rho_k \text{ is depth-1}\}$$

$$\tau^{(1)} \stackrel{\text{def}}{=} \Big\{(y_k)_{k \in D(1)} : \textstyle\prod_{k \in D(1)} (y_k\colon \rho_k) \;\Big|\; P\Big\}$$

Note that $(\tau)^{\sharp_1} = \tau$ if τ is at most order-1; hence we have the obvious projection $\mathbf{pr}^{(1)} : (\tau)^{\sharp_1} \to \tau^{(1)}$, which is used below.

$$(\mathbf{fix}(f, \lambda x.\, t))^{\sharp_1} \stackrel{\text{def}}{=} \mathbf{fix}(f, \lambda x.\, (\mathbf{pr}^{(1)} x, (t)^{\sharp_1}))$$

$$(t_1\, t_2)^{\sharp_1} \stackrel{\text{def}}{=} \mathbf{pr}_2((t_1)^{\sharp_1}\, (t_2)^{\sharp_1})$$

Figure 5. Returning Input Functions $(-)^{\sharp_1}$

$$\Big(\Big\{\, \nu\colon \prod_{i=1}^{n} (x_i{:}\mathbf{int}) \times \prod_{j=1}^{m} \big(f_j\colon (y_j{:}\tau_j) \to \tau'_j\big) \;\Big|\; P \,\Big\}\Big)^{\sharp_2} \stackrel{\text{def}}{=}$$

$$\Big\{\, \nu\colon \prod_{i=1}^{n} (x_i{:}\mathbf{int}) \times \prod_{j=1}^{m} \Big(f_j\colon \Big(y_j\colon (\tau_j)^{\sharp_2}\Big) \to \big(\tau'_j\big)^{\sharp_2}\Big) \;\Big|\; (P)^{\sharp_2} \,\Big\}$$

$$(\forall x_1, \ldots, x_n.\, \wedge_k\, t_k)^{\sharp_2}$$

$$\stackrel{\text{def}}{=} \mathtt{eOQ}\big(\forall \widetilde{x_i}.\forall \widetilde{z_{k,l}}\, \wedge_k\, \big(\mathtt{argEq}(\mathtt{app}(t_k)) \Rightarrow \mathtt{sArg}(t_k)\big)\big)$$

$$\stackrel{\text{def}}{=} \forall \widetilde{z_{k,l}}\, \wedge_k\, \big(\big(\mathtt{argEq}(\mathtt{app}(t_k)) \Rightarrow \mathtt{sArg}(t_k)\big)[x_i \mapsto z_k^i]\big)$$

where $\mathtt{sArg}$ and $\mathtt{argEq}$ are defined as below, and the variables $z_{k,1}, \ldots, z_{k,m_k}$ are all the elements of $\{\, z^{\langle (f\,t)^i \rangle} \mid (f\,t)^i \in \mathtt{app}(t_k)\,\}$.

$$\mathtt{sArg}((f\,t)^i) \stackrel{\text{def}}{=} f\, z^{\langle (f\,t)^i \rangle}$$

$\mathtt{sArg}(t^i)$ is defined compositionally when t is not an application

$$\mathtt{argEq}(\{a_1, \ldots, a_m\}) \stackrel{\text{def}}{=} \mathtt{argEq}(\{a_1\}) \,\&\, \cdots \,\&\, \mathtt{argEq}(\{a_m\})$$

$$\mathtt{argEq}(\{(f\,t)^i\}) \stackrel{\text{def}}{=} (z^{\langle (f\,t)^i \rangle} = \mathtt{sArg}(t))$$

Figure 6. Normalization of function arguments $(-)^{\sharp_2}$

can be transformed to

$$\big\{(x, f) : \mathbf{int} \times (\tau{\to}\tau') \;\big|\; P_1 \wedge P_2 \wedge P\big\}\,.$$

(The logical connective $\wedge$ was introduced as a primitive in Section 2 for this purpose.) For an expression t of the above type, we write $\mathbf{pr}_i^{\mathbf{int}}(t)$ to refer to the i-th integer (i.e., x_i), and $\mathbf{pr}_j^{\to}(t)$ to refer to the j-th function (i.e., f_j). The operators $\mathbf{pr}_i^{\mathbf{int}}$ and $\mathbf{pr}_j^{\to}$ can be expressed by compositions of the primitive $\mathbf{pr}_i$ in Section 2.1. Inside the refinement predicate P above, we sometimes write x_i and f_j to denote $\mathbf{pr}_i^{\mathbf{int}}\nu$ and $\mathbf{pr}_j^{\to}\nu$ respectively.

$\sharp_1$: Removal of Dependencies between Functional Arguments and Return Types

Figure 5 shows the key cases of the definition of the transformation $(-)^{\sharp_1}$ for types and terms. For types, $(-)^{\sharp_1}$ copies (the depth-1 components of) the argument type of a function type to the return type. For example, a refinement type of the form

$$((x, f)\colon \mathbf{int}{\times}(\mathbf{int}{\to}\mathbf{int})) \to \{r\colon \sigma \mid P(r, x, f)\}$$

is transformed to a type of the form

$$((x, f)\colon \mathbf{int}{\times}(\mathbf{int}{\to}\mathbf{int})){\to}(f'\colon (\mathbf{int}{\to}\mathbf{int})){\times}\{r\colon \sigma \mid P(r, x, f')\}.$$

Note that the return type no longer depends on the argument f.

As for the term transformation, in the rule for $\mathbf{fix}(f, \lambda x.\, t)$, (the depth-1 components of) the argument x is added to the return value. In the rule for $t_1 t_2$, $(t_1)^{\sharp_1}\, (t_2)^{\sharp_1}$ returns a pair of (the depth-1 components of) the value of t_2 and the value of $t_1 t_2$; therefore, we extract such the value of $t_1 t_2$ by applying the projection. For example, the term $\mathbf{fix}(f, \lambda x.\, f\, x)$ is transformed to $\mathbf{fix}(f, \lambda x.\, (\mathbf{pr}^{(1)} x, \mathbf{pr}_2(f\, x)))$.

After the transformation $(-)^{\sharp_1}$, the type of the program satisfies a more restricted well-formedness condition, obtained by replacing all judgments $\Gamma \mid \Delta \vdash_{\mathtt{WF}} P$ in Figure 4 with $\Gamma, \Delta \mid\; \vdash_{\mathtt{WF}} P$.

$\sharp_2$: Normalization of Function Arguments in Refinement Predicates

Figure 6 defines the transformation $(-)^{\sharp_2}$. In the figure, & is an expression-level Boolean conjunction, and $\wedge_k\, t_k$ abbreviates $t_1 \wedge \cdots \wedge t_k$. For each occurrence of application $(f\,t')^i$ in P (where i denotes its position in P, used to discriminate between multiple occurrences of the same term $f\,t'$; i is omitted if it is clear), we prepare a fresh variable $z^{\langle (f\,t')^i \rangle}$; for an occurrence of a term t^i in P, $\mathtt{app}(t^i)$ is the set of occurrences of applications in t^i; $\mathtt{sArg}(t^i)$ is the term obtained by replacing the argument t' of each $(f\,t')^i \in \mathtt{app}(t^i)$ with $z^{\langle (f\,t')^i \rangle}$; and $\mathtt{argEq}(-)$ equates such t' and $z^{\langle (f\,t')^i \rangle}$. In the figure, $\mathtt{eOQ}(-)$ eliminates the original quantifiers $\forall \widetilde{x_i}$ as follows: by the assumption 2 in Section 2.3, for each i and k, if x_i occurs in t_k, then x_i occurs at least once as the argument of an application, and so there is some z_k^i such that $(z_k^i = x_i) \in \mathtt{argEq}(t_k)$; hence $\forall x_i$ can be eliminated by substituting z_k^i for x_i.

For example, consider the type

$$\{(f, g)\colon (\mathbf{int} \to \mathbf{int})^2 \mid \forall x.\, f\,x = g\,x\}.$$

Let t be $(f\,x = g\,x)$ and P be $\forall x.\, t$, then

$$\begin{aligned}
&\mathtt{app}(t) = \{f\,x, g\,x\}\,,\\
&\mathtt{argEq}(\mathtt{app}(t)) = \mathtt{argEq}(f\,x) \,\&\, \mathtt{argEq}(g\,x)\\
&\qquad = (z^{\langle fx \rangle} = \mathtt{sArg}(x)) \,\&\, (z^{\langle gx \rangle} = \mathtt{sArg}(x))\\
&\qquad = (z^{\langle fx \rangle} = x) \,\&\, (z^{\langle gx \rangle} = x),\\
&\mathtt{sArg}(f\,x = g\,x) = (f\,z^{\langle fx \rangle} = g\,z^{\langle gx \rangle}),
\end{aligned}$$

and the transformed predicate before $\mathtt{eOQ}(-)$ is

$$\forall x, z^{\langle fx \rangle}, z^{\langle gx \rangle}.\; z^{\langle fx \rangle} = x \;\;\&\;\; z^{\langle gx \rangle} = x \;\; \texttt{=>} \;\; f\,z^{\langle fx \rangle} = g\,z^{\langle gx \rangle}.$$

By applying $\mathtt{eOQ}(-)$, we obtain:

$$\forall z^{\langle fx \rangle}, z^{\langle gx \rangle}.\; z^{\langle fx \rangle} = z^{\langle fx \rangle} \,\&\, z^{\langle gx \rangle} = z^{\langle fx \rangle} \;\texttt{=>}\; f\,z^{\langle fx \rangle} = g\,z^{\langle gx \rangle},$$

which may be simplified further to

$$\forall z^{\langle fx \rangle}, z^{\langle gx \rangle}.\; z^{\langle fx \rangle} = z^{\langle gx \rangle} \;\;\texttt{=>}\;\; f\,z^{\langle fx \rangle} = g\,z^{\langle gx \rangle}.$$

$\sharp_3$: Replication of Functions

As explained in Section 3.1, $(-)^{\sharp_3}$ replicates a function f_j according to the number m_j of occurrences of f_j in the predicate P of a

$$\left(\left\{\nu : \prod_{i=1}^{n}(x_i\colon \mathbf{int}) \times \prod_{j=1}^{m}\left(f_j\colon \tau_j \to \tau'_j\right) \;\middle|\; P\right\}\right)^{\sharp_3}_{\phi} \stackrel{\mathrm{def}}{=}$$

$$\left\{\nu : \prod_{i=1}^{n}(x_i\colon \mathbf{int}) \times \prod_{j=1}^{m}\prod_{l=1}^{m_j}(f_{j,l}\colon (\tau_j)^{\sharp_3}_{\phi_j} \to (\tau'_j)^{\sharp_3}_{\phi'_j})) \;\middle|\; P'\right\}$$

where, $\phi = \{\!\!\{\prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{m}(\phi_j \to \phi'_j) \mid M\}\!\!\}$; $m_j = M(j)$; let $a_{j,1}, \ldots, a_{j,m'_j}$ be all the occurrences of applications of f_j occurring in P and let m'_j be $\mathit{mul}(P,j)$ ($m'_j \leq m_j$ since $\tau \leq_{\mathit{mul}} \phi$); and

$$P' \stackrel{\mathrm{def}}{=} P[a_{j,l} \mapsto f_{j,l}\, t_{j,l}]_{j\in\{1,\ldots,m\}, l\in\{1,\ldots,m'_j\}}$$

(where $a_{j,l} = f_j\, t_{j,l}$)

$$(\mathbf{fix}(f, \lambda x.\, t))^{\sharp_3}_{T} \stackrel{\mathrm{def}}{=} \overrightarrow{\mathbf{fix}(f, \lambda x.\, (t)^{\sharp_3}_{T}\, [f \mapsto \overrightarrow{f}^{m}])}^{m}$$

$(m = T(\mathbf{fix}(f, \lambda x.\, t))$ and $\overrightarrow{t}^{m} = (\underbrace{t, \ldots, t}_{m})$ for a term $t)$

$$(t_1\, t_2)^{\sharp_3}_{T} \stackrel{\mathrm{def}}{=} \left(\mathbf{pr}_1 (t_1)^{\sharp_3}_{T}\right)(t_2)^{\sharp_3}_{T}$$

Figure 7. Replication of functions $(-)^{\sharp_3}$

refinement type $\tau = \left\{\nu : \prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{\ell}\left(f_j\colon \tau_j \to \tau'_j\right) \;\middle|\; P\right\}$; we call m_j *the multiplicity of* f_j and write $\mathit{mul}(\tau, j)$ or $\mathit{mul}(P, j)$. We call the sequence $(m_j)_j = m_1 \cdots m_\ell$ *the multiplicity of* τ.

The transformation $(t)^{\sharp_3}$ is parameterized by a *multiplicity type* ϕ for types, and a *multiplicity annotation* T for terms. The multiplicity types are defined by the following grammar:

$$\phi ::= \{\!\!\{\prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{m}(\phi_j \to \phi'_j) \mid M\}\!\!\}$$

Here, M is a function from $\{1, \ldots, m\}$ to positive integers such that $M(j) = 1$ if $\phi_j \to \phi'_j$ is not depth-1. Intuitively, $M(j)$ denotes how many copies should be prepared for the j-th function (of type $\phi_j \to \phi'_j$). For a refinement type $\tau = \left\{\nu : \prod_{i=1}^{n}(x_i\colon \mathbf{int}) \times \prod_{j=1}^{m}\left(f_j\colon \tau_j \to \tau'_j\right) \;\middle|\; P\right\}$ and a multiplicity type $\phi = \{\!\!\{\prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{m}(\phi_j \to \phi'_j) \mid M\}\!\!\}$, we write $\tau \leq_{\mathit{mul}} \phi$ if all the multiplicities in τ are pointwise less than or equal to those in ϕ, i.e., if $\mathit{mul}(P, j) \leq M(j)$, $\tau_j \leq_{\mathit{mul}} \phi_j$, and $\tau'_j \leq_{\mathit{mul}} \phi'_j$ for all j. Intuitively, $\tau \leq_{\mathit{mul}} \phi$ means that copying functions according to ϕ is sufficient for keeping track of the correlations between functions expressed by τ. Thus, in the transformation rule for types in Figure 7, we assume that $\tau \leq_{\mathit{mul}} \phi$, and replicate each function type according to ϕ.

The multiplicity annotation T used in the transformation of terms maps each (occurrence of) subterm to its *multiplicity*. Here, if a subterm has simple type $\mathbf{int}^n \times \prod_{j=1}^{\ell}(\tau_j \to \tau'_j)$, then its *multiplicity* is a sequence $m_1 \cdots m_\ell$ of positive integers. In the case for abstractions, as explained in Section 3.1, a function $\mathbf{fix}(f, \lambda x.\, t)$ is copied to an m-tupled function where m is the multiplicity of $\mathbf{fix}(f, \lambda x.\, t)$. In the case for applications, correspondingly to the case for abstractions, the function t_1 is replaced with its m-copies; after that we have to insert projection $\mathbf{pr}_1$ for matching types correctly.

Given a type checking problem $\stackrel{?}{\models} t : \tau$, we infer ϕ and T *automatically* (so that the transformation $(-)^{\sharp_3}$ is fully automatic). For multiplicity types, we can choose the least ϕ such that $\tau \leq_{\mathit{mul}} \phi$, and determine $T(t)$ according to ϕ. For some subterms, however, their multiplicity annotations are not determined by τ; for example, if $t = t_1 t_2$, then the multiplicity of t_2 depends on the refinement type of t_2 used for concluding $\models t_1\, t_2 : \tau$. For such a subterm t', we just infer the value of $T(t')$. Fortunately, as long as ϕ and T satisfy a certain consistency condition (for example, in $\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$, it should be the case that $T(t_1) = T(t_2)$), the transformation is sound (see Section 3.3). Since larger ϕ and T are more costly but allow us to keep track of the relationship among a larger number of more function calls (for example, if $T(f) = 2$, then we can keep track of the relationship between two function calls of f; that is sufficient for reasoning about the monotonicity of f), in the actual verification algorithm, we start with minimal consistent ϕ and T, and gradually increase them until the verification succeeds.

$$\left(\left\{\nu : \prod_{i=1}^{n}(x_i\colon \mathbf{int}) \times \prod_{j=1}^{m}\left(f_j\colon (y_j\colon \tau_j) \to \tau'_j\right) \mid P\right\}\right)^{\sharp_4} \stackrel{\mathrm{def}}{=}$$

$$\prod_{i=1}^{n}(x_i\colon \mathbf{int}) \times \left(\begin{array}{l} \left((y_j)_j\colon \prod_{j=1}^{m}\left((\tau_j)^{\sharp_4}\right)_{\perp}\right) \\ \to \left\{(r_j)_j\colon \prod_{j=1}^{m}\left((\tau'_j)^{\sharp_4}\right)_{\perp} \;\middle|\; (P)^{\sharp_4}\right\} \end{array}\right)$$

where, let $a_1, \ldots, a_{m'}$ be all the occurrences of applications in P, then, for $P = \forall z_1, \ldots, z_{m'}.\, \wedge_k t_k$,

$$(P)^{\sharp_4} \stackrel{\mathrm{def}}{=} ((\wedge_k t_k)[a_l \mapsto r_{\hat{\jmath}(a_l)}]_{l\in m'})[\hat{z}^{(a_l)} \mapsto y_{\hat{\jmath}(a_l)}]_{l\in m'}.$$

$$((t_1, \ldots, t_n, t'_1, \ldots, t'_m))^{\sharp_4} \stackrel{\mathrm{def}}{=}$$

$\mathbf{let}\ x_1 = (t_1)^{\sharp_4}\ \mathbf{in} \cdots \mathbf{let}\ x_n = (t_n)^{\sharp_4}\ \mathbf{in}$

$\mathbf{let}\ f_1 = (t'_1)^{\sharp_4}\ \mathbf{in} \cdots \mathbf{let}\ f_m = (t'_m)^{\sharp_4}\ \mathbf{in}$

$(x_1, \ldots, x_n, \lambda y.\, (f_1\, (\mathbf{pr}_1 y), \ldots, f_m\, (\mathbf{pr}_m y)))$

where t_i are integers and t'_i are functions.

$$\left(\mathbf{pr}^{\mathbf{int}}_i t\right)^{\sharp_4} \stackrel{\mathrm{def}}{=} \mathbf{pr}_i\, (t)^{\sharp_4}$$

$$\left(\mathbf{pr}^{\to}_j t\right)^{\sharp_4} \stackrel{\mathrm{def}}{=} \mathbf{let}\ w = (t)^{\sharp_4}\ \mathbf{in}\ t'$$

where $t' \stackrel{\mathrm{def}}{=} \lambda y.\, \mathbf{pr}_j((\mathbf{pr}_{n+1} w)\, \underbrace{(\overbrace{\perp, \ldots, \perp}^{j-1}, y, \perp, \ldots, \perp)}_{m})$

and n and m are the numbers of the integer components and the function type components in the simple type of t, respectively.

Figure 8. Elimination of universal quantifiers and function symbols from a refinement predicate $(-)^{\sharp_4}$

$\sharp_4$: Elimination of Universal Quantifier and Function Symbols

Figure 8 defines the transformation $(-)^{\sharp_4}$. For a type τ, we write $(\tau)_{\perp}$ for the *option type* $\tau + 1$; we explain this later.

For the transformation of refinement predicates, we use the functions $\hat{\jmath}(-)$ and $\hat{z}^{(-)}$ defined as follows. For an input type $\{((x_i)_{i\leq n}, (f_j)_{j\leq m}) : \ldots \mid P\}$ of $(-)^{\sharp_4}$, we can assume that by $(-)^{\sharp_1}$, function symbols occurring in a refinement predicate are in $\{f_j \mid j \leq m\}$; and that by $(-)^{\sharp_2}$ and $(-)^{\sharp_3}$, all application occurrences in P have distinct function variables, and have distinct argument variables that quantified universally. Thus, there is an injection $\hat{\jmath}(-)$ from the set X of occurrences of applications in P to $\{j \mid j \leq m\}$ such that for any application occurrence ft, $f = f_{\hat{\jmath}(ft)}$; and also there is a bijection $\hat{z}^{(-)}$ from the same set X to the set of the variables that are universally in P.

For example, let us continue the example used for $\sharp_2$:

$\{\ (f, g)\colon (\mathbf{int} \to \mathbf{int})^2 \mid$

$\quad \forall z^{\langle fx\rangle}, z^{\langle gx\rangle}.\ z^{\langle fx\rangle} = z^{\langle gx\rangle}\ \texttt{=>}\ f\, z^{\langle fx\rangle} = g\, z^{\langle gx\rangle}\ \}.$

The transformed type is of the form

$$\big((y_1, y_2)\colon (\mathbf{int})^2_\perp\big) \to \Big\{(r_1, r_2)\colon (\mathbf{int})^2_\perp \mid (\ldots)^{\sharp_4}\Big\}.$$

The occurrences of applications are:

$$a_1 = f\, z^{\langle fx\rangle}, \quad a_2 = g\, z^{\langle gx\rangle},$$

and

$$\hat{z}^{(f\, z^{\langle fx\rangle})} = z^{\langle fx\rangle}, \quad \hat{z}^{(g\, z^{\langle gx\rangle})} = z^{\langle gx\rangle}.$$

Since the functions f and g are declared in this order,

$$\hat{j}(f\, z^{\langle fx\rangle}) = 1, \quad \hat{j}(g\, z^{\langle gx\rangle}) = 2.$$

Hence, the predicate $(\ldots)^{\sharp_4}$ is $y_1 = y_2$ => $r_1 = r_2$ and the transformed type is

$$\big((y_1, y_2)\colon (\mathbf{int})^2_\perp\big) \to \big\{(r_1, r_2)\colon (\mathbf{int})^2_\perp \mid y_1 = y_2 \texttt{ => } r_1 = r_2\big\}.$$

The transformation of terms follows the ideas described in Section 3.1 except that option types have been introduced. For example, the term $(\lambda x.\, t_1, \lambda y.\, t_2)$ is transformed into the term

$$\lambda(x, y).\, \mathbf{let}\ r_1 = \mathbf{if}\ x = \perp\ \mathbf{then}\ \perp\ \mathbf{else}\ (t_1)^{\sharp_4}\ \mathbf{in}$$
$$\mathbf{let}\ r_2 = \mathbf{if}\ y = \perp\ \mathbf{then}\ \perp\ \mathbf{else}\ (t_2)^{\sharp_4}\ \mathbf{in}\ (r_1, r_2).$$

Here, $\perp$ is the exception of option types (i.e. `None` in OCaml or `Nothing` in Haskell), and we have omitted a projection from $(\tau)_\perp$ to τ above. The option type (and the conditional branch **if** $x = \perp$ **then** $\ldots$), is used to preserve the side effect (divergence or failure). For example, consider the following program:

```
let rec f x = ... and g y = g y in
let main n = assert (f n > 0)
```

This program defines functions `f` and `g` but does not use `g`. The body of the main function is transformed to `fst(fg(n,⊥))>0`, where `fg` is a (naïvely) tupled version of `(f,g)`, which simulates calls of `f` and `g` simultaneously. Without the option type, the simulation of a call of `g` would diverge.

As for the transformation of tuples in Figure 8, tuples of functions are transformed to functions on tuples as described in Section 3.1. Tuples of integers are just transformed in a compositional manner. In the case for projections, we can assume that $(t)^{\sharp_4}$ $(= x)$ is a tuple consisting of integers and a single function. If $\mathbf{pr}_i t$ is a function, $\mathbf{pr}_{i-n}(x\,(\perp, \ldots, \perp, w, \perp, \ldots, \perp))$ should correspond to $(\mathbf{pr}_i t)\, w$. Hence, the output of the transformation is $\lambda w.\, \mathbf{pr}_{i-n}(x\,(\perp, \ldots, \perp, w, \perp, \ldots, \perp))$. Otherwise, $\mathbf{pr}_i t$ is just transformed in a compositional manner.

Finally, we define $(-)^\sharp_T$ as the composition of the transformations:

$$(t)^\sharp_T = ((((t)^{\sharp_1})^{\sharp_2})^{\sharp_3}_T)^{\sharp_4}.$$

3.3 Soundness of the Transformation

The transformation $(-)^\sharp$ reduces type checking of general refinement types (with the assumptions in Section 2.3) into that of first-order refinement types, and its soundness is ensured by Theorem 1 below.

In the theorem, for a given typing judgment $\overset{?}{\models} t : \tau$, we assume a condition called *consistency* on multiplicity annotation T and multiplicity type ϕ. We give its formal definition in Appendix G; intuitively, T and ϕ are consistent (with respect to t and τ) if it makes consistent assumptions on each subterm, so that the result of the transformation is simply-typed.

Theorem 1 (Soundness of Verification by the Transformation)**.** *Let t be a closed term and τ be a type of at most order-2. Let T and ϕ be a multiplicity annotation and a multiplicity type for $((t)^{\sharp_1})^{\sharp_2}$ and $((\tau)^{\sharp_1})^{\sharp_2}$ and suppose that they are consistent and $\tau \leq_{mul} \phi$. Then,*

$$\models (t)^\sharp_T : (\tau)^\sharp_\phi \qquad \textit{implies} \qquad \models t : \tau.$$

Proof. See Appendix H. □

As explained in Section 3.2, ϕ and T above are automatically inferred, and gradually increased until the verification succeeds. Thus, the transformation is automatic as a whole. The converse of Theorem 1, completeness, holds for order-1 types, but not for order-2: see Section 4.2.

4. Transformations for Enabling First-Order Refinement Type Checking

The transformation $(-)^\sharp$ in the previous section allowed us to reduce the refinement type checking $\models t : \tau$ to the first-order refinement type checking $\models (t)^\sharp : (\tau)^\sharp$, but it does not necessarily enable us to prove the latter by using the existing automated verification tools [10, 13, 14, 17, 18, 20]. This is due to the incompleteness of the tools for proving $\models (t)^\sharp : (\tau)^\sharp$. They are either based on (variations of) the first-order refinement type system [21] (see Appendix B for such a refinement type system), or higher-order model checking [9, 10], whose verification power is also equivalent to a first-order refinement type system (with intersection types). In these systems, the proof of $\models t : \tau$ (where τ is a first-order refinement type) must be compositional: if $t = t_1 t_2$, then τ' such that $\models t_1 : \tau' \to \tau$ and $\models t_2 : \tau'$ is (somehow automatically) found, from which $\models t_1 t_2 : \tau$ is derived. The compositionality itself is fine, but the problem is that τ' must also be a first-order refinement type, and furthermore, most of the actual tools can only deal with linear arithmetic in refinement predicates. To see why this is a problem, recall the example of proving `sum` and `sum2` in Section 1. It is expressed as the following refinement type checking problem:

$$\overset{?}{\models} (\mathtt{sum}, \mathtt{sum2}) :$$
$$(\mathtt{sum} : \mathbf{int} \to \mathbf{int}) \times ((n : \mathbf{int}) \to \{r : \mathbf{int} \mid r = \mathtt{sum}(n)\}).$$

It can be translated to the following first-order refinement type checking problem:

$$\overset{?}{\models} \lambda(x, y).(\mathtt{sum}\, x, \mathtt{sum2}\, y) :$$
$$((x, y) : \mathbf{int}^2) \to \{(r_1, r_2) : \mathbf{int}^2 \mid x = y \Rightarrow r_1 = r_2\}.$$

However, for proving the latter in a compositional manner using only first-order refinement types, one would have to infer the following non-linear refinement types for `sum` and `sum2`:

$$(x : \mathbf{int}) \to$$
$$\{r : \mathbf{int} \mid (x \leq 0 \Rightarrow r = 0) \land (x > 0 \Rightarrow r = x(x+1)/2)\}.$$

To deal with the problem above, we further refine the transformation $(-)^\sharp$ by (i) tupling of recursive functions [5] and (ii) insertion of assumptions.

4.1 Tupling of Recursion

The idea is that when a tuple of function calls is introduced by $(-)^{\sharp_4}$ $((f_1\,(\mathbf{pr}_1 y), \ldots, f_m\,(\mathbf{pr}_m y))$ in Figure 8 and $(\mathtt{sum}\, x, \mathtt{sum2}\, y)$ in the example above), we introduce a new recursive function for computing those calls simultaneously. For the example above, we introduce a new recursive function `sum_sum2` defined by:

```
let rec sum_sum2 (x,y) = sum_sumacc(x,y,0)
and sum_sumacc(x,y,m) =
      if x<0 then if y<0 then (0,0) else ...
```

More generally, we combine simple recursive functions as follows. Consider the program:

$$\mathbf{let}\ f = \mathbf{fix}(f, \lambda x.\, \mathbf{if}\ t_{11}\ \mathbf{then}\ t_{12}\ \mathbf{else}\ E_1[f\, t_1])\ \mathbf{in}$$

$$\textbf{let } g = \textbf{fix}(g, \lambda y.\, \textbf{if } t_{21} \textbf{ then } t_{22} \textbf{ else } E_2[g\, t_2]) \textbf{ in } ...\, (f, g)\, ...$$

where E_1 and E_2 are evaluation contexts, and t_{ij}, E_i, and t_i have no occurrence of f nor g. Then, we replace $\lambda(x, y).\,(f\,x, g\,y)$ in $(-)^{\sharp_4}$ with the following tupled version:

$$
\begin{array}{l}
\lambda(x', y').\, \textbf{let } \_ = f\,x' \textbf{ in} \\
\quad \textbf{fix}\big(h,\ \lambda(x, y). \\
\qquad \textbf{if } t_{11} \textbf{then if } t_{21} \textbf{ then } (t_{12}, t_{22}) \textbf{ else } (t_{12}, E_2[g\, t_2]) \\
\qquad \textbf{else if } t_{21} \textbf{then } (E_1[f\, t_1], t_{22}) \\
\qquad \textbf{else let } (r_1, r_2) = h\,(t_1, t_2) \textbf{ in } (E_1[r_1], E_2[r_2])\,\big)(x', y').
\end{array}
$$

The first application $f\,x'$ is inserted to preserve side effects (i.e., divergence and failure *fail*). To see why it is necessary, consider the case where $t_{11} = \textbf{true}$, $t_{12} = \textbf{fail}$ and $t_{21} = \Omega$. The call to the original function fails, but without $\textbf{let } \_ = f\,x' \textbf{ in} \cdots$, the call to the tupled version would diverge.

The function `sum_sumacc` shown in Section 1 can be obtained by the above tupling (with some simplifications).

4.2 Insertion of Assume Expressions

The above refinement of $(-)^{\sharp_4}$ alone is often insufficient. For example, consider the problem of proving that the function:

```
let diff (f,g) = fun x -> f x - g x
```

has the type

$$
\begin{array}{rcl}
\tau & \stackrel{\text{def}}{=} & \{(f, g) : (\textbf{int} \to \textbf{int})^2 \mid \forall x.\, f\,x > g\,x\} \\
 & & \to \quad \{h : \textbf{int} \to \textbf{int} \mid \forall x.\, h\,x > 0\}.
\end{array}
$$

The function is transformed to the following one by $(-)^{\sharp_4}$:

```
let diff fg = fun x ->
  let r1,r2 = fg (x, ⊥) in
  let r1',r2' = fg (⊥, x) in r1 - r2'
```

and the type τ is transformed to

$$
\begin{array}{l}
\big(\, ((x_1, x_2) : \textbf{int}^2) \to \{(r_1, r_2){:}\, \textbf{int}^2 \mid x_1 = x_2 \Rightarrow r_1 > r_2\}\big) \\
\quad \to \quad \big(\textbf{int} \to \{r{:}\, \textbf{int} \mid r > 0\}\big).
\end{array}
$$

Here, $\perp$ is used as a dummy argument as explained in Section 3.2-$\sharp_4$. We cannot conclude that `r1` $-$ `r2'` has type $\{r{:}\, \textbf{int} \mid r > 0\}$ because there is no information about the correlation between `r1` and `r2'`: from the refinement type of `fg`, we can infer that $x = \perp \Rightarrow r_1 > r_2$ and $\perp = x \Rightarrow r'_1 > r'_2$, but $r1 > r2'$ cannot be derived.[5] In fact, $\models$ `(diff)`$^\sharp : (\tau)^\sharp$ does not hold,[6] which is a counterexample of the converse of Theorem 1.

To overcome the problem, we insert the following assertion just after the second call:

```
assume(let (r1'',r2'') = fg(x,x) in
       r1=r1'' & r2'=r2'')
```

Here, `assume`(t) is a shorthand for $\textbf{if } t \textbf{ then true else}$ `loop()` where `loop()` is an infinite loop. From `fg(x,x)`, we obtain `r1''` $>$ `r2''` by using the refinement type of `fg`. We can then use the assumed condition to conclude that `r1` $>$ `r2'`. In general, whenever there are two calls

[5] One may think that we can just combine the two calls of `fg` as

```
let diff fg =
fun x -> let r1,r2 = fg(x,x) in r1-r2'
```

This is certainly possible for the example above, but it is in general difficult if the occurrences of the two calls of `fg` are apart.

[6] To see this, apply `(diff)`$^\sharp$ to

$$\lambda(x_1, x_2).\, \textbf{if } x_1 = x_2 \textbf{ then } (1, 0) \textbf{ else } (0, 0)$$

and apply the returned value to, say, 0.

Table 1. Results of preliminary experiments

program	size (before $\sharp'$)	size (after $\sharp'$)	pred.	time[sec]
sum-acc	56	282	0	0.54
sum-simpl	40	270	0	0.75
sum-mono	27	279	0	0.45
mult-acc	63	347	0	0.38
a-max-gen	112	476	1	0.29
append-xs-nil	72	1364	0	45.57
append-nil-xs	63	725	0	16.43
rev	128	1868	0	176.24
insert	32	6262	0	52.49

```
let r1,r2 = fg (x, ⊥) in
C[let r1',r2' = fg (⊥, y) in ...]
```

(where `C` is some context), we insert an assume statement as in

```
let r1,r2 = fg (x, ⊥) in
C[let r1',r2' = fg (⊥, y) in
  assume(let (r1'',r2'') = fg(x,y)
              in r1=r1'' & r2'=r2''); ...]
```

We write $(-)^{\sharp'}$ for the above assume-inserted version of $(-)^\sharp$. The formal definition of $(-)^{\sharp'}$ is described in Appendix K. In the target language, **fail** is treated as an exception, and we define $\textbf{assume}\,(t)$ as a shorthand for:

$$\textbf{if } (\textbf{try } t \textbf{ with fail} \to \textbf{false}) \textbf{ then true else } \texttt{loop}().$$

Note that our backend model checker MoCHi [10, 14] supports exceptions. After replacing $(-)^\sharp$ with $(-)^{\sharp'}$, Theorem 1 is still valid:

$$\models (t)^{\sharp'}_T : (\tau)^\sharp_\phi \qquad \text{implies} \qquad \models t : \tau.$$

See Appendix L for the details of the proof.

5. Implementation and Experiments

We have implemented a prototype, automated verifier for higher-order functional programs as an extension to a software model checker MoCHi [10, 14] for a subset of OCaml.

Table 1 shows the results of the experiments. The columns "size" show the size of the programs before and after the transformations described in Section 4, where the size is measured by word counts.[7] The column "pred." shows the number of predicates manually given as hints for the backend model checker MoCHi. The experiment was conducted on Intel Core i7-3930K CPU and 16 GB memory. The implementation and benchmark programs are available at `http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi_rel/`.

The programs used in the experiments are as follows. The programs "sum-acc", "sum-simpl", and "append-xs-nil" are those given in Section 1. The program "mult-acc" is similar to "sum-acc" but calculates the multiplication. The program "sum-mono" asserts that the function `sum` is monotonic, i.e., $\forall m, n.\, m \le n \Rightarrow \texttt{sum}(m) \le \texttt{sum}(n)$. The program "a-max-gen" finds the max of a functional array; the checked specification is that "a-max-gen" returns an upper bound. Here is the main part of the code of "a-max-gen".

```
let rec array_max i n array =
  if i >= n then 0 else
  let x = array i in
  let m' = array_max (i+1) n array in
```

[7] Because the transformation is automatic, we consider the number of words is a more appropriate measure (at least for the output of the transformation) than the number of lines.

```
      if x > m' then x else m'
  let main i n =
    let array = make_array n in
    let m = array_max 0 n array in
      if i < n then assert (array i <= m)
```

The program "append-nil-xs" asserts that `append nil xs = xs`. The program "rev" asserts that two list reversal functions are the same, the one uses snoc function and the other one uses an accumulation parameter. The program "insert" asserts that `insert x xs` is sorted for a sorted list `xs`. Note that, for all the programs, invariant annotations were not supplied, except the specification being checked. For example, for "a-max-gen" above, the specification is that the main has type **int** $\to$ **int** $\to$ **unit**, which just means that the assertion `assert (array i <= m)` never fails; no type declaration for `array_max` was supplied. For the "append-xs-nil", as described in Section 1, the verifier checks that `append` has the type

$$xs\colon\tau \to (\{ys\colon\tau \mid ys(0) = \mathtt{None}\}) \to \{rs\colon\tau \mid \forall i.xs(i) = rs(i)\}$$

where $\tau \stackrel{\text{def}}{=} \mathbf{int} \to (\mathbf{int\ option})$. (See Appendix A for more details.)

In the table, one may notice that the program size is significantly increased by the transformation. This has been mainly caused by the tupling transformation for recursive functions. Since the size increase incurs a burden for the backend model checker, we plan to refine the transformation to suppress the size increase. Most of the time for verification has been spent by the backend model checker, not the transformation.

The programs above have been verified fully automatically except "a-max-gen", for which we had to provide one predicate by hand as a hint (for predicate abstraction) for the underlying model checker MoCHi. This is a limitation of the current implementation of MoCHi, rather than that of our approach. We have not been able to experiment with larger programs due to the limitation of MoCHi. We expect that with a further improvement of automated refinement type checkers, our verifier works for larger and more complex programs. Despite the limitation of the size of the experiments, we are not aware of any other verification tools that can verify all the above programs with the same degree of automation.

6. Related Work

Knowles and Flanagan [7, 8] gave a general refinement type system where refinement predicates can refer to functions. Their verification method is however a combination of static and dynamic checking, which delegates type constraints that could not be statically discharged to dynamic checking. The dynamic checking will miss potential bugs, depending on given arguments. On the other hand, our method is static and fully automatic.

Some of the recent work on (semi-)automated[8] refinement type checking [13, 24] supports the use of uninterpreted function symbols in refinement predicates. Uninterpreted functions can be used only for total functions. Furthermore, their method cannot be used to prove relational properties like the ones given in Section 1, since their method cannot refer to the definitions of the uninterpreted functions.

Unno et al. [19] have proposed another approach to increase the power of automated verification based on first-order refinement types. To overcome the limitation that refinement predicates cannot refer to functions, they added an extra integer parameter for each higher-order argument so that the extra parameter captures the behavior of the higher-order argument, and the dependency between the higher-order argument and the return value can be captured indirectly through the extra parameter. They have shown that the resulting first-order refinement type system is *in theory* relatively complete (in the same sense as Hoare logic is). With such an approach, however, a complex encoding of the information about a higher-order argument (essentially Gödel encoding) into the extra parameter would be required to properly reason about dependencies between functions, hence *in practice* (where only theorem provers for a restricted logic such as Presburger arithmetic is available), the verification of relational properties often fails. In fact, none of the examples used in the experiments of Section 5 (with encoding into the reachability verification problem considered in [19]) can be verified with their approach.

Suter et al. [15, 16] proposed a method for verifying correctness of first-order functional programs that manipulate recursive data structures. Their method is similar to our method in the sense that recursive functions can be used in a program specification. For example, the example programs "sum-simpl" and "append-nil-xs" can be verified by their method (if lists are not encoded as functions). Their method however can deal only with specifications which does not include partial functions. For this reason, if we rewrite the definition of `sum` as:

```
let rec sum n = if n=0 then 0 else n+sum(n-1)
```

their method cannot verify "sum-simpl" correctly, while our method can.

There are less automated approaches to refinement type checking, where programmers supply invariant annotations (in the form of refinement types) for all recursive functions [3, 4], and then verification conditions are generated and discharged by SMT solvers. Xu's method [22, 23] for contract checking also requires that contracts must be declared for all recursive functions. In contrast, in our method, a refinement type is used only for specifying the property to be verified, and no declaration is required for auxiliary functions.

There are several studies of interactive theorem provers (Coq, Agda, etc.) that can deal with general refinement types. These systems aim to support the verification, not to verify automatically. Therefore, one must give a complete proof of the correctness by hand. Moreover, these systems cannot deal directly with terminating programs and the proof of the termination is also required.

7. Conclusion and Future Work

We have proposed an automated method for verification of relational properties of functional programs, by reduction to the first-order refinement type checking. We have confirmed the effectiveness of the method using a prototype implementation. Future work includes a proof of the relative completeness of our verification method (with respect to a general refinement type system) and an extension of the method to deal with more expressive refinement types. As described in Section 2, we restrict refinement predicates to top-level quantifiers over the base type and first-order function variables. Relaxing this limitation is also left for future work.

Acknowledgment

We would like to thank Naohiko Hoshino and anonymous referees for useful comments. This work was supported by Kakenhi 23220001.

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP '06*, pages 69–83, 2006.

[2] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, Sept. 2001.

[8] Not fully automated in the sense that a user must supply hints on predicates.

[3] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL '14*, volume 49, pages 193–205, 2014.

[4] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *TOPLAS*, 33(2):8, Jan. 2011.

[5] W.-N. Chin. Towards an automated tupling strategy. In *PEPM 1993*, pages 119–132, 1993.

[6] D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. In *LICS '09*, pages 71–80, 2009.

[7] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP '07*, pages 505–519, 2007.

[8] K. L. Knowles and C. Flanagan. Hybrid type checking. *TOPLAS*, 32 (2), Jan. 2010.

[9] N. Kobayashi. Model checking higher-order programs. *J. ACM*, 60 (3):20, 2013.

[10] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI '11*, pages 222–233, 2011.

[11] E. Moggi. Computational lambda-calculus and monads. In *LICS '89*, pages 14–23, 1989.

[12] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL '11*, pages 587–598, 2011.

[13] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*, pages 159–169, 2008.

[14] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM '13*, pages 53–62, 2013.

[15] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL '10*, volume 45, page 199, 2010.

[16] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS '11*, pages 298–315, 2011.

[17] T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130, 2010.

[18] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288, 2009.

[19] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL '13*, page 75, 2013.

[20] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP '13*, 2013.

[21] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227, 1999.

[22] D. N. Xu. Hybrid contract checking via symbolic simplification. In *PEPM '12*, pages 107–116, 2012.

[23] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Workshop on Haskell*, pages 41–52, 2009.

[24] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *VMCAI '13*, 2013.

A. Verification of "append-xs-nil"

We show that how our verifier transforms and verifies the program "append-xs-nil". The whole program is shown below:

```
let rec make_list n =
  if n < 0 then []
  else Random.int 10 :: make_list (n-1)
let rec append xs ys =
  match xs with
    [] -> ys
  | x::xs' -> x :: append xs' ys
let main n i =
  let xs = make_list n in
  let rs = append xs [] in
  assert (List.nth rs i = List.nth xs i)
```

The goal is to verify that the main function has type $\mathbf{int} \to \mathbf{int} \to \mathbf{unit}$, which means that the assertion never fails. As mentioned in Section 5, only the program above is given to the verifier, without any annotations.

The verifier first encodes lists as functions. We use notations for lists and functions interchangeably below. The verifier next guesses a multiplicity annotation T by a heuristics. For this program, the verifier guesses that all the multiplicities are 1.

Then, the transformation $(-)^{\sharp_1}$ is applied to the program, and the following program is obtained.

```
let rec make_list n =
  if n < 0 then []
  else Random.int 10 :: make_list (n-1)
let rec append xs ys =
  match xs with [] -> [],ys,ys
  | x::xs' ->
      let xs'',ys',rs = append xs' ys in
      x::xs'', ys', x::rs
let main n i =
  let xs = make_list n in
  let xs',ys',rs = append xs [] in
  assert (List.nth rs i = List.nth xs' i)
```

The new `append` returns copies of its arguments `xs` and `ys`, and `xs'`, the copy of `xs`, is used in the assertion instead of `xs`.

The transformations $(-)^{\sharp_2}$ and $(-)^{\sharp_3}$ have no effect in this case. By applying the transformation $(-)^{\sharp_4}$, the following program is obtained:

```
let rec make_list n =
  if n < 0 then []
  else Random.int 10 :: make_list (n-1)
let rec append xs ys (i,j,k) =
  match xs with
    [] -> let r1,r2,r3 = None, ys j, ys k in
          assume (j=k => r2=r3); r1, r2, r3
  | x::xs' ->
      let xs''ys'rs = append xs' ys in
      if i = 0 & k = 0 then
        let _,r2,_ = xs''ys'rs(None,j,None) in
        x, r2, x
      else if i = 0 & k <> 0 then
        let _,r2,r3 = xs''ys'rs(None,j,k-1) in
        x, r2, r3
      else if k = 0 then
        let r1,r2,_ = xs''ys'rs(i-1,j,None) in
        r1, r2, x
      else
        xs''ys'rs(i-1,j,k-1)
let main n i =
  let xs = make_list n in
  let xs'_nil_rs = append xs [] in
  let xs'rs (i,j) =
    let r1,r2,r3 = xs'_nil_rs (i, None, j) in
    r1, r3
  in
```

```
  let r1,r2 = xs'rs (i,i) in
  assert (r2 = r1)
```

Here, we omit some constructors and pattern-matchings of option types.

The existing model checker MoCHi infers that the transformed `append` has the following first-order refinement type:

$$(\mathbf{int} \to \mathbf{int}) \to$$
$$((j : \mathbf{int}) \to \{y : \mathbf{int} \mid j = 0 \Rightarrow y = \texttt{None}\}) \to$$
$$\big((i, j, k) : \mathbf{int}^3\big) \to \big\{(r_1, r_2, r_3) : \mathbf{int}^3 \bigm| i = j \Rightarrow r_1 = r_2\big\}$$

From the result of MoCHi, the verifier reports that the original program is safe.

B. A Refinement Type System

This section gives a sound type system for proving $\models t : \tau$. Here we do not assume the restrictions in Section 2.3. We obtain also *first-order refinement type system* by restricting the type system so that function variables are disallowed to occur in predicates in all the refinement types. Various *automatic* verification methods [10, 13, 14, 17, 18, 20] are available for the first-order refinement types.

The type judgment used in the type system is of the form $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$, where Γ, called a type environment, is a sequence of type bindings of the form $x : \tau$, and $\mathcal{L}$ is (the name of) the underlying logic for deciding the validity of predicates, which we keep abstract through the paper. Below, we use general well-formedness $\vdash_{\texttt{GWF}}$ (defined in Appendix C), which represents usual scope rules of dependent types.

We define *value environments* as mappings from variables to closed values and use a meta variable η for them. For a value environment η and an environment Γ such that $\vdash_{\texttt{GWF}} \Gamma$, we define $\eta \models^n_{\mathrm{e}} \Gamma$ as follows:

$$\emptyset \models^n_{\mathrm{e}} \emptyset \overset{\text{def}}{\Longleftrightarrow} \text{true}$$

$$\eta \cup \{x \mapsto V\} \models^n_{\mathrm{e}} \Gamma, x : \tau \overset{\text{def}}{\Longleftrightarrow} \eta \models^n_{\mathrm{e}} \Gamma \text{ and } \models^n_{\mathrm{v}} V : \tau[\eta]$$

The type judgment $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$ semantically means that for any n and η, if $\eta \models^n_{\mathrm{e}} \Gamma$, then $\models^n_{\mathrm{v}} t[\eta] : \tau[\eta]$.

The *general refinement type system* is given in Figures 9 and 10. The judgment $\Gamma \mid P \vdash^{\mathcal{L}} P'$ means that, in $\mathcal{L}$, P implies P' under the type environment Γ. We assume that the logic $\mathcal{L}$ satisfies that, if $\Gamma \mid P \vdash^{\mathcal{L}} P'$, then for any n and η such that $\eta \models^n_{\mathrm{e}} \Gamma$ holds, $\models^n_{\mathrm{p}} P[\eta]$ implies $\models^n_{\mathrm{p}} P'[\eta]$. In Figure 9, we define $t'(\!|x \leftarrow t|\!)$ as $\mathbf{let}\ x = t\ \mathbf{in}\ t'$, and extend it to the operations $P(\!|x \leftarrow t|\!)$ and $\sigma(\!|x \leftarrow t|\!)$ compositionally. For example, $(\forall y.t_1 \wedge t_2)(\!|x \leftarrow t|\!) = \forall y.(t_1(\!|x \leftarrow t|\!)) \wedge (t_2(\!|x \leftarrow t|\!))$. We define $t(\!|x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n|\!)$ as $((t(\!|x_n \leftarrow t_n|\!)) \cdots)(\!|x_1 \leftarrow t_1|\!)$.

The typing rules are similar to those of Knowles and Flanagan [7]. We discuss some key rules. In the rule T-App, intuitively, y is assumed to have the type obtained by replacing formal arguments in the type of the return value of x with actual arguments. The rule T-Sub is for subsumption. For example, $\Gamma \vdash 42 : \{\nu : \mathbf{int} \mid \nu \geq 0\}$ is obtained by the following derivation.

$$\frac{\Gamma \vdash 42 : \{\nu : \mathbf{int} \mid \nu = 42\} \quad \Gamma \vdash \{\nu : \mathbf{int} \mid \nu = 42\} <: \{\nu : \mathbf{int} \mid \nu \geq 0\}}{\Gamma \vdash 42 : \{\nu : \mathbf{int} \mid \nu \geq 0\}}$$

In the rule T-Fail, **fail** is typable only if a contradiction occurs in the type environment.

We now show a typing of the running example introduced in Section 1. Here, as the underlying logic $\mathcal{L}$, we use linear integer

$$\frac{\Gamma(x) = \tau \qquad \Gamma \vdash_{\texttt{GWF}} \tau}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} x : \tau}\ (\text{T-Var}) \qquad \frac{}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} n : \mathbf{int}}\ (\text{T-Const})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \mathbf{int} \mid P\} \\ \Gamma, x : \{\nu : \mathbf{int} \mid P \wedge \nu = \mathbf{true}\} \vdash^{\mathcal{L}}_{\mathrm{t}} t_1 : \tau \\ \Gamma, x : \{\nu : \mathbf{int} \mid P \wedge \nu \neq \mathbf{true}\} \vdash^{\mathcal{L}}_{\mathrm{t}} t_2 : \tau \\ (x \notin FV(t_1) \cup FV(t_2))\end{array}}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \tau(\!|x \leftarrow t|\!)}\ (\text{T-If})$$

$$\frac{\text{The arity of } [\![\mathtt{op}]\!] \text{ is } n \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t_i : \mathbf{int}}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} \mathtt{op}(t_1, \dots, t_n) : \mathbf{int}}\ (\text{T-Op})$$

$$\frac{\Gamma,\ f : (x_1{:}\,\tau_1) \to \tau_2,\ x_1 : \tau_1 \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau_2 \quad (f \notin FV(\tau_1) \cup FV(\tau_2))}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} \mathbf{fix}(f, \lambda x_1.\, t) : (x_1{:}\,\tau_1) \to \tau_2}\ (\text{T-Fix})$$

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : (x_1{:}\,\tau_1) \to \tau_2 \mid P\} \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t_1 : \tau_1}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t t_1 : \tau_2(\!|x_1 \leftarrow t_1|\!)}\ (\text{T-App})$$

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t_i : \rho_i(\!|x_1 \leftarrow t_1, \dots, x_{i-1} \leftarrow t_{i-1}|\!) \quad \text{for all } i \leq n}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} (t_1, \dots, t_n) : \prod_{i=1}^n (x_i{:}\,\rho_i)}\ (\text{T-Tuple})$$

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \big\{\nu : \prod_{i=1}^n (x_i{:}\,\rho_i) \bigm| P\big\} \qquad \rho_i = \{\nu_i : \sigma_i \mid P_i\}}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} \mathbf{pr}_i t : \{\nu_i : \sigma_i \mid P_i\}\,(\!|x_1 \leftarrow \mathbf{pr}_1 t, \dots, x_{i-1} \leftarrow \mathbf{pr}_{i-1} t|\!)}\ (\text{T-Proj})$$

$$\frac{}{\Gamma,\ x : \{\nu : \sigma \mid \mathbf{false}\} \vdash^{\mathcal{L}}_{\mathrm{t}} \mathbf{fail} : \tau}\ (\text{T-Fail})$$

$$\frac{\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma' <: \Gamma \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau \qquad \Gamma' \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'}{\Gamma' \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau'}\ (\text{T-Sub})$$

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \sigma \mid P(\!|\nu \leftarrow t|\!)\}}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \sigma \mid P\}}\ (\text{T-Subst})$$

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \sigma \mid P\} \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \sigma \mid P'\}}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \{\nu : \sigma \mid P \wedge P'\}}\ (\text{T-Conj})$$

Figure 9. Typing rules

$$\frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \sigma <: \sigma' \qquad \Gamma, \nu : \sigma \mid P \vdash^{\mathcal{L}} P'}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \{\nu : \sigma \mid P\} <: \{\nu : \sigma' \mid P'\}}\ (\text{Sub-Refine})$$

$$\frac{}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \mathbf{int} <: \mathbf{int}}\ (\text{Sub-Int}) \qquad \frac{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau_1' <: \tau_1 \qquad \Gamma, x_1 : \tau_1' \vdash^{\mathcal{L}}_{\mathrm{s}} \tau_2 <: \tau_2'}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} (x_1 : \tau_1) \to \tau_2 <: (x_1 : \tau_1') \to \tau_2'}\ (\text{Sub-Fun})$$

$$\frac{\Gamma, x_1 : \rho_1, \dots, x_{i-1} : \rho_{i-1} \vdash^{\mathcal{L}}_{\mathrm{s}} \rho_i <: \rho_i' \quad \text{for all } i \leq n}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \prod_{i=1}^n (x_i{:}\,\rho_i) <: \prod_{i=1}^n (x_i{:}\,\rho_i')}\ (\text{Sub-Tuple})$$

$$\frac{}{\vdash^{\mathcal{L}}_{\mathrm{es}} \emptyset <: \emptyset}\ (\text{EnvSub-Nil}) \qquad \frac{\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma' \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'}{\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma,\ x : \tau <: \Gamma',\ x : \tau'}\ (\text{EnvSub-Cons})$$

Figure 10. Subtyping rules

$$\frac{\mathtt{ST}(\Gamma) \vdash_{\mathtt{ST}} t : \mathbf{int}}{\Gamma \vdash_{\mathtt{GWF}} t} \quad (\text{GWF-PredTerm})$$

$$\frac{\Gamma \vdash_{\mathtt{GWF}} P_1 \qquad \Gamma \vdash_{\mathtt{GWF}} P_2}{\Gamma \vdash_{\mathtt{GWF}} P_1 \wedge P_2} \quad (\text{GWF-PredAnd})$$

$$\frac{\Gamma, y_1 : \mathbf{int}, \ldots, y_n : \mathbf{int} \vdash_{\mathtt{GWF}} P}{\Gamma \vdash_{\mathtt{GWF}} \forall y_1, \ldots, y_n. P} \quad (\text{GWF-PredForAll})$$

$$\frac{\Gamma \vdash_{\mathtt{GWF}} \sigma \qquad \Gamma, x : \sigma \vdash_{\mathtt{GWF}} P}{\Gamma \vdash_{\mathtt{GWF}} \{x : \sigma \mid P\}} \quad (\text{GWF-Refine})$$

$$\frac{}{\Gamma \vdash_{\mathtt{GWF}} \mathbf{int}} \quad (\text{GWF-Int})$$

$$\frac{\Gamma \vdash_{\mathtt{GWF}} \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{GWF}} \tau_2}{\Gamma \vdash_{\mathtt{GWF}} (x : \tau_1) \to \tau_2} \quad (\text{GWF-Fun})$$

$$\frac{\Gamma \vdash_{\mathtt{GWF}} \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{GWF}} \tau_2}{\Gamma \vdash_{\mathtt{GWF}} (x : \tau_1) \times \tau_2} \quad (\text{GWF-Pair})$$

$$\frac{}{\vdash_{\mathtt{GWF}} \emptyset} \quad (\text{GWF-ENil})$$

$$\frac{\vdash_{\mathtt{GWF}} \Gamma \qquad \Gamma \vdash_{\mathtt{GWF}} \tau \qquad (x : \_) \notin \Gamma}{\vdash_{\mathtt{GWF}} \Gamma, x : \tau} \quad (\text{GWF-ECons})$$

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

Figure 11. General well-formedness of types

arithmetic with beta equality. The following derivation shows that $\mathtt{sum}$ has the type $\{f : \mathbf{int} \to \mathbf{int} \mid \forall x.\ x \geq 0 \Rightarrow f\ x = x + f\ (x-1)\}$.

$$\dfrac{\vdots \quad \dfrac{\dfrac{\vdots \qquad \nu : \mathbf{int} \to \mathbf{int} \mid \mathbf{true} \vdash^{\mathcal{L}} P(\!|f \leftarrow t|\!)}{\vdash^{\mathcal{L}}_{\mathrm{s}} \mathbf{int} \to \mathbf{int} <: \{f : \mathbf{int} \to \mathbf{int} \mid P(\!|f \leftarrow t|\!)\}} \text{ Sub-Refine}}{\vdash t : \{f : \mathbf{int} \to \mathbf{int} \mid P(\!|f \leftarrow t|\!)\}} \text{ T-Sub}}{\vdash t : \{f : \mathbf{int} \to \mathbf{int} \mid P\}} \text{ T-Subst}$$

where $t = \mathbf{fix}(\mathtt{sum}, \lambda x.\, \mathbf{if}\ x < 0\ \mathbf{then}\ 0\ \mathbf{else}\ x + \mathtt{sum}\,(x-1))$ and $P = x \geq 0 \Rightarrow f\,x = x + f\,(x-1)$. Since $P(\!|f \leftarrow t|\!) \iff x \geq 0 \Rightarrow t\,x = x + t\,(x-1) \iff x \geq 0 \Rightarrow x + t\,(x-1) = x + t\,(x-1)$ and $x + t\,(x-1) = x + t\,(x-1)$ is valid in $\mathcal{L}$, $P(\!|f \leftarrow t|\!)$ is valid in $\mathcal{L}$.

The type system is sound with respect to the semantics of types. A proof is given in Appendix D.

Theorem 2 (Soundness of the Type System)**.** *$\vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$ implies $\models t : \tau$.*

C. Well-formedness Conditions for Types

Here we show the definition of general well-formedness, which is applied through Sections 2.1, 2.2, and Appendix B. Figure 11 gives the definition of the general well-formedness.

D. Proof of Soundness of the Type System

Lemma 3. *For any P, t_0 and t_1 such that $t_0 \longrightarrow t_1$, and n,*

$$\models^{n+1}_{\mathrm{p}} P(\!|x \leftarrow t_0|\!) \quad \textit{iff} \quad \models^{n}_{\mathrm{p}} P(\!|x \leftarrow t_1|\!).$$

For any τ, t_0 and t_1 such that $t_0 \longrightarrow t_1$, V, and n,

$$\models^{n+1}_{\mathrm{v}} V : \tau(\!|x \leftarrow t_0|\!) \quad \textit{iff} \quad \models^{n}_{\mathrm{v}} V : \tau(\!|x \leftarrow t_1|\!).$$

Proof. We prove the two statements by the inductions on the syntax of P and τ, respectively.

$\boxed{P = \forall x. P}$ Trivial.
$\boxed{P = P_1 \wedge P_2}$ Trivial.
$\boxed{P = t}$ We show that

$$\models^{n+1}_{\mathrm{p}} \mathbf{let}\ x = t_0\ \mathbf{in}\ t \quad \text{iff} \quad \models^{n}_{\mathrm{p}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t.$$

Since $t_0 \longrightarrow t_1$,

$$\mathbf{let}\ x = t_0\ \mathbf{in}\ t \ \longrightarrow\ \mathbf{let}\ x = t_1\ \mathbf{in}\ t; \tag{1}$$

hence,

$$\begin{aligned} &\models^{n+1}_{\mathrm{p}} \mathbf{let}\ x = t_0\ \mathbf{in}\ t \\ \overset{\text{def}}{\iff}\ & \forall A\, \forall k \leq n+1. \big((\mathbf{let}\ x = t_0\ \mathbf{in}\ t) \longrightarrow^{k} A \implies A = 0\big) \\ \iff\ & \forall A\, \forall k \leq n. \quad \big((\mathbf{let}\ x = t_1\ \mathbf{in}\ t) \longrightarrow^{k} A \implies A = 0\big) \\ \overset{\text{def}}{\iff}\ & \models^{n}_{\mathrm{p}} \mathbf{let}\ x = t_1\ \mathbf{in}\ t, \end{aligned}$$

where the left-to-right implication is from (1) while the converse is from (1) and since the evaluation is deterministic.
$\boxed{\tau = \{\nu : \sigma \mid P\}}$ Trivial.
$\boxed{\tau = (x_1 \colon \tau_1) \to \tau_2}$ Trivial.
$\boxed{\tau = (x_1 \colon \tau_1) \times \tau_2}$ Trivial.

□

The following are typical lemmas for step-index.

Lemma 4.

1. *For any n, V, and τ, if $\models^{n+1}_{\mathrm{v}} V : \tau$ then $\models^{n}_{\mathrm{v}} V : \tau$.*
2. *If $t_0 \longrightarrow t_1$ and $\models^{n}_{\mathrm{c}} t_1 : \tau$, then $\models^{n+1}_{\mathrm{c}} t_0 : \tau$.*

Proof. Straightforward: 1 is by induction on τ, and 2 is from the definition of $\models^{n}_{\mathrm{c}}$. □

(The converse of Lemma 4.2 also holds similarly to Lemma 3; but we do not need it.)

Theorem 5 (Soundness of Type System)**.** *For any $\mathcal{L}$, if $\vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$, then $\models t : \tau$.*

We show this theorem by generalizing on environments as Lemma 7, which needs Lemma 6 and the following definition:

$$\Gamma \models t : \tau \ \overset{\text{def}}{\iff}\ \text{for any } n \text{ and } \eta, \text{ if } \eta \models^{n}_{\mathrm{e}} \Gamma \text{ then } \models^{n}_{\mathrm{c}} t[\eta] : \tau[\eta].$$

Lemma 6 (Soundness of Subtyping)**.** *For any $\mathcal{L}$, if $\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'$, then for any n and η such that $\eta \models^{n}_{\mathrm{e}} \Gamma$ and for any V, if $\models^{n}_{\mathrm{v}} V : \tau[\eta]$ then $\models^{n}_{\mathrm{v}} V : \tau'[\eta]$.*

Also, if $\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma'$, then for any n and η, if $\eta \models^{n}_{\mathrm{e}} \Gamma$ then $\eta \models^{n}_{\mathrm{e}} \Gamma'$.

Proof. By induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'$ and $\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma'$. (See Appendix E for details.) □

Lemma 7. *For any $\mathcal{L}$, if $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$, then $\Gamma \models t : \tau$.*

Proof. By induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$. (See Appendix F for details.) □

E. Proof of Lemma 6

We prove Lemma 6 by induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'$ and $\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma'$.
$\boxed{(\text{Sub-Refine})}$ Trivial from the inductive hypothesis and the assumption on $\mathcal{L}$, i.e., if $\Gamma \mid P \vdash^{\mathcal{L}} P'$, then for any n and η such that $\eta \models^{n}_{\mathrm{e}} \Gamma$, if $\models^{n}_{\mathrm{p}} P[\eta]$ then $\models^{n}_{\mathrm{p}} P'[\eta]$.
$\boxed{(\text{Sub-Int})}$ Trivial.
$\boxed{(\text{Sub-Fun})}$ This case is also trivial (and tedious), but we give the detail for demonstration. Suppose inductive hypotheses:

(i) for any n, $\eta \models^n_{\mathrm{e}} \Gamma$, and V_1, if $\models^n_{\mathrm{v}} V_1 : \tau'_1[\eta]$ then $\models^n_{\mathrm{v}} V_1 : \tau_1[\eta]$,

(ii) for any n, $\eta' \models^n_{\mathrm{e}} (\Gamma,\, x_1 : \tau'_1)$ and V_2, if $\models^n_{\mathrm{v}} V_2 : \tau_2[\eta']$ then $\models^n_{\mathrm{v}} V_2 : \tau'_2[\eta']$.

For given n, η such that $\eta \models^n_{\mathrm{e}} \Gamma$, and V, assume $\models^n_{\mathrm{v}} V : (x_1\!: \tau_1) \to \tau_2$, i.e.,

(iii) for any $n' \le n$, any V_1 such that $\models^{n'}_{\mathrm{v}} V_1 : \tau_1[\eta]$, and any A_2 and $k \le n'$ such that $V V_1 \longrightarrow^k A_2$, $\models^{n'-k}_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1]$.

Then, we show that $\models^n_{\mathrm{v}} V : (x_1\!: \tau'_1) \to \tau'_2$, i.e., for given $n' \le n$, given V_1 such that $\models^{n'}_{\mathrm{v}} V_1 : \tau'_1[\eta]$, and given A_2 and $k \le n'$ such that $V V_1 \longrightarrow^k A_2$, we show $\models^{n'-k}_{\mathrm{v}} A_2 : \tau'_2[\eta][x_1 \mapsto V_1]$.

Since $\eta \models^n_{\mathrm{e}} \Gamma$, by Lemma 4.1, $\eta \models^{n'}_{\mathrm{e}} \Gamma$; then since $\models^{n'}_{\mathrm{v}} V_1 : \tau'[\eta]$ and by (i), we have $\models^{n'}_{\mathrm{v}} V_1 : \tau_1[\eta]$. Hence, since $V V_1 \longrightarrow^k A_2$ and by (iii),

$$\models^{n'-k}_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1]. \tag{2}$$

Now, let $\eta' := \eta \cup \{x_1 \mapsto V_1\}$. Since

$$\eta \models^n_{\mathrm{e}} \Gamma \quad \text{and} \quad \models^{n'}_{\mathrm{v}} V_1 : \tau'_1[\eta]$$

and $n' - k \le n' \le n$, by Lemma 4.1,

$$\eta \models^{n'-k}_{\mathrm{e}} \Gamma \quad \text{and} \quad \models^{n'-k}_{\mathrm{v}} V_1 : \tau'_1[\eta]$$

i.e., $\eta' \models^{n'-k}_{\mathrm{e}} (\Gamma,\, x_1 : \tau'_1)$. By (2), $\models^{n'-k}_{\mathrm{v}} A_2 : \tau_2[\eta']$ (thus A_2 is a value); hence, by (ii), $\models^{n'-k}_{\mathrm{v}} A_2 : \tau'_2[\eta'] (= \tau'_2[\eta][x_1 \mapsto V_1])$.

$\boxed{\text{(SUB-PAIR)}}$ Trivial.

$\boxed{\text{(ENVSUB-NIL)}}$ Trivial.

$\boxed{\text{(ENVSUB-CONS)}}$ Trivial.

F. Proof of Lemma 7

We prove Lemma 7 by induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$. Important cases are (T-FIX), (T-APP), and (T-FAIL); especially, we use induction on n (only) in the case (T-FIX). Lemma 6 is used in the case (T-SUB).

$\boxed{\text{(T-VAR)}}$ Trivial.

$\boxed{\text{(T-CONST)}}$ Trivial.

$\boxed{\text{(T-IF)}}$ Suppose inductive hypotheses on the typing derivation:

(i) for any n, η, and V, if

$$\eta \cup \{x \mapsto V\} \models^n_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P \wedge \nu = \mathbf{true}\})$$

then $\models^n_{\mathrm{c}} t_1[\eta][x \mapsto V] : \tau[\eta][x \mapsto V]$.

(ii) for any n, η, and V, if

$$\eta \cup \{x \mapsto V\} \models^n_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P \wedge \nu \neq \mathbf{true}\})$$

then $\models^n_{\mathrm{c}} t_2[\eta][x \mapsto V] : \tau[\eta][x \mapsto V]$.

Then, for given n, η, and V such that

$$\eta \cup \{x \mapsto V\} \models^n_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P\}) \tag{3}$$

we show

$$\models^n_{\mathrm{c}} (\mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)[\eta][x \mapsto V] : \tau[\eta][x \mapsto V].$$

From (3), $\models^n_{\mathrm{v}} V : \mathbf{int}$, and hence V is an integer. We suppose $V = \mathbf{true}$; the case $V \neq \mathbf{true}$ can be proved similarly.

From (3) and since $V = \mathbf{true}$, n, η, and V satisfy the assumption of (i); hence,

$$\models^n_{\mathrm{c}} t_1[\eta][x \mapsto V] : \tau[\eta][x \mapsto V].$$

While, since $V = \mathbf{true}$,

$$\begin{aligned}&(\mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)[\eta][x \mapsto V]\\ &= \mathbf{if}\ V\ \mathbf{then}\ t_1[\eta][x \mapsto V]\ \mathbf{else}\ t_2[\eta][x \mapsto V]\\ &\longrightarrow t_1[\eta][x \mapsto V].\end{aligned}$$

Then, from Lemmas 4.1 and 4.2,

$$\models^n_{\mathrm{c}} (\mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)[\eta][x \mapsto V] : \tau[\eta][x \mapsto V].$$

$\boxed{\text{(T-OP)}}$ Trivial.

$\boxed{\text{(T-FIX)}}$ Suppose inductive hypotheses on the typing derivation:

(i) for any n and η', if $\eta' \models^n_{\mathrm{e}} \Gamma,\, f : (x_1\!: \tau_1) \to \tau_2,\, x_1 : \tau_1$, then $\models^n_{\mathrm{c}} t_2[\eta'] : \tau_2[\eta']$,

and that $f \notin FV(\tau_1) \cup FV(\tau_2)$. We show, by induction on n, that for any n and η such that $\eta \models^n_{\mathrm{e}} \Gamma$,

$$\models^n_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] : ((x_1\!: \tau_1) \to \tau_2)[\eta]. \tag{4}$$

The base case ($n = 0$) can be easily shown from the definitions. Next, for $n > 0$, we assume the induction hypothesis on n:

(ii) for any η such that $\eta \models^{n-1}_{\mathrm{e}} \Gamma$,

$$\models^{n-1}_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] : ((x_1\!: \tau_1) \to \tau_2)[\eta],$$

and then for given η such that $\eta \models^n_{\mathrm{e}} \Gamma$, we show (4), i.e., for given $n' \le n$ and given V_1 such that $\models^{n'}_{\mathrm{v}} V_1 : \tau_1[\eta]$, we show

$$\models^{n'}_{\mathrm{c}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] V_1 : \tau_2[\eta][x_1 \mapsto V_1].$$

Now let $n'' := n' - 1$; then $n'' \le n - 1$. For

$$\eta' := \eta \cup \{f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta]\} \cup \{x_1 \mapsto V_1\},$$

we show that $\eta' \models^{n''}_{\mathrm{e}} \Gamma,\, f : (x_1\!: \tau_1) \to \tau_2,\, x_1 : \tau_1$.

Using Lemma 4.1, since $\eta \models^n_{\mathrm{e}} \Gamma$ and $\models^{n'}_{\mathrm{v}} V_1 : \tau_1[\eta]$,

$$\eta \models^{n''}_{\mathrm{e}} \Gamma \quad \text{and} \quad \models^{n''}_{\mathrm{v}} V_1 : \tau_1[\eta].$$

Also, since $\eta \models^{n-1}_{\mathrm{e}} \Gamma$, by (ii) and since $n'' \le n - 1$,

$$\models^{n''}_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] : ((x_1\!: \tau_1) \to \tau_2)[\eta].$$

Hence, with the fact $f \notin FV(\tau_1)$,

$$\eta' \models^{n''}_{\mathrm{e}} \Gamma,\, f : (x_1\!: \tau_1) \to \tau_2,\, x_1 : \tau_1.$$

Then, by (i) where n is instantiated with $n'' = n' - 1$,

$$\models^{n'-1}_{\mathrm{c}} t_2[\eta'] : \tau_2[\eta'].$$

Now $\mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] V_1 \longrightarrow t_2[\eta']$; hence by Lemma 4.2,

$$\models^{n'}_{\mathrm{c}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] V_1 : \tau_2[\eta'].$$

Since now $f \notin FV(\tau_2)$, $\tau_2[\eta'] = \tau_2[\eta][x_1 \mapsto V_1]$; thus the goal has been proved.

$\boxed{\text{(T-APP)}}$ Suppose inductive hypotheses:

(i) for any n, any η such that $\eta \models^n_{\mathrm{e}} \Gamma$, and any A and $k_0 \le n$ such that $t[\eta] \longrightarrow^{k_0} A$, $\models^{n-k_0}_{\mathrm{v}} A : \{\nu : \sigma \mid P\}\,[\eta]$; and

(ii) for any n, any η such that $\eta \models_{\mathrm{e}} \Gamma$, and any A_1 and $k_1 \le n$ such that $t_1[\eta] \longrightarrow^{k_1} A_1$, $\models^{n-k_1}_{\mathrm{v}} A_1 : \tau_1[\eta]$;

and by Lemma 6 we have that

(iii) for any n and η such that $\eta \models^n_{\mathrm{e}} \Gamma$ and for any V and V_1,

$$\begin{aligned}&\models^n_{\mathrm{v}} (V, V_1) : \{(\nu, \nu_1) : \sigma \times \sigma_1 \mid P \wedge P_1\}\,[\eta]\\ &\text{implies} \models^n_{\mathrm{v}} (V, V_1) : \big\{(\nu, \nu_1) : \sigma \times \sigma_1 \;\big|\; P'_2(\!|\nu_2 \leftarrow \nu\nu_1|\!)\big\}\,[\eta].\end{aligned}$$

Then, for given n, η such that $\eta \models^n_{\mathrm{e}} \Gamma, A_2$, and $k \le n$ such that $(tt_1)[\eta] \longrightarrow^k A_2$, we show

$$\models^{n-k}_{\mathrm{v}} A_2 : \{\nu_2 : \sigma_2(\!|x_1 \leftarrow t_1|\!) \mid P_2(\!|x_1 \leftarrow t_1|\!) \wedge P_2'\}\,[\eta],$$

i.e., the following three:

$$\models^{n-k}_{\mathrm{v}} A_2 : \sigma_2(\!|x_1 \leftarrow t_1|\!)[\eta]$$
$$\models^{n-k}_{\mathrm{p}} P_2(\!|x_1 \leftarrow t_1|\!)[\eta][\nu_2 \mapsto A_2]$$
$$\models^{n-k}_{\mathrm{p}} P_2'[\eta][\nu_2 \mapsto A_2].$$

Since $(tt_1)[\eta] = t[\eta]t_1[\eta] \longrightarrow^k A_2$, there exist A and $k_0 \le k\ (\le n)$ such that

$$t[\eta] \longrightarrow^{k_0} A;$$

further, by (i), such A always satisfies

$$\models^{n-k_0}_{\mathrm{v}} A : \{\nu : \sigma \mid P\}\,[\eta]. \tag{5}$$

Thus, especially A is a value and

$$At_1[\eta] \longrightarrow^{k-k_0} A_2. \tag{6}$$

By (6), there exist A_1 and $k_1 \le k - k_0$ such that

$$t_1[\eta] \longrightarrow^{k_1} A_1; \tag{7}$$

further, by (ii), such A_1 always satisfies

$$\models^{n-k_1}_{\mathrm{v}} A_1 : \tau_1[\eta]. \tag{8}$$

Especially, A_1 is a value and

$$AA_1 \longrightarrow^{k-k_0-k_1} A_2. \tag{9}$$

Now, since

$$\models^{n-k_0}_{\mathrm{v}} A : \{\nu : \sigma \mid P\}\,[\eta] \ \big(= \{\nu : (x_1 {:} \tau_1[\eta]) \to \tau_2[\eta] \mid P[\eta]\}\big),$$

especially $\models^{n-k_0}_{\mathrm{v}} A : (x_1 {:} \tau_1[\eta]) \to \tau_2[\eta]$. Then, we shall utilize the semantics for function type: Let $n' := n - k_0 - k_1$, then $n' \le n - k_0$. Since $n' \le n - k_1$, by (8) and Lemma 4.1, $\models^{n'}_{\mathrm{v}} A_1 : \tau_1[\eta]$. Thus, we have

$$\models^{n'}_{\mathrm{c}} AA_1 : \tau_2[\eta][x_1 \mapsto A_1].$$

Then, by the definition of $\models^{n'}_{\mathrm{c}}$ and (9), since $n' - (k - k_0 - k_1) = n - k$,

$$\models^{n-k}_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto A_1].$$

Now, by (7) and Lemmas 3 and 4.1,

$$\models^{n-k}_{\mathrm{v}} A_2 : \tau_2[\eta](\!|x_1 \leftarrow t_1[\eta]|\!) \quad \big(= \tau_2(\!|x_1 \leftarrow t_1|\!)[\eta]\,\big),$$

i.e.,

$$\models^{n-k}_{\mathrm{v}} A_2 : \sigma_2(\!|x_1 \leftarrow t_1|\!)[\eta], \quad \models^{n-k}_{\mathrm{p}} P_2(\!|x_1 \leftarrow t_1|\!)[\eta][\nu_2 \mapsto A_2].$$

The remaining to show is

$$\models^{n-k}_{\mathrm{p}} P_2'[\eta][\nu_2 \mapsto A_2].$$

From (5) and (8), $\models^{n'}_{\mathrm{v}} A : \{\nu : \sigma \mid P\}\,[\eta]$ and $\models^{n'}_{\mathrm{v}} A_1 : \tau_1[\eta]$ $\big(= \{\nu_1 : \sigma_1 \mid P_1\}\,[\eta]\big)$; hence, by (iii), we have

$$\models^{n'}_{\mathrm{v}} (A, A_1) : \big\{(\nu, \nu_1) : \sigma \times \sigma_1 \ \big|\ P_2'(\!|\nu_2 \leftarrow \nu\nu_1|\!)\big\}\,[\eta].$$

Hence, $\models^{n'}_{\mathrm{p}} P_2'(\!|\nu_2 \leftarrow AA_1|\!)[\eta]\ (= P_2'[\eta](\!|\nu_2 \leftarrow AA_1|\!))$; then from Lemma 3 and (9), and since $n' - (k - k_0 - k_1) = n - k$, we have $\models^{n-k}_{\mathrm{p}} P_2'[\eta][\nu_2 \mapsto A_2]$.

$\boxed{\text{(T-PAIR)}}$ $\boxed{\text{(T-FST)}}$ $\boxed{\text{(T-SND)}}$ These cases are similar to (and easier than) the case (T-APP).

$\boxed{\text{(T-FAIL)}}$ For given n, η, and V such that $\eta \cup \{x \mapsto V\} \models^n_{\mathrm{e}} \Gamma, x : \{\nu : \sigma \mid \bot\}$, we show simply contradiction, instead of $\models^n_{\mathrm{c}}$ **fail** $: \tau[\eta][x \mapsto V]$.

By the assumption, $\models^n_{\mathrm{v}} V : \{\nu : \sigma \mid \bot\}\,[\eta]\ \big(= \{\nu : \sigma[\eta] \mid \bot\}\big)$; and thus $\models^n_{\mathrm{p}} \bot$. Hence for any A such that $\bot \longrightarrow^0 A$, $A = \mathbf{true}$; in fact $\bot (= \mathbf{false})$ itself can be such A, then $A = \mathbf{true}$ while $A = \bot = \mathbf{false}$, i.e., contradiction.

$\boxed{\text{(T-SUB)}}$ Trivial, from Lemma 6.

$\boxed{\text{(T-SUBST)}}$ Straightforward.

$$\frac{\Phi(x) = \phi}{\Phi \vdash_{\mathrm{c}} x : \phi} \quad \text{(C-VAR)}$$

$$\frac{}{\Phi \vdash_{\mathrm{c}} n : \{\mathbf{int} \mid \emptyset\}} \quad \text{(C-CONST)}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t : \{\mathbf{int} \mid \emptyset\} \qquad \Phi \vdash_{\mathrm{c}} t_1 : \phi \qquad \Phi \vdash_{\mathrm{c}} t_2 : \phi}{\Phi \vdash_{\mathrm{c}} \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \phi} \quad \text{(C-IF)}$$

$$\frac{\text{The arity of } [\![\mathsf{op}]\!] \text{ is } n \qquad \Phi \vdash_{\mathrm{c}} t_i : \{\mathbf{int} \mid \emptyset\}}{\Phi \vdash_{\mathrm{c}} \mathsf{op}(t_1, \ldots, t_n) : \{\mathbf{int} \mid \emptyset\}} \quad \text{(C-OP)}$$

$$\frac{\Phi, f : \{\phi_1 \to \phi_2 \mid M\}, x : \phi_1 \vdash_{\mathrm{c}} t : \phi_2}{\Phi \vdash_{\mathrm{c}} \mathbf{fix}(f, \lambda x.t) : \{\phi_1 \to \phi_2 \mid M\}} \quad \text{(C-FIX)}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t : \{\phi_1 \to \phi_2 \mid M\} \qquad \Phi \vdash_{\mathrm{c}} t_1 : \phi_1}{\Phi \vdash_{\mathrm{c}} t\,t_1 : \phi_2} \quad \text{(C-APP)}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t_i : \{\mathbf{int} \mid \emptyset\} \qquad \Phi \vdash_{\mathrm{c}} t_j' : \{\phi_j \to \phi_j' \mid M(j)\} \qquad (\forall i, j)}{\Phi \vdash_{\mathrm{c}} (\widetilde{t_i}, \widetilde{t_j'}) : \{\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\phi_j \to \phi_j') \mid M\}} \quad \text{(C-TUPLE)}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t : \{\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\phi_j \to \phi_j') \mid M\}}{\Phi \vdash_{\mathrm{c}} \mathbf{pr}_i^{\mathbf{int}} t : \{\mathbf{int} \mid \emptyset\}} \quad \text{(C-PROJI)}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t : \{\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\phi_j \to \phi_j') \mid M\}}{\Phi \vdash_{\mathrm{c}} \mathbf{pr}_j^{\to} t : \{\phi_j \to \phi_j' \mid M(j)\}} \quad \text{(C-PROJF)}$$

$$\frac{}{\Phi \vdash_{\mathrm{c}} \mathbf{fail} : \phi} \quad \text{(C-FAIL)}$$

$$\Phi ::= \emptyset \mid \Phi, x : \phi$$

Figure 12. Type system for multiplicity types

G. Consistency

G.1 Consistency and its Sufficient Condition

Figure 12 defines a type system for a term t and a multiplicity type ϕ. For a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ in this type system, we can define a multiplicity annotation T of t as below: every subterm t' of t has the judgement $\Phi' \vdash_{\mathrm{c}} t' : \phi'$ in the derivation, where

$$\phi' = \{\textstyle\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\phi_j \to \phi_j') \mid M\},$$

and then we can define $T(t')$ as $(M(j))_{j \le m}$.

For a multiplicity annotation T of a term t and a multiplicity type ϕ, T and ϕ are *consistent* if T is the multiplicity annotation defined as above from some derivations of $\Phi \vdash_{\mathrm{c}} t : \phi$ with some Φ. We also call such pair (T, ϕ) *consistent pair for* t. Conversely, for (T, ϕ) and Φ, such derivation is unique if exist; thus, for a closed term t, we (can) identify consistent pairs (T, ϕ) with derivations of $\vdash_{\mathrm{c}} t : \phi$.

The next proposition gives a sufficient condition for consistency, which can be used also to automatically guess consistent multiplicity annotations. Before that, we prepare terminology and a lemma.

A multiplicity annotation T of a term t is *constant with* k $(k \ge 0)$ if, for any subterm t' whose simple type is $\mathbf{int}^n \times \prod_{j=1}^{\ell} (\tau_j \to \tau_j')$, $T(t')(j) = k$ if $\tau_j \to \tau_j'$ is depth-1 and

$T(t')(j) = 1$ otherwise. Similarly, a multiplicity type $\phi = \{\prod_{i=1}^{n} \mathbf{int} \times \prod_{j=1}^{m} (\phi_j \to \phi'_j) \mid M\}$ is *constant with* k ($k \geq 0$) if $M(j) = k$ for every j such that f_j is depth-1, and also all the ϕ_j and ϕ'_j are constant with k, inductively. For a multiplicity type judgment $\Phi \vdash_{\mathrm{c}} t : \phi$, we say it is *constant with* k if all the multiplicity types in Φ and ϕ are constant with k.

Lemma 8. *If $\Phi \vdash_{\mathrm{c}} t : \phi$ is constant with k, there is a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ whose all the occurrences of judgments are constant with k.*

Proof. By induction on t: for any $\Phi \vdash_{\mathrm{c}} t : \phi$ there is one rule-schema among the ten rule-schemata in Figure 12 whose conclusion part agree with $\Phi \vdash_{\mathrm{c}} t : \phi$, and there is at least one rule instance of the rule-schema whose assumption part consists of only judgments that are constant with k. □

Proposition 9. *For a multiplicity annotation T of a term t and a multiplicity type ϕ, if both the T and ϕ are constant with some common $k \geq 0$, then T and ϕ are consistent.*

Proof. For given T and ϕ that are constant with k, let κ be the simple type of ϕ; then, we can infer a simple type environment Γ such that $\Gamma \vdash t : \kappa$. It is clear that the mapping from multiplicity types that are constant with k to simple types is bijective; by this correspondence we obtain from Γ the multiplicity type environments Φ whose all the multiplicity types are constant with k.

By Lemma 8, there is a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ whose all the occurrences of judgments are constant with k. Since T is constant with k, T is equal to the multiplicity annotation defined from the derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$. □

G.2 Multiplicity Annotations for Reduced Terms

Here we prove the subject reduction property of the type system for multiplicity types, and then we define a multiplicity annotation for a reduced term (used as T' in Lemma 14).

Lemma 10 (Substitution Lemma)**.** *If $\Phi_1, x : \phi_1 \vdash_{\mathrm{c}} t : \phi_2$ and $\Phi_1 \vdash_{\mathrm{c}} V : \phi_1$, then $\Phi_1 \vdash_{\mathrm{c}} t[x \mapsto V] : \phi_2$.*

Proof. By induction on the derivation of $\Phi_1, x : \phi_1 \vdash_{\mathrm{c}} t : \phi_2$. □

Proposition 11 (Subject Reduction)**.** *If $\Phi \vdash_{\mathrm{c}} t : \phi$ and $t \longrightarrow t'$, then $\Phi \vdash_{\mathrm{c}} t' : \phi$.*

Proof. Straightforward induction on the derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ except for the case of $t = \mathbf{fix}(f, \lambda x.\, t')\, V$. The case is shown by using Lemma 10. □

Now, for a multiplicity annotation T of a term t and a multiplicity type ϕ such that T and ϕ are consistent, suppose $t \longrightarrow t'$. We have a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ with some Φ and by the subject reduction we have the derivation of $\Phi \vdash_{\mathrm{c}} t' : \phi$. Thus we have a multiplicity annotation T' of t' that is consistent with ϕ; for this, we write as $T \longrightarrow T'$. It is easily shown that the definition of T' is independent of the choices of Φ and a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$.

H. Soundness of Verification by $(-)^\sharp$

Here we prove Theorem 1, the soundness of the verification by $(-)^\sharp$. The definition of consistency is given in Appendix G.

We prove the soundness by dividing it into four parts corresponding to $(-)^{\sharp_1}$, $(-)^{\sharp_2}$, $(-)^{\sharp_3}$, and $(-)^{\sharp_4}$:

Proposition 12. *Let $i = 1$, 2, or 4. For a term t and a type τ such that τ is at most order-2,*

$$\models (t)^{\sharp_i} : (\tau)^{\sharp_i} \qquad \textit{implies} \qquad \models t : \tau.$$

Proposition 13. *For a multiplicity annotation T of a term t, and a multiplicity type ϕ over a type τ such that T and ϕ are consistent and τ is at most order-2,*

$$\models (t)^{\sharp_3}_T : (\tau)^{\sharp_3}_\phi \qquad \textit{implies} \qquad \models t : \tau.$$

The soundness theorem is an immediate corollary of the above since each transformation preserves the property that τ is at most order-2.

All the above propositions can be proved in a similar way. Among them, the case for $(-)^{\sharp_3}$ is the most subtle since it use multiplicity annotation; so we basically focus on this case.

We need the following lemma:

Lemma 14 (Simulation Lemma for $(-)^{\sharp_3}$)**.** *For a multiplicity annotation T of a term t, and a multiplicity type ϕ such that T and ϕ are consistent, if*

$$t \longrightarrow t' \quad (\textit{with} \quad T \longrightarrow T')$$

then there are some natural numbers n, n', and a term t'' such that

$$(t)^{\sharp_3}_T \longrightarrow^n t'' \longleftarrow^{n'} (t')^{\sharp_3}_{T'}$$

$$n - n' = \begin{cases} 2 & \textit{if the redex of } t \textit{ is of the form of application} \\ 1 & \textit{otherwise.} \end{cases}$$

Proof. We prove the lemma in Appendix I. □

As seen above, $(-)^{\sharp_3}$ (and also $(-)^{\sharp_1}$ and $(-)^{\sharp_4}$) does not "simulate" reduction exactly on the number of reduction-steps when a redex is of the form of application. For this, a proof of Proposition 13 has some complication since the semantics of types is given by step-indexed logical relation, (which was adopted to prove the soundness of the refinement type system in Appendix B). We separate such the complication and put into Lemma 18: i.e., we introduce another semantics of types $\models^{\mathrm{LR}}$, which is defined in Figure 13 by usual logical relation without step-indexing; we prove a certain equivalence between the two semantics in Lemma 18; and we prove Proposition 13 with respect to $\models^{\mathrm{LR}}$.

From the above lemma, we have the following:

Lemma 15. *For a multiplicity annotation T of a term t, and a multiplicity type ϕ such that T and ϕ are consistent,*

- *if $t \longrightarrow^n V$ (with $T \longrightarrow^n T'$) for some n, then $(t)^{\sharp_3}_T \longrightarrow^* (V)^{\sharp_3}_{T'}$ and $(V)^{\sharp_3}_{T'}$ is a value,*
- *if $t \longrightarrow^*$ fail, then $(t)^{\sharp_3}_T \longrightarrow^*$ fail,*
- *if $t \uparrow$, then $(t)^{\sharp_3}_T \uparrow$.*

Now we prove Proposition 13. We write $=_{\mathrm{o}}$ for the *observational equivalence*.

Proof of Proposition 13. We prove that

$$\models^{\mathrm{LR}} (t)^{\sharp_3}_T : (\tau)^{\sharp_3}_\phi \qquad \text{implies} \qquad \models^{\mathrm{LR}} t : \tau$$

by induction on the size of the simple types of τ. For ease of presentation, we use the inductive definition of τ in Section 2.1 and omit the product case.

$\boxed{\tau = \{\nu : \mathbf{int} \mid P\}}$ From Lemma 17.

$\boxed{\tau = \{f : (x_1 \colon \tau_1) \to \tau_2 \mid P\}}$ By assumption,

$$\text{if } (t)^{\sharp_3}_T \longrightarrow^* A' \text{ then } \models^{\mathrm{LR}}_{\mathrm{v}} A' : (\{f : (x_1 \colon \tau_1) \to \tau_2 \mid P\})^{\sharp_3}_\phi.$$

Now we suppose $t \longrightarrow^n A$ and show that

$$\models^{\mathrm{LR}}_{\mathrm{v}} A : (x_1 \colon \tau_1) \to \tau_2 \qquad (10)$$

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f \mapsto A]. \qquad (11)$$

$\boxed{\text{(Predicate)}\ \models^{\mathrm{LR}}_{\mathrm{p}}\ \subseteq \{P : \text{closed}\}}$

- $\models^{\mathrm{LR}}_{\mathrm{p}} \forall x.\, P \overset{\mathrm{def}}{\iff} \models^{\mathrm{LR}}_{\mathrm{p}} P[x \mapsto m]$ for all integers m
- $\models^{\mathrm{LR}}_{\mathrm{p}} P_1 \wedge P_2 \overset{\mathrm{def}}{\iff} \models^{\mathrm{LR}}_{\mathrm{p}} P_1$ and $\models^{\mathrm{LR}}_{\mathrm{p}} P_1$
- $\models^{\mathrm{LR}}_{\mathrm{p}} t \overset{\mathrm{def}}{\iff} A = \mathbf{true}$ for all A s.t. $t \longrightarrow^* A$

$\boxed{\text{(Value)}\ \models^{\mathrm{LR}}_{\mathrm{v}} \subseteq \{V : \text{closed}\} \times \{\tau : \text{closed}\}}$

- $\models^{\mathrm{LR}}_{\mathrm{v}} V : \{\nu : \sigma \mid P\} \overset{\mathrm{def}}{\iff} \models^{\mathrm{LR}}_{\mathrm{v}} V : \sigma$ and $\models^{\mathrm{LR}}_{\mathrm{p}} P[\nu \mapsto V]$
- $\models^{\mathrm{LR}}_{\mathrm{v}} V : \mathbf{int} \overset{\mathrm{def}}{\iff} V = m$ for some integer m
- $\models^{\mathrm{LR}}_{\mathrm{v}} V : (x_1 : \tau_1) \to \tau_2 \overset{\mathrm{def}}{\iff}$ for all V_1, $\models^{\mathrm{LR}}_{\mathrm{v}} V_1 : \tau_1$ implies $\models^{\mathrm{LR}} V V_1 : \tau_2[x_1 \mapsto V_1]$
- $\models^{\mathrm{LR}}_{\mathrm{v}} (V_1, \ldots, V_n) : \prod_{i=1}^{n} (x_i : \rho_i) \overset{\mathrm{def}}{\iff}$ $\models^{\mathrm{LR}}_{\mathrm{v}} V_i : \rho_i[x_1 \mapsto V_1, \ldots, x_{i-1} \mapsto V_{i-1}]$ for all $i \leq n$

$\boxed{\text{(Environment)}\ \models^{n}_{\mathrm{e}} \subseteq \{\eta : \text{each value is closed}\} \times \{\Gamma \mid \vdash_{\mathtt{GWF}} \Gamma\}}$

- $\emptyset \models^{\mathrm{LR}}_{\mathrm{e}} \emptyset \overset{\mathrm{def}}{\iff}$ true
- $\eta \cup \{x \mapsto V\} \models^{\mathrm{LR}}_{\mathrm{e}} \Gamma, x : \tau \overset{\mathrm{def}}{\iff} \eta \models^{\mathrm{LR}}_{\mathrm{e}} \Gamma$ and $\models^{\mathrm{LR}}_{\mathrm{v}} V : \tau[\eta]$

$\boxed{\text{(Term)}\ \models^{\mathrm{LR}} \subseteq \{(\Gamma, t, \tau) \mid \Gamma \vdash_{\mathtt{GWF}} \tau\}}$

- $\Gamma \models^{\mathrm{LR}} t : \tau \overset{\mathrm{def}}{\iff}$ $\models^{\mathrm{LR}}_{\mathrm{v}} A : \tau$ for all η s.t. $\eta \models^{\mathrm{LR}}_{\mathrm{e}} \Gamma$ and for all A s.t. $t[\eta] \longrightarrow^* A$

Figure 13. Semantics of types without step-index

By Lemma 15, A is a value—so let $A = \mathbf{fix}(f, \lambda x_1.\, t_2)$—and

$$(t)^{\sharp_3}_{T} \longrightarrow^* (A)^{\sharp_3}_{T'} = \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])}^{m}$$

for T' such that $T \longrightarrow^n T'$ and $m \overset{\mathrm{def}}{=} T'(A)$; hence, by the assumption,

$$\models^{\mathrm{LR}}_{\mathrm{v}} \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])}^{m} : \left(\left(x_1 \colon (\tau_1)^{\sharp_3}_{\phi_1}\right) \to (\tau_2)^{\sharp_3}_{\phi_2}\right)^{m}$$

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f\, t_l \mapsto f_l\, t_l]_{l \leq m'} [f_l \mapsto \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])]_{l \leq m'}$$

where $f\, t_1, \ldots, f\, t_{m'}$ are all the occurrences of (application of) f and suppose $\phi = \{\phi_1 \to \phi_2 \mid M\}$; these mean

$$\models^{\mathrm{LR}}_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}]) : \left(x_1 \colon (\tau_1)^{\sharp_3}_{\phi_1}\right) \to (\tau_2)^{\sharp_3}_{\phi_2} \tag{12}$$

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f \mapsto \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])]. \tag{13}$$

Now we prove (10), i.e., for given V_1 such that $\models^{\mathrm{LR}}_{\mathrm{v}} V_1 : \tau_1$ and A_2 such that $\mathbf{fix}(f, \lambda x_1.\, t_2)V_1 \longrightarrow^* A_2$, we show

$$\models^{\mathrm{LR}}_{\mathrm{v}} A_2 : \tau_2[x_1 \mapsto V_1]\,. \tag{14}$$

Since τ is at most order-2, by using the reduction inspection method in the proof of Lemma 18 (see Appendix J), we can assume that V_1 is in the following normal form:

$$N ::= m \mid \begin{pmatrix} \lambda x.\, \mathbf{case}\ x\ \mathbf{of} & m_1 \to N_1 \\ & \vdots \\ & m_n \to N_n \\ & \_ \to \Omega \end{pmatrix}$$

where n is any natural number, meta-variable m runs over order-0 values, and Ω is some diverging term of a given type. Then, since V_1 is in this normal form, it is clear that there is a derivation of $\vdash_{\mathrm{c}} V_1 : \phi_1$; hence, we have a consistent pair (T'_1, ϕ_1) for V_1. Since $\models^{\mathrm{LR}}_{\mathrm{v}} V_1 : \tau_1$, by Lemma 16,

$$\models^{\mathrm{LR}} (V_1)^{\sharp_3}_{T'_1} : (\tau_1)^{\sharp_3}_{\phi_1}\,.$$

From (T', ϕ) and (T'_1, ϕ_1), we obtain a consistent pair (T'_2, ϕ_2) for $\mathbf{fix}(f, \lambda x_1.\, t_2)V_1$. Then,

$$\begin{aligned} &(\mathbf{fix}(f, \lambda x_1.\, t_2)V_1)^{\sharp_3}_{T'_2} \\ &= (\mathbf{pr}_1\, (\mathbf{fix}(f, \lambda x_1.\, t_2))^{\sharp_3}_{T'})\, (V_1)^{\sharp_3}_{T'_1} \\ &= (\mathbf{pr}_1 \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])}^{m})\, (V_1)^{\sharp_3}_{T'_1} \\ &\longrightarrow \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])\, (V_1)^{\sharp_3}_{T'_1}\,. \end{aligned}$$

Hence, by (12) and since $\models^{\mathrm{LR}} (V_1)^{\sharp_3}_{T'_1} : (\tau_1)^{\sharp_3}_{\phi_1}$,

$$\models^{\mathrm{LR}} (\mathbf{fix}(f, \lambda x_1.\, t_2)V_1)^{\sharp_3}_{T'_2} : (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}]\,.$$

Here x_1 occurs in $(\tau_2)^{\sharp_3}_{\phi_2}$ only if τ_1 is order-0, because, after $(-)^{\sharp_1}$, function variables must be declared inside of each refinement types. If τ_1 is order-0, $(V_1)^{\sharp_3}_{T'_1} = V_1$ by Lemma 17, and

$$(\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}] = (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto V_1] = (\tau_2[x_1 \mapsto V_1])^{\sharp_3}_{\phi_2}\,.$$

Hence,

$$\models^{\mathrm{LR}} (\mathbf{fix}(f, \lambda x_1.\, t_2)V_1)^{\sharp_3}_{T'_2} : (\tau_2[x_1 \mapsto V_1])^{\sharp_3}_{\phi_2}\,.$$

Now, by induction hypothesis,

$$\models^{\mathrm{LR}} \mathbf{fix}(f, \lambda x_1.\, t_2)V_1 : \tau_2[x_1 \mapsto V_1]\,.$$

Since $\mathbf{fix}(f, \lambda x_1.\, t_2)V_1 \longrightarrow^* A_2$, we have shown (14).

Next, we prove (11), i.e.,

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)]\,.$$

Since f has a depth-1 type if f occurs in P, by (13), it is enough to show

$$\mathbf{fix}(f, \lambda x_1.\, t_2) =_{\mathrm{o}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])$$

assuming these terms have depth-1 types. We only have to show that, for any closed value V_1 of $\mathtt{ST}(\tau_1)$,

$$\mathbf{fix}(f, \lambda x_1.\, t_2)V_1 =_{\mathrm{o}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])V_1$$

since the observational equivalence is extensional. Now since V_1 is order-0, $\vdash_{\mathrm{c}} V_1 : \phi_1$; thus we have a consistent pair (T'_1, ϕ_1) for V_1 and (T'_2, ϕ_2) for $\mathbf{fix}(f, \lambda x_1.\, t_2)V_1$. Then,

$$\begin{aligned} &\mathbf{fix}(f, \lambda x_1.\, t_2)V_1 \\ =_{\mathrm{o}}\ & t_2[x_1 \mapsto V_1][f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)] \\ =_{\mathrm{o}}\ & \{\text{by Lemma 17}\} \\ & (t_2[x_1 \mapsto V_1][f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)])^{\sharp_3}_{T'_2} \\ =_{\mathrm{o}}\ & \{\text{by Lemma 19 in Appendix I}\} \\ & (t_2)^{\sharp_3}_{T'}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}][f \mapsto (\mathbf{fix}(f, \lambda x_1.\, t_2))^{\sharp_3}_{T'}] \\ =_{\mathrm{o}}\ & (t_2)^{\sharp_3}_{T'}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}][f \mapsto \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])}^{m}\,] \\ =_{\mathrm{o}}\ & \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])\, (V_1)^{\sharp_3}_{T'_1} \\ =_{\mathrm{o}}\ & \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^{m}])V_1\,. \end{aligned}$$

□

Lemma 16. *For a multiplicity annotation T of a term t, and a multiplicity type ϕ over a type τ such that T and ϕ are consistent and τ is at most order-1,*

$$\models^{\mathrm{LR}} t : \tau \qquad \textit{implies} \qquad \models^{\mathrm{LR}} (t)^{\sharp_3}_T : (\tau)^{\sharp_3}_\phi \,.$$

Proof. By induction on the size of the simple types of τ, similarly to the previous lemma.

$\boxed{\tau = \{\nu : \mathbf{int} \mid P\}}$ From Lemma 17.

$\boxed{\tau = \{\nu : (x : \tau_1) \to \tau_2 \mid P\}}$ By assumption, if $t \longrightarrow^* A$, then

$$\models^{\mathrm{LR}}_{\mathrm{v}} A : (x_1 \colon \tau_1) \to \tau_2 \tag{15}$$

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f \mapsto A]. \tag{16}$$

Now we suppose $(t)^{\sharp_3}_T \longrightarrow^* A'$ and show that $\models^{\mathrm{LR}}_{\mathrm{v}} A' : (\{f : (x_1 \colon \tau_1) \to \tau_2 \mid P\})^{\sharp_3}_\phi$.

By Lemma 15 t does not diverge and by the assumption, there is some value $V = \mathbf{fix}(f, \lambda x_1.\, t_2)$ and n such that $t \longrightarrow^n V$; so we also have $T \longrightarrow^n T'$ and $m = T'(V)$.

From (15) and by Lemma 15,

$$A' = (V)^{\sharp_3}_{T'} = \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])}^m$$

and it is enough to show

$$\models^{\mathrm{LR}}_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m]) : \big(x_1 \colon (\tau_1)^{\sharp_3}_\phi\big) \to (\tau_2)^{\sharp_3}_\phi \tag{17}$$

$$\models^{\mathrm{LR}}_{\mathrm{p}} P[f \mapsto \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])]. \tag{18}$$

Now we prove (17), i.e., for given V_1' such that $\models^{\mathrm{LR}}_{\mathrm{v}} V_1' : (\tau_1)^{\sharp_3}_{\phi_1}$, we show

$$\models^{\mathrm{LR}}_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])V_1' : (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto V_1'] \,. \tag{19}$$

Since τ_1 is order-0, $\vdash_{\mathrm{c}} V_1' : \phi_1$; hence we have consistent pairs (T_1', ϕ_1) for V_1' and (T_2', ϕ_2) for VV_1'. By Lemma 17, $V_1' =_{\mathrm{o}} (V_1')^{\sharp_3}_{T_1'}$ and $\models^{\mathrm{LR}}_{\mathrm{v}} V_1' : \tau_1$. Hence,

$$\begin{aligned}
&\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])V_1' \\
\longrightarrow\ & (t_2)^{\sharp_3}_{T'}\, [x_1 \mapsto V_1'][f \mapsto \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])}^m\ ] \\
=_{\mathrm{o}}\ & (t_2)^{\sharp_3}_{T'}\, [x_1 \mapsto \big(V_1'\big)^{\sharp_3}_{T_1'}][f \mapsto (V)^{\sharp_3}_{T'}] \\
=\ & \{\text{by Lemma 19 in Appendix I}\} \\
& \big(t_2[x_1 \mapsto V_1'][f \mapsto V]\big)^{\sharp_3}_{T_2'} \\
\longleftarrow^*\ & \big(VV_1'\big)^{\sharp_3}_{T_2'} \,.
\end{aligned}$$

Now from (15), $\models^{\mathrm{LR}}_{\mathrm{v}} VV_1' : \tau_2[x_1 \mapsto V_1']$, and by induction hypothesis,

$$\models^{\mathrm{LR}}_{\mathrm{v}} \big(VV_1'\big)^{\sharp_3}_{T_2'} : \big(\tau_2[x_1 \mapsto V_1']\big)^{\sharp_3}_{\phi_2} = (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto V_1'] \,.$$

Thus, we have shown (19).

Finally, (18) is shown from (16) quite similarly to the proof of Proposition 13. ◻

Lemma 17. *For a multiplicity annotation T of a closed term t, and a multiplicity type ϕ over a closed type τ such that T and ϕ are consistent and τ is order-0,*

$$(\tau)^{\sharp_3}_\phi = \tau \qquad \textit{and} \qquad (t)^{\sharp_3}_T =_{\mathrm{o}} t \,.$$

Proof. On types, it is clear by definition.

On terms, it is clear from Lemma 15. ◻

Lemma 18. *For a term t and a type τ of at most order-2,*

$$\models t : \tau \quad \textit{iff} \quad \models^{\mathrm{LR}} t : \tau.$$

Proof. See Appendix J. ◻

The main reason for the restriction of order in Theorem 1 is in the proof of Proposition 13 above, which depends also on Lemmas 16 and 17; and the restriction is not essentially due to the restriction in the above lemma. The restriction in the above lemma itself may be avoided if we choose some other semantics that is "step irrelevant" one (say, denotational one) and also keeps the soundness result for the refinement type system. Or, the above lemma may be able to be generalized to any order of types.

H.1 Proof of Proposition 12

Proposition 12 for the other three transformations is proved similarly to the above proof for $(-)^{\sharp_3}$. The only subtle point is the base case for $(-)^{\sharp_4}$.

As explained in Section 3.1, by $(-)^{\sharp_4}$,

$$\begin{aligned}
&(f_1, f_2)\ : \\
&\quad \big\{(f_1, f_2) : (\mathbf{int} \to \mathbf{int})^2 \;\big|\; \forall x_1, x_2.\, P[f_1\, x_1, f_2\, x_2]\big\}
\end{aligned}$$

is transformed to:

$$\begin{aligned}
&f_1 \times f_2\ : \\
&\quad ((x_1, x_2) : \mathbf{int} \times \mathbf{int}) \to \{(r_1, r_2) : \mathbf{int} \times \mathbf{int} \mid P[r_1, r_2]\} \,.
\end{aligned}$$

By the semantics of types, the former is equivalent to

$$\text{for all } i, x_i \text{ and } A \text{ if } f_i x_i \longrightarrow^* A, \text{ then } A \text{ is value} \tag{20}$$

$$\text{and for all } x_1 \text{ and } x_2, \models_{\mathrm{p}} P[f_1 x_1, f_2 x_2], \tag{21}$$

while the latter is equivalent to

$$\begin{aligned}
&\text{for all } x_1, x_2, \text{and } A, \text{ if } (f_1 x_1, f_2 x_2) \longrightarrow^* A, \\
&\qquad \text{then } A \text{ is value } (V_1, V_2) \text{ and } \models_{\mathrm{p}} P[V_1, V_2],
\end{aligned} \tag{22}$$

$$\begin{aligned}
&\text{for all } x_1 \text{ and } A, \text{ if } f_1 x_1 \longrightarrow^* A, \\
&\qquad \text{then } A \text{ is value } V \text{ and } \models_{\mathrm{p}} P[V, \bot],
\end{aligned} \tag{23}$$

$$\begin{aligned}
&\text{for all } x_2 \text{ and } A, \text{ if } f_2 x_2 \longrightarrow^* A, \\
&\qquad \text{then } A \text{ is value } V \text{ and } \models_{\mathrm{p}} P[\bot, V],
\end{aligned} \tag{24}$$

$$\text{and } \models_{\mathrm{p}} P[\bot, \bot]. \tag{25}$$

Here (23), (24), and (25) happen because our $f_1 \times f_2$ is not just $\lambda(x_1, x_2).\, (f_1\, x_1, f_2\, x_2)$ but utilizes $\bot$ as explained in Section 3.2.

Now the implication from the former to the latter and that from the latter to (20) are obvious.

To show (21) from the latter, only the case when $f_1 x_1$ or $f_2 x_2$ diverges is subtle by the following reason. First by the assumption of use of "branch-strict if", every occurrence of application must be evaluated if before that there is no effect happens. Also, by the assumption that a predicate does not contain **fail**, in predicate *fail* essentially does not happen. I.e., if $f_i x_i$ fails in the former, it happens also in the latter. Thus we only have to take care about divergence that $f_1 x_1$ or $f_2 x_2$ may involve.

If $f_1 x_1$ or $f_2 x_2$ diverge, we cannot use (22), (23), and (24). On the other hand, we can show $\models_{\mathrm{p}} P[f_1 x_1, f_2 x_2]$ by (25) as below. In the evaluation of $\models_{\mathrm{p}} P[f_1 x_1, f_2 x_2]$, since, as explained above, every occurrence of application must be evaluated if before that there is no effect happens, and $f_1 x_1$ or $f_2 x_2$ diverge, $P[f_1 x_1, f_2 x_2]$ must diverge, because if $P[f_1 x_1, f_2 x_2]$ fails before the evaluation of $f_1 x_1$ or $f_2 x_2$, $\models_{\mathrm{p}} P[\bot, \bot]$ also fails.

I. Simulation Lemma

In this section, we prove Simulation Lemma for $\sharp_3$ (i.e., Lemma 14). First, we give several lemmas and definitions.

Lemma 19 (Substitution Lemma)**.** *If $\Phi, x' : \phi' \vdash_{\mathrm{c}} t : \phi$ and $\Phi \vdash_{\mathrm{c}} t' : \phi'$ are derived, so is $\Phi \vdash_{\mathrm{c}} t[x' \mapsto t'] : \phi$ (in a canonical way).*

For derivations of $\Phi, x' : \phi' \vdash_{\mathrm{c}} t : \phi$ and $\Phi \vdash_{\mathrm{c}} t' : \phi'$, let T, T', and $T[T']$ be the multiplicity annotations of t, t', and $t[x' \mapsto t']$ defined from the derivations, respectively. Then,

$$\left(t[x' \mapsto t']\right)^{\sharp_3}_{T[T']} = (t)^{\sharp_3}_{T} \, [x' \mapsto \left(t'\right)^{\sharp_3}_{T'}] \,.$$

Proof. The former is straightforward by induction on derivations of $\Phi, x' \colon \phi' \vdash_{\mathrm{c}} t : \phi$ and by case-analysis of t. We show only the case $t = \mathbf{fix}(f, \lambda x_1 .\, t_2)$.

$\boxed{t = \mathbf{fix}(f, \lambda x_1 .\, t_2)}$ For given derivations below,

$$\dfrac{\dfrac{\vdots}{\Phi,\, x' \colon \phi',\, f \colon \{\!\!\{ \phi_1 \to \phi_2 \mid M \}\!\!\},\, x_1 \colon \phi_1 \vdash_{\mathrm{c}} t_2 : \phi_2}\,\mathfrak{D}}{\Phi,\, x' \colon \phi' \vdash_{\mathrm{c}} \mathbf{fix}(f, \lambda x_1 .\, t_2) : \{\!\!\{ \phi_1 \to \phi_2 \mid M \}\!\!\}}$$

$$\dfrac{\vdots}{\Phi \vdash_{\mathrm{c}} t' : \phi'}\ \mathfrak{D}'$$

we have a derivation $\mathfrak{D}$ of $(\ldots \vdash_{\mathrm{c}} t_2 : \phi_2)$. By induction hypothesis, we have a derivation $\mathrm{IH}(\mathfrak{D}, \mathfrak{D}')$ of $(\ldots \vdash_{\mathrm{c}} t_2[x' \mapsto t'] : \phi_2)$, and the below for $\mathbf{fix}(f, \lambda x_1 .\, t_2[x' \mapsto t']) = \mathbf{fix}(f, \lambda x_1 .\, t_2)[x' \mapsto t']$.

$$\dfrac{\dfrac{\vdots}{\Phi,\, f \colon \{\!\!\{ \phi_1 \to \phi_2 \mid M \}\!\!\},\, x_1 \colon \phi_1 \vdash_{\mathrm{c}} t_2[x' \mapsto t'] : \phi_2}\,\mathrm{IH}(\mathfrak{D}, \mathfrak{D}')}{\Phi \vdash_{\mathrm{c}} \mathbf{fix}(f, \lambda x_1 .\, t_2[x' \mapsto t']) : \{\!\!\{ \phi_1 \to \phi_2 \mid M \}\!\!\}}$$

We prove the latter part by induction on t; again we show only the case $t = \mathbf{fix}(f, \lambda x_1 .\, t_2)$.

$\boxed{t = \mathbf{fix}(f, \lambda x_1 .\, t_2)}$

$$\begin{aligned}
&\left(t[x' \mapsto t']\right)^{\sharp_3}_{T[T']} \\
&= \left(\mathbf{fix}(f, \lambda x_1 .\, t_2)[x' \mapsto t']\right)^{\sharp_3}_{T[T']} \\
&= \left(\mathbf{fix}(f, \lambda x_1 .\, t_2[x' \mapsto t'])\right)^{\sharp_3}_{T[T']} \\
&= \left\{ \text{let } m \overset{\mathrm{def}}{=} T[T'](\mathbf{fix}(f, \lambda x_1 .\, t_2[x' \mapsto t'])) \right\} \\
&\quad \overrightarrow{\mathbf{fix}(f, \lambda x_1 .\, \left(t_2[x' \mapsto t']\right)^{\sharp_3}_{T[T']|_{t_2[x' \mapsto t']}} \, [f \mapsto \overrightarrow{f}^{m}])}^{m} \\
&= \left\{ \text{because } T[T']|_{t_2[x' \mapsto t']} = T|_{t_2}[T'] \text{ from proof of the former} \right\} \\
&\quad \overrightarrow{\mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T|_{t_2}} \, [x' \mapsto \left(t'\right)^{\sharp_3}_{T'}][f \mapsto \overrightarrow{f}^{m}])}^{m} \\
&= \overrightarrow{\mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T|_{t_2}} \, [f \mapsto \overrightarrow{f}^{m}])}^{m} \, [x' \mapsto \left(t'\right)^{\sharp_3}_{T'}] \\
&= \{ \text{because } m = T(\mathbf{fix}(f, \lambda x_1 .\, t_2)) \text{ from proof of the former} \} \\
&= (\mathbf{fix}(f, \lambda x_1 .\, t_2))^{\sharp_3}_{T[T']} \, [x' \mapsto t'] \\
&= (t)^{\sharp_3}_{T} \, [x' \mapsto \left(t'\right)^{\sharp_3}_{T'}]
\end{aligned}$$

□

In evaluation contexts, $[\,]$ does not occur in the scope of any variable binder. Hence, we can regard $[\,]$ as a variable and evaluation contexts as terms, and we derive notions for evaluation contexts from those for terms.

For an evaluation context E, $(E)^{\sharp_3}_{T}$ is not an evaluation context (only) when $E = V E'$. We modify this gap; for evaluation contexts E, we define $(E)^{\sharp_3^{\mathrm{e}}}_{T}$ in Figure 14 and the *step numbers* $\mathrm{s}(E)$ in Figure 15. In both the definitions, we give special treatment to the case of $V E$.

$$\begin{aligned}
([\,])^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} [\,] \\
(\mathsf{op}(\widetilde{V}, E, \widetilde{t}))^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} \mathsf{op}(\widetilde{(V)^{\sharp_3}_{T}}, (E)^{\sharp_3^{\mathrm{e}}}_{T}, \widetilde{(t)^{\sharp_3}_{T}}) \\
(\mathbf{if}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} \mathbf{if}\ (E)^{\sharp_3^{\mathrm{e}}}_{T}\ \mathbf{then}\ (t_1)^{\sharp_3}_{T}\ \mathbf{else}\ (t_2)^{\sharp_3}_{T} \\
(E\, t)^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} \left(\mathbf{pr}_1 (E)^{\sharp_3^{\mathrm{e}}}_{T}\right) (t)^{\sharp_3}_{T} \\
(\mathbf{fix}(f, \lambda x_1 .\, t_2)\, E)^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} \mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T} \, [f \mapsto \overrightarrow{f}^{m}])\, (E)^{\sharp_3^{\mathrm{e}}}_{T} \\
&\qquad (\text{where } m = T(\mathbf{fix}(f, \lambda x_1 .\, t_2))) \\
((\widetilde{V}, E, \widetilde{t}))^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} (\widetilde{(V)^{\sharp_3}_{T}}, (E)^{\sharp_3^{\mathrm{e}}}_{T}, \widetilde{(t)^{\sharp_3}_{T}}) \\
(\mathbf{pr}_i E)^{\sharp_3^{\mathrm{e}}}_{T} &\overset{\mathrm{def}}{=} \mathbf{pr}_i (E)^{\sharp_3^{\mathrm{e}}}_{T}
\end{aligned}$$

Figure 14. $(-)^{\sharp_3^{\mathrm{e}}}$: modified $(-)^{\sharp_3}$ for evaluation contexts

$$\begin{aligned}
\mathrm{s}([\,]) &\overset{\mathrm{def}}{=} 0 \\
\mathrm{s}(\mathsf{op}(\widetilde{V}, E, \widetilde{t})) &\overset{\mathrm{def}}{=} \mathrm{s}(E) \\
\mathrm{s}(\mathbf{if}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2) &\overset{\mathrm{def}}{=} \mathrm{s}(E) \\
\mathrm{s}(E\, t) &\overset{\mathrm{def}}{=} \mathrm{s}(E) \\
\mathrm{s}(V\, E) &\overset{\mathrm{def}}{=} \mathrm{s}(E) + 1 \\
\mathrm{s}((\widetilde{V}, E, \widetilde{t})) &\overset{\mathrm{def}}{=} \mathrm{s}(E) \\
\mathrm{s}(\mathbf{pr}_i E) &\overset{\mathrm{def}}{=} \mathrm{s}(E)
\end{aligned}$$

Figure 15. Step numbers of evaluation contexts

Lemma 20. *1. For any value V, $(V)^{\sharp_3}_{T}$ is a value.*

2. For any evaluation context E and a multiplicity annotation T of E, $(E)^{\sharp_3^{\mathrm{e}}}_{T}$ is an evaluation context.

3. For any evaluation context E, a multiplicity annotation T of E, and any term t such that $E[t]$ is closed,

$$(E)^{\sharp_3}_{T} \, [[\,] \mapsto t] \longrightarrow^{\mathrm{s}(E)} (E)^{\sharp_3^{\mathrm{e}}}_{T} [t] \,.$$

Proof. 1: Clear by induction on values V.
2: Clear by induction on evaluation contexts E.
3: Straightforward by induction on evaluation contexts E and by 1; we show only the key case of $V E$ i.e. $\mathbf{fix}(f, \lambda x_1 .\, t_2) E$.

$$\begin{aligned}
&(\mathbf{fix}(f, \lambda x_1 .\, t_2) E)^{\sharp_3}_{T} \, [[\,] \mapsto t] \\
&= \left(\mathbf{pr}_1 \, (\mathbf{fix}(f, \lambda x_1 .\, t_2))^{\sharp_3}_{T} \right) \left((E)^{\sharp_3}_{T} \, [[\,] \mapsto t] \right) \\
&= \left(\mathbf{pr}_1 \overrightarrow{\mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T} \, [f \mapsto \overrightarrow{f}^{m}])}^{m} \right) \left((E)^{\sharp_3}_{T} \, [[\,] \mapsto t] \right) \\
&\longrightarrow \mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T} \, [f \mapsto \overrightarrow{f}^{m}]) \left((E)^{\sharp_3}_{T} \, [[\,] \mapsto t] \right) \\
&\longrightarrow^{\mathrm{s}(E)} \mathbf{fix}(f, \lambda x_1 .\, (t_2)^{\sharp_3}_{T} \, [f \mapsto \overrightarrow{f}^{m}]) \left((E)^{\sharp_3^{\mathrm{e}}}_{T} [t] \right) \\
&= (\mathbf{fix}(f, \lambda x_1 .\, t_2) E)^{\sharp_3}_{T} \, [t]
\end{aligned}$$

□

Now we prove Lemma 14; recall the statement: For a multiplicity annotation T of a term t, and a multiplicity type ϕ such that T

$$
\begin{array}{lll}
\overline{V} \text{ (value)} & ::= x \mid n \mid \mathbf{fix}(f, \lambda x.\, t) \mid (\overline{V}_1, \ldots, \overline{V}_n) \\
A \text{ (answer)} & ::= \overline{V} \mid \mathit{fail} \\
\overline{E} \text{ (eval. ctx.)} & ::= [\,] \mid \mathtt{op}(\widetilde{\overline{V}}, \overline{E}, \widetilde{t}) \mid \mathbf{if}\ \overline{E}\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \\
& \quad \mid \overline{E}\, t \mid \overline{V}\, \overline{E} \mid (\widetilde{\overline{V}}, \overline{E}, \widetilde{t}) \mid \mathbf{pr}_i \overline{E}
\end{array}
$$

$$
\begin{array}{l}
\overline{E}[\mathtt{op}(n_1, \ldots, n_k)] \longrightarrow \overline{E}[[\![\mathtt{op}]\!](n_1, \ldots, n_k)] \\
\overline{E}[\mathbf{fail}] \longrightarrow \mathit{fail} \\
\overline{E}[\mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow \overline{E}[t_1] \\
\overline{E}[\mathbf{if}\ V\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow \overline{E}[t_2]\ (V\text{: closed}, V \neq \mathbf{true}) \\
\overline{E}[\mathbf{fix}(f, \lambda x.\, t)\overline{V}] \longrightarrow \overline{E}[t[f \mapsto \mathbf{fix}(f, \lambda x.\, t)][x \mapsto \overline{V}]] \\
\overline{E}[\mathbf{pr}_i(\overline{V}_1, \ldots, \overline{V}_n)] \longrightarrow \overline{E}[\overline{V}_i]
\end{array}
$$

Figure 16. Reduction for open terms

and ϕ are consistent, if

$$
t \longrightarrow t' \quad (\text{with} \quad T \longrightarrow T')
$$

then there are some natural numbers n, n', and a term t'' such that

$$
(t)^{\sharp_3}_T \longrightarrow^n t'' \longleftarrow^{n'} (t')^{\sharp_3}_{T'}
$$

$$
n - n' = \begin{cases} 2 & \text{if the redex of } t \text{ is of the form of application} \\ 1 & \text{otherwise.} \end{cases}
$$

Proof of Lemma 14. Let $t = E[r]$ where r is a redex. By Lemma 20, we define $n \stackrel{\text{def}}{=} \mathrm{s}(E) + 2$ if r is an application, $n \stackrel{\text{def}}{=} \mathrm{s}(E) + 1$ otherwise, and $n' \stackrel{\text{def}}{=} \mathrm{s}(E)$. Then, for each kind of redexes, proof goes straightforwardly: for the redex of application, we use Lemma 19. □

J. Equivalence between Logical Relation and Step-indexed Logical Relation

Here we prove Lemma 18.

In the proof below, we use a notion of values that may include variables, so we extend the definitions of values, evaluation contexts, and reduction as in Figure 16; we use meta-variables $\overline{V}$, $\overline{E}$ for these notions. This extension is consistent since we so far used only closed values, closed evaluation contexts, and reduction for closed terms, and this extension does not change the notions of closed values, closed evaluation contexts, and reduction for closed terms.

Lemma 21. *1. For a variable x and values $\overline{V}$ and $\overline{V}'$, $\overline{V}[x \mapsto \overline{V}']$ is a value.*

2. For a variable x, a evaluation context $\overline{E}$, and a value $\overline{V}$, $\overline{E}[x \mapsto \overline{V}]$ is an evaluation context.

3. For a variable x, terms t and t', and a value $\overline{V}$, if $t \longrightarrow t'$, then $t[x \mapsto \overline{V}] \longrightarrow t'[x \mapsto \overline{V}]$.

Proof. 1 and 2 are clear by induction on $\overline{V}$ and $\overline{E}$, respectively. 3 is also clear from 1 and 2. □

Proof of Lemma 18. If we prove the "if" part, the "only if" part follows from that immediately.

For the "if" part, we prove the contraposition. For simplicity, we omit product types. When τ is order-0, the proof is clear. Let

$$
\begin{aligned}
\tau &= \{\nu_1 : (x_1 \colon \tau_1) \to \tau_1' \mid P_1\} \\
\tau_1' &= \{\nu_2 : (x_2 \colon \tau_2) \to \tau_2' \mid P_2\} \\
&\ \ \vdots \\
\tau_{u-1}' &= \{\nu_u : (x_u \colon \tau_u) \to \tau_u' \mid P_u\} \\
\tau_u' &= \{\nu_{u+1} : \mathbf{int} \mid P_{u+1}\}\ .
\end{aligned}
$$

We assume $\not\models t : \tau$, which is equivalent to the below:

$$
\begin{array}{l}
\exists p \in \{1, \ldots, u\}.\ \exists n_0. \\
\exists! k_1 \leq n_0.\ \exists! V_1'.\ t \longrightarrow^{k_1} V_1' \wedge \\
\qquad \exists n_1 \leq n_0 - k_1.\ \exists V_1.\ \models_{\mathrm{v}}^{n_1} V_1 : \tau_1 \wedge \\
\exists! k_2 \leq n_1.\ \exists! V_2'.\ V_1' V_1 \longrightarrow^{k_2} V_2' \wedge \\
\qquad \exists n_2 \leq n_1 - k_2.\ \exists V_2.\ \models_{\mathrm{v}}^{n_2} V_2 : \tau_2 \wedge \\
\qquad\qquad \ldots \\
\exists! k_p \leq n_{p-1}.\ \exists! V_p'.\ V_{p-1}' V_{p-1} \longrightarrow^{k_p} V_p' \wedge \\
\qquad \exists n_p \leq n_{p-1} - k_p.\ \exists V_p.\ \models_{\mathrm{v}}^{n_p} V_p : \tau_p \wedge \\
\exists! k_{p+1} \leq n_p.\ \exists! A_{p+1}'.\ V_p' V_p \longrightarrow^{k_{p+1}} A_{p+1}' \wedge \\
\left(\begin{array}{l} A_{p+1}' = \mathit{fail}\ \vee \\ \left(\begin{array}{l} A_{p+1}'\text{: value} \wedge \exists! k_{p+2} \leq n_p - k_{p+1}. \\ \exists! A' \neq \mathbf{true}.\ P_{p+1}[\nu_{p+1} \mapsto A_{p+1}'] \longrightarrow^{k_{p+2}} A' \end{array} \right) \end{array} \right)
\end{array}
$$

We show $\not\models^{\mathrm{LR}} t : \tau$, which is equivalent to the below:

$$
\begin{array}{l}
\exists p \in \{1, \ldots, u\}. \\
\exists! V_1'.\ t \longrightarrow^* V_1' \ \wedge\ \exists V_1.\ \models_{\mathrm{v}}^{\mathrm{LR}} V_1 : \tau_1 \wedge \\
\exists! V_2'.\ V_1' V_1 \longrightarrow^* V_2' \ \wedge\ \exists V_2.\ \models_{\mathrm{v}}^{\mathrm{LR}} V_2 : \tau_2 \wedge \\
\qquad\qquad \ldots \\
\exists! V_p'.\ V_{p-1}' V_{p-1} \longrightarrow^* V_p' \ \wedge\ \exists V_p.\ \models_{\mathrm{v}}^{\mathrm{LR}} V_p : \tau_p \wedge \\
\exists! A_{p+1}'.\ V_p' V_p \longrightarrow^* A_{p+1}' \wedge \\
\left(\begin{array}{l} A_{p+1}' = \mathit{fail}\ \vee \\ (A_{p+1}'\text{: value} \wedge \exists! A' \neq \mathbf{true}.\ P_{p+1}[\nu_{p+1} \mapsto A_{p+1}'] \longrightarrow^* A') \end{array} \right)
\end{array}
\tag{26}
$$

Proofs for both the cases that A_{p+1}' is *fail* and that A_{p+1}' is a value are similar, so we show in the latter case. Witnesses for p, V_j', A_{p+1}', and A' are those from the assumption, while from V_j we create ${V_j}^\circ$ that "behaves as V_j in the current context" and fits to the above goal.

Now $\tau_1, \ldots, \tau_u$ are at most order-1; so suppose—ignoring refinement predicates—that

$$
\begin{aligned}
\tau_1 &= \tau_1^1 \to \ldots \tau_{l_1}^1 \to \tau^1 \\
&\ \ \vdots \\
\tau_u &= \tau_1^u \to \ldots \tau_{l_u}^u \to \tau^u
\end{aligned}
$$

where all the types τ_j^i are order-0.

Let $k \stackrel{\text{def}}{=} 1 + \Sigma_{i \in \{1, \ldots, p+2\}} k_i$. From now, we define sequences

$$
\begin{array}{lll}
(V^{(i)})_{i \in \{1, \ldots, h\}} & (\overline{V}^{(i)})_{i \in \{1, \ldots, h\}} & \\
(q^{(i)})_{i \in \{1, \ldots, h\}} & (r^{(i)})_{i \in \{1, \ldots, h\}} & (n^{(i)})_{i \in \{1, \ldots, h\}} \\
(E^{(i)})_{i \in \{p+1, \ldots, h\}} & (m^{(i)})_{i \in \{p+1, \ldots, h\}} & \\
(j^{(i)})_{i \in \{p+1, \ldots, h\}} & (k^{(i)})_{i \in \{p+1, \ldots, h\}} &
\end{array}
\tag{27}
$$

where h is at most k.

For $i \in \{1, \ldots, p\}$, we define $V^{(i)} \stackrel{\text{def}}{=} V_i$, $q^{(i)} \stackrel{\text{def}}{=} i$, $r^{(i)} \stackrel{\text{def}}{=} 1$, and $n^{(i)} \stackrel{\text{def}}{=} n_i$. From the assumption, for $i = 1, \ldots, p$,

$$
\models_{\mathrm{v}}^{n^{(i)}} V^{(i)} : \tau_{r^{(i)}}^{q^{(i)}} \to \ldots \tau_{l_{q^{(i)}}}^{q^{(i)}} \to \tau^{q^{(i)}}\ .
$$

We reserve enough number (at most k) of fresh variables $v_1, v_2, \dots$, and then for each $i \le p$ we define $\overline{V}^{(i)} \stackrel{\text{def}}{=} v_i$ if $V^{(i)}$ has function type, and $\overline{V}^{(i)} \stackrel{\text{def}}{=} V^{(i)}$ if $V^{(i)}$ has order-0 type. Also, we define

$$P \quad \stackrel{\text{def}}{=} \quad \mathbf{let}\ \nu_{p+1} = t\overline{V}^{(1)} \dots \overline{V}^{(p)}\ \mathbf{in}\ P_{p+1}\,.$$

From the assumption,

$$P[v_i \mapsto V^{(i)}]_{i \le p} \longrightarrow^k A'\,.$$

So far, we have defined the sequences (27) for $i \le p$; from now, we define for $i = p+1$. We find "v_j-redexes" of P, i.e., let $(E^{(p+1)}, m^{(p+1)})$ be—if exists—a pair of an evaluation context and an order-0 (closed) value such that

$$P \longrightarrow^* E^{(p+1)}[v_j m^{(p+1)}]$$

for some $j \le p$; such j is unique since the reductions gets stuck here, and we define $j^{(p+1)}$ as such the unique j. In fact, the reductions gets stuck only at this form of redex, and otherwise $P \longrightarrow^* A'$. This is shown by case analysis of the kinds of redexes as below: (i) P has only such free variables that their type is (first-order) function type; (ii) hence, all the redexes can be reduced except for function applications whose function parts are variables; (iii) thus, if there is no stuck of the above form, $P \longrightarrow^* A''$ for some A''; (iv) A'' has type **int** and hence has no (function) variable, so $A'' = A'$ by Lemma 21.

Now we show that the reduction sequence of P is "sufficiently long". First, if $1 < j^{(p+1)}$, for some k'_1 and some value $\overline{V}'_2$

$$V'_1\overline{V}^{(1)} \longrightarrow^{k'_1} \overline{V}'_2$$

since if this reductions gets stuck, $j^{(p+1)}$ becomes 1 by the definition. By Lemma 21,

$$(V'_1\overline{V}^{(1)})[v_1 \mapsto V^{(1)}] = V'_1 V^{(1)} \longrightarrow^{k'_1} \overline{V}'_2[v_1 \mapsto V^{(1)}]$$

and since $V'_1 V^{(1)}$ is closed term,

$$\overline{V}'_2[v_1 \mapsto V^{(1)}] = V'_2 \quad \text{and} \quad k'_1 = k_1\,.$$

Next, if $2 < j^{(p+1)}$, similarly to the above, for some k'_2 and some value $\overline{V}'_3$

$$\overline{V}'_2\overline{V}^{(2)} \longrightarrow^{k'_2} \overline{V}'_3\,.$$

By Lemma 21,

$$(\overline{V}'_2\overline{V}^{(2)})[v_i \mapsto V^{(i)}]_{i=1,2} = V'_2 V^{(2)} \longrightarrow^{k'_2} \overline{V}'_3[v_i \mapsto V^{(i)}]_{i=1,2}$$

and since $V'_2 V^{(2)}$ is closed term,

$$\overline{V}'_3[v_i \mapsto V^{(i)}]_{i=1,2} = V'_3 \quad \text{and} \quad k'_2 = k_2\,.$$

Repeating this, there exist values $\overline{V}'_i (i = 2, \dots, j^{(p+1)})$—let $\overline{V}'_{j^{(p+1)}}$ be of the form $\mathbf{fix}(f, \lambda x.\, t)$—such that

$$\begin{aligned}
P &\longrightarrow^{k_1} \mathbf{let}\ \nu_{p+1} = V'_1\overline{V}^{(1)} \dots \overline{V}^{(p)}\ \mathbf{in}\ P_{p+1}\\
&\longrightarrow^{k_2} \mathbf{let}\ \nu_{p+1} = \overline{V}'_2\overline{V}^{(2)} \dots \overline{V}^{(p)}\ \mathbf{in}\ P_{p+1}\\
&\quad \dots\\
&\longrightarrow^{k_{j^{(p+1)}}} \mathbf{let}\ \nu_{p+1} = \overline{V}'_{j^{(p+1)}}\overline{V}^{(j^{(p+1)})} \dots \overline{V}^{(p)}\ \mathbf{in}\ P_{p+1}\\
&= \mathbf{let}\ \nu_{p+1} = \mathbf{fix}(f, \lambda x.\, t)\overline{V}^{(j^{(p+1)})} \dots \overline{V}^{(p)}\ \mathbf{in}\ P_{p+1}\\
&\longrightarrow \mathbf{let}\ \nu_{p+1} =\\
&\qquad t[x \mapsto v_{j^{(p+1)}}][f \mapsto \overline{V}'_{j^{(p+1)}}]\overline{V}^{(j^{(p+1)}+1)} \dots \overline{V}^{(p)}\\
&\qquad \mathbf{in}\ P_{p+1}\\
&\longrightarrow^* E^{(p+1)}[v_{j^{(p+1)}} m^{(p+1)}]
\end{aligned}$$

Hence, by Lemma 21, there exist $k' \ge 1 + \Sigma_{i \le j^{(p+1)}} k_i$ such that

$$P[v_i \mapsto V^{(i)}]_{i \le p} \longrightarrow^{k'} (E^{(p+1)}[v_i \mapsto V^{(i)}]_{i \le p})[V^{(j^{(p+1)})} m^{(p+1)}]$$

where $E^{(p+1)}[v_i \mapsto V^{(i)}]_{i \le p}$ is a (closed) evaluation context. Since $P[v_i \mapsto V^{(i)}]_{i \in \{1,\dots,p\}} \longrightarrow^k A'$, there exist $A^{(p+1)}$ and $k^{(p+1)} \le k - k' \le \Sigma_{i > j^{(p+1)}} k_i$ such that

$$V^{(j^{(p+1)})} m^{(p+1)} \longrightarrow^{k^{(p+1)}} A^{(p+1)}\,.$$

Since

$$\models_{\mathrm{v}}^{n^{(j^{(p+1)})}} V^{(j^{(p+1)})} : \tau^{q^{(j^{(p+1)})}}_{r^{(j^{(p+1)})}} \to \dots \to \tau^{q^{(j^{(p+1)})}}$$

and from the assumption

$$k^{(p+1)} \le \Sigma_{i > j^{(p+1)}} k_i \le n_{j^{(p+1)}}\,,$$

we have

$$\models_{\mathrm{v}}^{n^{(j^{(p+1)})} - k^{(p+1)}} A^{(p+1)} : \tau^{q^{(j^{(p+1)})}}_{r^{(j^{(p+1)})}+1} \to \dots \to \tau^{q^{(j^{(p+1)})}}\,.$$

Thus, $A^{(p+1)}$ is a value; we define

$$V^{(p+1)} \stackrel{\text{def}}{=} A^{(p+1)} \qquad n^{(p+1)} \stackrel{\text{def}}{=} n^{(j^{(p+1)})} - k^{(p+1)}$$
$$q^{(p+1)} \stackrel{\text{def}}{=} q^{(j^{(p+1)})} \qquad r^{(p+1)} \stackrel{\text{def}}{=} r^{(j^{(p+1)})} + 1$$

then,

$$\models_{\mathrm{v}}^{n^{(p+1)}} V^{(p+1)} : \tau^{q^{(p+1)}}_{r^{(p+1)}} \to \dots \to \tau^{q^{(p+1)}}\,.$$

Also, we define $\overline{V}^{(p+1)} \stackrel{\text{def}}{=} v_{p+1}$ if $V^{(p+1)}$ has function type, and $\overline{V}^{(p+1)} \stackrel{\text{def}}{=} V^{(p+1)}$ if $V^{(p+1)}$ has order-0 type. We have defined the sequence (27) for $i = p+1$.

For the next round, i.e., for $i = p+2$, we find "v_j-redexes" of $E^{(p+1)}[\overline{V}^{(p+1)}]$, i.e., let $(E^{(p+2)}, m^{(p+2)})$ be—if exists—a pair of an evaluation context and an order-0 (closed) value such that

$$E^{(p+1)}[\overline{V}^{(p+1)}] \longrightarrow^* E^{(p+2)}[v_j m^{(p+2)}]$$

for some $j \le p+1$; we define $j^{(p+2)}$ as such the unique j.

Repeating as above, (formally, by induction on i,) for some $h \le k$, we obtain the finite sequences (27) y such that the following holds for any $i \in \{p+1, \dots, h\}$:

1. $j^{(i)} < i$ and
$$E^{(i-1)}[\overline{V}^{(i-1)}] \longrightarrow^* E^{(i)}[v_{j^{(i)}} m^{(i)}]$$
where we regard $E^{(p)}[\overline{V}^{(p+1)}]$ as P; also we have
$$E^{(h)}[\overline{V}^{(h)}] \longrightarrow^* A'\,.$$
2. for some $k' \ge 1 + \Sigma_{j \le j^{(i)}} k_j$,
$$P \longrightarrow^{k'} E^{(i)}[v_{j^{(i)}} m^{(i)}]$$
3. $q^{(i)} = q^{(j^{(i)})} \quad r^{(i)} = r^{(j^{(i)})}+1 \quad n^{(i)} = n^{(j^{(i)})} - k^{(i)} \ge 0$
$$V^{j^{(i)}} m^{(i)} \longrightarrow^{k^{(i)}} V^{(i)}$$
$$\models_{\mathrm{v}}^{n^{(i)}} V^{(i)} : \tau^{q^{(i)}}_{r^{(i)}} \to \dots \to \tau^{q^{(i)}}\,.$$
where if $r^{(i)} = l_{q^{(i)}} + 1$, then $\tau^{q^{(i)}}_{r^{(i)}} \to \dots \to \tau^{q^{(i)}}$ is regarded as $\tau^{q^{(i)}}$.

In the above inductive definition of the sequences, we explain $n^{(j^{(i)})} - k^{(i)} \ge 0$ in the case when $i > p$, which is a bit subtle. First, for any i, 2 above can be shown in the same way as the case when $i = p+1$ above. Now for any natural number e,

$$n^{(i)} = n^{(j^{(i)})} - k^{(i)} = n^{(j^{(j^{(i)})})} - (k^{(j^{(i)})} + k^{(i)}) = \dots$$
$$= n^{(j^{[e](i)})} - (k^{(j^{[e-1](i)})} + \dots + k^{(i)})$$

where $j^{[e](i)} \stackrel{\text{def}}{=} j^{(j^{[e-1](i)})}$ and $j^{[0](i)} \stackrel{\text{def}}{=} i$. Let e be a natural number such that $j^{[e](i)} \leq p$; since now $i > p$, $e > 0$. We prove $n^{(j^{(i)})} - k^{(i)} \geq 0$ by showing

$$k^{(j^{[e-1](i)})} + \dots + k^{(i)} \leq \Sigma_{j > j^{[e](i)}} k_j \leq n^{(j^{[e](i)})} .$$

Since $j^{[e](i)} \leq p$, the right inequality above follows from the assumption of this lemma. By the above 2 where we substitute $j^{[e-1](i)}$ for i, for some $k' \geq 1 + \Sigma_{j \leq j^{[e](i)}} k_j$,

$$P \longrightarrow^{k'} E^{(j^{[e-1](i)})}[v_{j^{[e](i)}} m^{(j^{[e-1](i)})}]$$

and also by the above 3 where we substitute $j^{[e-1](i)}, \dots, i$ for i,

$$V^{j^{[e](i)}} m^{(j^{[e-1](i)})} \longrightarrow^{k^{(j^{[e-1](i)})}} V^{(j^{[e-1](i)})}$$
$$\vdots$$
$$V^{j^{(i)}} m^{(i)} \longrightarrow^{k^{(i)}} V^{(i)}$$

Hence, by Lemma 21, $k' + k^{(j^{[e-1](i)})} + \dots + k^{(i)} \leq k$; thus,

$$k^{(j^{[e-1](i)})} + \dots + k^{(i)} \leq k - k' \leq \Sigma_{j > j^{[e](i)}} k_j .$$

Now, by using the above sequences, we define ${V_j}^\circ$ for the witnesses of our goal. For each i, we define a finite set $D^{(i)} \stackrel{\text{def}}{=} \{i' \mid i = j^{(i')}\}$; by the above 3, for each $i' \in D^{(i)}$,

$$V^{(i)} m^{(i')} \longrightarrow^* V^{(i')} .$$

Let $q \in \{1, \dots, p\}$. We define ${V^{(i)}}^\circ$ for i such that $q^{(i)} = q$. For i such that $q^{(i)} = q$ and $r^{(i)} = l_q + 1$, we define ${V^{(i)}}^\circ \stackrel{\text{def}}{=} V^{(i)}$, which is a value of order-0. Next, by induction on $r = l_q, \dots, 1$, for any i such that $q^{(i)} = q$ and $r^{(i)} = r$, we define ${V^{(i)}}^\circ$ as

$$\begin{aligned} \lambda x.\, \textbf{case}\ x\ \textbf{of}\ \ & m^{(i_1)} \to {V^{(i_1)}}^\circ \\ & \vdots \\ & m^{(i_d)} \to {V^{(i_d)}}^\circ \\ & \_ \to \Omega \end{aligned}$$

where $\{i_1, \dots, i_d\} = D^{(i)}$ and $\Omega \stackrel{\text{def}}{=} \textbf{fix}(f, \lambda x.\, f\, x) \lambda x_{r+1}. \dots x_{l_q}.0$.

Now, we show (26) with ${V^{(i)}}^\circ$ as witnesses of $\exists V_i$ for $i \in \{1, \dots, p\}$.

By the induction by which we defined ${V^{(i)}}^\circ$, it can be easily shown that, for any i,

$$\models^{\text{LR}}_{\text{v}} {V^{(i)}}^\circ : \tau^{q^{(i)}}_{r^{(i)}} \to \dots \to \tau^{q^{(i)}} .$$

Also, by the same induction, it is clear that, for any i and $i' \in D^{(i)}$, ${V^{(i)}}^\circ m^{(i')} \longrightarrow^* {V^{(i')}}^\circ$; thus, for any i, since $E^{(i)}[v_j \mapsto {V^{(j)}}^\circ]_{j \in \leq h}$ is an evaluation context,

$$\begin{aligned} &(E^{(i)}[v_j \mapsto {V^{(j)}}^\circ]_{j \in \leq h})[{V^{j^{(i)}}}^\circ m^{(i)}] \\ \longrightarrow^* &(E^{(i)}[v_j \mapsto {V^{(j)}}^\circ]_{j \in \leq h})[{V^{(i)}}^\circ] . \end{aligned} \tag{28}$$

$$\begin{aligned}
M \text{ (programs)} &::= (x_1, \dots, x_m) \mid \textbf{if}\ x\ \textbf{then}\ M_1\ \textbf{else}\ M_2 \\
&\quad \mid \textbf{let}\ x = e\ \textbf{in}\ M \\
t \text{ (terms)} &::= e \mid \textbf{if}\ x\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \mid \textbf{let}\ x = e\ \textbf{in}\ t \\
e &::= n \mid x \mid \mathsf{op}(x_1, \dots, x_n) \\
&\quad \mid \textbf{fix}(f, \lambda(x_1, \dots, x_n).\, t) \mid f\,(x_1, \dots, x_n) \mid \textbf{fail}
\end{aligned}$$

Figure 17. Normal form before $(-)^{\sharp_1}$

Now, P has the following "reduction":

$$\begin{aligned}
& P \\
\longrightarrow^* \ & E^{(p+1)}[v_{j^{(p+1)}} m^{(p+1)}] \\
\dashrightarrow^* \ & E^{(p+1)}[\overline{V}^{(p+1)}] \\
& \dots \\
\dashrightarrow^* \ & E^{(i-1)}[\overline{V}^{(i-1)}] \\
\longrightarrow^* \ & E^{(i)}[v_{j^{(i)}} m^{(i)}] \\
\dashrightarrow^* \ & E^{(i)}[\overline{V}^{(i)}] \\
& \dots \\
\dashrightarrow^* \ & E^{(h)}[\overline{V}^{(h)}] \\
\longrightarrow^* \ & A'
\end{aligned}$$

where the dashed arrow $\dashrightarrow^*$ means that, if we substitute ${V^{(i)}}^\circ$ for v_i for all i, there is the reductions by (28). Thus, by the substitution and Lemma 21, we have

$$\begin{aligned}
& \textbf{let}\ \nu_{p+1} = t {V^{(1)}}^\circ \dots {V^{(p)}}^\circ\ \textbf{in}\ P_{p+1} \\
= \ & P[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h} \\
\longrightarrow^* \ & (E^{(p+1)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(j^{(p+1)})}}^\circ m^{(p+1)}] \\
\dashrightarrow^* \ & (E^{(p+1)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(p+1)}}^\circ] \\
& \dots \\
\dashrightarrow^* \ & (E^{(i-1)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(i-1)}}^\circ] \\
\longrightarrow^* \ & (E^{(i)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(j^{(i)})}}^\circ m^{(i)}] \\
\dashrightarrow^* \ & (E^{(i)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(i)}}^\circ] \\
& \dots \\
\dashrightarrow^* \ & (E^{(h)}[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h})[{V^{(h)}}^\circ] \\
\longrightarrow^* \ & A'[v_j \mapsto {V^{(i)}}^\circ]_{j \leq h} \ = \ A'
\end{aligned}$$

where recall that

$$\overline{V}^{(i)} = V^{(i)} = {V^{(i)}}^\circ$$

if the type is order-0. This completes the proof. □

K. Definition of $(-)^{\sharp'}$

In this section, we define this refined transformation $(-)^{\sharp'}$. The refinement is needed for both $(-)^{\sharp_3}$ and $(-)^{\sharp_4}$, so we define $(-)^{\sharp'_{34}}$, which is a modification of $((-)^{\sharp_3})^{\sharp_4}$, and define $(-)^{\sharp'}$ as $((-)^{\sharp_1})^{\sharp'_{34}}$. Note that $(-)^{\sharp_2}$ is identity on terms.

For the simplicity of the definition, we assume without loss of generality that input programs of $(-)^{\sharp'}$ (and hence $(-)^{\sharp_1}$) are in a variant of A-normal form defined in Figure 17. Here, we write $\lambda(x_1, \dots, x_n).\, t$ for $\lambda x.\, \textbf{let}\ x_1 = \textbf{pr}_1 x\ \textbf{in}\ \dots \textbf{let}\ x_n = \textbf{pr}_n x\ \textbf{in}\ t$ where x is a fresh variable.

We redefine the transformation $(-)^{\sharp_1}$ according to the normal form; the essential part of the new definition of $(-)^{\sharp_1}$ is shown in

$$
\begin{aligned}
&(\mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, t'))^{\sharp_1} \overset{\mathrm{def}}{=} \mathbf{fix}(f, \lambda x.\, ((t')^{\sharp_1}, x_1, \ldots, x_n)) \\
&(\mathbf{let}\ x = f\,(x_1, \ldots, x_n)\ \mathbf{in}\ t)^{\sharp_1} \overset{\mathrm{def}}{=} \\
&\qquad \mathbf{let}\ z = f\,(x_1, \ldots, x_n)\ \mathbf{in}\ \mathbf{let}\ x = \mathbf{pr}_1 z\ \mathbf{in} \\
&\qquad \mathbf{let}\ x'_1 = \mathbf{pr}_1(\mathbf{pr}_2 z)\ \mathbf{in} \ldots \mathbf{let}\ x'_n = \mathbf{pr}_n(\mathbf{pr}_2 z)\ \mathbf{in} \\
&\qquad (t)^{\sharp_1}\,[x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n]
\end{aligned}
$$

Figure 18. $(-)^{\sharp_1}$ for normal forms

$$
\begin{aligned}
t &::= (x_1, \ldots, x_m) \mid \mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{let}\ x = e\ \mathbf{in}\ t \\
e &::= n \mid \mathtt{op}(x_1, \ldots, x_n) \mid \mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, t) \\
&\quad\ \mid f\,(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \mid \mathbf{pr}_i x \mid \mathbf{fail}
\end{aligned}
$$

Figure 19. Normal form before $(-)^{\sharp'_{34}}$

Figure 18. Output programs of this $(-)^{\sharp_1}$ are in another normal form defined in Figure 19, which is the domain of the transformation $(-)^{\sharp'_{34}}$, which is defined in the next subsection.

K.1 Definition of $(-)^{\sharp'_{34}}$ and Soundness of $(-)^{\sharp'}$

Here, we formalize the idea explained in Section 4.2 as a transformation $(-)^{\sharp'_{34}}$.

The transformation of $(-)^{\sharp'_{34}}$ for types is the same as $((-)^{\sharp_3})^{\sharp_4}$. Figure 20 shows the definition of $(-)^{\sharp'_{34}}_T$ on terms. The definition uses an auxiliary function `InstVar` defined below, and this is the essential part: `InstVar` synthesizes new applications (like $\mathtt{fg}\,(x, x)$) and inserts the assumptions illustrated above. In the figure, B is a set of bindings (i.e., pairs of variables and expressions) that are used in `InstVar`. As $(-)^{\sharp_3}_T$, $(-)^{\sharp'_{34}}_T$ also depends on a multiplicity annotation T. Since occurrences of subterms of t correspond variables in the normal form of t, a multiplicity annotation T is defined as a function from *variables* to multiplicities. The rules are almost the same except for applications, where we insert assumptions related to the function and its arguments by using `InstVar`.

Now, we define the auxiliary function `InstVar`:

$$
\mathtt{InstVar}(g, T, B, t) \overset{\mathrm{def}}{=}
$$

$$
\mathbf{let}\ g' = \left(\begin{array}{l}
\lambda\widetilde{y}.\ \mathbf{let}\ x = g\widetilde{y}\ \mathbf{in} \\
\quad \mathbf{let}\ w^{\langle(\alpha^1_j)_j\rangle} = B^\sharp_g((\alpha^1_j)_j)\ \mathbf{in} \\
\quad \cdots \\
\quad \mathbf{let}\ w^{\langle(\alpha^k_j)_j\rangle} = B^\sharp_g((\alpha^k_j)_j)\ \mathbf{in} \\
\quad \mathbf{assume}\,(p)\,;\mathbf{assume}\,(p')\,; x
\end{array}\right) \mathbf{in}\ t[g \mapsto g'] \tag{29}
$$

where $(\alpha^i_j)_j \in \prod_{j \in m} \mathtt{App}^*_j$ and $k = \left|\prod_{j \in m} \mathtt{App}^*_j\right|$; the function $B^\sharp_g$, the two formulas p and p', the set $\mathtt{App}^*_j$ and the variables $w^{\langle(\alpha^i_j)_j\rangle}$ are defined below.

Before the formal definition of the predicates p and p', we explain their semantical meaning. By $(-)^{\sharp_3}$ and $(-)^{\sharp_4}$, each *variable* is unchanged (i.e., $(x)^{\sharp_3} \overset{\mathrm{def}}{=} x$ and $(x)^{\sharp_4} \overset{\mathrm{def}}{=} x$), although by $(-)^{\sharp_3}$, each *subterm* of a function type (i.e., $\mathbf{fix}(f, \lambda x.\, t)$ and f in $\mathbf{fix}(f, \lambda x.\, t)$) is duplicated, and by $(-)^{\sharp_4}$, each *subterm* of a tuple type (i.e., $(t_1, \ldots, t_n, t'_1, \ldots, t'_m)$) is transformed to the product of the functions, where *the product of functions* t *and* t' means $\lambda(x, x').\, (t\,x, t\,x')$. Hence, a verifier cannot necessarily infer that

$$
\begin{aligned}
&(t)^{\sharp'_{34}}_T \overset{\mathrm{def}}{=} (t)^{\sharp'_{34}}_{T,\emptyset} \\[1ex]
&((x_1, ..., x_n, f_1, ..., f_m))^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \\
&\qquad (x_1, ..., x_n, \lambda(y_1, ..., y_m).\, (f_1\, y_1, ..., f_m\, y_m)) \\
&(\mathbf{if}\ x\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \mathbf{if}\ x\ \mathbf{then}\ (t_1)^{\sharp'_{34}}_{T,B}\ \mathbf{else}\ (t_2)^{\sharp'_{34}}_{T,B} \\
&(\mathbf{let}\ x = e\ \mathbf{in}\ t)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \mathbf{let}\ x = (e)^{\sharp'_{34}}_{T,B}\ \mathbf{in}\ (t)^{\sharp'_{34}}_{T,B} \\
&\qquad \text{if } e = n, \mathtt{op}(\widetilde{x}), (\widetilde{x}, \widetilde{f}), \text{ or } \mathbf{fail} \\
&(\mathbf{let}\ x = e\ \mathbf{in}\ t)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \mathbf{let}\ x = (e)^{\sharp'_{34}}_{T,B}\ \mathbf{in}\ (t)^{\sharp'_{34}}_{T,B \cup \{x = e\}} \\
&\qquad \text{if } e = f\,(\widetilde{x}, \widetilde{g}), \text{ or } \mathbf{pr}_i x \\
&\big(\mathbf{let}\ x = \mathbf{fix}(f, \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m).\, t')\ \mathbf{in}\ t\big)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \\
&\qquad \mathbf{let}\ x = \mathbf{fix}(f, \lambda(z_1, \ldots, z_{T(x)}).\, (t'_1, \ldots, t'_{T(x)}))\ \mathbf{in}\ (t)^{\sharp'_{34}}_{T,B} \\
&\qquad \text{where } t'_k \overset{\mathrm{def}}{=} (t')^{\sharp'_{34}}_{T,B}\,[x_i \mapsto \mathbf{pr}_i z_k]_{i \le n}[g_j \mapsto p^{z_k}_j]_{j \le m} \\
&\qquad\qquad p^{z_k}_j \overset{\mathrm{def}}{=} \lambda y_j.\, \mathbf{pr}_j((\mathbf{pr}_{n+1} z_k)(\overrightarrow{\bot}^{j-1}, y_j, \overrightarrow{\bot}^{m-j})) \\[1ex]
&(e)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} e \qquad \text{if } e = n, x, \mathtt{op}(\widetilde{x}), \text{ or } \mathbf{fail} \\
&(f(x_1, ..., x_n, g_1, ..., g_m))^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \mathtt{InstVar}(f, T, B, t_{m+1}) \\
&\qquad \text{where } z \overset{\mathrm{def}}{=} (x_1, ..., x_n, \lambda\widetilde{y_j}.\, (g_1\, y_1, ..., g_m\, y_m)) \\
&\qquad\qquad t_1 \overset{\mathrm{def}}{=} \mathbf{pr}_1(f(\overrightarrow{z}^{T(f)})) \\
&\qquad\qquad t_{j+1} \overset{\mathrm{def}}{=} \mathtt{InstVar}(g_j, T, B, t_j) \quad (\text{for } j = 1, ..., m) \\
&((x_1, ..., x_n, f_1, ..., f_m))^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \\
&\qquad (x_1, ..., x_n, \lambda(y_1, ..., y_m).\, (f_1\, y_1, ..., f_m\, y_m)) \\
&\left(\mathbf{pr}^{\mathbf{int}}_i x\right)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \mathbf{pr}_i\, x \\
&(\mathbf{pr}^{\rightarrow}_j x)^{\sharp'_{34}}_{T,B} \overset{\mathrm{def}}{=} \lambda y.\, \mathbf{pr}_j((\mathbf{pr}_{n+1} x)(\overrightarrow{\bot}^{j-1}, y_j, \overrightarrow{\bot}^{m-j}))
\end{aligned}
$$

where n and m are the numbers of the integer components and the function type components in the simple type of x, respectively.

Figure 20. Refined encoding function refinement $(-)^{\sharp'_{34}}$

function *variables* behave as the product of duplicated functions, while they in fact behave so since they are instantiated with some closed ground *terms* in programs. The assumed predicates p and p' state just that all the function variables behave as the product of duplicated functions (p and p' correspond to "product of functions" and "duplication", respectively).

Now let us return to the definition, which consists of the following five steps.

1. Let z be z' if $(g = \mathbf{pr}^{\rightarrow}_k z') \in B$ for some (unique) k and z', or be g otherwise.

   Note that the types of variables such as g and z might become different after the encoding $(-)^{\sharp'}$. Before applying $(-)^{\sharp'}$, the simple type of z is of the following form:

$$
\mathbf{int}^n \times \prod_{j=1}^{m} (\tau_j \to \tau'_j)
$$

and hence the type of g is of the form $\tau_k \to \tau'_k$; in the case $z = g$, we regard $n = 0$ and $k = m = 1$. After $(-)^{\sharp'}$, $(z)^{\sharp'}$ $(= z)$ has the following simple type:

$$\mathbf{int}^n \times \left(\prod_{j=1}^{m} \left((\tau_j)^{\sharp'}_T \right)^{\mathtt{m}_j} \to \prod_{j=1}^{m} \left((\tau'_j)^{\sharp'}_T \right)^{(\mathtt{m}_j)}_{[\cdot \mapsto \cdot]} \right) \tag{30}$$

where $\mathtt{m}_j$ is the multiplicity of $\mathbf{pr}^{\to}_j z$, and we access to the second component (function part) by $\mathbf{pr}^{\sharp}_{\to}$. Also, after $(-)^{\sharp'}$, the type of g becomes $\left((\tau_k)^{\sharp'}_T \right)^{\mathtt{m}_k} \to \left((\tau'_k)^{\sharp'}_T \right)^{\mathtt{m}_k}$ (because of the consistency of T, we can show that $T(g) = \mathtt{m}_k$). Hence, the type of $\widetilde{y} = (y_l)_{l \in \mathtt{m}_k}$ is $\left((\tau_k)^{\sharp'}_T \right)^{\mathtt{m}_k}$. Note that variables introduced when defining $(-)^{\sharp'}$, such as y_l, have no "types before $(-)^{\sharp'}$".

2. For each $j = 1, \ldots, m$, we define

$$\mathtt{App}'_j \stackrel{\text{def}}{=} \left\{ (u, v, w) \;\middle|\; \begin{array}{l} (v = \mathbf{pr}^{\to}_j z), (w = vu) \in B, \\ depth(v) = 1 \end{array} \right\}$$

and then define "application information" of z at j:

$$\mathtt{App}_j \stackrel{\text{def}}{=} \begin{cases} \mathtt{App}'_j \cup \{y_1, \ldots, y_{\mathtt{m}_k}\} & (\text{if } j = k) \\ \mathtt{App}'_j & (\text{otherwise, if } \mathtt{App}'_j \text{ is non-empty}) \\ \{(\bot, v) \mid (v = \mathbf{pr}^{\to}_j z) \in B\} & (\text{otherwise}) \end{cases}$$

where y is the bound variable in (29).

Let $\mathtt{Term}$ be the set of all terms. We define "argument part" as

$$\mathtt{arg}_j : \mathtt{App}_j \to \mathtt{Term} \quad \stackrel{\text{def}}{=} \quad \begin{cases} (u, v, w) & \mapsto u \\ y_l & \mapsto y_l \\ (\bot, v) & \mapsto \bot \end{cases}$$

and "function part" as

$$\mathtt{fun}_j : \mathtt{App}_j \to \mathtt{Term} \quad \stackrel{\text{def}}{=} \quad \begin{cases} (u, v, w) & \mapsto v \\ y_l & \mapsto g \\ (\bot, v) & \mapsto v. \end{cases}$$

3. We further add the information of multiplicity $\mathtt{m}_j$ for the notions thus defined:

$$\mathtt{App}^*_j \stackrel{\text{def}}{=} \{ (a_i)_{i \in \mathtt{m}_j} \in \mathtt{App}_j{}^{\mathtt{m}_j} \mid \mathtt{fun}_j(a_i) = \mathtt{fun}_j(a_1) \}$$

$$\mathtt{arg}^*_j : \mathtt{App}^*_j \to \mathtt{Term}^{\mathtt{m}_j}, \quad \mathtt{arg}^*_j((a_i)_i) \stackrel{\text{def}}{=} (\mathtt{arg}_j(a_i))_i$$

$$\mathtt{fun}^*_j : \mathtt{App}^*_j \to \mathtt{Term}, \quad \mathtt{fun}^*_j((a_i)_i) \stackrel{\text{def}}{=} \mathtt{fun}_j(a_1).$$

4. We define "encoding of functions around g":

$$B^{\sharp}_g : \textstyle\prod_{j \in m} \mathtt{App}^*_j \to \prod_{j \in m} \mathtt{Term}$$

$$B^{\sharp}_g((\alpha_j)_j) \stackrel{\text{def}}{=} \Big(\mathbf{pr}_j \big((\mathbf{pr}^{\sharp}_{\to} z)(\mathtt{arg}^*_1 \alpha_1, \ldots, \mathtt{arg}^*_m \alpha_m) \big) \Big)_{j \in m}.$$

5. Finally, we define p and p' in (29) as

$$p \stackrel{\text{def}}{=} \mathop{\&}_{\substack{(\alpha_j)_j, (\alpha'_j)_j \in \prod_{j \in m} \mathtt{App}^*_j, \\ j \in m,\ \mathtt{fun}^*_j(\alpha_j) = \mathtt{fun}^*_j(\alpha'_j)}} \left(\begin{array}{l} \mathtt{arg}^*_j(\alpha_j) = \mathtt{arg}^*_j(\alpha'_j) \ \texttt{=>} \\ \mathbf{pr}_j w^{\langle ((\alpha_j)_j) \rangle} = \mathbf{pr}_j w^{\langle ((\alpha'_j)_j) \rangle} \end{array} \right)$$

$$p' \stackrel{\text{def}}{=} \mathop{\&}_{\substack{(\alpha_j)_j \in \prod_{j \in m} \mathtt{App}^*_j, \\ j \in m,\ i \in \mathtt{m}_j,\ \mathbf{pr}_i \alpha_j = (u, v, w)}} \left(w = \mathbf{pr}_i \mathbf{pr}_j w^{\langle ((\alpha_j)_j) \rangle} \right)$$

where we prepare a fresh variable $w^{\langle (\alpha_j)_j \rangle}$ for each application $(\alpha_j)_j$.

L. Soundness of Verification by $(-)^{\sharp'}$

Here we prove the soundness of verification by $(-)^{\sharp'}$.

First we remark that the difference between $(-)^{\sharp'}$ and $(-)^{\sharp}$ is just the assume-expressions by $\mathtt{InstVar}(-)$. For any terms t, t', $\mathbf{assume}\,(t)\,; t' \leq_{\mathrm{o}} t'$ (since $\mathbf{assume}\,(\mathbf{fail})$ diverges); hence for any term t, $(t)^{\sharp'} \leq_{\mathrm{o}} (t)^{\sharp}$, and so

$$\models (t)^{\sharp}_T : (\tau)^{\sharp}_{\phi} \quad \text{implies} \quad \models (t)^{\sharp'}_T : (\tau)^{\sharp}_{\phi}\ .$$

Theorem 22 (Soundness of Verification by $(-)^{\sharp'}$)**.** *Let t be a closed term and τ be a type of at most order-2. Let T be a multiplicity annotation for $((t)^{\sharp_1})^{\sharp_2}$ and ϕ be a multiplicity type for $((\tau)^{\sharp_1})^{\sharp_2}$, and suppose that they are consistent and $\tau \leq_{mul} \phi$. Then,*

$$\models (t)^{\sharp'}_T : (\tau)^{\sharp}_{\phi} \qquad \textit{implies} \qquad \models t : \tau\ .$$

Proof. From now, we reduce the proof to Lemma 23; this reduction part is almost the same as the proof of Theorem 1, so we describe only essential points, simplifying the setting. Let $\tau = \tau_1 \to \mathbf{int}$ where τ_1 is order-1, and for given V_1 such that $\models^{\mathrm{LR}}_{\mathrm{v}} V_1 : \tau_1$, we prove $\models^{\mathrm{LR}} t\, V_1 : \mathbf{int}$.

By Lemma 16, we have $\models^{\mathrm{LR}}_{\mathrm{v}} (V_1)^{\sharp} : (\tau_1)^{\sharp}$, hence $\models^{\mathrm{LR}}_{\mathrm{v}} (V_1)^{\sharp'} : (\tau_1)^{\sharp}$. By the assumption that $\models^{\mathrm{LR}} (t)^{\sharp'}_T : (\tau)^{\sharp}_{\phi}$, we have $\models^{\mathrm{LR}}_{\mathrm{v}} (t)^{\sharp'} (V_1)^{\sharp'} : (\mathbf{int})^{\sharp}$. Now, $(t\, V_1)^{\sharp'} \leq_{\mathrm{o}} (t)^{\sharp'} (V_1)^{\sharp'}$ since the left hand side is the right hand side plus assume expressions; hence, $\models^{\mathrm{LR}} (t\, V_1)^{\sharp'} : (\mathbf{int})^{\sharp}$. Since $(\mathbf{int})^{\sharp}$ is order-0, by Lemma 23 below, we have $\models^{\mathrm{LR}} (t\, V_1)^{\sharp} : (\mathbf{int})^{\sharp}$, and by Lemma 17, $\models^{\mathrm{LR}} t\, V_1 : \mathbf{int}$. ☐

Lemma 23. *For any closed A-normal form t (defined in Figure 17), a type τ of order-0, and a consistent pair of a multiplicity annotation T of t and a multiplicity type ϕ over τ, $(t)^{\sharp'}_T$ and $(t)^{\sharp}_T$ are observationally equivalent; and hence,*

$$\models (t)^{\sharp'}_T : (\tau)^{\sharp'}_{\phi} \qquad \textit{iff} \qquad \models (t)^{\sharp}_T : (\tau)^{\sharp}_{\phi}.$$

Proof. Here we give only an overview of our proof and an example to explain our intuitive idea; for meticulous readers, we give a formal proof in the rest of this section. In this proof, by "A-normal forms" we mean those defined in Figure 19 (rather than Figure 17).

First, we define $(-)^{\sharp_{34}}_T$ by eliminating $\mathtt{InstVar}$ from $(-)^{\sharp'_{34}}_T$; i.e., in Figure 20, we drop the subscript B and replace the case of application with the below

$$(f(x_1, \ldots, x_n, g_1, \ldots, g_m))^{\sharp_{34}}_T \stackrel{\text{def}}{=} \mathbf{pr}_1(f(\overrightarrow{z}^{T(f)}))$$

where

$$z \stackrel{\text{def}}{=} (x_1, \ldots, x_n, \lambda \widetilde{y_j}.\, (g_1\, y_1, ..., g_m\, y_m))\ .$$

Since $(-)^{\sharp_{34}}_T$ is just an A-normal form version of $((-)^{\sharp_3}_T)^{\sharp_4}$, it suffices for the lemma to prove that $(t)^{\sharp_{34}}_T$ is observationally equivalent to $(t)^{\sharp'_{34}}_T$ for any ground closed A-normal form t. That is, we will prove that assume expressions inserted by $\mathtt{InstVar}(-)$ are satisfied and hence can be removed without changing the meaning.

The assume expressions inserted by $\mathtt{InstVar}(-)$ are properties satisfied naturally by the image of $(-)^{\sharp_{34}}$, and in fact, it is easy to prove that $(V)^{\sharp'_{34}}$ satisfies the properties by unfolding the definition of $(-)^{\sharp'_{34}}$ in Figure 20. However it is not obvious if such V are arbitrary terms e, so we transform such $(e)^{\sharp'_{34}}$ to a term of the form $(V)^{\sharp'_{34}}$. We call this transformation *N-reduction*; it is defined similarly to evaluation, but keeps the form of A-normal form.

In order for N-reduction to terminate, we can assume that the given whole (ground closed) term t terminates, because when t diverges, by Lemma 15 and since $(t)^{\sharp'_{34}} \leq_{\mathrm{o}} (t)^{\sharp_{34}}$, both $(t)^{\sharp_{34}}$ and $(t)^{\sharp'_{34}}$ diverge and then the current lemma holds. Since N-reduction is simulated by the evaluation, if evaluation terminates, N-reduction also terminates.

Though intuitively we reduce $(t)^{\sharp'_{34}}$, in fact we define N-reduction for t, and we show $(-)^{\sharp'_{34}}$ preserves N-reduction to observational equivalence, i.e.,

$$t \longrightarrow_{\mathrm{N}} t' \quad \text{implies} \quad (t)^{\sharp'_{34}} =_{\mathrm{o}} \left(t'\right)^{\sharp'_{34}} .$$

Also $(-)^{\sharp_{34}}$ preserves N-reduction to observational equivalence. Now since N-reduction terminates for given t, we have the normal form t', and for the normal form of N-reduction, it is easy to show

$$\left(t'\right)^{\sharp'_{34}} =_{\mathrm{o}} \left(t'\right)^{\sharp_{34}} .$$

Thus we can show

$$(t)^{\sharp'_{34}} =_{\mathrm{o}} \left(t'\right)^{\sharp'_{34}} =_{\mathrm{o}} \left(t'\right)^{\sharp_{34}} =_{\mathrm{o}} (t)^{\sharp_{34}} .$$

In the rest of this overview, we explain the above idea concretely with the following example of t:

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda x'.\, t')\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = f x\ \mathbf{in}\ t''\end{aligned} \tag{31}$$

where $T(f)$ is, say, 2.

Now $(t)^{\sharp'_{34}}$ is

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\, ((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y =\\ &\quad \mathbf{let}\ f'' = \lambda(y_1, y_2).\, \big(\mathbf{let}\ w = f(y_1, y_2)\ \mathbf{in}\ \mathbf{assume}\,(\ldots)\,; w\big)\\ &\quad \mathbf{in}\ \mathbf{pr}_1(f''(x, x))\\ &\mathbf{in}\ \left(t''\right)^{\sharp'_{34}} .\end{aligned} \tag{32}$$

Since f in t is bound to the value $\mathbf{fix}(f', \lambda x'.\, t')$, we could calculate by the definition in Figure 20 that (the body of) f in $(t)^{\sharp'_{34}}$ is syntactically the product

$$\lambda(x'_1, x'_2).\, \left(\left(t'\right)^{\sharp'_{34}}[x' \mapsto x'_1], \left(t'\right)^{\sharp'_{34}}[x' \mapsto x'_2]\right)$$

of the duplication of $\mathbf{fix}(f', \lambda x'.\, t')$ (it is not the case if f in t is bound to a non-value). Then, it is easy to show that such the syntactical product of the duplication satisfies $\mathbf{assume}\,(\ldots)$ (see Appendix L.4 for details; especially, Lemma 27); here recall that, as we explained in Section K.1, the predicates of the assume-expressions inserted by $\mathtt{InstVar}(-)$ just state that all the function variables after applying $(-)^{\sharp'_{34}}$ behave as the product of duplicated functions. Thus, we can remove the assume expression, and by simple reductions, we have

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\, ((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = \mathbf{pr}_1(f(x, x))\ \mathbf{in}\ \left(t''\right)^{\sharp'_{34}} .\end{aligned} \tag{33}$$

Now we want to transform the non-value $\mathbf{pr}_1(f(x,x))$ to the form $(V)^{\sharp'_{34}}$, as f was so and it helped the removal of $\mathbf{assume}$ as above.

Clearly, the above is observationally equivalent to

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\, ((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = \mathbf{pr}_1(((t')^{\sharp'_{34}}\,[x' \mapsto x][f' \mapsto f])_{i=1,2})\ \mathbf{in}\ \left(t''\right)^{\sharp'_{34}} .\end{aligned}$$

Since our language is deterministic, $\mathbf{pr}_1(t, t) =_{\mathrm{o}} t$ for any term t (Lemma 26); hence the above term is equivalent to

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\, ((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = \left(t'\right)^{\sharp'_{34}}[x' \mapsto x][f' \mapsto f]\ \mathbf{in}\ \left(t''\right)^{\sharp'_{34}} .\end{aligned} \tag{34}$$

We define N-reduction so that it reduces (31) to the following

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda x'.\, t')\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = t'[x' \mapsto x][f' \mapsto f]\ \mathbf{in}\ t'' .\end{aligned} \tag{35}$$

It is clear that $(-)^{\sharp'_{34}}$ of (35) becomes (34); thus, $(-)^{\sharp'_{34}}$ preserves N-reduction to observational equivalence.

As above, N-reduction is just evaluation but it keeps the form of A-normal form. Repeating this N-reduction, $f x$ in (31) becomes some value V, and $\mathbf{pr}_1(f(x,x))$ in (33) becomes $(V)^{\sharp'_{34}}$.

Repeating N-reduction, we finally get its normal form of the following form

$$\begin{aligned}&\mathbf{let}\ x_1 = V_1\ \mathbf{in}\\ &\ldots\\ &\mathbf{let}\ x_n = V_n\ \mathbf{in}\ x_i .\end{aligned}$$

Applying $(-)^{\sharp'_{34}}$, we have

$$\begin{aligned}&\mathbf{let}\ x_1 = (V_1)^{\sharp'_{34}}\ \mathbf{in}\\ &\ldots\\ &\mathbf{let}\ x_n = (V_n)^{\sharp'_{34}}\ \mathbf{in}\ x_i\end{aligned}$$

and applying $(-)^{\sharp_{34}}$, we have

$$\begin{aligned}&\mathbf{let}\ x_1 = (V_1)^{\sharp_{34}}\ \mathbf{in}\\ &\ldots\\ &\mathbf{let}\ x_n = (V_n)^{\sharp_{34}}\ \mathbf{in}\ x_i .\end{aligned}$$

Since now x_i has a ground type, so does V_i; hence $(V_i)^{\sharp'_{34}} = (V_i)^{\sharp_{34}} = V_i$. Therefore, the two normal forms are observationally equivalent, and so are $(t)^{\sharp'_{34}}$ and $(t)^{\sharp_{34}}$.

In the rest of this section, we give formal definitions and proofs based on the above idea.

L.1 N-reduction

From now, we define N-reduction. Though N-reduction reduces terms before applying $(-)^{\sharp'_{34}}$, our intuitive idea is to transform terms after applying $(-)^{\sharp'_{34}}$ as above; so we put labels to A-normal forms to track the information of the sets B in the definition of $(-)^{\sharp'_{34}}_{T,B}$. (Alternatively, we may be able to equivalently consider reduction for $(t)^{\sharp'_{34}}_{T,B}$ as "polynomial", i.e., not as the application of $(-)^{\sharp'_{34}}_{T,B}$ to t but as a formal term consisting of T, B, and t.)

We define *labeled A-normal forms* in Figure 21, where we fix a countable set of *labels* and we use b as a meta-variable for labels. If we drop all labels in labeled terms d and s, we obtain terms e and t defined in Figure 19, respectively. We implicitly use this label-dropping transformation to apply notions for t to s.

$$
\begin{aligned}
s &::= (x_1, \ldots, x_n) \mid \mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2} \mid \mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2} \\
d &::= n \mid \mathsf{op}(x_1, \ldots, x_n) \mid \mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, s^b) \\
&\quad \mid f\,(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \mid \mathbf{pr}_i x \mid \mathbf{fail}
\end{aligned}
$$

Figure 21. Labeled A-normal forms

$$
\begin{aligned}
&R[\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}] \longrightarrow_{\mathrm{N}} \\
&\qquad \begin{cases} R[s_1] & (\text{if } x \rightsquigarrow_R 0^{b'}) \\ R[s_2] & (\text{if } x \rightsquigarrow_R m^{b'}, m \neq 0) \end{cases} \\
&R[\mathbf{let}\ y = \mathsf{op}(x_1, \ldots, x_n)^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} \\
&\qquad R[\mathbf{let}\ y = ([\![\mathsf{op}]\!](m_1, \ldots, m_n))^{b_1}\ \mathbf{in}\ s^{b_2}] \\
&\qquad \text{where } x_i \rightsquigarrow_R {m_i}^{b'_i} \\
&R[\mathbf{let}\ y = (f\,(x_1, \ldots, x_n, g_1, \ldots, g_m))^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} \\
&\qquad R[\mathbf{cclet}\ y = \left(s'[x'_i \mapsto x_i]_i[g'_j \mapsto g_j]_j[f' \mapsto f]\right)^{b''}\ \mathbf{in}\ s^{b_2}] \\
&\qquad \text{where } f \rightsquigarrow_R \mathbf{fix}(f', \lambda(x'_1, \ldots, x'_n, g'_1, \ldots, g'_m).\, {s'}^{b''})^{b'}. \\
&R[\mathbf{let}\ y = (\mathbf{pr}_i x)^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{let}\ y = {U_i}^{b'_i}\ \mathbf{in}\ s^{b_2}] \\
&\qquad \text{where } x \rightsquigarrow_R (x_1, \ldots, x_n)^{b'}, x_i \rightsquigarrow_{R|_x} {U_i}^{b'_i} \\
&R[\mathbf{let}\ y = \mathbf{fail}^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} \mathbf{fail}
\end{aligned}
$$

Figure 22. N-reduction rules

We define *labeled value* U as

$$U ::= n \mid \mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, s^b) \mid (x_1, \ldots, x_n)$$

and *N-reduction context* R as

$$R ::= [\,] \mid \mathbf{let}\ x = U^{b_1}\ \mathbf{in}\ R^{b_2}\ .$$

For the definition of N-reduction, we need to prepare one relation: for R, x, and U^b, we write $x \rightsquigarrow_R U^b$ if x refers U^b in R; precisely, $x \rightsquigarrow_R U^b$ is defined as below.

$$
\begin{aligned}
&x \rightsquigarrow_{[\,]} U^b \overset{\text{def}}{\iff} \text{false} \\
&x \rightsquigarrow_{\mathbf{let}\ x' = {U'}^{b'_1}\ \mathbf{in}\ R^{b'_2}} U^b \overset{\text{def}}{\iff} \\
&\qquad x \rightsquigarrow_R U^b \text{ or } \left(x \not\rightsquigarrow_R {U''}^{b''} \text{ for any } {U''}^{b''}, x = x', U^b = {U'}^{b'_1}\right)
\end{aligned}
$$

Given $x \rightsquigarrow_R U^b$, U^b is uniquely determined from R and x, and there is the unique triple of b' and N-reduction contexts $R|_x$ and R' such that

$$R = R|_x[\mathbf{let}\ x = U^b\ \mathbf{in}\ {R'}^{b'}] \quad \text{and} \quad x \not\rightsquigarrow_{R'} U^b\ .$$

We can use $R|_x$ as a context for U.

Any closed labeled A-normal form s is in exactly one of the following three cases.

- $R[(x_1, \ldots, x_n)]$
- $R[\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}]$
- $R[\mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ s^{b_2}]$
  where $d = \mathsf{op}(x_1, \ldots, x_n),\ f\,(x_1, \ldots, x_n),\ \mathbf{pr}_i x,\ \mathbf{fail}$

We will think the first case as normal forms of N-reduction; so we define N-reduction rules for the other two cases.

We define N-reduction rules in Figure 22. Here, **cclet** is a kind of commuting-conversion of **let**: for labels b, b', a variable y, and

$$
\begin{aligned}
&\mathbf{cclet}\ y = (x_1, \ldots, x_n)^b\ \mathbf{in}\ {s'}^{b'} \overset{\text{def}}{=} \\
&\quad \mathbf{let}\ y = (x_1, \ldots, x_n)^b\ \mathbf{in}\ {s'}^{b'} \\
&\mathbf{cclet}\ y = (\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2})^b\ \mathbf{in}\ {s'}^{b'} \overset{\text{def}}{=} \\
&\quad \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{cclet}\ y = {s_1}^b\ \mathbf{in}\ {s'}^{b'})^{b_1} \\
&\qquad\quad \mathbf{else}\ (\mathbf{cclet}\ y = {s_2}^b\ \mathbf{in}\ {s'}^{b'})^{b_2} \\
&\mathbf{cclet}\ y = (\mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2})^b\ \mathbf{in}\ {s'}^{b'} \overset{\text{def}}{=} \\
&\quad \mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ (\mathbf{cclet}\ y = s^b\ \mathbf{in}\ {s'}^{b'})^{b_2}
\end{aligned}
$$

Figure 23. Commuting-conversion of **let**

labeled A-normal forms s, s', we define an labeled A-normal form $\mathbf{cclet}\ y = s^b\ \mathbf{in}\ {s'}^{b'}$ by induction on s as in Figure 23.

N-reduction reduces labeled A-normal forms to labeled A-normal forms or **fail**; hence, the normal forms of N-reduction are either $R[(x_1, \ldots, x_n)]$ or **fail**. It is clear that, if the evaluation of s terminates, so does N-reduction of s.

L.2 $(-)^{\sharp'_{\mathrm{N}}}$: $(-)^{\sharp'_{34}}$ for labeled A-normal forms

We define $(-)^{\sharp'_{\mathrm{N}}}$, which is a refined version of $(-)^{\sharp'_{34}}$ for labeled A-normal forms for tracking the information of the sets B.

First, for an labeled A-normal form s, we define the set $L(s)$ of *labels of* s as follows.

$$
\begin{aligned}
&L((x_1, \ldots, x_n)) && \overset{\text{def}}{=} \emptyset \\
&L(\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}) && \overset{\text{def}}{=} \{b_1\} \cup \{b_2\} \cup L(s_1) \cup L(s_2) \\
&L(\mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2}) && \overset{\text{def}}{=} \{b_1\} \cup \{b_2\} \cup L(d) \cup L(s) \\
&L(\mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, s^b)) && \overset{\text{def}}{=} \{b\} \cup L(s) \\
&L(d) && \overset{\text{def}}{=} \emptyset \qquad (d \neq \mathbf{fix}(\ldots))
\end{aligned}
$$

We call a labeled A-normal form s *label-disjoint* if all the occurrences of unions "$\cup$" above are disjoint unions when we calculate $L(s)$ by the above definition. It is obvious that we have a canonical way by which, for a A-normal form t, we obtain label-disjoint labeled A-normal form $(t)^{\mathrm{lb}}$.

Next, we give a way by which, via labels b, we can track the information of the sets B in the definition of $(-)^{\sharp'_{34}}_{T,B}$. For a label-disjoint labeled A-normal form s and $b \in L(s)$, we define B^s_b as in Figure 24; B^s_b is merely the set B used at the position b in s when we calculate $(s)^{\sharp'_{34}}_T$ by the definition in Figure 20. Note that, for labeled A-normal forms s_0, s and a context C such that $s_0 \longrightarrow^*_{\mathrm{N}} C[s]$, we have $L(s_0) \supseteq L(s)$; hence, when further s_0 is label-disjoint, for $y \in L(s)$, $B^{s_0}_b$ is well-defined.

Now, given a label-disjoint A-normal form s_0 and a labeled A-normal form s such that $s_0 \longrightarrow^*_{\mathrm{N}} C[s]$ for some C and a multiplicity-annotation T for s_0, we define a (non-labeled) term $(s)^{\sharp'_{\mathrm{N}}}_{T,s_0}$ by induction on s in Figure 25, where we also define this for **fail**, a normal form of N-reduction. For an A-normal form t_0, we write $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$ for $(-)^{\sharp'_{\mathrm{N}}}_{T,(t_0)^{\mathrm{lb}}}$.

Also, for an N-reduction context R such that $s_0 \longrightarrow^*_{\mathrm{N}} R[s]$ for some s, we define $(R)^{\sharp'_{\mathrm{N}}}_{T,s_0}$ as

$$
\begin{aligned}
([\,])^{\sharp'_{\mathrm{N}}}_{T,s_0} &\overset{\text{def}}{=} [\,] \\
\left(\mathbf{let}\ x = U^{b_1}\ \mathbf{in}\ R^{b_2}\right)^{\sharp'_{\mathrm{N}}}_{T,s_0} &\overset{\text{def}}{=} \mathbf{let}\ x = (U)^{\sharp'_{34}}_{T,B^{s_0}_{b_1}}\ \mathbf{in}\ (R)^{\sharp'_{\mathrm{N}}}_{T,s_0}\ .
\end{aligned}
$$

$$B_b^{\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}} \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } b = b_1 \text{ or } b_2 \\ B_b^{s_i} & \text{if } b \in L(s_i) \end{cases}$$

$$B_b^{\mathbf{let}\ x=d^{b_1}\ \mathbf{in}\ s^{b_2}} \stackrel{\text{def}}{=}$$

$$\begin{cases} \emptyset & \text{if } b = b_1 \\ \emptyset & \text{if } d = \mathbf{fix}(f, \lambda y.\, {s'}^{b''}), b = b'' \\ B_b^{s'} & \text{if } d = \mathbf{fix}(f, \lambda y.\, {s'}^{b''}), b \in L(s') \\ \{x = d\} & \text{if } b = b_2, d = f(\widetilde{x_i}), \mathbf{pr}_i x' \\ \emptyset & \text{if } b = b_2, d \neq f(\widetilde{x_i}), \mathbf{pr}_i x' \\ B_b^s \cup \{x = d\} & \text{if } b \in L(s), d = f(\widetilde{x_i}), \mathbf{pr}_i x' \\ B_b^s & \text{if } b \in L(s), d \neq f(\widetilde{x_i}), \mathbf{pr}_i x' \end{cases}$$

Figure 24. B_b^s: B used at b in the calculation of $(s)^{\sharp'_{34}}_T$

$$((x_1, ..., x_n, f_1, ..., f_m))^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=}$$
$$(x_1, ..., x_n, \lambda(y_1, ..., y_m).\,(f_1\, y_1, ..., f_m\, y_m))$$
$$\left(\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}\right)^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=}$$
$$\mathbf{if}\ x\ \mathbf{then}\ (s_1)^{\sharp'_{\mathrm{N}}}_{T,s_0}\ \mathbf{else}\ (s_2)^{\sharp'_{\mathrm{N}}}_{T,s_0}$$
$$\left(\mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2}\right)^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=} \mathbf{let}\ x = (d)^{\sharp'_{34}}_{T,B^{s_0}_{b_1}}\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,s_0}$$
$$(\mathbf{fail})^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=} \mathbf{fail}$$

Figure 25. $(-)^{\sharp'_{\mathrm{N}}}$: $(-)^{\sharp'_{34}}$ for labeled A-normal forms

Then, clearly

$$(R[s])^{\sharp'_{\mathrm{N}}}_{T,s_0} = (R)^{\sharp'_{\mathrm{N}}}_{T,s_0}\left[(s)^{\sharp'_{\mathrm{N}}}_{T,s_0}\right].$$

For Lemma 25, which is the main lemma for Lemma 23, we need the next lemma as well as notions of a *s-sb context*:

$$S ::= \mathbf{let}\ y = d^b\ \mathbf{in}\ l \mid \mathbf{let}\ y = \mathbf{fix}(f, \lambda x.\, l)^{b_1}\ \mathbf{in}\ s^{b_2}$$
$$\mid \mathbf{if}\ x\ \mathbf{then}\ l\ \mathbf{else}\ s^b \mid \mathbf{if}\ x\ \mathbf{then}\ s^b\ \mathbf{else}\ l$$
$$l ::= [\,] \mid S^b$$

and of a *s-db context*:

$$D ::= \mathbf{let}\ y = [\,]\ \mathbf{in}\ s^b$$
$$\mid \mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ D^{b_2} \mid \mathbf{let}\ y = \mathbf{fix}(f, \lambda x.\, D^b)^{b_1}\ \mathbf{in}\ s^{b_2}$$
$$\mid \mathbf{if}\ x\ \mathbf{then}\ D^{b_1}\ \mathbf{else}\ s^{b_2} \mid \mathbf{if}\ x\ \mathbf{then}\ s^{b_1}\ \mathbf{else}\ D^{b_2}$$

For any s, d, b, S, and D, $S[s^b]$ and $D[d^b]$ are terms of the class of the meta-variable s.

Lemma 24. *1. Given an s-db context D and $\mathbf{fix}(f, \lambda x.\, s^b)^{b'}$, let $s_0 \stackrel{\text{def}}{=} D[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}]$ and suppose that s_0 is label-disjoint. Then $B^{s_0}_b = B^{s_0}_{b'}$.*

2. Given an s-sb context S and s^b, let $s_0 \stackrel{\text{def}}{=} S[s^b]$ and suppose that s_0 is label-disjoint.
When $s = \mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}$, $B^{s_0}_{b_1} = B^{s_0}_{b_2} = B^{s_0}_b$.
When $s = \mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ {s'}^{b_2}$, $B^{s_0}_{b_1} = B^{s_0}_b$; further,

- *when $d \neq f\,(\widetilde{x})$ nor $\mathbf{pr}_i x$, $B^{s_0}_{b_2} = B^{s_0}_b$,*
- *when $d = f\,(\widetilde{x})$ or $\mathbf{pr}_i x$, $B^{s_0}_{b_2} = B^{s_0}_b \cup \{x = d\}$.*

3. Given an s-sb context S and s^b, let $s_0 \stackrel{\text{def}}{=} S[s^b]$ and suppose that s_0 is label-disjoint. Let T be a multiplicity annotation for s_0. Then

$$(s)^{\sharp'_{\mathrm{N}}}_{T,s_0} = (s)^{\sharp'_{34}}_{T,B^{s_0}_b}.$$

4. For a label-disjoint A-normal form s_0 such that

$$s_0 \longrightarrow^*_{\mathrm{N}} D[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}],$$

there is some D' such that

$$s_0 = D'[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}].$$

5. For a label-disjoint A-normal form s_0 such that

$$s_0 \longrightarrow^*_{\mathrm{N}} R[\mathbf{cclet}\ y = s^b\ \mathbf{in}\ {s'}^{b'}],$$

and a multiplicity annotation T for s_0,

$$\left(\mathbf{cclet}\ y = s^b\ \mathbf{in}\ {s'}^{b'}\right)^{\sharp'_{\mathrm{N}}}_{T,s_0} = \mathbf{cclet}\ y = (s)^{\sharp'_{\mathrm{N}}}_{T,s_0}\ \mathbf{in}\ (s')^{\sharp'_{\mathrm{N}}}_{T,s_0}.$$

Proof. 1. By induction on D.

2. By induction on S.

3. By induction on s and by 2 of this lemma.

4. By induction on the length of the N-reduction, it is enough to prove that, if

$$s_0 \longrightarrow^*_{\mathrm{N}} s_1 \longrightarrow_{\mathrm{N}} D[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}]$$

there is D' such that

$$s_1 = D'[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}].$$

We consider only the case that the redex of s_1 is application; the others are clear. Hence,

$$s_1 = R[\mathbf{let}\ y = (f\,(x_1, \ldots, x_n, g_1, \ldots, g_m))^{b_1}\ \mathbf{in}\ {s_2}^{b_2}]$$
$$D[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}] =$$
$$R[\mathbf{cclet}\ y = \left(s'[x'_i \mapsto x_i]_i[g'_j \mapsto g_j]_j[f' \mapsto f]\right)^{b''}\ \mathbf{in}\ {s_2}^{b_2}]$$
$$f \leadsto_R \mathbf{fix}(f', \lambda(x'_1, \ldots, x'_n, g'_1, \ldots, g'_m).\, {s'}^{b''})^{b'''}.$$

By α-renaming, there is s'' such that

$$D[\mathbf{fix}(f, \lambda x.\, s^b)^{b'}] = R[\mathbf{cclet}\ y = {s''}^{b''}\ \mathbf{in}\ {s_2}^{b_2}]$$
$$f \leadsto_R \mathbf{fix}(f, \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m).\, {s''}^{b''})^{b'''}.$$

If $\mathbf{fix}(f, \lambda x.\, s^b)^{b'}$ occurs in R or s_2, so does in s_1. Otherwise, $\mathbf{fix}(f, \lambda x.\, s^b)^{b'}$ occurs in s''. Since

$$f \leadsto_R \mathbf{fix}(f, \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m).\, {s''}^{b''})^{b'''},$$

s'' occurs in R; hence, $\mathbf{fix}(f, \lambda x.\, s^b)^{b'}$ occurs in s_1.

5. By induction on s.

□

We here give a remark on the definition of labeled A-normal forms. The labels for d are needed for the definition of $(-)^{\sharp'_{\mathrm{N}}}$; $(-)^{\sharp'_{\mathrm{N}}}$ and the labels for s in $\mathbf{fix}(f, \lambda x.\, s^b)$ with Lemma 24-1,3 are needed for the proof of Lemma 25. On the other hand, the labels for the other occurrences of s (i.e., s in **if** and **let** expressions) are needed just for the induction on s in the proof of Lemma 24-3.

L.3 $(-)^{\sharp'_{\mathrm{N}}}$ preserves N-reduction to observational equivalence

It is clear that Lemma 23 can be proved immediately by the following lemma.

Lemma 25. *1. For an A-normal form t,*

$$(t)^{\sharp'_{34}}_{T} \;=\; \left((t)^{\mathrm{lb}}\right)^{\sharp'_{\mathrm{N}}}_{T,t} .$$

2. For a closed ground A-normal form t,

$$(t)^{\mathrm{lb}} \longrightarrow^{*}_{\mathrm{N}} R[(x_1,\ldots,x_n)]$$

implies

$$(R[(x_1,\ldots,x_n)])^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (R[(x_1,\ldots,x_n)])^{\sharp'_{\mathrm{N}}}_{T,t} ,$$

and $(t)^{\mathrm{lb}} \longrightarrow^{*}_{\mathrm{N}}$ **fail** *implies*

$$(\mathbf{fail})^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (\mathbf{fail})^{\sharp'_{\mathrm{N}}}_{T,t} .$$

3. For a closed ground A-normal form t, $(t)^{\mathrm{lb}} \longrightarrow^{*}_{\mathrm{N}} s$ *implies*

$$(t)^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (s)^{\sharp_{34}}_{T} \qquad \textit{and} \qquad \left((t)^{\mathrm{lb}}\right)^{\sharp'_{\mathrm{N}}}_{T,t} \;=_{\mathrm{o}}\; (s)^{\sharp'_{\mathrm{N}}}_{T,t} .$$

In the proof of this lemma below, in addition to λ_{c}-calculus [11] (the standard call-by-value equational theory), we often use the following reasoning principle, which we call *referential transparency*:

$$\mathbf{let}\; x = t \;\mathbf{in}\; C[x] \quad = \quad \mathbf{let}\; x = t \;\mathbf{in}\; C[t] \qquad \text{(RT)}$$

where the occurrence x in $C[x]$ must be free (and bound by the let-declaration). Here we used contexts $C[x]$ and $C[t]$ rather than t' and $t'[x \mapsto t]$; this means that any *one* occurrence of x and t are interchangeable (and hence, so are *all* the occurrences, by repeating it). It is clear that (RT) is sound with respect to the observational equivalence of our language.

The axiom (RT) allows us to regard let-binding such as

- $(x = t)$ in B of $(-)^{\sharp'_{34}}_{T,B}$
- $x \rightsquigarrow_R t^b$

as "an equation already proved". Below, for a context C, we write $t \equiv t'$ *in* C to mean that $C[t] =_{\mathrm{o}} C[t']$. For example, when $x \rightsquigarrow_R t^b$, it is true that $x \equiv t$ in R by (RT), though x and t themselves are not necessarily observationally equivalent. We sometimes omit contexts C if they are clear.

Below, when we write $t = \{\ldots\}\, t'$, "$\{\ldots\}$" is an explanation of why the equation holds.

By (RT), we can prove:

Lemma 26. *1. For any term t,* $\mathbf{pr}_1(t,\ldots,t) =_{\mathrm{o}} t$.
2. For any term of a tuple type, $t =_{\mathrm{o}} (\mathbf{pr}_1 t, \mathbf{pr}_2 t)$.

Proof. 1.

$$\begin{aligned}
&\mathbf{pr}_1(t,t,t,\ldots,t)\\
=_{\mathrm{o}}\;&\{\lambda_{\mathrm{c}}\}\\
&\mathbf{let}\; x = t \;\mathbf{in}\; \mathbf{pr}_1(x,t,t,\ldots,t)\\
=_{\mathrm{o}}\;&\{\text{(RT)}\}\\
&\mathbf{let}\; x = t \;\mathbf{in}\; \mathbf{pr}_1(x,x,t,\ldots,t)\\
=_{\mathrm{o}}\;&\ldots\\
=_{\mathrm{o}}\;&\mathbf{let}\; x = t \;\mathbf{in}\; \mathbf{pr}_1(x,x,x,\ldots,x)\\
=_{\mathrm{o}}\;&\{\lambda_{\mathrm{c}}\}\\
&\mathbf{let}\; x = t \;\mathbf{in}\; x\\
=_{\mathrm{o}}\;&\{\lambda_{\mathrm{c}}\}\\
&t .
\end{aligned}$$

*2.*Similarly,

$$\begin{aligned}
&(\mathbf{pr}_1 t, \mathbf{pr}_2 t)\\
=_{\mathrm{o}}\;&\mathbf{let}\; x = t \;\mathbf{in}\; (\mathbf{pr}_1 x, \mathbf{pr}_2 t)\\
=_{\mathrm{o}}\;&\mathbf{let}\; x = t \;\mathbf{in}\; (\mathbf{pr}_1 x, \mathbf{pr}_2 x)\\
=_{\mathrm{o}}\;&\mathbf{let}\; x = t \;\mathbf{in}\; x\\
=_{\mathrm{o}}\;&t .
\end{aligned}$$

□

Now, let us return to proving Lemma 25.

Proof of Lemma 25.
1. Straightforward.
2. Since $R[(x_1,\ldots,x_n)]$ is ground, x_i are integer variables. Hence $x_i \rightsquigarrow_R {m_i}^{b_i}$ for some m_i, and so the both sides are observationally equivalent to $(m_1,\ldots,m_n)$. The case of **fail** is clear.
3. A proof on $(-)^{\sharp_{34}}$ is obvious if we can prove the case of $(-)^{\sharp'_{\mathrm{N}}}$. We show that, for a closed ground A-normal form t_0,

$$(t_0)^{\mathrm{lb}} \longrightarrow^{*}_{\mathrm{N}} s_1 \longrightarrow_{\mathrm{N}} s_2 \quad \text{implies} \quad (s_1)^{\sharp'_{\mathrm{N}}}_{T,t_0} =_{\mathrm{o}} (s_2)^{\sharp'_{\mathrm{N}}}_{T,t_0}$$

by induction on the length of the N-reduction; further, simultaneously we show that, if $s_1 = R[\mathbf{let}\; y = d^{b_1} \;\mathbf{in}\; s^{b_2}]$,

$$(d)^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv (d)^{\sharp_{34}}_{T}$$

in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\; y = [\,]\;\mathbf{in}\; (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]$. We show only the case of application:

$$\begin{aligned}
s_1 &= R[\mathbf{let}\; y = (f\,(\widetilde{x_i},\widetilde{g_j}))^{b_1} \;\mathbf{in}\; s^{b_2}]\\
s_2 &= R[\mathbf{cclet}\; y = \left(s'[x'_i \mapsto x_i]_i[g'_j \mapsto g_j]_j[f' \mapsto f]\right)^{b''} \;\mathbf{in}\; s^{b_2}]\\
f &\rightsquigarrow_R \left(\mathbf{fix}(f', \lambda(\widetilde{x'_i},\widetilde{g'_j}).\, s'^{b''})\right)^{b'}
\end{aligned}$$

since the other cases are clear. By α-renaming, we assume $x'_i = x_i$, $g'_j = g_j$, and $f' = f$.

We postpone to show

$$(f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv (f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp_{34}}_{T}$$

—i.e., to show that **assume** in $\mathtt{InstVar}(-)$ are satisfied—until Appendix L.4 and show the remaining part first.

Since $f \rightsquigarrow_R \left(\mathbf{fix}(f, \lambda(\widetilde{x_i},\widetilde{g_j}).\, s'^{b''})\right)^{b'}$, applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$\begin{aligned}
f \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} &\left(\mathbf{fix}(f, \lambda(\widetilde{x_i},\widetilde{g_j}).\, s')\right)^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\\
&= \mathbf{fix}(f, \lambda(z_1,\ldots,z_{T(f)}).\,(s'_1,\ldots,s'_{T(f)}))
\end{aligned} \tag{36}$$

where

$$s'_k \stackrel{\mathrm{def}}{=} (s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\,[x_i \mapsto \mathbf{pr}_i z_k]_{i\le n}[g_j \mapsto p_j^{z_k}]_{j\le m}$$

$$p_j^{z_k} \stackrel{\mathrm{def}}{=} \lambda y.\, \mathbf{pr}_j((\mathbf{pr}_{n+1} z_k)(\overrightarrow{\bot}^{j-1}, y, \overrightarrow{\bot}^{m-j})) .$$

Now,

$$\begin{aligned}
(s_1)^{\sharp'_{\mathrm{N}}}_{T,t_0} &= (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\; y = (f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \;\mathbf{in}\; (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]\\
(s_2)^{\sharp'_{\mathrm{N}}}_{T,t_0} &= (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\left[\left(\mathbf{cclet}\; y = s'^{b''} \;\mathbf{in}\; s^{b_2}\right)^{\sharp'_{\mathrm{N}}}_{T,t_0}\right]\\
&= \{\text{by Lemma 24-5}\}\\
&\quad (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\left[\mathbf{cclet}\; y = (s')^{\sharp'_{\mathrm{N}}}_{T,t_0} \;\mathbf{in}\; (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\right]
\end{aligned}$$

$$=_\circ (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\Big[\mathbf{let}\ y = \big(s'\big)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\Big]$$

and in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\ y = [\,]\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]$,

$$\begin{aligned}
& (f\ (\widetilde{x}_i, \widetilde{g}_j))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \\
\equiv\ & \{\text{Appendix L.4}\} \\
& (f\ (\widetilde{x}_i, \widetilde{g}_j))^{\sharp_{34}}_{T} \\
=\ & \mathbf{pr}_1(f(\overrightarrow{(\widetilde{x}_i, \lambda\widetilde{y}_j.\,(g_j\ y_j)_j)}^{T(f)})) \\
\equiv\ & \{\text{by (36)}\} \\
& \mathbf{pr}_1\Big(\big(s'_k[z_k \mapsto (\widetilde{x}_i, \lambda\widetilde{y}_j.\,(g_j\ y_j)_j)]\big)_{k\le T(f)}\Big) \\
\equiv\ & \mathbf{pr}_1\Big(\Big(\big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ \big[g_j \mapsto p_j^{z_k}[z_k \mapsto (\widetilde{x}_i, \lambda\widetilde{y}_j.\,(g_j\ y_j)_j)]\big]_j\Big)_{k\le T(f)}\Big) \\
\equiv\ & \mathbf{pr}_1\Big(\Big(\big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ \big[g_j \mapsto \\
& \qquad \lambda y.\,\mathbf{pr}_j((\lambda\widetilde{y}_j.\,(g_j\ y_j)_j)(\overrightarrow{\bot}, y, \overrightarrow{\bot})) \\
& \qquad \big]_j\Big)_{k\le T(f)}\Big) \\
\equiv\ & \mathbf{pr}_1\Big(\Big(\big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ [g_j \mapsto \lambda y.\,g_j\ y]_j\Big)_{k\le T(f)}\Big) \\
\equiv\ & \mathbf{pr}_1\Big(\Big(\big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\Big)_{k\le T(f)}\Big) \\
\equiv\ & \{\text{by Lemma 26-1}\} \\
& \big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b'}} \\
\equiv\ & \{\text{by Lemma 24-1, 4}\} \\
& \big(s'\big)^{\sharp'_{34}}_{T,B^{t_0}_{b''}} \\
\equiv\ & \{\text{by Lemma 24-3, 4}\} \\
& \big(s'\big)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ .
\end{aligned}$$

□

L.4 assume in InstVar(−) are satisfied

The remaining is to show

$$(f\ (\widetilde{x}_i, \widetilde{g}_j))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv (f\ (\widetilde{x}_i, \widetilde{g}_j))^{\sharp_{34}}_{T}$$

in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\ y = [\,]\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]$ in the proof of Lemma 25.

First, let us recall the definition of $(f(\widetilde{x}_i, \widetilde{g}_j))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}}$:

$$(f(x_1, ..., x_n, g_1, ..., g_m))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \stackrel{\text{def}}{=} \mathtt{InstVar}(f, T, B^{t_0}_{b_1}, t_{m+1})$$

where

$$t_1 \stackrel{\text{def}}{=} \mathbf{pr}_1(f(\overrightarrow{(\widetilde{x}_i, \lambda\widetilde{y}_j.\,(g_j\ y_j)_j)}^{T(f)}))$$
$$t_{j+1} \stackrel{\text{def}}{=} \mathtt{InstVar}(g_j, T, B^{t_0}_{b_1}, t_j) \quad (\text{for } j = 1, ..., m)\ .$$

From now, we will prove that, for $j = 1, \ldots, m{+}1$,

$$\mathtt{InstVar}(g, T, B, t) \quad \equiv \quad t$$

in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\ y = [\,]\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]$ where

$$g_{m+1} \stackrel{\text{def}}{=} f, \quad g \stackrel{\text{def}}{=} g_j, \quad B \stackrel{\text{def}}{=} B^{t_0}_{b_1}, \quad t \stackrel{\text{def}}{=} t_j\ .$$

Then, by applying this repeatedly for $j = m{+}1, \ldots, 1$ (in the reverse order), we get our goal, i.e.,

$$(f(x_1, ..., x_n, g_1, ..., g_m))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv t_1$$

in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\ y = [\,]\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}]$.

Thus, our goal is to prove

$$\mathbf{let}\ g = \left(\begin{array}{l} \lambda\widetilde{y}.\,\mathbf{let}\ x = g\widetilde{y}\ \mathbf{in} \\ \quad \mathbf{let}\ w^{\langle(\alpha^1_j)_j\rangle} = B^\sharp_g((\alpha^1_j)_j)\ \mathbf{in} \\ \quad \cdots \\ \quad \mathbf{let}\ w^{\langle(\alpha^k_j)_j\rangle} = B^\sharp_g((\alpha^k_j)_j)\ \mathbf{in} \\ \quad \mathbf{assume}\,(p)\,;\mathbf{assume}\,(p')\,;x \end{array}\right) \mathbf{in}\ t \quad \equiv \quad t\,,$$

i.e., to show that p and p' above are satisfied and hence $\mathbf{assume}\,(p)$ and $\mathbf{assume}\,(p')$ can be removed. They are defined using $B^\sharp_g((\alpha_j)_j)$, which is defined in Step 4 in Appendix K.1 as

$$\mathbf{pr}_j B^\sharp_g((\alpha_j)_j) = \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\mathtt{arg}^*_1(\alpha_1), \ldots, \mathtt{arg}^*_m(\alpha_m))\big)$$

and by (30),

$$\mathbf{pr}^\sharp_\to z\ :\ \prod_{j=1}^m \big((\tau_j)^{\sharp'_{34}}\big)^{\mathtt{m}_j} \to \prod_{j=1}^m \big((\tau'_j)^{\sharp'_{34}}\big)^{(\mathtt{m}_j)}_{[\cdot\mapsto\cdot]}\ .$$

From now, we calculate a concrete form of $\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j)$.

Below, we use the next lemma; it is this lemma that we needed to consider N-reduction for.

Lemma 27. *Let v be a variable declared in R, i.e., $v \rightsquigarrow_R U^{b_0}$ for some U and b_0.*

1. *For $j \in m$, $i \in \mathtt{m}_j$, $t = ((t_{j,i})_{i\in\mathtt{m}_j})_{j\in m}$, and $t' = ((t'_{j,i})_{i\in\mathtt{m}_j})_{j\in m}$,*
$$t_{j,i} \quad = \quad t'_{j,i}$$
implies
$$\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t\big) \quad \equiv \quad \mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t'\big)\ .$$
2. *Given $t = ((t_{j,i})_{i\in\mathtt{m}_j})_{j\in m}$, $j \in m$, and $i, i' \in \mathtt{m}_j$,*
$$t_{j,i} = t_{j,i'}$$
implies
$$\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)(t)\big) \quad \equiv \quad \mathbf{pr}_{i'}\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t\big)\ .$$

Proof. Roughly, *1* is because v is the product $\prod_{i,j} f_{i,j}$ of some functions $f_{i,j}$, and *2* is because, for each j, $(f_{i,j})_i$ is a replication of some one function f_j.

By unfolding the definition of $(-)^{\sharp'_{\mathrm{N}}}$ and $(-)^{\sharp'_{34}}$, proofs are straightforward because now v (before applying $(-)^{\sharp'_{\mathrm{N}}}$) is bound to a labeled value U, due to N-reduction. We give a proof only for *1*. Since v has a product type, we suppose $U = (x'_1, \ldots, x'_{n'}, f'_1, \ldots, f'_{m'})$.

Since $v \rightsquigarrow_R U^{b_0}$, applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$v \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} (\widetilde{x'_i}, \lambda(y'_1, ..., y'_{m'}).\,(f'_1\,y'_1, ..., f'_{m'}\,y'_{m'}))\ .$$

Then,

$$\begin{aligned}
& \mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t\big) \\
\equiv\ & \mathbf{pr}_i\mathbf{pr}_j\Big(\Big(\mathbf{pr}^\sharp_\to\Big(\widetilde{x'_i}, \lambda(\widetilde{y'_j}).\,(f'_j\ y'_j)_j\Big)\Big)t\Big) \\
\equiv\ & \mathbf{pr}_i\mathbf{pr}_j\Big(\lambda(\widetilde{y'_j}).\,(f'_j\ y'_j)_j\Big)t\Big) \\
\equiv\ & \mathbf{pr}_i\big(f'_j\ (t_{j,i})_{i\in\mathtt{m}_j}\big)
\end{aligned}$$

Now, since f'_j have function types, there are $\left(\mathbf{fix}(f''_j, \lambda x''_j.\, {s''_j}^{b'''_j})\right)^{b''_j}$ such that

$$f'_j \rightsquigarrow_{R|_v} \left(\mathbf{fix}(f''_j, \lambda x''_j.\, {s''_j}^{b'''_j})\right)^{b''_j} .$$

Then, applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$\begin{aligned} f'_j &\rightsquigarrow_{(R|_v)^{\sharp'_{\mathrm{N}}}_{T,t_0}} \left(\mathbf{fix}(f''_j, \lambda x''_j.\, s''_j)\right)^{\sharp'_{34}}_{T,B^{t_0}_{b''_j}} \\ &= \mathbf{fix}(f''_j, \lambda(z^j_1, \dots, z^j_{\mathtt{m}_j}).\, (t''_{j,1}, \dots, t''_{j,\mathtt{m}_j})) \end{aligned}$$

for some $t''_{j,i}$ that does not contain variables $z^j_{i'}$ for $i' \neq i$ (by the definition in Figure 20). Hence,

$$\begin{aligned} &\mathbf{pr}_i\big(f'_j\,(t_{j,i})_{i\in\mathtt{m}_j}\big) \\ \equiv\ &\mathbf{pr}_i\big((t''_{j,1}, \dots, t''_{j,T(f'_j)})[z^j_i \mapsto t_{j,i}]_{i\in\mathtt{m}_j}[f''_j \mapsto f'_j]\big) \\ \equiv\ &t''_{j,i}[z^j_i \mapsto t_{j,i}][f''_j \mapsto f'_j] . \end{aligned}$$

In the same way,

$$\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t'\big) \;\equiv\; t''_{j,i}[z^j_i \mapsto t'_{j,i}][f''_j \mapsto f'_j] .$$

By assumption, $t_{j,i} = t'_{j,i}$, therefore

$$\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t\big) \quad\equiv\quad \mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to v)t'\big) .$$

□

Now, for a given $(\alpha_j)_j \in \prod_{j=1}^{m} \mathtt{App}^*_j$, $j \in m$, and $i \in \mathtt{m}_j$, we calculate more concrete form of $\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j)$ separately in the following cases of $\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j)$:

$$\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j)(=\mathtt{arg}_j(\mathbf{pr}_i\alpha_j)) = \begin{cases} u \\ y_l \\ \bot \end{cases}$$

where the cases correspond to the cases in the definition of $\mathtt{arg}_j$.

In the case that $\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = u$ for some u, there are v and w such that

$$\mathbf{pr}_i\alpha_j = (u, v, w) \quad \text{and} \quad (v = \mathbf{pr}^{\to}_j z), (w = vu) \in B$$

and then, after $(-)^{\sharp'_{34}}$, the let-binding $(v = \mathbf{pr}^{\to}_j z)$ becomes the following let-binding

$$v = \lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big) \tag{37}$$

where a is in j-th position, and $(w = vu)$ becomes

$$w = (vu)^{\sharp'_{34}}_{T,B'} \tag{38}$$

for some B'. Hence,

$$\begin{aligned} &\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j) \\ =\ &\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\mathtt{arg}^*_1(\alpha_1), \dots, \mathtt{arg}^*_m(\alpha_m))\big) \\ \equiv\ &\{\text{by Lemma 27-1}\} \\ &\mathbf{pr}_i\mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, \mathtt{arg}^*_j(\alpha_j), \bot, \dots, \bot)\big) \\ \equiv\ &\big\{\beta\text{-equality since } \mathtt{arg}^*_j(\alpha_j) \text{ is a value}\big\} \\ &\mathbf{pr}_i\Big(\big(\lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big)\big)\mathtt{arg}^*_j(\alpha_j)\Big) \\ \equiv\ &\{\text{(RT) on (37)}\} \\ &\mathbf{pr}_i\big(v\ \mathtt{arg}^*_j(\alpha_j)\big) \\ \equiv\ &\big\{\text{since } \mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = u\text{, and by Lemma 27-1,2 where } m = 1\big\} \\ &\mathbf{pr}_1\big(v\ (u, \dots, u)\big) \\ =\ &(vu)^{\sharp_{34}}_{T} \\ \equiv\ &\{\text{by induction hypothesis, since } w = vu \text{ is in } B\} \\ &(vu)^{\sharp'_{34}}_{T,B'} \\ \equiv\ &\{\text{(RT) on (38)}\} \\ &w. \end{aligned}$$

In the case that $\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = y_l$ for some $l \in \mathtt{m}_k$,

$$j = k \qquad \mathbf{pr}_i\alpha_j = y_l \qquad (g = \mathbf{pr}^{\to}_j z) \in B,$$

and after $(-)^{\sharp'_{34}}$, $(g = \mathbf{pr}^{\to}_j z)$ becomes

$$g = \lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big). \tag{39}$$

Hence,

$$\begin{aligned} &\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j) \\ \equiv\ &\{\text{similarly to the previous}\} \\ &\mathbf{pr}_i\Big(\big(\lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big)\big)\mathtt{arg}^*_j(\alpha_j)\Big) \\ \equiv\ &\{\text{(RT) on (39)}\} \\ &\mathbf{pr}_i\big(g\ \mathtt{arg}^*_j(\alpha_j)\big) \\ \equiv\ &\big\{\text{since } \mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = y_l \text{ and by Lemma 27-1,2 where } m = 1\big\} \\ &\mathbf{pr}_l\big(g\widetilde{y}\big). \end{aligned}$$

In the case that $\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = \bot$,

$$\mathbf{pr}_i\alpha_j = (\bot, v) \qquad (v = \mathbf{pr}^{\to}_j z) \in B,$$

and after $(-)^{\sharp'_{34}}$, $(v = \mathbf{pr}^{\to}_j z)$ becomes

$$v = \lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big). \tag{40}$$

Hence,

$$\begin{aligned} &\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j) \\ \equiv\ &\{\text{similarly to the previous}\} \\ &\mathbf{pr}_i\Big(\big(\lambda a.\, \mathbf{pr}_j\big((\mathbf{pr}^\sharp_\to z)(\bot, \dots, \bot, a, \bot, \dots, \bot)\big)\big)\mathtt{arg}^*_j(\alpha_j)\Big) \\ \equiv\ &\{\text{(RT) on (40)}\} \\ &\mathbf{pr}_i\big(v\ \mathtt{arg}^*_j(\alpha_j)\big) \\ \equiv\ &\big\{\text{since } \mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = \bot\big\} \\ &\bot. \end{aligned}$$

Now we show that, in the definition of $\mathtt{InstVar}(g, T, B, t)$, $\mathbf{assume}\,(p)$ and $\mathbf{assume}\,(p')$ can be removed; then, after that, with λ_{c}-theory, $\mathtt{InstVar}(g, T, B, t)$ is equivalent to t, which concludes the proof of the lemma.

First, note that the equality $=$ and the implication $\Rightarrow$ used in p and p' are not genuine logical operators but boolean primitives, so if two terms t and t' both happen divergence (or fail), $t = t'$ is not true but divergence (or fail), and $\mathbf{assume}\,(\dots t = t' \dots)$ cannot necessarily be removed (and similarly for $\Rightarrow$). Thus, it is important to know if such t and t' are values or not. Now, since $\mathtt{arg}^*_j(\alpha_j)$ and $\mathtt{arg}^*_j(\alpha'_j)$ are values, this concern is in fact cleared.

Next, to calculate the $\mathbf{assume}$-expressions, we have to substitute $B^\sharp_g((\alpha_j)_j)$ for $w^{\langle(\alpha_j)_j\rangle}$ in p and p', and to do so we need to show that $B^\sharp_g((\alpha_j)_j)$ is (observationally equivalent to) a value. For this end, by Lemma 26-2, it is enough to show that, for any i and j, $\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j)$ is a value.

For each i, in the case that $\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = u$ or $\bot$, as seen above, $\mathbf{pr}_i\mathbf{pr}_j B^\sharp_g((\alpha_j)_j)$ is a value. In the case that

$$\mathbf{pr}_i\mathtt{arg}^*_j(\alpha_j) = y_l,$$

as above,

$$\mathbf{pr}_i \mathbf{pr}_j B_g^\sharp((\alpha_j)_j) \equiv \mathbf{pr}_l(g\widetilde{y}).$$

Here, $g\widetilde{y}$ itself is not a value, but can be replaced with a value x as below by (RT); i.e.,

$$\begin{aligned} &\begin{pmatrix} \textbf{let}\ x = g\widetilde{y}\ \textbf{in} \\ \textbf{let}\ w^{\langle(\alpha_j^1)_j\rangle} = B_g^\sharp((\alpha_j^1)_j)\ \textbf{in} \\ \cdots \\ \textbf{let}\ w^{\langle(\alpha_j^k)_j\rangle} = B_g^\sharp((\alpha_j^k)_j)\ \textbf{in} \\ \textbf{assume}\,(p[\mathbf{pr}_l(g\widetilde{y})])\,;\textbf{assume}\,\big(p'\big)\,;x \end{pmatrix} \\ =_{\mathrm{o}}\ &\begin{pmatrix} \textbf{let}\ x = g\widetilde{y}\ \textbf{in} \\ \textbf{let}\ w^{\langle(\alpha_j^1)_j\rangle} = B_g^\sharp((\alpha_j^1)_j)\ \textbf{in} \\ \cdots \\ \textbf{let}\ w^{\langle(\alpha_j^k)_j\rangle} = B_g^\sharp((\alpha_j^k)_j)\ \textbf{in} \\ \textbf{assume}\,(p[\mathbf{pr}_l(x)])\,;\textbf{assume}\,\big(p'\big)\,;x \end{pmatrix}. \end{aligned}$$

Thus, we can substitute $B_g^\sharp((\alpha_j)_j)$ for $w^{\langle(\alpha_j)_j\rangle}$.

On $\textbf{assume}\,(p)$, it is obvious that the term

$$\mathtt{arg}_j^*(\alpha_j) = \mathtt{arg}_j^*(\alpha'_j)\ \texttt{=>}\ \mathbf{pr}_i \mathbf{pr}_j B_g^\sharp((\alpha_j)_j) = \mathbf{pr}_i \mathbf{pr}_j B_g^\sharp((\alpha'_j)_j)$$

is observationally equivalent to $\mathbf{true}$, from the calculation of $\mathbf{pr}_i \mathbf{pr}_j B_g^\sharp((\alpha_j)_j)$ above. On $\textbf{assume}\,(p')$, similarly,

$$w = \mathbf{pr}_i \mathbf{pr}_j B_g^\sharp((\alpha_j)_j)$$

is observationally equivalent to $\mathbf{true}$. □