Verifying Relational Properties of Functional Programs by First-Order Refinement☆

Kazuyuki Asada[a], Ryosuke Sato[a], Naoki Kobayashi[a]

[a] *University of Tokyo*

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 implementation and experiments.

1. Introduction

There has been much progress in automated verification techniques for higher-order functional programs [12, 17, 16, 9, 11, 19, 13].[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 [12, 17, 16, 9], one can verify that `sum` has the first-order refinement type: $(n : \mathbf{int}) \to \{m : \mathbf{int} \mid m \geq n\}$, which means that `sum n` returns a value no less than `n`. Here, $\{m : \mathbf{int} \mid P(m)\}$ 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 $\mathtt{sum2}(n)$ equals $\mathtt{sum}(n)$ for every integer n. With general refinement types [6], that would amount to checking that `sumacc` and `sum2` have the following types:[2]

$$\begin{aligned}&\mathtt{sumacc} : (n : \mathbf{int}) \to (m : \mathbf{int}) \to \{r : \mathbf{int} \mid r = m + \mathtt{sum}(n)\}\\&\mathtt{sum2} : (n : \mathbf{int}) \to \{r : \mathbf{int} \mid r = \mathtt{sum}(n)\}\end{aligned}$$

☆This article is a revised and extended version of the paper that appeared in Proceedings of PEPM 2015 under the title "Verifying Relational Properties of Functional Programs by First-Order Refinement". We have added detailed definitions, proofs, and explanations.

Email addresses: asada@kb.is.s.u-tokyo.ac.jp (Kazuyuki Asada), ryosuke@kb.is.s.u-tokyo.ac.jp (Ryosuke Sato), koba@kb.is.s.u-tokyo.ac.jp (Naoki Kobayashi)

[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 [20, 2] where a user must declare refinement types for *all* recursive functions including auxiliary functions. Some of the automated verification techniques above require a hint [19], 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.

The type of `sum2` means that `sum2` takes an integer as an argument n 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 following program.

```
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)
```

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) [13], 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:

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

(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

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

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 [3] 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)
```

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

[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`.

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 $\tau_{\texttt{append3}}$:

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

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 $\tau_{\texttt{append3}}$. (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 [9, 13] 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.

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

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

Figure 1: Operational semantics of the source language

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:

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

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. We express Booleans by integers, and write $\mathbf{true}$ for 1, and $\mathbf{false}$ for 0. The term $\mathtt{op}(\widetilde{t})$ (where $\widetilde{t}$ denotes a sequence of expressions) applies the primitive operation $\mathtt{op}$ on integers to $\widetilde{t}$. We assume that we have the equality operator $=$, the conjunction $\&$, and the implication $\Rightarrow$ as primitive operations. We sometimes write $=$ also as $==$ to distinguish it from the mathematical equality. 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 (i.e., a term containing no free varibales) 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 $\mathbf{fail}$ can have every type).

The small-step semantics is shown in Figure 1. 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., $t \uparrow$). The semantics of the primitive operations $=$, $\&$, and $\Rightarrow$ are defined as follows.

$$[\![=]\!](n_1, n_2) = \begin{cases} \mathbf{true} & (\text{if } n_1 = n_2) \\ \mathbf{false} & (\text{otherwise}) \end{cases}$$

$$[\![\&]\!](b_1, b_2) = \begin{cases} b_2 & (\text{if } b_1 = \mathbf{true}) \\ \mathbf{false} & (\text{otherwise}) \end{cases}$$

$$[\![\Rightarrow]\!](b_1, b_2) = \begin{cases} b_2 & (\text{if } b_1 = \mathbf{true}) \\ \mathbf{true} & (\text{otherwise}) \end{cases}$$

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

$$\begin{array}{lll} \tau \text{ (types)} & ::= & \rho \mid \{\nu : \prod_{i=1}^{n} (x_i \colon \rho_i) \mid P\} \\ \rho \text{ (non-tuple types)} & ::= & \{\nu : \mathbf{int} \mid P\} \mid \{\nu : (x : \tau_1) \to \tau_2 \mid P\} \\ P \text{ (predicates)} & ::= & t \mid P \wedge P \mid \forall x.P \end{array}$$

where we have used a notational convention $\prod_{i=1}^{n} (x_i \colon \rho_i)$ to denote $(x_1 \colon \rho_1) \times \cdots \times (x_{n-1} \colon \rho_{n-1}) \times \rho_n$ (thus, the variable x_n actually does not occur). The type $(x \colon \rho_1) \times \rho_2$ is a dependent sum type, where x may occur in ρ_2, and $(x \colon \tau_1) \to \tau_2$ is a dependent product type, where x may occur in τ_2. We use a metavariable σ to denote $\mathbf{int}$, $(x \colon \tau_1) \to \tau_2$, or $\prod_{i=1}^{n} (x_i \colon \rho_i)$. 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 $\{f : \mathbf{int} \to \mathbf{int} \mid \forall x, y.\ x \leq y \text{ => } f(x) \leq f(y)\}$ 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 $(x \colon \rho_1) \times \rho_2$ if x does not occur in τ_2 and ρ_2 respectively. We write τ^m for $\tau \times \cdots \times \tau$ (the m-th power); $\{(\nu_i)_{i \leq n} : \prod_{i=1}^{n} (x_i \colon \rho_i) \mid P\}$ for $\{\nu : \prod_{i=1}^{n} (x_i \colon \rho_i) \mid P[\nu_1 \mapsto \mathbf{pr}_1 \nu, \ldots, \nu_n \mapsto \mathbf{pr}_n \nu]\}$; and $\forall \widetilde{x}.P$ for $\forall x_1, \ldots, x_n.P$.

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

$$\begin{aligned} &\mathtt{ST}(\{\nu : \sigma \mid P\}) = \mathtt{ST}(\sigma) \\ &\mathtt{ST}(\mathbf{int}) = \mathbf{int} \\ &\mathtt{ST}((x : \tau_1) \to \tau_2) = \mathtt{ST}(\tau_1) \to \mathtt{ST}(\tau_2) \\ &\mathtt{ST}(\textstyle\prod_{i=1}^{n} (x_i \colon \rho_i)) = \prod_{i=1}^{n} \mathtt{ST}(\rho_i) \end{aligned}$$

We define the *order of* τ by:

$$\begin{aligned} &\mathit{order}(\{\nu : \sigma \mid P\}) = \mathit{order}(\sigma) \\ &\mathit{order}(\mathbf{int}) = 0 \\ &\mathit{order}((x : \tau_1) \to \tau_2) = \max(\mathit{order}(\tau_1) + 1, \mathit{order}(\tau_2)) \\ &\mathit{order}(\textstyle\prod_{i=1}^{n} (x_i \colon \rho_i)) = \max_{1 \leq i \leq n} \{\mathit{order}(\rho_i)\}\,. \end{aligned}$$

The syntax of types is subject to the usual scope rule; in $(x \colon \rho_1) \times \rho_2$ and $(x \colon \tau_1) \to \tau_2$, the scope of x is ρ_2 and τ_2 respectively. Furthermore, we require that every refinement predicate is well-typed and have type $\mathbf{int}$. Figure 2 shows the definition of this general well-formedness, which is applied in Section 2.2, and Appendix B. Here, we write $\mathtt{ST}(x_1 : \tau_1, \ldots, x_n : \tau_n)$ for $x_1 : \mathtt{ST}(\tau_1), \ldots, x_n : \mathtt{ST}(\tau_n)$, and $\Gamma \vdash_{\mathtt{ST}} t : \kappa$ means that t has simple type κ under simple type environment Γ. The judgment $\Gamma \vdash_{\mathtt{ST}} t : \kappa$ is defined in Appendix A. 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 logical relations. The connectives $\forall$ and $\wedge$ have genuine logical meaning, and especially they are commutative, so we often use the prenex normal form. Notice that the semantics of $t_1 \wedge t_2$ and $t_1 \mathbin{\&} t_2$ 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 $\models 1 : \{x : \mathbf{int} \mid \Omega \mathbin{\&} x = 0\}$ DOES hold, since $\Omega \mathbin{\&} x = 0$ 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,[5] we aim to develop a sound but incomplete method below. As explained in Section 1, our

[5]For a program t, we can reduce the halting problem of t to the verification problem of $\models t; \mathbf{fail} : \mathbf{int}$. Therefore, if we could solve the verification problem, then we could solve the halting problem.

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

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

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

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

$$\frac{}{\Gamma \vdash_{\mathtt{GWF}} \mathbf{int}} \qquad \text{(GWF-INT)}$$

$$\frac{\Gamma \vdash_{\mathtt{GWF}} \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{\mathtt{GWF}} \tau_2}{\Gamma \vdash_{\mathtt{GWF}} (x : \tau_1) \rightarrow \tau_2} \qquad \text{(GWF-FUN)}$$

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

$$\frac{}{\vdash_{\mathtt{GWF}} \emptyset} \qquad \text{(GWF-ENIL)}$$

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

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

Figure 2: General well-formedness of types

$\boxed{\text{(Predicate)}\ \models_{\rm p}\ \subseteq \{P : \text{closed}\}}$

- $\models_{\rm p} \forall x.\, P \overset{\rm def}{\iff} \models_{\rm p} P[x \mapsto m]$ for any integer m
- $\models_{\rm p} P_1 \wedge P_2 \overset{\rm def}{\iff} \models_{\rm p} P_1$ and $\models_{\rm p} P_1$
- $\models_{\rm p} t \overset{\rm def}{\iff} A = \mathbf{true}$ for any A s.t. $t \longrightarrow^* A$

$\boxed{\text{(Value)}\ \models_{\rm v} \subseteq \{V : \text{closed}\} \times \{\tau : \text{closed}\}}$

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

$\boxed{\text{(Term)}\ \models\ \subseteq \{t : \text{closed}\} \times \{\tau : \text{closed}\}}$

- $\models t : \tau \overset{\rm def}{\iff} \models_{\rm v} A : \tau$ for any A s.t. $t \longrightarrow^* A$

Figure 3: Semantics of types

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 [9, 13, 18], which combines higher-order model checking [8] and predicate abstraction.

We assume that the input program t is closed. If one wishes to verify an open term, e.g., a program t which uses a library function f, one can simply use the lambda abstraction $\lambda f.\, t$ as an input program and use $(f : \tau_1) \to \tau$ as an input specification where τ is the original specification for t and τ_1 is a given refinement type of f.

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 $\emptyset \vdash_{\tt WF} \tau$ 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:

$$\begin{aligned}
&\mathit{depth}(\{\nu : \sigma \mid P\}) = \mathit{depth}(\sigma)\\
&\mathit{depth}(\mathbf{int}) = 0\\
&\mathit{depth}((x : \tau_1) \to \tau_2) = 1 + \max\{\mathit{depth}(\tau_1), \mathit{depth}(\tau_2)\}\\
&\mathit{depth}(\textstyle\prod_{i=1}^n (x_i\colon \rho_i)) = \max_{1 \le i \le n} \{\mathit{depth}(\rho_i)\}\,.
\end{aligned}$$

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 $(x : \tau_1) \to \{\nu : \sigma \mid P\}$

$$\frac{\Gamma \mid \emptyset \vdash_{\texttt{WF}} P}{\Gamma \vdash_{\texttt{WF}} P} \quad \text{(WF-PredInit)}$$

$$\frac{\Gamma \mid \emptyset \vdash_{\texttt{WF}} \tau}{\Gamma \vdash_{\texttt{WF}} \tau} \quad \text{(WF-Init)}$$

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

$$\frac{\Gamma \mid \Delta \vdash_{\texttt{WF}} P_1 \qquad \Gamma \mid \Delta \vdash_{\texttt{WF}} P_2}{\Gamma \mid \Delta \vdash_{\texttt{WF}} P_1 \land P_2} \quad \text{(WF-PredAnd)}$$

$$\frac{\Gamma \mid \Delta,\, y_1 : \mathbf{int}, \ldots, y_n : \mathbf{int} \vdash_{\texttt{WF}} P}{\Gamma \mid \Delta \vdash_{\texttt{WF}} \forall y_1, \ldots, y_n.\, P} \quad \text{(WF-PredForAll)}$$

$$\frac{\Gamma, \Delta \mid \emptyset \vdash_{\texttt{WF}} \sigma \qquad \Gamma \mid \Delta,\, x : \sigma \vdash_{\texttt{WF}} P}{\Gamma \mid \Delta \vdash_{\texttt{WF}} \{x : \sigma \mid P\}} \quad \text{(WF-Refine)}$$

$$\frac{}{\Gamma \mid \Delta \vdash_{\texttt{WF}} \mathbf{int}} \quad \text{(WF-Int)}$$

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

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

$$\frac{}{\vdash_{\texttt{WF}} \emptyset} \quad \text{(WF-ENil)}$$

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

$$\texttt{ElimHO}_n(\Gamma) \overset{\text{def}}{=} \{(x : \tau) \in \Gamma \mid \mathit{depth}(\tau) \leq n\}$$

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

Figure 4: Well-formedness of types

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 $(f_1\colon\tau_1)\times\{f_2 : \sigma_2 \mid P_2\}\times\cdots\times\{f_n : \sigma_n \mid P_n\}$, 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_1t_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 fails.)

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 $\overset{?}{\models} t : \{\nu : \sigma \mid P\}$ rather than the top level refinement P; since given

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

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

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

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:

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

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 occurs 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

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

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

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

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

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

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

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

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 3 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 $(f\colon\tau) \to \{\nu : \sigma \mid P\}$ as described below. A judgment

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

can be transformed to

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

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:

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

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:

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

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

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

can be transformed to

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

(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.

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

$$(((y_k)_k{:}\tau) \to \tau')^{\sharp_1} \stackrel{\text{def}}{=} \left((y_k)_k\colon (\tau)^{\sharp_1}\right) \to \left((y'_k)_{k\in D(1)} : \tau^{(1)}\right) \times \left((\tau')^{\sharp_1}\,[y_k \mapsto y'_k]_{k\in D(1)}\right)$$

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

$$D(1) \stackrel{\text{def}}{=} \{k \mid \rho_k \text{ is depth-1}\}$$
$$\tau^{(1)} \stackrel{\text{def}}{=} \left\{(y_k)_{k\in D(1)} : \textstyle\prod_{k\in D(1)} (y_k\colon \rho_k) \,\middle|\, P\right\}$$

Note that $(\tau)^{\sharp_1} = \tau$ if τ is order at most 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 $(-)^{\sharp_1}$

$\sharp_1$*: 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_1t_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_1t_2; therefore, we extract them by applying the projection. For example, the term

$$\mathbf{fix}(f, \lambda(x, g).\, \mathbf{if}\ x \le 0\ \mathbf{then}\ 0\ \mathbf{else}\ g\, x + f\, (x - 1, g))$$

is transformed to

$$\mathbf{fix}(f, \lambda(x, g).\, (g, \mathbf{if}\ x \le 0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathbf{pr}_2(g\, x) + \mathbf{pr}_2(f\, (x - 1, g)))).$$

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^i_k such that $(z^i_k = x_i) \in \mathtt{argEq}(t_k)$; hence $\forall x_i$ can be eliminated by substituting z^i_k for x_i.

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

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

$$(\forall x_1, \ldots, x_n.\ \wedge_k\, t_k)^{\sharp_2} \overset{\text{def}}{=} \mathtt{eOQ}\big(\forall \widetilde{x_i}.\forall \widetilde{z_{k,l}} \wedge_k \left(\mathtt{argEq}(\mathtt{app}(t_k)) \;\mathtt{=>}\; \mathtt{sArg}(t_k)\right)\big)$$

$$\overset{\text{def}}{=} \forall \widetilde{z_{k,l}} \wedge_k \left(\left(\mathtt{argEq}(\mathtt{app}(t_k)) \;\mathtt{=>}\; \mathtt{sArg}(t_k)\right)[x_i \mapsto z^i_k]\right)$$

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

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

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

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

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

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

For example, consider the type

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

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

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

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

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

which may be simplified further to

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

$\sharp_3$*: 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 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.

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

where, $\phi = \boldsymbol{\{}\prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{m}(\phi_j \to \phi'_j) \mid M\boldsymbol{\}}$; $m_j = M(j)$; 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 $mul(P, j)$ ($m'_j \leq m_j$ since $\tau \leq_{mul} \phi$); and

$P' \overset{\text{def}}{=} P[a_{j,l} \mapsto f_{j,l}\, t_{j,l}]_{j\in\{1,\ldots,m\},l\in\{1,\ldots,m'_j\}}$ (where $a_{j,l} = f_j\, t_{j,l}$)

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

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

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

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

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:

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

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 $\phi = \boldsymbol{\{}\prod_{i=1}^{n}\mathbf{int} \times \prod_{j=1}^{m}(\phi_j \to \phi'_j) \mid M\boldsymbol{\}}$, we write $\tau \leq_{mul} \phi$ if all the multiplicities in τ are pointwise less than or equal to those in ϕ, i.e., if $mul(P, j) \leq M(j)$, $\tau_j \leq_{mul} \phi_j$, and $\tau'_j \leq_{mul} \phi'_j$ for all j. Intuitively, $\tau \leq_{mul} \phi$ 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 $\tau \leq_{mul} \phi$, 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 $\overset{?}{\models} t : \tau$, 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 $\tau \leq_{mul} \phi$, 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.

$\sharp_4$*: 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

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

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

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

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

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

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

$$\mathbf{let}\ f_1 = (t'_1)^{\sharp_4}\ \mathbf{in} \cdots \mathbf{let}\ f_m = (t'_m)^{\sharp_4}\ \mathbf{in}\ \big(x_1, \ldots, x_n, \lambda y.\, (f_1\,(\mathbf{pr}_1 y), \ldots, f_m\,(\mathbf{pr}_m y))\big)$$

where t_i are integers and t'_i are functions.

$$\big(\mathbf{pr}_i^{\mathbf{int}} t\big)^{\sharp_4} \overset{\mathrm{def}}{=} \mathbf{pr}_i\,(t)^{\sharp_4}$$

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

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

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 $(-)^{\sharp_4}$

in a refinement predicate are in $\{f_j \mid j \le m\}$; 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 $\hat{j}(-)$ from the set X of occurrences of applications in P to $\{j \mid j \le m\}$ such that for any application occurrence $f t$, $f = f_{\hat{j}(f t)}$; 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 $\sharp_2$:

$$\{\ (f, g)\colon (\mathbf{int} \to \mathbf{int}) \times (\mathbf{int} \to \mathbf{int}) \mid \forall z^{\langle fx\rangle}, z^{\langle gx\rangle}.\ z^{\langle fx\rangle} = z^{\langle gx\rangle} \text{ => } f\, z^{\langle fx\rangle} = g\, z^{\langle gx\rangle}\ \}.$$

The transformed type is of the form

$$((y_1, y_2)\colon \mathbf{int}_\perp \times \mathbf{int}_\perp) \to \Big\{(r_1, r_2) : \mathbf{int}_\perp \times \mathbf{int}_\perp \;\Big|\; (\ldots)^{\sharp_4}\Big\}.$$

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

$$((y_1, y_2)\colon \mathbf{int}_\perp \times \mathbf{int}_\perp) \to \{(r_1, r_2) : \mathbf{int}_\perp \times \mathbf{int}_\perp \mid y_1 = y_2 \text{ => } r_1 = r_2\}.$$

The transformation of terms follows the ideas described in Section 3.1 except that option types have been introduced. For example, the term $(\lambda y_1.\, t_1, \lambda y_2.\, t_2)$ is transformed into the term

$$\lambda(y_1, y_2).\, \big(\textbf{if } y_1 = \perp \textbf{ then } \perp \textbf{ else } (t_1)^{\sharp_4},\ \textbf{if } y_2 = \perp \textbf{ then } \perp \textbf{ else } (t_2)^{\sharp_4}\big). \tag{1}$$

Here, $\perp$ is the exception of option types (i.e. `None` in OCaml or `Nothing` in Haskell), and we have omitted a destruction 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.

$$\frac{\Phi(x) = \phi}{\Phi \vdash_{\mathrm{c}} x : \phi} \tag{C-Var}$$

$$\frac{}{\Phi \vdash_{\mathrm{c}} n : \boldsymbol{\{}\mathbf{int} \mid \emptyset\boldsymbol{\}}} \tag{C-Const}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t : \boldsymbol{\{}\mathbf{int} \mid \emptyset\boldsymbol{\}} \qquad \Phi \vdash_{\mathrm{c}} t_1 : \phi \qquad \Phi \vdash_{\mathrm{c}} t_2 : \phi}{\Phi \vdash_{\mathrm{c}} \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \phi} \tag{C-If}$$

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

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

$$\frac{\Phi \vdash_{\mathrm{c}} t : \boldsymbol{\{}\phi_1 \to \phi_2 \mid M\boldsymbol{\}} \qquad \Phi \vdash_{\mathrm{c}} t_1 : \phi_1}{\Phi \vdash_{\mathrm{c}} t\, t_1 : \phi_2} \tag{C-App}$$

$$\frac{\Phi \vdash_{\mathrm{c}} t_i : \boldsymbol{\{}\mathbf{int} \mid \emptyset\boldsymbol{\}} \qquad \Phi \vdash_{\mathrm{c}} t'_j : \boldsymbol{\{}\phi_j \to \phi'_j \mid M(j)\boldsymbol{\}} \qquad (\forall i, j)}{\Phi \vdash_{\mathrm{c}} (\widetilde{t_i}, \widetilde{t'_j}) : \boldsymbol{\{}\prod_{i=1}^{n} \mathbf{int} \times \prod_{j=1}^{m} (\phi_j \to \phi'_j) \mid M\boldsymbol{\}}} \tag{C-Tuple}$$

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

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

$$\frac{}{\Phi \vdash_{\mathrm{c}} \mathbf{fail} : \phi} \tag{C-Fail}$$

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

Figure 9: Type system for multiplicity types

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. The definition, which we give now, may be skipped at the first reading; intuitively, T and ϕ are consistent (for t and τ) if it makes consistent assumptions on each subterm, so that the result of the transformation is simply-typed.

First, we introduce a type system for a term t and a multiplicity type ϕ in Figure 9. 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 $(M(j))_{j \leq m}$.

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 $\Phi \vdash_{\mathrm{c}} t : \phi$ for 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. Hence, for a closed term t, we identify a consistent pair (T, ϕ) with a derivation of $\vdash_{\mathrm{c}} t : \phi$.

Theorem 1 (Soundness of Verification by $(-)^{\sharp}$)**.** *Let t be a closed term and τ be a type of order at most 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,*

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

We give a proof of this theorem in Appendix C. The converse of Theorem 1, completeness, holds for order-1 types, but not for order-2: see Section 4.2.

As explained in Section 3.2, we automatically infer ϕ and T such that they are consistent and $\tau \leq_{mul} \phi$, and gradually increase them until the verification succeeds. Thus, the transformation is automatic as a whole.

3.4. Sufficient Condition for Consistency

The next proposition gives a sufficient condition for consistency, which can be used 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 ($k \geq 0$) 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 2. *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. Straightforward by induction on t: for any $\Phi \vdash_{\mathrm{c}} t : \phi$ there is exactly one rule among the ten rules in Figure 9 whose conclusion part agree with $\Phi \vdash_{\mathrm{c}} t : \phi$, and we can choose at least one rule instance of the rule so that the assumption part consists of only judgments that are constant with k. □

Proposition 3. *For a multiplicity annotation T of a term t and a multiplicity type ϕ, if both 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 2, 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 $\Phi \vdash_{\mathrm{c}} t : \phi$. □

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 [12, 17, 16, 9, 19, 13]. 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 [20] (see Appendix B for such a refinement type system), or higher-order model checking [9, 8], 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`:

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

To deal with the problem above, we further refine the transformation $(-)^\sharp$ by (i) tupling of recursive functions [3] and (ii) insertion of assumptions.

4.1. Tupling of Recursion

The idea is that when a tuple of function calls is introduced by $(-)^{\sharp_4}$ ($(f_1\,(\mathbf{pr}_1 y), \dots, f_m\,(\mathbf{pr}_m y))$ 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:

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f, \lambda x.\,\mathbf{if}\ t_{11}\ \mathbf{then}\ t_{12}\ \mathbf{else}\ E_1[f\,t_1])\ \mathbf{in}\\ &\mathbf{let}\ g = \mathbf{fix}(g, \lambda y.\,\mathbf{if}\ t_{21}\ \mathbf{then}\ t_{22}\ \mathbf{else}\ E_2[g\,t_2])\ \mathbf{in}\ \dots (f, g) \dots\end{aligned}$$

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:

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

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 $t_{11} = \mathbf{true}$, $t_{12} = \mathbf{fail}$ and $t_{21} = \Omega$. The call to the original function fails, but without $\mathbf{let}\ \_ = f\,x'\ \mathbf{in} \cdots$, 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

$$\tau \overset{\text{def}}{=} \{(f,g) : (\mathbf{int} \to \mathbf{int})^2 \mid \forall x.\, f\,x > g\,x\} \to \{h : \mathbf{int} \to \mathbf{int} \mid \forall x.\, h\,x > 0\}.$$

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

$$\Big(\big((x_1,x_2) : \mathbf{int}^2\big) \to \big\{(r_1,r_2){:}\,\mathbf{int}^2 \mid x_1 = x_2 \Rightarrow r_1 > r_2\big\}\Big) \to \big(\mathbf{int} \to \{r{:}\,\mathbf{int} \mid r > 0\}\big).$$

Here, $\bot$ is used as a dummy argument as explained in Section 3.2-$\sharp_4$. We cannot conclude that `r1` $-$ `r2'` has type $\{r{:}\,\mathbf{int} \mid r > 0\}$ because there is no information about the correlation between `r1` and `r2'`: from the refinement type of `fg`, we can infer that $x = \bot \Rightarrow r_1 > r_2$ and $\bot = x \Rightarrow r_1' > r_2'$, but $r1 > r2'$ cannot be derived.[6] In fact, $\models$ $(\texttt{diff})^{\sharp} : (\tau)^{\sharp}$ does not hold,[7] 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 **if** t **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

```
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 now define a refined transformation $(-)^{\sharp'}$, which is the above assume-inserted version of $(-)^{\sharp}$. 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 10. Here, we write $\lambda(x_1,\dots,x_n).\,t$ for $\lambda x.\,\mathbf{let}\ x_1 = \mathbf{pr}_1 x\ \mathbf{in}\ \dots\mathbf{let}\ x_n = \mathbf{pr}_n x\ \mathbf{in}\ t$ where x is a fresh variable. By α-renaming, we assume that the two variables declared by any two different occurrences of let-expressions in an A-normal form are different.

We redefine the transformation $(-)^{\sharp_1}$ according to the normal form; the essential part of the new definition of $(-)^{\sharp_1}$ is shown in Figure 11. Output programs of this $(-)^{\sharp_1}$ are in another normal form defined in Figure 12, which is the domain of the transformation $(-)^{\sharp'_{34}}$, which is defined in the next subsection.

[6]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.

[7]To see this, apply $(\texttt{diff})^{\sharp}$ to

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

and apply the returned value to, say, 0.

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

Figure 10: Normal form before $(-)^{\sharp_1}$

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

Figure 11: $(-)^{\sharp_1}$ for normal forms

4.3. *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 13 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 $\mathtt{f}\,\mathtt{g}\,(x,x)$) and inserts the assumptions illustrated above. As $(-)^{\sharp_3}_T$, $(-)^{\sharp'_{34}}_T$ also depends on a multiplicity annotation T. In the figure, B is a set of bindings (i.e., pairs of variables and terms) that are used in `InstVar`; for a subterm t of a term t_0, B of $(t)^{\sharp'_{34}}_{T,B}$ occurring in the definition of $(t_0)^{\sharp'_{34}}_T$ is the set of all the bindings of destruction terms that are already declared at the position of t in t_0. Except for the case of applications, the definition of $(-)^{\sharp'_{34}}$ is just an obvious combination of those of $(-)^{\sharp_3}$ and $(-)^{\sharp_4}$.

Now, we define the auxiliary function `InstVar`:

$$
\mathtt{InstVar}(g, T, B, t) \quad \overset{\text{def}}{=} \quad \textbf{let } g' = \left(\begin{array}{l}
\lambda\widetilde{y}.\, \textbf{let } x = g\,\widetilde{y} \textbf{ in} \\
\quad \textbf{let } z^{\langle \alpha^1 \rangle} = B^\sharp_g(\alpha^1) \textbf{ in} \\
\quad \cdots \\
\quad \textbf{let } z^{\langle \alpha^q \rangle} = B^\sharp_g(\alpha^q) \textbf{ in} \\
\quad \textbf{assume}\,(p)\,; \textbf{assume}\,(p')\,; x
\end{array}\right) \textbf{in } t[g \mapsto g'] \qquad (2)
$$

where $\alpha^1, \dots, \alpha^q$ are an enumeration: $\prod_{j \le m} \mathtt{App}^*_j = \{\alpha^1, \dots, \alpha^q\}$; and the set $\mathtt{App}^*_j$, the function $B^\sharp_g$, the two formulas p and p', and the variables $z^{\langle \alpha^i \rangle}$ 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* in an input program is unchanged (i.e., $(x)^{\sharp_3} \overset{\text{def}}{=} x$ and $(x)^{\sharp_4} \overset{\text{def}}{=} x$), although by $(-)^{\sharp_3}$, each *subterm* of a function type (i.e., $\textbf{fix}(f, \lambda x.\, t)$ and f in $\textbf{fix}(f, \lambda x.\, t)$) is duplicated, and by $(-)^{\sharp_4}$, each *subterm* of a tuple type (i.e., $(t_1, \dots, t_n, t'_1, \dots, t'_m)$) 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 function *variables* behave as the product of duplicated functions, while they in fact behave so since they are instantiated with some *subterms* of the input program. 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 $(g = \textbf{pr}^{\rightarrow}_k z') \in B$ 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

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

Figure 12: Normal form before $(-)^{\sharp'_{34}}$

$(t)^{\sharp'_{34}}_{T} \overset{\text{def}}{=} (t)^{\sharp'_{34}}_{T,\emptyset}$

$((x_1,...,x_n,f_1,...,f_m))^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} (x_1,...,x_n,\lambda(y_1,...,y_m).\,(f_1\,y_1,...,f_m\,y_m))$

$(\textbf{if}\ x\ \textbf{then}\ t_1\ \textbf{else}\ t_2)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \textbf{if}\ x\ \textbf{then}\ (t_1)^{\sharp'_{34}}_{T,B}\ \textbf{else}\ (t_2)^{\sharp'_{34}}_{T,B}$

$(\textbf{let}\ x = e\ \textbf{in}\ t)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \textbf{let}\ x = (e)^{\sharp'_{34}}_{T,B}\ \textbf{in}\ (t)^{\sharp'_{34}}_{T,B}$

if $e = n$, $\mathsf{op}(\widetilde{x})$, $\textbf{fix}(f,\lambda(\widetilde{x},\widetilde{g}).\,t')$, $(\widetilde{x},\widetilde{f})$, or $\textbf{fail}$

$(\textbf{let}\ x = e\ \textbf{in}\ t)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \textbf{let}\ x = (e)^{\sharp'_{34}}_{T,B}\ \textbf{in}\ (t)^{\sharp'_{34}}_{T,B\cup\{x=e\}}$

if $e = f\,(\widetilde{x},\widetilde{g})$, or $\textbf{pr}_i x$

$(e)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} e \qquad$ if $e = n$, $\mathsf{op}(\widetilde{x})$, or $\textbf{fail}$

$(\textbf{fix}(f,\lambda(x_1,\dots,x_n,g_1,\dots,g_m).\,t'))^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \textbf{fix}(f,\lambda(z_1,\dots,z_\ell).\,(t'_1,\dots,t'_\ell))$

where $\ell \overset{\text{def}}{=} T(\textbf{fix}(f,\lambda(x_1,\dots,x_n,g_1,\dots,g_m).\,t'))$

$t'_k \overset{\text{def}}{=} (t')^{\sharp'_{34}}_{T,B}\,[x_i \mapsto \textbf{pr}_i z_k]_{i\le n}[g_j \mapsto p^{z_k}_j]_{j\le m}$

$p^{z_k}_j \overset{\text{def}}{=} \lambda y_j.\,\textbf{pr}_j((\textbf{pr}_{n+1} z_k)(\overrightarrow{\bot}^{j-1}, y_j, \overrightarrow{\bot}^{m-j}))$

$(f(x_1,...,x_n,g_1,...,g_m))^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \texttt{InstVar}(f,T,B,t_{m+1})$

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

$t_1 \overset{\text{def}}{=} \textbf{pr}_1(f(\overrightarrow{z}^{T(f)}))$

$t_{j+1} \overset{\text{def}}{=} \texttt{InstVar}(g_j,T,B,t_j) \quad$ (for $j = 1,...,m$)

$((x_1,...,x_n,f_1,...,f_m))^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} (x_1,...,x_n,\lambda(y_1,...,y_m).\,(f_1\,y_1,...,f_m\,y_m))$

$\left(\textbf{pr}^{\textbf{int}}_i x\right)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \textbf{pr}_i\,x$

$\left(\textbf{pr}^{\to}_j x\right)^{\sharp'_{34}}_{T,B} \overset{\text{def}}{=} \lambda y.\,\textbf{pr}_j((\textbf{pr}_{n+1} x)(\overrightarrow{\bot}^{j-1}, y_j, \overrightarrow{\bot}^{m-j}))$

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

Figure 13: Refined encoding function refinement $(-)^{\sharp'_{34}}$

applying $(-)^{\sharp'}$, the simple type of z is of the following form:

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

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:

$$\mathbf{int}^n \times \left(\prod_{j=1}^{m} \left((\tau_j)^{\sharp'}_T \right)^{\mathtt{m}_j} \to \prod_{j=1}^{m} \left((\tau'_j)^{\sharp'}_T \right)^{(\mathtt{m}_j)}_{[\cdot \mapsto \cdot]} \right) \tag{3}$$

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 $\widetilde{y} = (y_l)_{l \le \mathtt{m}_k}$ 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

$$\mathtt{App}'_j \overset{\mathrm{def}}{=} \left\{ (u, v, w) \mid (v = \mathbf{pr}^{\to}_j z), (w = v\,u) \in B, \mathit{depth}(v) = 1 \right\}$$

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

$$\mathtt{App}_j \overset{\mathrm{def}}{=} \begin{cases} \mathtt{App}'_j \cup \{y_1, \ldots, y_{\mathtt{m}_k}\} & (\text{if } j = k \text{ and } \mathit{depth}(g) = 1) \\ \mathtt{App}'_j & (\text{otherwise, if } \mathtt{App}'_j \text{ is non-empty}) \\ \{(\bot, v) \mid (v = \mathbf{pr}^{\to}_j z) \in B\} & (\text{otherwise}) \end{cases}$$

where y is the bound variable in (2).

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

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

and "function part" as

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

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

$$\mathtt{App}^*_j \overset{\mathrm{def}}{=} \left\{ (a_i)_{i \le \mathtt{m}_j} \in \mathtt{App}_j^{\mathtt{m}_j} \mid \mathtt{fun}_j(a_i) = \mathtt{fun}_j(a_1) \right\}$$

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

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

4. We define "$(-)^{\sharp}$ of z", which is specialized to the "application information" collected so far from B.

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

$$B^{\sharp}_g((\alpha_j)_j) \overset{\mathrm{def}}{=} (\mathbf{pr}^{\sharp}_{\to} z)(\mathtt{arg}^*_1 \alpha_1, \ldots, \mathtt{arg}^*_m \alpha_m)\ .$$

5. Finally, we define p and p' in (2) as

$$p \overset{\text{def}}{=} \mathop{\&}_{\substack{(\alpha_j)_j,(\alpha'_j)_j \in \prod_{j \le m} \mathtt{App}^*_j,\\ j \le m,\ \mathtt{fun}^*_j(\alpha_j)=\mathtt{fun}^*_j(\alpha'_j)}} \Big(\mathtt{arg}^*_j(\alpha_j) \mathrel{==} \mathtt{arg}^*_j(\alpha'_j) \ \ \texttt{=>}\ \ \mathbf{pr}_j z^{\langle((\alpha_j)_j)\rangle} \mathrel{==} \mathbf{pr}_j z^{\langle((\alpha'_j)_j)\rangle}\Big)$$

$$p' \overset{\text{def}}{=} \mathop{\&}_{\substack{(\alpha_j)_j \in \prod_{j \le m} \mathtt{App}^*_j,\\ j \le m,\ i \le \mathtt{m}_j,\ \pi_i \alpha_j=(u,v,w)}} \Big(w \mathrel{==} \mathbf{pr}_i \mathbf{pr}_j z^{\langle((\alpha_j)_j)\rangle}\Big)$$

where we prepare a fresh variable $z^{\langle(\alpha_j)_j\rangle}$ for each $(\alpha_j)_j \in \prod_{j \le m} \mathtt{App}^*_j$.

In the target language, **fail** is treated as an exception, and we define $\mathbf{assume}\ (t)$ as a shorthand for:

$$\mathbf{if}\ (\mathbf{try}\ t\ \mathbf{with\ fail} \to \mathbf{false})\ \mathbf{then\ true\ else}\ \mathtt{loop}(\,).$$

Note that our backend model checker MoCHi [9, 13] supports exceptions.

After replacing $(-)^{\sharp}$ with $(-)^{\sharp'}$, Theorem 1 is still valid; we give the proof in Appendix D.

Theorem 4 (Soundness of Verification by $(-)^{\sharp'}$)**.** *Let t be a closed term and τ be a type of order at most 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 $\tau \le_{mul} \phi$. Then,*

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

4.4. Example: Verification of "append-xs-nil"

In this subsection, we show that how our method works for 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. 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:

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

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

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 [9, 13] for a subset of OCaml.

Table 1: Results of experiments

program	size (before $\sharp'$)	size (after $\sharp'$)	predicates	time [sec]	
				transformation	model checking
sum-acc	56	282	0	< 0.01	0.54
sum-simpl	40	270	0	< 0.01	0.75
sum-mono	27	279	0	< 0.01	0.45
mult-acc	63	347	0	< 0.01	0.38
a-max-gen	112	476	1	< 0.01	0.29
append-xs-nil	72	1364	0	< 0.01	45.57
append-nil-xs	63	725	0	< 0.01	16.43
rev	128	1868	0	< 0.01	176.24
insert	32	6262	0	< 0.01	52.49
insertsort	38	7044	2	< 0.01	14.33

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.[8] The column "predicates" 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., $\forall m, n.\ m \leq n \Rightarrow \mathtt{sum}(m) \leq \mathtt{sum}(n)$. 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
    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 "insertsort" sorts an input list by the standard insertion sort algorithm, and the program "insert" is the insertion function utilized in the insertion sort. 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** $\rightarrow$ **int** $\rightarrow$ **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", the verifier checks that `append` has the type

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

where $\tau \overset{\text{def}}{=} \textbf{int} \rightarrow (\textbf{int option})$. For the programs "insert" and "insertsort", the verifier checks that

$$\mathtt{insert} : \textbf{int} \rightarrow \mathtt{sortedlist} \rightarrow \mathtt{sortedlist}$$
$$\mathtt{insertsort} : \textbf{int} \rightarrow \textbf{int}\ \mathtt{list} \rightarrow \mathtt{sortedlist}$$

[8]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.

where

$$\texttt{sortedlist} \stackrel{\text{def}}{=} \{xs\colon \mathbf{int} \to (\mathbf{int\ option}) \mid \forall i, j.\ xs(i) \leq xs(j)\}.$$

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 for "a-max-gen" and "insort-sort", for which we had to provide predicates by hand as hints (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 [6, 7] 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[9] refinement type checking [12, 23] 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. [18] 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 [18]) can be verified with their approach.

Suter et al. [14, 15] 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 [2, 1], and then verification conditions are generated and discharged by SMT solvers. Xu's method [22, 21] 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.

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

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 non-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. For example, as described in Section 2, we allow only top-level quantifiers over the base type and first-order function variables in refinement predicates. Relaxing this limitation is left for future work. Also, our current implementation should be improved further. First, as shown in Section 5, the verifier may increase the code size significantly. We need to suppress the increase by some optimizations of the transformation for improving the scalability. Secondly, producing a better report of the verification result is also left for future work; currently, the verifier just outputs “No” when a program does not satisfy a given specification.

Acknowledgment

We would like to thank Naohiko Hoshino and anonymous referees for useful comments. This work was supported by Kakenhi 23220001 and 15H05706.

[1] 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.
[2] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *TOPLAS*, 33(2): 8:1–8:45, Jan. 2011.
[3] W.-N. Chin. Towards an automated tupling strategy. In *PEPM ’93*, pages 119–132, 1993.
[4] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-07143-6.
[5] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *Theor. Comput. Sci.*, 221(1-2):393–456, June 1999.
[6] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP ’07*, pages 505–519, 2007.
[7] K. L. Knowles and C. Flanagan. Hybrid type checking. *TOPLAS*, 32(2), Jan. 2010. ISSN 0304-3975.
[8] N. Kobayashi. Model checking higher-order programs. *J. ACM*, 60(3):20:1–20:62, 2013.
[9] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI ’11*, pages 222–233, 2011.
[10] E. Moggi. Computational lambda-calculus and monads. In *LICS ’89*, pages 14–23, 1989.
[11] 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.
[12] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI ’08*, pages 159–169, 2008.
[13] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM ’13*, pages 53–62, 2013.
[14] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL ’10*, pages 199–210, 2010.
[15] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS ’11*, pages 298–315, 2011.
[16] T. Terauchi. Dependent types from counterexamples. In *POPL ’10*, pages 119–130, 2010.
[17] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP ’09*, pages 277–288, 2009.
[18] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL ’13*, pages 75–86, 2013.
[19] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP ’13*, pages 209–228, 2013.
[20] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL ’99*, pages 214–227, 1999.
[21] D. N. Xu. Hybrid contract checking via symbolic simplification. In *PEPM ’12*, pages 107–116, 2012.
[22] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Workshop on Haskell*, pages 41–52, 2009.
[23] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *VMCAI ’13*, pages 295–314, 2013.

Appendix A. A Simple Type System

This section gives the simple type system mentioned in Section 2. The syntax of simple types is given by:

$$\kappa \text{ (simple types) } ::= \mathbf{int} \mid \kappa_1 \to \kappa_2 \mid \textstyle\prod_{i=1}^{n} \kappa_i$$

A simple type environment Γ is a set of type bindings of the form $x : \kappa$. The typing rules are given in Figure A.14.

$$\frac{\Gamma(x) = \kappa}{\Gamma \vdash_{\mathsf{ST}} x : \kappa} \tag{ST-VAR}$$

$$\frac{}{\Gamma \vdash_{\mathsf{ST}} n : \mathbf{int}} \tag{ST-CONST}$$

$$\frac{\Gamma \vdash_{\mathsf{ST}} t : \mathbf{int} \qquad \Gamma \vdash_{\mathsf{ST}} t_1 : \kappa \qquad \Gamma \vdash_{\mathsf{ST}} t_2 : \kappa}{\Gamma \vdash_{\mathsf{ST}} \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 : \kappa} \tag{ST-IF}$$

$$\frac{\text{The arity of } [\![\mathsf{op}]\!] \text{ is } n \qquad \Gamma \vdash_{\mathsf{ST}} t_i : \mathbf{int} \quad \text{for all } i \leq n}{\Gamma \vdash_{\mathsf{ST}} \mathsf{op}(t_1, \ldots, t_n) : \mathbf{int}} \tag{ST-OP}$$

$$\frac{\Gamma,\ f : \kappa_1 \to \kappa_2,\ x_1 : \kappa_1 \vdash_{\mathsf{ST}} t : \kappa_2}{\Gamma \vdash_{\mathsf{ST}} \mathbf{fix}(f, \lambda x_1.\, t) : \kappa_1 \to \kappa_2} \tag{ST-FIX}$$

$$\frac{\Gamma \vdash_{\mathsf{ST}} t : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\mathsf{ST}} t_1 : \kappa_1}{\Gamma \vdash_{\mathsf{ST}} t\, t_1 : \kappa_2} \tag{ST-APP}$$

$$\frac{\Gamma \vdash_{\mathsf{ST}} t_i : \kappa_i \quad \text{for all } i \leq n}{\Gamma \vdash_{\mathsf{ST}} (t_1, \ldots, t_n) : \prod_{i=1}^{n} \kappa_i} \tag{ST-TUPLE}$$

$$\frac{\Gamma \vdash_{\mathsf{ST}} t : \prod_{i=1}^{n} \kappa_i}{\Gamma \vdash_{\mathsf{ST}} \mathbf{pr}_i t : \kappa_i} \tag{ST-PROJ}$$

$$\frac{}{\Gamma \vdash_{\mathsf{ST}} \mathbf{fail} : \kappa} \tag{ST-FAIL}$$

Figure A.14: Typing rules

Appendix 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 [12, 17, 16, 9, 19, 13] 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 $\vdash_{\mathtt{GWF}}$ (defined in Section 2.1), 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 $\vdash_{\mathtt{GWF}} \Gamma$, we define $\eta \models_{\mathrm{e}} \Gamma$ as follows:

$$\emptyset \models_{\mathrm{e}} \emptyset \overset{\mathrm{def}}{\iff} \text{true}$$

$$\eta \cup \{x \mapsto V\} \models_{\mathrm{e}} \Gamma, x : \tau \overset{\mathrm{def}}{\iff} \eta \models_{\mathrm{e}} \Gamma \;\text{ and }\; \models_{\mathrm{v}} V : \tau[\eta]$$

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

The *general refinement type system* is given in Figures B.15 and B.16. 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 the following condition:

$$\begin{array}{l} \text{for any } \Gamma,\, P, \text{ and } P', \text{ if } \Gamma \mid P \vdash^{\mathcal{L}} P', \\ \text{then for any } \eta \text{ such that } \eta \models_{\mathrm{e}} \Gamma,\; \models_{\mathrm{p}} P[\eta] \text{ implies } \models_{\mathrm{p}} P'[\eta]. \end{array} \tag{B.1}$$

In Figure B.15, we define $t'\llparenthesis x \leftarrow t \rrparenthesis$ as $\mathbf{let}\ x = t\ \mathbf{in}\ t'$, and extend it to the operations $P\llparenthesis x \leftarrow t \rrparenthesis$ and $\sigma\llparenthesis x \leftarrow t \rrparenthesis$ compositionally. For example, $(\forall y.t_1 \wedge t_2)\llparenthesis x \leftarrow t \rrparenthesis = \forall y.(t_1\llparenthesis x \leftarrow t \rrparenthesis) \wedge (t_2\llparenthesis x \leftarrow t \rrparenthesis)$. We define $t\llparenthesis x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n \rrparenthesis$ as $((t\llparenthesis x_n \leftarrow t_n \rrparenthesis) \cdots)\llparenthesis x_1 \leftarrow t_1 \rrparenthesis$.

The typing rules are similar to those of Knowles and Flanagan [6]. 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.

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

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 arithmetic with beta equality. We can show that `sum` has type

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

by the following derivation

$$\dfrac{\dfrac{\vdots \qquad \dfrac{\vdots \qquad \nu : \mathbf{int} \to \mathbf{int} \mid \mathbf{true} \vdash^{\mathcal{L}} P\llparenthesis f \leftarrow t \rrparenthesis}{\vdash^{\mathcal{L}}_{\mathrm{s}} \mathbf{int} \to \mathbf{int} <: \{f : \mathbf{int} \to \mathbf{int} \mid P\llparenthesis f \leftarrow t \rrparenthesis\}}\;\textsc{Sub-Refine}}{\vdash t : \{f : \mathbf{int} \to \mathbf{int} \mid P\llparenthesis f \leftarrow t \rrparenthesis\}}\;\text{T-}\textsc{Sub}}{\vdash t : \{f : \mathbf{int} \to \mathbf{int} \mid P\}}\;\text{T-}\textsc{Subst}$$

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

$$\begin{aligned} P\llparenthesis f \leftarrow t \rrparenthesis &\iff \forall x.\, x \geq 0 \Rightarrow t\,x = x + t\,(x-1) \\ &\iff \forall x.\, x \geq 0 \Rightarrow x + t\,(x-1) = x + t\,(x-1) \end{aligned}$$

and $x + t\,(x-1) = x + t\,(x-1)$ is valid in $\mathcal{L}$, $P\llparenthesis f \leftarrow t \rrparenthesis$ is valid in $\mathcal{L}$.

The type system is sound with respect to the semantics of types:

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

$$\frac{}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} n : \mathbf{int}} \qquad \text{(T-}\textsc{Const}\text{)}$$

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

$$\frac{\text{The arity of } [\![\mathsf{op}]\!] \text{ is } n \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t_i : \mathbf{int} \quad \text{for all } i \leq n}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} \mathsf{op}(t_1, \ldots, t_n) : \mathbf{int}} \qquad \text{(T-}\textsc{Op}\text{)}$$

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

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

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

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

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

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

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

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

Figure B.15: Typing rules

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

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

$$\frac{\Gamma, x_1 : \rho_1, \ldots, x_{i-1} : \rho_{i-1} \vdash^{\mathcal{L}}_{\mathrm{s}} \rho_i <: \rho_i' \quad \text{for all } i \le n}{\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \prod_{i=1}^{n} (x_i \colon \rho_i) <: \prod_{i=1}^{n} (x_i \colon \rho_i')} \quad \text{(SUB-TUPLE)}$$

$$\frac{}{\vdash^{\mathcal{L}}_{\mathrm{es}} \emptyset <: \emptyset} \quad \text{(ENVSUB-NIL)} \qquad \frac{\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma' \qquad \Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'}{\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma,\, x : \tau <: \Gamma',\, x : \tau'} \quad \text{(ENVSUB-CONS)}$$

Figure B.16: Subtyping rules

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

The rest of this section is devoted to the proof of this theorem. We generalize the semantics of types with environments as below

$$\Gamma \models t : \tau \quad \overset{\text{def}}{\Longleftrightarrow} \quad \models t[\eta] : \tau[\eta] \text{ for any } \eta \text{ such that } \eta \models_{\mathrm{e}} \Gamma$$

and we show Theorem 5 as a corollary of the following lemma:

Lemma 6. *For any $\mathcal{L}$, if $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$, then $\Gamma \models t : \tau$.*

Before proving Lemma 6, we show three lemmas.

Lemma 7 (Soundness of Subtyping)**.** *For any $\mathcal{L}$, if $\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'$, then for any η such that $\eta \models_{\mathrm{e}} \Gamma$ and for any V,*

$$\models_{\mathrm{v}} V : \tau[\eta] \quad \textit{implies} \quad \models_{\mathrm{v}} V : \tau'[\eta].$$

Also, if $\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma'$, then for any η,

$$\eta \models_{\mathrm{e}} \Gamma \quad \textit{implies} \quad \eta \models_{\mathrm{e}} \Gamma'.$$

Proof. We prove this lemma by induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{s}} \tau <: \tau'$ and $\vdash^{\mathcal{L}}_{\mathrm{es}} \Gamma <: \Gamma'$. All the cases are straightforward; we explain only two cases.
$\boxed{\text{(SUB-REFINE)}}$ Immediate from the induction hypothesis and the assumption (B.1) on $\mathcal{L}$.
$\boxed{\text{(SUB-FUN)}}$ Suppose the induction hypotheses:

(i) for any η such that $\eta \models_{\mathrm{e}} \Gamma$ and for any V_1, if $\models_{\mathrm{v}} V_1 : \tau_1'[\eta]$ then $\models_{\mathrm{v}} V_1 : \tau_1[\eta]$,

(ii) for any η' such that $\eta' \models_{\mathrm{e}} (\Gamma,\, x_1 : \tau_1')$ and for any V_2, if $\models_{\mathrm{v}} V_2 : \tau_2[\eta']$ then $\models_{\mathrm{v}} V_2 : \tau_2'[\eta']$.

For given η such that $\eta \models_{\mathrm{e}} \Gamma$ and for given V, let us assume $\models_{\mathrm{v}} V : ((x_1 \colon \tau_1) \to \tau_2)[\eta]$, i.e.,

(iii) for any V_1 such that $\models_{\mathrm{v}} V_1 : \tau_1[\eta]$ and any A_2 such that $V V_1 \longrightarrow^* A_2$, we have $\models_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1]$.

Then, we show that $\models_{\mathrm{v}} V : ((x_1 \colon \tau_1') \to \tau_2')[\eta]$, i.e., for given V_1 such that $\models_{\mathrm{v}} V_1 : \tau_1'[\eta]$ and given A_2 such that $V V_1 \longrightarrow^* A_2$, we show $\models_{\mathrm{v}} A_2 : \tau_2'[\eta][x_1 \mapsto V_1]$.

Since $\eta \models_{\mathrm{e}} \Gamma$ and $\models_{\mathrm{v}} V_1 : \tau_1'[\eta]$, by (i), we have $\models_{\mathrm{v}} V_1 : \tau_1[\eta]$. Hence, since $V V_1 \longrightarrow^* A_2$ and by (iii),

$$\models_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1]. \tag{B.2}$$

Let $\eta' := \eta \cup \{x_1 \mapsto V_1\}$. Since $\eta \models_{\mathrm{e}} \Gamma$ and $\models_{\mathrm{v}} V_1 : \tau_1'[\eta]$, we have $\eta' \models_{\mathrm{e}} (\Gamma,\, x_1 : \tau_1')$. By (B.2), A_2 is a value and $\models_{\mathrm{v}} A_2 : \tau_2[\eta']$; hence, by (ii), $\models_{\mathrm{v}} A_2 : \tau_2'[\eta']$, i.e., $\models_{\mathrm{v}} A_2 : \tau_2'[\eta][x_1 \mapsto V_1]$. □

Lemma 8. *For any P, t_0, and t_1 such that $t_0 \longrightarrow t_1$,*

$$\models_{\mathrm{p}} P(\!\![x \leftarrow t_0]\!\!) \quad \textit{iff} \quad \models_{\mathrm{p}} P(\!\![x \leftarrow t_1]\!\!).$$

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

$$\models_{\mathrm{v}} V : \tau(\!\![x \leftarrow t_0]\!\!) \quad \textit{iff} \quad \models_{\mathrm{v}} V : \tau(\!\![x \leftarrow t_1]\!\!).$$

Proof. The proof is trivial by the inductions on the syntax of P and τ. $\square$

To prove Lemma 6, we use so-called *unwinding theorem* [4], which we now explain. First, for $n \geq 0$, we define "approximations" of **fix**:

$$\mathbf{fix}^{(0)}(f, \lambda x.\, t) \overset{\text{def}}{=} \lambda x.\, \Omega$$
$$\mathbf{fix}^{(n+1)}(f, \lambda x.\, t) \overset{\text{def}}{=} \lambda x.\, t[f \mapsto \mathbf{fix}^{(n)}(f, \lambda x.\, t)].$$

Also, we use the usual notion of a context:

$$C ::= [\,] \mid \mathtt{op}(\widetilde{t}, C, \widetilde{t}) \mid \mathbf{if}\ C\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \mathbf{if}\ t\ \mathbf{then}\ C\ \mathbf{else}\ t_2 \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ C$$
$$\mid \mathbf{fix}(f, \lambda x.\, C) \mid C\, t_2 \mid t_1\, C \mid (\widetilde{t}, C, \widetilde{t}) \mid \mathbf{pr}_i C$$

Lemma 9 (Unwinding Theorem)**.** *For a context C and a term $\mathbf{fix}(f, \lambda x.\, t)$ such that $C[\mathbf{fix}(f, \lambda x.\, t)]$ is closed, and for an answer A which is an integer or fail, if $C[\mathbf{fix}(f, \lambda x.\, t)] \longrightarrow^* A$, then for some n, $C[\mathbf{fix}^{(n)}(f, \lambda x.\, t)] \longrightarrow^* A$.*

Proof. Obvious from the adequacy of the standard (Scott's) CPO-semantics (with an exception *fail*), the continuity of C, the compactness of integers and *fail*, and the fact that $\mathbf{fix}(f, \lambda x.\, t)$ is the limit of $(\mathbf{fix}^{(n)}(f, \lambda x.\, t))_n$. (The usage of these properties is the same as that in the proof of Lemma 17.) $\square$

Proof of Lemma 6. We prove this lemma by induction on the derivations of $\Gamma \vdash^{\mathcal{L}}_{\mathrm{t}} t : \tau$. An important case is (T-FIX), where we use the unwinding theorem. The other cases are straightforward except that we use Lemmas 8 and 7 there.
$\boxed{\text{(T-VAR)}}$ Trivial.
$\boxed{\text{(T-CONST)}}$ Trivial.
$\boxed{\text{(T-IF)}}$ Suppose the induction hypotheses:

(i) for any η and V,

$$\begin{aligned}&\text{if } \ \eta \cup \{x \mapsto V\} \models_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P \wedge \nu = \mathbf{true}\})\\ &\text{then } \ \models t_1[\eta][x \mapsto V] : \tau[\eta][x \mapsto V],\end{aligned}$$

(ii) for any η and V,

$$\begin{aligned}&\text{if } \ \eta \cup \{x \mapsto V\} \models_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P \wedge \nu \neq \mathbf{true}\})\\ &\text{then } \ \models t_2[\eta][x \mapsto V] : \tau[\eta][x \mapsto V].\end{aligned}$$

Then, for given η and V such that

$$\eta \cup \{x \mapsto V\} \models_{\mathrm{e}} (\Gamma,\, x : \{\nu : \mathbf{int} \mid P\}) \tag{B.3}$$

we show

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

From (B.3), $\models_{\mathrm{v}} V : \mathbf{int}$, and hence V is an integer. We suppose $V = \mathbf{true}$; in the case $V \neq \mathbf{true}$ the proof is similar.

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

$$\models t_1[\eta][x \mapsto V] : \tau[\eta][x \mapsto V].$$

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

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

Therefore,

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

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

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

(i) for any η' such that $\eta' \models_{\mathrm{e}} \Gamma,\ f : (x_1 \colon \tau_1) \to \tau_2,\ x_1 : \tau_1$, we have $\models t_2[\eta'] : \tau_2[\eta']$,

and that $f \notin FV(\tau_1) \cup FV(\tau_2)$. For given η such that $\eta \models_{\mathrm{e}} \Gamma$, we show

$$\models_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, t_2)[\eta] : ((x_1 \colon \tau_1) \to \tau_2)[\eta]. \tag{B.4}$$

First, we show that, for any $n \geq 0$,

$$\models_{\mathrm{v}} \mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta]) : (x_1 : \tau_1[\eta]) \to \tau_2[\eta] \tag{B.5}$$

by induction on n. The base case ($n = 0$) is trivial.

For $n \geq 0$, we show the case of $n + 1$, i.e., for given V_1 such that

$$\models_{\mathrm{v}} V_1 : \tau_1[\eta] \tag{B.6}$$

and given A_2 such that

$$\mathbf{fix}^{(n+1)}(f, \lambda x_1.\, t_2[\eta])\, V_1 \longrightarrow^* A_2$$

we show that

$$\models_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1].$$

Let $\eta' \overset{\text{def}}{=} \eta \cup \left\{ f \mapsto \mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta]) \right\} \cup \{x_1 \mapsto V_1\}$; by the induction hypothesis of (B.5) on n, by (B.6), and since $f \notin FV(\tau_1)$,

$$\eta' \models_{\mathrm{e}} \Gamma,\ f : (x_1 \colon \tau_1) \to \tau_2,\ x_1 : \tau_1.$$

Hence, by (i), and since $f \notin FV(\tau_2)$, we have

$$\models t_2[\eta'] : \tau_2[\eta][x_1 \mapsto V_1].$$

Therefore, since

$$\begin{aligned}&\mathbf{fix}^{(n+1)}(f, \lambda x_1.\, t_2[\eta])\, V_1 \longrightarrow\\ &t_2[\eta][f \mapsto \mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta])][x_1 \mapsto V_1] \ = \ t_2[\eta'] \longrightarrow^* A_2\end{aligned}$$

we have $\models_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto V_1]$.

Now, we show (B.4), i.e., for given V_1 such that

$$\models_{\mathrm{v}} V_1 : \tau_1[\eta]$$

and given A_2 such that

$$\mathbf{fix}(f, \lambda x_1.\, t_2[\eta])\, V_1 \longrightarrow^* A_2$$

we show that

$$\models_{\mathrm{v}} A_2 : \tau_2[\eta^1] \tag{B.7}$$

where $\eta^1 \overset{\text{def}}{=} \eta \cup \{x_1 \mapsto V_1\}$.

If $A_2 = \mathit{fail}$, by the unwinding theorem, for some n,

$$\mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta])\, V_1 \longrightarrow^* A_2.$$

This contradicts (B.5); hence A_2 is a value.

The rest of the proof for (T-FIX) is basically similar to the above case that $A_2 = \mathit{fail}$: we use the unwinding theorem for a reduction sequence to *fail* or an integer, and use (B.5).

If τ_2 is of the form $\{\nu^2 : \mathbf{int} \mid P^2\}$, then A_2 is some integer; hence, by the unwinding theorem and (B.5), the goal (B.7) follows.

Let τ_2 be of the form $\{f^2 : (x_1^2 \colon \tau_1^2) \to \tau_2^2 \mid P^2\}$. First, we show $\models_{\mathrm{p}} P^2[\eta^1][f^2 \mapsto A_2]$. Let P^2 be of the form $\forall \widetilde{y}.\, \wedge_k P_k^2$ where P_k^2 are integer terms. For given integers $\widetilde{m}$, k, and A such that $P_k^2[\eta^1][f^2 \mapsto A_2][\widetilde{y} \mapsto \widetilde{m}] \longrightarrow^* A$, we have to show $A = \mathbf{true}$. Since

$$\begin{aligned}&\mathbf{let}\ f^2 = \mathbf{fix}(f, \lambda x_1.\, t_2[\eta])\, V_1\ \mathbf{in}\ P_k^2[\eta^1][\widetilde{y} \mapsto \widetilde{m}] \longrightarrow^* \\ &P_k^2[\eta^1][f^2 \mapsto A_2][\widetilde{y} \mapsto \widetilde{m}] \longrightarrow^* A,\end{aligned}$$

by the unwinding theorem, for some n

$$\mathbf{let}\ f^2 = \mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta])\, V_1\ \mathbf{in}\ P_k^2[\eta^1][\widetilde{y} \mapsto \widetilde{m}] \longrightarrow^* A.$$

Hence, by (B.5), we have $A = \mathbf{true}$.

To show $\models_{\mathrm{v}} A_2 : (x_1^2 \colon \tau_1^2[\eta^1]) \to \tau_2^2[\eta^1]$, for given V_1^2 and given A_2^2 such that

$$\models_{\mathrm{v}} V_1^2 : \tau_1^2[\eta^1] \quad \text{and} \quad A_2\, V_1^2 \longrightarrow^* A_2^2,$$

we have to show that

$$\models_{\mathrm{v}} A_2^2 : \tau_2^2[\eta^2]$$

where $\eta^2 \overset{\mathrm{def}}{=} \eta^1 \cup \{x_1^2 \mapsto V_1^2\}$. Repeating the above, finally we reach a ground type: i.e., there is l such that

$$\begin{aligned}&\tau_2^i = \{f^{i+1} : (x_1^{i+1} \colon \tau_1^{i+1}) \to \tau_2^{i+1} \mid P^{i+1}\} \text{ for } i < l \text{ and} \\ &\tau_2^l = \{\nu_{l+1} : \mathbf{int} \mid P^{l+1}\};\end{aligned}$$

and for given V_1^i and A_2^i ($i \leq l$) such that

$$\models_{\mathrm{v}} V_1^i : \tau_1^i[\eta^{i-1}] \quad \text{and} \quad A_2^{i-1}\, V_1^i \longrightarrow^* A_2^i$$

where $\eta^i \overset{\mathrm{def}}{=} \eta^{i-1} \cup \{x_1^i \mapsto V_1^i\}$ for $i \leq l$, we need to show that

$$\models_{\mathrm{v}} A_2^l : \tau_2^l[\eta^l]. \tag{B.8}$$

Since

$$\mathbf{fix}(f, \lambda x_1.\, t_2[\eta])\, V_1\, V_1^2 \ldots V_1^l \longrightarrow^* A_2^l$$

and A_2^l is a ground answer, by the unwinding theorem, for some n,

$$\mathbf{fix}^{(n)}(f, \lambda x_1.\, t_2[\eta])\, V_1\, V_1^2 \ldots V_1^l \longrightarrow^* A_2^l,$$

and by (B.5), the goal (B.8) follows.

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

(i) for any η such that $\eta \models_{\mathrm{e}} \Gamma$ and any A such that $t[\eta] \longrightarrow^* A$, $\models_{\mathrm{v}} A : \{\nu : (x_1 \colon \tau_1) \to \tau_2 \mid P\}\,[\eta]$,

(ii) for any η such that $\eta \models_{\mathrm{e}} \Gamma$ and any A_1 such that $t_1[\eta] \longrightarrow^* A_1$, $\models_{\mathrm{v}} A_1 : \tau_1[\eta]$.

Then, for given η such that $\eta \models_{\mathrm{e}} \Gamma$ and given A_2 such that $(tt_1)[\eta] \longrightarrow^* A_2$, we show

$$\models_{\mathrm{v}} A_2 : \tau_2 (\!(x_1 \leftarrow t_1)\!) [\eta]. \tag{B.9}$$

Since $(tt_1)[\eta] = t[\eta]t_1[\eta] \longrightarrow^* A_2$, there exists A such that $t[\eta] \longrightarrow^* A$, and by (i),

$$\models_{\mathrm{v}} A : \{\nu : (x_1 \colon \tau_1) \to \tau_2 \mid P\} [\eta]. \tag{B.10}$$

Thus, A is a value and $A\,(t_1[\eta]) \longrightarrow^* A_2$. Hence, there exists A_1 such that

$$t_1[\eta] \longrightarrow^* A_1, \tag{B.11}$$

and by (ii),

$$\models_{\mathrm{v}} A_1 : \tau_1[\eta]. \tag{B.12}$$

Especially, A_1 is a value and

$$A\,A_1 \longrightarrow^* A_2. \tag{B.13}$$

Now, by (B.10), $\models_{\mathrm{v}} A : (x_1 \colon \tau_1[\eta]) \to \tau_2[\eta]$; and by (B.12) and (B.13),

$$\models_{\mathrm{v}} A_2 : \tau_2[\eta][x_1 \mapsto A_1].$$

Hence, by (B.11) and Lemma 8,

$$\models_{\mathrm{v}} A_2 : \tau_2[\eta] (\!(x_1 \leftarrow t_1[\eta])\!).$$

Since $\tau_2[\eta] (\!(x_1 \leftarrow t_1[\eta])\!) = \tau_2 (\!(x_1 \leftarrow t_1)\!) [\eta]$, we have shown (B.9).

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

$\boxed{\text{(T-FAIL)}}$ For given η and V such that

$$\eta \cup \{x \mapsto V\} \models_{\mathrm{e}} \Gamma,\, x : \{\nu : \sigma \mid \mathbf{false}\},$$

we show a contradiction instead of $\models \mathbf{fail} : \tau[\eta][x \mapsto V]$.

By the assumption, $\models_{\mathrm{v}} V : \{\nu : \sigma \mid \mathbf{false}\} [\eta]$ and hence $\models_{\mathrm{p}} \mathbf{false}$, which is a contradiction.

$\boxed{\text{(T-SUB)}}$ Trivial, from Lemma 7.

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

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

$\square$

Appendix C. Proof of Soundness of Verification by $(-)^\sharp$

Here we prove Theorem 1, the soundness of the verification by $(-)^\sharp$. We prove the soundness by dividing it into four parts corresponding to $(-)^{\sharp_1}$, $(-)^{\sharp_2}$, $(-)^{\sharp_3}$, and $(-)^{\sharp_4}$:

Proposition 10. *Let $i = 1$, 2, or 4. Let t be a closed term and τ be a type of order at most 2. Then,*

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

Proposition 11. *Let t be a closed term and τ be a type of order at most 2. Let T and ϕ be a multiplicity annotation and a multiplicity type for t and τ and suppose that they are consistent and $\tau \leq_{mul} \phi$. Then,*

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

The soundness theorem is an immediate corollary of the above since each transformation preserves the property that τ is order at most 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 uses multiplicity annotations, so we focus on this case, which is proved in Appendix C.3; see Appendix C.4 for (the key of) a proof of Proposition 10. As lemmas for proving Proposition 11, we show a substitution lemma and a simulation lemma for $(-)^{\sharp_3}$ (Lemmas 12 and 14) in Appendix C.1 and two lemmas for obtaining multiplicity annotations for values in argument position of logical relation (Lemmas 17 and 18) in Appendix C.2.

Below, when we write $t = \{\dots\}\, t'$, the expression "$\{\dots\}$" is an explanation for why the equation holds. We define an *observational preorder* (denoted by $\leq_{\mathrm{o}}$) as usual: $t \leq_{\mathrm{o}} t'$ if for any context C such that $C[t]$ and $C[t']$ are closed ground terms and for any A,

$$C[t] \longrightarrow^* A \text{ implies } C[t'] \longrightarrow^* A.$$

We also define an *observational equivalence* $=_{\mathrm{o}}$ as

$$t =_{\mathrm{o}} t' \overset{\mathrm{def}}{\iff} t \leq_{\mathrm{o}} t' \wedge t' \leq_{\mathrm{o}} t.$$

Appendix C.1. Substitution Lemma and Simulation Lemma for $(-)^{\sharp_3}$

For the substitution lemma, we define a restriction of a type annotation. For a term t, we write $\mathtt{S}(t)$ for the set of all the subterm occurrences of t. For a type annotation T for a term t and a subterm occurrence t' of t, since there is a natural embedding of $\mathtt{S}(t')$ into $\mathtt{S}(t)$, we can define $T|_{t'}$ as the restriction of the function T to $\mathtt{S}(t')$. If T is consistent for a multiplicity type ϕ, i.e., given a multiplicity type environment Φ and a derivation of $\Phi \vdash_{\mathrm{c}} t : \phi$ that induces T, its sub-derivation that corresponds to the subterm t' induces exactly the type annotation $T|_{t'}$; thus, the restriction of consistent type annotation is consistent in a canonical way. In the definition of $(t)^{\sharp_3}_T$ in Figure 7, restrictions of type annotations are used implicitly (e.g., $(t_1\, t_2)^{\sharp_3}_T \overset{\mathrm{def}}{=} \big(\mathbf{pr}_1 (t_1)^{\sharp_3}_{T|_{t_1}}\big)\, (t_2)^{\sharp_3}_{T|_{t_2}}$); below we make them explicit only in subtle cases.

Lemma 12 (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 $\Phi \vdash_{\mathrm{c}} t' : \phi'$ with the induced type annotations T and T' for t and t', respectively, we define a multiplicity annotation $T[T']$ of $t[x' \mapsto t']$ as that defined from the derivation of $\Phi \vdash_{\mathrm{c}} t[x' \mapsto t'] : \phi$. Then,

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

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

For given derivations below,

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

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

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

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

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

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

□

Toward the simulation lemma, we first show the following property:

Lemma 13 (Subject Reduction of Multiplicity Type System)**.** *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 $\Phi \vdash_{\mathrm{c}} t : \phi$; in the case $t = \mathbf{fix}(f, \lambda x_1.\, t_2)\, V_1$, we use the former part of Lemma 12. □

Using the above, we define a multiplicity annotation for a reduced term. For a multiplicity annotation T of a closed term t and a multiplicity type ϕ such that T and ϕ are consistent, suppose $t \longrightarrow t'$. By the definition of consistency, we have a derivation of $\vdash_{\mathrm{c}} t : \phi$, and by the subject reduction, we have the derivation of $\vdash_{\mathrm{c}} t' : \phi$. Thus we obtain a multiplicity annotation T' of t' that is consistent with ϕ; we write

$$T \longrightarrow_{t,\phi} T' \quad \text{or simply} \quad T \longrightarrow T'$$

to refer to this T'.

We also remark the following fact: since in the above proof we used Lemma 12 for the application case, it can also be shown that, for a consistent pair (T, ϕ) for $\mathbf{fix}(f, \lambda x_1.\, t_2)\, V_1$,

$$T \longrightarrow T|_{t_2}[T|_{V_1}][T|_{\mathbf{fix}(f,\lambda x_1.\, t_2)}]. \tag{C.1}$$

Now, we give a "coarse-grained" simulation lemma, which is a corollary of a "fine-grained" simulation lemma below (Lemma 16). We use the next lemma in the proof of Proposition 11; a busy reader can skip the rest of this subsection.

Lemma 14. *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 $(t)^{\sharp_3}_{T} \uparrow$.* □

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

Figure C.17: $(-)^{\sharp_3^{\mathrm{e}}}$: 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 C.18: Step numbers of evaluation contexts

For Lemma 16, we still prepare some definitions and a lemma. 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 C.17 and the *step numbers* $\mathrm{s}(E)$ in Figure C.18. Notice that, in both the definitions, we give special treatment to the case of $V E$.

Lemma 15. *(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.

$\mathbf{fix}(f, \lambda x_1.\, t_2)E$.

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

Lemma 16 (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*

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

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. Let $t = E[r]$ where r is a redex. By Lemma 15, we define $n \overset{\mathrm{def}}{=} \mathrm{s}(E) + 2$ if r is an application, $n \overset{\mathrm{def}}{=} \mathrm{s}(E) + 1$ otherwise, and $n' \overset{\mathrm{def}}{=} \mathrm{s}(E)$. Then, for each kind of redexes, proof goes straightforwardly: For the redex of application, we use Lemma 12. □

Appendix C.2. Consistent Multiplicity Annotations for Arbitrary Arguments

In our proof of Proposition 11, we come across a problem whether a consistent multiplicity annotation can be obtained in argument position of the logical relation $\models$. To solve this problem, we give another semantics $\models^{\mathrm{F}}$ of types that is equivalent to the original one $\models$, using finite canonical forms [5] (*FCFs*, for short). We restrict arguments of functions to FCFs in the semantics of $\models^{\mathrm{F}}$, and we can find a consistent multiplicity annotation for any FCF as shown in Lemma 18. For the sake of simplicity, we impose a restriction on the order of types in the following discussion; we can generalize the discussion to the arbitrary order in an obvious way.

The set of *FCFs of order at most 1* (or simply *FCFs*) and the set of *FCF-values* are defined by the following rules, respectively:

$$\begin{aligned}
N &::= \Omega \mid m \mid \mathbf{fail} \mid \lambda x.N \mid (N_1, \dots, N_n) \mid \big(\,\mathbf{case}\ \mathbf{pr}_i x\ \mathbf{of}\ m_1 \to N_1 \mid \cdots \mid m_n \to N_n\big) \\
W &::= m \mid \lambda x.N \mid (W_1, \dots, W_n)
\end{aligned}$$

where $\Omega \overset{\mathrm{def}}{=} \mathbf{fix}(f, \lambda x.\, f\, x)0$ and the **case** expressions are syntactic sugar for the iterated **if** expressions:

$$\mathbf{if}\ x = m_1\ \mathbf{then}\ N_1\ \mathbf{else}\ (\dots \mathbf{if}\ x = m_n\ \mathbf{then}\ N_n\ \mathbf{else}\ \Omega).$$

We assume that all FCF-values are closed, that all the (possibly non-free) variables occurring in FCFs and FCF-values (except for Ω) are order-0, and that all the closed FCFs and FCF-values are simply typed with types of order at most 1. We remark two differences between the definition of FCFs in the work [5] and ours: The definition in ibid. contains the case of let-expression $\mathbf{let}\ x = y\, N\ \mathbf{in}\ N'$, which does, however, not happen in our setting since y must be order-0. Second, our language has product types (and **fail**), while that in ibid. does not have them.

We define the alternative semantics of types ($\models^{\mathrm{F}}$, $\models^{\mathrm{F}}_{\mathrm{v}}$, …) for types of at most order-2 in the same way as Figure 3 except for the function type case, which is defined as follows:

$$\models^{\mathrm{F}}_{\mathrm{v}} V : (x_1 : \tau_1) \to \tau_2 \quad \overset{\mathrm{def}}{\iff} \quad \text{for any FCF-value } W_1,\ \models^{\mathrm{F}}_{\mathrm{v}} W_1 : \tau_1 \text{ implies } \models^{\mathrm{F}} V W_1 : \tau_2[x_1 \mapsto W_1]$$

By a standard argument on domain theory, we have the following lemma; after this lemma, we do not distinguish between $\models$ and $\models^{\mathrm{F}}$ for types of order at most 2.

Lemma 17. *For any term t and type τ of order at most 2,*

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

Proof. The proof is by induction on the order of τ. The order-0 and order-1 cases are trivial. In the order-2 case, the "only if" direction is clear by the induction hypothesis. For the "if" direction, we prove its contraposition: we assume $\not\models t : \tau$ and prove $\not\models^{\mathrm{F}} t : \tau$. Let $\tau'_0 = \tau$ and

$$\begin{aligned}
\tau'_i &= \{\nu_{i+1} : (x_{i+1}{:}\tau_{i+1}) \to \tau'_{i+1} \mid P_{i+1}\} \quad (0 \le i < n)\\
\tau'_n &= \{\nu_{n+1} : \mathbf{int} \mid P_{n+1}\}.
\end{aligned}$$

The assumption $\not\models t : \tau$ is equivalent to the following condition:

$$\begin{aligned}
&\exists p \in \{0, \dots, n\}.\\
&\exists! V'_1.\ t \longrightarrow^* V'_1 \ \wedge\ \exists V_1.\ \models_{\mathrm{v}} V_1 : \tau_1 \wedge\\
&\exists! V'_2.\ V'_1 V_1 \longrightarrow^* V'_2 \ \wedge\ \exists V_2.\ \models_{\mathrm{v}} V_2 : \tau_2 \wedge\\
&\qquad\qquad\qquad \cdots\\
&\exists! V'_p.\ V'_{p-1} V_{p-1} \longrightarrow^* V'_p \ \wedge\ \exists V_p.\ \models_{\mathrm{v}} V_p : \tau_p \wedge\\
&\exists! A'_{p+1}.\ V'_p V_p \longrightarrow^* A'_{p+1} \wedge\\
&\big(A'_{p+1} = \mathit{fail}\ \vee (A'_{p+1}\text{: value} \wedge \exists! A' \neq \mathbf{true}.\ P_{p+1}[\nu_{p+1} \mapsto A'_{p+1}] \longrightarrow^* A')\big)
\end{aligned} \tag{C.2}$$

In the case $p = 0$ above, read the above formula as $\exists! A'_1.\ t \longrightarrow^* A'_1 \ \wedge (A'_1 = \mathit{fail}\ \vee (\dots))$. We give a proof for the case that A'_{p+1}: value $\wedge \dots$; the proof for the case that $A'_{p+1} = \mathit{fail}$ is almost the same: Just replace the term s defined below with t. The goal $\not\models^{\mathrm{F}} t : \tau$ is equivalent to the following condition:

$$\begin{aligned}
&\exists p \in \{0, \dots, n\}.\\
&\exists! V'_1.\ t \longrightarrow^* V'_1 \ \wedge\ \exists W_1.\ \models^{\mathrm{F}}_{\mathrm{v}} W_1 : \tau_1 \wedge\\
&\exists! V'_2.\ V'_1 W_1 \longrightarrow^* V'_2 \ \wedge\ \exists W_2.\ \models^{\mathrm{F}}_{\mathrm{v}} W_2 : \tau_2 \wedge\\
&\qquad\qquad\qquad \cdots\\
&\exists! V'_p.\ V'_{p-1} W_{p-1} \longrightarrow^* V'_p \ \wedge\ \exists W_p.\ \models^{\mathrm{F}}_{\mathrm{v}} W_p : \tau_p \wedge\\
&\exists! A'_{p+1}.\ V'_p W_p \longrightarrow^* A'_{p+1} \wedge\\
&\big(A'_{p+1} = \mathit{fail}\ \vee (A'_{p+1}\text{: value} \wedge \exists! A' \neq \mathbf{true}.\ P_{p+1}[\nu_{p+1} \mapsto A'_{p+1}] \longrightarrow^* A')\big)
\end{aligned} \tag{C.3}$$

We define a term

$$s \ \stackrel{\text{def}}{=}\ \lambda y_1, \dots, y_p.\ \mathbf{let}\ \nu_{p+1} = t y_1 \dots y_p\ \mathbf{in}\ P_{p+1}$$

and then, by (C.2),

$$s\, V_1 \dots V_p \longrightarrow^* A'.$$

Let us consider the interpretation in the call-by-value game model given in the work [5]:

$$[\![s\, V_1 \dots V_p]\!] \ = \ [\![A']\!].$$

It is proved in ibid. that any element in the game model is the supremum of some ω-chain of FCFs. Hence, for each i, we have an ω-chain $(N_i^{j_i})_{j_i}$ of FCFs such that $[\![V_i]\!] = \sup_{j_i} [\![N_i^{j_i}]\!]$. Since if $[\![W]\!] \le [\![N]\!]$ for some FCF-value W then N must be an FCF-value, and since there is some j_i such that $N_i^{j_i}$ is an FCF-value, we can choose a subchain $(W_i^{j_i})_{j_i}$ of $(N_i^{j_i})_{j_i}$ with

$$[\![V_i]\!] \ = \ \sup_{j_i} [\![W_i^{j_i}]\!].$$

By the continuity of $[\![s]\!]$,

$$[\![s\, V_1 \dots V_p]\!] \ = \ [\![s]\!]\,(\sup_{j_1} [\![W_1^{j_1}]\!]) \dots (\sup_{j_p} [\![W_p^{j_p}]\!]) \ = \ \sup_{j_1} \dots \sup_{j_p} [\![s]\!]\,[\![W_1^{j_1}]\!] \dots [\![W_p^{j_p}]\!] \ = \ [\![A']\!].$$

As A' is **false** or *fail*, $[\![A']\!]$ is a compact element; recall that an element d in a cpo D is *compact* if for any ω-chain $(d_i)_i$ in D such that $d \leq \sup_i d_i$ there exists i such that $d \leq d_i$ (and hence if $d = \sup_i d_i$ then $d = d_i$ for some i). Therefore, there are $j_1, \ldots, j_p$ such that

$$([\![s\, W_1^{j_1} \ldots W_p^{j_p}]\!] \;\; =) \;\; [\![s]\!]\, [\![W_1^{j_1}]\!] \ldots [\![W_p^{j_p}]\!] \;\; = \;\; [\![A']\!].$$

By the adequacy of the call-by-value game model, we have

$$s\, W_1^{j_1} \ldots W_p^{j_p} \longrightarrow^* A'. \tag{C.4}$$

For all i, since $[\![W_i^{j_i}]\!] \leq [\![V_i]\!]$,

$$W_i^{j_i} \leq_{\mathrm{o}} V_i.$$

By induction hypothesis, we have

$$\models^{\mathrm{F}}_{\mathrm{v}} V_i : \tau_i.$$

It is easy to show that, for any values V, V' and type τ,

$$V' \leq_{\mathrm{o}} V \quad \text{and} \quad \models^{\mathrm{F}}_{\mathrm{v}} V : \tau \quad \text{imply} \quad \models^{\mathrm{F}}_{\mathrm{v}} V' : \tau.$$

Hence, for all i,

$$\models^{\mathrm{F}}_{\mathrm{v}} W_i^{j_i} : \tau_i,$$

and by this and (C.4), we can show (C.3) with $W_i^{j_i}$ as witnesses of $\exists W_i$. □

The next lemma solves the problem whether we can obtain consistent multiplicity annotations in argument position. We define the following:

$$\begin{aligned}
\mathtt{ST}(\{\textstyle\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\phi_j \to \phi'_j) \mid M\}) &\stackrel{\mathrm{def}}{=} \textstyle\prod_{i=1}^n \mathbf{int} \times \prod_{j=1}^m (\mathtt{ST}(\phi_j) \to \mathtt{ST}(\phi'_j)) \\
\mathtt{ST}(x_1\colon \phi_1, \ldots, x_n\colon \phi_n) &\stackrel{\mathrm{def}}{=} x_1\colon \mathtt{ST}(\phi_1), \ldots, x_n\colon \mathtt{ST}(\phi_n) \\
\mathit{order}(x_1\colon \phi_1, \ldots, x_n\colon \phi_n; \phi) &\stackrel{\mathrm{def}}{=} \mathit{order}(\mathtt{ST}(\phi_1) \to \ldots \to \mathtt{ST}(\phi_n) \to \mathtt{ST}(\phi))
\end{aligned}$$

Lemma 18. *For a closed FCF N and a multiplicity type ϕ such that N is typed with $\mathtt{ST}(\phi)$ and $\mathtt{ST}(\phi)$ is order at most 1, there is a derivation of $\vdash_{\mathrm{c}} N : \phi$.*

Proof. We can straightforwardly prove the following more general statement by induction on N: for a (possibly open) FCF N, a multiplicity type ϕ, and a multiplicity type environment Φ such that

$$\mathtt{ST}(\Phi) \vdash_{\mathrm{S}} N : \mathtt{ST}(\phi), \qquad \mathit{order}(\Phi; \phi) \leq 1,$$

there is a derivation of $\Phi \vdash_{\mathrm{c}} N : \phi$. □

Appendix C.3. Proof of Proposition 11

Now we prove Proposition 11.

Proof of Proposition 11. We prove that

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

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 20 given later.

$\boxed{\tau = \{f : (x_1\colon \tau_1) \to \tau_2 \mid P\}}$ Let ϕ be of the form $\{\phi_1 \to \phi_2 \mid M\}$, and suppose $\models (t)^{\sharp_3}_{T} : (\tau)^{\sharp_3}_{\phi}$, i.e., for any A' such that $(t)^{\sharp_3}_{T} \longrightarrow^* A'$,

$$\models_{\mathrm{v}} A' : (\{f : (x_1\colon \tau_1) \to \tau_2 \mid P\})^{\sharp_3}_{\phi}. \tag{C.5}$$

Given $t \longrightarrow^n A$ ($n \in \mathbb{N}$), we show that

$$\models_{\mathrm{v}} A : (x_1 \colon \tau_1) \to \tau_2 \tag{C.6}$$
$$\models_{\mathrm{p}} P[f \mapsto A]. \tag{C.7}$$

By Lemma 14 and since $\models (t)^{\sharp_3}_T : (\tau)^{\sharp_3}_\phi$, A must be a value. Let $A = \mathbf{fix}(f, \lambda x_1.\, t_2)$, then

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

for T' such that $T \longrightarrow^n T'$ and $m \overset{\mathrm{def}}{=} T'(A)$. By (C.5), we have

$$\models_{\mathrm{v}} \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])}^m : \Big(\big(x_1 \colon (\tau_1)^{\sharp_3}_{\phi_1}\big) \to (\tau_2)^{\sharp_3}_{\phi_2} \Big)^m$$
$$\models_{\mathrm{p}} P[f\, t_l \mapsto f_l\, t_l]_{l \le m'}[f_l \mapsto \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])]_{l \le m'}$$

where $f\, t_1, \ldots, f\, t_{m'}$ are all the occurrences of applications of f. The two judgments above imply, respectively,

$$\models_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m]) : \big(x_1 \colon (\tau_1)^{\sharp_3}_{\phi_1}\big) \to (\tau_2)^{\sharp_3}_{\phi_2} \tag{C.8}$$
$$\models_{\mathrm{p}} P[f \mapsto \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])]. \tag{C.9}$$

Now we prove (C.6) by using Lemma 17, i.e., for given FCF-value W_1 such that $\models_{\mathrm{v}} W_1 : \tau_1$, we show

$$\models \mathbf{fix}(f, \lambda x_1.\, t_2) W_1 : \tau_2[x_1 \mapsto W_1]\,. \tag{C.10}$$

By Lemma 18 and the correspondence between derivations of multiplicity type judgments and consistent type annotations, we have a multiplicity annotation T'_1 for W_1 that is consistent with ϕ_1. Since $\models_{\mathrm{v}} W_1 : \tau_1$, by Lemma 19,

$$\models (W_1)^{\sharp_3}_{T'_1} : (\tau_1)^{\sharp_3}_{\phi_1}\,. \tag{C.11}$$

From the corresponding derivations of (T', ϕ) and (T'_1, ϕ_1) for $\mathbf{fix}(f, \lambda x_1.\, t_2)$ and W_1, respectively, we obtain a consistent pair (T'_2, ϕ_2) for $\mathbf{fix}(f, \lambda x_1.\, t_2) W_1$ by Rule (C-APP) in Figure 9. Then,

$$\begin{aligned}
&(\mathbf{fix}(f, \lambda x_1.\, t_2) W_1)^{\sharp_3}_{T'_2} \\
&= (\mathbf{pr}_1\, (\mathbf{fix}(f, \lambda x_1.\, t_2))^{\sharp_3}_{T'})\, (W_1)^{\sharp_3}_{T'_1} \\
&= (\mathbf{pr}_1 \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])}^m)\, (W_1)^{\sharp_3}_{T'_1} \\
&\longrightarrow \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])\, (W_1)^{\sharp_3}_{T'_1}\,.
\end{aligned}$$

Hence, by (C.8) and (C.11),

$$\models (\mathbf{fix}(f, \lambda x_1.\, t_2) W_1)^{\sharp_3}_{T'_2} : (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto (W_1)^{\sharp_3}_{T'_1}]\,.$$

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, $(W_1)^{\sharp_3}_{T'_1} = W_1$ by Lemma 20, and

$$(\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto (W_1)^{\sharp_3}_{T'_1}] = (\tau_2)^{\sharp_3}_{\phi_2}\, [x_1 \mapsto W_1] = (\tau_2[x_1 \mapsto W_1])^{\sharp_3}_{\phi_2}\,,$$

where the latter equation is obvious from the definition of $(-)^{\sharp_3}$ as W_1 is a ground value. Hence, in any case on τ_1,

$$\models (\mathbf{fix}(f, \lambda x_1.\, t_2) W_1)^{\sharp_3}_{T'_2} : (\tau_2[x_1 \mapsto W_1])^{\sharp_3}_{\phi_2}\,.$$

By induction hypothesis, we have obtained (C.10).

Next, we prove (C.7), i.e.,

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

If f occurs in P, f has a depth-1 type; hence, by (C.9), 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. In the above equation, precisely, the right hand side is

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

if we make it explicit the restrictions of type annotations in the definition of $(-)^{\sharp_3}$. Now since V_1 is order-0, we have a derivation of $\vdash_{\mathrm{c}} V_1 : \phi_1$, which induces a consistent pair (T'_1, ϕ_1) for V_1; also, by Rule (C-App), we have a consistent pair (T'_2, ϕ_2) for $\mathbf{fix}(f, \lambda x_1.\, t_2)V_1$ such that $T'_2|_{\mathbf{fix}(f,\lambda x_1.\, t_2)} = T'$ and $T'_2|_{V_1} = T'_1$. Then,

$$\begin{aligned}
&\mathbf{fix}(f, \lambda x_1.\, t_2)V_1\\
=_{\mathrm{o}}\ & t_2[x_1 \mapsto V_1][f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)]\\
=_{\mathrm{o}}\ & \{\text{by Lemma 20}\}\\
& (t_2[x_1 \mapsto V_1][f \mapsto \mathbf{fix}(f, \lambda x_1.\, t_2)])^{\sharp_3}_{T'|_{t_2}[T'_1][T']}\\
=_{\mathrm{o}}\ & \{\text{by Lemma 12}\}\\
& (t_2)^{\sharp_3}_{T'|_{t_2}}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}][f \mapsto (\mathbf{fix}(f, \lambda x_1.\, t_2))^{\sharp_3}_{T'}]\\
=_{\mathrm{o}}\ & (t_2)^{\sharp_3}_{T'|_{t_2}}\, [x_1 \mapsto (V_1)^{\sharp_3}_{T'_1}][f \mapsto \overrightarrow{\mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'}\, [f \mapsto \overrightarrow{f}^m])}^m\ ]\\
=_{\mathrm{o}}\ & \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'|_{t_2}}\, [f \mapsto \overrightarrow{f}^m])\, (V_1)^{\sharp_3}_{T'_1}\\
=_{\mathrm{o}}\ & \mathbf{fix}(f, \lambda x_1.\, (t_2)^{\sharp_3}_{T'|_{t_2}}\, [f \mapsto \overrightarrow{f}^m])V_1\,.
\end{aligned}$$

□

Lemma 19. *Let t be a closed term and τ be a type of order at most 1. Let T and ϕ be a multiplicity annotation and a multiplicity type for t and τ and suppose that they are consistent and $\tau \leq_{mul} \phi$. Then,*

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

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 20.

$\boxed{\tau = \{f : (x : \tau_1) \to \tau_2 \mid P\}}$ Let ϕ be of the form $\{\phi_1 \to \phi_2 \mid M\}$. By the assumption $\models t : \tau$, for any A such that $t \longrightarrow^* A$,

$$\models_{\mathrm{v}} A : (x_1 \colon \tau_1) \to \tau_2 \tag{C.12}$$
$$\models_{\mathrm{p}} P[f \mapsto A]. \tag{C.13}$$

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

By Lemma 14 and (C.12) and since $(t)^{\sharp_3}_{T} \longrightarrow^* A'$, there is some value $V = \mathbf{fix}(f, \lambda x_1.\, t_2)$ and n such that $t \longrightarrow^n V$ and

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

Let $T \longrightarrow^n T'$ and $m = T'(V)$. Hence, similarly to the previous proof, it is enough to show

$$\models_{\mathrm{v}} \mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'}\,[f \mapsto \overrightarrow{f}^{m}]) : \big(x_1\colon (\tau_1)^{\sharp_3}_{\phi}\big) \to (\tau_2)^{\sharp_3}_{\phi} \tag{C.14}$$

$$\models_{\mathrm{p}} P[f \mapsto \mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'}\,[f \mapsto \overrightarrow{f}^{m}])]. \tag{C.15}$$

Now we prove (C.14), i.e., for given V_1' such that $\models_{\mathrm{v}} V_1' : (\tau_1)^{\sharp_3}_{\phi_1}$, we show

$$\models \mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'|_{t_2}}\,[f \mapsto \overrightarrow{f}^{m}])V_1' : (\tau_2)^{\sharp_3}_{\phi_2}\,[x_1 \mapsto V_1']\,. \tag{C.16}$$

where we have made it explicit the restrictions of type annotations in the definition of $(-)^{\sharp_3}$. 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 20, $V_1' = (V_1')^{\sharp_3}_{T_1'}$ and $\models_{\mathrm{v}} V_1' : \tau_1$. Hence,

$$\begin{aligned}
&\mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'|_{t_2}}\,[f \mapsto \overrightarrow{f}^{m}])V_1' \\
\longrightarrow\ & (t_2)^{\sharp_3}_{T'|_{t_2}}\,[x_1 \mapsto V_1'][\overrightarrow{f \mapsto \mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'|_{t_2}}\,[f \mapsto \overrightarrow{f}^{m}])\ }^{m}] \\
=\ & (t_2)^{\sharp_3}_{T'|_{t_2}}\,[x_1 \mapsto (V_1')^{\sharp_3}_{T_1'}][f \mapsto (V)^{\sharp_3}_{T'}] \\
=\ & \{\text{by Lemma 12}\} \\
& (t_2[x_1 \mapsto V_1'][f \mapsto V])^{\sharp_3}_{T'|_{t_2}[T_1'][T']}\,.
\end{aligned}$$

By the result (C.1) remarked after Lemma 13, we have $T_2' \longrightarrow T'|_{t_2}[T_1'][T']$; hence, by Lemma 14,

$$(VV_1')^{\sharp_3}_{T_2'} \longrightarrow^* (t_2[x_1 \mapsto V_1'][f \mapsto V])^{\sharp_3}_{T'|_{t_2}[T_1'][T']}\,.$$

Hence,

$$\mathbf{fix}(f, \lambda x_1.\,(t_2)^{\sharp_3}_{T'|_{t_2}}\,[f \mapsto \overrightarrow{f}^{m}])V_1' \quad =_{\mathrm{o}} \quad (VV_1')^{\sharp_3}_{T_2'}\,. \tag{C.17}$$

Now from (C.12), $\models VV_1' : \tau_2[x_1 \mapsto V_1']$, and by induction hypothesis,

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

and hence, by (C.17), we have shown (C.16).

Finally, (C.15) is shown from (C.13) quite similarly to the proof of Proposition 11. □

Lemma 20. *Let t be a closed term and τ be a type of order 0. Let T and ϕ be a multiplicity annotation and a multiplicity type for t and τ and suppose that they are consistent and $\tau \leq_{mul} \phi$. Then,*

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

Proof. On types, it is clear by definition. On terms, it is clear from Lemma 14. □

Appendix C.4. Proof of Proposition 10

Proposition 10 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 $(-)^{\sharp_4}$,

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

is transformed to:

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

By the semantics of types, the former is equivalent to

$$\text{for all } i,\ x_i, \text{ and } A, \text{ if } f_i x_i \longrightarrow^* A, \text{ then } A \text{ is value, and} \tag{C.18}$$

$$\text{for all } x_1 \text{ and } x_2,\ \models_{\mathrm{p}} P[f_1 x_1, f_2 x_2], \tag{C.19}$$

while the latter is equivalent to

$$\text{for all } x_1,\ x_2, \text{ and } A, \text{ if } (f_1 x_1, f_2 x_2) \longrightarrow^* A, \text{ then } A \text{ is value } (V_1, V_2) \text{ and } \models_{\mathrm{p}} P[V_1, V_2], \tag{C.20}$$

$$\text{for all } x_1 \text{ and } A, \text{ if } f_1 x_1 \longrightarrow^* A, \text{ then } A \text{ is value } V \text{ and } \models_{\mathrm{p}} P[V, \bot], \tag{C.21}$$

$$\text{for all } x_2 \text{ and } A, \text{ if } f_2 x_2 \longrightarrow^* A, \text{ then } A \text{ is value } V \text{ and } \models_{\mathrm{p}} P[\bot, V], \text{ and} \tag{C.22}$$

$$\models_{\mathrm{p}} P[\bot, \bot]. \tag{C.23}$$

Here (C.21), (C.22), and (C.23) hold 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 (C.18) are obvious. We show (C.19) from the latter.

First note that, since $f_1 x_1$ and $f_2 x_2$ are the only application occurrences in $P[f_1 x_1, f_2 x_2]$ because of $(((-)^{\sharp_1})^{\sharp_2})^{\sharp_3}$, and by the assumption in Section 2.3 that a predicate does not contain **fail** nor **fix**, when we evaluate $P[f_1 x_1, f_2 x_2]$, an effect may happen only due to $f_1 x_1$ or $f_2 x_2$. By (C.21) and (C.22), for any x_1 and x_2, $f_1 x_1$ or $f_2 x_2$ never fail. If both $f_1 x_1$ and $f_2 x_2$ evaluate to values, by (C.20), we have $\models_{\mathrm{p}} P[f_1 x_1, f_2 x_2]$.

If $f_1 x_1$ or $f_2 x_2$ diverges, $P[f_1 x_1, f_2 x_2]$ also diverges as follows. By the assumption in Section 2.3 that we use "branch-strict if" in refinement predicates, during the evaluation of $P[f_1 x_1, f_2 x_2]$, every occurrence of application must be evaluated unless some effect happens. Concretely, if both $f_1 x_1$ and $f_2 x_2$ diverge, $P[f_1 x_1, f_2 x_2]$ diverges due to one of $f_1 x_1$ and $f_2 x_2$ that is evaluated earlier than the other; if $f_1 x_1$ diverges and $f_2 x_2$ terminates to a value, $P[f_1 x_1, f_2 x_2]$ diverges due to $f_1 x_1$; and if $f_2 x_2$ diverges and $f_1 x_1$ terminates to a value, $P[f_1 x_1, f_2 x_2]$ diverges due to $f_2 x_2$.

Appendix D. Proof of Soundness of Verification by $(-)^{\sharp'}$

Here we prove Theorem 4, i.e., the soundness of verification by $(-)^{\sharp'}$.

First we remark that the difference between $(-)^{\sharp'}$ and $(-)^{\sharp}$ is just the assume-expressions inserted by $\mathtt{InstVar}(-)$. For any terms t, t',

$$\mathbf{assume}\,(t)\,; t' \quad \leq_{\mathrm{o}} \quad t'$$

since $\mathbf{assume}\,(t)$ may only evaluate to $\mathtt{true}$ or diverge; recall that $\mathbf{assume}\,(\mathbf{fail})$ diverges. Therefore, for any term t, $(t)^{\sharp'} \leq_{\mathrm{o}} (t)^{\sharp}$, so we have:

$$\models (t)^{\sharp}_{T} : (\tau)^{\sharp}_{\phi} \quad \text{implies} \quad \models (t)^{\sharp'}_{T} : (\tau)^{\sharp}_{\phi}\,. \tag{D.1}$$

Proof of Theorem 4. We reduce Theorem 4 to Lemma 21 given below; 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 $\models_{\mathrm{v}} V_1 : \tau_1$, we prove $\models t\, V_1 : \mathbf{int}$.

By Lemma 19, we have $\models_{\mathrm{v}} (V_1)^{\sharp} : (\tau_1)^{\sharp}$, and hence $\models_{\mathrm{v}} (V_1)^{\sharp'} : (\tau_1)^{\sharp}$ by (D.1). By the assumption that $\models (t)^{\sharp'}_{T} : (\tau)^{\sharp}_{\phi}$, we have $\models_{\mathrm{v}} (t)^{\sharp'}\,(V_1)^{\sharp'} : (\mathbf{int})^{\sharp}$. 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, $\models (t\, V_1)^{\sharp'} : (\mathbf{int})^{\sharp}$. Since $\mathbf{int}$ is order-0, by Lemma 21 below, we have $\models (t\, V_1)^{\sharp} : (\mathbf{int})^{\sharp}$, and by Lemma 20, $\models t\, V_1 : \mathbf{int}$. □

Lemma 21. *For any closed A-normal form t (defined in Figure 10), a type τ of order-0, and a consistent pair of a multiplicity annotation T of t and a multiplicity type ϕ such that $\tau \leq_{mul} \phi$, $(t)^{\sharp'}_{T}$ 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 (Overview). Here we give an overview of our proof and an example to explain our intuitive idea; we give a formal proof after this overview. In the rest of this section, by "A-normal forms" we mean those defined in Figure 12 (rather than Figure 10).

First, we define $(-)^{\sharp_{34}}_{T}$ by eliminating `InstVar` from $(-)^{\sharp'_{34}}_{T}$; i.e., in Figure 13, we drop the subscript B and replace the case of application with the following

$$(f(x_1, \dots, x_n, g_1, \dots, g_m))^{\sharp_{34}}_{T} \overset{\text{def}}{=} \mathbf{pr}_1(f(\overrightarrow{z}^{T(f)}))$$
$$\text{where } z \overset{\text{def}}{=} (x_1, \dots, x_n, \lambda \widetilde{y_j}.\, (g_1\, y_1, ..., g_m\, y_m))\ .$$

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 the assume expressions inserted by $\texttt{InstVar}(-)$ are satisfied and hence can be removed without changing the meaning.

As explained in the definition of $\texttt{InstVar}(-)$, the assume expressions inserted by $\texttt{InstVar}(-)$ are properties satisfied naturally by the image of $(-)^{\sharp_{34}}$. These properties are true for the images of subterms of t or for variables that will be instantiated with the images of such subterms, but not necessarily true for arbitrary variables. Now since t is a ground closed term, variables should be instantiated with such images; to realize such an instantiation, we "evaluate" the two programs $(t)^{\sharp'}_{T}$ and $(t)^{\sharp'_{34}}_{T}$. Also, though it is easy to prove that $(V)^{\sharp'_{34}}$ satisfies the properties by unfolding the definition of $(-)^{\sharp'_{34}}$ in Figure 13, it is not obvious if such V is a non-value e. The above "evaluation" transforms $(e)^{\sharp'_{34}}$ to the form $(V)^{\sharp'_{34}}$, and allows us to consider only such value cases. We call this "evaluation" *N-reduction*; it is defined similarly to evaluation, but keeps the form of the A-normal forms.

In order for N-reduction to terminate, we can assume that the given whole (ground closed) term t terminates, because if t diverges, by Lemma 14 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 N-reduce $(t)^{\sharp'_{34}}$, in fact we define N-reduction for t, and we show that

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

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

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

because in N-normal form we have no application and $\texttt{InstVar}(-)$ happens only in applications. In this way, we can show that

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

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

$$\begin{aligned}&\mathbf{let}\ f = \mathbf{fix}(f', \lambda x'.\, t')\ \mathbf{in}\\ &\mathbf{let}\ x = 3\ \mathbf{in}\\ &\mathbf{let}\ y = f x\ \mathbf{in}\ t''\end{aligned} \tag{D.2}$$

where let $T(f) = 2$.

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

$$\begin{aligned}
&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\,((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\
&\mathbf{let}\ x = 3\ \mathbf{in}\\
&\mathbf{let}\ y =\\
&\quad \mathbf{let}\ f'' = \lambda(y_1, y_2).\,\big(\mathbf{let}\ w = f(y_1, y_2)\ \mathbf{in}\ \mathbf{assume}\,(\ldots)\,;w\big)\\
&\quad \mathbf{in}\ \mathbf{pr}_1(f''(x, x))\\
&\mathbf{in}\ (t'')^{\sharp'_{34}}\ .
\end{aligned} \tag{D.3}$$

Since f in t is bound to the value $\mathbf{fix}(f', \lambda x'.\,t')$, we could calculate by the definition in Figure 13 that (the body of) f in $(t)^{\sharp'_{34}}$ is syntactically the product

$$\lambda(x'_1, x'_2).\,((t')^{\sharp'_{34}}\,[x' \mapsto x'_1], (t')^{\sharp'_{34}}\,[x' \mapsto x'_2])$$

of the duplication of $\lambda x'.\,(t')^{\sharp'_{34}}$ (it is not the case if f in t is bound to a non-value). Then, it is easy to show that such syntactical product of the duplication satisfies the predicates in $\mathbf{assume}\,(\ldots)$ above (as shown in the last of the proof of Lemma 27); here recall that, the predicates 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

$$\begin{aligned}
&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\,((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\
&\mathbf{let}\ x = 3\ \mathbf{in}\\
&\mathbf{let}\ y = \mathbf{pr}_1(f(x, x))\ \mathbf{in}\ (t'')^{\sharp'_{34}}\ .
\end{aligned} \tag{D.4}$$

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 $\mathbf{assume}\,(\ldots)$ as above.

The above is observationally equivalent to

$$\begin{aligned}
&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\,((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\
&\mathbf{let}\ x = 3\ \mathbf{in}\\
&\mathbf{let}\ y = \mathbf{pr}_1(((t')^{\sharp'_{34}}\,[x' \mapsto x][f' \mapsto f])_{i=1,2})\ \mathbf{in}\ (t'')^{\sharp'_{34}}\ .
\end{aligned}$$

Since our language is deterministic, $\mathbf{pr}_1(t, t) =_{\mathrm{o}} t$ for any term t (Lemma 25 - (1)); hence the above term is equivalent to

$$\begin{aligned}
&\mathbf{let}\ f = \mathbf{fix}(f', \lambda(x'_1, x'_2).\,((t')^{\sharp'_{34}}\,[x' \mapsto x'_i])_{i=1,2})\ \mathbf{in}\\
&\mathbf{let}\ x = 3\ \mathbf{in}\\
&\mathbf{let}\ y = (t')^{\sharp'_{34}}\,[x' \mapsto x][f' \mapsto f]\ \mathbf{in}\ (t'')^{\sharp'_{34}}\ .
\end{aligned} \tag{D.5}$$

We define N-reduction so that it reduces (D.2) to the following

$$\begin{aligned}
&\mathbf{let}\ f = \mathbf{fix}(f', \lambda x'.\,t')\ \mathbf{in}\\
&\mathbf{let}\ x = 3\ \mathbf{in}\\
&\mathbf{let}\ y = t'[x' \mapsto x][f' \mapsto f]\ \mathbf{in}\ t''\ .
\end{aligned} \tag{D.6}$$

It is clear that $(-)^{\sharp'_{34}}$ of (D.6) becomes (D.5); thus, $(-)^{\sharp'_{34}}$ preserves N-reduction to observational equivalence.

As the above N-reduction from (D.2) to (D.6), N-reduction is just evaluation except that it keeps the form of A-normal form. Hence, repeating this N-reduction, fx in (D.2) becomes some value V; therefore, $\mathbf{pr}_1(f(x, x))$ in (D.4) turns out to be observationally equivalent to $(V)^{\sharp'_{34}}$.

Repeating N-reduction, we finally obtain its normal form, which is of the following form

$$\mathbf{let}\ x_1 = V_1\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = V_n\ \mathbf{in}\ x_i\ . \tag{D.7}$$

$$s ::= (x_1, \ldots, x_n) \mid \mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2} \mid \mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2}$$
$$d ::= n \mid \mathsf{op}(x_1, \ldots, x_n) \mid \mathbf{fix}(f, \lambda(x_1, \ldots, x_n).\, s^b) \mid f\,(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \mid \mathbf{pr}_i x \mid \mathbf{fail}$$

Figure D.19: Labeled A-normal forms

In the **fail**-case,
$$(t)^{\sharp_{34}'} \ =_{\mathrm{o}}\ (\mathbf{fail})^{\sharp_{34}'} \ =\ \mathbf{fail}\ =\ (\mathbf{fail})^{\sharp_{34}}\ =_{\mathrm{o}}\ (t)^{\sharp_{34}}.$$

Otherwise, by applying $(-)^{\sharp_{34}'}$ to (D.7), we have

$$\mathbf{let}\ x_1 = (V_1)^{\sharp_{34}'}\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = (V_n)^{\sharp_{34}'}\ \mathbf{in}\ x_i \tag{D.8}$$

and by applying $(-)^{\sharp_{34}}$ to (D.7), we have

$$\mathbf{let}\ x_1 = (V_1)^{\sharp_{34}}\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = (V_n)^{\sharp_{34}}\ \mathbf{in}\ x_i\ . \tag{D.9}$$

Since $(V_j)^{\sharp_{34}'}$ and $(V_j)^{\sharp_{34}}$ $(j = 1, \ldots, n)$ are values, by β-conversion, (D.8) and (D.9) are observationally equivalent to $(V_i)^{\sharp_{34}'}$ and $(V_i)^{\sharp_{34}}$, respectively. Since now x_i has a ground type, so does V_i; hence $(V_i)^{\sharp_{34}'} = (V_i)^{\sharp_{34}} = V_i$. Therefore, (D.8) and (D.9) are observationally equivalent, and so are $(t)^{\sharp_{34}'}$ and $(t)^{\sharp_{34}}$. □

The rest of this section is just a formalization of the proof sketch above.

Appendix D.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 $(-)^{\sharp_{34}'}_{T,B}$.

We define *labeled A-normal forms* in Figure D.19, 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 in the classes defined in Figure 12, respectively. We implicitly use this label-dropping transformation to derive notions for s from those for t.

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 prepare one relation: for R, x, and U^b, we write $x \rightsquigarrow_R U^b$ if x refers to U^b in R; precisely, $x \rightsquigarrow_R U^b$ is defined as below.

$$x \rightsquigarrow_{[\,]} U^b \overset{\text{def}}{\iff} \text{false}$$
$$x \rightsquigarrow_{\mathbf{let}\ x' = U'^{b_1'}\ \mathbf{in}\ R^{b_2'}} U^b \overset{\text{def}}{\iff} x \rightsquigarrow_R U^b \text{ or } \left(x = x', U^b = U'^{b_1'}, \text{ and } x \not\rightsquigarrow_R U''^{b''} \text{ for any } U''^{b''}\right)$$

Note that, if $x \rightsquigarrow_R U^b$, then U^b is uniquely determined from R and x, and for a closed term of the form $R[t]$ and $x \in FV(t)$, there exists U^b such that $x \rightsquigarrow_R U^b$. We sometimes omit b; by $x \rightsquigarrow_R U$, we mean $x \rightsquigarrow_R U^b$ for some b. Given $x \rightsquigarrow_R U^b$, we define $R|_x$ as an N-reduction context such that

$$R = R|_x[\mathbf{let}\ x = U^b\ \mathbf{in}\ R'^{b'}] \quad \text{and} \quad x \not\rightsquigarrow_{R'} U^b$$

for some b' and N-reduction context R'; such $R|_x$ always exists and is unique. We use $R|_x$ as a context for U.

Any closed labeled A-normal form s is in exactly one of the following three cases.

$$R[\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}] \longrightarrow_{\mathrm{N}} \begin{cases} R[s_1] & (\text{if } x \rightsquigarrow_R 0^{b'}) \\ R[s_2] & (\text{if } x \rightsquigarrow_R m^{b'}, m \neq 0) \end{cases}$$

$$R[\mathbf{let}\ y = \mathsf{op}(x_1,\dots,x_n)^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{let}\ y = (\llbracket \mathsf{op} \rrbracket(m_1,\dots,m_n))^{b_1}\ \mathbf{in}\ s^{b_2}]$$

$$\quad \text{where } x_i \rightsquigarrow_R {m_i}^{b'_i}$$

$$R[\mathbf{let}\ y = (f\ (x_1,\dots,x_n,g_1,\dots,g_m))^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{cclet}\ y = \left(s'[x'_i \mapsto x_i]_i[g'_j \mapsto g_j]_j[f' \mapsto f]\right)^{b''}\ \mathbf{in}\ s^{b_2}]$$

$$\quad \text{where } f \rightsquigarrow_R \mathbf{fix}(f', \lambda(x'_1,\dots,x'_n,g'_1,\dots,g'_m).\, {s'}^{b''})^{b'}.$$

$$R[\mathbf{let}\ y = (\mathbf{pr}_i x)^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{let}\ y = {U_i}^{b'_i}\ \mathbf{in}\ s^{b_2}]$$

$$\quad \text{where } x \rightsquigarrow_R (x_1,\dots,x_n)^{b'}, x_i \rightsquigarrow_{R|_x} {U_i}^{b'_i}$$

$$R[\mathbf{let}\ y = \mathbf{fail}^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} \mathbf{fail}$$

Figure D.20: N-reduction rules

$$\mathbf{cclet}\ y = (x_1,\dots,x_n)^b\ \mathbf{in}\ {s'}^{b'} \overset{\mathrm{def}}{=} \mathbf{let}\ y = (x_1,\dots,x_n)^b\ \mathbf{in}\ {s'}^{b'}$$

$$\mathbf{cclet}\ y = (\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2})^b\ \mathbf{in}\ {s'}^{b'} \overset{\mathrm{def}}{=}$$

$$\quad \mathbf{if}\ x\ \mathbf{then}\ (\mathbf{cclet}\ y = {s_1}^b\ \mathbf{in}\ {s'}^{b'})^{b_1}\ \mathbf{else}\ (\mathbf{cclet}\ y = {s_2}^b\ \mathbf{in}\ {s'}^{b'})^{b_2}$$

$$\mathbf{cclet}\ y = (\mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2})^b\ \mathbf{in}\ {s'}^{b'} \overset{\mathrm{def}}{=} \mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ (\mathbf{cclet}\ y = s^b\ \mathbf{in}\ {s'}^{b'})^{b_2}$$

Figure D.21: Commuting-conversion of **let**

- $R[(x_1,\dots,x_n)]$
- $R[\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}]$
- $R[\mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ s^{b_2}]$
  where $d = \mathsf{op}(x_1,\dots,x_n),\ f\ (x_1,\dots,x_n),\ \mathbf{pr}_i x,\ \mathbf{fail}$

We will treat the first case as normal forms of N-reduction; so we define N-reduction rules for the other two cases. We call $\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}$ and $\mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ s^{b_2}$ in the two latter cases *redexes of N-reduction* (or simply *redexes*).

We define N-reduction rules for closed labeled A-normal forms in Figure D.20. Here, **cclet** is a kind of commuting-conversion of **let**: for labels b, b', a variable y, and 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 D.21. (Details of commuting conversion will be explained in Appendix D.3, especially at (D.16).) For A-normal forms t and t', we define $\mathbf{cclet}\ y = t\ \mathbf{in}\ t'$ in a similar way to Figure D.21. On the application case, below, we often suppose that $f = f'$, $x_i = x'_i$, and $g_j = g'_j$ due to α-renaming. For multiplicity annotations T and T', we define $T \longrightarrow_{\mathrm{N}} T'$ in a similar way to the definition of $T \longrightarrow T'$.

N-reduction reduces labeled A-normal forms to either labeled A-normal forms or **fail**; hence, the normal forms of N-reduction are either $R[(x_1,\dots,x_n)]$ or **fail**. It is clear that, if the evaluation of s terminates, so does N-reduction of s, which can be shown by defining some simulation relation.

Appendix D.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; we define $(-)^{\sharp'_{\mathrm{N}}}$ by tracking the information of the sets B via the labels put to labeled A-normal forms.

$$B_b^{\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}} \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } b = b_1 \text{ or } b_2 \\ B_b^{s_i} & \text{if } b \in L(s_i) \end{cases}$$

$$B_b^{\mathbf{let}\ x=d^{b_1}\ \mathbf{in}\ s^{b_2}} \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } b = b_1 \\ \emptyset & \text{if } d = \mathbf{fix}(f, \lambda y.\, {s'}^{b''}), b = b'' \\ B_b^{s'} & \text{if } d = \mathbf{fix}(f, \lambda y.\, {s'}^{b''}), b \in L(s') \\ \{x = d\} & \text{if } b = b_2, d = f(\widetilde{x_i}), \mathbf{pr}_i x' \\ \emptyset & \text{if } b = b_2, d \neq f(\widetilde{x_i}), \mathbf{pr}_i x' \\ B_b^s \cup \{x = d\} & \text{if } b \in L(s), d = f(\widetilde{x_i}), \mathbf{pr}_i x' \\ B_b^s & \text{if } b \in L(s), d \neq f(\widetilde{x_i}), \mathbf{pr}_i x' \end{cases}$$

Figure D.22: B_b^s: B used at b in the calculation of $(s)_T^{\sharp'_{34}}$

$$((x_1, ..., x_n, f_1, ..., f_m))_{T,s_0}^{\sharp'_{\mathrm{N}}} \overset{\text{def}}{=} (x_1, ..., x_n, \lambda(y_1, ..., y_m).\, (f_1\, y_1, ..., f_m\, y_m))$$
$$\big(\mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}\big)_{T,s_0}^{\sharp'_{\mathrm{N}}} \overset{\text{def}}{=} \mathbf{if}\ x\ \mathbf{then}\ (s_1)_{T,s_0}^{\sharp'_{\mathrm{N}}}\ \mathbf{else}\ (s_2)_{T,s_0}^{\sharp'_{\mathrm{N}}}$$
$$\big(\mathbf{let}\ x = d^{b_1}\ \mathbf{in}\ s^{b_2}\big)_{T,s_0}^{\sharp'_{\mathrm{N}}} \overset{\text{def}}{=} \mathbf{let}\ x = (d)_{T'|_d, B_{b_1}^{s_0}}^{\sharp'_{34}}\ \mathbf{in}\ (s)_{T,s_0}^{\sharp'_{\mathrm{N}}}$$
$$(\mathbf{fail})_{T,s_0}^{\sharp'_{\mathrm{N}}} \overset{\text{def}}{=} \mathbf{fail}$$

Figure D.23: $(-)^{\sharp'_{\mathrm{N}}}$: $(-)^{\sharp'_{34}}$ for labeled A-normal forms

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. We assume that for any label-disjoint labeled A-normal form s, all the variables declared by let-expressions in s are distinct. It is obvious that we have a canonical way of labeling by which, for an 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 $(-)_{T,B}^{\sharp'_{34}}$. For a label-disjoint labeled A-normal form s and $b \in L(s)$, we define B_b^s as in Figure D.22; B_b^s is merely the set B used at the position b in s when we calculate $(s)_T^{\sharp'_{34}}$ by the definition in Figure 13. 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, if further s_0 is label-disjoint, for $b \in L(s)$, $B_b^{s_0}$ is well-defined. For an A-normal form t_0, we write $B_b^{(t_0)^{\mathrm{lb}}}$ as $B_b^{t_0}$.

Now, given a label-disjoint labeled A-normal form s_0 and a labeled A-normal form s such that $s_0 \longrightarrow_{\mathrm{N}}^n C[s]$ ($n \geq 0$) for some context C and a multiplicity-annotation T for s_0 with $T \longrightarrow_{\mathrm{N}}^n T'$, we define a (non-labeled) term $(s)_{T,s_0}^{\sharp'_{\mathrm{N}}}$ by induction on s in Figure D.23; here, we also define $(-)_{T,s_0}^{\sharp'_{\mathrm{N}}}$ for **fail**, a normal form of N-reduction. On the **let**-case, below, we write $(d)_{T'|_d, B_{b_1}^{s_0}}^{\sharp'_{34}}$ simply as $(d)_{T, B_{b_1}^{s_0}}^{\sharp'_{34}}$. For an A-normal form t_0, we write $(-)_{T,(t_0)^{\mathrm{lb}}}^{\sharp'_{\mathrm{N}}}$ as $(-)_{T,t_0}^{\sharp'_{\mathrm{N}}}$.

$$([\,])^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=} [\,]$$
$$\big(\textbf{let}\ x = U^{b_1}\ \textbf{in}\ R^{b_2}\big)^{\sharp'_{\mathrm{N}}}_{T,s_0} \stackrel{\text{def}}{=} \textbf{let}\ x = (U)^{\sharp'_{34}}_{T,B^{s_0}_{b_1}}\ \textbf{in}\ (R)^{\sharp'_{\mathrm{N}}}_{T,s_0}\ .$$

Figure D.24: $(-)^{\sharp'_{\mathrm{N}}}$ for N-reduction contexts

Also, for an N-reduction context R such that $s_0 \longrightarrow^*_{\mathrm{N}} R[s]$ for some s, we define $(R)^{\sharp'_{\mathrm{N}}}_{T,s_0}$ in Figure D.24. Then,

$$(R[s])^{\sharp'_{\mathrm{N}}}_{T,s_0} = (R)^{\sharp'_{\mathrm{N}}}_{T,s_0}\big[(s)^{\sharp'_{\mathrm{N}}}_{T,s_0}\big]\ .$$

Appendix D.3. Preliminaries to main lemma

We show several lemmas for proving Lemma 27 given in the next subsection, from which the goal (Lemma 21) follows.

First, we give a lemma for **cclet**.

Lemma 22. *(1) For any y, s_1, s_2, b_1, b_2, and R,*

$$\textbf{cclet}\ y = R[s_1]^{b_1}\ \textbf{in}\ {s_2}^{b_2} \quad = \quad R[\textbf{cclet}\ y = {s_1}^{b_1}\ \textbf{in}\ {s_2}^{b_2}].$$

(2) For any x, y, s_1, s_2, s', b_1, b_2, b, and b',

$$\textbf{cclet}\ y = \left(\textbf{cclet}\ x = {s_1}^{b_1}\ \textbf{in}\ {s_2}^{b_2}\right)^{b}\ \textbf{in}\ {s'}^{b'} \quad = \quad \textbf{cclet}\ x = {s_1}^{b_1}\ \textbf{in}\ \left(\textbf{cclet}\ y = {s_2}^{b}\ \textbf{in}\ {s'}^{b'}\right)^{b_2}\ .$$

(3) For R, y, b_1, b_2, s_1, s_2, and s'_1,

$$R[s_1] \longrightarrow_{\mathrm{N}} R[s'_1]$$

implies

$$R[\textbf{cclet}\ y = {s_1}^{b_1}\ \textbf{in}\ {s_2}^{b_2}] \longrightarrow_{\mathrm{N}} R[\textbf{cclet}\ y = {s'_1}^{b_1}\ \textbf{in}\ {s_2}^{b_2}].$$

(4) For R, y, s, b, s', b', and an A-normal form t_0 such that

$$(t_0)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} R[\textbf{cclet}\ y = s^b\ \textbf{in}\ {s'}^{b'}]\ ,$$

and for a multiplicity annotation T for t_0,

$$\left(\textbf{cclet}\ y = s^b\ \textbf{in}\ {s'}^{b'}\right)^{\sharp'_{\mathrm{N}}}_{T,t_0} \quad = \quad \textbf{cclet}\ y = (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \textbf{in}\ (s')^{\sharp'_{\mathrm{N}}}_{T,t_0}.$$

Proof. (1) Clear by induction on R.

(2) Straightforward by induction on s_1.

(3) Straightforward by a case analysis on the redex of $R[s_1] \longrightarrow_{\mathrm{N}} R[s'_1]$; we use Lemma 22 - (1), and in the case of application, we use Lemma 22 - (2). We show only the application case.

Let

$$s_1 = R_1[\textbf{let}\ x = (f(\widetilde{x},\widetilde{g}))^{b'_1}\ \textbf{in}\ {s'}^{b'_2}]$$
$$f \rightsquigarrow_{R[R_1]} \textbf{fix}(f, \lambda(\widetilde{x},\widetilde{g}).\,{s'}^{b''})^{b'}\ .$$

Now $R[s_1] \longrightarrow_{\mathrm{N}} R[s'_1]$ and

$$R[s_1] \quad = \quad R[R_1[\textbf{let}\ x = (f(\widetilde{x},\widetilde{g}))^{b'_1}\ \textbf{in}\ {s'}^{b'_2}]] \ \longrightarrow_{\mathrm{N}}\ R[R_1[\textbf{cclet}\ x = {s'}^{b''}\ \textbf{in}\ {s'}^{b'_2}]].$$

Hence,

$$s'_1 \;=\; R_1[\mathbf{cclet}\ x = s'^{b''}\ \mathbf{in}\ s'^{b'_2}].$$

Then,

$$\begin{aligned}
& R[\mathbf{cclet}\ y = {s_1}^{b_1}\ \mathbf{in}\ {s_2}^{b_2}] \\
={}& R[\mathbf{cclet}\ y = (R_1[\mathbf{let}\ x = (f(\widetilde{x},\widetilde{g}))^{b'_1}\ \mathbf{in}\ s'^{b'_2}])^{b_1}\ \mathbf{in}\ {s_2}^{b_2}] \\
={}& \{\text{by Lemma 22 - (1)}\} \\
& R[R_1[\mathbf{cclet}\ y = (\mathbf{let}\ x = (f(\widetilde{x},\widetilde{g}))^{b'_1}\ \mathbf{in}\ s'^{b'_2})^{b_1}\ \mathbf{in}\ {s_2}^{b_2}]] \\
={}& R[R_1[\mathbf{let}\ x = (f(\widetilde{x},\widetilde{g}))^{b'_1}\ \mathbf{in}\ (\mathbf{cclet}\ y = s'^{b_1}\ \mathbf{in}\ {s_2}^{b_2})^{b'_2}]] \\
\longrightarrow_{\mathrm{N}}{}& R[R_1[\mathbf{cclet}\ x = s'^{b''}\ \mathbf{in}\ (\mathbf{cclet}\ y = s'^{b_1}\ \mathbf{in}\ {s_2}^{b_2})^{b'_2}]] \\
={}& \{\text{by Lemma 22 - (2)}\} \\
& R[R_1[\mathbf{cclet}\ y = (\mathbf{cclet}\ x = s'^{b''}\ \mathbf{in}\ s'^{b'_2})^{b_1}\ \mathbf{in}\ {s_2}^{b_2}]] \\
={}& \{\text{by Lemma 22 - (1)}\} \\
& R[\mathbf{cclet}\ y = (R_1[\mathbf{cclet}\ x = s'^{b''}\ \mathbf{in}\ s'^{b'_2}])^{b_1}\ \mathbf{in}\ {s_2}^{b_2}] \\
={}& R[\mathbf{cclet}\ y = {s'_1}^{b_1}\ \mathbf{in}\ {s_2}^{b_2}].
\end{aligned}$$

(4) Straightforward by induction on s. □

Next, we give a lemma on the labeling by b and show that $(-)^{\sharp'_{\mathrm{N}}}$ and $(-)^{\sharp'_{34}}$ are equal in some sense. We define the sets of *s-sb contexts* (ranged by S_{s}) and *s-db contexts* (ranged by S_{d}) by the following rules:

$$\begin{aligned}
S_{\mathrm{s}} &::= \mathbf{let}\ y = d^{b}\ \mathbf{in}\ l \mid \mathbf{let}\ y = \mathbf{fix}(f, \lambda x.\, l)^{b_1}\ \mathbf{in}\ s^{b_2} \mid \mathbf{if}\ x\ \mathbf{then}\ l\ \mathbf{else}\ s^{b} \mid \mathbf{if}\ x\ \mathbf{then}\ s^{b}\ \mathbf{else}\ l \\
l &::= [\,] \mid {S_{\mathrm{s}}}^{b}
\end{aligned}$$

$$\begin{aligned}
S_{\mathrm{d}} ::={}& \mathbf{let}\ y = [\,]\ \mathbf{in}\ s^{b} \mid \mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ {S_{\mathrm{d}}}^{b_2} \mid \mathbf{let}\ y = \mathbf{fix}(f, \lambda x.\, {S_{\mathrm{d}}}^{b})^{b_1}\ \mathbf{in}\ s^{b_2} \\
& \mid \mathbf{if}\ x\ \mathbf{then}\ {S_{\mathrm{d}}}^{b_1}\ \mathbf{else}\ s^{b_2} \mid \mathbf{if}\ x\ \mathbf{then}\ s^{b_1}\ \mathbf{else}\ {S_{\mathrm{d}}}^{b_2}
\end{aligned}$$

For any s, d, b, S_{s}, and S_{d}, $S_{\mathrm{s}}[s^{b}]$ and $S_{\mathrm{d}}[d^{b}]$ are labeled A-normal forms.

Lemma 23. *(1) Given a label-disjoint labeled A-normal form s_0, an s-db context S_{d}, and $\mathbf{fix}(f, \lambda x.\, s^{b})^{b'}$ such that $s_0 = S_{\mathrm{d}}[\mathbf{fix}(f, \lambda x.\, s^{b})^{b'}]$,*

$$B^{s_0}_{b} = B^{s_0}_{b'}.$$

(2) Given a label-disjoint labeled A-normal form s_0, an s-sb context S_{s}, and s^{b} such that $s_0 = S_{\mathrm{s}}[s^{b}]$, the following holds.

When $s = \mathbf{if}\ x\ \mathbf{then}\ {s_1}^{b_1}\ \mathbf{else}\ {s_2}^{b_2}$, we have $B^{s_0}_{b_1} = B^{s_0}_{b_2} = B^{s_0}_{b}$.

When $s = \mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ s'^{b_2}$, we have $B^{s_0}_{b_1} = B^{s_0}_{b}$; further,

- *when $d \neq f\ (\widetilde{x})$ nor $\mathbf{pr}_i x$, we have $B^{s_0}_{b_2} = B^{s_0}_{b}$,*
- *when $d = f\ (\widetilde{x})$ or $\mathbf{pr}_i x$, we have $B^{s_0}_{b_2} = B^{s_0}_{b} \cup \{x = d\}$.*

(3) Given a label-disjoint labeled A-normal form s_0, an s-sb context S_{s}, and s^{b} such that $s_0 = S_{\mathrm{s}}[s^{b}]$, and a multiplicity annotation T for s_0,

$$(s)^{\sharp'_{\mathrm{N}}}_{T,s_0} \;=\; (s)^{\sharp'_{34}}_{T,B^{s_0}_{b}}\,.$$

Proof. (1) Clear by induction on $S_{\rm d}$.

(2) Clear by induction on $S_{\rm s}$.

(3) Straightforward by induction on s and by (2) of this lemma. □

The next lemma shows that d^b occurring after N-reduction also occurs before the N-reduction (Lemma 24 - (1)), and shows where a binding in $B_b^{t_0}$ is N-reduced (Lemma 24 - (4)). Lemmas 24 - (2), (3) below are used only for Lemma 24 - (4). For $S_{\rm d}$, we define the set $LBV(S_{\rm d})$ of "let-binding variables" of $S_{\rm d}$ as follows:

$$\begin{array}{ll}
LBV(\textbf{let}\ y = [\,]\ \textbf{in}\ s^b) & \overset{\rm def}{=} \emptyset \\
LBV(\textbf{let}\ y = d^{b_1}\ \textbf{in}\ {S_{\rm d}}^{b_2}) & \overset{\rm def}{=} LBV(S_{\rm d}) \cup \{y\} \\
LBV(\textbf{let}\ y = \textbf{fix}(f, \lambda x.\, {S_{\rm d}}^b)^{b_1}\ \textbf{in}\ s^{b_2}) & \overset{\rm def}{=} LBV(S_{\rm d}) \\
LBV(\textbf{if}\ x\ \textbf{then}\ {S_{\rm d}}^{b_1}\ \textbf{else}\ s^{b_2}) & \overset{\rm def}{=} LBV(S_{\rm d}) \\
LBV(\textbf{if}\ x\ \textbf{then}\ s^{b_1}\ \textbf{else}\ {S_{\rm d}}^{b_2}) & \overset{\rm def}{=} LBV(S_{\rm d})
\end{array}$$

Similarly, for R we define $LBV(R)$ as follows:

$$\begin{array}{ll}
LBV([\,]) & \overset{\rm def}{=} \emptyset \\
LBV(\textbf{let}\ x = U^{b_1}\ \textbf{in}\ R^{b_2}) & \overset{\rm def}{=} LBV(R) \cup \{x\}
\end{array}$$

We say that $(x = d)$ *occurs in* s if there exist some b_1, b_2, s', and C such that $s = C[\textbf{let}\ x = d^{b_1}\ \textbf{in}\ s'^{b_2}]$.

Lemma 24. *(1) For any d^b, s, and ${S_{\rm d}}'$ such that*

$$s \longrightarrow_{\rm N} {S_{\rm d}}'[d^b],$$

there exists $S_{\rm d}$ such that

$$s = S_{\rm d}[d^b] \quad \textit{and} \quad LBV(S_{\rm d}) \subseteq LBV({S_{\rm d}}').$$

(2) For any $S_{\rm d}$ and d^b such that $S_{\rm d}[d^b]$ is label-disjoint,

$$\{w \mid \exists d.\, (w = d) \in B_b^{S_{\rm d}[d^b]}\} \subseteq LBV(S_{\rm d}).$$

(3) For any s, s', x, and d such that d is not a labeled value and $s \longrightarrow_{\rm N} s'$, if $(x = d)$ occurs in s', so does in s.

(4) For a closed ground A-normal form t_0 such that

$$(t_0)^{\rm lb} \longrightarrow_{\rm N}^{*} R'[\textbf{let}\ y = d'^{b_1'}\ \textbf{in}\ {s_2'}^{b_2'}]$$

and for $(w = d) \in B_{b_1'}^{t_0}$, there exist b_1, b_2, R, and s such that

$$(t_0)^{\rm lb} \longrightarrow_{\rm N}^{*} R[\textbf{let}\ w = d^{b_1}\ \textbf{in}\ s^{b_2}] \longrightarrow_{\rm N}^{+} R'[\textbf{let}\ y = d'^{b_1'}\ \textbf{in}\ {s_2'}^{b_2'}]$$

where $\longrightarrow_{\rm N}^{+}$ is the transitive closure of $\longrightarrow_{\rm N}$.

Proof. (1) The proof is given by a case analysis on the redex of s. We show only the case of application; the other cases are clear. Hence, we suppose the following:

$$s \;=\; R[\textbf{let}\ y = (f\,(\widetilde{x}, \widetilde{g}))^{b_1}\ \textbf{in}\ {s_2}^{b_2}] \tag{D.10}$$

$$f \rightsquigarrow_R \textbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, s''^{b''})^{b'''} \tag{D.11}$$

$${S_{\rm d}}'[d^b] \;=\; R[\textbf{cclet}\ y = s''^{b''}\ \textbf{in}\ {s_2}^{b_2}] \tag{D.12}$$

By (D.12), d^b occurs in R, s'', or s_2; the cases of R and s_2 are clear. In the case of s'', there exists some S_{d}'' such that

$$s'' \;=\; S_{\mathrm{d}}''[d^b]$$

and

$$S_{\mathrm{d}}' \;=\; R[\mathbf{cclet}\; y = {S_{\mathrm{d}}''}^{b''} \;\mathbf{in}\; {s_2}^{b_2}].$$

By (D.11), there exist some R' and b' such that

$$R \;=\; R|_f\big[\mathbf{let}\; f = \mathbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, {s''}^{b''})^{b'''} \;\mathbf{in}\; {R'}^{b'}\big],$$

and so we define

$$S_{\mathrm{d}} \;\stackrel{\mathrm{def}}{=}\; R|_f\big[\mathbf{let}\; f = \mathbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, {S_{\mathrm{d}}''}^{b''})^{b'''} \;\mathbf{in}\; R'[\mathbf{let}\; y = (f\;(\widetilde{x}, \widetilde{g}))^{b_1} \;\mathbf{in}\; {s_2}^{b_2}]^{b'}\big].$$

Then,

$$S_{\mathrm{d}}[d^b] \;=\; s$$

and

$$LBV(S_{\mathrm{d}}) \;=\; LBV(R|_f) \cup LBV(S_{\mathrm{d}}'') \;\subseteq\; LBV(R) \cup LBV(S_{\mathrm{d}}'') \;=\; LBV(S_{\mathrm{d}}').$$

(2) Straightforward by induction on S_{d}.

(3) Straightforward by a case analysis on the redex of s.

(4) Let $S_{\mathrm{d}}' \stackrel{\mathrm{def}}{=} R'[\mathbf{let}\; y = [\,] \;\mathbf{in}\; {s_2'}^{b_2'}]$. By using (1) of this lemma repeatedly, there exists S_{d} such that

$$(t_0)^{\mathrm{lb}} = S_{\mathrm{d}}[{d'}^{b_1'}] \qquad LBV(S_{\mathrm{d}}) \subseteq LBV(S_{\mathrm{d}}'). \tag{D.13}$$

By (2) of this lemma,

$$w \in \{w' \mid \exists d'.\, (w' = d') \in B^{t_0}_{b_1'}\} \subseteq LBV(S_{\mathrm{d}}).$$

Since $LBV(S_{\mathrm{d}}') = LBV(R'),\; w \in LBV(R')$. Now, let us define R_i and r_i as

$$(t_0)^{\mathrm{lb}} = R_0[r_0] \longrightarrow_{\mathrm{N}} R_1[r_1] \longrightarrow_{\mathrm{N}} \ldots$$

until the N-reduction terminates where each r_i is the redex of $R_i[r_i]$. Then, we define

$$n \;\stackrel{\mathrm{def}}{=}\; \min\{i \mid w \in LBV(R_i)\} - 1.$$

Hence, we have

$$R_n[r_n] \longrightarrow_{\mathrm{N}} R_{n+1}[r_{n+1}]$$

where $w \notin LBV(R_n)$ and $w \in LBV(R_{n+1})$. Let $R_{n+1}[r_{n+1}]$ is of the form $R_n[s''']$; then, there exists U such that $(w = U)$ occurs in s'''.

Note that, if $n = -1$ above, $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$. If $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$, it yields a contradiction as follows. Since b_1' occurs in t_0 by (D.13), and since $(w = d) \in B^{t_0}_{b_1'}$, $(w = d)$ also occurs in $(t_0)^{\mathrm{lb}}$, which can be shown by induction on t_0. We defined the notion of an A-normal form as all the variables declared by let-expressions are distinct; hence, the fact that both $(w = U)$ and $(w = d)$ occur in t_0 is a contradiction, since d is not a labeled value and so $U \neq d$.

Also note that, if the following holds:

$$\text{for some } d'', b_1, b_2, R, \text{ and } s,\; \left(\begin{array}{l} d'' \text{ is not a labeled value and} \\ (t_0)^{\mathrm{lb}} \longrightarrow_{\mathrm{N}}^{*} R[\mathbf{let}\; w = {d''}^{b_1} \;\mathbf{in}\; s^{b_2}] \longrightarrow_{\mathrm{N}}^{+} R'[\mathbf{let}\; y = {d'}^{b_1'} \;\mathbf{in}\; {s_2'}^{b_2'}]\,, \end{array}\right) \tag{D.14}$$

then $d'' = d$ and hence the goal holds; this is shown as follows. As explained above, $(w = d)$ occurs in $(t_0)^{\mathrm{lb}}$ and the variables are distinct, and also $(w = d'')$ occurs in $(t_0)^{\mathrm{lb}}$ by using (3) of this lemma repeatedly; hence, d and d'' must be the same.

Now by using the fact that $(w = U)$ occurs in s''', we show that $(w = U)$ occurs in r_n or the goal of this lemma holds; we show this by a case analysis on the redex r_n. In the case that

$$R_n[\textbf{if } x \textbf{ then } {s_1}^{b_1} \textbf{ else } {s_2}^{b_2}] \longrightarrow_{\mathrm{N}} \begin{cases} R_n[s_1] & (x \rightsquigarrow_{R_n} 0^{b'}) \\ R_n[s_2] & (x \rightsquigarrow_{R_n} m^{b'}, m \neq 0) \end{cases}$$

$(w = U)$ occurs in s_i, and hence also in r_n. In the case that

$$R_n[\textbf{let } y = \mathtt{op}(x_1, \ldots, x_n)^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R_n[\textbf{let } y = (\llbracket \mathtt{op} \rrbracket(m_1, \ldots, m_n))^{b_1} \textbf{ in } s^{b_2}]$$

if $y = w$ and $\llbracket \mathtt{op} \rrbracket(m_1, \ldots, m_n) = d$, then (D.14) holds with $d'' = \mathtt{op}(x_1, \ldots, x_n)$; otherwise, $(w = U)$ occurs in s and hence in r_n. In the case that

$$R_n[\textbf{let } y = (f\ (\widetilde{x}, \widetilde{g}))^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R_n[\textbf{cclet } y = s'^{b''} \textbf{ in } s^{b_2}]$$
$$f \rightsquigarrow_{R_n} \textbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, s'^{b''})^{b'}$$

if $y = w$, then (D.14) holds; otherwise if $(w = U)$ occurs in s, so does in r_n; otherwise, $(w = U)$ occurs in s', and then by (1) whose d is instantiated by $\textbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, s'^{b''})$, $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$. In the case that

$$R_n[\textbf{let } y = (\textbf{pr}_i x)^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R_n[\textbf{let } y = {U_i}^{b'_i} \textbf{ in } s^{b_2}]$$

if $y = w$, then (D.14) holds; otherwise if $(w = U)$ occurs in s, so does in r_n; otherwise, $(w = U)$ occurs in U_i, and then by (1) whose d is instantiated by U_i, $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$.

Finally, we show that, for any $n \geq 0$, R', and r' such that

$$(t_0)^{\mathrm{lb}} \longrightarrow^n_{\mathrm{N}} R'[r']$$

and r' is the redex of $R'[r']$, if $(w = U)$ occurs in r', then the goal of this lemma holds; if this is proved, we instantiate $R'[r']$ with the $R_n[r_n]$ above. We show this by induction on n. In the case $n = 0$, $(w = U)$ occurs in t_0, which is a contradiction. In the case $n > 0$, suppose

$$(t_0)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} R[r] \longrightarrow_{\mathrm{N}} R'[r']$$

and $(w = U)$ occurs in r'; we show the goal of this lemma by a case analysis on the redex r. The following proof goes on quite similarly to the previous. In the case that

$$R[\textbf{if } x \textbf{ then } {s_1}^{b_1} \textbf{ else } {s_2}^{b_2}] \longrightarrow_{\mathrm{N}} \begin{cases} R[s_1] & (x \rightsquigarrow_{R} 0^{b'}) \\ R[s_2] & (x \rightsquigarrow_{R} m^{b'}, m \neq 0) \end{cases}$$

$(w = U)$ occurs in s_i, and so does in r; hence the goal holds by IH. In the case that

$$R[\textbf{let } y = \mathtt{op}(x_1, \ldots, x_n)^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R[\textbf{let } y = (\llbracket \mathtt{op} \rrbracket(m_1, \ldots, m_n))^{b_1} \textbf{ in } s^{b_2}]$$

if $y = w$, then (D.14) holds; if $(w = U)$ occurs in s, so does in r, and hence the goal holds by IH. In the case that

$$R[\textbf{let } y = (f\ (\widetilde{x}, \widetilde{g}))^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R[\textbf{cclet } y = s'^{b''} \textbf{ in } s^{b_2}]$$
$$f \rightsquigarrow_{R} \textbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, s'^{b''})^{b'}$$

if $y = w$, then (D.14) holds; if $(w = U)$ occurs in s, so does in r, and hence the goal holds by IH; otherwise, $(w = U)$ occurs in s', and then by (1) whose d is instantiated by $\textbf{fix}(f, \lambda(\widetilde{x}, \widetilde{g}).\, s'^{b''})$, $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$. In the case that

$$R[\textbf{let } y = (\textbf{pr}_i x)^{b_1} \textbf{ in } s^{b_2}] \longrightarrow_{\mathrm{N}} R[\textbf{let } y = {U_i}^{b'_i} \textbf{ in } s^{b_2}]$$

if $y = w$, then (D.14) holds; if $(w = U)$ occurs in s, so does in r, and hence the goal holds by IH; otherwise, $(w = U)$ occurs in U_i, and then by (1) whose d is instantiated by U_i, $(w = U)$ occurs in $(t_0)^{\mathrm{lb}}$. □

$$
\begin{array}{rcl}
\bar{V} & ::= & n \mid \mathbf{fix}(f, \lambda x.\, t) \mid (\bar{V}_1, \ldots, \bar{V}_n) \mid x \mid \mathbf{pr}_i \bar{V} \\
\bar{E} & ::= & [\,] \mid \mathsf{op}(\widetilde{\bar{V}}, \bar{E}, \widetilde{t}) \mid \mathbf{if}\ \bar{E}\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid \bar{E}\, t \mid \bar{V}\, \bar{E} \mid (\widetilde{\bar{V}}, \bar{E}, \widetilde{t}) \mid \mathbf{pr}_i \bar{E}
\end{array}
$$

$$
\begin{array}{rclrcl}
(\mathbf{fix}(f, \lambda x.\, t))\, \bar{V} & = & t[f \mapsto \mathbf{fix}(f, \lambda x.\, t)][x \mapsto \bar{V}] & \lambda x.\, \bar{V}\, x & = & \bar{V} \quad (x \notin FV(\bar{V})) \\
\mathbf{pr}_i(\bar{V}_1, \ldots, \bar{V}_n) & = & \bar{V}_i & (\mathbf{pr}_1 \bar{V}, \ldots, \mathbf{pr}_n \bar{V}) & = & \bar{V}
\end{array}
$$

$$
\bar{E}[t] \; = \; (\mathbf{let}\ x = t\ \mathbf{in}\ \bar{E}[x]) \quad (x \notin FV(\bar{E}))
$$

Figure D.25: λ_{c}-calculus

Below, we often use a computational lambda calculus (λ_{c}-calculus for short) [10], which is the standard call-by-value equational theory. Their axioms are listed in Figure D.25, where note that the notion of a value $\bar{V}$ for this equational theory can be open and hence the rule for $\bar{V}$ is extended from that for V in Figure 1 as follows: (i) any variable is a value and (ii) a projection of a value is also a value. The form of the rule for $\bar{E}$ is the same as that for E in Figure 1 except that the notion of a value $\bar{V}$ is extended as above. In the rest of this section, we do not use the original notion of a value and use only the extended notion of a value; so we call the latter simply a *value* and use the metavariable V rather than $\bar{V}$. We do similarly also for the notion of an evaluation context. The λ_{c}-calculus is sound for the observational equivalence, and we write equations proved by the λ_{c}-calculus simply by using $=_{\mathrm{o}}$.

The last axiom in Figure D.25 is called *commuting conversion*. To explain how this axiom can be used, we prove the following equation, which is used later. For any terms $t_1, \ldots, t_n$, and t',

$$
\mathbf{let}\ x = (t_1, \ldots, t_n)\ \mathbf{in}\ t' \quad =_{\mathrm{o}} \quad \left(\begin{array}{l} \mathbf{let}\ x_1 = t_1\ \mathbf{in} \\ \ldots \\ \mathbf{let}\ x_n = t_n\ \mathbf{in}\ t'[x \mapsto (x_1, \ldots, x_n)] \end{array} \right). \tag{D.15}
$$

This is proved as follows; below, we insert $[-]$ to indicate which evaluation context is used.

$$
\begin{array}{rcl}
\mathbf{let}\ x = ([t_1], \ldots, t_n)\ \mathbf{in}\ t' & =_{\mathrm{o}} & \left(\begin{array}{l} \mathbf{let}\ x_1 = t_1\ \mathbf{in} \\ \mathbf{let}\ x = ([x_1], t_2, \ldots, t_n)\ \mathbf{in}\ t' \end{array} \right) \\
& = & \left(\begin{array}{l} \mathbf{let}\ x_1 = t_1\ \mathbf{in} \\ \mathbf{let}\ x = (x_1, [t_2], \ldots, t_n)\ \mathbf{in}\ t' \end{array} \right) \\
& =_{\mathrm{o}} & \cdots \\
& =_{\mathrm{o}} & \left(\begin{array}{l} \mathbf{let}\ x_1 = t_1\ \mathbf{in} \\ \ldots \\ \mathbf{let}\ x_n = t_n\ \mathbf{in} \\ \mathbf{let}\ x = (x_1, \ldots, [x_n])\ \mathbf{in}\ t' \end{array} \right) \\
& =_{\mathrm{o}} & \left(\begin{array}{l} \mathbf{let}\ x_1 = t_1\ \mathbf{in} \\ \ldots \\ \mathbf{let}\ x_n = t_n\ \mathbf{in}\ t'[x \mapsto (x_1, \ldots, x_n)] \end{array} \right)
\end{array}
$$

Also, the equation below follows from the commuting conversion axiom:

$$
\mathbf{cclet}\ y = t\ \mathbf{in}\ t' \quad =_{\mathrm{o}} \quad \mathbf{let}\ y = t\ \mathbf{in}\ t', \tag{D.16}
$$

which can be shown by a straightforward induction on t.

In addition to λ_{c}-calculus, we use the following reasoning principle, which we call *referential transparency*:

$$
\mathbf{let}\ x = t\ \mathbf{in}\ C[x] \quad = \quad \mathbf{let}\ x = t\ \mathbf{in}\ C[t] \tag{RT}
$$

where the occurrence x in $C[x]$ must be free (and bound by the let-declaration). Here we used a context C for $C[x]$ and $C[t]$ rather than using a term t' as t' and $t'[x \mapsto t]$; this means that any *one* occurrence of x in $C[x]$ can be replaced with t (and hence, by repeating it, so can *all* the occurrences of x, too). It is clear that (RT) is sound with respect to the observational equivalence of our language.

The axiom (RT) allows us to regard a let-binding $x \rightsquigarrow_R t^b$ 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, if $x \rightsquigarrow_R t^b$, by (RT) it is true that $x \equiv t$ in R (though x and t themselves are not necessarily observationally equivalent). We sometimes omit the context C if it is clear.

By (RT), we can prove the following equations:

Lemma 25. *(1) For any term t,*

$$\mathbf{pr}_1(t, \dots, t) =_{\mathrm{o}} t.$$

(2) For any term t of an n-tuple type,

$$t =_{\mathrm{o}} (\mathbf{pr}_1 t, \dots, \mathbf{pr}_n t).$$

Proof. (1)

$$\begin{aligned} & \mathbf{pr}_1(t, t, t, \dots, t) \\ =_{\mathrm{o}} & \{\text{by commuting conversion}\} \\ & \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{pr}_1(x, t, t, \dots, t) \\ =_{\mathrm{o}} & \{\text{(RT)}\} \\ & \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{pr}_1(x, x, t, \dots, t) \\ =_{\mathrm{o}} & \cdots \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ \mathbf{pr}_1(x, x, x, \dots, x) \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ x \\ =_{\mathrm{o}} & t\,. \end{aligned}$$

(2) Similarly,

$$\begin{aligned} & (\mathbf{pr}_1 t, \mathbf{pr}_2 t, \mathbf{pr}_3 t, \dots, \mathbf{pr}_n t) \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ (\mathbf{pr}_1 x, \mathbf{pr}_2 t, \mathbf{pr}_3 t, \dots, \mathbf{pr}_n t) \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ (\mathbf{pr}_1 x, \mathbf{pr}_2 x, \mathbf{pr}_3 t, \dots, \mathbf{pr}_n t) \\ =_{\mathrm{o}} & \cdots \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ (\mathbf{pr}_1 x, \mathbf{pr}_2 x, \mathbf{pr}_3 x, \dots, \mathbf{pr}_n x) \\ =_{\mathrm{o}} & \mathbf{let}\ x = t\ \mathbf{in}\ x \\ =_{\mathrm{o}} & t\,. \end{aligned}$$

□

Since (RT) is a global reasoning principle, it does not have the compositionality; however, the following lemma recovers the compositionality under some restriction. For a context C—defined before Lemma 9—we define the set $BV(C)$ of binding variables of C as follows.

$$BV([\,]) \overset{\text{def}}{=} \emptyset$$

$$\left\{ \begin{array}{l} BV(\mathbf{if}\ C\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2)\ ,\ BV(\mathbf{if}\ t\ \mathbf{then}\ C\ \mathbf{else}\ t_2)\ ,\ BV(\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ C) \\ BV(\mathsf{op}(\widetilde{t}, C, \widetilde{t}))\ ,\ BV(C\, t_2)\ ,\ BV(t_1\, C)\ ,\ BV((\widetilde{t}, C, \widetilde{t}))\ ,\ BV(\mathbf{pr}_i C) \end{array} \right\} \overset{\text{def}}{=} BV(C)$$

$$BV(\mathbf{fix}(f, \lambda x.\, C)) \overset{\text{def}}{=} BV(C) \cup \{f, x\}$$

Lemma 26. *For a closed ground A-normal form t_0 such that*

$$(t_0)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} R[s]$$

and for any terms t, t', and a context C such that

$$BV(R) \cap BV(C) \;=\; \emptyset$$

and t, $C[t]$, t', and $C[t']$ have the same simple type,

$$t \equiv t' \textit{ in } (R)^{\sharp'_{\mathrm{N}}}_{T,t_0} \quad \textit{implies} \quad C[t] \equiv C[t'] \textit{ in } (R)^{\sharp'_{\mathrm{N}}}_{T,t_0} .$$

Proof. Now $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$ is of the form

$$(R)^{\sharp'_{\mathrm{N}}}_{T,t_0} \quad = \quad \mathbf{let}\ x_1 = V_1\ \mathbf{in} \ldots \mathbf{let}\ x_n = V_n\ \mathbf{in}\ [\,]$$

for some values V_i. Then,

$$t[x_i \mapsto V_i]_i \;=_{\mathrm{o}}\; (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[t] \;=_{\mathrm{o}}\; (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[t'] \;=_{\mathrm{o}}\; t'[x_i \mapsto V_i]_i$$

and hence

$$\begin{aligned}(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[C[t]] &=_{\mathrm{o}} C[t][x_i \mapsto V_i]_i \\ &=_{\mathrm{o}} (C[x_i \mapsto V_i]_i)[t[x_i \mapsto V_i]_i] \\ &=_{\mathrm{o}} (C[x_i \mapsto V_i]_i)[t'[x_i \mapsto V_i]_i] \\ &=_{\mathrm{o}} C[t'][x_i \mapsto V_i]_i \\ &=_{\mathrm{o}} (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[C[t']].\end{aligned}$$

□

Appendix D.4. Main lemma

The whole this subsection is devoted to the proof of the next lemma. Our goal, Lemma 21, can be proved easily by the following lemma; i.e., in the case of $t \longrightarrow^*_{\mathrm{N}} R[(x_1, \ldots, x_n)]$,

$$(t)^{\sharp'_{34}}_{T} \;=\; \left((t)^{\mathrm{lb}}\right)^{\sharp'_{\mathrm{N}}}_{T,t} \;=_{\mathrm{o}}\; (R[(x_1, \ldots, x_n)])^{\sharp'_{\mathrm{N}}}_{T,t} \;=_{\mathrm{o}}\; (R[(x_1, \ldots, x_n)])^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (t)^{\sharp_{34}}_{T},$$

and the case of $t \longrightarrow^*_{\mathrm{N}}$ **fail** is similar.

Lemma 27. *(1) For an A-normal form t and a multiplicity annotation T for t,*

$$(t)^{\sharp'_{34}}_{T} \;=\; \left((t)^{\mathrm{lb}}\right)^{\sharp'_{\mathrm{N}}}_{T,t} .$$

(2) For a closed ground A-normal form t and a multiplicity annotation T for t,

$$(t)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} R[(x_1, \ldots, x_n)]$$

implies

$$(R[(x_1, \ldots, x_n)])^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (R[(x_1, \ldots, x_n)])^{\sharp'_{\mathrm{N}}}_{T,t} .$$

(3) For a closed ground A-normal form t and a multiplicity annotation T for t,

$$(t)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} s$$

implies

$$(t)^{\sharp_{34}}_{T} \;=_{\mathrm{o}}\; (s)^{\sharp_{34}}_{T} \qquad \textit{and} \qquad \left((t)^{\mathrm{lb}}\right)^{\sharp'_{\mathrm{N}}}_{T,t} \;=_{\mathrm{o}}\; (s)^{\sharp'_{\mathrm{N}}}_{T,t} .$$

Proof. (1) The proof is almost the same as the proof of Lemma 23 - (3).

(2) Since $R[(x_1,\dots,x_n)]$ is ground, x_i are integer variables. Hence, for each i, there exists some integer m_i such that $x_i \rightsquigarrow_R {m_i}^{b_i}$. Therefore, both $(R[(x_1,\dots,x_n)])^{\sharp_{34}}_{T}$ and $(R[(x_1,\dots,x_n)])^{\sharp'_{\mathrm{N}}}_{T,t}$ are observationally equivalent to $(m_1,\dots,m_n)$ by β-conversions, where the β-conversions are possible since all the top-level let-expressions in $(R)^{\sharp_{34}}_{T}[(x_1,\dots,x_n)]$ and $(R)^{\sharp'_{\mathrm{N}}}_{T,t}[(x_1,\dots,x_n)]$ bind variables by *values*.

(3) A proof on $(-)^{\sharp_{34}}$ is obvious after we prove the case of $(-)^{\sharp'_{\mathrm{N}}}$. For the case of $(-)^{\sharp'_{\mathrm{N}}}$, we prove a bit stronger result: we prove that, for a closed ground A-normal form t_0,

$$(t_0)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} s_1 \longrightarrow_{\mathrm{N}} s_2 \quad \text{implies} \quad (s_1)^{\sharp'_{\mathrm{N}}}_{T,t_0} =_{\mathrm{o}} (s_2)^{\sharp'_{\mathrm{N}}}_{T,t_0}$$

and moreover,

- if
$$s_1 = R[\mathbf{let}\ y = d^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{let}\ y = U^{b'_1}\ \mathbf{in}\ s^{b_2}]$$
where $d = \mathtt{op}(x_1,\dots,x_n)$ or $\mathbf{pr}_i x$, then in context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,
$$(d)^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv (d)^{\sharp_{34}}_{T} \equiv (U)^{\sharp'_{34}}_{T,B^{t_0}_{b'_1}}\ ,$$

- and if
$$\begin{aligned} s_1 \ &=\ R[\mathbf{let}\ y = (f\,(\widetilde{x_i},\widetilde{g_j}))^{b_1}\ \mathbf{in}\ s^{b_2}] \longrightarrow_{\mathrm{N}} R[\mathbf{cclet}\ y = {s'}^{b''}\ \mathbf{in}\ s^{b_2}] \\ f &\rightsquigarrow_R \left(\mathbf{fix}(f, \lambda(\widetilde{x_i},\widetilde{g_j}).\,{s'}^{b''})\right)^{b'} \end{aligned}$$
then in context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,
$$(f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \equiv (f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp_{34}}_{T} \equiv (s')^{\sharp'_{\mathrm{N}}}_{T,t_0}\ . \tag{D.17}$$

We prove this by induction on the length of the N-reduction $(t_0)^{\mathrm{lb}} \longrightarrow^*_{\mathrm{N}} s_1$ and by a case analysis on the redex of s_1. We show only the case of application, which is subtle since it involves $\mathtt{InstVar}(-)$; the other cases are obvious.

Since $f \rightsquigarrow_R \left(\mathbf{fix}(f, \lambda(\widetilde{x_i},\widetilde{g_j}).\,{s'}^{b''})\right)^{b'}$, applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$f \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} (\mathbf{fix}(f, \lambda(\widetilde{x_i},\widetilde{g_j}).\,s'))^{\sharp'_{34}}_{T,B^{t_0}_{b'}} = \mathbf{fix}(f, \lambda(z_1,\dots,z_{T(f)}).\,(s'_1,\dots,s'_{T(f)})) \tag{D.18}$$

where

$$\begin{aligned} s'_k &\overset{\mathrm{def}}{=} (s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\,[x_i \mapsto \mathbf{pr}_i z_k]_{i\le n}[g_j \mapsto p^{z_k}_j]_{j\le m} \\ p^{z_k}_j &\overset{\mathrm{def}}{=} \lambda y.\,\mathbf{pr}_j((\mathbf{pr}_{n+1} z_k)(\overrightarrow{\bot}^{j-1}, y, \overrightarrow{\bot}^{m-j}))\ . \end{aligned}$$

Now,

$$\begin{aligned} (s_1)^{\sharp'_{\mathrm{N}}}_{T,t_0} \ &=\ (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[\mathbf{let}\ y = (f\,(\widetilde{x_i},\widetilde{g_j}))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}}\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}], \\ (s_2)^{\sharp'_{\mathrm{N}}}_{T,t_0} \ &=\ (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\left[\left(\mathbf{cclet}\ y = {s'}^{b''}\ \mathbf{in}\ s^{b_2}\right)^{\sharp'_{\mathrm{N}}}_{T,t_0}\right] \\ &=\ \{\text{by Lemma 22 - (4)}\} \\ &\quad\ (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\left[\mathbf{cclet}\ y = (s')^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\right] \\ &=_{\mathrm{o}}\ \{\text{by (D.16)}\} \\ &\quad\ (R)^{\sharp'_{\mathrm{N}}}_{T,t_0}\left[\mathbf{let}\ y = (s')^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\right], \end{aligned}$$

and hence, $(s_1)^{\sharp'_{\mathrm{N}}}_{T,t_0} =_{\mathrm{o}} (s_2)^{\sharp'_{\mathrm{N}}}_{T,t_0}$ follows from (D.17) by Lemma 26 where $C = \big(\mathbf{let}\ y = [\,]\ \mathbf{in}\ (s)^{\sharp'_{\mathrm{N}}}_{T,t_0}\big)$.

In the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$\begin{aligned}
&(f\ (\widetilde{x_i}, \widetilde{g_j}))^{\sharp_{34}}_{T} \\
&= \mathbf{pr}_1(f(\overrightarrow{(\widetilde{x_i}, \lambda\widetilde{y_j}.\,(g_j\ y_j)_j)}^{T(f)})) \\
&\equiv\ \{\text{by (D.18) and } \beta\text{-conversion}\} \\
&\quad \mathbf{pr}_1\Big(\big(s'_k[z_k \mapsto (\widetilde{x_i}, \lambda\widetilde{y_j}.\,(g_j\ y_j)_j)]\big)_{k\le T(f)}\Big) \\
&= \mathbf{pr}_1\Big(\Big((s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ [x_i \mapsto \mathbf{pr}_i(\widetilde{x_i}, \lambda\widetilde{y_j}.\,(g_j\ y_j)_j)]_{i\le n}\ \big[g_j \mapsto p_j^{z_k}[z_k \mapsto (\widetilde{x_i}, \lambda\widetilde{y_j}.\,(g_j\ y_j)_j)]\big]_{j\le m}\Big)_{k\le T(f)}\Big) \\
&\equiv \mathbf{pr}_1\Big(\Big((s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ \big[g_j \mapsto \lambda y.\,\mathbf{pr}_j((\lambda\widetilde{y_j}.\,(g_j\ y_j)_j)(\overrightarrow{\bot}, y, \overrightarrow{\bot}))\big]_j\Big)_{k\le T(f)}\Big) \\
&\equiv \mathbf{pr}_1\Big(\Big((s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\ [g_j \mapsto \lambda y.\,g_j\ y]_j\Big)_{k\le T(f)}\Big) \\
&\equiv \mathbf{pr}_1\Big(\Big((s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}}\Big)_{k\le T(f)}\Big) \\
&\equiv\ \{\text{by Lemma 25 - (1)}\} \\
&\quad (s')^{\sharp'_{34}}_{T,B^{t_0}_{b'}} \\
&=\ \big\{B^{t_0}_{b'} = B^{t_0}_{b''} \text{ by Lemma 23 - (1) and Lemma 24 - (1), as explained below}\big\} \\
&\quad (s')^{\sharp'_{34}}_{T,B^{t_0}_{b''}} \\
&=\ \{\text{by Lemma 23 - (3) and Lemma 24 - (1) as explained below}\} \\
&\quad (s')^{\sharp'_{\mathrm{N}}}_{T,t_0}\ .
\end{aligned}$$

Here, the last two equations above are shown as follows. We show only the latter equation; the former can be shown similarly. Since $f \rightsquigarrow_R \Big(\mathbf{fix}(f, \lambda(\widetilde{x_i}, \widetilde{g_j}).\,s'^{b''})\Big)^{b'}$, the term $\mathbf{fix}(f, \lambda(\widetilde{x_i}, \widetilde{g_j}).\,s'^{b''})$ occurs in R and hence in s_1. By Lemma 24 - (1), $\mathbf{fix}(f, \lambda(\widetilde{x_i}, \widetilde{g_j}).\,s'^{b''})$ occurs in $(t_0)^{\mathrm{lb}}$; hence, since $s'^{b''}$ occurs in $\mathbf{fix}(f, \lambda(\widetilde{x_i}, \widetilde{g_j}).\,s'^{b''})$, so does in $(t_0)^{\mathrm{lb}}$. By applying Lemma 23 - (3) where we substitute $(t_0)^{\mathrm{lb}}$ and $s'^{b''}$ for s_0 and s^b respectively, we obtain the last equation above.

Hence, what remains for proving (D.17) 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 $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$.

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

$$\begin{aligned}
t_1 &\stackrel{\text{def}}{=} \mathbf{pr}_1(f(\overrightarrow{(\widetilde{x_i}, \lambda\widetilde{y_j}.\,(g_j\ y_j)_j)}^{T(f)})) \\
t_{j+1} &\stackrel{\text{def}}{=} \mathtt{InstVar}(g_j, T, B^{t_0}_{b_1}, t_j) \quad (\text{for } j = 1, ..., m)\ .
\end{aligned}$$

From now, we will prove that, for each $j = 1, \ldots, m{+}1$,

$$\mathtt{InstVar}(g, T, B, t) \quad \equiv \quad t$$

in the context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$ where

$$g_{m+1} \overset{\mathrm{def}}{=} f, \quad g \overset{\mathrm{def}}{=} g_j, \quad B \overset{\mathrm{def}}{=} B^{t_0}_{b_1}, \quad t \overset{\mathrm{def}}{=} t_j \ .$$

If we prove this, by applying this repeatedly for $j = m{+}1, \ldots, 1$ (in the reverse order), we obtain

$$(f\,(\widetilde{x_i}, \widetilde{g_j}))^{\sharp'_{34}}_{T,B^{t_0}_{b_1}} \ \equiv\ t_1 \ =\ (f\,(\widetilde{x_i}, \widetilde{g_j}))^{\sharp_{34}}_{T} \ .$$

Thus, our goal is to prove

$$(\ \mathtt{InstVar}(g,T,B,t) \quad =\ ) \quad \textbf{let}\ g = \left(\begin{array}{l} \lambda\widetilde{y}.\,\textbf{let}\ x = g\,\widetilde{y}\ \textbf{in} \\ \quad \textbf{let}\ z^{\langle\alpha^1\rangle} = B^{\sharp}_{g}(\alpha^1)\ \textbf{in} \\ \quad \cdots \\ \quad \textbf{let}\ z^{\langle\alpha^q\rangle} = B^{\sharp}_{g}(\alpha^q)\ \textbf{in} \\ \quad \textbf{assume}\,(p)\,;\textbf{assume}\,(p')\,;x \end{array} \right) \textbf{in}\, t \quad \equiv \quad t\,. \tag{D.19}$$

First, we transform the above equation to a more convenient form. By the definition of $\mathtt{InstVar}(-)$, there exist k and z such that $(g = \mathbf{pr}_k z) \in B$. Hence, there exist $(x'_1, \ldots, x'_{n'}, f'_1, \ldots, f'_{m'})$ and b_0 such that

$$z \rightsquigarrow_R (x'_1, \ldots, x'_{n'}, f'_1, \ldots, f'_{m'})^{b_0}, \tag{D.20}$$

and by applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$, we have

$$z \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} (\widetilde{x'_i}, \lambda(y'_1, ..., y'_{m'}).\,(f'_1\,y'_1, ..., f'_{m'}\,y'_{m'}))\ . \tag{D.21}$$

For each $j \le m'$, since f'_j has a function type, there is $\left(\mathbf{fix}(f'_j, \lambda x'_j.\, s''^{b'''_j}_j)\right)^{b''_j}$ such that

$$f'_j \rightsquigarrow_{R|_z} \left(\mathbf{fix}(f'_j, \lambda x'_j.\, s''^{b'''_j}_j)\right)^{b''_j}\ . \tag{D.22}$$

Then, applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$f'_j \rightsquigarrow_{(R|_z)^{\sharp'_{\mathrm{N}}}_{T,t_0}} \big(\mathbf{fix}(f'_j, \lambda x'_j.\, s''_j)\big)^{\sharp'_{34}}_{T,B^{t_0}_{b''_j}} = \mathbf{fix}(f'_j, \lambda(z^j_1, \ldots, z^j_{\mathtt{m}_j}).\,(t^j_1, \ldots, t^j_{\mathtt{m}_j})) \tag{D.23}$$

for some t^j_i. Here, note that, by the definition in Figure 13, each t^j_i does not contain variables $z^j_{i'}$ for $i' \ne i$, and "t^j_i do not depend on i", precisely,

$$t^j_i[z^j_i \mapsto z^j_{i'}] = t^j_{i'}. \tag{D.24}$$

Also note that, as explained at (1) in Step $\sharp_4$ in Section 3.2, t^j_i is of the form

$$t^j_i \quad = \quad \textbf{if}\ z^j_i = \bot\ \textbf{then}\ \bot\ \textbf{else}\ \overline{t}^j_i$$

for some $\overline{t}^j_i$. Hence,

$$u^j_i = \bot \quad \text{implies} \quad t^j_i[z^j_i \mapsto u^j_i] =_{\mathrm{o}} \bot. \tag{D.25}$$

Now, since $(g = \mathbf{pr}^{\to}_k z) \in B$, by Lemma 24 - (4) with (D.20) and (D.22), in s_1,

$$g \rightsquigarrow_R \left(\mathbf{fix}(f'_k, \lambda x'_k.\, s''^{b'''_k}_k)\right)^{b''_k}$$

and by applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$ to s_1,

$$g \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} (\mathbf{fix}(f'_k, \lambda x'_k.\, s''_k))^{\sharp'_{34}}_{T,B^{t_0}_{b''_k}} = \mathbf{fix}(f'_k, \lambda(z^k_1, \ldots, z^k_{\mathtt{m}_k}).\, (t^k_1, \ldots, t^k_{\mathtt{m}_k}))\ . \tag{D.26}$$

Let C be the following context:

$$C \overset{\text{def}}{=} \mathbf{let}\ g = \begin{pmatrix} \lambda\widetilde{y}.\, \mathbf{let}\ x = [\,]\ \mathbf{in} \\ \quad \mathbf{let}\ z^{\langle\alpha^1\rangle} = B^{\sharp}_{g}(\alpha^1)\ \mathbf{in} \\ \quad \cdots \\ \quad \mathbf{let}\ z^{\langle\alpha^q\rangle} = B^{\sharp}_{g}(\alpha^q)\ \mathbf{in} \\ \quad \mathbf{assume}\,(p)\,;\mathbf{assume}\,(p')\,; x \end{pmatrix} \mathbf{in}\ t.$$

Then, in context $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}[C]$,

$$\begin{aligned}
& \big(g\,(y_1, \ldots, y_{\mathtt{m}_k})\big) \\
\equiv\ & \{\text{(RT) on (D.26)}\} \\
& \big(\mathbf{fix}(f'_k, \lambda(z^k_1, \ldots, z^k_{\mathtt{m}_k}).\, (t^k_1, \ldots, t^k_{\mathtt{m}_k}))\,(y_1, \ldots, y_{\mathtt{m}_k})\big) \\
=_{\mathrm{o}}\ & (t^k_1[z^k_1 \mapsto y_1][f'_k \mapsto \mathbf{fix}(f'_k, \lambda(\widetilde{z^k_i}).\, (\widetilde{t^k_i}))], \ldots, t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}][f'_k \mapsto \mathbf{fix}(f'_k, \lambda(\widetilde{z^k_i}).\, (\widetilde{t^k_i}))]) \\
\equiv\ & \{\text{(RT) on (D.23)}\} \\
& (t^k_1[z^k_1 \mapsto y_1], \ldots, t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}]).
\end{aligned} \tag{D.27}$$

Hence, by (D.15), the left hand side of (D.19) is equivalent ($\equiv$ in $(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}$) to the following:

$$t' \overset{\text{def}}{=} \mathbf{let}\ g = \begin{pmatrix} \lambda\widetilde{y}.\, \mathbf{let}\ x_1 = t^k_1[z^k_1 \mapsto y_1]\ \mathbf{in} \\ \quad \cdots \\ \quad \mathbf{let}\ x_{\mathtt{m}_k} = t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}]\ \mathbf{in} \\ \quad \mathbf{let}\ z^{\langle\alpha^1\rangle} = B^{\sharp}_{g}(\alpha^1)\ \mathbf{in} \\ \quad \cdots \\ \quad \mathbf{let}\ z^{\langle\alpha^q\rangle} = B^{\sharp}_{g}(\alpha^q)\ \mathbf{in} \\ \quad \mathbf{assume}\,(p)\,;\mathbf{assume}\,(p')\,; (x_1, \ldots, x_{\mathtt{m}_k}) \end{pmatrix} \mathbf{in}\ t\ . \tag{D.28}$$

From now, we prove that $B^{\sharp}_{g}(\alpha^l)$ ($l = 1, \ldots, q$) in t' can be replaced with values, and then prove that p and p' are satisfied and hence $\mathbf{assume}\,(p)$ and $\mathbf{assume}\,(p')$ can be removed. If these are proved, we can prove (D.19) as follows:

$$\begin{aligned}
& t' \\
\equiv\ & \{\text{to be proved}\} \\
& \mathbf{let}\ g = \begin{pmatrix} \lambda\widetilde{y}.\, \mathbf{let}\ x_1 = t^k_1[z^k_1 \mapsto y_1]\ \mathbf{in} \\ \quad \cdots \\ \quad \mathbf{let}\ x_{\mathtt{m}_k} = t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}]\ \mathbf{in}\ (x_1, \ldots, x_{\mathtt{m}_k}) \end{pmatrix} \mathbf{in}\ t \\
=_{\mathrm{o}}\ & \mathbf{let}\ g = \lambda\widetilde{y}.\, (t^k_1[z^k_1 \mapsto y_1], \ldots, t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}])\ \mathbf{in}\ t \\
\equiv\ & \{\text{the same as (D.27) (context is a bit different, but essentially the same)}\} \\
& \mathbf{let}\ g = \lambda\widetilde{y}.\, g\,\widetilde{y}\ \mathbf{in}\ t \\
=_{\mathrm{o}}\ & t
\end{aligned}$$

Recall that p and p' are defined in Step 5 in Section 4.3 by using $z^{\langle(\alpha_j)_j\rangle}$, and $B^\sharp_g((\alpha_j)_j)$ is defined in Step 4 as

$$B^\sharp_g((\alpha_j)_j) \stackrel{\text{def}}{=} (\mathbf{pr}^\sharp_\rightarrow z)(\mathtt{arg}^*_1\alpha_1, \ldots, \mathtt{arg}^*_m\alpha_m) \ ,$$

where, by (3), $\mathbf{pr}^\sharp_\rightarrow z$ has the following type for some m', τ_j, τ'_j, and $\mathtt{m}_j = T(\mathbf{pr}^\rightarrow_j z)$:

$$\mathbf{pr}^\sharp_\rightarrow z \ : \ \prod_{j=1}^{m'} \left((\tau_j)^{\sharp'_{34}} \right)^{\mathtt{m}_j} \rightarrow \prod_{j=1}^{m'} \left((\tau'_j)^{\sharp'_{34}} \right)^{(\mathtt{m}_j)}_{[\cdot\mapsto\cdot]} .$$

Note that the equality == and the implication => used in p and p' are not genuine logical operators but boolean primitives, so if two terms t_1 and t_2 both diverge (or fail), then t_1 == t_2 is not true but divergence (or fail), and **assume** $(\ldots t_1 == t_2 \ldots)$ cannot necessarily be removed. Thus, it is important to know if such t_1 and t_2 are values or not. In the current case, since $\mathtt{arg}^*_j(\alpha_j)$ and $z^{\langle(\alpha_j)_j\rangle}$ are values, this concern is in fact cleared.

To calculate p and p', we substitute $B^\sharp_g((\alpha_j)_j)$ for $z^{\langle(\alpha_j)_j\rangle}$ in p and p', and to do so we show that $B^\sharp_g((\alpha_j)_j)$ is interchangeable (in the sense of $\equiv$) with a value. From now, we calculate a concrete form of $B^\sharp_g((\alpha_j)_j)$ in the context $(R)^{\sharp'_\mathrm{N}}_{T,t_0}$.

For a given $\alpha = (\alpha_j)_j \in \prod_{j=1}^{m} \mathtt{App}^*_j$, for each j, let α_j and $\mathtt{arg}^*_j(\alpha_j)$ be of the forms $(a^j_i)_{i\le\mathtt{m}_j}$ and $(u^j_i)_{i\le\mathtt{m}_j}$, respectively, where $u^j_i = \mathtt{arg}_j(a^j_i)$ by definition. Note that u^j_i is a variable or $\bot$, and hence is a value. In $(R)^{\sharp'_\mathrm{N}}_{T,t_0}$,

$$\begin{aligned}
B^\sharp_g((\alpha_j)_j) \ &= \ (\mathbf{pr}^\sharp_\rightarrow z)\,((u^1_i)_i, \ldots, (u^m_i)_i) \\
&\equiv \ \{\text{(RT) on (D.21)}\} \\
&\quad \ \Big(\mathbf{pr}^\sharp_\rightarrow \Big(\widetilde{x'_i}, \lambda(y'_1, ..., y'_{m'}).\,(f'_1\, y'_1, ..., f'_{m'}\, y'_{m'})\Big)\Big)\,((u^1_i)_i, \ldots, (u^m_i)_i) \\
&=_\mathrm{o} \ \Big(\lambda(y'_1, ..., y'_{m'}).\,(f'_1\, y'_1, ..., f'_{m'}\, y'_{m'})\Big)\,((u^1_i)_i, \ldots, (u^m_i)_i) \\
&=_\mathrm{o} \ \{(*)\text{ A remark is given below.}\} \\
&\quad \ \big(f'_1\,(u^1_i)_i, \ldots, f'_{m'}\,(u^{m'}_i)_i\big) \\
&\equiv \ \big((t^1_1, \ldots, t^1_{\mathtt{m}_1})[z^1_i \mapsto u^1_i]_i, \ldots, (t^{m'}_1, \ldots, t^{m'}_{\mathtt{m}_{m'}})[z^{m'}_i \mapsto u^{m'}_i]_i\big) \\
&\equiv \ \big((t^1_1[z^1_1 \mapsto u^1_1], \ldots, t^1_{\mathtt{m}_1}[z^1_{\mathtt{m}_1} \mapsto u^1_{\mathtt{m}_1}]), \ldots, (t^{m'}_1[z^{m'}_1 \mapsto u^{m'}_1], \ldots, t^{m'}_{\mathtt{m}_{m'}}[z^{m'}_{\mathtt{m}_{m'}} \mapsto u^{m'}_{\mathtt{m}_{m'}}])\big) \ .
\end{aligned}$$

On the equation marked by $(*)$, as explained at (1) in Step $\sharp_4$ in Section 3.2, the term $f'_j\, y'_j$ in (D.21) is precisely

$$\mathbf{if}\ y'_j = \bot\ \mathbf{then}\ \bot\ \mathbf{else}\ f'_j\, y'_j.$$

However, now the j-th argument $(u^j_i)_i$ is a tuple and hence not $\bot$; hence, the above is equivalent to $f'_j\, y'_j$ in this case.

For each $j \le m$, and $i \le \mathtt{m}_j$, we define a value $V^\alpha_{j,i}$ as the following.

$$V^\alpha_{j,i} \ \stackrel{\text{def}}{=} \ \begin{cases} \bot & (a^j_i = (\bot, v)) \\ x_l & (a^j_i = y_l) \\ w & (a^j_i = (u, v, w)) \end{cases}$$

From now, we show that

$$t^j_i[z^j_i \mapsto u^j_i] \ \equiv \ V^\alpha_{j,i}$$

by a case analysis on a^j_i. Let t'' be a term obtained by replacing every $B^\sharp_g(\alpha)$ in t' (given in (D.28)) with a matrix $((t^j_i[z^j_i \mapsto u^j_i])_i)_j$, and let C' be a context obtained by replacing one component of one of the matrices in t'' with $[\,]$. In the following case analysis, we use $\equiv$ in this context $(R)^{\sharp'_\mathrm{N}}_{T,t_0}[C']$ if we do not mention contexts.

$\boxed{a_i^j = (\bot, v)}$ This case is trivial: since now $u_i^j = \mathtt{arg}_j(a_i^j) = \bot$, by (D.25),

$$t_i^j[z_i^j \mapsto u_i^j] =_{\mathrm{o}} \bot = V_{j,i}^{\alpha} \ .$$

$\boxed{a_i^j = y_l \ (l \leq \mathtt{m}_j)}$ In this case, j must be k, and $u_i^k = \mathtt{arg}_k(a_i^k) = y_l$. Then,

$$\begin{aligned} t_i^k[z_i^k \mapsto y_l] &= \{\text{by (D.24)}\} \\ &\quad t_l^k[z_l^k \mapsto y_l] \\ &\equiv \{\text{(RT)}\} \\ &\quad x_l \\ &= V_{j,i}^{\alpha} \ . \end{aligned}$$

$\boxed{a_i^j = (u, v, w)}$ This case is long. In this case, by the definition of $\mathtt{App}_j'$ in Step 2 in Section 4.3, we have

$$(v = \mathbf{pr}_j^{\rightarrow} z), (w = v\, u) \in B, \qquad \mathit{depth}(v) = 1 \ .$$

Since $(v = \mathbf{pr}_j^{\rightarrow} z) \in B$, by Lemma 24 - (4) with (D.20) and (D.22), in s_1,

$$v \rightsquigarrow_R \left(\mathbf{fix}(f_j', \lambda x_j'.\, {s_j''}^{b_j'''})\right)^{b_j''}$$

and by applying $(-)_{T,t_0}^{\sharp_{\mathrm{N}}'}$ to s_1,

$$v \rightsquigarrow_{(R)_{T,t_0}^{\sharp_{\mathrm{N}}'}} \left(\mathbf{fix}(f_j', \lambda x_j'.\, s_j'')\right)_{T,B_{b_j''}^{t_0}}^{\sharp_{34}'} = \mathbf{fix}(f_j', \lambda(z_1^j, \ldots, z_{\mathtt{m}_j}^j).\, (t_1^j, \ldots, t_{\mathtt{m}_j}^j)) \ . \tag{D.29}$$

Next, since $(w = v\, u) \in B$, by Lemma 24 - (4), there exist b_1', b_2', s_0', and R' such that

$$\begin{aligned} (t_0)^{\mathrm{lb}} &\longrightarrow_{\mathrm{N}}^* R'[\mathbf{let}\ w = (v\, u)^{b_1'}\ \mathbf{in}\ {s_0'}^{b_2'}] \\ &\longrightarrow_{\mathrm{N}} R'[\mathbf{cclet}\ w = (s_j''[x_j' \mapsto u][f_j' \mapsto v])^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\ &\longrightarrow_{\mathrm{N}}^* s_1 \end{aligned}$$

By induction hypothesis, in $(R')_{T,t_0}^{\sharp_{\mathrm{N}}'}$,

$$(v\, u)_T^{\sharp_{34}} \equiv \left(s_j''[x_j' \mapsto u][f_j' \mapsto v]\right)_{T,t_0}^{\sharp_{\mathrm{N}}'} \ .$$

Now, given the following situation

$$R'[s_j''[x_j' \mapsto u][f_j' \mapsto v]] \longrightarrow_{\mathrm{N}}^* R'[R_0[r]] \longrightarrow_{\mathrm{N}} R'[R_0[s_r]],$$

where r is the redex of $R'[R_0[r]]$, we show that, in $(R')_{T,t_0}^{\sharp_{\mathrm{N}}'}$,

$$(R_0[r])_{T,t_0}^{\sharp_{\mathrm{N}}'} \equiv (R_0[s_r])_{T,t_0}^{\sharp_{\mathrm{N}}'} \ .$$

If this is proved, for the N-normal form $R'[R''[\widetilde{z'}]]$ of $R'[s_j''[x_j' \mapsto u][f_j' \mapsto v]]$, we have

$$(v\, u)_T^{\sharp_{34}} \equiv \left(R''[\widetilde{z'}]\right)_{T,t_0}^{\sharp_{\mathrm{N}}'} \tag{D.30}$$

in $(R')^{\sharp'_{\mathrm{N}}}_{T,t_0}$.

If r is of the form $\mathbf{if}\ x^r\ \mathbf{then}\ {s_1^r}^{b_1^r}\ \mathbf{else}\ {s_2^r}^{b_2^r}$, and if

$$x^r \rightsquigarrow_{R'[R_0]} \mathbf{true}$$

(the case of $\mathbf{false}$ is similar), in $(R'[R_0])^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$\begin{aligned}
(r)^{\sharp'_{\mathrm{N}}}_{T,t_0} &= \mathbf{if}\ x^r\ \mathbf{then}\ (s_1^r)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \mathbf{else}\ (s_2^r)^{\sharp'_{\mathrm{N}}}_{T,t_0} \\
&\equiv \Big\{\text{by (RT), applying } (-)^{\sharp'_{\mathrm{N}}}_{T,t_0} \text{ to } x^r \rightsquigarrow_{R'[R_0]} \mathbf{true}\Big\} \\
&\quad\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ (s_1^r)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ \mathbf{else}\ (s_2^r)^{\sharp'_{\mathrm{N}}}_{T,t_0} \\
&=_{\mathrm{o}} (s_1^r)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ .
\end{aligned}$$

If r is of the form $\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}$ where d^r is of the form $\mathsf{op}(\widetilde{x^r})$ or $\mathbf{pr}_i x^r$, there exists ${U^r}^{b^r}$ such that

$$R'[R_0[\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]] \longrightarrow_{\mathrm{N}} R'[R_0[\mathbf{let}\ y^r = {U^r}^{b^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]]$$

and

$$\begin{aligned}
(t_0)^{\mathrm{lb}} &\longrightarrow^{*}_{\mathrm{N}}\ R'[\mathbf{cclet}\ w = (s''_j[x'_j \mapsto u][f'_j \mapsto v])^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2}] \\
&\longrightarrow^{n'}_{\mathrm{N}}\ R'[\mathbf{cclet}\ w = R_0[\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2}] \\
&\left(\begin{aligned}
&= \{\text{by Lemma 22 - (1)}\} \\
&\quad\ R'[R_0[\mathbf{cclet}\ w = (\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r})^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2}]] \\
&= R'[R_0[\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = {s_2^r}^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2})^{b_2^r}]]
\end{aligned}\right) \\
&\longrightarrow_{\mathrm{N}}\ \{\text{by Lemma 22 - (3)}\} \\
&\qquad R'[\mathbf{cclet}\ w = R_0[\mathbf{let}\ y^r = {U^r}^{b^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2}] \\
&\quad\ =\ \{\text{by Lemma 22 - (1)}\} \\
&\qquad R'[R_0[\mathbf{let}\ y^r = {U^r}^{b^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = {s_2^r}^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2})^{b_2^r}]].
\end{aligned}$$

Since

$$\begin{aligned}
&R'[R_0[\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = {s_2^r}^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2})^{b_2^r}]] \\
\longrightarrow_{\mathrm{N}}\ &R'[R_0[\mathbf{let}\ y^r = {U^r}^{b^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = {s_2^r}^{b'''_j}\ \mathbf{in}\ {s'_0}^{b'_2})^{b_2^r}]],
\end{aligned}$$

by induction hypothesis, in $(R'[R_0])^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$(d^r)^{\sharp'_{34}}_{T,B^{t_0}_{b_1^r}} \equiv (U^r)^{\sharp'_{34}}_{T,B^{t_0}_{b^r}}\ ,$$

and by Lemma 26, in $(R'[R_0])^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$\Big(\mathbf{let}\ y^r = {d^r}^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}\Big)^{\sharp'_{\mathrm{N}}}_{T,t_0} \equiv \Big(\mathbf{let}\ y^r = {U^r}^{b^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}\Big)^{\sharp'_{\mathrm{N}}}_{T,t_0}\ .$$

If r is of the form $\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}$, there exists ${s^r}^{b^r}$ such that

$$R'[R_0[\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]] \longrightarrow_{\mathrm{N}} R'[R_0[\mathbf{cclet}\ y^r = {s^r}^{b^r}\ \mathbf{in}\ {s_2^r}^{b_2^r}]]$$

and

$$
\begin{aligned}
(t_0)^{\mathrm{lb}} &\longrightarrow_{\mathrm{N}}^{*} R'[\mathbf{cclet}\ w = (s_j''[x_j' \mapsto u][f_j' \mapsto v])^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\
&\longrightarrow_{\mathrm{N}}^{n'} R'[\mathbf{cclet}\ w = R_0[\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ s_2^{r\,b_2^r}]^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\
&\left(\begin{aligned} = \ &\{\text{by Lemma 22 - (1)}\} \\ &R'[R_0[\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = s_2^{r\,b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'})^{b_2^r}]]\end{aligned}\right) \\
&\longrightarrow_{\mathrm{N}} \{\text{by Lemma 22 - (3)}\} \\
&\qquad R'[\mathbf{cclet}\ w = R_0[\mathbf{cclet}\ y^r = s^{r\,b^r}\ \mathbf{in}\ s_2^{r\,b_2^r}]^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\
&\quad = R'[R_0[\mathbf{cclet}\ w = (\mathbf{cclet}\ y^r = s^{r\,b^r}\ \mathbf{in}\ s_2^{r\,b_2^r})^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}]] \\
&\quad = \{\text{by Lemma 22 - (2)}\} \\
&\qquad R'[R_0[\mathbf{cclet}\ y^r = s^{r\,b^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = s_2^{r\,b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'})^{b_2^r}]].
\end{aligned}
$$

Since

$$
\begin{aligned}
&R'[R_0[\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = s_2^{r\,b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'})^{b_2^r}]] \\
\longrightarrow_{\mathrm{N}}\ &R'[R_0[\mathbf{cclet}\ y^r = s^{r\,b^r}\ \mathbf{in}\ (\mathbf{cclet}\ w = s_2^{r\,b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'})^{b_2^r}]],
\end{aligned}
$$

by induction hypothesis, in $(R'[R_0])^{\sharp_{\mathrm{N}}'}_{T,t_0}$,

$$
\left(f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r})\right)^{\sharp_{34}'}_{T,B^{t_0}_{b_1^r}} \equiv (s^r)^{\sharp_{\mathrm{N}}'}_{T,t_0}\ ,
$$

and by Lemma 26, in $(R'[R_0])^{\sharp_{\mathrm{N}}'}_{T,t_0}$,

$$
\left(\mathbf{let}\ y^r = (f^r\ (\widetilde{x_i^r}, \widetilde{g_j^r}))^{b_1^r}\ \mathbf{in}\ s_2^{r\,b_2^r}\right)^{\sharp_{\mathrm{N}}'}_{T,t_0} \equiv \left(\mathbf{cclet}\ y^r = s^{r\,b^r}\ \mathbf{in}\ s_2^{r\,b_2^r}\right)^{\sharp_{\mathrm{N}}'}_{T,t_0}\ .
$$

In $(R')^{\sharp_{\mathrm{N}}'}_{T,t_0}$,

$$
\begin{aligned}
&(R'')^{\sharp_{\mathrm{N}}'}_{T,t_0}[(v\,u)^{\sharp_{34}}_{T}] \\
\equiv\ &\{\text{by (D.30) and Lemma 26}\} \\
&(R'')^{\sharp_{\mathrm{N}}'}_{T,t_0}[(R'')^{\sharp_{\mathrm{N}}'}_{T,t_0}[(\widetilde{z'})^{\sharp_{\mathrm{N}}'}_{T,t_0}]] \\
=_{\mathrm{o}}\ &\{\text{by } \beta\text{-conversion}\} \\
&(R'')^{\sharp_{\mathrm{N}}'}_{T,t_0}[(\widetilde{z'})^{\sharp_{\mathrm{N}}'}_{T,t_0}].
\end{aligned}
$$

Since $R = R'[R''[R''']]$ for some R''', by Lemma 26, in $(R)^{\sharp_{\mathrm{N}}'}_{T,t_0}[C']$

$$
(v\,u)^{\sharp_{34}}_{T} \quad \equiv \quad (\widetilde{z'})^{\sharp_{\mathrm{N}}'}_{T,t_0}\ . \tag{D.31}
$$

Now, by Lemma 22 - (3), we have

$$
\begin{aligned}
(t_0)^{\mathrm{lb}} &\longrightarrow_{\mathrm{N}}^{*} R'[\mathbf{cclet}\ w = (s_j''[x_j' \mapsto u][f_j' \mapsto v])^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\
&\longrightarrow_{\mathrm{N}}^{*} R'[\mathbf{cclet}\ w = R''[\widetilde{z'}]^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}] \\
&\quad = R'[R''[\mathbf{let}\ w = \widetilde{z'}^{b_j'''}\ \mathbf{in}\ {s_0'}^{b_2'}]] \\
&\longrightarrow_{\mathrm{N}}^{*} s_1.
\end{aligned}
$$

Hence,

$$w \rightsquigarrow_R \widetilde{z'}^{b'''_j}\,.$$

and by applying $(-)^{\sharp'_{\mathrm{N}}}_{T,t_0}$,

$$w \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} \big(\widetilde{z'}\big)^{\sharp'_{\mathrm{N}}}_{T,t_0}\,. \tag{D.32}$$

Finally, we have

$$\begin{aligned}
& t^j_i[z^j_i \mapsto u^j_i] \\
={} & \Big\{\text{since now } u^j_i = \mathtt{arg}_j(a^j_i) = u\Big\} \\
& t^j_i[z^j_i \mapsto u] \\
\equiv{} & \{\text{by Lemma 25 - (1)}\} \\
& \mathbf{pr}_1(t^j_i[z^j_i \mapsto u], \ldots, t^j_i[z^j_i \mapsto u]) \\
={} & \{\text{by (D.24)}\} \\
& \mathbf{pr}_1(t^j_1[z^j_1 \mapsto u], \ldots, t^j_{\mathtt{m}_j}[z^j_{\mathtt{m}_j} \mapsto u]) \\
\equiv{} & \{\text{(RT) on (D.23)}\} \\
& \mathbf{pr}_1(t^j_1[z^j_1 \mapsto u][f'_j \mapsto \mathbf{fix}(f'_j, \lambda(\widetilde{z^j_i}).\,(\widetilde{t^j_i}))], \ldots, t^j_{\mathtt{m}_j}[z^j_{\mathtt{m}_j} \mapsto u][f'_j \mapsto \mathbf{fix}(f'_j, \lambda(\widetilde{z^j_i}).\,(\widetilde{t^j_i}))]) \\
\equiv{} & \mathbf{pr}_1\big(\mathbf{fix}(f'_j, \lambda(z^j_1, \ldots, z^j_{\mathtt{m}_j}).\,(t^j_1, \ldots, t^j_{\mathtt{m}_j}))\,(\overrightarrow{u}^{\mathtt{m}_j})\big) \\
\equiv{} & \{\text{(RT) on (D.29)}\} \\
& \mathbf{pr}_1\big(v\,(\overrightarrow{u}^{\mathtt{m}_j})\big) \\
={} & (v\,u)^{\sharp_{34}}_T \\
\equiv{} & \{\text{by (D.31)}\} \\
& \big(\widetilde{z'}\big)^{\sharp'_{\mathrm{N}}}_{T,t_0} \\
\equiv{} & \{\text{by (D.32)}\} \\
& w \\
={} & V^\alpha_{j,i}\,.
\end{aligned}$$

Note that, from the above, we have

$$V^\alpha_{j,i} \quad\equiv\quad \mathbf{pr}_1\big(v\,(\overrightarrow{u}^{\mathtt{m}_j})\big) \quad\equiv\quad t^j_i[z^j_i \mapsto u]. \tag{D.33}$$

Thus, we have proved that $B^\sharp_g(\alpha)$ ($\equiv ((t^j_i[z^j_i \mapsto u^j_i])_i)_j$) can be replaced with the value $((V^\alpha_{j,i})_i)_j$ and we can substitute $((V^{\alpha^l}_{j,i})_i)_j$ for $z^{\langle\alpha^l_j\rangle_j\rangle}$ in **assume** (p) and **assume** (p'). Finally, we prove that p and p' (after the substitution) are satisfied.

On **assume** (p), we show that for $\alpha = (\alpha_j)_j, \alpha' = (\alpha'_j)_j \in \prod_{j \le m'} \mathtt{App}^*_j$ and $j \le m'$ such that $\mathtt{fun}^*_j(\alpha_j) = \mathtt{fun}^*_j(\alpha'_j)$,

$$\Big(\mathtt{arg}^*_j(\alpha_j) == \mathtt{arg}^*_j(\alpha'_j) \;\texttt{=>}\; (V^\alpha_{j,i})_i == (V^{\alpha'}_{j,i})_i\Big) \quad\equiv\quad \mathbf{true}.$$

Suppose that $\mathtt{arg}^*_j(\alpha_j)$, $\mathtt{arg}^*_j(\alpha'_j)$, α, and α' are of the following forms:

$$\mathtt{arg}^*_j(\alpha_j) = (u^j_i)_i \qquad \mathtt{arg}^*_j(\alpha'_j) = (u'^j_i)_i \qquad \alpha = ((a^j_i)_i)_j \qquad \alpha' = ((a'^j_i)_i)_j\,.$$

It is enough to show that for each i,

$$\Big(u^j_i == u'^j_i \;\texttt{=>}\; V^\alpha_{j,i} == V^{\alpha'}_{j,i}\Big) \quad\equiv\quad \mathbf{true}.$$

We show this by a case analysis on a^j_i and a'^j_i.

$\boxed{a^j_i = (\bot, v)\ (a'^j_i \text{ is arbitrary})}$ If $u'^j_i \neq \bot$, $(u^j_i == u'^j_i) =_o \mathbf{false}$.
If $u'^j_i = \bot$, $V^{\alpha}_{j,i} = \bot = V^{\alpha'}_{j,i}$, and $(V^{\alpha}_{j,i} == V^{\alpha'}_{j,i}) =_o \mathbf{true}$.

$\boxed{a^j_i = y_l,\ a'^j_i = y_{l'}}$ In this case, $j = k$, and we have

$$\begin{aligned}
& (u^k_i == u'^k_i \ \texttt{=>}\ V^{\alpha}_{k,i} == V^{\alpha'}_{k,i}) \\
= \ & (y_l == y_{l'} \ \texttt{=>}\ x_l == x_{l'}) \\
\equiv \ & \{\text{by (RT) on } x_l \text{ and } x_{l'}\} \\
& (y_l == y_{l'} \ \texttt{=>}\ t^k_l[z^k_l \mapsto y_l] == t^k_{l'}[z^k_{l'} \mapsto y_{l'}]) \\
=_o \ & \mathbf{true},
\end{aligned}$$

where the last equation is shown as follows.

Let C'' be the context obtained by replacing the occurrence of

$$(u^j_i == u'^j_i \ \texttt{=>}\ V^{\alpha}_{j,i} == V^{\alpha'}_{j,i})$$

in

$$\left(\begin{array}{l}
\mathbf{let}\ x_1 = t^k_1[z^k_1 \mapsto y_1]\ \mathbf{in} \\
\cdots \\
\mathbf{let}\ x_{\mathtt{m}_k} = t^k_{\mathtt{m}_k}[z^k_{\mathtt{m}_k} \mapsto y_{\mathtt{m}_k}]\ \mathbf{in} \\
\mathbf{assume}\left(p[z^{\langle \alpha^l \rangle} \mapsto ((V^{\alpha^l}_{j,i})_i)_j]_{l \le q}\right); \\
\mathbf{assume}\left(p'[z^{\langle \alpha^l \rangle} \mapsto ((V^{\alpha'^l}_{j,i})_i)_j]_{l \le q}\right); (x_1, \ldots, x_{\mathtt{m}_k})
\end{array}\right)$$

with $[\,]$. From now we show that

$$\lambda \widetilde{y}.\, C''[(y_l == y_{l'} \ \texttt{=>}\ t^k_l[z^k_l \mapsto y_l] == t^k_{l'}[z^k_{l'} \mapsto y_{l'}])] \quad =_o \quad \lambda \widetilde{y}.\, C''[\mathbf{true}].$$

By extensionality of the observational equivalence, it is enough to show that for any closed values $\widetilde{V}$,

$$C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_{l'} \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_{l'}[z^k_{l'} \mapsto V_{l'}])] \quad =_o \quad C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[\mathbf{true}].$$

Note that, by the definition of $\mathtt{App}_j$, $\mathit{depth}(g) = 1$, and hence V_i are ground values. If $V_l \neq V_{l'}$, then

$$(V_l == V_{l'}) \quad =_o \quad \mathbf{false}$$

and hence

$$(V_l == V_{l'} \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_{l'}[z^k_{l'} \mapsto V_{l'}]) \quad =_o \quad \mathbf{true}.$$

If $V_l = V_{l'}$, then

$$\begin{aligned}
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_{l'} \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_{l'}[z^k_{l'} \mapsto V_{l'}])] \\
= \ & C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_{l'}[z^k_{l'} \mapsto V_l])] \\
= \ & \{\text{by (D.24)}\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_l[z^k_l \mapsto V_l])] \\
=_o \ & \{\text{(RT) on } x_l\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ x_l == x_l)] \\
=_o \ & \{(x_l == x_l) =_o \mathbf{true} \text{ as explained below.}\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ \mathbf{true})] \\
=_o \ & C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[\mathbf{true}].
\end{aligned}$$

Above, $(x_l == x_l) =_{\mathrm{o}} \mathbf{true}$ is obtained by the extensionality of the observational equivalence, as $(V == V) =_{\mathrm{o}} \mathbf{true}$ for any closed value V.

$\boxed{a^j_i = (u, v, w),\ a'^j_i = y_l}$ This case is almost the same as the previous case. In this case, $j = k$, and we have

$$\begin{aligned}
& (u^k_i == u'^k_i \ \texttt{=>}\ V^{\alpha}_{k,i} == V^{\alpha'}_{k,i}) \\
= \ & (u == y_l \ \texttt{=>}\ w == x_l) \\
\equiv \ & \{\text{by (D.33) and by (RT) on } x_l\} \\
& (u == y_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto u] == t^k_l[z^k_l \mapsto y_l]) \\
=_{\mathrm{o}} \ & \mathbf{true},
\end{aligned}$$

where the last equation is shown as follows.

Let C'' be the context defined in the same way as above. From now we show that

$$\lambda \widetilde{y}.\, C''[(u == y_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto u] == t^k_l[z^k_l \mapsto y_l])] \quad =_{\mathrm{o}} \quad \lambda \widetilde{y}.\, C''[\mathbf{true}].$$

Similarly to the previous case, it is enough to show that for any values $\widetilde{V}$,

$$C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(u == V_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto u] == t^k_l[z^k_l \mapsto V_l])] \quad =_{\mathrm{o}} \quad C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[\mathbf{true}].$$

If $u \neq V_l$, then $(u == V_l) =_{\mathrm{o}} \mathbf{false}$ and hence

$$(u == V_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto u] == t^k_l[z^k_l \mapsto V_l]) \quad =_{\mathrm{o}} \quad \mathbf{true}.$$

If $u = V_l$, then

$$\begin{aligned}
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(u == V_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto u] == t^k_l[z^k_l \mapsto V_l])] \\
= \ & C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ t^k_i[z^k_i \mapsto V_l] == t^k_l[z^k_l \mapsto V_l])] \\
= \ & \{\text{by (D.24)}\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ t^k_l[z^k_l \mapsto V_l] == t^k_l[z^k_l \mapsto V_l])] \\
=_{\mathrm{o}} \ & \{\text{(RT) on } x_l\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[(V_l == V_l \ \texttt{=>}\ x_l == x_l)] \\
=_{\mathrm{o}} \ & \{\text{similarly to the previous case}\} \\
& C''[y_i \mapsto V_i]_{i \le \mathtt{m}_k}[\mathbf{true}].
\end{aligned}$$

$\boxed{a^j_i = (u, v, w),\ a'^j_i = (u', v', w')}$ Since $\mathtt{fun}^*_j(\alpha_j) = \mathtt{fun}^*_j(\alpha'_j)$, $v = v'$. Now there exist ground values V and V' such that

$$\begin{aligned}
u & \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} V \\
u' & \rightsquigarrow_{(R)^{\sharp'_{\mathrm{N}}}_{T,t_0}} V'\ .
\end{aligned} \tag{D.34}$$

Then,

$$\begin{aligned}
& (u^k_i == u'^k_i \ \texttt{=>}\ V^{\alpha}_{k,i} == V^{\alpha'}_{k,i}) \\
= \ & (u == u' \ \texttt{=>}\ w == w') \\
\equiv \ & \{\text{by (D.33)}\} \\
& (u == u' \ \texttt{=>}\ \mathbf{pr}_1\big(v\,(\overrightarrow{u}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{u'}^{\mathtt{m}_j})\big)) \\
\equiv \ & \{\text{(RT) on (D.34)}\} \\
& (V == V' \ \texttt{=>}\ \mathbf{pr}_1\big(v\,(\overrightarrow{V}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{V'}^{\mathtt{m}_j})\big)).
\end{aligned}$$

If $V \neq V'$, then $(V == V') =_{\mathrm{o}}$ **false** and hence

$$\begin{aligned}
&\big(V == V' \;\; \texttt{=>} \;\; \mathbf{pr}_1\big(v\,(\overrightarrow{V}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{V'}^{\mathtt{m}_j})\big)\big) \\
=_{\mathrm{o}}\;& \mathbf{true}.
\end{aligned}$$

If $V = V'$, then

$$\begin{aligned}
& \big(V == V' \;\; \texttt{=>} \;\; \mathbf{pr}_1\big(v\,(\overrightarrow{V}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{V'}^{\mathtt{m}_j})\big)\big) \\
=\;& \big(V == V \;\; \texttt{=>} \;\; \mathbf{pr}_1\big(v\,(\overrightarrow{V}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{V}^{\mathtt{m}_j})\big)\big) \\
\equiv\;& \{\text{(RT) on (D.34)}\} \\
& \big(V == V \;\; \texttt{=>} \;\; \mathbf{pr}_1\big(v\,(\overrightarrow{u}^{\mathtt{m}_j})\big) == \mathbf{pr}_1\big(v\,(\overrightarrow{u}^{\mathtt{m}_j})\big)\big) \\
\equiv\;& \{\text{by (D.33)}\} \\
& (V == V \;\; \texttt{=>} \;\; w == w) \\
=_{\mathrm{o}}\;& \Big\{\text{similarly to the case that } a^j_i = y_l \text{ and } {a'}^j_i = y_{l'}\Big\} \\
& \mathbf{true}.
\end{aligned}$$

By the symmetry of α and α', now all the cases have been completed.

On $\mathbf{assume}\,(p')$, we show that for $\alpha = (\alpha_j)_j \in \prod_{j \leq m} \mathtt{App}^*_j,\ j \leq m,\ i \leq \mathtt{m}_j,$ and u, v, w such that $\pi_i \alpha_j = (u, v, w)$,

$$(w == V^{\alpha}_{j,i}) \quad \equiv \quad \mathbf{true}.$$

This is trivial: now $V^{\alpha}_{j,i} = w$, and hence

$$(w == V^{\alpha}_{j,i}) \quad = \quad (w == w) \quad =_{\mathrm{o}} \quad \mathbf{true}.$$

□