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 require a hint [20], however. Automated verification techniques are, however, based on first-order refinement types, hence unable to verify certain properties of functions (such as the equality of two recursive functions and the monotonicity of a function, which we call relational properties). To relax this limitation, we introduce a restricted form of higher-order refinement types where refinement predicates can refer to functions, and formalize a systematic program transformation to reduce type checking/inference for higher-order refinement types to that for first-order refinement types, so that the latter can be automatically solved by using an existing software model checker. We also prove the soundness of the transformation, and report on preliminary implementation and experiments.

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

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

1. Introduction

There has been much progress in automated verification techniques for higher-order functional programs [10, 12–14, 17, 18, 20]. 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:

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

Using the existing techniques [10, 13, 17, 18], one can verify that sum has the first-order refinement type: \( n : \text{int} \rightarrow \{ m : \text{int} | m \geq n \} \), which means that sum \( n \) returns a value no less than \( n \). Here, \( \{ m : \text{int} | P(m) \} \) 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:

```plaintext
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 sum2\( (n) \) equals sum\( (n) \) for every integer \( n \). With general refinement types [7], that would amount to checking that sumacc and sum2 have the following types:

\[
\text{sumacc} : (n : \text{int}) \rightarrow (m : \text{int}) \rightarrow \{ r : \text{int} | r = m + \text{sum}(n) \}
\]

\[
\text{sum2} : (n : \text{int}) \rightarrow \{ r : \text{int} | r = \text{sum}(n) \}
\]

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

\[
\text{sum, sum2} : (n : \text{int}) \rightarrow \{ r : \text{int} | \begin{array}{l}
(n \geq 0 \land r = n(n+1)/2) \lor (n < 0 \land r = 0) \end{array} \}
\]

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 \( \text{sum} \) satisfies \( \text{sum} x = x + \text{sum} (x-1) \) for every \( x \geq 0 \). This is almost the definition of \( \text{sum} \) function, and it can be expressed and verified using the general refinement type:

\[
\text{sum} : \{ f : \text{int} \rightarrow \text{int} | \forall x. x \geq 0 \Rightarrow f(x) = x + f(x-1) \}
\]

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

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

```plaintext
append : (x : int \rightarrow \text{int option}) \rightarrow \{ y : \text{int} \rightarrow \text{int option} | y(0) = \text{None} \rightarrow \{ r : \text{int} \rightarrow \text{int option} | r = x \}
```

1 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.

2 Another way would be to use uninterpreted function symbols, but for that purpose, one would first need to check that \( \text{sum} \) is total.
let nil i = None in
let tl xs = fun i-> xs(i+1) in
let cons x xs =
  fun i -> if i=0 then Some(x) else xs(i-1) in
let rec append xs ys =
  match xs(0) with None -> ys
  | Some(x) -> let xs' = tl xs in
    cons x (append xs' ys)

Figure 1. Append function for functional encoding of lists

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

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

\[
\text{sum2: } (n : \text{int}) \rightarrow \{r : \text{int} \mid r = \text{sum}(n)\}
\]

\[
\text{append: } (x : \text{int} \rightarrow \text{int option}) \rightarrow \\
\{y : \text{int} \rightarrow \text{int option} \mid y(0) = \text{None} \rightarrow \\
\{r : \text{int} \rightarrow \text{int option} \mid \forall i. r(i) = x(i)\}.
\]

To take advantage of the recent advance of verification techniques based on first-order refinement types, however, we employ automated program transformation, so that the resulting program can be verified by using only first-order refinement types. The key idea of the transformation is to apply a kind of tupling transformation [5] to capture the relationship between two (or more) function calls at the level of first-order refinement. For example, for the \( \text{sum} \) program above, one can apply the standard tupling transformation (to combine two functions \( \text{sum} \) and \( \text{sum2} \) into one) and obtain:

\[
\text{let rec sum_sumacc } (n, m) = \\
\text{if } n<0 \text{ then } (0, m) \text{ else } \\
\text{let } (x1, x2) = \text{sum_sumacc } (n-1, m+n) \text{ in } \\
(x1+n, x2)
\]

Checking the equivalence of \( \text{sum} \) and \( \text{sum2} \) then amounts to checking that \( \text{sum_sumacc} \) has the following first-order refinement type:

\[
((n, m) : \text{int} \times \text{int}) \rightarrow \{((r1, r2) : \text{int} \times \text{int} \mid r2 = r_1 + m\}.
\]

The transformation for \( \text{append} \) 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:

\[
\text{let append2 } xs ys = (xs, \text{ append } xs \text{ ys}).
\]

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

\[
\text{let append3 } xs ys (i, j) = \\
(xs(i), \text{ append } xs \text{ ys } j).
\]

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

\[
(x : \text{int} \rightarrow \text{int option}) \rightarrow \\
\{y : (x : \text{int}) \rightarrow \{r : \text{int option} \mid x = 0 \Rightarrow r = \text{None}\}) \rightarrow \\
\{(i, j) : \text{int} \times \text{int} \rightarrow \{(r1, r2) : \text{int} \times \text{int} \mid i = j \Rightarrow r1 = r2\}.
\]

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 \( \text{append} \), we need information about the relation between the two function calls \( \text{xs}(i) \) and \( \text{append} \) \( \text{xs} \) \( \text{ys} \) \( j \), 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 \( \text{xs} \) and \( \text{append} \). To avoid that limitation, we further transform the program, by inlining \( \text{append} \) and tupling the two calls of the body of \( \text{append3} \):

\[
\text{let append4 } xs ys (i, j) = \\
\text{match xs(0) with None -> nil2 } (i, j) \\
\mid \text{ Some(x) } \rightarrow \\
\text{ let xs' = tl xs in } \\
\text{ let xszs' = append4 xs' ys in } \\
\text{ let xszs'' = cons2 x xszs' in } \\
xszs'' \ (i, j)
\]

Here, \( \text{nil2} \) and \( \text{cons2} \) are respectively tupled versions of \( \text{nil} \) and \( \text{cons} \), \( \text{var} \) and \( \text{var} \)’s, where \( \text{xszs‘} \) is a tupled one of \( \text{xs‘} \) and \( \text{zs‘} \).

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

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

The rest of the paper is organized as follows. Section 2 introduces the source language. Section 3 presents the basic transformation for auxiliary functions) by applying the above transformations and calling MoCHi as a backend. Section 4 (which roughly corresponds to the transformation from \( \text{append3} \) to \( \text{append4} \) above). Section 5 reports on experiments and Section 6 discusses related work. We conclude the paper in Section 7.

2. Source Language

This section formalizes the source language and the verification problem.

2.1 Source Language

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

\[
t \text{ (terms) } ::= x \mid n \mid \text{op}(t_1, \ldots, t_n) \mid \text{if } t \text{ then } t_1 \text{ else } t_2 \mid \text{fix}(f, \text{Ax. } t) \mid t_1 t_2 \mid (t_1, \ldots, t_n) \mid \text{pr } t \text{ fail}
\]

We use meta-variables \( x, y, z, \ldots, f, g, h, \ldots, \) and \( \nu \) for variables. We have only integers as base values, which are denoted by the meta-variable \( n \). The term \( \text{op}(t) \) (where \( t \) denotes a sequence of expressions) applies the primitive operation \( \text{op} \) on integers to \( t \). We assume that we have the equality operator \( = \) as a primitive operation. We express Booleans by integers, and write \( \text{true} \) for...
1, and false for 0. The term \( \text{fix}(f, \lambda x. t) \) denotes the recursive function defined by \( f = \lambda x.t \). When \( f \) does not occur in \( t \), we write \( \lambda x.t \) for \( \text{fix}(f, \lambda x. t) \). The term \( t_1 t_2 \) applies the function \( t_1 \) to \( t_2 \). We write \( \lambda x = t \in t' \) for \( (\lambda x.t')t \), and write also \( t = t' \) for when \( x \) does not occur in \( t' \). The terms \( (t_1 \ldots t_n) \) and \( pr \) recognize and destroy tuples. The special term fail aborts the execution. It is typically used to express assertions; assert(t), which asserts that \( t \) should evaluate to \text{true}, is expressed by \( t \) \text{then} \text{true} \text{else} \text{false} \). We call a closed term a program. We often write \( t \) for a sequence \( t_1 \ldots t_n \).

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

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

We express the specification of a program by using refinement types.

The syntax of refinement types is given by:

\[
\begin{align*}
\tau (\text{types}) & ::= \rho \mid \{ \nu : \prod_{i=1}^{n} (x_i : \rho_i) \mid P \} \\
\rho (\text{non-tuple types}) & ::= \nu : \text{int} \mid P \mid \{ \nu : (x : \tau_1) \rightarrow \tau_2 \mid P \} \\
\text{P predicates} & ::= t \mid P \land P \mid \forall x.P \\
\end{align*}
\]

where we have used a notational convention \( \prod_{i=1}^{n} (x_i : \rho_i) \) to denote \( (x_1 : \rho_1) \times \cdots \times (x_{n-1} : \rho_{n-1}) \times \rho_n \) (thus, the variable \( x_n \) actually does not occur). The type \( (x_1 : \rho_1) \times \rho_2 \) is a dependent sum type, where \( x \) may occur in \( \rho_2 \), and \( (x : \tau_1) \rightarrow \tau_2 \) is a dependent product type, where \( x \) may occur in \( \tau_2 \). We use a metavariable \( \sigma \) to denote \( \text{int} \mid (x : \tau_1) \rightarrow \tau_2 \) or \( \prod_{i=1}^{n} (x_i : \rho_i) \).

Intuitively, a refinement type \( \{ \nu : \sigma \mid P \} \) describes a value \( \nu \) of type \( \sigma \) that satisfies the refinement predicate \( P \). For example, \( \{ \nu : \text{int} \mid \nu > 0 \} \) describes a monotonic function on integers.

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

We often write just \( \sigma \) for \( \{ \nu : \sigma \mid \text{true} \} \); \( \tau_1 \rightarrow \tau_2 \) for \( (x : \tau_1) \rightarrow \tau_2 \), and \( \rho_1 \times \rho_2 \) for \( (x : \rho_1) \times \rho_2 \) if \( x \) is not important; \( \tau^m \) for \( \tau \times \cdots \times \tau \) (the \( m \)-th power); \( \{ \nu_1 : \prod_{i=1}^{n} (x_i : \rho_i) \mid P \} \) for \( \{ \nu_1 : \prod_{i=1}^{n} (x_i : \rho_i) \mid P \} \) and \( \forall x.P \) for \( \forall x_1, \ldots, x_n.P \).

\[\text{Figure 2. Operational semantics of the source language}\]

\[\text{Figure 3. Semantics of types}\]

For a type \( \tau \) we define the simple type \( \text{ST}(\tau) \) of \( \tau \) as follows:

\[\text{ST}(\{ \nu : \sigma \mid P \}) = \text{ST}(\sigma) \]
\[\text{ST}((x : \tau_1) \rightarrow \tau_2) = \text{ST}(\tau_1) \rightarrow \text{ST}(\tau_2) \]
\[\text{ST}(\prod_{i=1}^{n} (x_i : \rho_i)) = \text{ST}(\tau_1) \times \cdots \times \text{ST}(\tau_n) \]

Also, we define the order of \( \tau \) by:

\[\text{order}(\{ \nu : \sigma \mid P \}) = \text{order}(\sigma) \]
\[\text{order}((x : \tau_1) \rightarrow \tau_2) = \max(\text{order}(\tau_1) + 1, \text{order}(\tau_2)) \]
\[\text{order}(\prod_{i=1}^{n} (x_i : \rho_i)) = \max(\text{order}(\tau_1), \text{order}(\tau_2)) \]

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

2.2 Semantics of Refinement Types

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

Notice that, by the definition, \( \models^n \tau \) holds for every \( n \) if \( t \) diverges. We write \& and \| for (expression-level) Boolean conjunction and disjunction. Notice that the semantics of \( t_1 \land t_2 \) and
2.3 Restriction on Refinement Types

To enable the reduction of the refinement type checking problem \( \vdash t : \tau \) to the first-order one \( \vdash t' : \tau' \), we have to impose some restrictions on the type \( \tau \). The most important restriction is that only first-order function variables (i.e., functions whose simple types are of the form \( \text{int} \times \cdots \times \text{int} \rightarrow \text{int} \times \cdots \times \text{int} \)) 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 \( \tau \) satisfies the well-formedness condition \( \emptyset \vdash \tau \) defined in Figure 4. In the figure, \( \text{EliHO}_n(\Gamma) \) filters out all the bindings of types whose depth is greater than \( n \), where the depth of a type is defined by:

\[
\text{depth}(\{\nu : \sigma | P\}) = \text{depth}(\sigma) \\
\text{depth}(\text{int}) = 0 \\
\text{depth}(x : \tau_1 \rightarrow \tau_2) = 1 + \max\{\text{depth}(\tau_1), \text{depth}(\tau_2)\} \\
\text{depth}(x : \tau_1 \times \tau_2) = \max\{\text{depth}(\tau_1), \text{depth}(\tau_2)\}.
\]

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 \( (x : \tau_1) \rightarrow \{\nu : \sigma | P\} \) where \( \tau_1 \) is a depth-1 function type, \( x \) may occur in \( P \) but not in \( \sigma \) (there is no such restriction if \( \tau_1 \) is a depth-0 type), and (iii) in a type of the form \( (f_1 : \tau_1) \times \{f_2 : \sigma_2 | P_2\} \times \cdots \times \{f_n : \sigma_n | P_n\} \), \( f_1 \) may occur in \( P_2 \), \ldots, \( P_n \), but not in \( \sigma_2, \ldots, \sigma_n \).

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

3. The special primitive \( \text{fail} \) must not occur in any refinement predicate. Also, in every application \( t_2 t_3 \) in a refinement predicate, \( t_2 \) must not contain function applications nor \( \text{fail} \). (In other words, \( t_2 \) must be effect-free, in the sense that it neither diverges nor fails.)

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

In refinement predicates, usual if-expressions are not allowed; instead we allow “branch-strict” if-expression if \( t \text{ then } t_1 \text{ else } t_2 \) where \( t_1 \) and \( t_2 \) are both evaluated before the evaluation of \( t \). This is equivalent to \( t_1 ; t_2 \) if \( t \text{ then } t_1 \text{ else } t_2 \); 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 \( \sigma \) of a given type checking problem \( \vdash t : \{\nu : \sigma | P\} \) rather than the top level refinement \( P \); since given

\[
\vdash t : \{\nu : \sigma | P\} \\
\vdash \nu = t \text{ in } (\nu, (\lambda x. t_i)) : \sigma \times \prod_i (\text{int}^n \rightarrow \{r : \text{int} | r \})
\]

where \( \forall x \cdot \land t_i \) does not satisfy the restrictions above, we can replace it by an equivalent problem

\[
\vdash \text{let } \nu = t \text{ in } (\nu, (\lambda x. t_i)) : \sigma \times \prod_i (\text{int}^n \rightarrow \{r : \text{int} | r \})
\]

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:

\[
\text{let } \nu = t \text{ in } (\nu, (\lambda x. t_i)) : \sigma \times \prod_i (\text{int}^n \rightarrow \{r : \text{int} | r \})
\]

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 \( 1 \leq x \leq n \land f(x) = 0 \). This is a typical specification for a search function.

3. Encoding Functional Refinement

In this section, we present a transformation \( (-)^2 \) 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 \( (-)^2 \) 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 \( (-)^2 \).

3.1 Idea of the Transformation

The transformation \( (-)^2 \) is in fact the composition of four transformations: \( (((-)^3)^2)^2)^2 \). We explain the idea of each transformation from \( (-)^3 \) to \( (-)^2 \) in the reverse order of the applications,
since \((-)^3\) is the key step and the other ones perform preprocessing to enable the transformation \((-)^4\).

\section{Elimination of universal quantifiers and function symbols from a refinement predicate}

We first discuss a simple case, where there occur only one universal quantifier and one function symbol in a refinement predicate. Consider a refinement type of the form
\[
\{ f : \mathsf{int} \rightarrow \mathsf{int} \mid \forall x. P[f \, x] \}
\]
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
\[
(x : \mathsf{int}) \rightarrow \{ r : \mathsf{int} \mid P[r] \}.
\]
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
\[
\frac{\vdash (t_1, t_2)}{\{ (f, g) : (\tau_1 \rightarrow \tau_1') \times (\tau_2 \rightarrow \tau_2') \mid \forall x_1, x_2. P[f \, x_1, g \, x_2] \}},
\]
where each of the two different function variables occurs once in \(P[f \, x_1, g \, x_2]\), we can transform it to:
\[
\text{let } t = t_1 \text{ in let } g = t_2 \text{ in } \lambda x_1, x_2. (f \, x_1, g \, x_2): \{
(x_1, x_2) : \tau_1 \times \tau_2 \rightarrow \{ (r_1, r_2) : \tau_1' \times \tau_2' \mid P[r_1, r_2] \}).
\]
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 constants, 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 \(\nu\). Those conditions are achieved by the preprocessing \((-)^3\), \((-)^2\), and \((-)^1\) explained below.

\section{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
\[
t : \{ f : \mathsf{int} \rightarrow \mathsf{int} \mid P[f \, x, f \, y] \}
\]
where \(f\) occurs exactly twice, we transform this to
\[
\text{let } t = t \text{ in } (f, f):
\]
\[
\{ (f_1, f_2) : (\mathsf{int} \rightarrow \mathsf{int})^2 \mid P[f_1 \, x, f_2 \, y] \},
\]
so that each of the function variables \(f_1\) and \(f_2\) now occurs just once in the refinement predicate.

\section{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:
\[
\{ f : \mathsf{int} \rightarrow \mathsf{int} \mid \forall x. P[f \, t] \}
\]
where \(P[-]\) is a context with one occurrence of the hole \([-]\) and \(t\) is either a non-variable, or a quantified variable \(x_i \in \{ x \}\) but there is another occurrence of \(x_i\), we transform this to
\[
\{ f : \mathsf{int} \rightarrow \mathsf{int} \mid \forall \tilde{x}. y, y = t \Rightarrow P[y] \}
\]
where \(y\) is a fresh variable.

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

\section{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 \(\nu\);” this can be relaxed so that a function variable in \(P\) may be bound at the position of \(f\) in \(\{ f : \tau \rightarrow \{ \nu : \sigma \mid P \}\}\) as described below. A judgment
\[
\frac{\vdash t : \{ f : \tau_1 \rightarrow \tau_2 \} \rightarrow \{ \nu : \tau \mid P \}}{\vdash \text{let } g = t \text{ in } \lambda f'. (f', g'f') : \{ (f' : \tau_1 \rightarrow \tau_2) \rightarrow \{ (f', \nu) : (f' : \tau_1 \rightarrow \tau_2) \times \tau \mid P[f' \rightarrow f'] \}\}
\]
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, \((-)^{11}\) has been used for the example of \texttt{append2}. We now demonstrate uses of \((-)^{22}\) and \((-)^{44}\) with the other example in Section 1:
\[
\frac{\vdash (\text{sum, sum2}) : \{ (f : \mathsf{int} \rightarrow \mathsf{int}) \times g : \mathsf{int} \rightarrow \mathsf{int} \mid \forall n. g(n) = f(n) \}}{\vdash \text{let } g = \text{ let } t = \lambda x, x_1, x_2 \rightarrow (f \, x, g \, x), \text{ let } \lambda x_1, x_2 \rightarrow \{ (r_1, r_2) : \mathsf{int}^2 \mid r_1 = n2 \Rightarrow r_2 = r_1 \}}
\]
The refinement predicate is transformed by \((-)^{22}\) to
\[
\forall n, n1, n2. n_1 = n \Rightarrow n_2 = n \Rightarrow g(n) = f(n),
\]
which is equivalent to
\[
\forall n1, n2. n_1 = n_2 \Rightarrow g(n_1) = f(n_2).
\]
By \((-)^{44}\), the above type checking problem is reduced to the following one:
\[
\frac{\vdash \lambda n_1, n_2. (\text{sum, sum2}) n_1 n_2 : \{ (n_1, n_2) : \mathsf{int}^4 \rightarrow \{ (r_1, r_2) : \mathsf{int}^2 \mid n_1 = n_2 \Rightarrow r_2 = r_1 \}}{\vdash \text{let } \lambda i, j \rightarrow (i, j), \text{ let } \lambda i, j \rightarrow \{ (i, j) : \mathsf{int} \mid i = j \Rightarrow j = i \}}
\]
One may notice that the result of the transformation above is different from that of \texttt{sum} and \texttt{sumacc} in Section 1, which is obtained by applying a further transformation explained in Section 4.

\subsection{Transformations}

We give formal definitions of the transformations \((-)^{11}, (-)^{22},\)
\((-)^{33}, \) and \((-)^{44}\) in this order. For the sake of simplicity, w.l.o.g., we assume that every term has a type of the following form:
\[
\tau ::= \{ \nu : \prod_{i=1}^{m} x_i : \mathsf{int} \times \prod_{j=1}^{n} (f_j : (y_j : \tau_j) \rightarrow \tau_j) \mid P \}\.
\]
In fact, any type (and accordingly terms of that type) can be transformed to the above form; e.g.
\[
\{ (f, x) : \{ f : \mathsf{int} \rightarrow \tau \mid P_1 \} \times \{ x : \mathsf{int} \mid P_2 \} \mid P \}\]
can be transformed to

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

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

2.1: Removal of Dependencies between Functional Arguments and Return Types

Figure 5 shows the key cases of the definition of the transformation $(-)^{2\uparrow}$ for types and terms. For types, $(-)^{2\uparrow}$ copies (the depth-1 components of) the argument type of a function type to the return type. For example, a refinement of the type form

$$(x, f): \text{int} \times (\tau \to \tau') \rightarrow \{r : \sigma \mid P(r, x, f)\}$$

is transformed to a type of the form

$$(x, f): \text{int} \times (\text{int} \rightarrow \text{int}) \rightarrow (f : \text{int} \rightarrow \text{int}) \times \{r : \sigma \mid P(r, x, f')\}.$$  

Note that the return type no longer depends on the argument $f$. As for the term transformation, in the rule for $\text{fix}(f, x, t)$, (the depth-1 components of) the argument $x$ is added to the return value. In the rule for $t_1 t_2$, $(t_1)^{2\uparrow} (t_2)^{2\uparrow}$ returns a pair of (the depth-1 components of) the value of $t_2$ and the value of $t_2$; therefore, we extract such the value of $t_2$ by applying the projection. For example, the term $\text{fix}(f, x, f x)$ is transformed to $\text{fix}(f, \lambda x. (\text{pr}^\text{\texttt{int}}_1 x, \text{pr}^\tau_0 f(x)))$.

After the transformation $(-)^{2\uparrow}$, the type of the program satisfies a more restricted well-formedness condition, obtained by replacing all judgments $\Gamma \mid \Delta \vdash P$ in Figure 4 with $\Gamma, \Delta \mid \vdash P$.

2.2: Normalization of Function Arguments in Refinement Predicates

Figure 6 defines the transformation $(-)^{2\downarrow}$. In the figure, $\&$ is an expression-level Boolean conjunction, and $\land k \& t_k$ abbreviates $t_1 \land \cdots \land t_k$. For each occurrence of application $(f t')^{2\downarrow}$ 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 $z((f t')^i)$; for an occurrence of a term $t'$ in $P$, $\text{app}(t')$ is the set of occurrences of applications in $t'$; $\text{arg}(t')$ is the term obtained by replacing the argument $t'$ of each $(t t')' \in \text{app}(t')$ with $z((f t')^i)$; and $\text{argEq}(t')$ denotes each $z'$ and $z((f t')'^i)$. In the figure, $\text{eqOQ}(-)$ eliminates the original quantifiers $\forall x_i$ as follows: by the assumption 2 in Section 2.3, for each $i$ and $k$, if $x_i$ occurs in $t_k$, then $x_i$ occurs at least once as the argument of an application, and so there is some $z_k^i$ such that $(z_k^i = x_i) \in \text{argEq}(t_k)$; hence $\forall x_i$ can be eliminated by substituting $z_k^i$ for $x_i$.

For example, consider the type

$$\{(f, g) : \text{int} \rightarrow \text{int} \mid \forall x. f x = x\}.$$  

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

$$\text{app}(t) = \{f x, g x\},$$

$$\text{argEq}(\text{app}(t)) = \text{argEq}(f x) \& \text{argEq}(g x) = (z((f x)) = \text{arg}(x)) \& (z((g x)) = \text{arg}(x)) = (z((f x)) = x) \& (z((g x)) = x),$$

and the transformed predicate before $\text{eqOQ}(-)$ is

$$\forall x, z((f x)) = z((g x)) \& x \Rightarrow f z((f x)) = g z((g x)).$$

By applying $\text{eqOQ}(-)$, we obtain:

$$\forall x, z((f x)) = z((g x)) \& z((g x)) = z((f x)) \Rightarrow f z((f x)) = g z((g x)),$$

which may be simplified further to

$$\forall x, z((f x)) = z((g x)) \Rightarrow f z((f x)) = g z((g x)).$$

2.3: Replication of Functions

As explained in Section 3.1, $(-)^{2\downarrow}$ replicates a function $f_j$ according to the number $m_j$ of occurrences of $f_j$ in the predicate $P$ of a
\[(\nu : \prod_{i=1}^{m} (x_i : \text{int}) \times \prod_{j=1}^{m} (f_j : \tau_j \to \tau_j') \mid P)\] 

\[(\nu : \prod_{i=1}^{m} (x_i : \text{int}) \times \prod_{j=1}^{m} (f_j : (\tau_j)_\phi \to (\tau'_j)_\phi) \mid P')\]

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

and

\[P' \equiv P[a_{j,1} \mapsto f_1, t_1, i \in (1, \ldots, m), i \in (1, \ldots, m')]

\[(\text{fix}(f, x, t))_\phi \equiv \text{fix}(f, x, t)\]

\[(m = T(\text{fix}(f, x, t)) \text{ and } \overrightarrow{t} = (t, \ldots, t) \text{ for a term } t)

\[(t_1, t_2)_\phi \equiv (\text{pr}_1(t_1)_\phi, (t_2)_\phi)

Figure 7. Replication of functions \((-)^{\text{t}_\phi}\)

refinement type \(\tau = \{\nu : \prod_{i=1}^{m} \text{int} \times \prod_{j=1}^{m} (f_j : \tau_j \to \tau_j') \mid P\};\)
we call \(m_j\) the multiplicity of \(f_j\) and write \(\text{mul}(\tau, j) = \text{mul}(P, j)\).
We call the sequence \((m_1, \ldots, m_n)\) the multiplicity type of \(\tau\).

The transformation \((-)^{\text{t}_\phi}\) is parameterized by a multiplicity type \(\phi\) for types, and a multiplicity annotation \(T\) for terms. The multiplicity types are defined by the following grammar:

\[\phi ::= \{\prod_{i=1}^{m} \text{int} \times \prod_{j=1}^{m} (\phi_j \to \phi'_j) \mid M\}\]

Here, \(M\) is a function from \(\{1, \ldots, m\}\) to positive integers such that \(M(j) = 1\) if \(\phi_j \to \phi'_j\) is not depth-1. Intuitively, \(M(j)\) denotes how many copies should be prepared for the \(j\)-th function (of type \(\phi_j \to \phi'_j\)).

For a refinement type \(\tau = \{\nu : \prod_{i=1}^{m} (x_i : \text{int}) \times \prod_{j=1}^{m} (f_j : \tau_j \to \tau_j') \mid P\}\) and a multiplicity type \(\phi = \{\prod_{i=1}^{m} \text{int} \times \prod_{j=1}^{m} (\phi_j \to \phi'_j) \mid M\}\), we write \(\tau \leq \text{mul}(\phi)\) if all the multiplicities in \(\tau\) are pointwise less than or equal to those in \(\phi\), i.e., if \(\text{mul}(P, j) \leq M(j), \tau_j \leq \text{mul}(\phi_j)\) for all \(j\). Intuitively, \(\tau \leq \text{mul}(\phi)\) means that copying functions according to \(\phi\) is sufficient for keeping track of the correlations between functions expressed by \(\tau\). Thus, in the transformation rule for types in Figure 7, we assume that \(\tau \leq \text{mul}(\phi)\) and replicate each function type according to \(\phi\).

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 \(\text{int}^{a} \times \prod_{j=1}^{m} (\tau_j \to \tau_j')\), then its multiplicity is a sequence \(m_1, \ldots, m_j\) of positive integers. In the case for abstractions, the function \(\text{fix}(f, x, t)\) is copied to an \(m\)-tupled function where \(m\) is the multiplicity of \(\text{fix}(f, x, t)\). 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 \(\text{pr}_1\) for matching types correctly.

Given a type checking problem \(\vdash t : \tau\), we infer \(\phi\) and \(T\) automatically (so that the transformation \((-)^{\phi, T}\) is fully automatic). For multiplicity types, we can choose the least \(\phi\) such that \(\tau \leq \text{mul}(\phi)\) and determine \(T(t)\) according to \(\phi\). For some subterms, however, their multiplicity annotations are not determined by \(\tau\); 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 \(\vdash t_1 t_2 : \tau\). For such a subterm \(t'\), we just infer the value of \(T(t')\). Fortunately, as long as \(\phi\) and \(T\) satisfy a certain consistency condition (for example, in if \(t_0\) then \(t_1\) 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 \(\phi\) 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 \(\phi\) and \(T\), and gradually increase them until the verification succeeds.

\(\exists^{\phi, T}\): Elimination of Universal Quantifiers and Function Symbols

Figure 8 defines the transformation \((-)^{\phi, T}\). For a type \(\tau\), we write \((\tau)\) for the option type \(\tau + 1\); we explain this later.

For the transformation of refinement predicates, we use the functions \(\hat{f}(-)\) and \(\hat{z}(-)\) defined as follows. For an input type \((\xi(x), f, y, \ldots)\) \(\mid P\) of \((-)^{\phi, T}\), we can assume that by \((-)^{\phi, T}\), function symbols occurring in a refinement predicate are in \((f, y, \ldots) \mid \leq m\); and that by \((-)^{\phi, T}\), all application occurrences in \(P\) have distinct function variables, and have distinct argument variables that quantified universally. Thus, there is an injection \(\hat{f}(-)\) from the set \(X\) of occurrences of applications in \(P\) to \((f, y, \ldots) \mid \leq m\) such that for any application occurrence \(f\), \(f = \hat{f}(f(t))\); and also there is a bijection \(\hat{z}(-)\) from the same \(X\) to the set of the variables that are universally in \(P\).

For example, let us continue the example used for \(\exists^{\phi, T}\):

\[\{ (f, g) : (\text{int} \to \text{int}) \mid \forall x (f(x), g(x)) \} \]

\[\hat{z}(f(x), g(x)) = z(g(x)) \Rightarrow f(z(f(x)) = g(z(x))\].
The transformed type is of the form

\[(y 1, y 2): (\text{\textbf{int}})^{2} \rightarrow \{ (r 1, r 2): (\text{\textbf{int}})^{2} | (\ldots)^{2} \} \].

The occurrences of applications are:

\[a_{1} = f \ z^{(f z)} , \quad a_{2} = g \ z^{(g z)} ,\]

and

\[z^{(f z)} = z^{(f z)} , \quad z^{(g z)} = z^{(g z)} .\]

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

\[j^{(f z)} = 1 , \quad j^{(g z)} = 2 .\]

Hence, the predicate \((\ldots)^{2}\) is \( y 1 = y 2 \rightarrow r 1 = r 2 \) and the transformed type is

\[(y 1, y 2): (\text{\textbf{int}})^{2} \rightarrow \{ (r 1, r 2): (\text{\textbf{int}})^{2} | y 1 = y 2 \rightarrow r 1 = r 2 \} .\]

The transformation of terms follows the ideas described in Section 3.1 except that option types have been introduced. For example, the term \( (\lambda x. t 1 , \lambda y. t 2) \) is transformed into the term

\[\lambda (x, y). \text{\textbf{let}} r 1 = \text{\textbf{if}} x = \perp \text{\textbf{then}} \perp \text{\textbf{else}} (t 1)^{2} \text{\textbf{in}} \]

\[\text{\textbf{let}} r 2 = \text{\textbf{if}} y = \perp \text{\textbf{then}} \perp \text{\textbf{else}} (t 2)^{2} \text{\textbf{in}} (r 1, r 2) .\]

Here, \( \perp \) is the exception of option types (i.e. \texttt{None} in OCaml or \texttt{nothing} in Haskell), and we have omitted a projection from \( (\perp)^{2} \) to \( \tau \). The option type (and the conditional branch)

\[\text{\textbf{if}} x = \perp \text{\textbf{then}} \ldots \text{\textbf{else}} \]

is used to preserve the side effect (divergence or failure). For example, consider the following program:

\[\text{\textbf{let}} \text{rec} \ f x = \ldots \text{\textbf{and}} \ g y = g y \text{\textbf{in}} \]

\[\text{\textbf{let}} \text{main} n = \text{\textbf{assert}} (f n > 0) .\]

This program defines functions \( f \) and \( g \) but does not use \( q \). The body of the main function is transformed to \( f \text{\textbf{est}} \{ f g (n, \perp) \} > 0 \), where \( f g \) is a (na"ively) 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

\[(t)^{2} (\ldots x)\]

is a tuple consisting of integers and a single function. If \( pr_{1} t \) is a function, \( pr_{1,n} (x, \perp, \ldots, \perp, w, \perp, \ldots, \perp) \) should correspond to \( pr_{1} (t) w \). Hence, the output of the transformation is \( \lambda w. pr_{1,n} (x, \perp, \ldots, \perp, w, \perp, \ldots, \perp) \). Otherwise, \( pr_{1} t \) is just transformed in a compositional manner.

Finally, we define \( (-)^{2} \) as the composition of the transformations:

\[(t)^{2} = (((((t)^{2})^{2})^{2})^{2})^{2} .\]

3.3 Soundness of the Transformation

The transformation \((-)^{2}\) reduces type checking of general refinement types (with the assumptions in Section 2.3) into that of first-order refinement types, and its soundness is ensured by Theorem 1 below.

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

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

\[\begin{align*}
\models (t)^{2}_{\tau} : (\tau)_{\phi}^{2} & \quad \text{implies} \quad \models t : \tau .
\end{align*}\]

\textbf{Proof. See Appendix H.}\]

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

4. Transformations for Enabling First-Order Refinement Type Checking

The transformation \((-)^{2}\) in the previous section allowed us to reduce the refinement type checking \( \models t : \tau \) to the first-order refinement type checking \( \models (t)^{2} : (\tau)^{2} \), but it does not necessarily enable us to prove the latter by using the existing automated verification tools [10, 13, 14, 17, 18, 20]. This is due to the incompleteness of the tools for proving \( \models (t)^{2} : (\tau)^{2} \). They are either based on (variations of) the first-order refinement type system [21] (see Appendix B for such a refinement type system), or higher-order model checking [9, 10], whose verification power is also equivalent to a first-order refinement type system (with intersection types). In these systems, the proof of \( \models t : \tau \) (where \( \tau \) is a first-order refinement type) must be compositional: if \( t = t 1 t 2 \), then \( \tau \) such that \( \models t 1 : 1 \rightarrow \tau \) and \( \models t 2 : \tau \) is (somehow automatically) found, from which \( \models t 1 t 2 : \tau \) is derived. The compositional property is fine, but the problem is that \( \tau \) 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 \( \text{sum} \) and \( \text{sum2} \) in Section 1. It is expressed as the following refinement type checking problem:

\[\begin{align*}
\models & (\text{\textbf{sum}}) \times (\text{\textbf{int}}) \rightarrow (\text{\textbf{int}}) | r = \text{\textbf{sum}}(n) .
\end{align*}\]

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

\[\begin{align*}
\models & \lambda (x, y). (\text{\textbf{sum}} x, \text{\textbf{sum2}} y) : \ldots \rightarrow (r : \text{\textbf{int}}) | x = y \Rightarrow r = r 1 = r 2 .
\end{align*}\]

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 \( \text{sum} \) and \( \text{sum2} \):

\[\begin{align*}
(x : \text{\textbf{int}}) & \rightarrow (r : \text{\textbf{int}} | x = 0 \Rightarrow r = 0) \wedge (x > 0 \Rightarrow r = x(x + 1)/2) .
\end{align*}\]

To deal with the problem above, we further refine the transformation \((-)^{2}\) by (i) tupling of recursive functions [5] and (ii) insertion of assumptions.

4.1 Tupling of Recursion

The idea is that when a tuple of function calls is introduced by \( (-)^{2} ((f 1, pr_{1} y), \ldots, f n (pr_{n} y)) \) in Figure 8 and \( \text{\textbf{sum}} x, \text{\textbf{sum2}} y \) 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 \( \text{sum2} \) defined by:

\[\begin{align*}
\text{let rec} & \text{sum2} (x, y) = \text{\textbf{sum}} \text{\textbf{sumacc}}(x, y, 0) \text{\textbf{and}} \text{\textbf{sumacc}}(x, y, m) = \ldots \text{\textbf{if}} x < 0 \text{\textbf{then}} \text{\textbf{if}} y < 0 \text{\textbf{then}} (0, 0) \text{\textbf{else}} \ldots
\end{align*}\]

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

\[\begin{align*}
\text{let} & f = \mathbf{fix}(f, \lambda x. \text{\textbf{if}} t 11 \text{\textbf{then}} t 12 \text{\textbf{else}} E 1[f t 1]) \text{\textbf{in}}
\end{align*}\]
let \( g = \text{fix}(g, \lambda y. \text{if } t_2 \text{ then } t_2 \text{ else } E_2[g t_2]) \) in \( (f, g) \) ... where \( E_1 \) and \( E_2 \) are evaluation contexts, and \( t_1, t_2, E_1, \) and \( t_4 \) have no occurrence of \( f \) nor \( g \). Then, we replace \( \lambda (x, y). (f, g, x, y) \) in \((-)^2\) with the following tupled version:

\[
\lambda (x', y'). \text{let } _= f x' \text{ in }
\]

\[
\text{fix}(h, \lambda (x, y). \text{if } t_{11} \text{ then } t_2 \text{ then } (t_{12}, t_{22}) \text{ else } (t_{12}, E_2[g t_2]) \text{ else if } t_{21} \text{ then } (E_1[f t_1], t_{22}) \text{ else let } (r_1, r_2) = h (t_1, t_2) \text{ in } (E_1[r_1], E_2[r_2]) \}) (x', y').
\]

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 \( t_{11} = \text{true}, t_{12} = \text{fall} \) and \( t_{21} = \Omega \). The call to the original function fails, but without let \(_= f x' \) in \( \ldots \), the call to the tupled version would diverge.

The function \( \text{sum, sumacc} \) shown in Section 1 can be obtained by the above tuple (with some simplifications).

4.2 Insertion of Assume Expressions

The above refinement of \((-)^2\) alone is often sufficient. For example, consider the problem of proving that the function:

\[
\text{let diff } (f, g) = \text{fun } x \rightarrow f x - g x \text{ has type}
\]

\[
\tau \overset{\text{def}}{=} \{(f, g) : (\text{int} \rightarrow \text{int})^2 | \forall x. f x > g x\}
\]

\[
ightarrow \{h : \text{int} \rightarrow \text{int} | \forall x. h x > 0\}.
\]

The function is transformed to the following one by \((-)^2\):

\[
\text{let diff } fg = \text{fun } x \rightarrow \text{let } r_1, r_2 = fg (x, x) \text{ in } r_1 - r_2,
\]

and the type \( \tau \) is transformed to

\[
\{(x_1, x_2) : \text{int}^2 \} \rightarrow \{(r_1, r_2) : \text{int}^2 | x_1 = x_2 \Rightarrow r_1 > r_2\}
\]

\[
\rightarrow \{(\text{int} \rightarrow \{r : \text{int} | r > 0\}\}
\]

Here, \( \perp \) is used as a dummy argument as explained in Section 3.2-\( \sharp \). We cannot conclude that \( r_1 > r_2 \) has type \( \{r : \text{int} | r > 0\}\) because there is no information about the correlation between \( r_1 \) and \( r_2 \); from the refinement type of \( f g \), we can infer that \( x = \perp \Rightarrow r_1 > r_2 \) and \( \perp = x \Rightarrow r_1 > r_2 \), but \( r_1 > r_2 \) cannot be derived.\(^3\) In fact, \( \vdash (\text{diff} \{r : \}\) does not hold,\(^4\) which is a counterexample of the converse of Theorem 1.

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

\[
\text{assume}\{\text{let } (r_1', r_2') = fg (x, x) \text{ in } r_1 = r_1' \& r_2 = r_2'\}
\]

Here, \( \text{assume}(t) \) is a shorthand for if \( t \) then \text{true else loop()}\(^5\) where loop() is an infinite loop. From \( fg (x, x), \) we obtain \( r_1' > r_2' \) by using the refinement type of \( f g \). We can then use the assumed condition to conclude that \( r_1 > r_2' \). In general, whenever there are two calls
\[
\text{let } \text{diff } fg = \text{fun } x \rightarrow \text{let } r_1, r_2 - fg (x, x) \text{ in } r_1 - r_2,
\]

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.

\(^4\) To see this, apply \( (\text{diff} \{r : \}) \) to

\[
\lambda (x_1, x_2). \text{if } x_1 = x_2 \text{ then } (1, 0) \text{ else } (0, 0)
\]

and apply the returned value to, say, \( 0 \).

5. Implementation and Experiments

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

Table 1 shows the results of the experiments. The columns “size” show the size of the programs before and after the transformations described in Section 4, where the size is measured by word counts.\(^7\) The column “pred.” shows the number of predicates manually given as hints for the backend model checker MoCHi. The experiment was conducted on Intel Core i7-3930K CPU and 16 GB memory. The implementation and benchmark programs are available at http://www-ku.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-xx-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 \( \text{sum} \) is monotonic, i.e., \( \forall m, n, m \leq n \Rightarrow \text{sum}(m) \leq \text{sum}(n) \). 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”:

\[
\text{let } r_1, r_2 = fg (x, \perp) \text{ in }
\]

\[
\text{C[let } r_1', r_2' = fg (\perp, y) \text{ in } ...
\]

(\( \perp \) is some context), we insert an assume statement as in

\[
\text{let } r_1, r_2 = fg (x, \perp) \text{ in }
\]

\[
\text{C[let } r_1', r_2' = fg (\perp, y) \text{ in } assume\{let } (r_1'', r_2'') = fg (x, y) \text{ in } r_1 = r_1'' \& r_2 = r_2''\}; ...
\]

We write \((-)^2\) for the above assume-inserted version of \((-)^2\).

The formal definition of \((-)^2\) is described in Appendix K. In the target language, \text{fail} is treated as an exception, and we define \text{assume}(t) as a shorthand for:

\[
\text{if } (\text{try } t \text{ with } \text{fail } \rightarrow \text{false}) \text{ then } \text{true else loop()}.\]

Note that our backend model checker MoCHi [10, 14] supports exceptions. After replacing \((-)^2\) with \((-)^2\), Theorem 1 is still valid:

\[
\vdash (t)^2 \; : \; (\tau)^2 \text{ implies } \vdash t : \tau.
\]

See Appendix L for the details of the proof.

---

\(^3\) One may think that we can just combine the two calls of \( fg \) as

\[
\text{let } \text{diff } fg = \\text{fun } x \rightarrow \text{let } r_1, r_2 - fg (x, x) \text{ in } r_1 - r_2,
\]

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.

\(^4\) To see this, apply \( (\text{diff} \{r : \}) \) to

\[
\lambda (x_1, x_2). \text{if } x_1 = x_2 \text{ then } (1, 0) \text{ else } (0, 0)
\]

and apply the returned value to, say, \( 0 \).

\(^5\) 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.
The program “append-nil-xs” asserts that $\text{append} \text{ nil} \ xs = \xs$. The program “rev” asserts that two list reversal functions are the same, the one uses $\text{snoc}$ function and the other one uses an accumulation parameter. The program “insert” asserts that $\text{insert} \ x \ xs$ is sorted for a sorted list $\xs$. Note that, for all the programs, invariant annotations were not supplied, except the specification being checked. For example, for “a-max-gen” above, the specification is that the main has type $\text{int} \rightarrow \text{unit}$, which just means that the assertion $\text{assert} \ (\text{array} \ i \ <= \ m)$ never fails; no type declaration for $\text{array\_max}$ was supplied. For the “append-xs-nil”, as described in Section 1, the verifier checks that $\text{append}$ has the type

$$\tau : \text{int} \rightarrow (\{\text{ys}: \tau \mid \text{ys}(0) = \text{None}\}) \rightarrow (\text{int} \rightarrow (\text{int} \rightarrow \text{unit}))$$

where $\tau \triangleq \text{int} \rightarrow (\text{int} \rightarrow \text{option})$. (See Appendix A for more details.)

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

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

### 6. Related Work

Knowles and Flanagan [7, 8] gave a general refinement type system where refinement predicates can refer to functions. Their verification method is however a combination of static and dynamic checking, which delegates type constraints that cannot 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 refinement type checking [13, 24] supports the use of uninterpreted function symbols in refinement predicates. Uninterpreted functions can be used only for total functions. Furthermore, their method cannot be used to prove relational properties like the ones given in Section 1, since their method cannot refer to the definitions of the uninterpreted functions.

Uno et al. [19] have proposed another approach to increase the power of automated verification based on first-order refinement types. To overcome the limitation that refinement predicates cannot refer to functions, they added an extra parameter for each higher-order argument so that the extra parameter captures the behavior of the higher-order argument, and the dependency be-

---

---

8 Not fully automated in the sense that a user must supply hints on predicates.
let rec make_list n =  |
  "append-xs-nil". The whole program is shown below:

A. Verification of “append-xs-nil”

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

let rec make_list n = |
  if n < 0 then [] |
  else Random.int 10 :: make_list (n-1) |
let rec append xs ys = |
  match xs with |
  | [] -> ys |
  | x::xs' -> let xs'', ys', rs = append xs' ys in |
  | x::xs'', ys'', x::rs |
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 \texttt{int} \rightarrow \texttt{unit}, which means that the assertion never fails. As mentioned in Section 5, only the program above is given to the verifier, without any annotations.

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

Then, the transformation \((-)^3\) 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 (i,j,k) = |
  match xs with |
  | [] -> let r1, r2, r3 = None, ys, ys k in |
  | 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 \((-)^2\) and \((-)^3\) have no effect in this case. By applying the transformation \((-)^3\), 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, ys k in |
  | 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-matches of option types.

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

$$(\text{int} \rightarrow \text{int}) \rightarrow$$

$$(j : \text{int}) \rightarrow (y : \text{int} \mid j = 0 \Rightarrow y = \text{None}) \rightarrow$$

$$(i, j, k : \text{int}^3) \rightarrow \{r_1, r_2, r_3 : \text{int}^3 \mid i = j \Rightarrow r_1 = r_2\}$$

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

### B. A Refinement Type System

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

The type judgment used in the type system is of the form $\Gamma \vdash \cdot : \cdot$ where $\Gamma$, 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 $\vdash^G_{\mathcal{L}}$ (defined in Appendix C), which represents usual scope rules of dependent types.

We define value environments as mappings from variables to closed values and use a meta variable $\eta$ for them. For a value environment $\eta$ and an environment $\Gamma$ such that $\vdash^G_{\mathcal{L}} \Gamma$, we define $\eta \vdash^G_{\mathcal{L}} \Gamma$ as follows:

$\emptyset \vdash^G_{\mathcal{L}} \emptyset$ def true

$\eta \cup \{x \mapsto V\} \vdash^G_{\mathcal{L}} \Gamma, x : \tau$ def $\eta \vdash^G_{\mathcal{L}} \Gamma$ and $\vdash^G_{\mathcal{L}} V : \tau[\eta]$

The type judgment $\Gamma \vdash \cdot : \cdot$ semantically means that for any $n$ and $\eta$ such that $\eta \vdash^G_{\mathcal{L}} \Gamma$, then $\vdash^G_{\mathcal{L}} t[\eta] : \tau[\eta]$. The general refinement type system is given in Figures 9 and 10. The judgment $\Gamma \vdash P : \cdot$ means that, in $\mathcal{L}$, $P$ implies $\mathcal{P}'$ under the type environment $\Gamma$. We assume that the logic $\mathcal{L}$ satisfies that, if $\Gamma \vdash P : \cdot$, then for any $n$ and $\eta$ such that $\eta \vdash^G_{\mathcal{L}} \Gamma$, holds, $\vdash^G_{\mathcal{L}} P[\eta]$ implies $\vdash^G_{\mathcal{L}} P'[\eta]$. In Figure 9, we define $\nu(x \leftarrow t)$ as let $x = t$ in $\nu$, and extend it to the operations $P(x \leftarrow t)$ and $\sigma(x \leftarrow t)$ compositionally. For example, $(\forall \nu, t_1 \land t_2)(x \leftarrow t) = \nu(x_1 \leftarrow t_1), \ldots, \nu(x_n \leftarrow t_n)$.

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

$\Gamma \vdash 42 : \{\nu : \text{int} \mid \nu \geq 0\}$

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
Trivial.

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

$P = \forall x. P$ Trivial.

$P = P_1 \land P_2$ Trivial.

$P = \bot$ We show that

\[ \models^p_{n+1} \text{let } x = t_0 \text{ in } t \iff \models^p_n \text{let } x = t_1 \text{ in } t. \]

Since $t_0 \to t_1$,

\[ \text{let } x = t_0 \text{ in } t \to \text{let } x = t_1 \text{ in } t; \]

hence,

\[ \models^p_{n+1} \text{let } x = t_0 \text{ in } t \]

\[ \text{let } x = t_0 \text{ in } t \to \text{let } x = t_1 \text{ in } t, \]

where the left-to-right implication is from (1) while the converse is from (1) and since the evaluation is deterministic.

\[ \tau = (\nu : \sigma [P]) \text{ Trivial.} \]

\[ \tau = (x : \tau_1) \to \tau_2 \text{ Trivial.} \]

\[ \tau = (x_1 : \tau_1) \times \tau_2 \text{ Trivial.} \]

\[ \square \]

The following are typical lemmas for step-index.

**Lemma 4.**

1. For any $n, V, \text{ and } \tau$, if $\models^n_{\tau+1} V : \tau$ then $\models^n V : \tau$.
2. For $n_0 \to n_1$ and $\models^n_{\tau+1} t : \tau$, then $\models^{n_0 + n_1} t : \tau$.

Proof. Straightforward: 1 is by induction on $\tau$, and 2 is from the definition of $\models^n$.

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

**Theorem 5** (Soundness of Type System). For any $\mathcal{L}$, if $\vdash \epsilon \tau \rightarrow \tau$, then $\models \tau$.

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

\[ \Gamma \vdash \tau \iff \forall n \in \mathbb{N} \text{, if } \models^n \tau \text{ then } \models^n \Gamma \vdash \tau. \]

**Lemma 6** (Soundness of Subtyping). For any $\mathcal{L}$, if $\Gamma \vdash \tau \rightarrow \tau'$, then for any $n$ and $\eta$ such that $\eta \models^n \Gamma$ and for any $V$, if $\models^n V : \tau \rightarrow \tau'$ then $\models^n \Gamma \vdash \eta \models^n V : \tau'$.

Also, if $\Gamma \vdash \eta : \tau'$, then for any $n$ and $\eta$, if $\eta \models^n \Gamma$ then $\eta \models^n \Gamma'$.

Proof. By induction on the derivations of $\Gamma \vdash \epsilon \tau \rightarrow \tau'$ and $\models^n \Gamma \vdash \tau$. (See Appendix E for details.)

**Lemma 7.** For any $\mathcal{L}$, if $\Gamma \vdash \epsilon \tau \rightarrow \tau$, then $\models \tau$.

Proof. By induction on the derivations of $\Gamma \vdash \epsilon \tau \rightarrow \tau$. (See Appendix F for details.)

**E. Proof of Lemma 6**

We prove Lemma 6 by induction on the derivations of $\Gamma \vdash \epsilon \tau \rightarrow \tau'$ and $\models^n \Gamma \vdash \tau'$. (See Appendix G for details.)

**SUB-REFINE** Trivial from the inductive hypothesis and the assumption on $\mathcal{L}$, i.e., if $\models^n P \vdash \epsilon \tau'$, then for any $n$ and $\eta$ such that $\eta \models^n \Gamma$, if $\models^n P \eta$, then $\models^n P' \eta$.

**SUB-INT** Trivial.

**SUB-FUN** This case is also trivial (and tedious), but we give the detail for demonstration. Suppose inductive hypotheses:
(i) for any $n$, $\eta \vdash_0 \Gamma$, and $V_1$, if $\vdash_0 V_1 : \tau_1[\eta]$ then $\vdash_0 V_1 : \tau_1[\eta]$.
(ii) for any $n$, $\eta' \vdash_0 (\Gamma', x_1 : \tau_1')$ and $V_2$, if $\vdash_0 V_2 : \tau_2[\eta']$ then $\vdash_0 V_2 : \tau_2[\eta']$.

For given $n$, $\eta$ such that $\vdash_0 \Gamma$, and $V$, assume $\vdash_0 V : (x_1: \tau_1) \rightarrow \tau_2$, i.e.,

(iii) for any $n' \leq n$, any $V_1$ such that $\vdash_0 V_1 : \tau_1[\eta]$, and any $A_2$ and $k \leq n'$ such that $VV_1 \rightarrow^k A_2$, $\vdash_0 A_2 : \tau_2[\eta][x_1 \mapsto V_1]$.

Then, we show that $\vdash_0 V : (x_1: \tau_1) \rightarrow \tau_2$, i.e., for given $n' \leq n$, given $V_1$ such that $\vdash_0 V_1 : \tau_1[\eta]$, and given $A_2$ and $k \leq n'$ such that $VV_1 \rightarrow^k A_2$, we show $\vdash_0 A_2 : \tau_2[\eta][x_1 \mapsto V_1]$.

Since $\eta \vdash_0 \Gamma$, by Lemma 4.1, $\eta \vdash_0 \Gamma$; then since $\vdash_0 V_1 : \tau_1[\eta]$ by (i), we have $\vdash_0 V_1 : \tau_1[\eta]$. Hence, since $VV_1 \rightarrow A_2$ and by (iii),

$$\vdash_0 A_2 : \tau_2[\eta][x_1 \mapsto V_1].$$

(2)

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

$$\eta \vdash_0 \Gamma$$

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

$$\eta \vdash_0 A_2 : \tau_1[\eta]'$$

i.e., $\eta' \vdash_0 \Gamma$, by Lemma 4.1.

From (2), $\vdash_0 A_2 : \tau_2[\eta']$ (thus $A_2$ is a value); hence, by (ii), $\vdash_0 A_2 : \tau_2[\eta'][x_1 \mapsto V_1]$.

**[SUB-PAIR]** Trivial.

**[ENV-PAIR]** Trivial.

**F. Proof of Lemma 7**

We prove Lemma 7 by induction on the derivations of $\Gamma \vdash_0 \xi \vdash_0 \tau$. Important cases are (T-FIX), (T-APP), and (T-FAIL); especially, we use induction on $n$ (only) in the case (T-FIX). Lemma 6 is used in the case (T-SUB).

**[T-VAR]** Trivial.

**[T-CONST]** Trivial.

**[T-IF]** Suppose inductive hypotheses on the typing derivation:

(i) for any $n$, $\eta$, and $V$, if

$$\eta \cup \{x \mapsto \text{true} \} : (\xi \vdash_0 (\Gamma', x : \nu \rightarrow \text{true}))$$

then $\vdash_0 t_1[\eta][x \mapsto V] : \tau_1[\eta][x \mapsto V]$.

(ii) for any $n$, $\eta$, and $V$, if

$$\eta \cup \{x \mapsto \text{true} \} : (\xi \vdash_0 (\Gamma', x : \nu \rightarrow \text{true} \vdash_0 A \text{ and } \nu \neq \text{true}))$$

then $\vdash_0 t_2[\eta][x \mapsto V] : \tau_1[\eta][x \mapsto V]$.

Then, for given $n$, $\eta$, and $V$ such that

$$\eta \cup \{x \mapsto \text{true} \} : \vdash_0 \Gamma, x : (\nu \rightarrow \text{true} \vdash_0 P \text{ and } \nu \neq \text{true})$$

we show

$$\vdash_0 (x \mapsto \text{true} \vdash_0 \Gamma, x : \nu \rightarrow \text{true} \vdash_0 P \text{ and } \nu \neq \text{true})$$

From (3), $\vdash_0 V : \text{int}$, and hence $V$ is an integer. We suppose $V = \text{true}$; the case $V \neq \text{true}$ can be proved similarly.

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

$$\vdash_0 t_1[\eta][x \mapsto V] : \tau_1[\eta][x \mapsto V].$$

While, since $V = \text{true}$, we show

$$\text{if } x \text{ then } t_1 \text{ else } t_2[\eta][x \mapsto V] = \text{if } V \text{ then } t_1[\eta][x \mapsto V] \text{ else } t_2[\eta][x \mapsto V] \rightarrow t_1[\eta][x \mapsto V].$$

Then, from Lemmas 4.1 and 4.2,

$$\vdash_0 (x \mapsto \text{true} \vdash_0 \eta) : \tau_1[\eta][x \mapsto V].$$

**[T-OP]** Trivial.

**[T-FIX]** Suppose inductive hypotheses on the typing derivation:

(i) for any $n$ and $\eta'$, if $\eta' \vdash_0 \Gamma$, $f : (x_1 : \tau_1) \rightarrow \tau_2$, $x_1 : \tau_1$, then $\vdash_0 t_2[\eta'][x_1 \mapsto \tau_2]$. and that $f \not\in \text{fv} \{\tau_1, \text{fv} \{\tau_2\}\}$. We show, by induction on $n$, that for any $n$ and $\eta$ such that $\eta \vdash_0 \Gamma$,

$$\vdash_0 \text{fix}(f, \lambda x_1. t_2)[\eta][x_1 \mapsto \tau_2][\eta] \rightarrow \tau_2[\eta][x_1 \mapsto \tau_2].$$

(4)

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

(ii) for any $\eta$ such that $\eta \vdash_0 \Gamma$,

$$\vdash_0 \text{fix}(f, \lambda x_1. t_2)[\eta][x_1 \mapsto \tau_2][\eta] \rightarrow \tau_2[\eta][x_1 \mapsto \tau_2].$$

and then for given $\eta$ such that $\eta \vdash_0 \Gamma$, we show (4), i.e., for given $n' \leq n$ and given $V_1$ such that $\vdash_0 V_1 : \tau_1[\eta]$, we show

$$\vdash_0 \text{fix}(f, \lambda x_1. t_2)[\eta][x_1 \mapsto \tau_2][\eta] \rightarrow \tau_2[\eta][x_1 \mapsto \tau_2].$$

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

$$\eta' := \eta \cup \{f \mapsto \text{fix}(f, \lambda x_1. t_2)[\eta][x_1 \mapsto \tau_2]\}$$

we show that $\eta' \vdash_0 \Gamma$, $f : (x_1 : \tau_1) \rightarrow \tau_2$, $x_1 : \tau_1$. Using Lemma 4.1, since $\eta \vdash_0 \Gamma$ and $\vdash_0 V_1 : \tau_1[\eta]$, $\eta \vdash_0 \Gamma$ and $\vdash_0 V_1 : \tau_1[\eta]$. Also, since $\eta \vdash_0 \Gamma$, by (ii) and since $n'' \leq n - 1$,

$$\vdash_0 \text{fix}(f, \lambda x_1. t_2)[\eta][x_1 \mapsto \tau_2][\eta] \rightarrow \tau_2[\eta][x_1 \mapsto \tau_2].$$

Hence, with the fact $f \not\in \text{fv} \{\tau_1\}$,

$$\eta' \vdash_0 \Gamma, f : (x_1 : \tau_1) \rightarrow \tau_2, x_1 : \tau_1.$$
Then, for given \( n, \eta \) such that \( \eta \models_0 \Gamma, A_2 \), and \( k \leq n \) such that \( (tt_1)[\eta] \rightarrow^k A_2 \), we show
\[
|n-k|_{A_2} : \{v_2 : \sigma_2(x_1 \leftarrow t_1) \mid P_2(x_1 \leftarrow t_1) \land P_2'[\eta]\},
\]
i.e., the following three:
\[
|n-k|_{A_2} : \sigma_2(x_1 \leftarrow t_1)[\eta],
|n-k|_{P_2(x_1 \leftarrow t_1)[\eta]}[v_2 \mapsto A_2],
|n-k|_{P_2'[\eta]}[v_2 \mapsto A_2].
\]

Since \( (tt_1)[\eta] = t_1[\eta]t_1[\eta] \rightarrow^k A_2 \), there exist \( A \) and \( k_0 \leq n \) such that
\[
t_1[\eta] \rightarrow^{k_0} A_1;
\]

\[
|n-k_0|_{A_1} : \tau_1[\eta].
\]

Especially, \( A_1 \) is a value and
\[
AA_1 \rightarrow^{k-k_0-k_1} A_2.
\]

Now, since
\[
|n-k_0|_{A} : \{\nu : \sigma \mid P\}[\eta] \quad (= \{\nu : (x_1: \tau_1[\eta]) \rightarrow \tau_2[\eta] \mid P[\eta]\},
\]
e specially \( |n-k_0|_{A} : \{x_1: \tau_1[\eta]\} \rightarrow \tau_2[\eta] \). Then, we shall utilize the semantics for function type: Let \( n' := n - k_0 - k_1 \), then \( n' \leq n - k_0 \). Since \( n' \leq n - k_1 \), by (8) and Lemma 4.1,
\[
|n'|_{A} : \tau_1[\eta].
\]
Thus, we have
\[
|n'|_{AA_1} : \tau_2[\eta][x_1 \mapsto A_1].
\]

Then, by the definition of \( |n'| \) and (9), since \( n' - (k - k_0 - k_1) = n - k \),
\[
|n-k_0|_{A_2} : \tau_2[\eta][x_1 \mapsto A_1].
\]

Now, by (7) and Lemmas 3 and 4.1,
\[
|n-k|_{A_2} : \tau_2[\eta][x_1 \leftarrow t_1[\eta]],
\]
i.e.,
\[
|n-k|_{A_2} : \sigma_2(x_1 \leftarrow t_1[\eta]),
|n-k|_{P_2(x_1 \leftarrow t_1[\eta])[v_2 \mapsto A_2].
\]

The remaining to show is
\[
|n-k|_{P_2'[\eta][v_2 \mapsto A_2].
\]

From (5) and (8), \( |n'|_{A} : \{\nu : \sigma \mid P\}[\eta] \) and \( |n'|_{A_1} : \tau_1[\eta] \)
\[
 (= \{\nu_1 : \sigma_1 \mid P_1\}[\eta]);
\]
hence, by (iii), we have
\[
|n'|_{A_1} : \{\nu, \nu_1 : \sigma \times \sigma_1 \mid P_2'\}[\eta].
\]

Hence, \( \models_n' P_2'[\nu_2 \mapsto AA_1][\eta] \) (\( P_2'[\eta][\nu_2 \mapsto AA_1][\eta] \)) then from

Lemma 3 and (9), and since \( n' - (k - k_0 - k_1) = n - k \), we have
\[
\models_{n-k} P_2'[\eta][v_2 \mapsto A_2].
\]

**[T-PAIR]** **[T-FST]** **[T-SND]** These cases are similar to (and easier than) the case (T-APP).

**[T-FAIL]** For given \( n, \eta \), and \( V \) such that \( \eta \cup \{x \mapsto V\} \models_n \Gamma, x : \{\nu : \sigma \mid \bot\} \), we show simply contradiction, instead of \( |n|_{\eta} \) fail : \( \tau[\eta][x \mapsto V].
\]

G. Consistency

G.1 Consistency and its Sufficient Condition

Figure 12 defines a type system for a term \( t \) and a multiplicity type \( \phi \). For a derivation of \( \Phi \vdash_c t : \phi \) in this type system, we can define a multiplicity annotation \( T(t) \) as below: every subterm \( t' \) of \( t \) has the judgement \( \Phi \vdash_{t'} c : \phi' \) in the derivation, where
\[
\phi' = \prod_{n=1}^m \int \times \prod_{n=1}^m (\phi_j \rightarrow \phi_j') \mid M
\]
and then we can define \( \tau \mid \phi, \phi' \mid M \).

For a multiplicity annotation \( T \) of a term \( t \) and a multiplicity type \( \phi, T \) and \( \phi \) are consistent if \( T \) is the multiplicity annotation defined as above from some derivations of \( \Phi \vdash_c t : \phi \) with some \( \Phi \).

We also call such pair \( (T, \phi) \) consistent pair for \( t \). Conversely, for \( (T, \phi) \) and \( \phi \), such derivation is unique if exist; thus, for a closed term \( t \), we (can) identify consistent pairs \( (T, \phi) \) with derivations of \( \vdash_{t} t : \phi \).

The next proposition gives a sufficient condition for consistency, which can be used also to automatically guess consistent multiplicity annotations.

Before that, we prepare terminology and a lemma.

A multiplicity annotation \( T \) of a term \( t \) is constant with \( k \) \( (k \geq 0) \) if, for any subterm \( t' \) whose simple type is \( \int^n \times \prod_{j=1}^n (\tau_j \rightarrow \tau_j') \), \( T(t')(j) = k \) if \( \tau_j \rightarrow \tau_j' \) is depth-1 and
occurrences of judgments are constant with Φ reduction we have the derivation of Φ with whose all the multiplicity types are constant with k. For a multiplicity type judgment Φ ⊢ c t : φ, we say it is constant with k if all the multiplicity types in Φ and φ are constant with k.

Lemma 8. If Φ ⊢ c t : φ is constant with k, there is a derivation of Φ ⊢ c t : φ whose all the occurrences of judgments are constant with k.

Proof. By induction on t: for any Φ ⊢ c t : φ there is one rule-schema among the ten rule-schemata in Figure 12 whose conclusion part agrees with Φ ⊢ c t : φ, and there is at least one rule instance of the rule-schema whose assumption part consists of only judgments that are constant with k.

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

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

By Lemma 8, there is a derivation of Φ ⊢ c t : φ 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 Φ ⊢ c t : φ.

G.2 Multiplicity Annotations for Reduced Terms

Here we prove the subject reduction property of the type system for multiplicity types, and then we define a multiplicity annotation for a reduced term (used as T′ in Lemma 14).

Lemma 10 (Substitution Lemma). If Φ1, x : φ1 ⊢ c t : φ2 and Φ1 ⊢ c V : φ1, then Φ1 ⊢ c [x := V] t : φ2.

Proof. By induction on the derivation of Φ1, x : φ1 ⊢ c t : φ2.

Proposition 11 (Subject Reduction). If Φ c t : φ and t → t′, then Φ ⊢ c t′ : φ.

Proof. Straightforward indiction on the derivation of Φ ⊢ c t : φ except for the case of t = fix(f, λx. t′) V. The case is shown by using Lemma 10.

Now, for a multiplicity annotation T of a term t and a multiplicity type φ such that T and φ are consistent, suppose t → t′. We have a derivation of Φ c t : φ with some Φ and by the subject reduction we have the derivation of Φ c t′ : φ. Thus we have a multiplicity annotation T′ of t′ that is consistent with φ; for this, we write as T → T′. It is easily shown that the definition of T′ is independent of the choices of Φ and a derivation of Φ ⊢ c t : φ.

H. Soundness of Verification by (−)T

Here we prove Theorem 1, the soundness of the verification by (−)T. The definition of consistence is given in Appendix G.

We prove the soundness by dividing it four parts corresponding to (−)S1, (−)S2, (−)S3, and (−)S4:

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

\[ \models (t)_{S}^{3} : (\tau)_{S}^{3} \implies \models t : \tau. \]

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

\[ \models (t)_{T}^{3} : (\tau)_{T}^{3} \implies \models t : \tau. \]

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

All the above propositions can be proved in a similar way. Among them, the case for (−)S2 is the most subtle since it use multiplicity annotation; so we basically focus on this case.

We need the following lemma:

Lemma 14 (Simulation Lemma for (−)S2). For a multiplicity annotation T of a term t, and a multiplicity type φ such that T and φ are consistent, if

\[ t \rightarrow t′ \text{ (with } T \rightarrow T′) \]

then there are some natural numbers n, n′, and a term t′′ such that

\[ (t)_{T}^{3} \rightarrow^{n} n′ \left( (t′)_{T′}^{3} \rightarrow^{n′} n′ \right) \]

\[ n - n′ = \begin{cases} 2 & \text{if the redex of } t \text{ is of the form of application} \\ 1 & \text{otherwise} \end{cases} \]

Proof. We prove the lemma in Appendix I.

As seen above, (−)S2 (and also (−)S1 and (−)S4) does not “simulate” reduction exactly on the number of reduction-steps when a redex is of the form of application. For this, a proof of Proposition 13 has some complication since the semantics of types is given by step-indexed logical relation, (which was adopted to prove the soundness of the refinement type system in Appendix B). We separate such the complication and put in Lemma 18: i.e., we introduce another semantics of types |=LR, which is defined in Figure 13 by usual logical relation without step-indexing; we prove a certain equivalence between the two semantics in Lemma 18; and we prove Proposition 13 with respect to |=LR.

From the above lemma, we have the following:

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

- if \[ t \rightarrow^{n} V \text{ (with } T \rightarrow^{n} T′) \] for some n, then \[ (t)_{T}^{3} \rightarrow^{*} (V)_{T′}^{3} \]
  - where \( (V)_{T′}^{3} \) is a value,
- if \( t \rightarrow^{*} \text{ fail} \), then \( (t)_{T}^{3} \rightarrow^{*} \text{ fail} \),
- if \( t \uparrow \), then \( (t)_{T}^{3} \uparrow \).

Now we prove Proposition 13. We write =o for the observational equivalence.

Proof of Proposition 13. We prove that

\[ \models_{LR} (t)_{T}^{3} : (\tau)_{T}^{3} \implies \models_{LR} t : \tau \]

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

\[ \tau = \{ \nu : \text{int} | P \} \] From Lemma 17.

\[ \tau = \{ f : \nu \rightarrow \tau \} \] By assumption.

\[ (t)_{T}^{3} \rightarrow^{*} A′ \text{ then } \models_{LR} A′ : \{ \nu \rightarrow \tau \} \]

Now we suppose \( t \rightarrow^{*} A \) and show that

\[ \models_{LR} A : \{ \nu \rightarrow \tau \} \]

\[ \models_{p} P[f \rightarrow A]. \]
By Lemma 15, A is a value—so let \(A = \text{fix}(f, \lambda x_1, t_2)\) and
\[
(t)_{t'} \rightarrow^* (A)_{t'} = \text{fix}(f, \lambda x_1, t_2^*_{t'} [f \mapsto m]_{t'})
\]
for \(T'\) such that \(T \rightarrow^* T'\) and \(m \triangleq T'(A)\); hence, by the assumption,
\[
\models^L_R \text{fix}(f, \lambda x_1, (t_2^*_{t'}) [f \mapsto m]_{t'}) : (x_1 : (t_1)_{\phi}) \rightarrow (t_2)_{\phi}
\]
assuming these terms have depth-1 types. We only have to show that, for any closed value \(V_1\) of \(\tau_1\), since the observational equivalence is extensional. Now since \(V_1\) is in this normal form, it becomes clear that there is a derivation of \(\tau_1 = \phi_1\); hence, we have a consistent pair \((T_1', \phi_1)\) for \(V_1\).

From \((T'', \phi)\) and \((T_1', \phi_1)\), we obtain a consistent pair \((T_2', \phi_2)\) for \(\text{fix}(f, \lambda x_1, t_2)\).

Now we prove (10), i.e., for given \(V_1\) such that \(\models^L_R \tau_1 \rightarrow^* A_2\), we show
\[
\models^L_R A_2 : \tau_2 \rightarrow \text{fix}(f, \lambda x_1, t_2) V_1.
\]
Lemma 16. For a multiplicity annotation $T$ of a term $t$, and a multiplicity type $\phi$ over a type $\tau$ such that $T$ and $\phi$ are consistent and $\tau$ is at most order-1,

$$\vdash_{LR} t : \tau \text{ implies } \vdash_{LR} (t)^{\phi} : (\tau)^{\phi}.$$

Proof. By induction on the size of the simple types of $\tau$, similarly to the previous lemma.

$$\vdash \exists \nu : \text{int} \mid P$$

From Lemma 17.

$$\vdash\exists \nu : (x : \tau) \rightarrow \tau_2 \mid P$$

By assumption, if $t \rightarrow^* A$, then

$$\vdash\exists \nu A : (x_1 : \tau_1) \rightarrow \tau_2 \quad (15)$$

$$\vdash\exists \nu P[f \rightarrow A]. \quad (16)$$

Now we suppose $(t)^{\phi} \rightarrow^* A'$ and show that $\vdash_{LR} A' : ((f : (x_1 : \tau_1) \rightarrow \tau_2) \mid P)^{\phi}$.

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

From (15) and by Lemma 15,

$$A' = (V)^{\phi}_{\tau_2} = \text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T'])$$

and it is enough to show

$$\vdash\exists \nu \text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T']) : (x_1 : (\tau_1)^{\phi}_{\tau_1}) \rightarrow (\tau_2)^{\phi}_{\tau_2} \quad (17)$$

$$\vdash\exists \nu P[f \rightarrow \text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T'])]. \quad (18)$$

Now we prove (17), i.e., for given $V_1'$ such that $\vdash_{LR} V_1' : (\tau_1)^{\phi}_{\tau_1}$, we show

$$\vdash_{LR} \text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T']) : (\tau_2)^{\phi}_{\tau_2}[x_1 \rightarrow V_1']. \quad (19)$$

Since $\tau_1$ is order-0, $\vdash_{LR} V_1' : \phi_1$; hence we have consistent pairs $(T_1', \phi_1)$ for $V_1'$ and $(T_2', \phi_2)$ for $V_1'$. By Lemma 17, $V_1' =_{o} (V_1')^{T_1'}$ and $\vdash_{LR} V_1' : \tau_1$. Hence,

$$\text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T'])' \quad (20)$$

$$\rightarrow (t_2)^{\phi}_{\tau_2} \mid [x_1 \rightarrow V_1'][f \rightarrow \text{fix}(f, \lambda x_1, (t_2)^{\phi}_{\tau_2} [f \rightarrow T'])] \quad (21)$$

$$=_{o} (t_2)^{\phi}_{\tau_2} \mid [x_1 \rightarrow (V_1')^{T_1'}[f \rightarrow V_1'][f \rightarrow (V_1')^{T_1'}]) \quad (22)$$

$$= \{\text{by Lemma 19 in Appendix I}\}$$

$$= \{\text{by Lemma 19 in Appendix I}\}$$

$$\dashv \vdash \exists \nu V_1' : \tau_2[x_1 \rightarrow V_1']. \quad (23)$$

Now from (15), $\vdash_{LR} V_1' : \tau_2[x_1 \rightarrow V_1']$, and by induction hypothesis,

$$\vdash_{LR} (V_1')^{T_2'} : (\tau_2[x_1 \rightarrow V_1'])^{T_2'} = (\tau_2)^{\phi}_{\tau_2}[x_1 \rightarrow V_1']. \quad (24)$$

Thus, we have shown (19).

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

Lemma 17. For a multiplicity annotation $T$ of a closed term $t$, and a multiplicity type $\phi$ over a closed type $\tau$ such that $T$ and $\phi$ are consistent and $\tau$ is order-0,

$$(\tau)^{\phi} = \tau \quad \text{and} \quad (t)^{\phi} =_{o} t .$$

Proof. On types, it is clear by definition.

On terms, it is clear from Lemma 15.

Lemma 18. For a term $t$ and a type $\tau$ of at most order-2,

$$\vdash t : \tau \iff \vdash_{LR} t : \tau.$$
I. Simulation Lemma

In this section, we prove Simulation Lemma for \( z_3 \) (i.e., Lemma 14). First, we give several lemmas and definitions.

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

For derivations of \( \Phi, x' : \phi' \vdash c : \phi \) and \( \Phi \vdash c' : \phi' \), let \( T, T' \) and \( T[T'] \) be the multiplicity annotations of \( t, t' \), and \( l[x' \mapsto t'] \) defined from the derivations, respectively, then,

\[
(t[x' \mapsto t'])_{T[T']}^{z_3} = (t)^{z_3}_{T[T']} \quad [x' \mapsto t']^{z_3}_{T[T']}
\]

**Proof.** The former is straightforward by induction on derivations of \( \Phi, x' : \phi' \vdash c : \phi \) and by case-analysis of \( t \).

For given derivations below,

\[
\Phi, x' : \phi' \vdash c : \phi \quad \vdash E, x_1 : \phi_1 \vdash t_2 : \phi_2 \quad \vdash E, x_2 : \phi_2 \vdash t_3 : \phi_3
\]

we have a derivation \( \mathcal{D} \) of \( (\ldots \vdash E, t_2 : \phi_2) \). By induction hypothesis, we have a derivation \( \mathcal{D}'(\mathcal{D},\mathcal{D}') \) of \( (\ldots \vdash E, t_2' : \phi_2) \), and the below for \( \text{fix}(f, \lambda x_1. t_2[x' \mapsto t']) = \text{fix}(f, \lambda x_1. t_2)[x' \mapsto t'] \).

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

\[
\text{fix}(f, \lambda x_1. t_2)_{T[T']}^{z_3} = (f[x' \mapsto t'])_{T[T']}^{z_3}
\]

\[
= (\text{fix}(f, \lambda x_1. t_2[x' \mapsto t'])_{T[T']}^{z_3})_{T[T']}
\]

\[
= \left\{ \begin{array}{l}
\text{let } m \overset{\text{def}}{=} T[T']((\text{fix}(f, \lambda x_1. t_2[x' \mapsto t'])))
\end{array} \right.
\]

\[
\text{fix}(f, \lambda x_1. t_2[x' \mapsto t'])_{T[T']}^{z_3} = (f[x' \mapsto t'])_{T[T']}^{z_3}
\]

\[
= \left\{ \begin{array}{l}
\text{let } m \overset{\text{def}}{=} T[T']((\text{fix}(f, \lambda x_1. t_2[x' \mapsto t'])))
\end{array} \right.
\]

\[
\text{fix}(f, \lambda x_1. t_2[x' \mapsto t'])_{T[T']}^{z_3} = (f[x' \mapsto t'])_{T[T']}^{z_3}
\]

\[
= (t)^{z_3}_{T[T']} \quad [x' \mapsto t']_{T[T']}
\]

**Figure 14.** \((-)^{z_3}; \text{modified } (-)^{z_3} \text{ for evaluation contexts}

\[
s([-]) \overset{\text{def}}{=} 0
\]

\[
s(\text{op}(V, E, t)) \overset{\text{def}}{=} s(E)
\]

\[
s(\text{if } E \text{ then } t_1 \text{ else } t_2) \overset{\text{def}}{=} s(E)
\]

**Figure 15.** Step numbers of evaluation contexts

\[
E \overset{\text{def}}{=} \text{E}_1 \text{E}_2 \text{E}_3
\]

\[
\text{E}_1 \overset{\text{def}}{=} \text{E}_2 \text{E}_3
\]

\[
\text{E}_2 \overset{\text{def}}{=} \text{E}_3
\]

\[
\text{E}_3 \overset{\text{def}}{=} \text{E}_4 \text{E}_5
\]

E, we define \((E)^{z_3}_{T,T'}\) in Figure 14 and the step numbers \(s(E)\) in Figure 15. In both the definitions, we give special treatment to the case of \( \text{E}_1 \).

**Lemma 20.**

1. For any value \( V \), \((V)^{z_3}_{T} \) is a value.
2. For any evaluation context \( E \) and a multiplicity annotation \( T \) of \( (E)^{z_3}_{T} \) 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.

\[
(E)^{z_3}_{T,T'}[[t]] \rightarrow s(E) \quad (E)^{z_3}_{T,T'}[t]
\]

**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. \( \text{fix}(f, \lambda x_1. t_2)E \).

\[
\text{fix}(f, \lambda x_1. t_2)E_{T,T'}[t] \rightarrow s(E) \quad (E)^{z_3}_{T,T'}[t]
\]

Now we prove Lemma 14; recall the statement: For a multiplicity annotation \( T \) of a term \( t \), and a multiplicity type \( \phi \) such that \( T \)}
Lemma 21. for these notions. This extension is consistent since we so far used
Proof of Lemma 18.
Proof.
\[ \phi \] is also clear from 1 and 2.
\[ n \] otherwise, and
\[ \text{for } \] \( n \text{ natural numbers} \)
\[ \text{then there are some natural numbers} \)
\[ \text{if } \]
\[ \text{for } \]
\[ \text{then } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
\[ \text{if } \]
We reserve enough number (at most \( k \)) of fresh variables \( v_1, v_2, \ldots \), and then for each \( i \leq p \) we define \( V^{(i)} \equiv v_i \) if \( V^{(i)} \) has function type, and \( V^{(i)} \equiv V^{(j)} \) if \( V^{(i)} \) has order-0 type. Also, we define

\[
P \overset{\text{def}}{=} \text{let } \nu_{p+1} = V^{(1)} \ldots V^{(p)} \text{ in } P_{p+1}.
\]

From the assumption,

\[
P[v_i \mapsto V^{(i)}]_{1 \leq p} \longrightarrow^k A'.
\]

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

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

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

Now we show that the reduction sequence of \( P \) is "sufficiently long". First, if \( 1 < j^{(p+1)} \), for some \( k_1 \) and some value \( V_2' \)

\[
V_1 V^{(1)} \longrightarrow^{k_1} V_2
\]

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

\[
(V_1 V^{(1)})_{[v_i \mapsto V^{(i)}]} = V_1 V^{(1)} \longrightarrow V_2 [v_i \mapsto V^{(i)}]
\]

and since \( V_1 V^{(1)} \) is closed term,

\[
V_2'_{[v_i \mapsto V^{(i)}]} = V_2' \text{ and } k_1 = k_1.
\]

Next, if \( 2 < j^{(p+1)} \), similarly to the above, for some \( k_2 \) and some value \( V_3' \)

\[
V_2 V^{(2)} \longrightarrow^{k_2} V_3
\]

By Lemma 21,

\[
(V_2 V^{(2)})_{[v_i \mapsto V^{(i)}]}_{1 = 1, 2} = V_2 V^{(2)} \longrightarrow V_3 [v_i \mapsto V^{(i)}]_{1 = 1, 2}
\]

and since \( V_2 V^{(2)} \) is closed term,

\[
V_3'_{[v_i \mapsto V^{(i)}]}_{1 = 1, 2} = V_3' \text{ and } k_2 = k_2.
\]

Repeating this, there exist values \( V_i(i = 2, \ldots, j^{(p+1)}) \)—let \( V_j^{(j^{(p+1)})} \) be of the form \( \text{fix}(f, \lambda x. t) \)—such that

\[
P \overset{\text{def}}{=} \text{let } \nu_{p+1} = V_1^{(1)} \ldots V_p^{(p)} \text{ in } P_{p+1}
\]

\[
\longrightarrow^{k_2} \text{let } \nu_{p+1} = V_2^{(2)} \ldots V_p^{(p)} \text{ in } P_{p+1}
\]

\[
\ldots
\]

\[
\longrightarrow^{k_1} \text{let } \nu_{p+1} = V_j^{(j+1)} V_j^{(j+1)} \ldots V_p^{(p)} \text{ in } P_{p+1}
\]

\[
= \text{let } \nu_{p+1} = \text{fix}(f, \lambda x. t) V_j^{(j+1)} \ldots V_p^{(p)} \text{ in } P_{p+1}
\]

\[
\longrightarrow \text{let } \nu_{p+1} = t[x \mapsto V_j^{(j+1)}]_{[f \mapsto V_j^{(j+1)}]} V_j^{(j+1)+1} \ldots V_p^{(p)} \text{ in } P_{p+1}
\]

\[
\longrightarrow E^{(p+1)}[v_j^{(j+1)} m^{(p+1)}]
\]

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

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

where \( E^{(p+1)}[v_i \mapsto V^{(i)}]_{1 \leq p} \) is a (closed) evaluation context. Since \( P[v_i \mapsto V^{(i)}]_{1 \leq (p+1)} \overset{k'}{\longrightarrow} A' \), there exist \( A^{(p+1)} \) and \( k^{(p+1)} \leq k \), \( k' \leq \Sigma_{1 \leq j^{(p+1)} k_j} \), such that

\[
V_j^{(j^{(p+1)})} m^{(p+1)} \longrightarrow^{k^{(p+1)}} A^{(p+1)}.
\]

Since

\[
E^{(p+1)}[v_j^{(j^{(p+1)})}] : \tau_j^{(j^{(p+1)})} \longrightarrow \ldots \tau_j^{(j^{(p+1)})}
\]

and from the assumption

\[
k^{(p+1)} \leq \Sigma_{1 \leq j^{(p+1)}} k_j \leq n^{(p+1)}
\]

we have

\[
E^{(p+1)}[v_j^{(j^{(p+1)})}] = A^{(p+1)} : \tau_j^{(j^{(p+1)})} \longrightarrow \ldots \tau_j^{(j^{(p+1)})}
\]

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

\[
q^{(p+1)} \overset{\text{def}}{=} m^{(p+1)} - n^{(p+1)} = k^{(p+1)} - n^{(p+1)}
\]

then,

\[
E^{(p+1)}[V^{(p+1)}] \overset{\text{def}}{=} E^{(p+1)}[v_j^{(j^{(p+1)})}]
\]

Also, we define \( E^{(p+1)}[V^{(p+1)}] \) if \( V^{(p+1)} \) has function type, and \( E^{(p+1)}[V^{(p+1)}] \) if \( V^{(p+1)} \) has order-0 type. We have defined the sequence (27) for \( i = p + 1 \).

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

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

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

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

1. \( j^{(i)} < i \) and

\[
E^{(i-1)}[V^{(i-1)}] \overset{*}{\longrightarrow} E^{(i)}[v_j^{(j^{(i)})} m^{(i)}]
\]

where we regard \( E^{(i)}[V^{(p+1)}] \) as \( P \); also we have

\[
E^{(h)}[V^{(h)}] \overset{*}{\longrightarrow} A'.
\]

2. for some \( k' \geq 1 + \Sigma_{1 \leq j^{(i)}} k_j \)

\[
P \longrightarrow^{k'} E^{(i)}[v_j^{(j^{(i)})} m^{(i)}]
\]

3. \( q^{(i)} = q^{(j^{(i)})} \) \( r^{(i)} = r^{(j^{(i)})} + 1 \) \( n^{(i)} = n^{(j^{(i)})} - k^{(i)} \geq 0 \)

\[
V_j^{(j^{(i)})} m^{(i)} \longrightarrow^{k^{(i)}} V_j^{(j^{(i)})}
\]

where if \( r^{(i)} = l^{(i)} + 1 \), then \( \tau_r^{(i)} \longrightarrow \ldots \tau_r^{(i)} \) is regarded as \( \tau_r^{(i)} \).

In the above inductive definition of the sequences, we explain \( n^{(j^{(i)})} - k^{(i)} \geq 0 \) in the case when \( i > p \), which is a bit subtle. First, for any \( i, j \) above can be shown in the same way as the case
when \( i = p + 1 \) above. Now for any natural number \( e \),
\[
\begin{align*}
n^{(i)} &= n^{(i-1)} - k^{(i)} = n^{(i-1)} - (k^{(i-1)} + k^{(i)}) = \\
&= n^{(i-1)} - (k^{(i-1)} + \ldots + k^{(i)})
\end{align*}
\]
where \( j^{(i)} \equiv j^{(i-1)} \) and \( j^{(0)} \equiv i \). Let \( e \) be a natural number such that \( j^{(e)} \leq p \); since now \( i > p, e > 0 \). We prove \( n^{(i)} - k^{(i)} \geq 0 \) by showing
\[
k^{(i-1)} + \ldots + k^{(i)} \leq \sum_{j > j^{(i)}} k^{(j)} \leq n^{(i)}.
\]
Since \( j^{(i)} \leq p \), the right inequality above follows from the assumption of this lemma. By the above \( 2 \) where we substitute \( j^{(i)} \) for \( i \), for some \( k' \geq 1 + \sum_{j < j^{(i)}} k^{(j)} \),
\[
P \rightarrow^{*} k' \ E^{(i-1)}[v^{(i)} m^{(j-1)}],
\]
and also by the above \( 3 \) where we substitute \( j^{(i-1)} \), \ldots, \( i \) for \( i \),
\[
V^{(i)}[m^{(i-1)}] \rightarrow^{*} k^{(i-1)} \ V^{(j^{(i-1)})} \ V^{(i)},
\]
Hence, by Lemma 21, \( k' + k^{(j^{(i-1)})} + \ldots + k^{(i)} \leq k \); thus,
\[
k^{(i-1)} + \ldots + k^{(i)} \leq k - k' \leq \sum_{j > j^{(i)}} k^{(j)}.
\]
Now, by using the above sequences, we define \( V^{(i)} \) for the witnesses of our goal. For each \( i \), we define a finite set \( D^{(i)} \equiv \{ i' \mid i = j^{(i')} \} \); by the above \( 3 \), for each \( i' \in D^{(i)} \),
\[
V^{(i)}[m^{(i)}] \rightarrow^{*} V^{(i)}. \]
Let \( q \in \{ 1, \ldots, p \} \). We define \( V^{(i)} \) for \( i \) such that \( q^{(i)} = q \). For \( i \) such that \( q^{(i)} = q \) and \( r^{(i)} = l_{q} + 1 \), we define \( V^{(i)} \) as \( V^{(q)} \), which is a value of order-0. Next, by induction on \( r = l_{q}, \ldots, 1 \), for any \( i \) such that \( q^{(i)} = q \) and \( r^{(i)} = r \), we define \( V^{(i)} \) as \( \lambda x \cdot \text{case } x \text{ of } m^{(i)} \rightarrow V^{(i)}[m^{(i)}] \)
\[
: m^{(q)} \rightarrow V^{(q)}[m^{(q)}]
\]
where \( \{ i_{1}, \ldots, i_{d} \} = D^{(i)} \) and \( \Omega \equiv \text{fix}(f, \lambda x \cdot f x) \lambda x_{r+1}, \ldots, x_{n} \).
Now, we show \( 26 \) with \( V^{(i)}[m^{(i)}] \) as witnesses of \( \exists V_{q} \) for \( i \in \{ 1, \ldots, p \} \).
By the induction by which we defined \( V^{(i)}[m^{(i)}] \), it can be easily shown that, for any \( i \),
\[
|_{v_{i}}^{LR}_{\tau} V^{(i)} \ 
\]
Also, by the same induction, it is clear that, for any \( i \) and \( i' \in D^{(i)} \), \( V^{(i)}[m^{(i)}] \rightarrow^{*} V^{(i)}[m^{(i)}] \); thus, for any \( i \), since \( V^{(i)}[v_{j} \rightarrow V^{(j)}[m^{(j)}]] \) is an evaluation context,
\[
\begin{align*}
\left( E^{(i)}[v_{j} \rightarrow V^{(j)}[m^{(j)}]] \right)_{j < l_{h}} &\rightarrow^{*} V^{(i)}[m^{(i)}] \ \\
\left( E^{(i)}[v_{j} \rightarrow V^{(j)}[m^{(j)}]] \right)_{j \leq l_{h}} &\rightarrow^{*} V^{(i)}[m^{(i)}].
\end{align*}
\]
\[
M \text{ (programs):}= (x_{1}, \ldots, x_{n}) \mid \text{if } x \text{ then } M_{1} \text{ else } M_{2} \ 
\]
\[
\text{let } x = e \in M \ 
\]
\[
\text{if } e \text{ then } t_{1} \text{ else } t_{2} \ \text{let } x = e \in t \ 
\]
\[
\text{fix}(f, \lambda x_{1}, \ldots, x_{n}). t \ | f(x_{1}, \ldots, x_{n}) | \text{ fail}
\]
\[
\text{Figure 17. Normal form before } (-)^{2}
\]
Now, \( P \) has the following “reduction”:
\[
P \rightarrow^{*} E^{(p+1)}[v_{j(p+1)} m^{(p+1)}] \ 
\]
\[
\rightarrow^{*} E^{(p+1)}[v_{j(p+1)}] \ 
\]
\[
\rightarrow^{*} E^{(i-1)}[v_{j(i-1)}] \ 
\]
\[
\rightarrow^{*} E^{(0)}[v_{j(0)}] \ 
\]
\[
\rightarrow^{*} A'
\]
where the dashed arrow \( \rightarrow^{*} \) means that, if we substitute \( V^{(i)}[m^{(i)}] \) for \( v_{i} \) for all \( i \), there is the reductions by \( 28 \). Thus, by the substitution and Lemma 21, we have
\[
\text{let } v_{p+1} = t v_{p} \ldots v_{1} \text{ in } P_{p+1} \ 
\]
\[
= P[v_{j} \rightarrow V^{(j)}]_{j \leq h} \ 
\]
\[
\rightarrow^{*} (E^{(p+1)}[v_{j} \rightarrow V^{(j+1)}])_{j \leq h} \ 
\]
\[
\rightarrow^{*} (E^{(p+1)}[v_{j} \rightarrow V^{(j)})]_{j \leq h}) \ 
\]
\[
\rightarrow^{*} \left( E^{(i-1)}[v_{j} \rightarrow V^{(j-1)}] \right)_{j \leq h} \ 
\]
\[
\rightarrow^{*} \left( E^{(i)}[v_{j} \rightarrow V^{(j)})] \right)_{j \leq h} \ 
\]
\[
\rightarrow^{*} \left( E^{(0)}[v_{j} \rightarrow V^{(0)})] \right)_{j \leq h} \ 
\]
\[
\rightarrow^{*} A'[v_{j} \rightarrow V^{(j)}]_{j \leq h} = A'
\]
where recall that \( V^{(i)} = V^{(i)} \) if the type is order-0. This completes the proof. 

\[
\text{K. Definition of } (-)^{2}
\]
In this section, we define this refined transformation \( (-)^{2} \). The refinement is needed for both \( (-)^{3} \) and \( (-)^{4} \), so we define \( (-)^{5z} \), which is a modification of \( ((-)^{3})^{5} \), and define \( (-)^{2} \) as \( ((-)^{2})^{5z} \). Note that \( (-)^{2} \) is identity on terms. For the simplicity of the definition, we assume without loss of generality that input programs of \( (-)^{2} \) (and hence \( (-)^{3} \)) are in a variant of A-normal form defined in Figure 17. Here, we write \( \lambda x_{1}, \ldots, x_{n}. t \) for \( \lambda x_{1} \cdot \ldots \cdot \lambda x_{n} \cdot t \) where \( x \) is a fresh variable. We redefine the transformation \( (-)^{5z} \) according to the normal form; the essential part of the new definition of \( (-)^{2z} \) is shown in
\[(\text{fix}(f, \lambda(x_1, \ldots, x_n), t'))^{z_1} \overset{\text{def}}{=} \text{fix}(f, \lambda(x_1, \ldots, x_n))\]

(\text{let } x = f(x_1, \ldots, x_n) \text{ in } t^{z_1})^{z_1} \overset{\text{def}}{=} \text{fix}(f, \lambda(x_1, \ldots, x_n))

let \( z = f(x_1, \ldots, x_n) \) \text{ in } let \( x = \text{pr}_z \) \text{ in }

let \( x_1' = \text{pr}_z(\text{pr}_y z) \) \text{ in } \ldots \text{let } \( x_n' = \text{pr}_n(\text{pr}_y z) \) \text{ in } (t^{z_1})_{x_1 \mapsto x_1', \ldots, x_n \mapsto x_n'}

Figure 18. \((-)^{z_1}\) for normal forms

t := \langle x_1, \ldots, x_n \rangle \mid \text{if } x \text{ then } t_1 \text{ else } t_2 \mid \text{let } x = e \text{ in } t
eq:\langle \text{f}, \text{pr}(x_1, \ldots, x_n), t \rangle \mid \text{let } x_1, \ldots, x_n = \text{pr}(x) \text{ in } (t)^{z_1}

\begin{align*}
\text{let } & \lambda(y). y \text{ in } \text{let } x = g \text{ in } \text{let } w((\alpha)^j_j) = B^2_\delta((\alpha)^j_j) \text{ in } \text{let } (p); \text{assume } (p') \text{; } x
\end{align*}

Figure 19. Normal form before \((-)^{z_4}\)

Figure 18. Output programs of this \((-)^{z_1}\) are in another normal form defined in Figure 19; which is the domain of the transformation \((-)^{z_4}\), which is defined in the next subsection.

K.1 Definition of \((-)^{z_4}\) and Soundness of \((-)^{z'}\)

Here, we formalize the idea explained in Section 4.2 as a transformation \((-)^{z_4}\).

The transformation of \((-)^{z_4}\) for types is the same as \((-)^{z_5}\) for terms. The definition uses an auxiliary function \text{InstVar} defined below, and this is the essential part: \text{InstVar} synthesizes new applications (like \(fg(x, x)\)) and inserts the assumptions illustrated above. In the figure, \(B\) is a set of bindings (i.e., pairs of variables and expressions) that are used in \text{InstVar}. As \((-)^{z_5}\) and \((-)^{z_4}\) also depends on a multiplicity annotation \(T\). Since occurrences of subterms of \(t\) correspond variables in the normal form of \(t\), any multiplicity annotation \(T\) is defined as a function from variables to multiplicities. The rules are almost the same except for applications, where we insert assumptions related to the function and its arguments by using \text{InstVar}.

Now, we define the auxiliary function \text{InstVar}:

\[\text{InstVar}(g, T, B, t) \overset{\text{def}}{=} \langle \lambda y. y \text{ in } \text{let } w((\alpha)^j_j) = B^2_\delta((\alpha)^j_j) \text{ in } \text{let } (p); \text{assume } (p') \text{; } x \rangle\]

where \(\langle \alpha^j_j \rangle \in \prod_{j \in m} \text{App}^j_k\) and \(k = \prod_{j \in m} \text{App}^j_k\) : the function \(B^2_\delta\), the two formulas \(p\) and \(p'\), the set \(\text{App}^j_k\) and the variables \(w((\alpha)^j_j)\) are defined below.

Before the formal definition of the predicates \(p\) and \(p'\), we explain their semantical meaning. By \((-)^{z_5}\) and \((-)^{z_4}\), each variable is unchanged (i.e., \((x)^{z_5} \overset{\text{def}}{=} x\) and \((x)^{z_4} \overset{\text{def}}{=} x\)), although by \((-)^{z_5}\), each subterm of a function type (i.e., \(\text{fix}(f, \lambda.x, t)\) and \(f\) in \(\text{fix}(f, \lambda.x, t)\)) is duplicated; and by \((-)^{z_4}\), each subterm of a tuple type (i.e., \((t_1, \ldots, t_n, t'_1, \ldots, t'_n)\)) is transformed to the product of the functions, where the product of functions \(t\) and \(t'\) means \(\lambda(x, x'). (t, t \times t')\). Hence, a verifier cannot necessarily infer that \((-)^{z_4}\) is a function variables behave as the product of duplicated functions, while in fact behave so since they are instantiated with some closed ground terms in programs. The assumed predicates \(p\) and \(p'\) state just that all the function variables behave as the product of duplicated functions \((p, p')\) correspond to “product of functions” and “duplication", respectively.

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

1. Let \(z = \varphi'\) if \((g = \text{pr}_n z') \in B\) 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 \((-)^{z'}\). Before applying \((-)^{z}\), the simple type of \(z\) is of the following form:

\[\text{int}^n \times \prod_{j = 1}^{\tau_j} (r_j \rightarrow r'_j)\]
and hence the type of $g$ is of the form $\tau_0 \to \tau_k'$; in the case $z = g$, we regard $n = 0$ and $k = m = 1$. After $(-)^x$, $(z)^x$ $(= z)$ has the following simple type:

$$\text{int}^n \times \bigg( \sum_{j=1}^{m} \left( \left( \tau_j \right)^{x_j}_{T'} \right)^{\#_j} \to \bigg( \sum_{j=1}^{m} \left( \left( \tau_j \right)^{x_j}_{T} \right)^{\#_j} \bigg) \bigg)$$

(30)

where $m_j$ is the multiplicity of $\text{pr}_{x_j}^\#$, and we access to the second component (function part) by $\text{pr}_{y_j}^\#$. Also, after $(-)^x$, the type of $g$ becomes $(\left( \tau_k \right)^{x_k}_{T'} \#_k) \to (\left( \tau_k \right)^{x_k}_{T} \#_k)$ (because of the consistency of $T$, we can show that $T(g) = m_k$). Hence, the type of $\tilde{y} = (y_l)_{l \in n_k}$ is $(\left( \tau_k \right)^{x_k}_{T} \#_k)$. Note that variables introduced when defining $(-)^x$, such as $y_l$, have no "types before $(-)^x$".

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

$$\text{App}_j^{\#} \overset{\text{def}}{=} \{ (u, v, w) \bigg\mid \begin{array}{l}
    (v = \text{pr}_{x_j}^\# z, (w = vu) \in B, \\
    \text{depth}(v) = 1
  \end{array}\}$$

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

$$\text{App}_j \overset{\text{def}}{=} \begin{cases}
    \text{App}_j^{\#} \cup \{ (y_l, \ldots, y_{k_j}) \} & \text{if } j = k \\
    \text{App}_j^{\#} & \text{otherwise, if } \text{App}_j^{\#} \text{ is non-empty}
  \end{cases}$$

(31)

where $y$ is the bound variable in (29).

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

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

and "function part" as

$$\text{fun}_j : \text{App}_j \to \text{Term} \overset{\text{def}}{=} \begin{cases}
    (u, v, w) \mapsto v & y_l \mapsto g \\
    (\perp, v) \mapsto v
  \end{cases}$$

3. We further add the information of multiplicity $m_j$ for the notions thus defined:

$$\text{App}_j^{\#} \overset{\text{def}}{=} \{(a_i)_i \in \text{App}_j^{\#} \mid \text{fun}_j(a_i) = \text{fun}_j(a_1)\}$$

$$\text{arg}_j^{\#} : \text{App}_j^{\#} \to \text{Term}^{\#}, \quad \text{arg}_j^{\#}(a_i) \overset{\text{def}}{=} \text{arg}_j(a_i)$$

$$\text{fun}_j^{\#} : \text{App}_j^{\#} \to \text{Term}, \quad \text{fun}_j^{\#}(a_i) \overset{\text{def}}{=} \text{fun}_j(a_i).$$

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

$$B_j^g : \prod_{i \in \text{App}_j} B_j \to \prod_{i \in \text{Term}} \text{Term} \overset{\text{def}}{=} \left( \text{pr}_j \left( \text{pr}_{x_j}^\# z \left( \text{fun}_j^{\#}(a_1), \ldots, \text{fun}_j^{\#}(a_m) \right) \right) \right)_{j \in m}$$

(32)

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

$$p \overset{\text{def}}{=} \land_{j \in m, \text{fun}_j = \text{fun}_j^{\#}} \left( \text{arg}_j(a_j) = \text{arg}_j^{\#}(a_j) \Rightarrow \text{pr}_j w^{((\text{fun}_j)(a_j))} = \text{pr}_j^{\#} w^{((\text{fun}_j^{\#})(a_j))} \right)$$

$$p' \overset{\text{def}}{=} \land_{j \in m, (\text{fun}_j, \text{fun}_j^{\#}) \in \text{App}_j^{\#}} \left( w = \text{pr}_j \text{pr}_j^{\#} w^{((\text{fun}_j)(a_j))} \right)$$

where we prepare a fresh variable $w^{((\text{fun}_j)(a_j))}$ for each application $(a_j)$.  

\section{Soundness of Verification by $(-)^x$}

Here we prove the soundness of verification by $(-)^x$.

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

$$\models (t)^x : (\tau)^{x}_{\phi} \text{ implies } \models (t)^y : (\tau)^{y}_{\phi}.$$ 

\textbf{Theorem 22 (Soundness of Verification by $(-)^x$).} Let $t$ be a closed term and $\tau$ be a type of at most order-2. Let $T$ be a multiplicity annotation for ($(t)^{21}_{x}$) and $\phi$ be a multiplicity type for ($(\tau)^{21}_{\phi}$) and suppose that they are consistent and $\tau \leq_{\text{rel}} \phi$. Then,

$$\models (t)^{x}_{\tau} : (\tau)^{x}_{\phi} \text{ implies } \models t : \tau.$$ 

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

By Lemma 16, we have $\models_{\text{LR}} (V_1)^{j}_x : (\tau_1)^{j}_x$, hence $\models_{\text{LR}} (V_1)^{j}_x : (\tau)^{j}_x$. By the assumption that $\models_{\text{LR}} (t) : (\tau)^{j}_x$, we have $\models_{\text{LR}} (t)^{j}_x : (V_1)^{j}_x : (\tau)^{j}_x$. Now, $(t V_1)^{j}_x \leq_{o} (t)^{j}_x (V_1)^{j}_x$ since the left hand side is the right hand side plus assume expressions; hence, $\models_{\text{LR}} (t V_1)^{j}_x : (\tau)^{j}_x$. Since $(\tau)^{j}_x$ is order-0, by Lemma 23 below, we have $\models_{\text{LR}} (t V_1)^{j}_x : (\tau)^{j}_x$, and by Lemma 17, $\models_{\text{LR}} t V_1 : \text{int}$.  

\textbf{Lemma 23.} For any closed A-normal form $t$ (defined in Figure 17), a type $\tau$ of order-0, and a consistent pair of a multiplicity annotation $T$ of $t$ and a multiplicity type $\phi$ over $\tau$, $(t)^{x}_{\tau}$ and $(t)^{y}_{\tau}$ are observationally equivalent; and hence,

$$\models (t)^{x}_{\tau} : (\tau)^{x}_{\phi} \iff \models (t)^{y}_{\tau} : (\tau)^{y}_{\phi}.$$ 

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

First, we define $(-)^{x_{\phi}}$ by eliminating $\text{InstVar}$ from $(-)^{x_{24}}$; i.e., in Figure 20, we drop the subscript $B$ and replace the case of application with the below

$$\left( f(x_1, \ldots, x_n, g_1, \ldots, g_m) \right)^{x_{\phi}}_{t^x} \overset{\text{def}}{=} \text{pr}_1(f_{\tau}(z^{T(f)}))$$

where

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

Since $(\tau)^{x_{\phi}}$ is just an A-normal form version of $((-)^{x_{24}})$, it suffices for the lemma to prove that $(-)^{x_{\phi}}$ is observationally equivalent to $(-)^{x_{24}}$ for any ground closed A-normal form $t$. That is, we will prove that assume expressions inserted by $\text{InstVar}(-)$ are satisfied and hence can be removed without changing the meaning.

The assume expressions inserted by $\text{InstVar}(-)$ are properties satisfied naturally by the image of $(-)^{x_{24}}$; and in fact, it is easy to prove that $(V)^{x_{24}}$ satisfies the properties by unfolding the definition of $(-)^{x_{24}}$ in Figure 20. However it is not obvious if such $V$ are arbitrary terms $e$, so we transform such $e^{x_{24}}$ to a term of the form $(V)^{x_{24}}$. We call this transformation N-reduction; it is defined similarly to evaluation, but keeps the form of A-normal form.
In order for N-reduction to terminate, we can assume that the given whole (ground closed) term \( t \) terminates, because when \( t \) diverges, by Lemma 15 and since \((t)^{^2}_{^5}\) \(\leq_o\) \((t)^{^2}_{^5}\), both \((t)^{^2}_{^5}\) and \((t)^{^2}_{^5}\) diverge and then the current lemma holds. Since N-reduction is simulated by the evaluation, if evaluation terminates, N-reduction also terminates.

Though intuitively we reduce \((t)^{^2}_{^5}\), in fact we define N-reduction for \( t \), and we show \((-)^{^2}_{^5}\) preserves N-reduction to observational equivalence, i.e.,
\[
 t \longrightarrow_N t' \quad \text{implies} \quad (t)^{^2}_{^5} =_o (t')^{^2}_{^5} .
\]

Also \((-)^{^2}_{^5}\) preserves N-reduction to observational equivalence. Now since N-reduction terminates for given \( t \), we have the normal form \( t' \), and for the normal form of N-reduction, it is easy to show
\[
 (t')^{^2}_{^5} =_o (t)^{^2}_{^5} .
\]

Thus we can show
\[
 (t)^{^2}_{^5} =_o (t')^{^2}_{^5} =_o (t')^{^2}_{^5} =_o (t)^{^2}_{^5} .
\]

In the rest of this overview, we explain the above idea concretely with the following example of \( t \):
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda x'. t') \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= f x \text{ in } t''
\end{align*}
\]

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

Now \((t)^{^2}_{^5}\) is
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda (x'_1, x'_2). ((t')^{^2}_{^5} [x' \mapsto x'_1])_{i=1,2}) \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= \text{pr}_1(f''(x, x) \text{ in }) \\
\text{in } (t'')^{^2}_{^5}
\end{align*}
\]

Since \( f \) is value bound to the value \( \text{fix}(f', \lambda x'. t') \), we could calculate by the definition in Figure 20 that the (body of) \( f \) in \((t)^{^2}_{^5}\) is syntactically the product
\[
\lambda(x'_1, x'_2). ((t')^{^2}_{^5} [x' \mapsto x'_1], (t')^{^2}_{^5} [x' \mapsto x'_2])
\]

of the duplication of \( \text{fix}(f', \lambda x'. t') \) (it is not the case if \( f \) in \( t \) is bound to a non-value). Then, it is easy to show that such the syntactic product of the duplication satisfies assume \((\ldots)\) (see Appendix L.4 for details; especially, Lemma 27); here recall that, as we explained in Section K.1, the predicates of the assumption-expressions inserted by \text{InatVar}(\ldots) just state that all the function variables after applying \((t')^{^2}_{^5}\) behave as the product of duplicated functions. Thus, we can remove the assume expression, and by simple reductions, we have
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda (x'_1, x'_2). ((t')^{^2}_{^5} [x' \mapsto x'_1])_{i=1,2}) \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= \text{pr}_1(f(x, x)) \text{ in } (t'')^{^2}_{^5}
\end{align*}
\]

Now we want to transform the non-value \( \text{pr}_1(f(x, x)) \) to the form \((V)^{^2}_{^5}\), as \( f \) was so and it helped the removal of assume as above.

Clearly, the above is observationally equivalent to
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda (x'_1, x'_2). ((t')^{^2}_{^5} [x' \mapsto x'_1])_{i=1,2}) \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= \text{pr}_1(((t')^{^2}_{^5} [x' \mapsto x][f' \mapsto f])_{i=1,2}) \text{ in } (t'')^{^2}_{^5} .
\end{align*}
\]

Since our language is deterministic, \( \text{pr}_1(t, t) =_o t \) for any term \( t \) (Lemma 26); hence the above term is equivalent to
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda (x'_1, x'_2). ((t')^{^2}_{^5} [x' \mapsto x'_1])_{i=1,2}) \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= (t')^{^2}_{^5} [x' \mapsto x][f' \mapsto f] \text{ in } (t'')^{^2}_{^5} .
\end{align*}
\]

We define N-reduction so that it reduces \((31)\) to the following
\[
\begin{align*}
\text{let } f &= \text{fix}(f', \lambda x'. t') \text{ in} \\
\text{let } x &= 3 \text{ in} \\
\text{let } y &= t'[x' \mapsto x][f' \mapsto f] \text{ in } t'' .
\end{align*}
\]

It is clear that \((-)^{^2}_{^5}\) of \((35)\) becomes \((34)\); thus, \((-)^{^2}_{^5}\) preserves N-reduction to observational equivalence.

As above, N-reduction is just evaluation but it keeps the form of A-normal form. Repeating this N-reduction, \( f x \) in \((31)\) becomes some value \( V \), and \( \text{pr}_1(f(x, x)) \) in \((33)\) becomes \((V)^{^2}_{^5}\).

Repeating N-reduction, we finally get its normal form of the following form
\[
\begin{align*}
\text{let } x &= V_1 \text{ in } \\
\text{let } x &= V_n \text{ in } x_i .
\end{align*}
\]

Applying \((-)^{^2}_{^5}\), we have
\[
\begin{align*}
\text{let } x &= (V_1)^{^2}_{^5} \text{ in } \\
\text{let } x &= (V_n)^{^2}_{^5} \text{ in } x_i ,
\end{align*}
\]

and applying \((-)^{^2}_{^5}\), we have
\[
\begin{align*}
\text{let } x &= (V_1)^{^2}_{^5} \text{ in } \\
\text{let } x &= (V_n)^{^2}_{^5} \text{ in } x_i .
\end{align*}
\]

Since now \( x_i \) has a ground type, so does \( V_i \); hence \((V_1)^{^2}_{^5} = (V_n)^{^2}_{^5} = V_i \). Therefore, the two normal forms are observationally equivalent, and so are \((t)^{^2}_{^5}\) and \((t)^{^2}_{^5}\).

In the rest of this section, we give formal definitions and proofs based on the above idea.

### L.1 N-reduction

From now, we define N-reduction. Though N-reduction reduces terms before applying \((-)^{^2}_{^5}\), our intuitive idea is to transform terms after applying \((-)^{^2}_{^5}\) as above; so we put labels to A-normal forms to track the information of the sets \( B \) in the definition of \((-)^{^2}_{^5}\). (Alternatively, we may be able to equivalently consider reduction for \((t)^{^2}_{^5}\) as “polynomial”, i.e., not as the application of \((-)^{^2}_{^5}\) to \( t \) but as a formal term consisting of \( T, B, \) and \( t \).

We define labeled A-normal forms in Figure 21, where we fix a countable set of labels and we use \( b \) as a meta-variable for labels. If we drop all labels in labeled terms \( d \) and \( s \), we obtain terms \( e \) and \( t \) defined in Figure 19, respectively. We implicitly use this labeling-dropping transformation to apply notions for \( t \) to \( s \).
\[ s ::= (x_1, \ldots, x_n) \mid \text{if } x \text{ then } s_1 \text{ else } s_2 \mid \text{let } x = d^R \text{ in } s^b \\
\]
\[ d ::= n \mid \text{op}(x_1, \ldots, x_n) \mid \text{fix}(f(x, \lambda(x_1, \ldots, x_n), s^b) \mid f(x_1, \ldots, x_n) \mid \text{pr}_x \mid \text{fail} \]

**Figure 21.** Labeled A-normal forms

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

\[
R[\text{let } y = \text{op}(x_1, \ldots, x_n)^{b'} \text{ in } s^{b_2}][\rightsquigarrow]N \\
R[\text{let } y = ((\text{op})(m_1, \ldots, m_n))^{b'} \text{ in } s^{b_2}] \\
\text{where } x_i \rightsquigarrow_R m_i^{b'} \\
R[\text{let } y = (f(x_1, \ldots, x_n, g_1, \ldots, g_m))^{b'} \text{ in } s^{b_2}][\rightsquigarrow]N \\
R[\text{let } y = (\text{pr}_x)^{b'} \text{ in } s^{b_2}][\rightsquigarrow]N \\
R[\text{let } y = \text{fail}^{b'} \text{ in } s^{b_2}][\rightsquigarrow]N \text{ fail}
\]

**Figure 22.** N-reduction rules

We define labeled value \( U \)
\[ U ::= n \mid \text{fix}(f, \lambda(x_1, \ldots, x_n), s^b) \mid (x_1, \ldots, x_n) \]
and N-reduction context \( R \)
\[ R ::= [] \mid \text{let } x = U^{b'} \text{ in } R^{b''} \]

For the definition of N-reduction, we need to prepare one relation: for \( R, x, \) and \( U^{b'} \), we write \( x \rightsquigarrow_R U^{b'} \) if \( x \) refers \( U^{b'} \) in \( R \); precisely, \( x \rightsquigarrow_R U^{b'} \) is defined as below.

\[ x \rightsquigarrow_R U^{b'} \overset{\text{def}}{\iff} \text{false} \]
\[ x \overset{\text{let } x' = (U)x^b_1}{\rightsquigarrow_R} U^{b'} \overset{\text{def}}{=} x' \overset{\text{let } x' = (U)x^b_2}{\rightsquigarrow_R} U^{b'} \]
\[ x \overset{\text{let } x' = (U)x^b_2}{\rightsquigarrow_R} U^{b'} \overset{\text{def}}{=} \text{false} \]

Given \( x \rightsquigarrow_R U^{b'} \), \( U^{b'} \) is uniquely determined from \( R, x, \) and there is the unique triple of \( b' \) and N-reduction contexts \( R_x \) and \( R' \)
\[ R = R_x[\text{let } x = U^{b'} \text{ in } R^{b''} \] and \( x^{b''} \overset{\text{R}}{\rightsquigarrow} U^{b''} \)

We can use \( R_x \) as a context for \( U \).

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

- \( R[(x_1, \ldots, x_n)] \)
- \( R[\text{if } x \text{ then } s_1 \text{ else } s_2] \)
- \( R[\text{let } y = d^{b'} \text{ in } s^{b_2}] \)

where \( d \overset{\text{def}}{=} \text{op}(x_1, \ldots, x_n), f(x_1, \ldots, x_n), \text{pr}_x, \text{fail} \)

We will think the first case as normal forms of N-reduction; so we define N-reduction rules for the other two cases.

We define N-reduction rules in Figure 22. Here, cclet is a kind of commuting-conversion of let: for labels \( b, b' \), a variable \( y \), and labeled A-normal forms \( s, s' \), we define an labeled A-normal form cclet
\[ \text{cclet } y \overset{\text{def}}{=} (x_1, \ldots, x_n)^{b'} \text{ in } s^{b'} \]

- \( \text{let } y = (x_1, \ldots, x_n)^{b'} \text{ in } s^{b'} \)
- \( \text{cclet } y = (x_1, \ldots, x_n)^{b'} \text{ in } s^{b'} \)

Let \( y = (x_1, \ldots, x_n)^{b'} \text{ in } s^{b'} \)

- cclet
- let

**Figure 23.** Commuting-conversion of let

\[ \text{N-reduction reduces labeled A-normal forms to labeled A-normal forms or fail; hence, the normal forms of N-reduction are either } R[(x_1, \ldots, x_n)] \text{ or fail. It is clear that, if the evaluation of } s \text{ terminates, so does N-reduction of } s. \]

1.2 \((-)^{S_0} \ll (-)^{S_4} \) for labeled A-normal forms

We define \((-)^{S_0} \), which is a refined version of \((-)^{S_4} \) for labeled A-normal forms for tracking the information of the sets \( B \).

First, for an labeled A-normal form \( s \), we define the set \( L(s) \) of labels of \( s \) as follows.

\[ L((x_1, \ldots, x_n)) \overset{\text{def}}{=} \emptyset \]
\[ L(\text{if } x \text{ then } s_1 \text{ else } s_2) \overset{\text{def}}{=} \{ b_1 \} \cup \{ b_2 \} \cup L(s_1) \cup L(s_2) \]
\[ L(\text{let } x = d^{b'} \text{ in } s^{b_2}) \overset{\text{def}}{=} \{ b_1 \} \cup \{ b_2 \} \cup L(d) \cup L(s) \]
\[ L(\text{fix}(f, \lambda(x_1, \ldots, x_n), s^{b})) \overset{\text{def}}{=} \{ b \} \cup L(s) \]
\[ L(d) \overset{\text{def}}{=} \emptyset \quad (d \neq \text{fix}()) \]

We call a labeled A-normal form \( s \) label-disjoint if all the occurrences of unions “\( \cup \)” above are disjoint unions when we calculate \( L(s) \) by the above definition. It is obvious that we have a canonical way by which, for a A-normal form \( t \), we obtain label-disjoint labeled A-normal form \( (t)^{b_0} \).

Next, we give a way by which, via labels \( b \), we can track the information of the sets \( B \) in the definition of \((-)^{S_4}_{T,B} \). For a label-disjoint labeled A-normal form \( s \) and \( b \in L(s) \), we define \( B_b^{s_0} \) as in Figure 24: \( B_b^{s_0} \) is merely the set \( B \) used at the position \( b \) in \( s \) when we calculate \((-)^{S_4}_{T,B} \) by the definition in Figure 20. Note that, for labeled A-normal forms \( s_0, s \) and a context \( C \) such that \( s_0 \rightsquigarrow_N C[s] \), we have \( L(s_0) \supseteq L(s) \); hence, when further \( s_0 \) is label-disjoint, for \( y \in L(s), B_b^{s_0} \) is well-defined.

Now, given a labeled-disjoint A-normal form \( s_0 \) and a labeled A-normal form \( s \) such that \( s_0 \rightsquigarrow_N C[s] \) for some \( C \) and a multiplicity-annotation term \( T \) for \( s_0 \), we define a (non-labeled) term \((s)^{S_4}_{T,B} \) by induction on \( s \) in Figure 25, where we also define this for fail, a normal form of N-reduction.

For an A-normal form \( t_0 \), we write \((-)^{S_4}_{T,(t_0)^{b_0}} \).

Also, for a N-reduction context \( s_0 \) such that \( s_0 \rightsquigarrow_N R[s] \) for some \( s \), we define \((R)^{S_4}_{T,s_0} \) as
\[ ([1])^{S_4}_{T,s_0} \overset{\text{def}}{=} [] \]
\[ (\text{let } x = U^{b_1} \text{ in } R^{b_2})^{S_4}_{T,s_0} \overset{\text{def}}{=} (U)^{S_4}_{T,B_b^{s_0}} \text{ in } (R)^{S_4}_{T,s_0}. \]
\[ B^1_{\text{if } x \text{ then } s^1 \text{ else } s^2} \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } b = b_1 \text{ or } b_2 \\ B^1_{s^1} & \text{if } b \in L(s_1) \end{cases} \]

\[ B^1_{\text{let } x = d^1 \text{ in } s^2} \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } b = b_1 \\ \emptyset & \text{if } d = \text{fix}(f, \lambda y. s^3), b = b'' \\ B^1_{s'} & \text{if } d = \text{fix}(f, \lambda y. s^4), b \in L(s') \\ \{x = d\} & \text{if } b = b_2, d = f(\tilde{x}_i), \text{pr}_x x' \\ \emptyset & \text{if } b = b_2, d \neq f(\tilde{x}_i), \text{pr}_x x' \\ B^1_b \cup \{x = d\} & \text{if } b \in L(s), d = f(\tilde{x}_i), \text{pr}_x x' \\ B^1_b & \text{if } b \in L(s_1), d \neq f(\tilde{x}_i), \text{pr}_x x' \end{cases} \]

**Figure 24.** \( B^1_b \): B used at b in the calculation of \( (s)^{2,3}_T \)

\[(x_1, \ldots, x_n, f_1, \ldots, f_m)^{t_i}_{T, s_0} \overset{\text{def}}{=} (x_1, \ldots, x_n, \lambda(y_1, \ldots, y_m). (f_1 y_1, \ldots, f_m y_m)) \]

\[ \left( \begin{array}{l} \text{if } x \text{ then } s^1 \text{ else } s^2 \right)^{t_i}_{T, s_0} \overset{\text{def}}{=} \left( \begin{array}{l} \text{if } x \text{ then } (s^1)^{t_i}_{T, s_0} \text{ else } (s^2)^{t_i}_{T, s_0} \end{array} \right) \]

\[ \left( \begin{array}{l} \text{let } x = d^1 \text{ in } s^2 \right)^{t_i}_{T, s_0} \overset{\text{def}}{=} \text{let } x = (d)^{t_i}_{T, B^0} \text{ in } (s)^{t_i}_{T, s_0} \]

\[ \text{fail}^{t_i}_{T, s_0} \overset{\text{def}}{=} \text{fail} \]

**Figure 25.** \((-)^{t_i}_T; (-)^{t_i}_T\) for labeled A-normal forms

Then, clearly

\[ (R[s])^{t_i}_{T, s_0} = (R)^{t_i}_{T, s_0} (s)^{t_i}_{T, s_0} \]

For Lemma 25, which is the main lemma for Lemma 23, we need the next lemma as well as notions of a s-b context:

\[ S ::= \text{let } y \overset{\text{def}}{=} d^b \text{ in } l \mid \text{let } y = \text{fix}(f, \lambda x.l)^{b_1} \text{ in } s^{b_2} \mid \text{if } x \text{ then } l \text{ else } s^b \mid \text{if } x \text{ then } s^b \text{ else } l \mid \text{let } y \overset{\text{def}}{=} \] and of a s-db context:

\[ D ::= \text{let } y \overset{\text{def}}{=} \] and of a s-db context:

\[ D ::= \text{let } y \overset{\text{def}}{=} \]

For any s, d, b, S, and D, \( S^{[s]} \) and \( D^{[d]} \) are terms of the class of the meta-variable s.

**Lemma 24.** 1. Given an s-db context \( D \) and \( \text{fix}(f, \lambda x. s^b)^{b'} \), let \( s_0 \overset{\text{def}}{=} D[\text{fix}(f, \lambda x. s^b)^{b'}] \) and suppose that \( s_0 \) is label-disjoint. Then \( B^b_{s_0} = B^b_{s_0} \).

2. Given an s-db context \( S \) and \( s^b \), let \( s_0 \overset{\text{def}}{=} S[s^b] \) and suppose that \( s_0 \) is label-disjoint.

- When \( s = \) \text{if } x \text{ then } s^1 \text{ else } s^2 \text{, } B^1_{s_0} = B^1_{s_0} = B^1_{s_0}.
- When \( s = \) \text{let } y = d^1 \text{ in } s^{b_2}, B^2_{s_0} = B^2_{s_0} = B^2_{s_0} \)
  - when \( d \neq f(\tilde{x}) \) or \( \text{pr}_x x, B^2_{s_0} = B^2_{s_0} \)

3. Given an s-db context \( S \) and \( s^b \), let \( s_0 \overset{\text{def}}{=} S[s^b] \) and suppose that \( s_0 \) is label-disjoint. Let \( T \) be a multiplicity annotation for \( s_0 \). Then

\[ (s)^{t_i}_{T, s_0} = (s)^{t_i}_{T, B^0} \]

4. For a label-disjoint A-normal form \( s_0 \) such that

\[ s_0 \not\overset{\text{N}}{\rightarrow} D[\text{fix}(f, \lambda x. s^b)^{b'}] \]

there is some \( D' \) such that

\[ s_0 = D'[\text{fix}(f, \lambda x. s^b)^{b'}] \]

5. For a label-disjoint A-normal form \( s_0 \) such that

\[ s_0 \not\overset{\text{N}}{\rightarrow} R[\text{cellet } y = s^b \text{ in } s^{b'}] \]

and a multiplicity annotation \( T \) for \( s_0 \),

\[ \text{cellet } y = s^b \text{ in } s^{b'} (\text{fix})^{t_i}_{T, s_0} = \text{cellet } y = (s)^{t_i}_{T, s_0} \]

**Proof.** 1. By induction on D.

3. By induction on \( s \) and by 2 of this lemma.

4. By induction on the length of the N-reduction, it is enough to prove that, if

\[ s_0 \not\overset{\text{N}}{\rightarrow} s_1 \not\overset{\text{N}}{\rightarrow} D[\text{fix}(f, \lambda x. s^b)^{b'}] \]

there is \( D' \) such that

\[ s_1 = D'[\text{fix}(f, \lambda x. s^b)^{b'}] \]

We consider only the case that the redex of \( s_1 \) is application; the others are clear. Hence,

\[ s_1 = R[\text{let } y = (f(x_1, \ldots, x_n, g_1, \ldots, g_m))^b_1 \text{ in } s^{b_2}] \]

\[ \text{fix}(f, \lambda x. s^b)^{b'} = R[\text{cellet } y = s^b \text{ in } s^{b'}] \]

\[ f \not\overset{\text{R}}{\rightarrow} \text{fix}(f', \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m). s'^{b''})^{b''} \]

By α-renaming, there is \( s'' \) such that

\[ \text{fix}(f, \lambda x. s^b)^{b'} = R[\text{cellet } y = s'^{b''} \text{ in } s^{b_2}] \]

\[ f \not\overset{\text{R}}{\rightarrow} \text{fix}(f', \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m). s'^{b''})^{b''} \]

If \( \text{fix}(f, \lambda x. s^b)^{b'} \) occurs in \( R \) or \( s_2 \), so does in \( s_1 \). Otherwise, \( \text{fix}(f, \lambda x. s^b)^{b'} \) occurs in \( s'' \). Since

\[ f \not\overset{\text{R}}{\rightarrow} \text{fix}(f', \lambda(x_1, \ldots, x_n, g_1, \ldots, g_m). s'^{b''})^{b''} \]

\( s'' \) occurs in \( R \); hence, \( \text{fix}(f, \lambda x. s^b)^{b'} \) occurs in \( s_1 \).

5. By induction on \( s \).

We here give a remark on the definition of labeled A-normal forms. The labels for \( d \) are needed for the definition of \((-)^{t_i}_T \) and the labels for \( s \) in \( \text{fix}(f, \lambda x. s^b)^{b'} \) with Lemma 24-1,3 are needed for the proof of Lemma 25. On the other hand, the labels for the other occurrences of \( s \) (i.e., \( s \) in if and let expressions) are needed just for the induction on \( s \) in the proof of Lemma 24-3.
L.3 \((-\)Tₙ\) preserves N-reduction to observational equivalence

It is clear that Lemma 23 can be proved immediately by the following lemma.

Lemma 25. 1. For an A-normal form \(t\),
\[
(t)_{T,t}^{234} = \left((t)^{ib}\right)_{T,t}^{\mathcal{N}}.
\]

2. For a closed ground A-normal form \(t\),
\[
(t)^{ib} \rightarrow_{\mathcal{N}}^* R([x_1, \ldots, x_n])
\]

implies
\[
(R([x_1, \ldots, x_n]))_{T,t}^{234} =_o (R([x_1, \ldots, x_n]))_{T,t}^{\mathcal{N}},
\]
and \((t)^{ib} \rightarrow_{\mathcal{N}}^* \) fail implies
\[
(fail)_{T,t}^{234} =_o (fail)_{T,t}^{\mathcal{N}}.
\]

3. For a closed A-normal form \(t\), \((t)^{ib} \rightarrow_{\mathcal{N}}^* s\) implies
\[
(t)_{T,t}^{234} =_o (s)_{T,t}^{234} \quad \text{and} \quad \left((t)^{ib}\right)_{T,t}^{\mathcal{N}} =_o (s)_{T,t}^{\mathcal{N}}.
\]

In the proof of this lemma below, in addition to \(\lambda\)-calculus [11] (the standard call-by-value equational theory), we often use the following reasoning principle, which we call referential transparency:

\[
\text{let } x = t \text{ in } C[x] = \text{ let } x = t \text{ in } C[t], \quad \text{(RT)}
\]

where the occurrence \(x\) in \(C[x]\) must be free (and bound by the let-declaration). Here we used contexts \(C[x]\) and \(C[t]\) rather than \(t\) and \([x \mapsto t]\); this means that any one occurrence of \(x\) and \(t\) are interchangeable (and hence, so are all the occurrences, by repeating it). It is clear that (RT) is sound with respect to the observational equivalence of our language.

The axiom (RT) allows us to regard let-binding such as

\[
\begin{align*}
& (x = t) \in B \text{ of } (-T,b) \\
& x \rightsquigarrow_{T,b} t
\end{align*}
\]

as “an equation already proved”. Below, for a context \(C\), we write \(t \equiv t' \in C\) to mean that \(C[t] =_o C[t']\). For example, when \(x \rightsquigarrow_{T,b} t\), it is true that \(x \equiv t \in R\) by (RT), though \(x\) and \(t\) themselves are not necessarily observationally equivalent. We sometimes omit contexts \(C\) when they are clear.

Below, when we write \(t = \{\ldots\} t', \{\ldots\\} t\) is an explanation of why the equation holds.

By (RT), we can prove:

Lemma 26. 1. For any term \(t\), \(\text{pr}_1(t, \ldots, t) =_o t\).

2. For any term of a tuple type, \(t =_o (\text{pr}_1 t, \text{pr}_2 t)\).

Proof. 1.
\[
\begin{align*}
\text{pr}_1(t, t, \ldots, t) &= (\lambda c) \\{ \{RT\} \\
&\text{let } x = t \text{ in } \text{pr}_1(x, t, \ldots, t) \\}
\end{align*}
\]

2. Similarly,
\[
\begin{align*}
(\text{pr}_1 t, \text{pr}_2 t) &= (\lambda x) \{ \text{let } x = t \text{ in } (\text{pr}_1 x, \text{pr}_2 t) \} \\
&= (\lambda x) \{ \text{let } x = t \text{ in } (\text{pr}_1 x, \text{pr}_2 x) \} \\
&= (\lambda x) \{ \text{let } x = t \} \\
&= t.
\end{align*}
\]

Now, let us return to proving Lemma 25.

Proof of Lemma 25.

1. Straightforward.

2. Since \(R([x_1, \ldots, x_n])\) is ground, \(x_i\) are integer variables. Hence \(x_i \rightsquigarrow_{R} m_i\) for some \(m_i\), so the left sides are observationally equivalent to \((m_1, \ldots, m_n)\). The case of fail is clear.

3. A proof on \((-\)Tₙ\) is obvious if we can prove the case of \((-\)Tₙ\). We show that, for a closed ground A-normal form \(t_0\),
\[
(t_0)^{ib} \rightarrow_{\mathcal{N}}^* s_1 \text{ implies } (s_1)_{T,t_0}^{\mathcal{N}} =_o (s_2)_{T,t_0}^{\mathcal{N}}
\]

by induction on the length of the N-reduction; further, simultaneously we show that, if \(s_1 = R[\text{let } y = d]\) in \(s^2\),
\[
(d)_{T,t_0}^{\mathcal{N}} \equiv (d)^{T,\mathcal{N}}
\]

in the context \((R)_{T,t_0}[\text{let } y = \{\ldots\} (s)_{T,t_0}^{\mathcal{N}}]\). We show only the case of application:

\[
\begin{align*}
& s_1 = R[\text{let } y = \{\ldots\} (s)_{T,t_0}^{\mathcal{N}}] \\
& s_2 = R[\text{cfct } \{\ldots\} (s')_{T,t_0}^{\mathcal{N}}] \\
& f \rightsquigarrow_{R} \{\text{cfct } \{\ldots\} (s')_{T,t_0}^{\mathcal{N}}\} \\
& \text{since the other cases are clear. By } \alpha\text{-renaming, we assume } x_i = x, \\
g' = f.
\end{align*}
\]

We postpone to show
\[
(f(\bar{x}, \bar{g}))_{T,t_0}^{\mathcal{N}} \equiv (f(\bar{x}, \bar{g}))_{T,t_0}^{234}
\]

—i.e., to show that \(\text{assume in } \text{InstVar}(\bar{x})\) are satisfied—until Appendix L.4 and show the remaining part first.

Since \(f \rightsquigarrow_{R} \{\text{cfct } \{\ldots\} (s')_{T,t_0}^{\mathcal{N}}\} \text{, applying } (-T,t_0)^{\mathcal{N}},
\[
\begin{align*}
& f \rightsquigarrow_{(R)} (f(\lambda(\bar{x}, \bar{g})), s')_{T,t_0}^{\mathcal{N}} \\
& = \{\text{cfct } \{\ldots\} (s')_{T,t_0}^{\mathcal{N}}\} (s_1, \ldots, s(T,f))
\end{align*}
\]

where
\[
\begin{align*}
& s_k \overset{\text{def}}{=} (s')_{T,t_0}^{\mathcal{N}} \{\text{cfct } \{\ldots\} (s')_{T,t_0}^{\mathcal{N}}\} (s_1, \ldots, s(T,f)) \\
& p_j \overset{\text{def}}{=} \lambda y. \text{pr}_j((\text{pr}_{n+1} z_k)(\underline{m-j}, y, \underline{m-j})).
\end{align*}
\]

Now,
\[
\begin{align*}
& (s_1)_{T,t_0}^{\mathcal{N}} = (R)_{T,t_0}^{\mathcal{N}}[\text{let } y = \{\ldots\} (s)_{T,t_0}^{\mathcal{N}}] \\
& (s_2)_{T,t_0}^{\mathcal{N}} = (R)_{T,t_0}^{\mathcal{N}}[(\text{cfct } y = \{\ldots\} (s)_{T,t_0}^{\mathcal{N}})] \\
& = \{\text{by Lemma 24}\} \\
& (R)_{T,t_0}^{\mathcal{N}}[\text{cfct } y = \{\ldots\} (s)_{T,t_0}^{\mathcal{N}}]
\end{align*}
\]
Lemma 25.

In the context $(R)^{\mathcal{N}}_{T,T_0}$, let $y = []$ in $(s)^{\mathcal{N}}_{T,T_0}$.

Then, by applying this repeatedly for $j = m+1, \ldots, 1$ (in the reverse order), we get our goal, i.e.,

$$(f(x_1, \ldots, x_n, g_1, \ldots, g_m))^{\mathcal{N}}_{T,B_{t_0}^{m+1}} \equiv t_1$$

in the context $(R)^{\mathcal{N}}_{T,T_0}$, let $y = []$ in $(s)^{\mathcal{N}}_{T,T_0}$.

Thus, our goal is to prove

$$\{(\tilde{\alpha}_j)_{i} \mid \text{let } \tilde{w}((\alpha_j)_{i}) \equiv B_{k}^{i}((\alpha_j)_{i}) \text{ in } t \equiv t\},$$

i.e., to show that $p$ and $p'$ above are satisfied and hence assume ($p$) and assume ($p'$) can be removed. They are defined using $B_{k}^{i}((\alpha_j))$, which is defined in Step 4 in Appendix K.1 as

$$\text{pr}_{j}B_{k}^{i}((\alpha_j)) = \text{pr}_{j}(\text{pr}_{j}^{\mathcal{N}}z(\arg_{1}(\alpha_1), \ldots, \arg_{m}(\alpha_m)))$$

and by (30),

$$\text{pr}_{j}^{\mathcal{N}}z : \prod_{j=1}^{m}((\tau_{j})^{\mathcal{N}})^{\cdot y} \rightarrow \prod_{j=1}^{m}((\tau_{j})^{\mathcal{N}})_{[1 \rightarrow a_{j}]}^{(a_{j})}.$$
Now, since \( f'_j \) have function types, there are \( \left( \text{fix}(f''_j, \lambda x''_j, s''_j) \right)^{v''_j} \) such that
\[
f'_j \rightarrow_{T_{v''_j}} \left( \text{fix}(f''_j, \lambda x''_j, s''_j) \right)^{v''_j}
\]
Then, applying \((-)^{v''_j}T_{v''_j} = \text{fix}(f''_j, \lambda x''_j, s''_j)\frac{\text{fix}(f''_j, \lambda x''_j, s''_j) \cdot B_j}{B_j}[v''_j]
\]
for some \( t''_j \), which does not contain variables \( z''_j \) for \( i' \neq i \) (by the definition in Figure 20). Hence,
\[
\text{pr}_i(f'_j(t, i), e_n) = \{ \text{pr}_i(\left( t''_j, \ldots, t''_j, T(f_j) \right)[z''_j \mapsto t, j], i, e_n[f''_j \mapsto f'_j]) \}
\]
In the same way,
\[
\text{pr}_i(\left( \text{pr}^j_z v \right)(t')) = t''_i[z''_j \mapsto t, j, i] \cdot f'_j \mapsto f'_j.
\]
By assumption, \( t, j, i = t''_j, i \), therefore
\[
\text{pr}_i(\left( \text{pr}^j_z v \right)(t)) = \text{pr}_i(\left( \text{pr}^j_z v \right)(t'))
\]
Now, for a given \((\alpha_j)_{i} \in \prod_{j=1}^{m} \text{App}_j, j \in m, i \in m_j\), we calculate more concrete form of \( \text{pr}_i(\text{pr}_j B^j_{s}(\alpha_j)) \) separately in the following cases of \( \text{pr}_i(\text{arg}^j_{s}(\alpha_j)) \):
\[
\text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = \text{arg}^j_{s}(\text{pr}_i(\text{arg}^j_{s}(\alpha_j))) = \begin{cases} u & \text{if } \alpha_j = u \text{ for some } u \text{ and } (v = \text{pr}_{s}^{-1}z), (w = vu) \in B \\
y_i & \text{if } \alpha_j \neq u \text{ for some } u \text{ and } (v = \text{pr}_{s}^{-1}z), (w = vu) \in B \\
\bot & \text{otherwise}
\end{cases}
\]
where the cases correspond to the cases in the definition of \( \text{arg}^j_{s} \).
In the case that \( \text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = u \) for some \( u \), there are \( v \) and \( w \) such that
\[
\text{pr}_i(\alpha_j) = (u, v, w) \quad \text{and} \quad (v = \text{pr}_{s}^{-1}z), (w = vu) \in B
\]
and then, after \((-)^{t_4^j}a\), the let-binding \( (v = \text{pr}_{s}^{-1}z) \) becomes the following let-binding
\[
v = \lambda a. \text{pr}_j(\left( \text{pr}^j_z z \right)(\bot, \ldots, \bot, a, \bot, \ldots, \bot))
\]
where \( a \) is in \( j \)-th position, and \( (w = vu) \) becomes
\[
w = (vu)^{t_4^j}T_{v''_j, B''_j}
\]
for some \( B' \).
Hence,
\[
\text{pr}_i(\text{pr}_j B^j_{s}(\alpha_j)) = \text{pr}_i(\text{pr}_j(\left( \text{pr}^j_z z \right)(\bot, \ldots, \bot, \text{arg}^j_{s}(\alpha_j)), \bot, \ldots, \bot)) \quad \text{[by Lemma 27-1]}
\]
\[
\text{pr}_i(\left( \text{pr}^j_z z \right)(\bot, \ldots, \bot, \text{arg}^j_{s}(\alpha_j)), \bot, \ldots, \bot) \quad \text{\{\( \beta \)-equality since \( \text{arg}^j_{s}(\alpha_j) \) is a value\}}
\]
\[
\text{pr}_i(\left( \lambda a. \text{pr}_j(\left( \text{pr}^j_z z \right)(\bot, \ldots, \bot, a, \bot, \ldots, \bot)) \right)(\alpha_j)) \quad \text{\{(RT) on (37)\}}
\]
\[
\text{pr}_i(v \arg^j_{s}(\alpha_j)) \quad \text{\{since \( \text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = u \) and by Lemma 27-1 where \( m = 1 \}}
\]
\[
\text{pr}_i(v (u, \ldots, u)) = \{\text{by induction hypothesis, since } w = vu \text{ is in } B \}
\]
\[
\text{pr}_i(v)^{t_4^j}T_{v''_j, B''_j} \quad \text{\{(RT) on (38)\}}
\]
In the case that \( \text{pr}_i(\arg^j_{s}(\alpha_j)) = y_i \) for some \( i \) in \( m_k \),
\[
j = k \quad \text{pr}_i(\alpha_j) = y_i \quad (g = \text{pr}_{s}^{-1}z) \in B,
\]
and after \((-)^{t_4^j}a\), \( (g = \text{pr}_{s}^{-1}z) \) becomes
\[
g = \lambda a. \text{pr}_j(\left( \text{pr}^j_{s} z \right)(\bot, \ldots, \bot, a, \bot, \ldots, \bot)).
\]
Hence,
\[
\text{pr}_i(\text{pr}_j B^j_{s}(\alpha_j)) = \{\text{similarly to the previous} \}
\]
\[
\text{pr}_i(\left( \lambda a. \text{pr}_j(\left( \text{pr}^j_{s} z \right)(\bot, \ldots, \bot, a, \bot, \ldots, \bot)) \right)(\alpha_j)) \quad \text{\{(RT) on (39)\}}
\]
\[
\text{pr}_i(v \arg^j_{s}(\alpha_j)) \quad \text{\{since \( \text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = \bot \}}
\]
\[
\bot.
\]
Now we show that, in the definition of \( \text{InstVar}(g, T, B, t) \), \( \text{assume}(p) \) and \( \text{assume}(p') \) can be removed; then, after that, with \( \lambda \)-theory, \( \text{InstVar}(g, T, B, t) \) is equivalent to \( t \), which concludes the proof of the lemma.
First, note that the equality = and the implication \( => \) used in \( p \) and \( p' \) are not genuine logical operators but boolean primitives, so if two terms \( t \) and \( t' \) both happen divergence (or fail), \( t = t' \) is not true but divergence (or fail), and \( \text{assume} \{ \ldots t = t' \ldots \} \) cannot necessarily be removed (and similarly for \( \Rightarrow \) ). Thus, it is important to know if such \( t \) and \( t' \) are values or not. Now, since \( \text{arg}^j_{s}(\alpha_j) \) and \( \text{arg}^j_{s}(\alpha_j') \) are values, this concern is in fact cleared.
Next, to calculate the \( \text{assume}\)-expressions, we have to substitute \( B^j_{s}(\alpha_j) \) for \( w(\alpha_j) \) in \( p \) and \( p' \), and to do so we need to show that \( B^j_{s}(\alpha_j) \) is (observationally equivalent to) a value. For this end, by Lemma 26-2, it is enough to show that, for any \( i \) and \( j \), \( \text{pr}_i(\text{pr}_j B^j_{s}(\alpha_j)) \) is a value.
For each \( i \), in the case that \( \text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = u \) or \( \bot \), as seen above, \( \text{pr}_i(\text{pr}_j B^j_{s}(\alpha_j)) \) is a value. In the case that
\[
\text{pr}_i(\text{arg}^j_{s}(\alpha_j)) = y_i,
\]
as above,

\[ \text{pr}_i \text{pr}_j B^2_{\gamma_i}((\alpha_{j})_i) \equiv \text{pr}_i(g\tilde{y}). \]

Here, \( g\tilde{y} \) itself is not a value, but can be replaced with a value \( x \) as below by (RT); i.e.,

\[
\begin{align*}
&\text{let } x = g\tilde{y} \text{ in} \\
&\quad \text{let } w^{((\alpha_{j})_i)} = B^2_{\gamma_i}((\alpha_{j})_i) \text{ in} \\
&\quad \quad \ldots \\
&\quad \text{let } w^{((\alpha_{k})_j)} = B^2_{\gamma_j}((\alpha_{k})_j) \text{ in} \\
&\quad \quad \text{assume } (p|\text{pr}_i(g\tilde{y})); \text{assume } (p' ; x) \\
&\text{let } x = g\tilde{y} \text{ in} \\
&\quad \text{let } w^{((\alpha_{j})_i)} = B^2_{\gamma_i}((\alpha_{j})_i) \text{ in} \\
&\quad \quad \ldots \\
&\quad \text{let } w^{((\alpha_{k})_j)} = B^2_{\gamma_j}((\alpha_{k})_j) \text{ in} \\
&\quad \quad \text{assume } (p|\text{pr}_i(x)); \text{assume } (p' ; x)
\end{align*}
\]

Thus, we can substitute \( B^2_{\gamma_i}((\alpha_{j})_i) \) for \( w^{((\alpha_{j})_i)} \).

On \text{assume } (p), it is obvious that the term

\[ \text{arg}^*_j((\alpha_{j})_i) = \text{arg}^*_j((\alpha'_{j})_i) \Rightarrow \text{pr}_i \text{pr}_j B^2_{\gamma_i}((\alpha_{j})_i) \]

is observationally equivalent to \text{true}, from the calculation of \( \text{pr}_i \text{pr}_j B^2_{\gamma_i}((\alpha_{j})_i) \) above. On \text{assume } (p'), similarly,

\[ w = \text{pr}_i \text{pr}_j B^2_{\gamma_i}((\alpha_{j})_i) \]

is observationally equivalent to \text{true}. \qed