Verifying Relational Properties of Functional Programs by First-Order Refinement

Kazuyuki Asada Ryosuke Sato Naoki Kobayashi
University of Tokyo
{asada,ryosuke,koba}@kb.is.s.u-tokyo.ac.jp

Abstract

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

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

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

1. Introduction

There has been much progress in automated verification techniques for higher-order functional programs [11–14, 17, 18, 20].[1] Most of those techniques abstract programs by using *first-order* predicates on base values (such as integers), due to the limitation of underlying theorem provers and predicate discovery procedures. For example, consider the program:

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

Using the existing techniques [11, 13, 17, 18], one can verify that `sum` has the first-order refinement type: $(n : \mathbf{int}) \to \{m : \mathbf{int} \mid m \geq n\}$, which means that `sum n` returns a value no less than `n`. Here, $\{m : \mathbf{int} \mid P(m)\}$ is the (refinement) type of integers m that satisfy $P(m)$.

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

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

Suppose we wish to check that $\texttt{sum2}(n)$ equals $\texttt{sum}(n)$ for every integer n. With general refinement types [8], that would amount to checking that `sumacc` and `sum2` have the following types:[2]

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

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

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

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

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

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

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

We face even more difficulties when dealing with higher-order functions. Consider the program in Figure 1. Here, a list is encoded as a function that maps each index to the corresponding element

[1] In the present paper, by *automated* verification, we mean (almost) fully automated one, where a tool can automatically verify a given program satisfies a given specification (expressed either in the form of assertions or refinement type declarations), without requiring invariant annotations (such as pre/post conditions for each function). It should be contrasted with refinement type checkers [5, 21] where a user must declare refinement types for *all* recursive functions including auxiliary functions. Some of the automated verification techniques above require a hint [20], however.

PEPM '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3297-2/15/01...$15.00.
http://dx.doi.org/10.1145/2678015.2682546

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

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

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

Figure 1. Append function for functional encoding of lists

(or `None` if the index is out of bounds) [14], and the append function is defined. Suppose that we wish to verify that `append xs nil = xs`. With general refinement types, the property would be expressed by:

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

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

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

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

To take advantage of the recent advance of verification techniques based on first-order refinement types, however, we employ automated program transformation, so that the resulting program can be verified by using only first-order refinement types. The key idea of the transformation is to apply a kind of tupling transformation [6] to capture the relationship between two (or more) function calls at the level of first-order refinement. For example, for the `sum` program above, one can apply the standard tupling transformation (to combine two functions `sum` and `sumacc` into one) and obtain:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We formalize the idea sketched above and prove the soundness of the transformation. We also report on a prototype implementation of the approach as an extension to the software model checker MoCHi [11, 14] for a subset of OCaml. The implementation takes a program and its specification (in the form of refinement types) as input, and verifies them automatically (without invariant annotations for auxiliary functions) by applying the above transformations and calling MoCHi as a backend.

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

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

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

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

Figure 2. Operational semantics of the source language

with recursion. The syntax of *terms* is given by:

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

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

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

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

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

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

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

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

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

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

- $\models^n_{\mathrm{v}} V : \{\nu : \sigma \mid P\} \overset{\text{def}}{\iff} \models^n_{\mathrm{v}} V : \sigma$ and $\models^n_{\mathrm{p}} P[\nu \mapsto V]$
- $\models^n_{\mathrm{v}} V : \mathbf{int} \overset{\text{def}}{\iff} V = m$ for some integer m
- $\models^n_{\mathrm{v}} V : (x_1 : \tau_1) \to \tau_2 \overset{\text{def}}{\iff}$ for all $n' \leq n$ and V_1, $\models^{n'}_{\mathrm{v}} V_1 : \tau_1$ implies $\models^{n'}_{\mathrm{c}} VV_1 : \tau_2[x_1 \mapsto V_1]$
- $\models^n_{\mathrm{v}} (V_1, \ldots, V_n) : \prod_{i=1}^n (x_i : \rho_i) \overset{\text{def}}{\iff}$ $\models^n_{\mathrm{v}} V_i : \rho_i[x_1 \mapsto V_1, \ldots, x_{i-1} \mapsto V_{i-1}]$ for all $i \leq n$
- $\models_{\mathrm{v}} V : \tau \overset{\text{def}}{\iff} \models^n_{\mathrm{v}} V : \tau$ for all n

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

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

Figure 3. Semantics of types

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

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

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

$$
\begin{array}{ll}
\mathtt{ST}(\{\nu : \sigma \mid P\}) = \mathtt{ST}(\sigma) & \mathtt{ST}(\mathbf{int}) = \mathbf{int} \\
\mathtt{ST}((x : \tau_1) \to \tau_2) = \mathtt{ST}(\tau_1) \to \mathtt{ST}(\tau_2) & \\
\mathtt{ST}((x : \tau_1) \times \tau_2) = \mathtt{ST}(\tau_1) \times \mathtt{ST}(\tau_2) &
\end{array}
$$

Also, we define the *order of* τ by:

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

The syntax of types is subject to the usual scope rule; in $(x{:}\,\rho_1) \times \rho_2$ and $(x{:}\,\tau_1) \to \tau_2$, the scope of x is ρ_2 and τ_2 respectively. Furthermore, we require that every refinement predicate is well-typed and have type $\mathbf{int}$. (See the full version of this paper [3] 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, 7]. Roughly speaking, $\models^n_{\mathrm{c}} t : \tau$ means that t behaves like a term of type τ within n steps computation. For example, $\models^n_{\mathrm{c}} t : \mathbf{int}$ means that if t evaluates to an answer within

n steps, then the answer is not *fail* but an integer, (otherwise if t needs more than n steps to evaluate, t may diverge or fail). Also, the condition $\models^n_{\mathrm{c}}\ t : \mathbf{int} \to \mathbf{int}$ means that if t evaluates to an answer A within n steps, say at k-step ($k \leq n$), then A must be a function, and $\models^{n-k}_{\mathrm{c}}\ A\, m : \mathbf{int}$ must hold for every integer m, i.e., if $A\, m$ converges to an answer within $n-k$ steps, then the answer is not *fail* but an integer. The connectives $\forall$ and $\wedge$ have genuine logical meaning, and especially they are commutative, so we often use the prenex normal form.

Notice that, by the definition, $\models^n_{\mathrm{p}}\ t$ holds for every n if t diverges. We write & and || for (expression-level) Boolean conjunction and disjunction. Notice that the semantics of $t_1 \wedge t_2$ and $t_1\ \&\ t_2$ are different. For example, let Ω be a divergent term. Then $\models\ 1 : \{x : \mathbf{int} \mid \Omega \wedge x = 0\}$ does NOT hold, but $\models\ 1 : \{x : \mathbf{int} \mid \Omega\ \&\ x = 0\}$ DOES hold, since $\Omega\ \&\ x = 0$ diverges.

The goal of our verification is to check whether $\models t : \tau$ holds, given a program t and a type τ. Since the verification problem is undecidable, we aim to develop a sound but incomplete method below. As explained in Section 1, our approach is to use program transformation to reduce the (semantic) type checking problem $\models t : \tau$ to the first-order refinement type checking problem $\models t' : \tau'$ where τ' does not contain any function variables in refinement predicates, and to check $\models t' : \tau'$ using an automated verification tool such as MoCHi [11, 14, 19], which combines higher-order model checking [10] and predicate abstraction.

2.3 Restriction on Refinement Types

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

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

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

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 $(x : \tau_1) \to \{\nu : \sigma \mid P\}$ where τ_1 is a depth-1 function type, x may occur in P but not in σ (there is no such restriction if τ_1 is a depth-0 type), and (iii) in a type of the form $(f_1 : \tau_1) \times \{f_2 : \sigma_2 \mid P_2\} \times \cdots \times \{f_n : \sigma_n \mid P_n\}$, f_1 may occur in $P_2, \ldots, P_n$ but not in $\sigma_2, \ldots, \sigma_n$.

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathtt{ElimHO}_n(\Gamma) \stackrel{\mathrm{def}}{=} \{(x : \tau) \in \Gamma \mid depth(\tau) <= n\}$$
$$\Gamma, \Delta ::= \emptyset \mid \Gamma, x : \tau$$

Figure 4. Well-formedness of types

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

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

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

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

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

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

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

3. Encoding Functional Refinement

In this section, we present a transformation $(-)^\sharp$ for reducing a general refinement type checking problem to the first-order refine-

ment type checking problem. In the rest of the paper, we use the assumptions explained in Section 2.1.

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

3.1 Idea of the Transformation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where f occurs exactly twice, we transform this to

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

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

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

Given a type of the form:

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

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

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

where y is a fresh variable.

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

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

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

can be transformed to

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

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

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

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

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

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

which is equivalent to

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

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

$$\overset{?}{\models} \lambda(n_1,n_2).\,(\mathtt{sum}\,n_1, \mathtt{sum2}\,n_2)\ :\ \left((n_1,n_2) : \mathbf{int}^2\right) \to \left\{(r_1,r_2) : \mathbf{int}^2 \;\middle|\; n_1 = n_2 \mathrel{=>} r_2 = r_1\right\}.$$

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

3.2 Transformations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

can be transformed to

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

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

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

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

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

is transformed to a type of the form

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

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

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

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

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

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

For example, consider the type

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

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

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

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

$$\mathtt{sArg}(f\, x = g\, x) = (f\, z^{\langle f x \rangle} = g\, z^{\langle g x \rangle}),$$

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

$$\forall x, z^{\langle f x \rangle}, z^{\langle g x \rangle}.\ z^{\langle f x \rangle} = x\ \ \&\ \ z^{\langle g x \rangle} = x\ \Rightarrow\ f\, z^{\langle f x \rangle} = g\, z^{\langle g x \rangle}.$$

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

$$\forall z^{\langle f x \rangle}, z^{\langle g x \rangle}.\ z^{\langle f x \rangle} = z^{\langle f x \rangle}\ \&\ z^{\langle g x \rangle} = z^{\langle f x \rangle} \Rightarrow f\, z^{\langle f x \rangle} = g\, z^{\langle g x \rangle},$$

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

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

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

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

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

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

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

$$(t_1\, t_2)_T^{\sharp_3} \stackrel{\text{def}}{=} \left(\mathbf{pr}_1 (t_1)_T^{\sharp_3}\right) (t_2)_T^{\sharp_3}$$

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

which may be simplified further to

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

$\sharp_3$: Replication of Functions

As explained in Section 3.1, $(-)^{\sharp_3}$ replicates a function f_j according to the number m_j of occurrences of f_j in the predicate P of a refinement type $\tau = \left\{\nu : \prod_{i=1}^{n} \mathbf{int} \times \prod_{j=1}^{\ell} \left(f_j\colon \tau_j \to \tau'_j\right) \;\middle|\; P\right\}$; we call m_j *the multiplicity of* f_j and write $mul(\tau, j)$ or $mul(P, j)$. We call the sequence $(m_j)_j = m_1 \cdots m_\ell$ *the multiplicity of* τ.

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

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

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

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

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

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

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

$$(P)^{\sharp_4} \stackrel{\text{def}}{=} ((\wedge_k t_k)[a_l \mapsto r_{\hat{j}(a_l)}]_{l \in m'})[\hat{z}^{(a_l)} \mapsto y_{\hat{j}(a_l)}]_{l \in m'}.$$
$$\left((t_1, \ldots, t_n, t'_1, \ldots, t'_m)\right)^{\sharp_4} \stackrel{\text{def}}{=}$$
$$\mathbf{let}\ x_1 = (t_1)^{\sharp_4}\ \mathbf{in} \cdots \mathbf{let}\ x_n = (t_n)^{\sharp_4}\ \mathbf{in}$$
$$\mathbf{let}\ f_1 = \left(t'_1\right)^{\sharp_4}\ \mathbf{in} \cdots \mathbf{let}\ f_m = \left(t'_m\right)^{\sharp_4}\ \mathbf{in}$$
$$(x_1, \ldots, x_n, \lambda y.\, (f_1\, (\mathbf{pr}_1 y), \ldots, f_m\, (\mathbf{pr}_m y)))$$

where t_i are integers and t'_i are functions.

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

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

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

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

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

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

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

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

For the transformation of refinement predicates, we use the functions $\hat{j}(-)$ and $\hat{z}^{(-)}$ defined as follows. For an input type $\{((x_i)_{i \le n}, (f_j)_{j \le m}) : \ldots \mid P\}$ of $(-)^{\sharp_4}$, we can assume that by $(-)^{\sharp_1}$, function symbols occurring in a refinement predicate are in $\{f_j \mid j \le m\}$; and that by $(-)^{\sharp_2}$ and $(-)^{\sharp_3}$, all application occurrences in P have distinct function variables, and have distinct argument variables that quantified universally. Thus, there is an

injection $\hat{j}(-)$ from the set X of occurrences of applications in P to $\{j \mid j \leq m\}$ such that for any application occurrence ft, $f = f_{\hat{j}(ft)}$; and also there is a bijection $\hat{z}^{(-)}$ from the same set X to the set of the variables that are universally in P.

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

$$\{\ (f,g)\colon (\mathbf{int} \to \mathbf{int})^2 \mid \\ \forall z^{\langle fx\rangle}, z^{\langle gx\rangle}.\ z^{\langle fx\rangle} = z^{\langle gx\rangle}\ \texttt{=>}\ f\,z^{\langle fx\rangle} = g\,z^{\langle gx\rangle}\ \}.$$

The transformed type is of the form

$$\left((y_1,y_2)\colon (\mathbf{int})^2_\bot\right) \to \left\{(r_1,r_2)\colon (\mathbf{int})^2_\bot \mid (\ldots)^{\sharp_4}\right\}.$$

The occurrences of applications are:

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

and

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Soundness of the Transformation

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

In the theorem, for a given typing judgment $\overset{?}{\models} t : \tau$, we assume a condition called *consistency* on multiplicity annotation T and multiplicity type ϕ. We give its formal definition in the full version of this paper [3]; intuitively, T and ϕ are consistent (with respect to t and τ) if it makes consistent assumptions on each subterm, so that the result of the transformation is simply-typed.

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

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

Proof. See the full version [3]. □

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

4. Transformations for Enabling First-Order Refinement Type Checking

The transformation $(-)^\sharp$ in the previous section allowed us to reduce the refinement type checking $\models t : \tau$ to the first-order refinement type checking $\models (t)^\sharp : (\tau)^\sharp$, but it does not necessarily enable us to prove the latter by using the existing automated verification tools [11, 13, 14, 17, 18, 20]. This is due to the incompleteness of the tools for proving $\models (t)^\sharp : (\tau)^\sharp$. They are either based on (variations of) the first-order refinement type system [21] (see Appendix B for such a refinement type system), or higher-order model checking [10, 11], whose verification power is also equivalent to a first-order refinement type system (with intersection types). In these systems, the proof of $\models t : \tau$ (where τ is a first-order refinement type) must be compositional: if $t = t_1 t_2$, then τ' such that $\models t_1 : \tau' \to \tau$ and $\models t_2 : \tau'$ is (somehow automatically) found, from which $\models t_1 t_2 : \tau$ is derived. The compositionality itself is fine, but the problem is that τ' must also be a first-order refinement type, and furthermore, most of the actual tools can only deal with linear arithmetic in refinement predicates. To see why this is a problem, recall the example of proving `sum` and `sum2` in Section 1. It is expressed as the following refinement type checking problem:

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

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

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

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

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

To deal with the problem above, we further refine the transformation $(-)^\sharp$ by (i) tupling of recursive functions [6] and (ii) insertion of assumptions.

4.1 Tupling of Recursion

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

computing those calls simultaneously. For the example above, we introduce a new recursive function `sum_sum2` defined by:

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

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

$$\textbf{let } f = \textbf{fix}(f, \lambda x.\, \textbf{if } t_{11} \textbf{ then } t_{12} \textbf{ else } E_1[f\, t_1]) \textbf{ in}$$
$$\textbf{let } g = \textbf{fix}(g, \lambda y.\, \textbf{if } t_{21} \textbf{ then } t_{22} \textbf{ else } E_2[g\, t_2]) \textbf{ in } \ldots (f, g) \ldots$$

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

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

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

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

4.2 Insertion of Assume Expressions

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

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

has the type

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

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

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

and the type τ is transformed to

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

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

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

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

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

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

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

and apply the returned value to, say, 0.

Table 1. Results of preliminary experiments

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

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

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

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

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

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

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

We write $(-)^{\sharp'}$ for the above assume-inserted version of $(-)^\sharp$. The formal definition of $(-)^{\sharp'}$ is described in the full version of this paper [3]. In the target language, **fail** is treated as an exception, and we define $\textbf{assume}\,(t)$ as a shorthand for:

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

Note that our backend model checker MoCHi [11, 14] supports exceptions. After replacing $(-)^\sharp$ with $(-)^{\sharp'}$, Theorem 1 is still valid:

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

See the full version [3] for the details of the proof.

5. Implementation and Experiments

We have implemented a prototype, automated verifier for higher-order functional programs as an extension to a software model checker MoCHi [11, 14] for a subset of OCaml.

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

The programs used in the experiments are as follows. The programs "sum-acc", "sum-simpl", and "append-xs-nil" are those

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

given in Section 1. The program "mult-acc" is similar to "sum-acc" but calculates the multiplication. The program "sum-mono" asserts that the function `sum` is monotonic, i.e., $\forall m, n.\ m \leq n \Rightarrow \mathtt{sum}(m) \leq \mathtt{sum}(n)$. The program "a-max-gen" finds the max of a functional array; the checked specification is that "a-max-gen" returns an upper bound. Here is the main part of the code of "a-max-gen".

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

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

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

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

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

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

6. Related Work

Knowles and Flanagan [8, 9] gave a general refinement type system where refinement predicates can refer to functions. Their verification method is however a combination of static and dynamic checking, which delegates type constraints that could not be statically discharged to dynamic checking. The dynamic checking will miss potential bugs, depending on given arguments. On the other hand, our method is static and fully automatic.

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

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

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

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

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

There are less automated approaches to refinement type checking, where programmers supply invariant annotations (in the form of refinement types) for all recursive functions [4, 5], and then verification conditions are generated and discharged by SMT solvers. Xu's method [22, 23] for contract checking also requires that contracts must be declared for all recursive functions. In contrast, in our method, a refinement type is used only for specifying the property to be verified, and no declaration is required for auxiliary functions.

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

7. Conclusion and Future Work

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

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

Acknowledgment

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

References

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

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

[3] K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. An extended version, available from `http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/pepm2015.pdf`, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A. Verification of "append-xs-nil"

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

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

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

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

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

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

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

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

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

```
        let r1,r2,_ = xs''ys'rs(i-1,j,None) in
        r1, r2, x
      else
        xs''ys'rs(i-1,j,k-1)
let main n i =
  let xs = make_list n in
  let xs'_nil_rs = append xs [] in
  let xs'rs (i,j) =
    let r1,r2,r3 = xs'_nil_rs (i, None, j) in
    r1, r3
  in
  let r1,r2 = xs'rs (i,i) in
  assert (r2 = r1)
```

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

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

$$(\mathbf{int} \to \mathbf{int}) \to$$
$$((j : \mathbf{int}) \to \{y : \mathbf{int} \mid j = 0 \Rightarrow y = \mathtt{None}\}) \to$$
$$\left((i,j,k) : \mathbf{int}^3\right) \to \left\{(r_1,r_2,r_3) : \mathbf{int}^3 \;\middle|\; i = j \Rightarrow r_1 = r_2\right\}$$

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

B. A Refinement Type System

This section gives a sound type system for proving $\models t : \tau$. Here we do not assume the restrictions in Section 2.3. We obtain also *first-order refinement type system* by restricting the type system so that function variables are disallowed to occur in predicates in all the refinement types. Various *automatic* verification methods [11, 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 $\Gamma \vdash_{\mathrm{t}}^{\mathcal{L}} t : \tau$, where Γ, called a type environment, is a sequence of type bindings of the form $x : \tau$, and $\mathcal{L}$ is (the name of) the underlying logic for deciding the validity of predicates, which we keep abstract through the paper. Below, we use general well-formedness $\vdash_{\mathtt{GWF}}$ (defined in the full version of this paper [3]), which represents usual scope rules of dependent types.

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

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

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

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

The type system is sound with respect to the semantics of types. A proof is given in the full version of this paper [3].

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9. Typing rules

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

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

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

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

Figure 10. Subtyping rules