Regular Paper

Refinement Type Checking via Assertion Checking

Ryosuke Sato[1,a)] Kazuyuki Asada[1,b)] Naoki Kobayashi[1,c)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: A refinement type can be used to express a detailed specification of a higher-order functional program. Given a refinement type as a specification of a program, we can verify that the program satisfies the specification by checking that the program has the refinement type. Refinement type checking/inference has been extensively studied and a number of refinement type checkers have been implemented. Most of the existing refinement type checkers, however, need type annotations, which is a heavy burden on users. To overcome this problem, we reduce a refinement type checking problem to an assertion checking problem, which asks whether the assertions in a program never fail; and then we use an existing assertion checker to solve it. This reduction enables us to construct a fully automated refinement type checker by using a state-of-the-art fully automated assertion checker. We also prove the soundness and the completeness of the reduction, and report on implementation and preliminary experiments.

1. Introduction

A refinement type [3], [11] can be used to express a detailed specification of a higher-order functional program. Given a refinement type as a specification of a program, we can verify that the program satisfies the specification by checking that the program has the refinement type. Refinement type checking/inference has been extensively studied [2], [6], [8], [9], [11], [12] and a number of refinement type checkers have been implemented. Most of the existing refinement type checkers [2], [6], [11], [12], however, force users to provide invariant annotations, which is a heavy burden. For example, consider the following program:

```
let rec fsum f n =
  if n <= 0 then 0
  else f n + fsum f (n-1)
let double n = n + n
let main n = fsum double n
```

Using a refinement type checker, one can verify that the function `main` has type $(x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\}$,

$$\models \texttt{main} : (x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\},$$

i.e., that for any integer x, if `main` x evaluates to r, then $r \geq x$ holds. (Note that refinement types in the current paper specify partial correctness, not total correctness.) To verify the program above, one has to provide the following type annotations.

$$\begin{aligned}\texttt{fsum} : ((x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\}) \to \\ (y : \mathbf{int}) \to \{s : \mathbf{int} \mid s \geq y\} \\ \texttt{double} : (x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\}\end{aligned}$$

The meaning of the second annotation is the same as that for `main` above, and the first annotation means that, for any value f that has type $(x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\}$, if `fsum` f evaluates to g, then g is a value of type $(y : \mathbf{int}) \to \{s : \mathbf{int} \mid s \geq y\}$. Providing such type annotations is a heavy burden on users.

To overcome this problem, we reduce a refinement type checking problem to an assertion checking problem, which asks whether the assertions in a program never fail; then we can use an existing automated assertion checker to solve it. For example, the refinement type checking problem

$$\overset{?}{\models} \texttt{main} : (x : \mathbf{int}) \to \{r : \mathbf{int} \mid r \geq x\}$$

can be reduced to the assertion checking problem that the assertion in the following program never fails.

```
let rec fsum f n = ... in
...
let n = rand_int in
let r = main n in assert(r ≥ n)
```

Here, **rand_int** generates a random integer. While the original problem asks whether `main` n returns a value no less than n for any integer n, the reduced problem asks it by using a random integer and an assertion expression.

Although the reduction for the above program is straightforward, it is not obvious for the case of higher-order functions. For example, consider the following problem:

---
[1] Graduate School of Information Science and Technology, The University of Tokyo
[a)] ryosuke@kb.is.s.u-tokyo.ac.jp
[b)] asada@kb.is.s.u-tokyo.ac.jp
[c)] koba@is.s.u-tokyo.ac.jp

$$\overset{?}{\models} \mathtt{fsum} : (\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}) \to (y : \mathbf{int}) \to \{s : \mathbf{int} \mid s \geq y\}. \qquad (1)$$

Following the random number generator approach above, one may be tempted to prepare a term $\mathtt{gen}()$ that non-deterministically generates every function of type $\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}$. Unfortunately, however, there is no such term $\mathtt{gen}()$, because the set of values of type $\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}$ is not recursively enumerable.

Instead of defining such a generator, we prepare a "universal" term t' that simulates all the terms of type $\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}$, in the sense that for any t'' of type $\{x : \mathbf{int} \mid x > 0\} \to \{s : \mathbf{int} \mid s \geq x\}$ and any $n > 0$, $t''\, n \longrightarrow^* m$ implies $t'\, n \longrightarrow^* m$, and $t''\, n \longrightarrow^* \mathbf{fail}$ implies $t'\, n \longrightarrow^* \mathbf{fail}$. Using t', we can reduce the problem (1) to the following problem:

$$\overset{?}{\models} \mathtt{fsum}\, t' : (y : \mathbf{int}) \to \{s : \mathbf{int} \mid s \geq y\}. \qquad (2)$$

The term t' above can be expressed as follows, by using non-determinism.

$$\begin{array}{l} \lambda x.\, \mathbf{if}\ x > 0\ \mathbf{then} \\ \qquad \mathbf{let}\ r = \mathbf{rand\_int}\ \mathbf{in}\ \mathbf{assume}\,(r \geq x);\, r \\ \quad \mathbf{else\ if}\ *\ \mathbf{then\ rand\_int\ else\ fail} \end{array}$$

Here, $*$ is a non-deterministic Boolean, and $\mathbf{assume}\,(b)$ returns the unit value if $b = \mathbf{true}$ and diverges otherwise. Given an integer x, the term first checks whether $x > 0$. If $x > 0$ holds, then it is expected to return a value no less than x; thus, it generates a random integer r, and returns it only if $r \geq x$. If $x > 0$ does not hold, then nothing is specified by the type; thus, it returns an arbitrary integer or fails. In general, a term $\alpha(\tau)$ that simulates all the values of type τ can be constructed by induction on the structure of τ.

Using the term t' above, we can reduce the problem (2) to the assertion checking problem for the following program.

```
let rec fsum f n = ...
let g x =
  if x > 0 then
    let r = rand_int in assume(r ≥ x); r
  else if * then rand_int else fail
let n = rand_int in assert(fsum g n ≥ n)
```

The reduction sketched above enables us to construct a fully automated refinement type checker by using a state-of-the-art fully automated assertion checker. In fact, the above assertion checking problems can be solved by MoCHi, a software model checker for higher-order functional programs [5], [7], [10] without any annotations.

We formalize the idea sketched above and prove the correctness (i.e., the soundness and the completeness) of the reduction for call-by-value PCF extended with a random number generator. We also report on an implementation of our approach as an extension of MoCHi. Note that the availability of non-determinism (provided by the random number generator) is a crucial assumption for our method. Although our method is applicable to a deterministic source language as long as the target language admits non-determinism, the completeness of the reduction would be lost. For example, in a deterministic language, the type judgment:

$$\models \lambda f.(f\,0 = f\,0) : (\mathbf{int} \to \mathbf{int}) \to \{r : \mathbf{bool} \mid r = \mathbf{true}\}$$

should be semantically valid. Our method reduces it to the assertion checking problem for:

$$\mathbf{assert}((\lambda f.(f\,0 = f\,0))\ \alpha(\mathbf{int} \to \mathbf{int})),$$

where $\alpha(\mathbf{int} \to \mathbf{int})$ is a non-deterministic function $\lambda x.\mathbf{rand\_int}$. The program may fail, since $\lambda x.\mathbf{rand\_int}$ may return an arbitrary number upon each call; thus we fail to show that the type judgment above holds.

A possible remedy to the problem above for dealing with determinism would be to embed the assumption on determinism explicitly in the refinement type specification. For the example above, the resulting type would be:

$$\{f : \mathbf{int} \to \mathbf{int} \mid \forall x. f(x) = f(x)\} \to \{r : \mathbf{bool} \mid r = \mathbf{true}\}\,.$$

It now contains dependency on functions, but this dependency can sometimes be removed by using the technique of our previous work [1]. Please note that existing first-order refinement type checkers [6], [8], [9], [11] do not take into account the determinism either, so that they fail to prove the type judgment above.

The rest of the article is organized as follows. Section 2 introduces the source language and the verification problem. Section 3 presents the reduction from refinement type checking problems to assertion checking problems, and Section 4 proves the correctness of the reduction. Section 5 reports on experiments and Section 6 discusses related work. We conclude the paper in Section 7.

2. Language

This section formalizes the source language and the verification problem. This language is the target of our verification method and is the source and target language of the transformation for the reduction explained in the introduction.

The language is a simply-typed, call-by-value, higher-order functional language with recursion. The syntax of *terms* is given by:

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

We use meta-variables $x, y, z, r, s, f, g, h, \ldots$ for variables. We have only integers as base values, which are denoted by the meta-variable n. The meta-variable $\mathtt{op}$ ranges over primitive operations on integers and a term $\mathtt{op}(t_1, \ldots, t_n)$ is the application of $\mathtt{op}$ to $t_1, \ldots, t_n$. We express Booleans by integers, and write **true** for 1, and **false** for 0. The

$$
\begin{aligned}
v \text{ (value)} &::= n \mid \mathbf{fix}(f, \lambda x.\, t)\\
a \text{ (answer)} &::= v \mid \mathbf{fail}\\
E \text{ (eval. ctx.)} &::= [\,] \mid \mathtt{op}(v_1, \dots, v_n, E, t_1, \dots, t_m)\\
&\qquad \mid E\,t \mid v\,E \mid \mathbf{if}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2
\end{aligned}
$$

$$
\begin{aligned}
&E[\mathtt{op}(n_1, \dots, n_k)] \longrightarrow E[[\![\mathtt{op}]\!](n_1, \dots, n_k)]\\
&E[\mathbf{rand\_int}] \longrightarrow E[n]\\
&E[\mathbf{fail}] \longrightarrow \mathbf{fail}\\
&E[\mathbf{if\ true\ then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow E[t_1]\\
&E[\mathbf{if}\ v\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow E[t_2] \qquad (v \neq \mathbf{true})\\
&E[\mathbf{fix}(f, \lambda x.\, t)\, v] \longrightarrow E[t[\mathbf{fix}(f, \lambda x.\, t)/f][v/x]]
\end{aligned}
$$

Fig. 1 Operational semantics of the source language

term **rand_int** is a non-deterministic integer. We write $*$ for a non-deterministic Boolean, which can be expressed by $\mathbf{rand\_int} = 0$, and $t_1 \,\Box\, t_2$ for $\mathbf{if}\ *\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$. A 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 $\mathbf{fix}(f, \lambda x.\, t)$. A term $t_1\, t_2$ is the application of 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 special term **fail** aborts the execution. It is typically used to express assertions; $\mathbf{assert}(t)$—which asserts that t should evaluate to **true**—is expressed by $\mathbf{if}\ t\ \mathbf{then\ true\ else\ fail}$.

Bound and free variables are defined in a standard manner, and we identify α-equivalent terms. We call a closed term a *program*.

A small-step semantics is given in Figure 1. In the figure, $[\![\mathtt{op}]\!]$ is a given integer function for each $\mathtt{op}$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$; and write $t \preceq t'$ if $t \longrightarrow^* a$ implies $t' \longrightarrow^* a$ for any a.

We express a specification of a program by using a refinement type. The syntax of *refinement types* is given by the following rules.

$$
\begin{aligned}
\tau \text{ (types)} &::= \{x : \mathbf{int} \mid P\} \mid (x : \tau_1) \to \tau_2\\
P \text{ (predicates)} &::= n \mid x \mid \mathtt{op}(P_1, \dots, P_n)
\end{aligned}
$$

A type $(x : \tau_1) \to \tau_2$ is a dependent product type, where x may occur in τ_2. Intuitively, a *refinement type* $\{x : \mathbf{int} \mid P\}$ represents the set of integers x that satisfy the *refinement predicate* P. For example, $\{x : \mathbf{int} \mid x > 0\}$ describes positive integers. The type $(x : \mathbf{int}) \to \{r : \mathbf{int} \mid r > x\}$ describes functions that take an integer x and return an integer r greater than x. The syntax of types is subject to the usual scope rule; in $(x : \tau_1) \to \tau_2$, the scope of x is τ_2. Furthermore, we require that every refinement predicate is well-typed and has type **int** (recall that Booelans are expressed by integers), and that all the variables occurring in a predicate are integer variables. We often write just **int** for $\{x : \mathbf{int} \mid \mathbf{true}\}$, and $\tau_1 \to \tau_2$ for $(x : \tau_1) \to \tau_2$ if x does not occur in τ_2.

A type τ is *simple* if all the predicates in τ are **true**. For a type τ, we define the *simple type* $\mathtt{ST}(\tau)$ *of* τ as follows:

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

- $\models_{\mathrm{p}} P \overset{\mathrm{def}}{\Longleftrightarrow} P \preceq \mathbf{true}$

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

- $\models_{\mathrm{v}} v : \{x : \mathbf{int} \mid P\} \overset{\mathrm{def}}{\Longleftrightarrow}$ $v = n$ for some integer n and $\models_{\mathrm{p}} P[v/x]$
- $\models_{\mathrm{v}} v : (x_1 : \tau_1) \to \tau_2 \overset{\mathrm{def}}{\Longleftrightarrow}$ for all v_1, $\models_{\mathrm{v}} v_1 : \tau_1$ implies $\models v\, v_1 : \tau_2[v_1/x_1]$

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

- $\models t : \tau \overset{\mathrm{def}}{\Longleftrightarrow} \models_{\mathrm{v}} a : \tau$ for all a s.t. $t \longrightarrow^* a$

Fig. 2 Semantics of types

$$
\begin{aligned}
\mathtt{ST}(\{x : \mathbf{int} \mid P\}) &= \mathbf{int}\\
\mathtt{ST}((x : \tau_1) \to \tau_2) &= \mathtt{ST}(\tau_1) \to \mathtt{ST}(\tau_2).
\end{aligned}
$$

We use a meta-variable σ for simple types. For a simple type σ, we define the *size of* σ as follows:

$$
\begin{aligned}
&\mathit{size}(\mathbf{int}) = 1\\
&\mathit{size}(\tau_1 \to \tau_2) = 1 + \mathit{size}(\tau_1) + \mathit{size}(\tau_2)
\end{aligned}
$$

The semantics of types is defined in Figure 2 using logical relations. Here, note that the evaluation is nondeterministic, and that the statement $\models_{\mathrm{v}} a : \tau$ implies that a is a value (i.e., a must not be **fail**). Since $\models v : \tau$ if and only if $\models_{\mathrm{v}} v : \tau$, we often write $\models v : \tau$ for $\models_{\mathrm{v}} v : \tau$.

For a program t and a type τ, the *type checking problem* $\overset{?}{\models} t : \tau$ asks whether $\models t : \tau$ holds. An *assertion checking problem* is a special case, where $\tau = \mathbf{int}$; note that $\models t : \mathbf{int}$ holds if and only if t does not fail.

Our goal is to develop an automated verification method for type checking problems. As explained in Section 1, our approach is to reduce the (semantic) type checking problem $\overset{?}{\models} t : \tau$ to the assertion checking problem $\overset{?}{\models} t' : \mathbf{int}$ by synthesizing t' from t and τ. Then, we can solve the reduced problem by using an existing automated assertion checker such as MoCHi [5], [7], [10].

3. Reduction from Refinement Type Checking to Assertion Checking

Given a program t and a refinement type τ, our goal is to check whether t has type τ by reducing it to an assertion checking problem. If τ is an integer type of the form $\{x : \mathbf{int} \mid P\}$, then we can easily reduce the problem to the assertion checking problem of the program $\mathbf{let}\ r = t\ \mathbf{in\ assert}(P[r/x])$. If τ is a function type $(x : \tau_1) \to \tau_2$, we, roughly speaking, reduce the problem $\overset{?}{\models} t : (x : \tau_1) \to \tau_2$ to the problem $\overset{?}{\models} t\, t' : \tau_2[t'/x]$ for a "universal" term $t' = \alpha(\tau_1)$ that simulates all the terms of type τ_1. By using the synthesizer $\alpha(-)$ of universal terms, we reduce the refinement type checking problem

$$\overset{?}{\models} t : (x_1 : \tau_1) \to \cdots \to (x_n : \tau_n) \to \{r : \mathbf{int} \mid P\}$$

to the assertion checking problem

$$
\begin{aligned}
\overset{?}{\models}\ &\mathbf{let}\ x_1 = \alpha(\tau_1)\ \mathbf{in}\ \dots\ \mathbf{let}\ x_n = \alpha(\tau_n)\ \mathbf{in}\\
&\mathbf{let}\ r = t\, x_1 \dots x_n\ \mathbf{in\ assert}(P) : \mathbf{int}.
\end{aligned}
$$

$$\begin{aligned}\alpha(\{x:\mathbf{int}\mid P\}) &= \mathbf{let}\ x=\mathbf{rand\_int}\ \mathbf{in}\\ &\qquad \mathbf{assume}\,(P);\ x\\ \alpha((x:\tau_1)\to\tau_2) &= \lambda x.\,\mathbf{if}\ *\vee\beta(x:\tau_1)\ \mathbf{then}\ \alpha(\tau_2)\\ &\qquad \mathbf{else}\ \alpha_{\mathrm{S}}(\mathtt{ST}(\tau_2))\end{aligned}$$

$$\begin{aligned}\beta(v:\{x:\mathbf{int}\mid P\}) &= P[v/x]\\ \beta(v:(x:\tau_1)\to\tau_2) &= \mathbf{let}\ x=\alpha(\tau_1)\ \mathbf{in}\\ &\qquad \mathbf{let}\ r=v\,x\ \mathbf{in}\ \beta(r:\tau_2)\end{aligned}$$

$$\begin{aligned}\alpha_{\mathrm{S}}(\mathbf{int}) &= \mathbf{fail}\,\Box\,\mathbf{rand\_int}\\ \alpha_{\mathrm{S}}(\sigma_1\to\sigma_2) &= \mathbf{fail}\,\Box\,\lambda x.\,\alpha_{\mathrm{S}}(\sigma_2)\end{aligned}$$

Fig. 3 Synthesis of universal terms

We now define the synthesizer $\alpha(-)$: Types $\to$ Terms in Figure 3, where Types and Terms are the sets of types and terms respectively. Here, $\mathbf{assume}\,(t)$ is syntactic sugar for $\mathbf{if}\ t\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{fix}(f,\lambda x.\,f\,x)\,()$, and $t\vee t'$ is that for $\mathbf{if}\ t\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ t'$. In Figure 3, two auxiliary functions $\beta(-:-)$: Values $\times$ Types $\to$ Terms and $\alpha_{\mathrm{S}}(-)$: SimpleTypes $\to$ Terms are defined, where Values and SimpleTypes are the sets of values and simple types respectively. Roughly speaking, $\alpha(\tau)$ simulates all the values of type τ, and $\alpha_{\mathrm{S}}(\sigma)$ simulates all the answers of simple type σ. An integer term $\beta(v:\tau)$ is a Boolean expression that represents "v has type τ"; precisely, $\models v:\tau$ holds if and only if $\models_{\mathrm{P}} \beta(v:\tau)$ holds, i.e., $a=\mathbf{true}$ for all a s.t. $\beta(v:\tau)\longrightarrow^* a$ (which follows from Lemmas 9 and 10 in Section 4).

We now explain $\alpha(-)$ in more detail. For type $\tau=\{x:\mathbf{int}\mid P\}$, $\alpha(\tau)$ first generates a random integer value x, and checks whether x satisfies P or not. If x satisfies P, then $\alpha(\tau)$ returns x, and if not, $\alpha(\tau)$ diverges. Next, consider the case for $\tau=(x:\tau_1)\to\tau_2$. If the argument x of $\alpha(\tau)$ has type τ_1, then $\beta(x:\tau_1)$ always either diverges or evaluates to **true**; thus the body of $\alpha(\tau)$ non-deterministically diverges or is reduced to $\alpha(\tau_2)$. If $\models x:\tau_1$ does not hold, then $\beta(x:\tau_1)\longrightarrow^* \mathbf{false}$ or $\beta(x:\tau_1)\longrightarrow^* \mathbf{fail}$. Thus, the body of $\alpha(\tau)$ can be reduced to **fail** or $\alpha_{\mathrm{S}}(\mathtt{ST}(\tau_2))$, depending on the actual value of the argument x. (It can also be reduced to $\alpha(\tau_2)$ non-deterministically, but that does not matter.) In either case, **fail** or $\alpha_{\mathrm{S}}(\mathtt{ST}(\tau_2))$ serves as a universal term that simulates all the terms of the simple type $\mathtt{ST}(\tau_2)$, with respect to the simulation relation defined in Section 4.

For example, consider the type

$$\tau=((x:\mathbf{int})\to\{r:\mathbf{int}\mid r\geq x\})\to\{s:\mathbf{int}\mid s\geq 0\}\,.$$

Let $\tau_1=(x:\mathbf{int})\to\{r:\mathbf{int}\mid r\geq x\}$ and $\tau_2=\{s:\mathbf{int}\mid s\geq 0\}$, then

$$\begin{aligned}\alpha(\tau) &= \alpha(\tau_1\to\tau_2)\\ &= \lambda x.\,\mathbf{if}\ *\vee\beta(x:\tau_1)\ \mathbf{then}\ \alpha(\tau_2)\ \mathbf{else}\ \alpha_{\mathrm{S}}(\mathbf{int})\\ &= \lambda x.\,\mathbf{if}\ *\vee\beta(x:\tau_1)\ \mathbf{then}\\ &\qquad\quad \mathbf{let}\ s=\mathbf{rand\_int}\ \mathbf{in}\ \mathbf{assume}\,(s\geq 0);\ s\\ &\qquad \mathbf{else}\ \mathbf{fail}\,\Box\,\mathbf{rand\_int}\end{aligned}$$

where

$$\begin{aligned}\beta(x:\tau_1) &= \mathbf{let}\ x'=\alpha(\mathbf{int})\ \mathbf{in}\\ &\qquad \mathbf{let}\ r=x\,x'\ \mathbf{in}\ \beta(r:\{r:\mathbf{int}\mid r\geq x\})\\ &= \mathbf{let}\ x'=\mathbf{rand\_int}\ \mathbf{in}\ \mathbf{let}\ r=x\,x'\ \mathbf{in}\ r\geq x.\end{aligned}$$

The following theorem describes the correctness of the reduction.

Theorem 1. *Let t be a closed term of type $\mathtt{ST}(\tau_1)\to\cdots\to\mathtt{ST}(\tau_n)\to\mathbf{int}$. Then the following holds:*

$$\begin{aligned}&\models t:(x_1:\tau_1)\to\cdots\to(x_n:\tau_n)\to\{r:\mathbf{int}\mid P\}\\ &\iff\\ &\models \mathbf{let}\ r_0=t\ \mathbf{in}\\ &\quad \mathbf{let}\ x_1=\alpha(\tau_1)\ \mathbf{in}\ \mathbf{let}\ r_1=r_0\,x_1\ \mathbf{in}\\ &\qquad\vdots\\ &\quad \mathbf{let}\ x_n=\alpha(\tau_n)\ \mathbf{in}\ \mathbf{let}\ r_n=r_{n-1}\,x_n\ \mathbf{in}\\ &\quad \mathbf{assert}(P[r_n/r]):\mathbf{int}\end{aligned}$$

The theorem states that the reduction is sound and complete in the sense that the given program has the given refinement type if and only if the transformed program does not fail. We prove the theorem in the next section.

4. Proof of the Correctness of the Reduction

In this section, we prove the correctness of the reduction (Theorem 1). We first briefly sketch the proof of the following main lemma.

Lemma 2. $\models v_1:(x:\tau_1)\to\tau_2$ *if and only if* $\models v_1\,v_2:\tau_2[v_2/x]$ *for any* v_2 *such that* $\alpha(\tau_1)\longrightarrow^* v_2$.

The lemma intuitively states that, to check that v has function type $\tau_1\to\tau_2$, it is sufficient (and necessary) to check that $v\,(\alpha(\tau_1))$ has type τ_2. The "only-if" direction is trivial from the definition of $(\models)$ and (1) of Lemma 9 below. To show the "if" direction, we first show that $\alpha(\tau)$ simulates all the terms of type τ, i.e., for any term t of type τ and any context C, if $C[t]\longrightarrow^* \mathbf{fail}$, then $C[\alpha(\tau)]\longrightarrow^* \mathbf{fail}$. We also show that the simulation relation preserves typability, i.e., if t simulates t', then $\models t:\tau$ implies $\models t':\tau$. By the two properties above, we can show that $v\,(\alpha(\tau_1))$ simulates $v\,v'$ for any v' of type τ_1, and hence we have that $\models v\,(\alpha(\tau_1)):\tau_2$ implies $\models v\,v':\tau_2$.

In the above sketch, we used the observational (contextual) preorder to explain the notion of simulation simply, but in the proof below, we use the following definition of simulation.

Definition 3 (Simulation)**.** A *simulation* is a family of relations $\{\mathcal{R}^\sigma\}_\sigma$ such that $\mathcal{R}^\sigma$ is a relation between terms of simple type σ, and if $t_1\ \mathcal{R}^\sigma\ t_2$, then either $t_2 \longrightarrow^* \mathbf{fail}$ or the following hold:

- If $t_1 \longrightarrow^* n$, then $t_2 \longrightarrow^* n$.
- If σ is of the form $\sigma_1 \to \sigma_2$ and $t_1 \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_1')$, then there exists t_2' such that $t_2 \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_2')$ and $t_1'[\mathbf{fix}(f, \lambda x.\, t_1')/f][v_1/x]\ \mathcal{R}^{\sigma_2}\ t_2'[\mathbf{fix}(f, \lambda x.\, t_2')/f][v_2/x]$ for any values v_1 and v_2 such that $v_1\ \mathcal{R}^{\sigma_1}\ v_2$.
- If $t_1 \longrightarrow^* \mathbf{fail}$, then $t_2 \longrightarrow^* \mathbf{fail}$.

We define $\{\lesssim^\sigma\}_\sigma$ as the greatest simulation. For open terms t_1 and t_2, we also write $t_1 \lesssim^\sigma t_2$ if, for some simple type environment $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$,

- t_1 and t_2 have simple type σ under Γ, and
- $t_1[v_1/x_1, \ldots, v_n/x_n] \lesssim^\sigma t_2[v_1/x_1, \ldots, v_n/x_n]$ for any $v_1, \ldots, v_n$ such that v_i has type σ_i for each i.

To prove the main lemma (Lemma 2), we first show some basic properties of the simulation relation (Lemmas 4–6).

Lemma 4. *Suppose $t_1 \lesssim^\sigma t_2$. If $\models t_2 : \tau$, then $\models t_1 : \tau$.*

Proof. By induction on σ. Suppose $t_1 \lesssim^\sigma t_2$ and $\models t_2 : \tau$. If $t_1 \longrightarrow^* \mathbf{fail}$, by the assumption $t_1 \lesssim^\sigma t_2$, we have $t_2 \longrightarrow^* \mathbf{fail}$, which contradicts $\models t_2 : \tau$. We show $\models v : \tau$ for any v such that $t_1 \longrightarrow^* v$.

Case $v = n$: By the assumption $t_1 \lesssim^\sigma t_2$, we have $t_2 \longrightarrow^* n$ and $\models n : \tau$, as required.

Case $v = \mathbf{fix}(f, \lambda x.\, t_1')$: We have $\sigma = \sigma_1 \to \sigma_2$ for some σ_1 and σ_2. By the assumption $t_1 \lesssim^\sigma t_2$, there exists t_2' such that $t_2 \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_2')$ and $t_1'[\mathbf{fix}(f, \lambda x.\, t_1')/f][v_1/x] \lesssim^{\sigma_2} t_2'[\mathbf{fix}(f, \lambda x.\, t_2')/f][v_2/x]$ for any values v_1 and v_2 such that $v_1 \lesssim^{\sigma_1} v_2$. By the assumption $\models t_2 : \tau$, we have $\models t_2'[\mathbf{fix}(f, \lambda x.\, t_2')/f][v_2/x] : \tau_2[v_2/x]$ for any v_2 such that $\models v_2 : \tau_1$. Let $\tau = (x : \tau_1) \to \tau_2$, and v' be a value such that $\models v' : \tau_1$. Since $v' \lesssim^{\sigma_1} v'$, we get

$$\begin{aligned}
&\models t_2'[\mathbf{fix}(f, \lambda x.\, t_2')/f][v'/x] : \tau_2[v'/x]\\
\Rightarrow\ &\models t_1'[\mathbf{fix}(f, \lambda x.\, t_1')/f][v'/x] : \tau_2[v'/x]\\
&\text{(by I.H.)}\\
\Rightarrow\ &\models v\, v' : \tau_2[v'/x]\\
&\text{(since } v\, v' \preceq t_1'[\mathbf{fix}(f, \lambda x.\, t_1')/f][v'/x]).
\end{aligned}$$

Thus, we obtain $\models v : \tau$. □

Lemma 5. *If $v_1 \lesssim^\sigma v_2$, then $\tau[v_2/x] = \tau[v_1/x]$.*

Proof. If $\sigma = \mathbf{int}$, then we have $v_1 = v_2$. Therefore, we get $\tau[v_2/x] = \tau[v_1/x]$. If σ is a function type, since a variable of a function type cannot occur in τ, we have $\tau[v_2/x] = \tau = \tau[v_1/x]$. □

Lemma 6. *If $t_1 \lesssim^{\sigma_1 \to \sigma_2} t_2$ and $t_1' \lesssim^{\sigma_1} t_2'$, then $t_1\, t_1' \lesssim^{\sigma_2} t_2\, t_2'$.*

Proof. Suppose $t_1\, t_1' \longrightarrow^* v$. We have $t_1 \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_3)$, $t_1' \longrightarrow^* v_1$, $t_3[\mathbf{fix}(f, \lambda x.\, t_3)/f][v_1/x] \longrightarrow^* v$ for some t_3 and v_1. By the assumption that $t_1' \lesssim^{\sigma_1} t_2'$, we have $v_1 \lesssim^{\sigma_1} v_2$ for some v_2 such that $t_2' \longrightarrow^* v_2$. Therefore, by the assumption that $t_1 \lesssim^{\sigma_1 \to \sigma_2} t_2$, we get $t_3[\mathbf{fix}(f, \lambda x.\, t_3)/f][v_1/x] \lesssim^{\sigma_2} t_4[\mathbf{fix}(f, \lambda x.\, t_4)/f][v_2/x]$ for some t_4 such that $t_2 \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_4)$. □

Next, we show some properties of $\alpha(-)$ (Lemmas 7–11).

Lemma 7. *If $v \lesssim^\sigma \alpha(\tau)$, then there exists v' such that $\alpha(\tau) \longrightarrow^* v'$ and $v \lesssim^\sigma v'$.*

Proof. By case analysis on τ. □

Lemma 8. *Suppose $FV(\tau) = \{x\}$ and $\tau[v/x]$ is valid type, i.e., predicates in $\tau[v/x]$ are well-typed and have type* **int**. *Then, $\alpha(\tau)[v/x] = \alpha(\tau[v/x])$, and $\beta(v' : \tau)[v/x] = \beta(v' : \tau[v/x])$.*

Proof. By induction on the size of $\mathtt{ST}(\tau)$. □

Lemma 9. *For any type τ, the following hold.*
(1) $\models \alpha(\tau) : \tau$.
(2) If $\models v : \tau$, then $\beta(v : \tau) \preceq$ **true**.

Proof. By induction on the size of $\mathtt{ST}(\tau)$.

Case $\tau = \{x : \mathbf{int} \mid P\}$: By the definition of $\alpha(-)$, we have

$$\alpha(\tau) = \mathbf{let}\ x = \mathbf{rand\_int}\ \mathbf{in}\ \mathbf{assume}\,(P)\,;\ x.$$

We show that $\models \mathbf{assume}\,(P[n/x])\,;\ n : \tau$ for any integer n. Since P does not include applications and **rand_int**, there exist a unique v such that $P[n/x] \preceq v$. If $v = \mathbf{true}$, since $\models P[n/x]$ holds, we obtain $\models n : \tau$. If $v \neq \mathbf{true}$, since $\mathbf{assume}\,(P[n/x]) \Uparrow$, we have $\models \mathbf{assume}\,(P[n/x])\,;\ n : \tau$. Suppose $\models n' : \tau$ for some integer n'. $\beta(n' : \tau) = P[n'/x] \preceq \mathbf{true}$ follows from the definition of $\models n' : \tau$.

Case $\tau = (x : \tau_1) \to \tau_2$: By the definition of $\alpha(-)$, we have

$$\begin{aligned}
\alpha(\tau) = \lambda x.\, &\mathbf{if}\ * \vee \beta(x : \tau_1)\ \mathbf{then}\ \alpha(\tau_2)\\
&\mathbf{else}\ \alpha_{\mathrm{S}}(\mathtt{ST}(\tau_2)).
\end{aligned}$$

We show that $\models \alpha(\tau)\, v : \tau_2[v/x]$ for any v such that $\models v : \tau_1$. We get $\beta(v : \tau_1) \preceq \mathbf{true}$ by I.H. Therefore, we have $\alpha(\tau)\, v \preceq \alpha(\tau_2[v/x])$ by Lemma 8. Since $\models \alpha(\tau_2[v/x]) : \tau_2[v/x]$ by I.H., we get $\models \alpha(\tau)\, v : \tau_2[v/x]$. We next show that $\beta(v : \tau) \preceq \mathbf{true}$ for any v such that $\models v : \tau$. By the definition of $\beta(-)$, we have

$$\beta(v : \tau) = \mathbf{let}\ x = \alpha(\tau_1)\ \mathbf{in}\ \mathbf{let}\ r = v\, x\ \mathbf{in}\ \beta(r : \tau_2).$$

Suppose $\alpha(\tau_1) \longrightarrow^* v'$, $v\, v' \longrightarrow^* v''$, and

$$\begin{aligned}
\beta(v : \tau) &\longrightarrow^* \mathbf{let}\ r = v\, v'\ \mathbf{in}\ \beta(r : \tau_2)[v'/x]\\
&\longrightarrow^* \beta(v'' : \tau_2)[v'/x].
\end{aligned}$$

Since $\models \alpha(\tau_1) : \tau_1$ by I.H., we have $\models v' : \tau_1$ and $\models v'' : \tau_2[v'/x]$. By I.H., we get $\beta(v'' : \tau_2[v'/x]) \preceq \mathbf{true}$, and hence, $\beta(v'' : \tau_2)[v'/x] \preceq \mathbf{true}$ by Lemma 8. □

Lemma 10. *Let i be an integer and v be a value of simple type $\mathtt{ST}(\tau)$. Suppose $t \lesssim^{\mathtt{ST}(\tau')} \alpha(\tau')$ for any t and τ' such that $\models t : \tau'$ and $size(\mathtt{ST}(\tau')) < i$. If $\not\models v : \tau$ and $size(\mathtt{ST}(\tau)) = i$, one of the following holds:*

- *$\beta(v : \tau) \longrightarrow^*$* **false**, *or*
- *$\beta(v : \tau) \longrightarrow^*$* **fail**.

Proof. By induction on the simple type of τ.

Case $\tau = \{x : \mathbf{int} \mid P\}$: We have $v = n$ for some n. By the assumption that $\not\models v : \tau$, $P[v/x] \longrightarrow^* \mathbf{false}$.

Case $\tau = (x : \tau_1) \to \tau_2$: We have $v = \mathbf{fix}(f, \lambda x.\, t)$ for some t. Since $\not\models v : (x : \tau_1) \to \tau_2$, there exists v' such that $\models v' : \tau_1$ and $\not\models v\, v' : \tau_2[v'/x]$. By the assumption and $\mathit{size}(\mathtt{ST}(\tau_1)) < \mathit{size}(\mathtt{ST}(\tau)) = i$, we have $v' \lesssim^{\mathtt{ST}(\tau_1)} \alpha(\tau_1)$. Hence, we get $v\, v' \lesssim^{\mathtt{ST}(\tau_2)} v\, \alpha(\tau_1)$ by Lemma 6. Therefore, by Lemma 4, we get $\not\models v\, \alpha(\tau_1) : \tau_2[v'/x]$, i.e., there exists v_1 and a such that $\alpha(\tau_1) \longrightarrow^* v_1$, $v\, v_1 \longrightarrow^* a$, and $\not\models a : \tau_2[v'/x]$. If $a = \mathbf{fail}$, then we obtain $\beta(v : \tau) \longrightarrow^* \mathbf{fail}$. If $a = v_2$ for some v_2, since

$$\begin{aligned}\beta(v : \tau) &\longrightarrow^* \mathbf{let}\ r = v\, v_1\ \mathbf{in}\ \beta(r : \tau_2[v'/x]) \\ &\longrightarrow^* \beta(v_2 : \tau_2[v'/x]),\end{aligned}$$

we get $\beta(v : \tau) \longrightarrow^* \mathbf{false}$ or $\beta(v : \tau) \longrightarrow^* \mathbf{fail}$ by I.H. □

Lemma 11. *The followings hold:*
- *If* $\models t : \tau$, *then* $t \lesssim^{\mathtt{ST}(\tau)} \alpha(\tau)$.
- *If* t *has simple type* $\mathtt{ST}(\tau)$, *then* $t \lesssim^{\mathtt{ST}(\tau)} \alpha_{\mathrm{S}}(\mathtt{ST}(\tau))$.

Proof. By induction on the size of $\mathtt{ST}(\tau)$.

Case $\mathtt{ST}(\tau) = \mathbf{int}$ and $t \longrightarrow^* \mathbf{fail}$: If $\models t : \tau$, then it contradicts to the assumption. If t has simple type $\mathbf{int}$, we have $\alpha_{\mathrm{S}}(\mathbf{int}) = \mathbf{fail} \,\Box\, \mathbf{rand\_int} \longrightarrow \mathbf{fail}$.

Case $\mathtt{ST}(\tau) = \mathbf{int}$ and $t \longrightarrow^* n$: Suppose $\models t : \{x : \mathbf{int} \mid P\}$. Since $\models_{\mathrm{P}} P[n/x]$ and

$$\alpha(\tau) = \mathbf{let}\ x = \mathbf{rand\_int}\ \mathbf{in}\ \mathbf{assume}\,(P);\, x,$$

we get $\alpha(\tau) \longrightarrow^* n$. Therefore, we obtain $t \lesssim^{\mathbf{int}} \alpha(\tau)$. If t has simple type $\mathbf{int}$, we have $\alpha_{\mathrm{S}}(\mathbf{int}) = \mathbf{fail} \,\Box\, \mathbf{rand\_int} \longrightarrow^* n$.

Case $\mathtt{ST}(\tau) = \sigma_1 \to \sigma_2$ and $t \longrightarrow^* \mathbf{fail}$: Similar to the first case.

Case $\mathtt{ST}(\tau) = \sigma_1 \to \sigma_2$ and $t \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t')$: Suppose $\models t : (x : \tau_1) \to \tau_2$. We show that, there exists t_1 such that $\alpha(\tau) \longrightarrow^* \mathbf{fix}(f, \lambda x.\, t_1)$ and $t'[\mathbf{fix}(f, \lambda x.\, t')/f][v_1/x] \lesssim^{\sigma_2} t_1[\mathbf{fix}(f, \lambda x.\, t_1)/f][v_2/x]$ for any values v_1 and v_2 such that $v_1 \lesssim^{\sigma_1} v_2$. Let t_1 be $\mathbf{if}\ * \vee \beta(x : \tau_1)\ \mathbf{then}\ \alpha(\tau_2)\ \mathbf{else}\ \alpha(\mathtt{ST}(\tau_2))$, then $\alpha(\tau) = \mathbf{fix}(f, \lambda x.\, t_1)$. If $\models v_1 : \tau_1$, then we have $\models t'[\mathbf{fix}(f, \lambda x.\, t')/f][v_1/x] : \tau_2[v_1/x]$. Since $t_1[\mathbf{fix}(f, \lambda x.\, t_1)/f][v_2/x] \longrightarrow^* \alpha(\tau_2)[v_2/x] = \alpha(\tau_2[v_2/x]) = \alpha(\tau_2[v_1/x])$ by Lemmas 8 and 5, we get $t'[\mathbf{fix}(f, \lambda x.\, t')/f][v_1/x] \lesssim^{\mathtt{ST}(\tau_2)} t_1[\mathbf{fix}(f, \lambda x.\, t_1)/f][v_2/x]$ by I.H. If $\not\models v_1 : \tau_1$, then we have $\not\models v_2 : \tau_1$ by Lemma 4, and hence, $\beta(v_2 : \tau_1) \longrightarrow^* \mathbf{false}$ or $\beta(v_2 : \tau_1) \longrightarrow^* \mathbf{fail}$ by Lemma 10. Since $t_1[\mathbf{fix}(f, \lambda x.\, t_1)/f][v_2/x] \longrightarrow^* \mathbf{fail}$, we obtain $t'[\mathbf{fix}(f, \lambda x.\, t')/f][v_1/x] \lesssim^{\sigma_2} t_1[\mathbf{fix}(f, \lambda x.\, t_1)/f][v_2/x]$. Suppose t has simple type $\mathtt{ST}(\tau)$. Let t_1 be $\alpha_{\mathrm{S}}(\sigma_2)$ and v_1 and v_2 be values such that $v_1 \lesssim^{\sigma_1} v_2$, then $\alpha_{\mathrm{S}}(\mathtt{ST}(\tau)) = \lambda x.\, t_1$ and $t_1[v_2/x] = \alpha_{\mathrm{S}}(\sigma_2)[v_2/x] = \alpha_{\mathrm{S}}(\sigma_2)$. Hence, by I.H., we get $t'[\mathbf{fix}(f, \lambda x.\, t')/f][v_1/x] \lesssim^{\sigma_2} \alpha_{\mathrm{S}}(\sigma_2)[v_2/x]$. □

We now show the main lemma and Theorem 1.

Proof of Lemma 2. "Only-if" direction: Obvious from (1) of Lemma 9.

"If" direction: Suppose $\models v_1\, v_2 : \tau_2[v_2/x]$ for any v_2 such that $\alpha(\tau_1) \longrightarrow^* v_2$. We show that $\models v_1\, v_2' : \tau_2[v_2'/x]$ for any v_2' such that $\models v_2' : \tau_1$. We have $v_2' \lesssim^{\mathtt{ST}(\tau_1)} \alpha(\tau_1)$ by Lemma 11, and hence, by Lemma 7, there exists v_2'' such that $\alpha(\tau_1) \longrightarrow^* v_2''$ and $v_2' \lesssim v_2''$. By the assumption, we get $\models v_1\, v_2'' : \tau_2[v_2''/x]$. Therefore, we obtain $\models v_1\, v_2' : \tau_2[v_2'/x]$ by Lemmas 4 and 5. □

Proof of Theorem 1.

$$\begin{aligned}
&\models t : (x_1 : \tau_1) \to \cdots \to (x_n : \tau_n) \to \{r : \mathbf{int} \mid P\} \\
\iff\ &\forall a. t \longrightarrow^* a \Rightarrow \\
&\quad \models a : (x_1 : \tau_1) \to \cdots \to (x_n : \tau_n) \to \{r : \mathbf{int} \mid P\} \\
&\quad \text{(by the definition of } (\models)) \\
\iff\ &\forall a. t \longrightarrow^* a \Rightarrow \forall v_1, \ldots, v_n. \\
&\quad \bigwedge_{i \in \{1,\ldots,n\}} \alpha(\tau_i[v_j/x_j]_{j \in \{1,\ldots,i-1\}}) \longrightarrow^* v_i \Rightarrow \\
&\quad \models a\, v_1 \ldots v_n : \big\{r : \mathbf{int} \bigm| P[v_j/x_j]_{j \in \{1,\ldots,n\}}\big\} \\
&\quad \text{(by Lemma 2)} \\
\iff\ &\forall a. t \longrightarrow^* a \Rightarrow \forall v_1, \ldots, v_n. \\
&\quad \bigwedge_{i \in \{1,\ldots,n\}} \alpha(\tau_i[v_j/x_j]_{j \in \{1,\ldots,i-1\}}) \longrightarrow^* v_i \Rightarrow \\
&\quad \forall a'. a\, v_1 \ldots v_n \longrightarrow^* a' \Rightarrow \\
&\quad \models a' : \big\{r : \mathbf{int} \bigm| P[v_j/x_j]_{j \in \{1,\ldots,n\}}\big\} \\
&\quad \text{(by the definition of } (\models)) \\
\iff\ &\forall a. t \longrightarrow^* a \Rightarrow \forall v_1, \ldots, v_n. \\
&\quad \bigwedge_{i \in \{1,\ldots,n\}} \alpha(\tau_i[v_j/x_j]_{j \in \{1,\ldots,i-1\}}) \longrightarrow^* v_i \Rightarrow \\
&\quad \forall a'. a\, v_1 \ldots v_n \longrightarrow^* a' \Rightarrow (a' \neq \mathbf{fail} \wedge \\
&\quad \models \mathbf{assert}(P[v_j/x_j]_{j \in \{1,\ldots,n\}}[a'/r]) : \mathbf{int}) \\
&\quad \text{(by the definition of the semantics)} \\
\iff\ &\forall a. t \longrightarrow^* a \Rightarrow \forall v_1, \ldots, v_n. \\
&\quad \bigwedge_{i \in \{1,\ldots,n\}} \alpha(\tau_i[v_j/x_j]_{j \in \{1,\ldots,i-1\}}) \longrightarrow^* v_i \Rightarrow \\
&\quad \models \begin{matrix}\mathbf{let}\ r = a\, v_1 \ldots v_n\ \mathbf{in} \\ \mathbf{assert}(P[v_j/x_j]_{j \in \{1,\ldots,n\}})\end{matrix} : \mathbf{int} \\
&\quad \text{(by the definition of the semantics)} \\
\iff\ &\forall v_1, \ldots, v_n. \\
&\quad \bigwedge_{i \in \{1,\ldots,n\}} \alpha(\tau_i[v_j/x_j]_{j \in \{1,\ldots,i-1\}}) \longrightarrow^* v_i \Rightarrow \\
&\quad \models \begin{matrix}\mathbf{let}\ r = t\, v_1 \ldots v_n\ \mathbf{in} \\ \mathbf{assert}(P[v_j/x_j]_{j \in \{1,\ldots,n\}})\end{matrix} : \mathbf{int} \\
&\quad \text{(by the definition of the semantics)} \\
\iff\ &\models \mathbf{let}\ r_0 = t\ \mathbf{in}\ \mathbf{let}\ x_1 = \alpha(\tau_1)\ \mathbf{in}\ \mathbf{let}\ r_1 = r_0\, x_1\ \mathbf{in} \\
&\quad \ldots\ \mathbf{let}\ x_n = \alpha(\tau_n)\ \mathbf{in}\ \mathbf{let}\ r_n = r_{n-1}\, x_n\ \mathbf{in} \\
&\quad \mathbf{assert}(P[r_n/r]) : \mathbf{int} \\
&\quad \text{(by the definition of the semantics)}
\end{aligned}$$

Table 1 Results of preliminary experiments

problem	size	time [sec]
fsum_intro1	45	0.227
fsum_intro2	43	0.266
sum	28	0.096
mult	38	0.271
max	52	0.185
mc91	35	0.232
ack	41	0.131
a-cppr	155	1.481
a-dotprod	74	0.742
l-zipunzip	86	0.304
l-zipmap	80	0.168
sum_intro	36	0.108
copy_intro	26	0.113
sum-e	28	0.099
mult-e	38	0.179
mc91-e	35	0.112
harmonic-e	75	2.872
fold_right	55	0.950
forall_eq_pair	51	0.322
forall_leq	47	0.283
iter	43	0.224
harmonic	81	0.501
fold_left	55	0.941
fold_fun_list	81	0.269

□

5. Preliminary Experiments

To evaluate our method, we have implemented a refinement type checker. Our type checker uses MoCHi [5], [7], [10] as the underlying assertion checker. Most of the benchmark programs are taken from the benchmark of MoCHi [7]. The specification of each program is given by hand. To test the implementation for various programs, we have extended our method to deal with Booleans, pairs, and lists. We did not use some programs in the benchmark of MoCHi since the extended method cannot deal with algebraic data types, exceptions, and predicates about lengths of lists, which is just a limitation on the current implementation. We can naturally extend our method to deal with these features.

Table 1 shows the experimental results. The column "size" shows the word counts of the program and the refinement type as the specification. The experiment was conducted on Intel Core i7-3930K CPU with 12 MB cache and 16 GB memory. The implementation can be tested and all the programs are available at `http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/mochi_ref_assert/`.

All the programs have been verified correctly and fully automatically. Most of the program are verified within less than a second. Most of the time for verification has been spent by MoCHi, not the transformation given in the current paper. The problems "fsum_intro1" and "fsum_intro2" are the examples in Section 1. The other programs are taken from the benchmark of MoCHi. The problems below "fold_right" are about list manipulating programs. For example, "forall_eq_pair" is the problem to check that the forall function for lists have type

$$(\{(x,y) : \mathbf{int} \times \mathbf{int} \mid x = y\} \to \{r : \mathbf{bool} \mid r\}) \to$$
$$\{(x,y) : \mathbf{int} \times \mathbf{int} \mid x = y\}\ \mathbf{list} \to \{r : \mathbf{bool} \mid r\}.$$

If a programmer checks it by MoCHi alone instead of using our method, he/she needs to write the generators for $\{(x,y) : \mathbf{int} \times \mathbf{int} \mid x = y\} \to \{r : \mathbf{bool} \mid r\}$ and $\{(x,y) : \mathbf{int} \times \mathbf{int} \mid x = y\}$ **list**, which is harder than providing the refinement type of the specification above. The problems "xxx-e" are about wrong specifications. Since our reduction is complete and MoCHi can also check that the given program is actually unsafe, our verifier can also report that the given program actually does not have the given type.

6. Related Work

There are several pieces of work on automatic or semi-automatic inference on refinement types [4], [6], [8], [9], [12]. Unno and Kobayashi [9] and Jhala et al. [4] proposed automated refinement type inference methods based on constraints. Their methods first prepare templates of refinement types, generate constraints, and then solve them. Unno and Kobayashi [9] solve the constraints by using an interpolating theorem prover, and Jhala et al. [4] reduce the constraints to a verification problem for a first-order imperative program, and then verify it by using an existing model checker. Terauchi [8] proposed an automated refinement type inference method based on counterexample-guided refinement of refinement types. All the methods above are based on refinement type systems that are incomplete with respect to the semantics of refinement types; thus their overall methods are incomplete for the semantic refinement type checking problem. In contrast, our method reduces refinement type checking to assertion checking in a sound and complete manner; thus, our method is relatively complete with respect to the (hypothetical) completeness of an assertion checker. Even though there is actually no complete assertion checker, a stronger assertion checker enables stronger refinement type checking. For example, when using MoCHi as an underlying assertion checker, our method can verify that the following judgment is semantically valid [5].

$$\models \begin{array}{l} \mathbf{let}\ f\ x\ y = \\ \quad \mathbf{if}\ (x() > 0) \& (y() \leq 0)\ \mathbf{then\ fail\ else}\ 0\ \mathbf{in} \\ \mathbf{let}\ h\ x\ y = x\ \mathbf{in} \\ \mathbf{let}\ \mathtt{main}\ x = f\ (h\ x)\ (h\ x)\ \mathbf{in} \\ \mathtt{main} \\ \ : \mathbf{int} \to \mathbf{int} \end{array}$$

None of the three methods above [4], [8], [9] can verify this example, due to the limitation of the underlying refinement type systems. Another advantage of our approach is that, when a given program does not satisfy a refinement type specification, we can generate a concrete execution sequence in which the specification is violated. Terauchi's method [8] also generates a counterexample, but it is a fragment of the program that cannot be typed in the underlying type system, which is not necessarily a counterexample against the semantic refinement type checking problem.

Rondon et al. [6], and Zhu and Jagannathan [12] also proposed refinement type inference methods. Their methods are semi-automated, in the sense that these verifiers require users to give hints on predicates. In contrast, our verification method is fully automated; users need not supply any hints nor type annotations.

Dependent ML [11] is a functional language equipped with a restricted form of dependent types. Users must provide type annotations for all the functions.

Dependent types have been used in the context of interactive theorem provers. While more expressive types are allowed in such a context (e.g., function variables may be used in refinement predicates), users have to provide not just type annotations but also "proofs" that a given term has a given type.

7. Conclusion and Future Work

We have proposed a reduction from a refinement type checking problem for functional programs to an assertion checking problem, and proved its correctness. We have implemented a prototype verifier based on the reduction and confirmed that it works well for several programs.

There are several limitations in our method, as described below. Relaxing them is left for future work. First, the refinement types in this paper are restricted to first-order ones, where refinement predicates may contain only base-type variables. Second, we have not considered polymorphic types. It is an interesting issue whether and how we can define $\alpha(\tau)$ for a polymorphic type τ. Thirdly, as mentioned in Section 1, our method relies on the existence of non-determinism.

Acknowledgment

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

References

[1] Asada, K., Sato, R. and Kobayashi, N.: Verifying Relational Properties of Functional Programs by First-Order Refinement, *Proceedings of the ACM SIGPLAN 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM 2015)*, pp. 61–72 (2015).

[2] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D. and Maffeis, S.: Refinement types for secure implementations, *ACM Transactions on Programming Languages and Systems*, Vol. 33, No. 2, pp. 1–45 (2011).

[3] Freeman, T. and Pfenning, F.: Refinement types for ML, *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI 1991)*, pp. 268–277 (1991).

[4] Jhala, R., Majumdar, R. and Rybalchenko, A.: HMC: Verifying functional programs using abstract interpreters, *Proceedings of 23rd International Conference on Computer Aided Verification (CAV 2011)*, pp. 470–485 (2011).

[5] Kobayashi, N., Sato, R. and Unno, H.: Predicate abstraction and CEGAR for higher-order model checking, *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, pp. 222–233 (2011).

[6] Rondon, P. M., Kawaguchi, M. and Jhala, R.: Liquid types, *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2008)*, pp. 159–169 (2008).

[7] Sato, R., Unno, H. and Kobayashi, N.: Towards a scalable software model checker for higher-order programs, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM 2013)*, pp. 53–62 (2013).

[8] Terauchi, T.: Dependent Types from Counterexamples, *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pp. 119–130 (2010).

[9] Unno, H. and Kobayashi, N.: Dependent type inference with interpolants, *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pp. 277–288 (2009).

[10] Unno, H., Terauchi, T. and Kobayashi, N.: Automating relatively complete verification of higher-order functional programs, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*, pp. 75–86 (2013).

[11] Xi, H. and Pfenning, F.: Dependent types in practical programming, *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1999)*, pp. 214–227 (1999).

[12] Zhu, H. and Jagannathan, S.: Compositional and Lightweight Dependent Type Inference for ML, *Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2013)*, pp. 295–314 (2013).

Ryosuke Sato was born in 1985, and received his B.S. (in 2008), M.S. (in 2010), and D.S. degrees (in 2013) from Tohoku University. He is a post-doctoral researcher in the University of Tokyo. He is interested in program verification based on formal methods. He is a member of the ACM.

Kazuyuki Asada was born in 1981, and received his B.S. (in 2004), M.S. (in 2006), and D.S. degrees (in 2009) from Kyoto University. He is a post-doctoral researcher in the University of Tokyo. He is interested in semantics of programming languages and logic.

Naoki Kobayashi was born in 1968. He received his B.S., M.S., and D.S. degrees from University of Tokyo in 1991, 1993 and 1996, respectively. He is a professor in Department of Computer Science, Graduate School of Information Science and Technology, the University of Tokyo. His current major research interests are in principles of programming languages. In particular, he is interested in type systems and program verification.