Refinement Type Checking via Assertion Checking

RYOSUKE SATO¹.a) KAZUYUKI ASADA¹.b) NAOKI KOBAYASHI¹.c)

Received: February 9, 2015, Accepted: June 9, 2015

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.

Keywords: refinement types, assertion checking, automated verification

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:

\[ \text{let rec fsum f n =} \]
\[ \text{if } n \leq 0 \text{ then 0} \]
\[ \text{else } f n + fsum f (n-1) \]
\[ \text{let double n = n + n} \]
\[ \text{let main n = fsum double n} \]

Using a refinement type checker, one can verify that the function \( \text{main} \) has type \((x : \text{int}) \rightarrow [r : \text{int} \mid r \geq x]\), i.e., that for any integer \( x \), if \( \text{main} x \) 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.

\[ \vdash \text{main} : (x : \text{int}) \rightarrow [r : \text{int} \mid r \geq x], \]

The meaning of the second annotation is the same as that for \( \text{main} \) above, and the first annotation means that, for any value \( f \) that has type \((x : \text{int}) \rightarrow [r : \text{int} \mid r \geq x]\), if \( f \) evaluates to \( g \), then \( g \) is a value of type \((y : \text{int}) \rightarrow [s : \text{int} \mid s \geq y]\). 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

\[ \vdash \text{main} : (x : \text{int}) \rightarrow [r : \text{int} \mid r \geq x], \]

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

\[ \text{let rec fsum f n = ... in} \]
\[ \ldots \]
\[ \text{let n = rand_int in} \]
\[ \text{let r = main n in assert(r \geq n)) \]

Here, \( \text{rand_int} \) generates a random integer. While the original problem asks whether \( \text{main} n \) 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:

\[ \vdash \text{fsum} : (x : \text{int} \mid x > 0) \rightarrow [r : \text{int} \mid r \geq x] \rightarrow (y : \text{int}) \rightarrow [s : \text{int} \mid s \geq y]. \]

References:


Footnotes:

a) ryosuke@is.s.u-tokyo.ac.jp
b) asada@kb.is.s.u-tokyo.ac.jp
c) koba@is.s.u-tokyo.ac.jp
Following the random number generator approach above, one may be tempted to prepare a term \( \text{gen}() \) that non-deterministically generates every function of type \( \{ x : \text{int} \mid x > 0 \} \rightarrow \{ r : \text{int} \mid r \geq x \} \). Unfortunately, however, there is no such term \( \text{gen}() \), because the set of values of type \( \{ x : \text{int} \mid x > 0 \} \rightarrow \{ r : \text{int} \mid r \geq x \} \) is not recursively enumerable.

Instead of defining such a generator, we prepare a “universal” term \( r' \) that simulates all the terms of type \( \{ x : \text{int} \mid x > 0 \} \rightarrow \{ r : \text{int} \mid r \geq x \} \), in the sense that for any \( r'' \) of type \( \{ x : \text{int} \mid x > 0 \} \rightarrow \{ x : \text{int} \mid x \geq x \} \) and any \( n > 0 \), \( r'' n \rightarrow^* m \) implies \( r' n \rightarrow^* m \), and \( r' n \rightarrow^* \text{fail} \) implies \( r'' n \rightarrow^* \text{fail} \). Using \( r' \), we can reduce the problem (1) to the following problem:

\[
\tau \vdash f \text{sum} : (y : \text{int}) \rightarrow (s : \text{int} \mid s \geq y).
\]

The term \( r' \) above can be expressed as follows, by using non-determinism.

\[
\lambda x. \begin{cases} \text{if } x > 0 \text{ then} & r = \text{rand int in assume}(r \geq x); r \\
\text{else if } * \text{ then rand int else fail} & \end{cases}
\]

Here, \( * \) is a non-deterministic Boolean, and \( \text{assume}(b) \) returns the unit value if \( b = \text{true} \) 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 \( \tau \) can be constructed by induction on the structure of \( \tau \).

Using the term \( r' \) above, we can reduce the problem (2) to the assertion checking problem for the following program.

\[
\begin{align*}
\text{let } & \text{rec fsum f n } = \ldots \\
& \text{let } g x = \\
& \quad \text{if } x > 0 \text{ then} \\
& \quad \text{let } r = \text{rand int in assume}(r \geq x); r \\
& \quad \text{else if } * \text{ then rand int else fail} \\
& \text{let } n = \text{rand int in assert}(\text{fsum} g n \geq n)
\end{align*}
\]

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:

\[
\vdash \lambda f.(f 0 = 0) : (\text{int} \rightarrow \text{int}) \rightarrow (r : \text{bool} \mid r = \text{true})
\]

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

\[
\text{assert}((\lambda f.(f 0 = 0)) \alpha(\text{int} \rightarrow \text{int})),
\]

where \( \alpha(\text{int} \rightarrow \text{int}) \) is a non-deterministic function \( \lambda x.\text{rand int} \). The program may fail, since \( \lambda x.\text{rand int} \) 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:

\[
\{ f : \text{int} \rightarrow \text{int} \mid \forall x.f(x) = f(x) \rightarrow (r : \text{bool} \mid r = \text{true}) \}
\]

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:

\[
t \text{(terms) ::= } x \mid n \mid \text{op}(t_1, \ldots, t_n) \mid \text{rand int} \\
& \quad | \text{fix}(f, \lambda x. t) \mid t_1 t_2 \mid \text{fail} \\
& \quad | \text{if } t \text{ then } t_1 \text{ else } t_2
\]

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 metavariable \( n \). The meta-variable \( op \) ranges over primitive operations on integers and a term \( \text{op}(t_1, \ldots, t_n) \) is the application of \( op \) to \( t_1, \ldots, t_n \). We express Booleans by integers, and write \( \text{true} \) for 1, and \( \text{false} \) for 0. The term \( \text{rand int} \) is a non-deterministic integer.

We write \( * \) for a non-deterministic Boolean, which can be expressed by \( \text{rand int} = 0 \), and \( t_1 \sqcup t_2 \) for \( \text{if } \ast \text{ then } t_1 \text{ else } t_2 \). A term \( \text{fix}(f, \lambda x. t) \) denotes the recursive function defined by \( f = \lambda x. t \). When \( f \) does not occur in \( t \), we write \( \lambda x.t \) for \( \text{fix}(f, \lambda x. t) \).

A term \( t_1 t_2 \) is the application of \( t_1 \) to \( t_2 \). We write \( \text{let } x = t \in t' \) for \( (\lambda x.t')t \), and write also \( r, t' \) for it when \( x \) does not occur in \( t' \).
The special term \texttt{fail} aborts the execution. It is typically used to express assertions; \texttt{assert(t)}—which asserts that \( t \) should evaluate to \texttt{true}—is expressed by \( \texttt{if} \ t \ \texttt{then} \ \texttt{true} \ \texttt{else} \ \texttt{fail} \).

Bound and free variables are defined in a standard manner, and we identify \( a \)-equivalent terms. We call a closed term a \textit{value}. The syntax of \textit{types} is subject to the usual scope rule; in \( \texttt{integer x} \) (where \( \texttt{int} \) is a simple type, and that all the variables occurring in a predicate are integers), and that all the variables occurring in a predicate are integers.

We express a specification of a program by using a refinement type. The syntax of \textit{refinement types} is given by the following rules.

\[
\begin{align*}
\tau \text{ (types)} &::= [x : \texttt{int} \mid P] \mid (x : \tau_1) \rightarrow \tau_2 \\
P \text{ (predicates)} &::= n \mid \text{op}(P_1, \ldots, P_n)
\end{align*}
\]

A type \( (x : \tau_1) \rightarrow \tau_2 \) is a dependent product type, where \( x \) may occur in \( \tau_2 \). Intuitively, a \textit{refinement type} \( [x : \texttt{int} \mid P] \) represents the set of integers \( x \) that satisfy the \textit{refinement predicate} \( P \). For example, \( [x : \texttt{int} \mid x > 0] \) describes positive integers. The type \( (x : \texttt{int}) \rightarrow [r : \texttt{int} \mid r > x] \) 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 \( (x : \tau_1) \rightarrow \tau_2 \), the scope of \( x \) is \( \tau_2 \). Furthermore, we require that every refinement predicate is well-typed and has type \texttt{int} (recall that Booleans are expressed by integers), and that all the variables occurring in a predicate are integer variables. We often write just \( \texttt{int} \) for \( [x : \texttt{int} \mid \texttt{true}] \), and \( \tau_1 \rightarrow \tau_2 \) for \( (x : \tau_1) \rightarrow \tau_2 \) if \( x \) does not occur in \( \tau_2 \).

A type \( \tau \) is \textit{simple} if all the predicates in \( \tau \) are \texttt{true}. For a type \( \tau \), we define the \textit{simple type} \( \text{ST}(\tau) \) as follows:

\[
\begin{align*}
\text{ST}(x : \texttt{int} \mid P) &= \texttt{int} \\
\text{ST}((x : \tau_1) \rightarrow \tau_2) &= \text{ST}(\tau_1) \rightarrow \text{ST}(\tau_2).
\end{align*}
\]

We use a meta-variable \( \sigma \) for simple types. For a simple type \( \sigma \), we define the \textit{size} of \( \sigma \) as follows:

\[
\begin{align*}
\text{size}(\texttt{int}) &= 1 \\
\text{size}(\tau_1 \rightarrow \tau_2) &= 1 + \text{size}(\tau_1) + \text{size}(\tau_2)
\end{align*}
\]

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

For a program \( t \) and a type \( \tau \), the \textit{type checking problem} \( \models : : t : \tau \) asks whether \( \models : : t : \tau \) holds. An \textit{assertion checking problem} is a special case, where \( \tau = \texttt{int} \); note that \( \models : : t : \texttt{int} \) 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 \( \models : : t : \tau \) to the assertion checking problem \( \models : : t' : \texttt{int} \) by synthesizing \( t' \) from \( t \) and \( \tau \). 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 \( \tau \), our goal is to check whether \( t \) has type \( \tau \) by reducing it to an assertion checking problem. If \( \tau \) is an integer type of the form \( [x : \texttt{int} \mid P] \), then we can easily reduce the problem to the assertion checking problem of the program \( \text{let} \ r = t \ \text{in} \ \text{assert}(P[r/x]) \). The type \( \tau \) is a function type \( (x : \tau_1) \rightarrow \tau_2 \); we, roughly speaking, reduce the problem \( \models : : t : (x : \tau_1) \rightarrow \tau_2 \) to the problem \( \models : : t' : (x : \tau_1)[r/x] \) for a “universal” term \( t' = \sigma(t_1) \) that simulates all the terms of type \( \tau_1 \). By using the synthesizer \( \sigma(\cdot) \) of universal terms, we reduce the refinement type checking problem

\[
\models : : t : (x_1 : \tau_1) \rightarrow \ldots \rightarrow (x_n : \tau_n) \rightarrow [r : \texttt{int} \mid P]
\]

to the assertion checking problem

\[
\models : : \text{let} \ x_1 = \sigma(t_1) \text{ in } \ldots \text{ let } x_n = \sigma(t_n) \text{ in } \\
\text{let } t = t_1 \ldots t_n \text{ in } \text{assert}(P) : \texttt{int}.
\]

We now define the synthesizer \( \sigma(\cdot) : \text{Types} \rightarrow \text{Terms} \) in \textbf{Fig. 3}, where Types and Terms are the sets of types and terms respectively. Here, \texttt{assume}(t) is syntactic sugar for \( \text{if} \ t \ \text{then} \ \text{true} \ \text{else} \ \text{fix} \ (f, \lambda x. \ f(x)) \), and \( t \land t' \) is that for \( \text{if} \ t \ \text{then} \ \text{true} \ \text{else} \ t' \). In \textbf{Fig. 3}, two auxiliary functions \( \beta(\cdot) : \text{Values} \rightarrow \text{Terms} \) and \( \alpha(\cdot) : \text{SimpleTypes} \rightarrow \text{Terms} \) 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 \( \tau \), and \( \alpha(\tau) \) simulates all the answers of simple type \( \tau \). An integer term \( \beta(v : \tau) \) is a Boolean expression that represents \( \text{"} v \text{ has type } \tau \text{"} \); precisely, \( \models : : v : \tau \) holds if and only if \( \beta(\tau) \) holds, i.e., \( v = \texttt{true} \) for all \( a \) s.t. \( \beta(v : \tau) \rightarrow a \) (which follows from Lemmas 9 and 10 in Section 4).
Theorem 1. Let $t$ be a closed term of type $\text{ST}(\tau_1) \rightarrow \cdots \rightarrow \text{ST}(\tau_n) \rightarrow \text{int}$. Then the following holds:

\[ \vdash t : (x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow [r : \text{int} | P] \]

\[ \iff \]

\[ \vdash \text{let } x_1 = \alpha(\tau_1) \text{ in } \ldots \text{ let } x_n = \alpha(\tau_n) \text{ in } \]

\[ \vdash t = t_1 \ldots t_n \text{ in assert}(P) : \text{int} \]

Theorem 1 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.** $\vdash v_1 : (x : \tau_1) \rightarrow \tau_2$ if and only if $\vdash v_1 v_2 : \tau_2 v_2/x$ for any $v_2$ such that $\alpha(\tau_1) \rightarrow^* v_2$.

The lemma intuitively states that, to check that $v$ has function type $\tau_1 \rightarrow \tau_2$, it is sufficient (and necessary) to check that $v (\alpha(\tau_1))$ has type $\tau_2$. 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 $\tau$, i.e., for any term $t$ of type $\tau$ and any context $C$, if $C[t] \rightarrow^* \text{fail}$, then $C[\alpha(\tau)] \rightarrow^* \text{fail}$. We also show that the simulation relation preserves typability, i.e., if $t$ simulates $t'$, then $\vdash t : \tau$ implies $\vdash t' : \tau$. By the two properties above, we can show that $\alpha(\tau)$ simulates $v v'$ for any $v'$ of type $\tau_1$, and hence we have that $\vdash v (\alpha(\tau_1)) : \tau_2$ implies $\vdash v v' : \tau_2$.

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 $\times$ is a family of relations $\times_{\tau}$ such that $\times_{\tau}$ is a relation between terms of simple type $\sigma$, and if $t : \tau \rightarrow^* t'$, then either $t \rightarrow^* \text{fail}$ or the following hold:

- If $t_1 \rightarrow^* n$, then $t_2 \rightarrow^* n$.
- If $t_1 = t_1'$, then there exists $t_2$ such that $t_2 \rightarrow^* \text{fix}(f, \lambda x. t_1')$.
- If $t_1 \rightarrow^* \text{fail}$, then $t_2 \rightarrow^* \text{fail}$.

We define $\vdash \alpha(\tau) : \tau$ as the greatest simulation. For open terms $t_1$ and $t_2$, we also write $t_1 \leq^* t_2$ if, for some simple type environment $\Gamma = x_1 : \sigma_1, \ldots, x_n : \sigma_n$,

- $t_1$ and $t_2$ have the same type $\sigma$ under $\Gamma$, and
- $t_1[v_1/x_1, \ldots, v_n/x_n] \leq^* t_2[v_1/x_1, \ldots, v_n/x_n]$ for any $v_1, \ldots, v_n$ such that $v_i$ has type $\sigma_i$ 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 $t_1 \leq^* t_2$. If $\vdash t_2 : \tau$, then $\vdash t_1 : \tau$.

**Proof.** By induction on $\sigma$. Suppose $t_1 \leq^* t_2$ and $\vdash t_2 : \tau$. If $t_1 \rightarrow^* \text{fail}$, by the assumption $t_1 \leq^* t_2$, we have $t_2 \rightarrow^* \text{fail}$, which contradicts $\vdash t_2 : \tau$. We show that $\vdash v : \tau$ for any $v$ such that $t_1 \rightarrow^* v$.

Case $v = n$: By the assumption $t_1 \leq^* t_2$, we have $t_2 \rightarrow^* n$ and $\vdash n : \tau$, as required.

Case $v = \text{fix}(f, \lambda x. t_1')$: We have $\sigma = \sigma_1 \rightarrow \sigma_2$ for some $\sigma_1$ and $\sigma_2$. By the assumption $t_1 \leq^* t_2$, there exists $t_2'$ such that $t_2' \rightarrow^* \text{fix}(f, \lambda x. t_1')$ and $t_2' [\text{fix}(f, \lambda x. t_1')/f][v_1/x] \leq^* t_2' [\text{fix}(f, \lambda x. t_1')/f][v_2/x]$ for any values $v_1$ and $v_2$ such that $v_1 \leq^* v_2$. By the assumption $\vdash t_2 : \tau$, we have $\vdash t_2' [\text{fix}(f, \lambda x. t_1')/f][v_2/x] : \tau_2[v_2/x]$ for any $v_2$ such that $\vdash v_2 : \tau_1$.

Let $\tau = (x : \tau_1) \rightarrow \tau_2$, and $v'$ be a value such that $\vdash v' : \tau_1$. Since
Lemma 7. Suppose $v \not\vdash \beta$. Thus, we obtain $\Rightarrow v' : \tau_2[v'/x]$.

Thus, we obtain $\vdash v : \tau$. □

Lemma 5. If $v_1 \not\vdash \tau_2$, then $\tau[v_2/x] = \tau[v_1/x]$.

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

Lemma 6. Suppose $t_1 \not\vdash \sigma_1$, $t_2$ and $t_1' \not\vdash \sigma_2$, then $t_1 t_1' \not\vdash \sigma_2 t_2$.

Proof. Suppose $t_1 t_1' \rightarrow v$. We have $t_1 \rightarrow * \text{fix}(f, \lambda t_3. t_1 f)[v_1/x] \rightarrow v$ for some $t_1$ and $v_1$. By the assumption that $t_1' \not\vdash \sigma_1$, we have $v_1 \not\vdash \tau_1$ for some $v_1$ such that $t_1' \vdash \tau_1$. Therefore, by the assumption that $t_1 \not\vdash \sigma_1$, we get $t_1 \text{fix}(f, \lambda t_3. t_1 f)[v_1/x] \not\vdash \sigma_1 t_1 \text{fix}(f, \lambda t_4. t_2 f)[v_2/x]$ for some $t_2$ such that $t_2 \rightarrow * \text{fix}(f, \lambda t_4. t_4)$.

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

Lemma 10. Let $i$ be an integer and $v$ be a value of simple type ST$(\tau)$. Suppose $t \not\in \text{ST}(\tau)$ for any $t \neq \tau$ and size$(\text{ST}(\tau)) < i$. If $\not\vdash v : \tau$ and size$(\text{ST}(\tau)) = i$, one of the following holds:

- $\beta(v : \tau) \rightarrow \text{false}$, or
- $\beta(v : \tau) \rightarrow \text{fail}$.

Proof. By induction on the simple type of $\tau$.

Case $\tau = \{x : \text{int} | P\}$: We have $v = n$ for some $n$. By the assumption that $\not\vdash v : \tau$, $P[n/x] \rightarrow \text{false}$.

Case $\tau = \{x : \tau_1 \rightarrow \tau_2\}$: We have $v = \text{fix}(f, \lambda x. t)$ for some $t$. Since $\not\vdash v : \tau_1 \rightarrow \tau_2$, there exists $\alpha$ such that $\not\vdash \alpha : \tau_1$ and $\not\vdash \alpha' : \tau_2$. By the assumption and size$(\text{ST}(\tau)) < \text{size}(\text{ST}(\tau)) = i$, we have $\not\vdash v' : \text{ST}(\tau)$.

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

Lemma 7. If $v \not\vdash \alpha(\tau)$, then there exists $v'$ such that $\alpha(\tau) \rightarrow \alpha(\tau)$.

Proof. By case analysis on $\tau$.

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

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

Lemma 9. For any type $\tau$, the following hold.

1. $\vdash \alpha(\tau) : \tau$.
2. $\vdash v : \tau$, then $\beta(v : \tau) \rightarrow \text{true}$.

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

Case $\tau = \{x : \text{int} | P\}$: By the definition of $\alpha(\cdot)$, we have $\alpha(\tau) = \lambda x . \text{random} \text{ int} \text{ in } \text{assume}(P); x$.

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

Case $\tau = \{x : \tau_1 \rightarrow \tau_2\}$: By the definition of $\alpha(\cdot)$, we have $\alpha(\tau) = \lambda x . \text{if } v \not\vdash \beta(x : \tau_1) \text{ then } \alpha(\tau_2) \text{ else } \alpha(\text{ST}(\tau_2))$.

We show that $\vdash \alpha(\tau)(v : \tau)(v_1/x) : \tau_2[v_1/x]$ for any $v$ such that $\vdash v : \tau_1$. We get $\beta(v : \tau_1) \rightarrow \text{true}$ by L.H. Therefore, we have $\alpha(\tau)(v) \not\vdash \alpha(\text{ST}(\tau_2))$ by Lemma 8. Since $\vdash \alpha(\text{ST}(\tau_2))(v_1/x)$ by L.H., we get $\vdash \alpha(\tau)(v : \tau_2)[v_1/x]$. We next show that $\beta(v : \tau) \rightarrow \text{true}$ for any $v$ such that $\vdash v : \tau$. By the definition of $\beta(\cdot)$, we have $\beta(v : \tau) = \lambda x . \alpha(\tau_1) \text{ in } \lambda r . v \times v \in \beta(r : \tau_2)$.

Suppose $\alpha(\tau_1) \rightarrow \alpha(\tau_2) \text{ fail}$, and $\beta(v : \tau) \rightarrow \text{let } r \rightarrow v \rightarrow \alpha(\tau_2) \text{ in } \beta(r : \tau_2)[v'/x]$. Since $\vdash \alpha(\tau_1) : \tau_1$ by L.H., we have $\vdash v' : \tau_1$ and $\vdash v' : \tau_2[v'/x]$. By L.H., we get $\beta(v' : \tau_2)[v'/x] \rightarrow \text{true}$, and hence, $\beta(v' : \tau_2)[v'/x] \rightarrow \text{true}$ by Lemma 8.
Since $t_1(\text{fix}(f, \lambda x.t)/f)(v_2/x) \rightarrow^* \alpha(t_2)_2/x = \alpha(t_2)(v_2/x)$ by Lemmas 8 and 5, we get $t(\text{fix}(f, \lambda x.t)/f)(v_1/x) \not\rightarrow^{ST(t)} t_1(\text{fix}(f, \lambda x.t)/f)(v_2/x)$ by I.H. If $\not\exists v_1 : \tau_1$, then we have $\not\exists v_2 : \tau_1$ by Lemma 4, and hence, $\beta(t_2 : \tau_1) \not\rightarrow \text{false}$ or $\beta(v_2 : \tau_1) \not\rightarrow \text{fail}$ by Lemma 10. Since $t_1(\text{fix}(f, \lambda x.t)/f)(v_2/x) \rightarrow^* \text{false}$, we obtain $t(\text{fix}(f, \lambda x.t)/f)(v_1/x) \not\rightarrow^{S(T)} t_1(\text{fix}(f, \lambda x.t)/f)(v_2/x)$. 

Suppose $t$ has simple type $ST(t)$. Let $t_2$ be $\alpha_S(v_2)$ and $v_1$ and $v_2$ be values such that $v_1 \not\equiv v_2$, then $\alpha_S(ST(t)) = \lambda x.t_1$ and $t_1(v_2/x) = \alpha_S(v_2)(v_2/x) = \alpha_S(v_2)$. Hence, by I.H., we get $t(\text{fix}(f, \lambda x.t)/f)(v_1/x) \not\rightarrow^{S(T)} \alpha_S(v_2)(v_2/x)$. □

We now show the main lemma and Theorem 1.


“IF” direction: Suppose $\exists v_1 v_2 : \tau_1[v_2/x]$ for any $v_2$ such that $\alpha(t_1) \rightarrow v_2$. We show that $\exists v_1 v_2 : \tau_1[v_2/x]$ for any $v_2$ such that $\exists v_2 : \tau_1$. We have $v_2 \not\equiv \tau_1$ by $\alpha_S(ST(t))$ by Lemma 11, and hence, by Lemma 7, there exists $v_2'$ such that $\alpha(t_1) \rightarrow v_2'$ and $v_2' \not\equiv v_2$. By the assumption, we get $\exists v_1 v_2 : \tau_1[v_2'/x]$. Therefore, we obtain $\exists v_1 v_2 : \tau_1[v_2/x]$ by Lemmas 4 and 5. □

Proof of Theorem 1.

$\models \tau \vdash (x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow [r : \text{int} \mid P]$

$\equiv \forall \tau \vdash a \Rightarrow$

$\models \tau \vdash (x_1 : \tau_1) \rightarrow \cdots \rightarrow (x_n : \tau_n) \rightarrow [r : \text{int} \mid P]$

(by the definition of $\models$)

$\equiv \forall \tau \vdash a \Rightarrow \forall v_1 \ldots v_n$. 

$\bigwedge_{i \in [1 \ldots n]} \alpha(t_i[v_i/x_i]_{i \in [1 \ldots n]}) \rightarrow v_i \Rightarrow$

$\models \tau \vdash a \Rightarrow v_1 \ldots v_n : [r : \text{int} \mid P[v_i/x_i]_{i \in [1 \ldots n]}]$

(by Lemma 2)

$\equiv \forall \tau \vdash a \Rightarrow \forall v_1 \ldots v_n$. 

$\bigwedge_{i \in [1 \ldots n]} \alpha(t_i[v_i/x_i]_{i \in [1 \ldots n]}) \rightarrow v_i \Rightarrow$

$\forall a' a v_1 \ldots v_n \rightarrow a' \Rightarrow$

$\models \tau \vdash a' \Rightarrow [r : \text{int} \mid P[v_i/x_i]_{i \in [1 \ldots n]}]$

(by the definition of $\models$)

$\equiv \forall \tau \vdash a \Rightarrow \forall v_1 \ldots v_n$. 

$\bigwedge_{i \in [1 \ldots n]} \alpha(t_i[v_i/x_i]_{i \in [1 \ldots n]}) \rightarrow v_i \Rightarrow$

$\forall a' a v_1 \ldots v_n \rightarrow a' \Rightarrow (a' \neq \text{fail} \land$

$\models \text{assert}(P[v_i/x_i]_{i \in [1 \ldots n]}(a'/r)) : \text{int})$

(by the definition of the semantics)

$\equiv \forall \tau \vdash a \Rightarrow \forall v_1 \ldots v_n$. 

$\bigwedge_{i \in [1 \ldots n]} \alpha(t_i[v_i/x_i]_{i \in [1 \ldots n]}) \rightarrow v_i \Rightarrow$

$\equiv \forall v_1 \ldots v_n$. 

$\bigwedge_{i \in [1 \ldots n]} \alpha(t_i[v_i/x_i]_{i \in [1 \ldots n]}) \rightarrow v_i \Rightarrow$

$\models r = a v_1 \ldots v_n \text{ in }$

$\text{assert}(P[v_i/x_i]_{i \in [1 \ldots n]} : \text{int})$

(by the definition of the semantics)

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 auto-

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

\[(x, y) : \text{int} \times \text{int} \mid x = y \rightarrow (r : \text{bool} \mid r) \rightarrow (x, y) : \text{int} \times \text{int} \mid x = y \rightarrow \text{list} \rightarrow (r : \text{bool} \mid r).
\]

If a programmer checks it by MoChI alone instead of using our method, he/she needs to write the generators for \([(x, y) : \text{int} \times \text{int} \mid x = y \rightarrow (r : \text{bool} \mid r)\) and \([(x, y) : \text{int} \times \text{int} \mid x = y \rightarrow \text{list} \rightarrow (r : \text{bool} \mid r)\]., which is harder than providing the refinement type of the specification above. The problems “xxx-e” are about wrong specifications. Since our reduction is relatively complete with respect to the (hypothetical) completeness of our method, any assertion checker. Even though there is actually no hint 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 \(o(r)\) for a polymorphic type \(\tau\). Third, as mentioned in Section 1, our method relies on the existence of non-determinism.

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

References

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 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 the 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.