Relatively Complete Refinement Type System for Verification of Higher-Order Non-deterministic Programs

HIROSHI UNNO, University of Tsukuba, Japan
YUKI SATAKE, University of Tsukuba, Japan
TACHIO TERAUCHI, Waseda University, Japan

This paper considers verification of *non-deterministic* higher-order functional programs. Our contribution is a novel type system in which the types are used to express and verify (conditional) safety, termination, non-safety, and non-termination properties in the presence of $\forall$-$\exists$ branching behavior due to non-determinism. For instance, the judgement $\vdash e : \{u{:}\mathtt{int} \mid \phi(u)\}^{\forall\forall}$ says that every evaluation of e either diverges or reduces to some integer u satisfying $\phi(u)$, whereas $\vdash e : \{u{:}\mathtt{int} \mid \psi(u)\}^{\exists\forall}$ says that there exists an evaluation of e that either diverges or reduces to some integer u satisfying $\psi(u)$. Note that the former is a safety property whereas the latter is a counterexample to a (conditional) termination property. Following the recent work on type-based verification methods for deterministic higher-order functional programs, we formalize the idea on the foundation of *dependent refinement types*, thereby allowing the type system to express and verify rich properties involving program values, branching behaviors, and the combination thereof.

Our type system is able to seamlessly combine deductions of both universal and existential facts within a unified framework, paving the way for an exciting opportunity for new type-based verification methods that combine both universal and existential reasoning. For example, our system can prove the existence of a path violating some safety property from a proof of termination that uses a well-foundedness termination argument. We prove that our type system is sound and relatively complete, and further, thanks to having both modes of non-determinism, we show that our types are closed under complement.

CCS Concepts: • **Theory of computation → Programming logic**; **Program verification**; • **Software and its engineering → Formal software verification**;

Additional Key Words and Phrases: Program Verification, Relative Completeness, Refinement Types, Higher-Order Programs, Non-Deterministic Programs

ACM Reference Format:
Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively Complete Refinement Type System for Verification of Higher-Order Non-deterministic Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 12 (January 2018), 29 pages. https://doi.org/10.1145/3158100

1 INTRODUCTION

Recent years have seen remarkable advances in automated (and semi-automated) methods for verifying higher-order functional programs. Powerful verification methods have emerged for various important classes of properties, including safety properties [Jhala et al. 2011; Kobayashi et al. 2011; Ong and Ramsay 2011; Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009; Unno et al.

Authors' addresses: Hiroshi Unno, Computer Science, University of Tsukuba, Japan, uhiro@cs.tsukuba.ac.jp; Yuki Satake, Computer Science, University of Tsukuba, Japan, satake@logic.cs.tsukuba.ac.jp; Tachio Terauchi, Computer Science and Engineering, Waseda University, Japan, terauchi@waseda.jp.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
2475-1421/2018/1-ART12
https://doi.org/10.1145/3158100

2013; Zhu and Jagannathan 2013; Zhu et al. 2015], termination [Kuwahara et al. 2014; Vazou et al. 2014], non-termination [Hashimoto and Unno 2015; Kuwahara et al. 2015], and properties expressed in temporal logics such as LTL and linear modal μ-calculus [Koskinen and Terauchi 2014; Murase et al. 2016].

However, the existing proposals are quite disparate in that they employ rather different techniques to verify the different classes of properties. For instance, the termination verification proposed in [Kuwahara et al. 2014] and the linear modal μ-calculus verification proposed in [Murase et al. 2016] use program transformation to iteratively convert the verification problem to a (binary) reachability problem [Cook et al. 2007, 2006] which is checked by a dependent-refinement-type-based safety property verifier such as [Unno et al. 2013], whereas the non-termination verification of [Kuwahara et al. 2015] uses predicate abstraction to iteratively build over and under approximations to reduce the problem to that of higher-order model checking [Kobayashi 2009; Ong 2006]. (Recall that the *termination property* asks that every evaluation of the given program terminates, whereas the *non-termination property* is its converse and checks that the given program has a non-terminating evaluation.)

It is our belief that the disparity partly comes from the fact that each class of properties concerns only one side of *non-determinism*. That is, while properties such as safety and termination (and, more generally, any *linear* properties) only ask that a certain fact holds for *every* run of the program, properties such as non-termination only ask whether there *exists* a run that satisfies a certain fact. Likewise, the disparity between safety and termination can be understood as the difference due to the former asking that *every* value that can be obtained as the final result satisfies a certain fact whereas the latter asking that there *exists* a value that can be obtained as the final result (for every run, in both cases).

In the present paper, we take a step toward a more unified reasoning framework. Our contribution is a novel type system that can *express and verify both universal and existential behavior of the program*. Following the recent trend [Kobayashi et al. 2011; Koskinen and Terauchi 2014; Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009; Unno et al. 2013; Vazou et al. 2014; Zhu and Jagannathan 2013; Zhu et al. 2015], we formalize our approach as a *dependent refinement type system*. A dependent refinement type system allows types to embed predicates on program values, thereby allowing precise type-based value-and-path-sensitive reasoning.

To support both modes of non-determinism, we qualify the types with "$\forall$" and "$\exists$" at appropriate places. Specifically, each type is qualified with one of the four modes Q_1Q_2 where Q_i is either $\forall$ or $\exists$ (for $i \in \{1, 2\}$). Roughly, Q_1 specifies whether the fact expressed by the type holds for every evaluation of the expression being typed ($Q_1 = \forall$) or there exists an evaluation for which the fact holds ($Q_1 = \exists$), and Q_2 says whether the fact holds for every value obtained as the result of the evaluation ($Q_2 = \forall$) or there exists a final value for which the fact holds ($Q_2 = \exists$). For example, $\{u{:}\mathtt{int} \mid u > 0\}^{\forall\forall}$ is the type of expressions satisfying the property that every evaluation of the expression either diverges or reduces to a positive integer value, whereas $\{u{:}\mathtt{int} \mid u > 0\}^{\forall\exists}$ is the type of expressions that always terminate and reduce to a positive integer value, and $\{u{:}\mathtt{int} \mid u > 0\}^{\exists\exists}$ is the type of expressions such that there exists an evaluation of the expression that reduces to a positive value. To also cope with non-determinism from program inputs (i.e., whether the program should behave in a certain way for *every* input or for *some* input), we further qualify the bindings in the type environment and the arguments of function types by $\forall$ or $\exists$. For example, $(x{:}^{\forall}\,\mathtt{int}) \to \{u{:}\mathtt{int} \mid u > x\}^{\forall\exists}$ is the type of functions that, given any integer argument x, always returns some integer u greater than x. Likewise, $(x{:}^{\exists}\,\mathtt{int}) \to \{u{:}\mathtt{int} \mid u > x\}^{\forall\exists}$ is the type of functions where there exists an integer argument x such that every evaluation of the function with the argument returns some integer u greater than x. The modes exhibit a natural form of *duality*, that is, $\overline{\forall} = \exists$, $\overline{\exists} = \forall$ and $\overline{Q_1 Q_2} = (\overline{Q_1})(\overline{Q_2})$.[1]

The types enable a seamless combination of *both universal and existential reasoning within a single deduction system*. To our knowledge, such a combination has not been fully explored in the program verification literature, especially in the presence of higher-order functions (cf. Section 7 for further discussion). We briefly demonstrate the power of the combined reasoning by examples in Section 2. We defer a more detailed exposition to the later sections of the paper.

We show that our type system enjoys a number of desirable meta-theoretic properties. First, thanks to the duality property, the types are closed under complement. More precisely, for any type σ, there is a (syntactically definable) *complement type* $\neg\sigma$ that satisfies $[\![\sigma]\!] \cap [\![\neg\sigma]\!] = \emptyset$ and $[\![\sigma]\!] \cup [\![\neg\sigma]\!] = [\![T]\!]$ where T is the simple type that σ and $\neg\sigma$ refine. Here, $[\![\sigma]\!]$ denotes the semantics of the type σ and is, roughly, the set of expressions that behave according to σ (cf. Section 5 for details). As an example, let $\sigma_{id}^{pc} = (x{:}^{\forall}\,\texttt{int}) \rightarrow \{u{:}\texttt{int} \mid u = x\}^{\forall\forall}$. Then, $[\![\sigma_{id}^{pc}]\!]$ is the set of functions that takes an integer argument and either diverges or returns the given integer. The complement $\neg\sigma_{id}^{pc}$ is $(x{:}^{\exists}\,\texttt{int}) \rightarrow \{u{:}\texttt{int} \mid u \neq x\}^{\exists\exists}$, and $[\![\neg\sigma_{id}^{pc}]\!]$ is the set of functions where there exists an integer argument and an evaluation of the function that returns a different integer value from the given argument. Indeed, $[\![\sigma_{id}^{pc}]\!] \cap [\![\neg\sigma_{id}^{pc}]\!] = \emptyset$ and $[\![\sigma_{id}^{pc}]\!] \cup [\![\neg\sigma_{id}^{pc}]\!] = [\![\texttt{int} \rightarrow \texttt{int}]\!]$, where $[\![\texttt{int} \rightarrow \texttt{int}]\!]$ is the set of partial (non-deterministic) functions from integers to integers.[2]

We also show that our type system is sound and relatively complete. Soundness says that if the type judgement $\Gamma \vdash e : \sigma$ is derivable, then $e \in [\![\Gamma \vdash \sigma]\!]$, where $[\![\Gamma \vdash \sigma]\!]$ is the semantics of the type σ under the type environment Γ (cf. Theorem 6.12). Conversely, completeness says that $e \in [\![\Gamma \vdash \sigma]\!]$ implies that $\Gamma \vdash e : \sigma$ is derivable (cf. Theorem 6.25). As expected, the completeness result is relative to the assumption that the background theory for refinement predicates is sufficiently expressible to encode arbitrary functions definable in the target programming language [Damm and Josko 1983; German et al. 1983, 1989; Goerdt 1985; Honda et al. 2006; Olderog 1984; Reus and Streicher 2011; Unno et al. 2013]. Additionally, we need to assume that the background theory is sufficiently expressible to handle termination of non-deterministic programs. As usual for type systems of this kind, soundness is an imperative requirement that ensures the veracity of the verification result, while completeness says that, at least in theory, the verification system is as precise as possible.

We summarize the main contributions of the paper below.

- We present a novel type system for verification of non-deterministic higher-order functional programs. The type system facilitates combined universal and existential reasoning, and is able to verify rich classes of properties including (conditional) safety, non-safety, termination, and non-termination.
- We show that the type system enjoys desirable meta-theoretic properties. Specifically, we show that the type system is sound and relatively complete, and that the types are closed under complement.

The rest of the paper is organized as follows. In Section 2, we give an informal overview of the type system by examples, focusing on the combined reasoning aspect. Section 3 formalizes the target programming language, and Section 4 and Section 5 formally present the type system. For exposition, we split the presentation in two parts so that we first present a simpler type system that is sound but incomplete in Section 4 and then describe the relatively-complete full type system in Section 5. The simpler system lacks the intricacies that are needed for relative completeness, such as Gödel encoding of function-type values and intersection and union types. However, it is suited for conveying the essence of how universal and existential reasoning can be integrated in

[1]We refer to Section 4 for the formal definition of the types.

[2]Technically, for relative completeness, we need to restrict the domain of functions to the definable ones (cf. Section 6).

$e_{(a)}$	$e_{(b)}$	$e_{(c)}$
`let rec f x y =`	`let rec f x y =`	`let rec app f x =`
`  if x <= y then`	`  if x <= y then`	`  if x > 0 then`
`    0`	`    0`	`    app f (x-1)`
`  else`	`  else`	`  else`
`    f (x-1) y`	`    let w = * in`	`    f x`
`in`	`    if w > x then`	`in`
`let x = 10**9 in`	`      f (x-1) y`	`let rec g x =`
`let y = 0 in`	`    else`	`  if x = 0 then`
`let z = * in`	`      f x y`	`    ()`
`  (f x y) + z`	`in`	`  else`
	`let x = 10**9 in`	`    app g x`
	`let y = 0 in`	`in`
	`let z = * in`	`let b = * in`
	`  (f x y) + z`	`if b then g (-1)`
		`else ()`
$\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$	$\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$	$\{u{:}\mathtt{unit} \mid \bot\}^{\exists\forall}$

Fig. 1. Examples and the corresponding properties

a type system. Section 6 discusses the meta-theoretic properties of the type system. We discuss related work in Section 7, and conclude the paper in Section 8. Omitted proofs are given in the extended version of this paper [Unno et al. 2017].

2 INFORMAL OVERVIEW

We give an informal overview of the type system, with the focus on demonstrating the power of combined universal and existential reasoning.

Example 2.1. Consider the program $e_{(a)}$ shown in Fig. 1, written in an OCaml like syntax. The program calls the function f with the argument x set to 10^9 and y set to 0, and adds a non-deterministic integer value z to the returned result. Here, the expression $*$ evaluates to a non-deterministic integer. We would like to prove that there exists an evaluation of the program that results in the final value of 1. The property is expressed by the type $\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$. Therefore, we would like derive the judgement $\vdash e_{(a)} : \{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$. To do this, we first derive that f has the following type $\sigma_{\mathsf{f}^{\forall\exists}}$.

$$\sigma_{\mathsf{f}^{\forall\exists}} \triangleq (x{:}^{\forall}\,\mathtt{int}) \to (y{:}^{\forall}\,\{u{:}\mathtt{int} \mid u \le x\}) \to \{u{:}\mathtt{int} \mid u = 0\}^{\forall\exists}$$

The type says that the function, given any arguments x and y such that $y \le x$, always terminates and returns 0. Note that this is a *conditional termination property*. As we shall show in more detail later in the paper, such a derivation can be done by using the well-founded relation $x > x' \wedge y = y' \wedge x \ge y$ and the typing rule T-Total (cf. Fig. 5).

Then, by applying the subtyping rule T-Sub (cf. Fig. 5), we derive

$$\frac{\Gamma \vdash \mathsf{f} : \sigma_{\mathsf{f}^{\forall\exists}} \qquad \Gamma \vdash \sigma_{\mathsf{f}^{\forall\exists}} <: \sigma_{\mathsf{f}^{\exists\exists}}}{\Gamma \vdash \mathsf{f} : \sigma_{\mathsf{f}^{\exists\exists}}}\ (\text{T-Sub})$$

where Γ is the type environment at the sub-derivation and $\sigma_{\mathsf{f}^{\exists\exists}}$ is the following type.

$$\sigma_{\mathsf{f}^{\exists\exists}} \triangleq (x{:}^{\forall}\,\mathtt{int}) \to (y{:}^{\forall}\,\{u{:}\mathtt{int} \mid u \le x\}) \to \{u{:}\mathtt{int} \mid u = 0\}^{\exists\exists}$$

The type says that, given any arguments x and y such that $y \leq x$, there is an evaluation of the function that results in the return value of 0. Indeed, the property holds for any function having the type $\sigma_{\mathtt{f}^{\forall\exists}}$. Using $\sigma_{\mathtt{f}^{\exists\exists}}$, we derive the type $\{u{:}\mathtt{int} \mid u = 0\}^{\exists\exists}$ for the function application `f x y` (cf. T-App∀ in Fig. 5). From this and the rule T-Rnd for typing non-deterministic choices, we obtain the type $\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$ for the whole program. The type system handles existential non-deterministic choices by the Skolemization technique (cf. T-Skolem and S-Skolem in Fig. 5). Roughly, the technique synthesizes a predicate that specifies a sufficient condition for the existential bound variable to be used universally in the context, and checks the non-emptiness of the bound type conditioned on the predicate and the current typing context. In the case of this example, we have the existential variable binding $\mathtt{z}{:}^{\exists}\{u{:}\mathtt{int} \mid \top\}$, where $\top$ represents tautology. We synthesize the Skolemization predicate $\phi(\mathtt{z}) \triangleq \mathtt{z} = 1$, and check the non-emptiness of $\exists \mathtt{z}.\top \land \phi(\mathtt{z})$ (which holds trivially). ▲

The example above demonstrates the application of (conditional) termination verification, which proves that for *every* inputs (satisfying a certain condition) the function terminates and reduces to a certain value, to prove the *existence* of an evaluation satisfying a certain property. Importantly, the termination verification only requires a quite simple well-foundedness termination argument. Note that verifying the example by more standard methods such as state space exploration and testing can be much more costly, because such methods would go through one billion recursive function calls in order to discover an evaluation path reaching the program output.

Example 2.2. Next, we consider the program $e_{(b)}$ shown in Fig. 1. The program is a modification of $e_{(a)}$. The function `f` now has an additional control path so that, when $\mathtt{x} > \mathtt{y}$, it non-deterministically chooses an integer value for `w` and calls $\mathtt{f}\ (\mathtt{x} - 1)\ \mathtt{y}$ if $\mathtt{w} > \mathtt{x}$ or calls `f x y` otherwise. The program still satisfies the property $\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$ because, starting from $\mathtt{x} = 10^9$ and $\mathtt{y} = 0$, there exists an evaluation path of the function that leads to the return value of 0. To verify the property, the type system derives the type $\sigma_{\mathtt{f}^{\exists\exists}}$ for the modified `f`.

The derivation of $\sigma_{\mathtt{f}^{\exists\exists}}$ for `f` is similar to that of deriving the type $\sigma_{\mathtt{f}^{\forall\exists}}$ for the unmodified `f` in Example 2.1, and it is done by the application of T-Total. It may use the same well-founded relation $x > x' \land y = y' \land x \geq y$, except that we now must handle the internal non-deterministic choice in the function. The latter is done by T-Rnd and the Skolemization of the existential binding $\mathtt{w}{:}^{\exists}\{u{:}\mathtt{int} \mid \top\}$ with the predicate $\phi(\mathtt{x}, \mathtt{w}) \triangleq \mathtt{w} > \mathtt{x}$. Then, using the type $\sigma_{\mathtt{f}^{\exists\exists}}$ for `f`, we derive the type $\{u{:}\mathtt{int} \mid u = 1\}^{\exists\exists}$ for the whole program in the same way as in Example 2.1. ▲

In Example 2.2, the modified `f` has an internal non-determinism that needs to be properly handled to derive the target property. The example demonstrates the ability of the type system to combine reasoning via well-foundedness termination argument and existential non-determinism. As result, we obtain a succinct proof of the existence of a non-trivial evaluation path.

Example 2.3. Next, we consider the program $e_{(c)}$ shown in Fig. 1, which is adopted from [Kuwahara et al. 2014]. It contains a higher-order recursive function `app` and a recursive function `g`. We would like to verify that the program has the type $\{u{:}\mathtt{unit} \mid \bot\}^{\exists\forall}$. Here, $\bot$ represents contradiction. Therefore, the type specifies that there exists a non-terminating evaluation, that is, it expresses the (unconditional) non-termination property. Indeed, the program satisfies the property because the evaluation of the function application $\mathtt{g}\ n$ with any negative integer n induces the following infinite sequence of function calls: $\mathtt{g}\ n \longrightarrow^{*} \mathtt{app}\ \mathtt{g}\ n \longrightarrow^{*} \mathtt{g}\ n \longrightarrow^{*} \ldots$.

In order to derive the judgement $\vdash e_{(c)} : \{u{:}\mathtt{unit} \mid \bot\}^{\exists\forall}$, we derive the following types σ_{app} and σ_{g} for `app` and `g`, respectively.

$$\begin{aligned}\sigma_{\mathsf{g}} &\triangleq (x{:}^{\forall}\{u{:}\mathtt{int} \mid u < 0\}) \rightarrow \{u{:}\mathtt{unit} \mid \bot\}^{\forall\forall}\\ \sigma_{\mathsf{app}} &\triangleq (f{:}^{\forall}(x{:}^{\forall}\{u{:}\mathtt{int} \mid u \leq 0\}) \rightarrow \{u{:}\mathtt{unit} \mid \bot\}^{\forall\forall}) \rightarrow \{u{:}\mathtt{unit} \mid \bot\}^{\forall\forall}\end{aligned}$$

$$\begin{array}{rrl}
\text{(expressions)} & e ::= & x \mid n \mid v_1\ op\ v_2 \mid \texttt{ifz}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
 & \mid & \texttt{rec}(f,\widetilde{x},e) \mid v_1\ v_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ x = *\ \texttt{in}\ e \\
\text{(values)} & v ::= & x \mid n \mid \texttt{rec}(f,\widetilde{x},e)\ \widetilde{v} \\
\text{(simple types)} & T ::= & \texttt{int} \mid T_1 \to T_2 \\
\text{(simple type environments)} & A ::= & \emptyset \mid A, x : T
\end{array}$$

Fig. 2. The syntax of $\mathcal{L}$.

$$n_1\ op\ n_2 \xrightarrow{\epsilon} [\![op]\!]\,(n_1,n_2) \quad \text{(E-Op)}$$

$$\texttt{ifz}\ 0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\epsilon} e_1 \quad \text{(E-IfZ)}$$

$$\frac{n \neq 0}{\texttt{ifz}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow{\epsilon} e_2} \quad \text{(E-IfNZ)}$$

$$\frac{|\widetilde{x}| = |\widetilde{v}|}{\texttt{rec}(f,\widetilde{x},e)\ \widetilde{v} \xrightarrow{\epsilon} [\texttt{rec}(f,\widetilde{x},e)/f,\widetilde{v}/\widetilde{x}]e} \quad \text{(E-App)}$$

$$\texttt{let}\ x = *\ \texttt{in}\ e \xrightarrow{n} [n/x]e \quad \text{(E-Rnd)}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \xrightarrow{\epsilon} [v/x]e \quad \text{(E-LetV)}$$

$$\frac{e_1 \xrightarrow{\widetilde{n}} e_1'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \xrightarrow{\widetilde{n}} \texttt{let}\ x = e_1'\texttt{in}\ e_2} \quad \text{(E-Let)}$$

$$e \overset{\epsilon}{\Longrightarrow} e \quad \text{(E-Refl)}$$

$$\frac{e_1 \xrightarrow{\widetilde{n_1}} e_2 \qquad e_2 \overset{\widetilde{n_2}}{\Longrightarrow} e_3}{e_1 \overset{\widetilde{n_1}\cdot\widetilde{n_2}}{\Longrightarrow} e_3} \quad \text{(E-Trans)}$$

Fig. 3. The operational semantics of $\mathcal{L}$.

The derivation of the function types are done by the applications of the rule T-Partial (cf. Fig. 5). Note that the types express *safety properties*. For instance, σ_g says that the function always diverges given any negative integer argument. Equipped with these types for g and app, we derive the desired type $\{u{:}\texttt{unit} \mid \bot\}^{\exists\forall}$ for the program. This involves applying T-Rnd, and the Skolemization of the existential binding $\texttt{b}{:}^{\exists}\{u{:}\texttt{bool} \mid \top\}$ with the predicate $\phi(\texttt{b}) \triangleq \texttt{b} = \texttt{true}$. ▲

Example 2.3 demonstrates how our type system allows proofs of safety properties to be used for proving non-termination. While such a combination has been observed previously [Chen et al. 2014; Kuwahara et al. 2015], to our knowledge, our work is the first to realize the combination in a unified framework of a type system.

3 PRELIMINARIES

In this section, we introduce a simple higher-order strict functional language with non-determinism, $\mathcal{L}$, which is the target of our refinement type system. Fig. 2 shows the syntax of $\mathcal{L}$. Here, x is a meta-variable ranging over variables, and n represents integer constants. $\widetilde{x}$ represents a sequence of variables. We write ϵ for the empty sequence and $|\widetilde{x}|$ for the length of $\widetilde{x}$. For simplicity, our language has only integers as a base type. The symbol op ranges over binary integer operators and is either the integer addition $+$ or the integer multiplication $\times$. An expression $\texttt{rec}(f,\widetilde{x},e)$ represents a (possibly recursive) function f with arguments $\widetilde{x}$ and a body e. We here assume that $|\widetilde{x}| \neq 0$. An expression $\texttt{let}\ x = *\ \texttt{in}\ e$ generates a random integer n and evaluates e with x bound to n, and is the only source of (internal) non-determinism. The value $\texttt{rec}(f,\widetilde{x},e)\ \widetilde{v}$ represents a function closure where $|\widetilde{v}| < |\widetilde{x}|$. We write $fvs(e)$ for the set of free variables that occur in e. We say that e is *closed* if $fvs(e) = \emptyset$.

$$\begin{array}{rrcl}
\text{(qualified types)} & \sigma & ::= & \phi \rhd \tau^{Q_1 Q_2} \\
\text{(dependent refinement types)} & \tau & ::= & \{x \mid \phi\} \mid (x{:}^{Q}\,\tau) \rightarrow \sigma \\
\text{(type environments)} & \Gamma & ::= & \emptyset \mid \Gamma,\ x{:}^{Q}\,\tau
\end{array}$$

Fig. 4. The syntax of refinement types.

We assume that the expressions are simply-typed. Also, to simplify the proof of relative completeness (cf. Section 6.3), we assume that the sub-expressions e_1 and e_2 of $\mathtt{ifz}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$, $\mathtt{rec}(f, \widetilde{x}, e_1)$, $\mathtt{let}\ x = e\ \mathtt{in}\ e_1$, and $\mathtt{let}\ x = *\ \mathtt{in}\ e_1$ only evaluate to integers. This does not lose generality because an arbitrary program can be converted to such a form by, for example, the continuation-passing style (CPS) transformation.[3]

Fig. 3 shows the call-by-value operational semantics of the language. Here, $e \xrightarrow{\widetilde{n}} e'$ means that e is reduced to e' in one step with $\widetilde{n} = n$ if an integer n is randomly generated during the reduction and $\widetilde{n} = \epsilon$ if the reduction is deterministic. The multi-step reduction relation $e \overset{\widetilde{n}}{\Longrightarrow} e'$ means that e is reduced to e' in zero or more number of steps and $\widetilde{n}$ is the sequence of integers randomly generated during the reduction.

The evaluation rules are mostly self-explanatory. We comment on the less standard rules. In E-Rnd, the expression $\mathtt{let}\ x = *\ \mathtt{in}\ e$ is reduced to $[n/x]\,e$ for a randomly generated integer n. In E-Op, $[\![+]\!]$ and $[\![\times]\!]$ respectively represent the integer addition and multiplication. The evaluation relation satisfies the following property.

Proposition 3.1. *If* $e \overset{\widetilde{n}}{\Longrightarrow} e_1$ *and* $e \overset{\widetilde{n}}{\Longrightarrow} e_2$*, then* $e_1 \overset{\epsilon}{\Longrightarrow} e_2$ *or* $e_2 \overset{\epsilon}{\Longrightarrow} e_1$.

4 REFINEMENT TYPE SYSTEM

We present our refinement type system. For exposition, we present a simpler type system that is sound but incomplete in this section, and then describe the relatively-complete full type system in Section 5. The simpler system lacks the intricacy that are needed for relative completeness, such as Gödel encoding of function-type values and intersection and union types. However, it is suited for conveying the essence of how universal and existential reasoning can be integrated in a type system.

Fig. 4 shows the syntax of refinement types. Here, Q is either $\forall$ or $\exists$. A refinement base type $\{x \mid \phi\}$, equipped with a refinement predicate ϕ, represents the type of integers x that satisfy ϕ. Here, ϕ is an integer arithmetic formula.[4] We write $\top$ and $\bot$ respectively for tautology and contradiction. We write $\models \phi$ when ϕ is valid. We often abbreviate $\{x \mid \phi\}$ as $\mathtt{int}$ when ϕ is valid (e.g., $\{x \mid \top\} = \mathtt{int}$). The type $(x{:}^{\forall}\,\tau) \rightarrow \sigma$ is a *dependent function type*, consisting of the argument type τ and the return type σ. It represents a type of functions that, given *any* argument x of the type τ, behave according to the type σ. By contrast, the dependent function type $(x{:}^{\exists}\,\tau) \rightarrow \sigma$ with the existentially bound argument is the type of functions that, given *some* argument x of the type τ, behave according to σ. We sometimes abbreviate $(x{:}^{\forall}\,\tau) \rightarrow \sigma$ as $\tau \rightarrow \sigma$ if x does not occur in σ. In $(x{:}^{\forall}\,\tau) \rightarrow \sigma$ and $(x{:}^{\exists}\,\tau) \rightarrow \sigma$, the variable x is interpreted as *integers* in the refinement predicates in σ, even when the argument type τ is a function type. In the latter, the passed functions are assumed to be encoded as integers (cf. Section 5.1). We let σ range over types qualified by the $\forall$-$\exists$

[3] An analogous assumption is used in the previous work on higher-order program verification [Kobayashi 2009; Terauchi 2010; Unno et al. 2013].

[4] For relative completeness, we require that the background theory of refinement predicates is second-order arithmetic, but any subset (e.g., quantifier-free first-order theory of linear arithmetic) is sufficient for soundness.

modes. A qualified type is also equipped with a logical formula ϕ, which we call a *guard*. We often abbreviate $\phi \rhd \tau^{Q_1Q_2}$ as $\tau^{Q_1Q_2}$ when ϕ is $\top$. Later in Section 5, we extend σ to range over *guarded intersection and union types*. For this section, it is safe to assume that guard formulas are always $\top$, except for the return type of the function at the typing rule T-Total where the formula is used to express the well-foundedness condition (cf. Section 4.1). We often write $ty(\tau)$ (resp. $ty(\sigma)$) for the simple type obtained from τ (resp. σ) by removing refinement predicates, qualifiers, guards, intersections, and unions.

We write $fvs(\tau)$ and $fvs(\sigma)$ for the set of free variables of τ and σ respectively, which are defined inductively as shown below.

$$\begin{array}{rcl} fvs(\{x \mid \phi\}) & \triangleq & fvs(\phi) \setminus \{x\} \\ fvs((x :^{Q} \tau) \to \sigma) & \triangleq & fvs(\tau) \cup (fvs(\sigma) \setminus \{x\}) \\ fvs(\phi \rhd \tau^{Q_1Q_2}) & \triangleq & fvs(\phi) \cup fvs(\tau) \end{array}$$

Note that the scope of x in $\{x \mid \phi\}$ is ϕ, and the scope of x in $(x :^{Q} \tau) \to \sigma$ is σ (but not τ).

We define the order $\sqsubseteq$ among the qualifiers as follows: $Q\exists \sqsubseteq Q\forall$ and $\forall Q \sqsubseteq \exists Q$ for any Q. Note that $(\{\forall\forall, \forall\exists, \exists\forall, \exists\exists\}, \sqsubseteq)$ is a lattice. The order is used to formalize the subtyping relationship (cf. Section 4.1). The intuition behind the order can be understood as follows: total correctness types can be used as partial correctness types and demonic types can be used as angelic types. Note that $\forall\exists$ is the strongest element, and we often abbreviate $\tau^{\forall\exists}$ as τ.

We remark that a value can always be assigned the qualifier $\forall\exists$ because it is *pure*, i.e., it never causes non-termination or non-determinism. Note also that, because our target language $\mathcal{L}$ is strict, an argument passed to a function is always a value. Therefore, they too can be safely assumed to be qualified by $\forall\exists$. This is why the function argument types and the types bound in type environments do not carry qualifiers (i.e., they are of the form $x :^{Q} \tau$ rather than $x :^{Q} \sigma$). Also, a partial application $\mathtt{rec}(f, \widetilde{x}, e)\ \widetilde{v}$ (i.e., where $|\widetilde{v}| < |\widetilde{x}|$) is a value and is pure. Then, because function bodies only evaluate to integers, each function can be assigned a type of the form $(x_1 :^{Q_1} \tau_1) \to (\cdots \to ((x_m :^{Q_m} \tau_m) \to \sigma)^{\forall\exists} \cdots)^{\forall\exists}$ where σ is a (qualified and guarded) refinement base type. We often abbreviate the type as $(\widetilde{x} :^{\widetilde{Q}} \widetilde{\tau}) \to \sigma$, where $\widetilde{x} = x_1, \cdots, x_m$, $\widetilde{\tau} = \tau_1, \cdots, \tau_m$, and $\widetilde{Q} = Q_1, \cdots, Q_m$. In what follows, we assume that τ in $\phi \rhd \tau^{Q_1Q_2}$ is of the form $\{x \mid \phi\}$.

As usual, we assume that the variables bound in a type environment are distinct. We write Γ, ϕ for $\Gamma, \nu :^{\forall} \{\nu \mid \phi\}$ where ν is fresh. We define $\mathrm{dom}(\Gamma) = \{x \mid x :^{Q} \tau \in \Gamma\}$, and write $\Gamma(x) = \tau$ if $x :^{Q} \tau \in \Gamma$. We also use Δ as a meta-variable ranging over type environments that do not contain an existential binding $x :^{\exists} \tau$ (and use Γ to range over type environments that contain or do not contain existential bindings).

4.1 Typing Rules

Fig. 5 shows the typing and subtyping rules. A typing judgement $\Gamma \vdash e : \sigma$ means that an expression e has a type σ under a type environment Γ. A subtyping judgement $\Gamma \vdash \sigma_1 <: \sigma_2$ means that σ_1 is a subtype of σ_2 under Γ.

In Fig. 6, the auxiliary functions $\lfloor \Gamma \vdash \phi \rfloor$, $\lfloor \vdash x : \tau \rfloor$ and $\lfloor \vdash x : \sigma \rfloor$ return an arithmetic formula. Intuitively, $\lfloor \Gamma \vdash \phi \rfloor$ holds if ϕ is valid for any valuation of the variables conforming to Γ whereas $\lfloor \vdash x : \tau \rfloor$ (resp. $\lfloor \vdash x : \sigma \rfloor$) represents the condition for x to satisfy the property denoted by τ (resp. σ). Roughly, these functions provide Gödel encodings of the denotational semantics of the types (cf. Section 5.2). The formal definition of the notations $\llbracket e \rrbracket$, $x \Downarrow$, $\mathbf{ev}(x, y)$, $\mathbf{val}_T(x)$, $\mathbf{exp}_T(x)$, and $\mathbf{app}(x, y)$ are deferred to Section 5.1 ($\llbracket e \rrbracket$ is used in the definition of the auxiliary function $\lfloor v \rfloor$ in Fig. 6). Roughly, $\llbracket e \rrbracket$ is the encoding function for expressions, $x \Downarrow$ is the predicate which states that the expression encoded by x terminates, $\mathbf{ev}(x, y)$ says that the expression encoded by x evaluates

$$\frac{\begin{array}{c}\tau_f = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \tau^{Q\forall} \\ \Delta, \widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, f{:}^{\forall}\tau_f \vdash e : \tau^{Q\forall}\end{array}}{\Delta \vdash \mathtt{rec}(f, \widetilde{x}, e) : \tau_f} \quad \text{(T-Partial)}$$

$$\frac{\begin{array}{c}\tau'_f = (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau'}) \to \phi \rhd (\tau')^{Q\exists} \\ \tau_f = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \tau^{Q\exists} =_{\alpha} (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau'}) \to (\tau')^{Q\exists} \\ \{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset \quad fvs(\phi) \subseteq (\{\widetilde{x}, \widetilde{y}\} \cup \mathrm{dom}(\Delta)) \\ \models \lfloor \Delta \vdash WF(\lambda\widetilde{x}\widetilde{y}.\phi) \rfloor \quad \Delta, \widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, f{:}^{\forall}\tau'_f \vdash e : \tau^{Q\exists}\end{array}}{\Delta \vdash \mathtt{rec}(f, \widetilde{x}, e) : \tau_f} \quad \text{(T-Total)}$$

$$\frac{\Delta \vdash v_1 : (x{:}^{\forall}\tau) \to \sigma \qquad \Delta \vdash v_2 : \tau}{\Delta \vdash v_1\ v_2 : [\lfloor v_2 \rfloor / x]\sigma} \quad \text{(T-App}\forall\text{)}$$

$$\frac{\begin{array}{c}\Delta \vdash v_1 : (x{:}^{\exists}\tau) \to \sigma \\ \models \lfloor \Delta, x{:}^{\forall}\tau, \Gamma \vdash x = \lfloor v_2 \rfloor \rfloor\end{array}}{\Delta, \Gamma \vdash v_1\ v_2 : [\lfloor v_2 \rfloor / x]\sigma} \quad \text{(T-App}\exists\text{)}$$

$$\frac{ty(\Delta(x)) = \mathtt{int}}{\Delta \vdash x : \{\nu \mid \nu = x\}} \quad \text{(T-VInt)}$$

$$\frac{ty(\Delta(x)) \neq \mathtt{int}}{\Delta \vdash x : \Delta(x)} \quad \text{(T-VFun)}$$

$$\Delta \vdash n : \{x \mid x = n\} \quad \text{(T-Int)}$$

$$\frac{\Delta \vdash v_1 : \mathtt{int} \qquad \Delta \vdash v_2 : \mathtt{int}}{\Delta \vdash v_1\ op\ v_2 : \{x \mid x = (v_1\ op\ v_2)\}} \quad \text{(T-Op)}$$

$$\frac{\begin{array}{c}\Delta \vdash v : \mathtt{int} \\ \Delta, v = 0 \vdash e_1 : \sigma \qquad \Delta, v \neq 0 \vdash e_2 : \sigma\end{array}}{\Delta \vdash \mathtt{ifz}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \sigma} \quad \text{(T-If)}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 : \tau_1^{Q_1Q_2} \\ \Delta, x{:}^{\forall}\tau_1 \vdash e_2 : \tau_2^{Q_1Q_2} \qquad x \notin fvs(\tau_2)\end{array}}{\Delta \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2^{Q_1Q_2}} \quad \text{(T-Let)}$$

$$\frac{\Delta, x{:}^{Q_1}\mathtt{int} \vdash e : \tau^{Q_1Q_2} \qquad x \notin fvs(\tau)}{\Delta \vdash \mathtt{let}\ x = *\ \mathtt{in}\ e : \tau^{Q_1Q_2}} \quad \text{(T-Rnd)}$$

$$\frac{\begin{array}{c}\models \lfloor \Delta, x{:}^{\exists}\tau \vdash \phi \rfloor \\ \Delta, x{:}^{\forall}\tau, \phi, \Gamma \vdash e : \sigma\end{array}}{\Delta, x{:}^{\exists}\tau, \Gamma \vdash e : \sigma} \quad \text{(T-Skolem)}$$

$$\frac{\Delta \vdash e : \sigma' \qquad \Delta \vdash \sigma' <: \sigma}{\Delta \vdash e : \sigma} \quad \text{(T-Sub)}$$

$$\frac{\models \lfloor \Delta \vdash \phi_2 \Rightarrow \phi_1 \rfloor \quad \Delta, \phi_2 \vdash \sigma_1 <: \sigma_2}{\Delta \vdash \phi_1 \rhd \sigma_1 <: \phi_2 \rhd \sigma_2} \quad \text{(S-Guard)}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad Q_1Q'_1 \sqsubseteq Q_2Q'_2}{\Delta \vdash \tau_1^{Q_1Q'_1} <: \tau_2^{Q_2Q'_2}} \quad \text{(S-Qual)}$$

$$\frac{\models \lfloor \Delta \vdash \phi_1 \Rightarrow \phi_2 \rfloor}{\Delta \vdash \{\nu \mid \phi_1\} <: \{\nu \mid \phi_2\}} \quad \text{(S-Int)}$$

$$\frac{\Delta \vdash \tau_2 <: \tau_1 \qquad \Delta, x{:}^{\forall}\tau_2 \vdash \sigma_1 <: \sigma_2}{\Delta \vdash (x{:}^{\forall}\tau_1) \to \sigma_1 <: (x{:}^{\forall}\tau_2) \to \sigma_2} \quad \text{(S-Fun}\forall\forall\text{)}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau <: \tau_1 \qquad \Delta \vdash \tau <: \tau_2 \\ \Delta, x{:}^{\exists}\tau \vdash \sigma_1 <: \sigma_2\end{array}}{\Delta \vdash (x{:}^{\forall}\tau_1) \to \sigma_1 <: (x{:}^{\exists}\tau_2) \to \sigma_2} \quad \text{(S-Fun}\forall\exists\text{)}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta, x{:}^{\forall}\tau_1 \vdash \sigma_1 <: \sigma_2}{\Delta \vdash (x{:}^{\exists}\tau_1) \to \sigma_1 <: (x{:}^{\exists}\tau_2) \to \sigma_2} \quad \text{(S-Fun}\exists\exists\text{)}$$

$$\frac{\begin{array}{c}\Delta' = \Delta, x_1{:}^{\forall}\tau_1, x_2{:}^{\forall}\tau_2, y_1{:}^{\forall}\tau_1, y_2{:}^{\forall}\tau_2 \\ \models \lfloor \Delta' \vdash x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 = y_1 \rfloor \\ \Delta, x_1{:}^{\forall}\tau_1, x_2{:}^{\forall}\tau_2, x_1 = x_2 \vdash \sigma_1 <: \sigma_2 \\ \sigma_{\phi,Q} = \widetilde{T} \to \{\nu \mid \phi\}^{QQ} \\ \Delta, x_1{:}^{\forall}\tau_1, x_2{:}^{\exists}\tau_2, x_1 \neq x_2 \vdash \sigma_1 <: \sigma_{\bot,\exists} \\ \Delta, x_2{:}^{\forall}\tau_2, x_1{:}^{\exists}\tau_1, x_1 \neq x_2 \vdash \sigma_{\top,\forall} <: \sigma_2\end{array}}{\Delta \vdash (x_1{:}^{\exists}\tau_1) \to \sigma_1 <: (x_2{:}^{\forall}\tau_2) \to \sigma_2} \quad \text{(S-Fun}\exists\forall\text{)}$$

$$\frac{\begin{array}{c}\models \lfloor \Delta, x{:}^{\exists}\tau \vdash \phi \rfloor \\ \Delta, x{:}^{\forall}\tau, \phi, \Gamma \vdash \tau_1 <: \tau_2\end{array}}{\Delta, x{:}^{\exists}\tau, \Gamma \vdash \tau_1 <: \tau_2} \quad \text{(S-Skolem)}$$

Fig. 5. The derivation rules of $\Gamma \vdash e : \sigma$ and $\Gamma \vdash \sigma_1 <: \sigma_2$ (except those for intersections and unions).

to the value encoded by y, $\mathbf{val}_T(x)$ (resp. $\mathbf{exp}_T(x)$) represents that x encodes an $\mathcal{L}$-definable value (resp. expression) of the simple type T, and $\mathbf{app}(x, y)$ returns the integer that encodes the application

$$\begin{array}{rclrcl}
\lfloor \emptyset \vdash \phi \rfloor &=& \phi & \lfloor \vdash x : (\tau, \forall\exists) \rfloor &=& x\Downarrow \land \left\lfloor \vdash x : \tau^{\forall\forall} \right\rfloor \\
\left\lfloor \Gamma, x{:}^{\forall}\tau \vdash \phi \right\rfloor &=& \lfloor \Gamma \vdash \forall x.(\lfloor \vdash x : \tau \rfloor \Rightarrow \phi) \rfloor & \lfloor \vdash x : (\tau, \forall\forall) \rfloor &=& \forall y.(\mathbf{ev}(x, y) \Rightarrow \lfloor \vdash y : \tau \rfloor) \\
\left\lfloor \Gamma, x{:}^{\exists}\tau \vdash \phi \right\rfloor &=& \lfloor \Gamma \vdash \exists x.(\lfloor \vdash x : \tau \rfloor \land \phi) \rfloor & \lfloor \vdash x : (\tau, \exists\exists) \rfloor &=& \exists y.(\mathbf{ev}(x, y) \land \lfloor \vdash y : \tau \rfloor) \\
\lfloor \vdash x : \{\nu \mid \phi\} \rfloor &=& [x/\nu]\phi & \lfloor \vdash x : (\tau, \exists\forall) \rfloor &=& \neg(x\Downarrow) \lor \left\lfloor \vdash x : \tau^{\exists\exists} \right\rfloor
\end{array}$$

$$\left\lfloor \vdash x : \left(\left(\nu{:}^{Q}\tau\right) \to \sigma\right) \right\rfloor = \mathbf{val}_{ty(\tau\to\sigma)}(x) \ \land \left\lfloor \nu{:}^{Q}\tau \vdash \forall r.(r = \mathbf{app}(x, \nu) \Rightarrow \lfloor \vdash r : \sigma \rfloor) \right\rfloor$$

$$\left\lfloor \vdash x : \left(\phi \rhd \tau^{Q_1Q_2}\right) \right\rfloor = \mathbf{exp}_{ty(\tau)}(x) \land (\phi \Rightarrow \lfloor \vdash x : (\tau, Q_1Q_2) \rfloor)$$

$$\lfloor v \rfloor = \begin{cases} v & \text{(if } v \text{ is an integer value)} \\ \llbracket v \rrbracket & \text{(otherwise)} \end{cases}$$

Fig. 6. The auxiliary functions for the typing rules.

of the function expression encoded by x to the value encoded by y. Note that the encoding function $\lfloor v \rfloor$ for values does nothing if v is an integer value.

We describe the typing rules. The rule T-Partial (resp. T-Total) assigns a partial (resp. total) correctness type to a recursive function f. T-Partial is the standard rule for typing recursive functions, and as expected, it ensures partial correctness. By contrast, T-Total requires that the pair of the arguments $\widetilde{x}$ passed to a call to f and those $\widetilde{y}$ passed to its recursive call is included in a well-founded relation in order to guarantee termination. We call a predicate $p = \lambda\widetilde{x}.\phi$ *well-founded* and write $WF(p)$, if the arity of p is $2 \times n$ for some n and there is no infinite sequence $\widetilde{t_1}, \widetilde{t_2}, \ldots$ such that $|\widetilde{t_i}| = n$ and $p(\widetilde{t_i}, \widetilde{t_{i+1}})$ holds for all $i \geq 1$. Also, we write $\tau_1 =_\alpha \tau_2$ if τ_1 and τ_2 are alpha equivalent. Let us consider the following expression as an example:

$$e_{\mathsf{sum}} \triangleq \mathtt{rec}(\mathsf{sum}, x, \mathtt{ifz}\ x\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{let}\ r = \mathsf{sum}\ (x-1)\ \mathtt{in}\ x + r)$$

We can derive the judgement $\vdash e_{\mathsf{sum}} : (x : \{x \mid x \geq 0\}) \to \{y \mid y \geq x\}^{\forall\exists}$ because we can derive $\Gamma_{\mathsf{sum}} \vdash \mathtt{let}\ r = \mathsf{sum}\ (x-1)\ \mathtt{in}\ x + r : \{y \mid y \geq x\}^{\forall\exists}$ for

$$\Gamma_{\mathsf{sum}} \triangleq x{:}^{\forall}\{x \mid x \geq 0\}, \mathsf{sum}{:}^{\forall}(x' : \{x' \mid x' \geq 0\}) \to x > x' \geq 0 \rhd \{y' \mid y' \geq x'\}^{\forall\exists}, x \neq 0$$

and $\models WF(\lambda(x, x').x > x' \geq 0)$. The rule is analogous to those proposed previously for asserting termination in dependent-refinement type systems [Vazou et al. 2014; Xi 2001].

In the rule T-Let, the qualifiers of the types of the consecutively executed expressions e_1 and e_2 must coincide, and the variable x which stores the value of e_1 is always bound universally. By contrast, in the rule T-Rnd, the variable x which stores the randomly generated integer is allowed to be bound existentially, assuming that the qualifier of the type of the body e is of the form $\exists Q$. For example, the derivation of $x{:}^{\forall}\mathtt{int} \vdash \mathtt{let}\ y\ =\ *\ \mathtt{in}\ x + y : \{\nu \mid \nu = 0\}^{\exists\exists}$ has the following root.

$$\frac{x{:}^{\forall}\mathtt{int}, y{:}^{\exists}\mathtt{int} \vdash x + y : \{\nu \mid \nu = 0\}^{\exists\exists}}{x{:}^{\forall}\mathtt{int} \vdash \mathtt{let}\ y\ =\ *\ \mathtt{in}\ x + y : \{\nu \mid \nu = 0\}^{\exists\exists}}$$

As we shall show below, the antecedent $x :^{\forall} \mathtt{int}, x :^{\exists} \mathtt{int} \vdash x + y : \{\nu \mid \nu = 0\}^{\exists\exists}$ can be derived by applying the T-Skolem rule.

T-App∀ and T-App∃ are rules for function applications $v_1\ v_2$. T-App∀ is the standard rule for function applications in dependent refinement type systems (see, e.g., [Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009]). Recall that the notation $\lfloor v \rfloor$ denotes the encoding of the value v. By contrast, T-App∃ is a non-standard rule introduced for handling function types $(x{:}^{\exists}\tau) \to \sigma$ with the existential binding. For the sake of exposition, assume here that Γ only contains existential bindings. Then, roughly, the antecedent $\models \left\lfloor \Delta, x{:}^{\forall}\tau, \Gamma \vdash x = \lfloor v_2 \rfloor \right\rfloor$ says that for any valuation conforming to the type environment $\Delta, x :^{\forall} \tau$, we can select some valuation of the existentially bound variables in Γ such that $\lfloor v_2 \rfloor$ becomes equivalent to the valuation of x. That is, it checks if every value of τ can be obtained by selecting the values of existentially bound variables in Γ. If the check succeeds, the rule concludes that $v_1\ v_2$ has the type $[\lfloor v_2 \rfloor / x]\sigma$. For example, let $e \triangleq \texttt{let}\ y\ =\ *\ \texttt{in}\ f\ y$ and $\Delta \triangleq f :^{\forall} (x :^{\exists} \texttt{int}) \to \{z \mid \bot\}^{\exists\forall}$. The type of f says that there is an integer n such that $f\ n$ may diverge. By T-Rnd and T-App∃, we have

$$\dfrac{\dfrac{\Delta \vdash f : (x :^{\exists} \texttt{int}) \to \{z \mid \bot\}^{\exists\forall} \quad \models \lfloor \Delta, x :^{\forall} \texttt{int}, y :^{\exists} \texttt{int} \vdash x = y \rfloor}{\Delta, y :^{\exists} \texttt{int} \vdash f\ y : \{z \mid \bot\}^{\exists\forall}}}{\Delta \vdash e : \{z \mid \bot\}^{\exists\forall}}$$

Because $\models \lfloor \Delta, x :^{\forall} \texttt{int}, y :^{\exists} \texttt{int} \vdash x = y \rfloor$ holds, we can conclude that e also has the type $\{z \mid \bot\}^{\exists\forall}$, and therefore may diverge. Note that if the type of f were $(x :^{\exists} \texttt{int}) \to \{z \mid \bot\}^{\forall\forall}$ instead, T-Rnd would not allow e to have the incorrect type $\{z \mid \bot\}^{\forall\forall}$ saying that e always diverges.

The rules T-Skolem and S-Skolem skolemize an existentially bound variable $x :^{\exists} \tau$ in the type environment, which may be introduced by the rules T-Partial, T-Total, and T-Rnd. As remarked in Section 2, the rules introduce a *Skolemization predicate* ϕ that specifies a sufficient condition for the existentially bound variable to be used universally in the context (which is expressed by the antecedents $\Delta, x :^{\forall} \tau, \phi, \Gamma \vdash e : \sigma$ and $\Delta, x :^{\forall} \tau, \phi, \Gamma \vdash \tau_1 <: \tau_2$ respectively), and check the non-emptiness of the bound type conditioned on the predicate and the current typing context (expressed by the antecedent $\models \lfloor \Delta, x :^{\exists} \tau \vdash \phi \rfloor$). For example, $x :^{\forall} \texttt{int}, y :^{\exists} \texttt{int} \vdash x + y : \{z \mid z = 0\}^{\exists\exists}$ from the above discussion is derivable by using T-Skolem because $\models \lfloor x :^{\forall} \texttt{int}, y :^{\exists} \texttt{int} \vdash y = -x \rfloor$ and $x :^{\forall} \texttt{int}, y :^{\forall} \texttt{int}, y = -x \vdash x + y : \{z \mid z = 0\}^{\exists\exists}$ are derivable. Note that we need to apply the Skolemization rules eagerly before we apply the other rules because they require the type environment Δ to contain no existential binding.[5]

The rule T-Sub weakens a type assigned to an expression into a subtype. The rule S-Guard can be used to eliminate the guard of types introduced by T-Total. The rule S-Qual changes the qualifier of types according to the order $\sqsubseteq$. The rules S-Int and S-Fun∀∀ are standard and adopted from the previous work on dependent refinement types for partial correctness [Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009]. The subtyping rules S-Fun∀∃, S-Fun∃∃, and S-Fun∃∀ are for deriving subtyping judgements related to function types that handle function types $(x :^{\exists} \tau) \to \sigma$ with the existential binding. In the rule S-Fun∃∀, the antecedent $\models \lfloor \Delta' \vdash x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 = y_1 \rfloor$ checks that the intersection of the sets of values represented by τ_1 and τ_2 is empty or singleton, and $\sigma_{\bot,\exists}$ (resp. $\sigma_{\top,\forall}$) denotes the empty set (resp. the set of all the expressions whose simple type is $ty(\sigma_2)$). For example, we can derive $\vdash (x :^{\exists} \{x \mid x \geq 0\}) \to \{y \mid y = x\}^{\forall\exists} <: (x :^{\exists} \texttt{int}) \to \{y \mid y \geq 0\}^{\forall\exists}$ by S-Fun∃∃, $\vdash (x :^{\forall} \{x \mid x \geq 0\}) \to \{y \mid y = x\}^{\forall\exists} <: (x :^{\exists} \{x \mid x \leq 1\}) \to \{y \mid y = 1\}^{\forall\exists}$ by S-Fun∀∃, and $\vdash (x :^{\exists} \{x \mid x = 1\}) \to \{y \mid y = 0\}^{\forall\forall} <: (x :^{\forall} \{x \mid x = 1 \vee x = 2\}) \to \{y \mid x = 1 \Rightarrow y = 0\}^{\forall\forall}$ by S-Fun∃∀. These rules are useful for adjusting the types of function arguments and free variables that are provided by external environments.

[5]Though we could design the type system based on a lazy Skolemization approach, the Skolemization rules must be anyway applied before we type-check expressions consisting of multiple sub-expressions so that we can synchronize the valuation of existentially bound variables across the sub-expressions.

4.2 Examples

We show the type derivations of Examples 2.1, 2.2 and 2.3. The examples are desugared as follows.

$$\begin{aligned}
e_{(a)} \triangleq\ & \mathtt{let}\ \mathsf{f} = \mathtt{rec}(\mathsf{f}, (x, y), e^{\mathsf{f}}_{(a)})\ \mathtt{in} \\
& \mathtt{let}\ x = 10{*}{*}9\ \mathtt{in}\ \mathtt{let}\ y = 0\ \mathtt{in}\ \mathtt{let}\ z = *\ \mathtt{in}\ \mathtt{let}\ w = \mathsf{f}\ x\ \mathtt{in}\ \mathtt{let}\ w' = w\ y\ \mathtt{in}\ w' + z
\end{aligned}$$

$$\begin{aligned}
e_{(b)} \triangleq\ & \mathtt{let}\ \mathsf{f} = \mathtt{rec}(\mathsf{f}, (x, y), e^{\mathsf{f}}_{(b)})\ \mathtt{in} \\
& \mathtt{let}\ x = 10{*}{*}9\ \mathtt{in}\ \mathtt{let}\ y = 0\ \mathtt{in}\ \mathtt{let}\ z = *\ \mathtt{in}\ \mathtt{let}\ w = \mathsf{f}\ x\ \mathtt{in}\ \mathtt{let}\ w' = w\ y\ \mathtt{in}\ w' + z
\end{aligned}$$

$$\begin{aligned}
e_{(c)} \triangleq\ & \mathtt{let}\ \mathsf{app} = \mathtt{rec}(\mathsf{app}, \mathsf{f}\ x, e_{\mathsf{app}})\ \mathtt{in}\ \mathtt{let}\ \mathsf{g} = \mathtt{rec}(\mathsf{g}, x, e_{\mathsf{g}})\ \mathtt{in} \\
& \mathtt{let}\ b = *\ \mathtt{in}\ \mathtt{ifz}\ b\ \mathtt{then}\ \mathsf{g}\ (-1)\ \mathtt{else}\ 0
\end{aligned}$$

where $e^{\mathsf{f}}_{(a)}$, $e^{\mathsf{f}}_{(b)}$, e_{app} and e_{g} are the following expressions.

$$e^{\mathsf{f}}_{(a)} \triangleq \mathtt{let}\ v = x \le y\ \mathtt{in}\ \mathtt{ifz}\ v\ \mathtt{then}\ 0\ \mathtt{else}\ (\mathtt{let}\ v = x - 1\ \mathtt{in}\ \mathtt{let}\ \mathsf{f}' = \mathsf{f}\ v\ \mathtt{in}\ \mathsf{f}'\ y)$$

$$\begin{aligned}
e^{\mathsf{f}}_{(b)} \triangleq\ & \mathtt{let}\ v = x \le y\ \mathtt{in} \\
& \mathtt{ifz}\ v\ \mathtt{then}\ 0 \\
& \mathtt{else}\ \mathtt{let}\ w = *\ \mathtt{in}\ \mathtt{let}\ v = w > x\ \mathtt{in} \\
& \qquad \mathtt{ifz}\ v\ \mathtt{then}\ \mathtt{let}\ v = x - 1\ \mathtt{in}\ \mathtt{let}\ \mathsf{f}' = \mathsf{f}\ v\ \mathtt{in}\ \mathsf{f}'\ y\ \mathtt{else}\ \mathtt{let}\ \mathsf{f}' = \mathsf{f}\ x\ \mathtt{in}\ \mathsf{f}'\ y
\end{aligned}$$

$$e_{\mathsf{app}} \triangleq \mathtt{let}\ v = x > 0\ \mathtt{in}\ \mathtt{ifz}\ v\ \mathtt{then}\ \mathsf{app}\ \mathsf{f}\ (x - 1)\ \mathtt{else}\ \mathsf{f}\ x$$

$$e_{\mathsf{g}} \triangleq \mathtt{let}\ v = x = 0\ \mathtt{in}\ \mathtt{ifz}\ v\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{app}\ \mathsf{g}\ x$$

Here, Boolean values are encoded as integers. That is, for integers m, n and $\lhd \in \{<, \le\}$, $m\ [\![\lhd]\!]\ n$ is defined to be the binary operation that returns 0 if $m \lhd n$ and returns 1 otherwise.

$$\dfrac{\begin{array}{c}\dfrac{\dfrac{}{\Gamma_2 \vdash \mathsf{f} : \sigma_{\mathsf{f}^{\forall\exists}}}\,(\textsc{T-VFun}) \quad \dfrac{\cdots \quad \forall\exists \sqsubseteq \exists\exists}{\Gamma_2 \vdash \sigma_{\mathsf{f}^{\forall\exists}} <: \sigma_{\mathsf{f}^{\exists\exists}}}\,(\textsc{S-Qual})}{\Gamma_2 \vdash \mathsf{f} : \sigma_{\mathsf{f}^{\exists\exists}}}\,(\textsc{T-Sub}) \\ \vdots\end{array} \quad \dfrac{\vdots}{\Gamma_2, w : \tau^{\exists\exists}_{u=0} \vdash w + z : \tau^{\exists\exists}_{e_{(a)}}}\,(\textsc{T-Sub})}{\dfrac{\Gamma_1, z{:}^{\forall}\,\mathsf{int}, z = 1 \vdash e'_{(a)} : \tau^{\exists\exists}_{e_{(a)}}}{\Gamma_1, z{:}^{\exists}\,\mathsf{int} \vdash e'_{(a)} : \tau^{\exists\exists}_{e_{(a)}}}\,(\textsc{T-Skolem})}\,(\textsc{T-Let})$$

$$\dfrac{\dfrac{\dfrac{\vdots}{x : \mathsf{int}, y : \mathsf{int}, \mathsf{f} : \tau_{\mathsf{f}} \vdash e^{\mathsf{f}}_{(a)} : \{u \mid u = 0\}^{\forall\exists}} \quad \models \lfloor \vdash \mathit{WF}(p) \rfloor}{\vdash \mathtt{rec}(\mathsf{f}, (x, y), e^{\mathsf{f}}_{(a)}) : \sigma_{\mathsf{f}^{\forall\exists}}}\,(\textsc{T-Total}) \quad \begin{array}{c}\dfrac{\dfrac{\Gamma_1, z{:}^{\exists}\,\mathsf{int} \vdash e'_{(a)} : \tau^{\exists\exists}_{e_{(a)}}}{\Gamma_1 \vdash \mathtt{let}\ z = *\ \mathtt{in}\ e'_{(a)} : \tau^{\exists\exists}_{e_{(a)}}}\,(\textsc{T-Rnd})}{\vdots}\,(\textsc{T-Let})\end{array}}{\vdash e_{(a)} : \tau^{\exists\exists}_{e_{(a)}}}\,(\textsc{T-Let})$$

Fig. 7. Type derivation of Example 2.1

Derivation of Example 2.1. Fig. 7 shows the derivation for Example 2.1. We focus only on the key parts of the derivation. Here, we would like to type $e_{(a)}$ as $\tau^{\exists\exists}_{e_{(a)}}$, where $\tau_{e_{(a)}} = \{u \mid u = 1\}$. As shown in the lower part of the figure, we first deconstruct the `let` bindings f, x, and y by applying the rule T-LET to the bodies three times.

Then, as shown in the left branch of the figure, we apply T-TOTAL with the well-founded relation $p(x,y,x',y') \triangleq x > x' \land y = y' \land x \geq y$ to type the term $\mathtt{rec}(\mathsf{f},(x,y),e^{\mathsf{f}}_{(a)})$ as $\sigma_{\mathsf{f}\forall\exists}$ where $\tau_{\mathsf{f}} = (x'{:}^{\forall}\,\mathtt{int}) \to (y'{:}^{\forall}\,\mathtt{int}) \to p(x,y,x',y') \rhd \{u \mid u = 0\}^{\forall\exists}$ (cf. Example 2.1 for the definition of $\sigma_{\mathsf{f}\forall\exists}$). Hence, f is also given the type $\sigma_{\mathsf{f}\forall\exists}$. In the right branch, we apply T-RND to the judgement $\Gamma_1 \vdash \mathtt{let}\ z = * \ \mathtt{in}\ e'_{(a)} : \tau^{\exists\exists}_{e(a)}$, where

$$\begin{aligned} e'_{(a)} &\triangleq \mathtt{let}\ w = \mathsf{f}\ x\ \mathtt{in}\ \mathtt{let}\ w' = w\ y\ \mathtt{in}\ w' + z \\ \Gamma_1 &\triangleq \mathsf{f} : \sigma_{\mathsf{f}\forall\exists}, x : \{u \mid u = 10{*}{*}9\}, y : \{u \mid u = 0\} \end{aligned}$$

We now would like to deconstruct `let` binding w, however, the environment $\Gamma_1, z{:}^{\exists}\mathtt{int}$ includes an existential binding of z, so that we first apply T-SKOLEM and eliminate the existential binding from the environment, as shown in the upper part of the figure.

Let Γ_2 be an environment $\Gamma_1, z{:}^{\exists}\mathtt{int}, z = 1$. We apply subtyping rules to derive $\Gamma_2 \vdash \mathsf{f} : \sigma_{\mathsf{f}\exists\exists}$ and derive the judgement $\Gamma_2 \vdash e'_{(a)} : \tau^{\exists\exists}_{e(a)}$.

$$\dfrac{\cdots \qquad \dfrac{\dfrac{\dfrac{\vdots}{x{:}^{\forall}\mathtt{int}, y{:}^{\forall}\{u \mid u \leq x\}, \mathsf{f}{:}^{\forall}\tau_{\mathsf{f}}, w{:}^{\forall}\mathtt{int}, w > x \vdash e^{\mathsf{f}'}_{(b)} : \{u \mid u = 0\}^{\exists\exists}}}{x{:}^{\forall}\mathtt{int}, y{:}^{\forall}\{u \mid u \leq x\}, \mathsf{f}{:}^{\forall}\tau_{\mathsf{f}}, w{:}^{\exists}\mathtt{int} \vdash e^{\mathsf{f}'}_{(b)} : \{u \mid u = 0\}^{\exists\exists}}\,(\text{T-SKOLEM})}{x{:}^{\forall}\mathtt{int}, y{:}^{\forall}\{u \mid u \leq x\}, \mathsf{f}{:}^{\forall}\tau_{\mathsf{f}} \vdash \mathtt{let}\ w = *\ \mathtt{in}\ e^{\mathsf{f}'}_{(b)} : \{u \mid u = 0\}^{\exists\exists}}\,(\text{T-RND})}{\dfrac{x{:}^{\forall}\mathtt{int}, y{:}^{\forall}\{u \mid u \leq x\}, \mathsf{f}{:}^{\forall}\tau_{\mathsf{f}} \vdash e^{\mathsf{f}}_{(b)} : \{u \mid u = 0\}^{\exists\exists}}{\vdash \mathtt{rec}(\mathsf{f},(x,y),e^{\mathsf{f}}_{(b)}) : \sigma_{\mathsf{f}\exists\exists}}\,(\text{T-TOTAL})}\,(\text{T-LET})(\text{T-IF})$$

Fig. 8. Type derivation of Example 2.2

Derivation of Example 2.2. The derivation of $e_{(b)}$ is similar to that of $e_{(a)}$ except for the derivation of $\vdash \mathtt{rec}(\mathsf{f},(x,y),e^{\mathsf{f}}_{(b)}) : \sigma_{\mathsf{f}\exists\exists}$ (cf. Example 2.1 for the definition of $\sigma_{\mathsf{f}\exists\exists}$). Fig. 8 shows the key parts of the derivation. We first apply T-TOTAL to the judgement $\vdash \mathtt{rec}(\mathsf{f},(x,y),e^{\mathsf{f}}_{(b)}) : \sigma_{\mathsf{f}\exists\exists}$ and obtain the judgement $x{:}^{\forall}\mathtt{int}, y{:}^{\forall}\{u \mid u \leq x\}, \mathsf{f}{:}^{\forall}\tau_{\mathsf{f}} \vdash e^{\mathsf{f}}_{(b)} : \{u \mid u = 0\}^{\exists\exists}$ where $\tau_{\mathsf{f}} \triangleq (x'{:}^{\forall}\mathtt{int}) \to (y'{:}^{\forall}\mathtt{int}) \to (x > x' \land y = y' \land x \geq y) \rhd \{u \mid u = 0\}^{\exists\exists}$.

This is followed by the applications of T-LET and T-IF. Here, $e^{\mathsf{f}'}_{(b)}$ is the following expression.

$$\begin{gathered} \mathtt{let}\ v = w > x\ \mathtt{in}\ \mathtt{ifz}\ v\ \mathtt{then}\ \mathtt{let}\ v = x - 1\ \mathtt{in}\ \mathtt{let}\ \mathsf{f}' = \mathsf{f}\ v\ \mathtt{in}\ \mathsf{f}'\ y \\ \mathtt{else}\ \mathtt{let}\ \mathsf{f}' = \mathsf{f}\ x\ \mathtt{in}\ \mathsf{f}'\ y \end{gathered}$$

In the `then` branch, we apply T-RND to deconstruct the `let` binding of w and apply T-SKOLEM with respect to w. The rest of the derivation is analogous to Example 2.1.

Derivation of Example 2.3. Fig. 9 shows the key parts of the derivation for Example 2.3. Here, $\Gamma_3 \triangleq \mathsf{app} :^{\forall} \sigma_{\mathsf{app}}, \mathsf{g} :^{\forall} \sigma_{\mathsf{g}}$. The derivation of the types σ_{app} for app and σ_{g} for g are standard and omitted (cf. e.g., [Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009]).

To derive $\Gamma_3 \vdash\ \mathtt{let}\ b = *\ \mathtt{in}\ \mathtt{ifz}\ b\ \mathtt{then}\ \mathsf{g}\ (-1)\ \mathtt{else}\ 0 : \{u \mid \bot\}^{\exists\forall}$, we first deconstruct the `let` binding of b and apply T-SKOLEM with $\phi \triangleq b = 0$. Next, we apply T-IF. In the left branch of the derivation, we obtain the judgement $\Gamma_3, b :^{\forall} \mathtt{int}, b = 0 \vdash \mathsf{g}\ (-1) : \{u \mid \bot\}^{\exists\forall}$, which is justified by

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_3, b{:}^{\forall}\,\mathtt{int}, b = 0 \vdash \mathtt{g}\,(-1) : \{u \mid \bot\}^{\exists\forall}} \qquad \dfrac{\dfrac{}{\Gamma_3, b{:}^{\forall}\,\mathtt{int}, b = 0, b \neq 0 \vdash 0 : \{u \mid u = 0\}^{\forall\exists}}\;(\text{T-I}{\small NT})}{\Gamma_3, b{:}^{\forall}\,\mathtt{int}, b = 0, b \neq 0 \vdash 0 : \{u \mid \bot\}^{\exists\forall}}\;(\text{T-S}{\small UB})}{\Gamma_3, b{:}^{\forall}\,\mathtt{int}, b = 0 \vdash \mathtt{ifz}\; b\; \mathtt{then}\; \mathtt{g}\,(-1)\; \mathtt{else}\; 0 : \{u \mid \bot\}^{\exists\forall}}\;(\text{T-I}{\small F})}{\Gamma_3, b{:}^{\exists}\,\mathtt{int} \vdash \mathtt{ifz}\; b\; \mathtt{then}\; \mathtt{g}\,(-1)\; \mathtt{else}\; 0 : \{u \mid \bot\}^{\exists\forall}}\;(\text{T-S}{\small KOLEM})}{\Gamma_3 \vdash \mathtt{let}\; b = * \;\mathtt{in}\; \mathtt{ifz}\; b\; \mathtt{then}\; \mathtt{g}\,(-1)\; \mathtt{else}\; 0 : \{u \mid \bot\}^{\exists\forall}}\;(\text{T-R}{\small ND})}{\vdots}$$

Fig. 9. Type derivation of Example 2.3

the types of g and app. In the right branch, we obtain the judgement $\Gamma_3, b{:}^{\forall}\,\mathtt{int}, b = 0, b \neq 0 \vdash 0 : \{u \mid \bot\}^{\exists\forall}$. Note that T-Sub can be applied because $b = 0 \wedge b \neq 0 \models \bot$.

5 RELATIVE COMPLETENESS EXTENSION

We present the relative completeness extension to the refinement type system. The key components of the extension are Gödel encoding of function-type values (that was treated only informally in Section 4) and *guarded intersection and union types.*

We extend the syntax of refinement types (cf. Fig. 4) by extending the qualified types σ to guarded intersection and union types:

$$\text{(guarded intersection and union types)} \quad \sigma ::= \quad \bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} \left(\phi_{ij} \rhd \tau_{ij}^{Q_{ij} Q'_{ij}}\right)$$

Here, we assume that $\ell, m_1, \ldots, m_\ell \geq 1$. We often omit $\bigvee_{i=1}^{\ell}$ if $\ell = 1$, and omit $\bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i}$ if $\ell = m_\ell = 1$. Note that the qualified types in the non-extended system are of the latter restricted form. Note also that we can safely assume that σ is always of the form $(\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} (\phi_{ij} \rhd \tau_{ij}^{Q_{ij} Q'_{ij}})$ or $\bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} (\phi_{ij} \rhd \tau_{ij}^{Q_{ij} Q'_{ij}})$ such that $ty(\tau_{ij}) = \mathtt{int}$.

The union types are used to express the complement of intersection types, as shown later in this section. On the other hand, the guarded intersection types allow the type system to collectively express different behaviors of functions and expressions depending on their arguments and free variables, and are essential for the proof of relative completeness (cf. Section 6.3) where such types are used to build the *strongest type* $\mathbf{ST}(e; \widetilde{p})$ of an expression e with respect to given postconditions $\widetilde{p}$, whose formal definition is deferred to Section 5.1.

The notion of free variables are extended to intersection and union types as shown below.

$$fvs\left(\bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} \sigma_{ij}\right) \triangleq \bigcup_{i=1}^{\ell} \bigcup_{j=1}^{m_i} fvs(\sigma_{ij})$$

Example 5.1. We can use a guarded intersection type to summarize (conditional) termination/non-termination behavior of a function. The type

$$(x : \mathtt{int}) \to (x > 0 \rhd \mathtt{int}^{\forall\exists}) \wedge (x < 0 \rhd \{y \mid \bot\}^{\forall\forall}) \wedge (x = 0 \rhd \mathtt{int}^{\exists\exists}) \wedge (x = 0 \rhd \{y \mid \bot\}^{\exists\forall})$$

says functions of this type always terminate if the argument x is positive, always diverge if x is negative, and otherwise, non-deterministically terminate or diverge. ▲

Next, we formally define the *complement types.* As discussed in Section 1, we define the complements of the qualifiers as follows: $\overline{\forall} = \exists$ and $\overline{\exists} = \forall$.

Definition 5.2 (Complement Types). The complement type $\neg\sigma$ of σ is defined by:

$$\neg\left(\tau^{QQ'}\right) \triangleq (\neg\tau)^{(\overline{Q})(\overline{Q'})}$$

$$\neg(\phi \rhd \sigma) \triangleq (\neg\phi \rhd \{x \mid \bot\}^{\forall\exists}) \wedge \neg\sigma$$

$$\neg\left(\bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} \sigma_{ij}\right) \triangleq \mathit{DNF}\left(\bigwedge_{i=1}^{\ell} \bigvee_{j=1}^{m_i} \neg\sigma_{ij}\right)$$

$$\neg\{x \mid \phi\} \triangleq \{x \mid \neg\phi\}$$

$$\neg((x{:}^{Q}\,\tau) \to \sigma) \triangleq (x{:}^{\overline{Q}}\,\tau) \to \neg\sigma$$

where *DNF* transforms a given intersection and union type to the disjunctive normal form.

Note here that $\neg(\phi \rhd \sigma)$ can be interpreted as $\phi \wedge \neg\sigma$ because $\rhd$ means a (type-level) logical implication. In the definition, the left-hand side ϕ of the intersection is represented as the guarded type $\neg\phi \rhd \{x \mid \bot\}^{\forall\exists}$. The complement $\neg\sigma$ of a given type σ can be used to witness that a given expression violates the specification represented by σ. Thus, we believe complement types pave a way to develop, in a future work, a verification method that simultaneously attempts a proof and a refutation in a cooperative and unified manner. In Section 6.1, we show the correctness of the definition, thereby showing that the types are closed under complement.

5.1 Encoding

Relative completeness for higher-order programs requires a means of precisely expressing properties of function values (i.e., partial applications). Following the previous work [Damm and Josko 1983; German et al. 1983, 1989; Goerdt 1985; Honda et al. 2006; Olderog 1984; Reus and Streicher 2011; Unno et al. 2013], we accomplish this by Gödel encoding. We remark that, as shown in [Unno et al. 2013], it is often possible to avoid explicit encoding in practice and that the encoding is unnecessary for soundness.[6]

We use a meta-variable w to range over closed values. A *value substitution* θ maps each variable $x \in \mathrm{dom}(\theta)$ to a closed value of the same simple type as x. We write $\theta(e)$ for e but with each variable x replaced by $\theta(x)$. For an expression e and a closed value w, let $\mathit{EvaluatesTo}(e, w)$ be the condition $\exists\widetilde{n}.\ e \xRightarrow{\widetilde{n}} w$, and $\mathit{AlwaysTerminates}(e)$ be the condition $\forall\pi \in \mathbb{Z}^{\omega}.\exists\widetilde{n} \in \mathit{Pref}(\pi).\exists w.\ e \xRightarrow{\widetilde{n}} w$ where $\mathbb{Z}^{\omega}$ denotes the set of infinite sequences of integers and $\mathit{Pref}(\pi)$ denotes the set of finite prefixes of the infinite sequence π. Note that $\mathit{EvaluatesTo}(e, w)$ means that there exists a (terminating) evaluation of e that reduces to the closed value w, and $\mathit{AlwaysTerminates}(e)$ means that the evaluation of e always terminates. We define $e \sim e'$ by induction on the simple type of e and e' as follows: for any closed value w and substitution θ with $\mathrm{dom}(\theta) = \mathit{fvs}(e) \cup \mathit{fvs}(e')$,

- $\mathit{AlwaysTerminates}(\theta(e)) \Leftrightarrow \mathit{AlwaysTerminates}(\theta(e'))$,
- if $\vdash_s \theta(e) : \mathtt{int}$, $\mathit{EvaluatesTo}(\theta(e), w) \Leftrightarrow \mathit{EvaluatesTo}(\theta(e'), w)$,
- if $\vdash_s \theta(e) : T_1 \to T_2$ and $\mathit{EvaluatesTo}(\theta(e), w)$, then there is w' such that $\mathit{EvaluatesTo}(\theta(e'), w')$ and $w\ w'' \sim w'w''$ for any closed value w'' with $\vdash_s w'' : T_1$, and
- if $\vdash_s \theta(e) : T_1 \to T_2$ and $\mathit{EvaluatesTo}(\theta(e'), w)$, then there is w' such that $\mathit{EvaluatesTo}(\theta(e), w')$ and $w\ w'' \sim w'w''$ for any closed value w'' with $\vdash_s w'' : T_1$.

Here, $\vdash_s$ is the typing relation of the simple type system.

We are now ready to describe the encoding. We write $\llbracket e \rrbracket$ for an integer term (in the theory of second-order integer arithmetic) that encodes the expression e where $\mathit{fvs}(\llbracket e \rrbracket) = \mathit{fvs}(e)$. We assume

[6] Technically, without encoding, the meaning of soundness would be different for open programs with free function-type variables as the variables would also range over non-definable functions.

that if $e \sim e'$ then $\models \llcorner e \lrcorner = \llcorner e' \lrcorner$, which indicates that the quotient set of expressions induced by the encoding is no more precise than the one induced by the equivalence relation $\sim$. We write $\llcorner \theta \lrcorner$ (resp. $\lfloor \theta \rfloor$) for the integer substitution that maps each (possibly non-integer) variable $x \in \mathrm{dom}(\theta)$ to the integer $\llcorner \theta(x) \lrcorner$ (resp. $\lfloor \theta(x) \rfloor$). (Recall the definition of $\lfloor v \rfloor$ in Fig. 6.)

We assume that there exist a binary function $\mathbf{app} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, a binary relation $\mathbf{ev} \subseteq \mathbb{Z} \times \mathbb{Z}$, and unary relations $\Downarrow, \mathbf{val}_T, \mathbf{exp}_T \subseteq \mathbb{Z}$ for each type T such that, for any expression e, a value v, and a value substitution θ where $\mathit{fvs}(e) \cup \mathit{fvs}(v) \subseteq \mathrm{dom}(\theta)$, the following conditions hold:

- $\models \llcorner \theta \lrcorner(\llcorner e \lrcorner) = \llcorner \theta(e) \lrcorner$
- $\models \mathbf{app}(\llcorner \theta \lrcorner(\llcorner e \lrcorner), \lfloor \theta \rfloor (\lfloor v \rfloor)) = \llcorner \texttt{let}\ x = \theta(e)\ \texttt{in}\ x\ (\theta(v)) \lrcorner$
- $\models \mathbf{ev}(\llcorner \theta \lrcorner(\llcorner e \lrcorner), \lfloor \theta \rfloor (\lfloor v \rfloor))$ if and only if $\exists w.(\mathit{EvaluatesTo}(\theta(e), w) \wedge w \sim \theta(v))$
- $\models \llcorner \theta \lrcorner(\llcorner e \lrcorner)\Downarrow$ if and only if $\mathit{AlwaysTerminates}(\theta(e))$
- $\models \mathbf{val}_T(\lfloor \theta \rfloor (\lfloor v \rfloor))$ if and only if $\vdash_s \theta(v) : T$
- $\models \mathbf{exp}_T(\llcorner \theta \lrcorner(\llcorner e \lrcorner))$ if and only if $\vdash_s \theta(e) : T$

Note that it follows from the conditions that $\models \llcorner x \lrcorner = x$, $\models \lfloor \theta \rfloor (\lfloor v \rfloor) = \lfloor \theta(v) \rfloor$, and if $\models \llcorner e_1 \lrcorner = \llcorner e_2 \lrcorner$ then $e_1 \sim e_2$. We write $\llcorner e \lrcorner \Uparrow$, meaning that e always diverges, as an abbreviation of the formula $\neg \exists x.\mathbf{ev}(\llcorner e \lrcorner, x)$. Accordingly, we define $\mathit{AlwaysDiverges}(e)$ by $\neg \exists x.\mathit{EvaluatesTo}(e, x)$. We also write $\mathbf{app}(x, y_1, \dots, y_m)$ as an abbreviation of $\mathbf{app}(\dots(\mathbf{app}(\mathbf{app}(x, y_1), y_2), \dots), y_m)$.

We now formalize the strongest type $\mathbf{ST}(e; \widetilde{p})$ of an expression e with respect to given postconditions $\widetilde{p}$, using the Gödel encoding and the guarded intersection types. Let the simple type of e be $\widetilde{T} \to \texttt{int}$. Note that, in our language, e is a function value if $|\widetilde{T}| \neq 0$, and an integer expression otherwise. Thus, $\mathbf{ST}(e; p_1, \dots, p_m)$ is defined as follows:

$$(\widetilde{x} :^{\forall} \widetilde{T}) \to \bigwedge \left\{ \begin{array}{rcl} \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Downarrow & \rhd & \left\{ v \,\middle|\, \mathbf{ev}(\mathbf{app}(\llcorner e \lrcorner, \widetilde{x}), v) \right\}^{\forall\exists}, \\ \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Uparrow & \rhd & \{ v \mid \bot \}^{\forall\forall}, \\ \neg \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Downarrow \wedge \neg \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Uparrow & \rhd & \{ v \mid \bot \}^{\exists\forall}, \\ \exists v.(\mathbf{ev}(\mathbf{app}(\llcorner e \lrcorner, \widetilde{x}), v) \wedge p_1(\widetilde{x}, v)) & \rhd & \{ v \mid p_1(\widetilde{x}, v) \}^{\exists\exists}, \\ & \vdots & \\ \exists v.(\mathbf{ev}(\mathbf{app}(\llcorner e \lrcorner, \widetilde{x}), v) \wedge p_m(\widetilde{x}, v)) & \rhd & \{ v \mid p_m(\widetilde{x}, v) \}^{\exists\exists} \end{array} \right\}$$

Here, $\left\{ v \,\middle|\, \mathbf{ev}(\mathbf{app}(\llcorner e \lrcorner, \widetilde{x}), v) \right\}^{\forall\exists}$ represents the strongest postcondition for the $\forall\exists$ mode, and the guard $\mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Uparrow$ and $\neg \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Downarrow \wedge \neg \mathbf{app}(\llcorner e \lrcorner, \widetilde{x})\Uparrow$ for the postcondition $\{ v \mid \bot \}^{\forall\forall}$ and $\{ v \mid \bot \}^{\exists\forall}$ represent the weakest precondition for e to always diverge and non-deterministically diverge, respectively. For each $i \in \{1, \dots, m\}$, the guard $\exists v.(\mathbf{ev}(\mathbf{app}(\llcorner e \lrcorner, \widetilde{x}), v) \wedge p_i(\widetilde{x}, v))$ represents the weakest precondition of the given postcondition $\{ v \mid p_i(\widetilde{x}, v) \}^{\exists\exists}$. In Section 6.3, we use strongest types to prove the relative completeness.

5.2 Denotational Semantics

We define the denotational semantics of types. The denotation $[\![\Gamma \vdash \sigma]\!]$ of a guarded intersection and union type σ under a type environment Γ is the set of expressions defined as shown in Fig. 10.

The denotation $[\![\tau]\!]$ of a refinement type τ is the set of closed values defined as follows:

$$[\![\{x \mid \phi\}]\!] \triangleq \{ n \mid \models [n/x]\,\phi \}$$
$$[\![(x :^{Q} \tau) \to \sigma]\!] \triangleq \{ w \in [\![\mathit{ty}(\tau \to \sigma)]\!] \mid Qw' \in [\![\tau]\!]\,.w\ w' \in [\![[\lfloor w' \rfloor /x]\,\sigma]\!] \}$$

Finally, the denotation $[\![T]\!]$ of a simple type T is the set of closed expressions defined as:

$$[\![T]\!] \triangleq \{ e \mid \vdash_s e : T \}$$

$$\llbracket \emptyset \vdash \sigma \rrbracket \triangleq \llbracket \sigma \rrbracket$$
$$\llbracket x{:}^{Q}\tau, \Gamma \vdash \sigma \rrbracket \triangleq \{e \mid Qw \in \llbracket \tau \rrbracket \,.\, [w/x]\, e \in \llbracket [\lfloor w \rfloor /x]\, \Gamma \vdash [\lfloor w \rfloor /x]\, \sigma \rrbracket\}$$
$$\left\llbracket \bigvee_{i=1}^{m} \sigma_i \right\rrbracket \triangleq \bigcup_{i=1}^{m} \llbracket \sigma_i \rrbracket \qquad \left\llbracket \bigwedge_{i=1}^{m} \sigma_i \right\rrbracket \triangleq \bigcap_{i=1}^{m} \llbracket \sigma_i \rrbracket$$
$$\llbracket \phi \rhd \sigma \rrbracket \triangleq \text{if } \models \phi \text{ then } \llbracket \sigma \rrbracket \text{ else } \llbracket ty(\sigma) \rrbracket$$
$$\llbracket \tau^{Q_1 Q_2} \rrbracket \triangleq \left\{ e \in \llbracket ty(\tau) \rrbracket \,\middle|\, Q_1 \pi \in \mathbb{Z}^{\omega}. Q_2 \widetilde{n} \in \mathit{Pref}(\pi). Q_2 w \in \left\{ w \,\middle|\, e \overset{\widetilde{n}}{\Longrightarrow} w \right\} . w \in \llbracket \tau \rrbracket \right\}$$

Fig. 10. The denotational semantics of types.

For a type environment Δ consisting of only universal bindings, we write $\theta \models \Delta$ if $\mathrm{dom}(\theta) = \mathrm{dom}(\Delta)$ and $\theta(x) \in \llbracket \theta(\tau) \rrbracket$ for all $(x{:}^{\forall}\tau) \in \Delta$.

The following are immediate from the definition of the denotational semantics.

Proposition 5.3 (Subject Reduction). *We have*

- *If* $e \in \llbracket \tau^{\forall Q} \rrbracket$ *and* $e \xrightarrow{\widetilde{n}} e'$, *then* $e' \in \llbracket \tau^{\forall Q} \rrbracket$.
- *If* $e \in \llbracket \tau^{\exists Q} \rrbracket$ *and* e *is not a value, then* $e \xrightarrow{\widetilde{n}} e' \in \llbracket \tau^{\exists Q} \rrbracket$ *for some* $\widetilde{n}$ *and* e'.

Proposition 5.4 (Subject Expansion). *Suppose that* $e \xrightarrow{\widetilde{n}_1} e_1 \in \llbracket \sigma \rrbracket$. *If* $e_1 = e_2$ *holds for any* $\widetilde{n}_2$ *and* e_2 *such that* $e \xrightarrow{\widetilde{n}_2} e_2$, *then* $e \in \llbracket \sigma \rrbracket$.

Next, we state several lemmas about the denotational semantics that we use to prove meta-theoretic properties of the type system in Section 6.

Lemma 5.5. *The following two are equivalent:*

- $e \in \llbracket \Delta, \Gamma \vdash \sigma \rrbracket$
- $\theta(e) \in \llbracket \lfloor \theta \rfloor (\Gamma) \vdash \lfloor \theta \rfloor (\sigma) \rrbracket$ *for all value substitutions* θ *with* $\theta \models \Delta$.

The following lemma states that $\llbracket \tau^{Q_1 Q_2} \rrbracket$ restricted to closed values is equivalent to $\llbracket \tau \rrbracket$.

Lemma 5.6. $\llbracket \tau \rrbracket = \{ w \mid w \in \llbracket \tau^{Q_1 Q_2} \rrbracket \}$ *holds for any* τ, Q_1, *and* Q_2.

The definition of $\llbracket \tau^{Q_1 Q_2} \rrbracket$ can be rewritten as shown in the following lemma:

Lemma 5.7. *We have*

- $\llbracket \tau^{\forall\exists} \rrbracket = \{ e \in \llbracket ty(\tau) \rrbracket \mid \mathit{AlwaysTerminates}(e) \land e \in \llbracket \tau^{\forall\forall} \rrbracket \}$
- $\llbracket \tau^{\forall\forall} \rrbracket = \{ e \in \llbracket ty(\tau) \rrbracket \mid \forall w \in \llbracket ty(\tau) \rrbracket . (\mathit{EvaluatesTo}(e, w) \Rightarrow w \in \llbracket \tau \rrbracket) \}$
- $\llbracket \tau^{\exists\exists} \rrbracket = \{ e \in \llbracket ty(\tau) \rrbracket \mid \exists w \in \llbracket ty(\tau) \rrbracket . (\mathit{EvaluatesTo}(e, w) \land w \in \llbracket \tau \rrbracket) \}$
- $\llbracket \tau^{\exists\forall} \rrbracket = \{ e \in \llbracket ty(\tau) \rrbracket \mid \neg(\mathit{AlwaysTerminates}(e)) \lor e \in \llbracket \tau^{\exists\exists} \rrbracket \}$

The following lemma says that refinement types are no more precise than the relation $\sim$:

Lemma 5.8. *If* $e \sim e'$ *then* $e \in \llbracket \Delta \vdash \sigma \rrbracket \Leftrightarrow e' \in \llbracket \Delta \vdash \sigma \rrbracket$ *for any* Δ *and* σ.

Proof. The statement follows from Lemma 5.7 and the assumption explained in Section 5.1 that the Gödel encoding we adopted is not more precise than the equivalence relation $\sim$. □

We next define the type subsumption relation $[\![\Gamma \vdash \sigma_1 <: \sigma_2]\!]$ for guarded intersection and union types:

$$[\![\emptyset \vdash \sigma_1 <: \sigma_2]\!] \triangleq [\![\sigma_1]\!] \subseteq [\![\sigma_2]\!]$$
$$[\![x{:}^Q \tau, \Gamma \vdash \sigma_1 <: \sigma_2]\!] \triangleq Qw \in [\![\tau]\!] \,.\, [\![[\lfloor w \rfloor / x]\Gamma \vdash [\lfloor w \rfloor / x]\sigma_1 <: [\lfloor w \rfloor / x]\sigma_2]\!]$$

Similarly, we define the subsumption relation $[\![\Gamma \vdash \tau_1 <: \tau_2]\!]$ for refinement types:

$$[\![\emptyset \vdash \tau_1 <: \tau_2]\!] \triangleq [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$$
$$[\![x{:}^Q \tau, \Gamma \vdash \tau_1 <: \tau_2]\!] \triangleq Qw \in [\![\tau]\!] \,.\, [\![[\lfloor w \rfloor / x]\Gamma \vdash [\lfloor w \rfloor / x]\tau_1 <: [\lfloor w \rfloor / x]\tau_2]\!]$$

We often abbreviate $[\![\emptyset \vdash \sigma_1 <: \sigma_2]\!]$ and $[\![\emptyset \vdash \tau_1 <: \tau_2]\!]$ respectively as $[\![\sigma_1 <: \sigma_2]\!]$ and $[\![\tau_1 <: \tau_2]\!]$. We obtain the following properties of the subsumption relations:

Lemma 5.9. *The following two are equivalent:*

- $[\![\Delta, \Gamma \vdash \sigma_1 <: \sigma_2]\!]$
- $[\![\lfloor\theta\rfloor(\Gamma) \vdash \lfloor\theta\rfloor(\sigma_1) <: \lfloor\theta\rfloor(\sigma_2)]\!]$ *for all value substitutions θ with $\theta \models \Delta$.*

Also, the following two are equivalent:

- $[\![\Delta, \Gamma \vdash \tau_1 <: \tau_2]\!]$
- $[\![\lfloor\theta\rfloor(\Gamma) \vdash \lfloor\theta\rfloor(\tau_1) <: \lfloor\theta\rfloor(\tau_2)]\!]$ *for all value substitutions θ with $\theta \models \Delta$.*

5.3 Extended Typing Rules

We extend the typing rules by the rules shown in Fig 11 that handle guarded intersection and union types. The figure also shows the extension of the auxiliary function $\lfloor \vdash x : \sigma \rfloor$ to intersection and union types. Now, having the complete definition of the encoding functions $\lfloor \vdash x : \tau \rfloor$, $\lfloor \vdash x : \sigma \rfloor$ and $\lfloor \Gamma \vdash \phi \rfloor$, we formalize their relationship to the denotational semantics. As shown below, the encoding functions are related to the semantics as follows.

Lemma 5.10. $\models [\lfloor w \rfloor / x] \lfloor \vdash x : \tau \rfloor$ *if and only if* $w \in [\![\tau]\!]$.

Lemma 5.11. *Let e be a closed expression. We have* $\models [\lfloor [\![e]\!] \rfloor / x] \lfloor \vdash x : \sigma \rfloor$ *if and only if* $e \in [\![\sigma]\!]$.

Lemma 5.12. *The following two are equivalent:*

- $\models \lfloor \Delta, \Gamma \vdash \phi \rfloor$
- $\models \lfloor\theta\rfloor(\lfloor \Gamma \vdash \phi \rfloor)$ *for any value substitutions θ with $\theta \models \Delta$.*

We explain the additional rules for handling guarded intersection and union types shown in Fig. 11. The auxiliary function *CNF* in the rule S-∨ transforms a given type to the conjunctive normal form. The rule T-Rec∧ generalizes T-Partial and T-Total to guarded intersection types. Thus, T-Rec∧ can replace the specific rules T-Partial and T-Total. The rule T-VFun∧ is for function variables. We already have the rule T-VFun for them but the rule T-VFun∧ can assign the strongest type of the variable expressed by using the Gödel encoding. The type assigned by T-VFun is in general not the strongest and insufficient for establishing the relative completeness. The rule T-Guard∧ can introduce a guarded intersection type of an arbitrary expression. The rules S-∨, S-∧, and S-∧∨⊥ are for rearranging a given intersection and union type. For example, we can derive:

$$x{:}^\forall \{x \mid x \geq 0\}, x \neq 0 \vdash \left(x \geq 1 \rhd \{y \mid y \geq x-1\}^{\forall\exists}\right) \wedge \left(x < 1 \rhd \{y \mid y \geq x-1\}^{\forall\forall}\right) <: \{y \mid y \geq 0\}^{\forall\exists}$$

The rules S-Qual⊥ and S-Fun⊥ (resp. the rules S-Guard⊤, S-Qual⊤, and S-Fun⊤) are necessary for relatively-completely deriving subtyping judgements $\Delta \vdash \sigma_1 <: \sigma_2$ such that $[\![\theta(\sigma_1)]\!] = \emptyset$ (resp. $[\![\theta(\sigma_2)]\!] = [\![ty(\sigma_2)]\!]$) for any $\theta \models \Delta$. Finally, the rules S-Case and S-Trans can be used to combine subtyping rules.

$$\frac{\begin{array}{c}\tau_f = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \bigwedge_{i=1}^{m}(\phi_i \rhd \tau_i^{Q_iQ'_i}) =_\alpha (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to \bigwedge_{i=1}^{m}\left(\phi'_i \rhd (\tau'_i)^{Q_iQ'_i}\right) \\ \{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset \qquad \tau_f^{(j)} = (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to \bigwedge_{i=1}^{m}\left(\phi_i^{(j)} \rhd (\tau'_i)^{Q_iQ'_i}\right) \\ \phi_i^{(j)} = \text{if } Q'_i = \exists \text{ then } \phi'_i \wedge \phi_{WF}^{(j)} \text{ else } \phi'_i \\ \mathit{fvs}(\phi_{WF}^{(j)}) \subseteq (\{\widetilde{x},\widetilde{y}\} \cup \mathrm{dom}(\Delta)) \qquad \models \left\lfloor \Delta \vdash \mathit{WF}(\lambda\widetilde{x}\widetilde{y}.\phi_{WF}^{(j)}) \right\rfloor \\ \Delta, \widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, \phi_j, f{:}^{\forall}\tau_f^{(j)} \vdash e : \tau_j^{Q_jQ'_j} \qquad (j = 1,\dots,m)\end{array}}{\Delta \vdash \mathtt{rec}(f,\widetilde{x},e) : \tau_f} \quad (\text{T-Rec}\wedge)$$

$$\frac{\mathit{ty}(\Delta(x)) \neq \mathtt{int}}{\Delta \vdash x : \mathbf{ST}(x;\widetilde{p})} \quad (\text{T-VFun}\wedge) \qquad \frac{\Delta, \phi_i \vdash e : \tau_i^{Q_iQ'_i} \quad (i = 1,\dots,m)}{\Delta \vdash e : \bigwedge_{i=1}^{m}(\phi_i \rhd \tau_i^{Q_iQ'_i})} \quad (\text{T-Guard}\wedge)$$

$$\frac{\Delta \vdash \sigma_i <: \sigma \qquad (i = 1,\dots,\ell)}{\Delta \vdash \bigvee_{i=1}^{\ell}\sigma_i <: \sigma} \quad (\text{S-}\vee) \qquad \frac{\begin{array}{c}\mathit{CNF}(\sigma') = \bigwedge_{i=1}^{\ell}\sigma_i \\ \Delta \vdash \sigma <: \sigma_i \qquad (i = 1,\dots,\ell)\end{array}}{\Delta \vdash \sigma <: \sigma'} \quad (\text{S-}\wedge)$$

$$\frac{\models \left\lfloor \Delta \vdash \neg\exists r. \left\lfloor \vdash r : \bigwedge_{i=1}^{m}\sigma_i \right\rfloor \right\rfloor}{\Delta \vdash \bigwedge_{i=1}^{\ell}\sigma_i <: \bigvee_{i=\ell+1}^{m}\neg\sigma_i} \quad (\text{S-}\wedge\vee\bot) \qquad \frac{\mathit{ty}(\sigma_1) = \mathit{ty}(\sigma_2)}{\Delta \vdash \sigma_1 <: \bot \rhd \sigma_2} \quad (\text{S-Guard}\top)$$

$$\frac{\mathit{ty}(\sigma) = \mathtt{int}}{\Delta \vdash \{\nu \mid \bot\}^{Q\exists} <: \sigma} \quad (\text{S-Qual}\bot) \qquad \frac{\mathit{ty}(\sigma) = \mathtt{int}}{\Delta \vdash \sigma <: \phi \rhd \{\nu \mid \top\}^{Q\forall}} \quad (\text{S-Qual}\top)$$

$$\frac{\mathit{ty}(\tau) = \widetilde{T} \to \mathtt{int}}{\Delta \vdash \widetilde{T} \to \{\nu \mid \bot\}^{Q\exists} <: \tau} \quad (\text{S-Fun}\bot) \qquad \frac{\mathit{ty}(\tau) = \widetilde{T} \to \mathtt{int}}{\Delta \vdash \tau <: \widetilde{T} \to \{\nu \mid \top\}^{Q\forall}} \quad (\text{S-Fun}\top)$$

$$\frac{\begin{array}{c}\mathit{fvs}(\phi) \subseteq \mathrm{dom}(\Delta) \\ \Delta, \phi \vdash \sigma_1 <: \sigma_2 \qquad \Delta, \neg\phi \vdash \sigma_1 <: \sigma_2\end{array}}{\Delta \vdash \sigma_1 <: \sigma_2} \quad (\text{S-Case}) \qquad \frac{\Delta \vdash \sigma_1 <: \sigma_2 \qquad \Delta \vdash \sigma_2 <: \sigma_3}{\Delta \vdash \sigma_1 <: \sigma_3} \quad (\text{S-Trans})$$

$$\left\lfloor \vdash x : \bigvee_{i=1}^{\ell}\bigwedge_{j=1}^{m_i}\sigma_{ij} \right\rfloor = \bigvee_{i=1}^{\ell}\bigwedge_{j=1}^{m_i}\left\lfloor \vdash x : \sigma_{ij} \right\rfloor$$

Fig. 11. The extension of the typing rules and the auxiliary function $\lfloor \vdash x : \sigma \rfloor$ for intersections and unions.

5.4 Toward Automation

We briefly remark on how one may automate the type checking and type inference for our type system. Though automated type inference is out of the scope of the present paper, we have carefully designed the type system to allow a future extension to automated inference, based on our experiences on developing refinement type inference and related methods [Hashimoto and Unno 2015; Kobayashi et al. 2011; Kuwahara et al. 2015, 2014; Terauchi 2010; Unno and Kobayashi 2009; Unno et al. 2013].

To develop a practical type checking or inference method, we need to devise a sound approximation of the Gödel encoding used to establish relative completeness (in the rule T-VFun∧ and the auxiliary functions). One possibility is to adopt the approach of [Unno et al. 2013] and use as the background theory an efficiently decidable theory such as the quantifier-free first-order theory of linear integer arithmetic instead of the second-order arithmetic, and employ a less precise but sound encoding of function values.[7] Note however that, unlike [Unno et al. 2013], we also need check the non-emptiness of $\exists x \in [\![\sigma]\!] .\phi$ at Skolemization. For the type semantics that we defined, this can be difficult when σ is a function type. One way to make the non-emptiness checking practical is to enlarge the domain of functions to arbitrary mathematical higher-order non-deterministic functions instead of the definable ones. This substantially simplifies the problem (indeed, the check can then be done by calling the decision procedure for the background theory). As remarked in Section 5.1, such a change still retains soundness, except that now function-type variables are assumed to also range over non-definable ones.

We remark that, the type inference requires, besides a sound approximation of the Gödel encoding, the synthesis of inductive invariants (expressed as dependent refinement types), well-founded relations (in T-Total and T-Rec∧), and predicates for Skolemization (in T-Skolem and S-Skolem). A possible approach to automating such tasks is to leverage constraint solving of existentially-quantified Horn clauses with well-foundedness constraints, for which existing techniques are available [Beyene et al. 2013; Hashimoto and Unno 2015; Kuwahara et al. 2015, 2014; Popeea and Rybalchenko 2012; Unno et al. 2013].

6 META-PROPERTIES OF THE TYPE SYSTEM

This section shows meta-properties of the type system. In Section 6.1, we prove the correctness of the type complement operator $\neg$, thereby proving that the types are closed under complement in our system. We then prove the soundness and the relative completeness respectively in Sections 6.2 and 6.3.

6.1 Correctness of Complement Types

The correctness is stated and proved using the denotational semantics as follows.

Theorem 6.1 (Correctness of Complement Types). *We have*

- *For any σ and integer substitution ρ with* $\mathrm{dom}(\rho) = \mathit{fvs}(\sigma)$, $[\![\rho(\neg\sigma)]\!] = [\![ty(\sigma)]\!] \setminus [\![\rho(\sigma)]\!]$ *holds.*
- *For any τ and integer substitution ρ with* $\mathrm{dom}(\rho) = \mathit{fvs}(\tau)$, $[\![\rho(\neg\tau)]\!] = [\![ty(\tau)]\!] \setminus [\![\rho(\tau)]\!]$ *holds.*

Proof. By mutual induction on the structure of σ and τ.

- **Case $\sigma = \tau^{Q_1 Q_2}$:** Let ρ be an integer substitution with $\mathrm{dom}(\rho) = \mathit{fvs}(\sigma)$. By I.H., we obtain $[\![\rho(\neg\tau)]\!] = [\![ty(\tau)]\!] \setminus [\![\rho(\tau)]\!]$. We then have

$$\begin{aligned}
[\![\rho(\neg\sigma)]\!] &= [\![(\rho(\neg\tau))^{\overline{Q_1 Q_2}}]\!] \\
&= \{e \in [\![ty(\rho(\neg\tau))]\!] \mid \overline{Q_1}\pi \in \mathbb{Z}^\omega.\overline{Q_2}\widetilde{n} \in \mathit{Pref}(\pi).\overline{Q_2}w \in \{w \mid e \overset{\widetilde{n}}{\Longrightarrow} w\}.w \in [\![\rho(\neg\tau)]\!]\} \\
&= \{e \in [\![ty(\tau)]\!] \mid \neg(Q_1\pi \in \mathbb{Z}^\omega.Q_2\widetilde{n} \in \mathit{Pref}(\pi).Q_2 w \in \{w \mid e \overset{\widetilde{n}}{\Longrightarrow} w\}.w \in [\![\rho(\tau)]\!])\} \\
&= [\![ty(\sigma)]\!] \setminus [\![\rho(\sigma)]\!]
\end{aligned}$$

- **Case $\sigma = \phi \rhd \sigma'$:** Let ρ be an integer substitution with $\mathrm{dom}(\rho) = \mathit{fvs}(\sigma)$. By I.H., we obtain $[\![\rho(\neg\sigma')]\!] = [\![ty(\sigma')]\!] \setminus [\![\rho(\sigma')]\!]$. If $\models \rho(\phi)$ holds, we get

$$[\![\rho(\neg\sigma)]\!] = [\![\rho(\neg\phi) \rhd \{x \mid \bot\}^{\forall\exists}]\!] \cap [\![\rho(\neg\sigma')]\!] = [\![\rho(\neg\sigma')]\!] = [\![ty(\sigma')]\!] \setminus [\![\rho(\sigma')]\!] = [\![ty(\sigma)]\!] \setminus [\![\rho(\sigma)]\!]$$

[7]The approximation is motivated by the observation that programmers rarely write programs whose correctness relies heavily on function-type arguments.

Otherwise (i.e., $\models \rho(\neg\phi)$), we have

$$[\![\rho(\neg\sigma)]\!] = [\![\rho(\neg\phi) \rhd \{x \mid \bot\}^{\forall\exists}]\!] \cap [\![\rho(\neg\sigma')]\!] = \emptyset = [\![ty(\sigma)]\!] \setminus [\![ty(\sigma')]\!] = [\![ty(\sigma)]\!] \setminus [\![\rho(\sigma)]\!]$$

- **Case** $\sigma = \bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} \sigma_{ij}$**:** Let ρ be an integer substitution with $\mathrm{dom}(\rho) = fvs(\sigma)$. By I.H., we obtain $[\![\rho(\neg\sigma_{ij})]\!] = [\![ty(\sigma_{ij})]\!] \setminus [\![\rho(\sigma_{ij})]\!]$. We then have
$$\begin{aligned}[\![\rho(\neg\sigma)]\!] &= \left[\!\!\left[DNF\left(\bigwedge_{i=1}^{\ell}\bigvee_{j=1}^{m_i}\rho(\neg\sigma_{ij})\right)\right]\!\!\right] = \bigcap_{i=1}^{\ell}\left(\bigcup_{j=1}^{m_i}\left([\![ty(\sigma_{ij})]\!] \setminus [\![\rho(\sigma_{ij})]\!]\right)\right)\\ &= [\![ty(\sigma)]\!] \setminus \bigcup_{i=1}^{\ell}\left(\bigcap_{j=1}^{m_i}[\![\rho(\sigma_{ij})]\!]\right) = [\![ty(\sigma)]\!] \setminus [\![\rho(\sigma)]\!]\end{aligned}$$
- **Case** $\tau = \{x \mid \phi\}$**:** Let ρ be an integer substitution with $\mathrm{dom}(\rho) = fvs(\tau)$. We then have $[\![\rho(\neg\tau)]\!] = [\![\{x \mid \neg\rho(\phi)\}]\!] = [\![ty(\tau)]\!] \setminus [\![\rho(\tau)]\!]$.
- **Case** $\tau = ((x :^{Q} \tau') \to \sigma)$**:** Let ρ be an integer substitution with $\mathrm{dom}(\rho) = fvs(\tau)$. By I.H., we obtain $[\![\rho'(\neg\sigma)]\!] = [\![ty(\sigma)]\!] \setminus [\![\rho'(\sigma)]\!]$ for any ρ' with $\mathrm{dom}(\rho') = fvs(\sigma)$. We thus get
$$\begin{aligned}[\![\rho(\neg\tau)]\!] &= \left[\!\!\left[(x :^{\overline{Q}} \rho(\tau')) \to \rho(\neg\sigma)\right]\!\!\right]\\ &= \left\{w \in [\![ty(\rho(\tau') \to \rho(\neg\sigma))]\!] \;\middle|\; \overline{Q}w' \in [\![\rho(\tau')]\!].w\,w' \in [\![[\lfloor w' \rfloor/x]\,\rho(\neg\sigma)]\!]\right\}\\ &= \{w \in [\![ty(\tau)]\!] \mid \neg(Qw' \in [\![\rho(\tau')]\!].w\,w' \in [\![[\lfloor w' \rfloor/x]\,\rho(\sigma)]\!])\}\\ &= [\![ty(\tau)]\!] \setminus [\![\rho(\tau)]\!]\end{aligned}$$

□

6.2 Soundness

We now prove the soundness of our type system with respect to the denotational semantics. We first show the necessary lemmas.

The following lemmas are used respectively to establish the soundness of the subtyping rules S-Guard, S-Qual, S-Fun∀∀, S-Fun∀∃, S-Fun∃∃, and S-Fun∃∀.

Lemma 6.2. $[\![\phi_1 \rhd \sigma_1 <: \phi_2 \rhd \sigma_2]\!]$ *if* $\models \phi_2 \Rightarrow \phi_1$ *and* $[\![\sigma_1 <: \sigma_2]\!]$.

Lemma 6.3. $\left[\!\!\left[\tau_1^{Q_1 Q_1'} <: \tau_2^{Q_2 Q_2'}\right]\!\!\right]$ *if* $[\![\tau_1 <: \tau_2]\!]$ *and* $Q_1 Q_1' \sqsubseteq Q_2 Q_2'$.

Lemma 6.4. $[\![(x :^{\forall} \tau_1) \to \sigma_1 <: (x :^{\forall} \tau_2) \to \sigma_2]\!]$ *if* $[\![\tau_2 <: \tau_1]\!]$ *and* $[\![x :^{\forall} \tau_2 \vdash \sigma_1 <: \sigma_2]\!]$.

Lemma 6.5. $[\![(x :^{\forall} \tau_1) \to \sigma_1 <: (x :^{\exists} \tau_2) \to \sigma_2]\!]$ *if* $[\![\tau <: \tau_1]\!]$, $[\![\tau <: \tau_2]\!]$, *and* $[\![x :^{\exists} \tau \vdash \sigma_1 <: \sigma_2]\!]$ *for some* τ.

Lemma 6.6. $[\![(x :^{\exists} \tau_1) \to \sigma_1 <: (x :^{\exists} \tau_2) \to \sigma_2]\!]$ *if* $[\![\tau_1 <: \tau_2]\!]$ *and* $[\![x :^{\forall} \tau_1 \vdash \sigma_1 <: \sigma_2]\!]$.

Lemma 6.7. $[\![(x_1 :^{\exists} \tau_1) \to \sigma_1 <: (x_2 :^{\forall} \tau_2) \to \sigma_2]\!]$ *if the following conditions hold:*

- $\models \lfloor x_1 :^{\forall} \tau_1, x_2 :^{\forall} \tau_2, y_1 :^{\forall} \tau_1, y_2 :^{\forall} \tau_2 \vdash x_1 = x_2 \land y_1 = y_2 \Rightarrow x_1 = y_1 \rfloor$,
- $x_1 :^{\forall} \tau_1, x_2 :^{\forall} \tau_2, x_1 = x_2 \vdash \sigma_1 <: \sigma_2$,
- $x_1 :^{\forall} \tau_1, x_2 :^{\exists} \tau_2, x_1 \neq x_2 \vdash \sigma_1 <: \widetilde{T} \to \{\nu \mid \bot\}^{\exists\exists}$,
- $x_2 :^{\forall} \tau_2, x_1 :^{\exists} \tau_1, x_1 \neq x_2 \vdash \widetilde{T} \to \{\nu \mid \top\}^{\forall\forall} <: \sigma_2$, *and*
- $ty(\sigma_1) = \widetilde{T} \to \mathtt{int}$.

We thus obtain the soundness of the subtyping rules with respect to the subsumption relations.

Lemma 6.8 (Soundness of Subtyping). *We have:*

- $[\![\Gamma \vdash \sigma_1 <: \sigma_2]\!]$ *if* $\Gamma \vdash \sigma_1 <: \sigma_2$.
- $[\![\Gamma \vdash \tau_1 <: \tau_2]\!]$ *if* $\Gamma \vdash \tau_1 <: \tau_2$.

We next show lemmas that are used to establish the soundness of the typing rules T-Partial, T-Total, and T-Rec$\wedge$ for recursive functions. The following lemma provides a sufficient condition for a recursive function to be included in the denotation of a partial correctness function type.

Lemma 6.9. *Suppose that* $e \in \left[\!\!\left[\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, f : \tau_f \vdash \tau^{Q\forall}\right]\!\!\right]$. *Then,* $\mathsf{rec}(f, \widetilde{x}, e) \in \left[\!\!\left[\tau_f\right]\!\!\right]$, *where* $\tau_f = (\widetilde{x}{:}^{\widetilde{Q}} \widetilde{\tau}) \to \tau^{Q\forall}$.

Next, we state the sufficient condition for a recursive function to be included in the denotation of a total correctness function type.

Lemma 6.10. *Let* τ_f *and* τ'_f *be types of the form* $\tau_f = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \tau^{Q\exists}$ *and* $\tau'_f = (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to \phi \triangleright (\tau')^{Q\exists}$ *that satisfy the following condition*

- $\{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset$;
- $\tau_f =_\alpha (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to (\tau')^{Q\exists}$;
- $\models WF(\lambda \widetilde{x}\widetilde{y}.\phi)$; *and*
- $fvs(\phi) \subseteq \{\widetilde{x}, \widetilde{y}\}$.

Then, $e \in \left[\!\!\left[\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, f : \tau'_f \vdash \tau^{Q\exists}\right]\!\!\right]$ *implies* $\mathsf{rec}(f, \widetilde{x}, e) \in \left[\!\!\left[\tau_f\right]\!\!\right]$.

For guarded intersection types, we get the following generalization of Lemmas 6.9 and 6.10.

Lemma 6.11. *Let* τ_f *and* $\tau_f^{(j)}$ *be types of the form* $\tau_f = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \bigwedge_{i=1}^m(\phi_i \triangleright \tau_i^{Q_iQ'_i})$ *and* $\tau_f^{(j)} = (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to \bigwedge_{i=1}^m(\phi_i^{(j)} \triangleright (\tau'_i)^{Q_iQ'_i})$ *that satisfy the following condition:*

- $\{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset$;
- $\tau_f =_\alpha (\widetilde{y}{:}^{\widetilde{Q}}\widetilde{\tau}') \to \bigwedge_{i=1}^m (\phi'_i \triangleright (\tau'_i)^{Q_iQ'_i})$;
- $\phi_i^{(j)} = \phi'_i \wedge \phi_{WF}^{(j)}$ *if* $Q'_i = \exists$ *and* $\phi_i^{(j)} = \phi'_i$ *otherwise;*
- $\models \left\lfloor \Delta \vdash WF(\lambda\widetilde{x}\widetilde{y}.\phi_{WF}^{(j)})\right\rfloor$; *and*
- $fvs(\phi_{WF}^{(j)}) \subseteq (\{\widetilde{x}, \widetilde{y}\} \cup \mathrm{dom}(\Delta))$.

Then, we have $\mathsf{rec}(f, \widetilde{x}, e) \in \left[\!\!\left[\tau_f\right]\!\!\right]$ *if* $e \in \left[\!\!\left[\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}, \phi_j, f : \tau_f^{(j)} \vdash \tau_j^{Q_jQ'_j}\right]\!\!\right]$ *for each* $j \in \{1, \ldots, m\}$.

Finally, we obtain the following soundness theorem by induction on the derivation of $\Gamma \vdash e : \sigma$.

Theorem 6.12 (Soundness). $e \in [\![\Gamma \vdash \sigma]\!]$ *if* $\Gamma \vdash e : \sigma$.

6.3 Relative Completeness

We prove the relative completeness of our type system with respect to the denotational semantics. We first show the necessary lemmas.

The following lemma ensures that we can always apply Skolemization first to eliminate existentially bound variables.

Lemma 6.13 (Skolemization). *If* $e \in \left[\!\!\left[\Delta, x{:}^{\exists}\tau, \Gamma \vdash \sigma\right]\!\!\right]$, *then there exists* ϕ *with* $fvs(\phi) \subseteq (fvs(\Delta) \cup \{x\})$ *such that* $e \in \left[\!\!\left[\Delta, x{:}^{\forall}\tau, \phi, \Gamma \vdash \sigma\right]\!\!\right]$ *and* $\models \left\lfloor \Delta, x{:}^{\exists}\tau \vdash \phi \right\rfloor$.

Proof. Suppose that $e \in \left[\!\!\left[\Delta, x{:}^{\exists}\tau, \Gamma \vdash \sigma\right]\!\!\right]$. By Lemma 5.5, we get $\exists w \in [\![\lfloor\theta\rfloor(\tau)]\!].[w/x](\theta(e)) \in [\![\lfloor\theta\rfloor(\Gamma) \vdash [\lfloor w\rfloor/x](\lfloor\theta\rfloor(\sigma))]\!]$ for any value substitution θ with $\theta \models \Delta$. There exists ϕ with $fvs(\phi) \subseteq (\mathrm{dom}(\Delta) \cup \{x\})$ such that:

- $\forall w \in [\![\lfloor\theta\rfloor(\tau)]\!] .\ \models [\lfloor w\rfloor/x](\lfloor\theta\rfloor(\phi)) \Rightarrow [w/x](\theta(e)) \in [\![\lfloor\theta\rfloor(\Gamma) \vdash [\lfloor w\rfloor/x](\lfloor\theta\rfloor(\sigma))]\!]$ and
- $\exists w \in [\![\lfloor\theta\rfloor(\tau)]\!] .\ \models [\lfloor w\rfloor/x](\lfloor\theta\rfloor(\phi))$ for any value substitution θ with $\theta \models \Delta$.

By Lemma 5.5, we obtain $e \in [\![\Delta, x{:}^{\forall}\tau, \phi, \Gamma \vdash \sigma]\!]$. By Lemma 5.12, we get $\models \lfloor\Delta, x{:}^{\exists}\tau \vdash \phi\rfloor$. □

The following lemmas are used respectively to establish the relative completeness of the subtyping rules S-Guard, S-Qual⊥, S-Qual⊤, S-Qual, S-Fun∀∀, S-Fun∀∃, S-Fun∃∃, and S-Fun∃∀.[8]

Lemma 6.14. *If $\emptyset \neq [\![\phi_1 \rhd \sigma_1]\!] \subseteq [\![\phi_2 \rhd \sigma_2]\!] \neq [\![ty(\sigma_2)]\!]$, then $\models \phi_2 \Rightarrow \phi_1$ and $[\![\sigma_1 <: \sigma_2]\!]$.*

Lemma 6.15. *If $[\![\{x \mid \phi\}^{Q_1Q_2}]\!] = \emptyset$, then $\models \neg\phi$ and $Q_2 = \exists$.*

Lemma 6.16. *If $[\![\{x \mid \phi\}^{Q_1Q_2}]\!] = [\![\mathtt{int}]\!]$, then $\models \phi$ and $Q_2 = \forall$.*

Lemma 6.17. *If $\emptyset \neq [\![\tau_1^{Q_1Q_1'}]\!] \subseteq [\![\tau_2^{Q_2Q_2'}]\!] \neq [\![ty(\tau_2)]\!]$, then $[\![\tau_1 <: \tau_2]\!]$ and $Q_1Q_1' \sqsubseteq Q_2Q_2'$.*

Lemma 6.18. *If $\emptyset \neq [\![(x{:}^{\forall}\tau_1) \to \sigma_1]\!] \subseteq [\![(x{:}^{\forall}\tau_2) \to \sigma_2]\!] \neq \{w \mid \vdash_s w : ty(\tau_2) \to ty(\sigma_2)\}$, then $[\![\tau_2 <: \tau_1]\!]$ and $[\![x{:}^{\forall}\tau_2 \vdash \sigma_1 <: \sigma_2]\!]$.*

Lemma 6.19. *If $\emptyset \neq [\![(x{:}^{\forall}\tau_1) \to \sigma_1]\!] \subseteq [\![(x{:}^{\exists}\tau_2) \to \sigma_2]\!] \neq \{w \mid \vdash_s w : ty(\tau_2) \to ty(\sigma_2)\}$, then $[\![\tau <: \tau_1]\!]$, $[\![\tau <: \tau_2]\!]$, and $[\![x{:}^{\exists}\tau \vdash \sigma_1 <: \sigma_2]\!]$ for some τ.*

Lemma 6.20. *If $\emptyset \neq [\![(x{:}^{\exists}\tau_1) \to \sigma_1]\!] \subseteq [\![(x{:}^{\exists}\tau_2) \to \sigma_2]\!] \neq \{w \mid \vdash_s w : ty(\tau_2) \to ty(\sigma_2)\}$, then $[\![\tau_1 <: \tau_2]\!]$ and $[\![x{:}^{\forall}\tau_1 \vdash \sigma_1 <: \sigma_2]\!]$.*

Lemma 6.21. *If $\emptyset \neq [\![(x_1{:}^{\exists}\tau_1) \to \sigma_1]\!] \subseteq [\![(x_2{:}^{\forall}\tau_2) \to \sigma_2]\!] \neq \{w \mid \vdash_s w : ty(\tau_2) \to ty(\sigma_2)\}$, then the following conditions hold:*

- $\models \lfloor x_1{:}^{\forall}\tau_1, x_2{:}^{\forall}\tau_2, y_1{:}^{\forall}\tau_1, y_2{:}^{\forall}\tau_2 \vdash x_1 = x_2 \wedge y_1 = y_2 \Rightarrow x_1 = y_1\rfloor$,
- $x_1{:}^{\forall}\tau_1, x_2{:}^{\forall}\tau_2, x_1 = x_2 \vdash \sigma_1 <: \sigma_2$,
- $x_1{:}^{\forall}\tau_1, x_2{:}^{\exists}\tau_2, x_1 \neq x_2 \vdash \sigma_1 <: \widetilde{T} \to \{\nu \mid \bot\}^{\exists\exists}$,
- $x_2{:}^{\forall}\tau_2, x_1{:}^{\exists}\tau_1, x_1 \neq x_2 \vdash \widetilde{T} \to \{\nu \mid \top\}^{\forall\forall} <: \sigma_2$, and
- $ty(\sigma_1) = \widetilde{T} \to \mathtt{int}$.

We then obtain the following relative completeness of the subtyping rules with respect to the subsumption relations.

Lemma 6.22 (Relative Completeness of Subtyping). *We have:*

- $\Delta \vdash \tau_1 <: \tau_2$ *if* $[\![\Delta \vdash \tau_1 <: \tau_2]\!]$.
- $\Delta \vdash \sigma_1 <: \sigma_2$ *if* $[\![\Delta \vdash \sigma_1 <: \sigma_2]\!]$.

The following lemma states the correctness of the strongest type $\mathbf{ST}(e;\widetilde{p})$.

Lemma 6.23. *Suppose that $e \in [\![\Delta \vdash \sigma]\!]$ for some $\sigma = (\widetilde{x}{:}^{\widetilde{Q}}\widetilde{\tau}) \to \bigvee_{i=1}^{\ell} \bigwedge_{j=1}^{m_i} (\phi_{ij} \rhd \tau_{ij}^{Q_{ij}Q'_{ij}})$. Let $\{\widetilde{p}\} = \{p \mid i \in \{1,\dots,\ell\}, j \in \{1,\dots,m_i\}, Q_{ij} = \exists, \tau_{ij} = \{\nu \mid p(\widetilde{x},\nu)\}\}$. We have $[\![\Delta \vdash \mathbf{ST}(e;\widetilde{p}) <: \sigma]\!]$.*

The following lemma says that the typing rules can in fact assign the strongest type $\mathbf{ST}(e;\widetilde{p})$ to any simply-typed expression e.

Lemma 6.24. *Suppose that $A \vdash_s e : \widetilde{T} \to \mathtt{int}$ for some A and $\widetilde{T}$. We then have $\Delta \vdash e : \mathbf{ST}(e;\widetilde{p})$ for any Δ and $\widetilde{p}$ such that $ty(\Delta) = A$, and each element p of $\widetilde{p}$ is well-formed (i.e., the arity of p is $|\widetilde{T}| + 1$ and $fvs(p) \subseteq \mathrm{dom}(A)$).*

[8]To prove the lemmas, we use an extension of the denotational semantics of types that ranges over $\mathcal{L}$-definable expressions that may use the equivalence relation $\sim$.

Finally, we obtain the following relative completeness theorem.

Theorem 6.25 (Relative Completeness). *$\Gamma \vdash e : \sigma$ if $e \in [\![\Gamma \vdash \sigma]\!]$.*

Proof. By the rule T-Skolem and Lemma 6.13, it suffices to show that $\Delta \vdash e : \sigma$ is implied by $e \in [\![\Delta \vdash \sigma]\!]$ for any Δ. By Lemmas 6.23 and 6.22, we obtain $\Delta \vdash \mathbf{ST}(e; \widetilde{p}) <: \sigma$ for some well-formed $\widetilde{p}$. By Lemma 6.24 and the rule T-Sub, we get $\Delta \vdash e : \sigma$. □

7 RELATED WORK

This paper proposes a refinement-type-based approach to verifying non-deterministic higher-order functional programs. The type system combines universal and existential reasoning in a unified framework, and is able to soundly-and-relatively-completely verify various important classes of properties including safety, non-safety, conditional termination, and conditional non-termination. To our knowledge, our work is the first of its kind.

As remarked in Section 1, previous work on higher-order program verification employs rather disparate methods to verify different classes of properties [Hashimoto and Unno 2015; Jhala et al. 2011; Kobayashi et al. 2011; Koskinen and Terauchi 2014; Kuwahara et al. 2015, 2014; Murase et al. 2016; Ong and Ramsay 2011; Rondon et al. 2008; Terauchi 2010; Unno and Kobayashi 2009; Unno et al. 2013; Vazou et al. 2014; Zhu and Jagannathan 2013; Zhu et al. 2015]. To our knowledge, there has only been a limited consideration for combined universal and existential reasoning in the setting of higher-order program verification (with a few notable exceptions such as the use of safety to prove non-termination by way of iterative predicate abstraction and higher-order model checking in [Kuwahara et al. 2015]). By contrast, we have proposed a unified type-based framework in which the type judgements expressing universal and existential facts can be readily combined as sub-derivations to derive the goal fact. As demonstrated in Section 2, the seamless combined reasoning offers powerful verification tactics that permit succinct proofs of non-trivial verification instances.

In the context of verification of programs with first-order functions and procedures, previous work has considered combining universal and existential reasoning [Ball et al. 2005; Godefroid and Huth 2005; Godefroid et al. 2001, 2010; Gurfinkel and Chechik 2006; Gurfinkel et al. 2006, 2008]. (In the literature, universal-existential reasoning is sometimes referred to as *may-must*.) However, to our knowledge, no previous work covers the combination patterns of this paper nor proposes a unified deduction system to carry out the combined reasoning. For instance, [Godefroid et al. 2001] proposes a safety and non-safety property verification method for first-order programs that allows a combined use of *may function summaries* and *must function summaries*. May summaries correspond to the types of the form $(\widetilde{x} :^{\widetilde{\forall}} \widetilde{\tau}) \rightarrow \tau^{\forall\forall}$ of our type system (where $\widetilde{\tau}$ and τ are restricted to base types), whereas must summaries say that, for every state satisfying the postcondition, there is a state satisfying the precondition and an evaluation of the function from the pre state that converges to the post state. Their approach does not have notions corresponding to the other modes supported by our system. Therefore, for example, their approach cannot use a conditional termination proof via well-foundedness termination argument to deduce the existence of an evaluation path satisfying a certain fact, as we have done in Example 2.1 and Example 2.2. Such reasoning is also beyond the scope of the approaches proposed in [Gurfinkel and Chechik 2006; Gurfinkel et al. 2006, 2008]. On the other hand, our system lacks an exact counterpart of their must summaries, and we leave for future work to investigate the implications of incorporating such a notion.[9]

[9]However, must summaries are only built from concrete evaluation paths and used as such in [Godefroid et al. 2001], and it is possible to do the examples in their paper with our system by using the types of the form $(\widetilde{x} :^{\widetilde{\exists}} \widetilde{\tau}) \rightarrow \tau^{\exists\exists}$ in place of must summaries.

The combined universal and existential reasoning is often considered in the context of verifying temporal logic properties [Beyene et al. 2013; Cook et al. 2015; Cook and Koskinen 2013; Gabbay and Pnueli 2008; Godefroid and Huth 2005; Godefroid et al. 2001; Shoham and Grumberg 2004, 2007]. This is because such properties can themselves be a combination of safety and liveness properties, or contain alternations of universal and existential branches (for branching logics). Compared to our work that handle higher-order and non-deterministic programs, these works typically focus on much simpler transition systems but handle richer temporal properties.[10] Similar to our work, they reason about *AG* and *AF* based on inductive invariants and well-founded relations, and some have rules for reasoning about *EG* and *EF* via Skolemization [Cook et al. 2015; Cook and Koskinen 2013]. We generalize such reasoning to the higher-order setting using refinement types and well-founded relations over (Gödel encoded) higher-type objects. An advantage of our system, especially in the context of expressive programming languages like higher-order functional languages, is that it allows modular reasoning thanks to the compositional nature of the type system.

By contrast, the existing work on temporal logic property verification for higher-order programs has paid little attention to the combined reasoning aspect. For instance, the method proposed by [Murase et al. 2016] applies the automata-theoretic reduction to convert the verification problem wholly to a fair termination problem, and the method proposed by [Koskinen and Terauchi 2014] considers only one-sided "blackbox" combination patterns that connect external oracle analyses' results to their type and effect system (also, neither approach addresses branching logics).[11] Our current system does not support temporal logics. Nonetheless, we believe that the ideas developed in this paper will contribute to improving the methods for verifying temporal logic properties of higher-order programs. We leave for future work to extend our approach to the full range of temporal logic properties.

The previous work has proposed dependent refinement type systems for partial and total correctness verification [Bengtson et al. 2011; Rondon et al. 2008; Swamy et al. 2011, 2016; Terauchi 2010; Unno and Kobayashi 2009; Vazou et al. 2014; Xi 2001]. The majority of such works only address partial correctness, with [Swamy et al. 2016; Vazou et al. 2014; Xi 2001] being the exception that also address total correctness (or just termination). Total correctness is verified via typing rules similar to the ones of our work. Also, while many of them (often implicitly) support demonic non-determinism, to our knowledge, none of them supports angelic non-determinism. We remark that the complexity of our type system is mainly due to supporting (1) angelic non-determinism, (2) the combined universal and existential reasoning, and (3) relative completeness. If we drop the support for (1), then our system can be simplified by removing the existential bindings in the type environment and the arguments of function types, and removing the rules T-Rnd, T-Skolem, T-App∃, and S-Skolem. If we drop (2), then our system can be further simplified by removing the rules S-Qual, S-Fun∀∃, and S-Fun∃∀. And, if we drop (3), then we can remove Gödel encoding, guarded intersection and union types, and the additional rules from Section 5 for handling them.

Our type system is sound and relatively-complete. The well-known issue in achieving relative completeness for higher-order programs is the representation of function-type parameters. Following the previous work on relatively complete verification of higher-order programs [Damm and Josko 1983; German et al. 1983, 1989; Goerdt 1985; Honda et al. 2006; Olderog 1984; Reus and Streicher 2011; Unno et al. 2013], we completely handle function parameters by encoding them as first-order data ([Unno et al. 2013] shows how such a formalism can be made practical

[10]In the terminology of temporal logics, the specifications in our system are limited to the forms $p \Rightarrow AGq$, $p \Rightarrow AFq$, $p \Rightarrow EGq$, and $p \Rightarrow EFq$.

[11]Sound and complete verification methods exist for *finite-data* higher-order programs (i.e., higher-order recursion schemes) for the expressive class of (branching) modal μ-calculus [Kobayashi and Ong 2009; Ong 2006]. However, it is unclear how such methods may be adopted to infinite data programs.

for automation by avoiding explicit encoding). However, while the previous work only considered safety properties (i.e., partial correctness), our system is sound and relatively-complete also for non-safety, conditional termination and conditional non-termination.

8 CONCLUSION

We have presented a novel type system for verification of non-deterministic higher-order functional programs. The type system is a dependent refinement type system extended to support universal and existential reasoning, and is sound and relatively-complete for the verification of various important classes of properties including safety, non-safety, conditional termination, and conditional non-termination. We have shown that the type system allows combined universal and existential reasoning, which can lead to succinct proofs of hard verification instances. We believe that this work will serve as a theoretical foundation toward automatic verification of non-deterministic higher-order programs, and also pave the way for new verification methods that combine universal and existential reasoning.

ACKNOWLEDGMENTS

We thank Naoki Kobayashi and anonymous referees for useful comments and Jean Yang for being our shepherd. This work was supported by JSPS KAKENHI Grant Numbers 15H05706, 16H05856, 17H01720, and 17H01723; JSPS Bilateral Programs; and JSPS Core-to-Core Program, A.Advanced Research Networks.

REFERENCES

Thomas Ball, Orna Kupferman, and Greta Yorsh. 2005. Abstraction for Falsification. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science)*, Kousha Etessami and Sriram K. Rajamani (Eds.), Vol. 3576. Springer, 67–81. https://doi.org/10.1007/11513988_8

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2 (2011), 8:1–8:45. https://doi.org/10.1145/1890028.1890031

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 869–882. https://doi.org/10.1007/978-3-642-39799-8_61

Rastislav Bodík and Rupak Majumdar (Eds.). 2016. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* ACM. http://dl.acm.org/citation.cfm?id=2837614

Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 156–171. https://doi.org/10.1007/978-3-642-54862-8_11

Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving that programs eventually do something good. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 265–276. https://doi.org/10.1145/1190216.1190257

Byron Cook, Heidy Khlaaf, and Nir Piterman. 2015. On Automation of CTL* Verification for Infinite-State Systems. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9206. Springer, 13–29. https://doi.org/10.1007/978-3-319-21690-4_2

Byron Cook and Eric Koskinen. 2013. Reasoning about nondeterminism in programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 219–230. https://doi.org/10.1145/2491956.2491969

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 415–426. https://doi.org/10.1145/1133981.1134029

Werner Damm and Bernhard Josko. 1983. A Sound and Relatively * Complete Hoare-Logic for a Language With Higher Type Procedures. *Acta Inf.* 20 (1983), 59–101. https://doi.org/10.1007/BF00264295

Dov M. Gabbay and Amir Pnueli. 2008. A Sound and Complete Deductive System for CTL* Verification. *Logic Journal of the IGPL* 16, 6 (2008), 499–536. https://doi.org/10.1093/jigpal/jzn018

Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. 1983. Reasoning About Procedures as Parameters. In *Logics of Programs, Workshop, Carnegie Mellon University, Pittsburgh, PA, USA, June 6-8, 1983, Proceedings (Lecture Notes in Computer Science)*, Edmund M. Clarke and Dexter Kozen (Eds.), Vol. 164. Springer, 206–220. https://doi.org/10.1007/3-540-12896-4_365

Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. 1989. Reasoning about Procedures as Parameters in the Language L4. *Inf. Comput.* 83, 3 (1989), 265–359. https://doi.org/10.1016/0890-5401(89)90040-0

Patrice Godefroid and Michael Huth. 2005. Model Checking Vs. Generalized Model Checking: Semantic Minimizations for Temporal Logics. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*. IEEE Computer Society, 158–167. https://doi.org/10.1109/LICS.2005.28

Patrice Godefroid, Michael Huth, and Radha Jagadeesan. 2001. Abstraction-Based Model Checking Using Modal Transition Systems. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings (Lecture Notes in Computer Science)*, Kim Guldstrand Larsen and Mogens Nielsen (Eds.), Vol. 2154. Springer, 426–440. https://doi.org/10.1007/3-540-44685-0_29

Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. 2010. Compositional may-must program analysis: unleashing the power of alternation, See [Hermenegildo and Palsberg 2010], 43–56. https://doi.org/10.1145/1706299.1706307

Andreas Goerdt. 1985. A Hoare Calculus for Functions Defined by Recursion on Higher Types. In *Logics of Programs, Conference, Brooklyn College, June 17-19, 1985, Proceedings (Lecture Notes in Computer Science)*, Rohit Parikh (Ed.), Vol. 193. Springer, 106–117. https://doi.org/10.1007/3-540-15648-8_9

Arie Gurfinkel and Marsha Chechik. 2006. Why Waste a Perfectly Good Abstraction?. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (Lecture Notes in Computer Science)*, Holger Hermanns and Jens Palsberg (Eds.), Vol. 3920. Springer, 212–226. https://doi.org/10.1007/11691372_14

Arie Gurfinkel, Ou Wei, and Marsha Chechik. 2006. Yasm: A Software Model-Checker for Verification and Refutation. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science)*, Thomas Ball and Robert B. Jones (Eds.), Vol. 4144. Springer, 170–174. https://doi.org/10.1007/11817963_18

Arie Gurfinkel, Ou Wei, and Marsha Chechik. 2008. Model Checking Recursive Programs with Exact Predicate Abstraction. In *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings (Lecture Notes in Computer Science)*, Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan (Eds.), Vol. 5311. Springer, 95–110. https://doi.org/10.1007/978-3-540-88387-6_9

Kodai Hashimoto and Hiroshi Unno. 2015. Refinement Type Inference via Horn Constraint Optimization. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science)*, Sandrine Blazy and Thomas Jensen (Eds.), Vol. 9291. Springer, 199–216. https://doi.org/10.1007/978-3-662-48288-9_12

Manuel V. Hermenegildo and Jens Palsberg (Eds.). 2010. *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM. http://dl.acm.org/citation.cfm?id=1706299

Kohei Honda, Martin Berger, and Nobuko Yoshida. 2006. Descriptive and Relative Completeness of Logics for Higher-Order Functions. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II (Lecture Notes in Computer Science)*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.), Vol. 4052. Springer, 360–371. https://doi.org/10.1007/11787006_31

Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. 2011. HMC: Verifying Functional Programs Using Abstract Interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 470–485. https://doi.org/10.1007/978-3-642-22110-1_38

Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 416–428. https://doi.org/10.1145/1480881.1480933

Naoki Kobayashi and C.-H. Luke Ong. 2009. A Type System Equivalent to the Modal Mu-Calculus Model Checking of Higher-Order Recursion Schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*. IEEE Computer Society, 179–188. https://doi.org/10.1109/LICS.2009.29

Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. https://doi.org/10.1145/1993498.1993525

Eric Koskinen and Tachio Terauchi. 2014. Local temporal reasoning. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 59:1–59:10. https://doi.org/10.1145/2603088.2603138

Takuya Kuwahara, Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. 2015. Predicate Abstraction and CEGAR for Disproving Termination of Higher-Order Functional Programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Pasareanu (Eds.), Vol. 9207. Springer, 287–303. https://doi.org/10.1007/978-3-319-21668-3_17

Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 392–411. https://doi.org/10.1007/978-3-642-54833-8_21

Akihiro Murase, Tachio Terauchi, Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2016. Temporal verification of higher-order functional programs, See [Bodík and Majumdar 2016], 57–68. https://doi.org/10.1145/2837614.2837667

Ernst-Rüdiger Olderog. 1984. Correctness of Programs with Pascal-Like Procedures without Global Variables. *Theor. Comput. Sci.* 30 (1984), 49–90. https://doi.org/10.1016/0304-3975(84)90066-5

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. https://doi.org/10.1109/LICS.2006.38

C.-H. Luke Ong and Steven J. Ramsay. 2011. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 587–598. https://doi.org/10.1145/1926385.1926453

Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-threaded Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science)*, Cormac Flanagan and Barbara König (Eds.), Vol. 7214. Springer, 237–251. https://doi.org/10.1007/978-3-642-28756-5_17

Bernhard Reus and Thomas Streicher. 2011. Relative Completeness for Logics of Functional Programs. In *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings (LIPIcs)*, Marc Bezem (Ed.), Vol. 12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 470–480. https://doi.org/10.4230/LIPIcs.CSL.2011.470

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. https://doi.org/10.1145/1375581.1375602

Sharon Shoham and Orna Grumberg. 2004. Monotonic Abstraction-Refinement for CTL. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science)*, Kurt Jensen and Andreas Podelski (Eds.), Vol. 2988. Springer, 546–560. https://doi.org/10.1007/978-3-540-24730-2_40

Sharon Shoham and Orna Grumberg. 2007. A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. *ACM Trans. Comput. Log.* 9, 1 (2007), 1. https://doi.org/10.1145/1297658.1297659

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 266–278. https://doi.org/10.1145/2034773.2034811

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F, See [Bodík and Majumdar 2016], 256–270. https://doi.org/10.1145/2837614.2837655

Tachio Terauchi. 2010. Dependent types from counterexamples, See [Hermenegildo and Palsberg 2010], 119–130. https://doi.org/10.1145/1706299.1706315

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent type inference with interpolants. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 7-9, 2009, Coimbra, Portugal*, António Porto and Francisco Javier López-Fraguas (Eds.). ACM, 277–288. https://doi.org/10.1145/1599410.1599445

Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2017. Relatively Complete Refinement Type System for Verification of Higher-Order Non-Deterministic Programs. Extended version, available from http://www.cs.tsukuba.ac.jp/~uhiro/.

Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. 2013. Automating relatively complete verification of higher-order functional programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 75–86. https://doi.org/10.1145/2429069.2429081

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Hongwei Xi. 2001. Dependent Types for Program Termination Verification. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 231–242. https://doi.org/10.1109/LICS.2001.932500

He Zhu and Suresh Jagannathan. 2013. Compositional and Lightweight Dependent Type Inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.), Vol. 7737. Springer, 295–314. https://doi.org/10.1007/978-3-642-35873-9_19

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 400–411. https://doi.org/10.1145/2784731.2784766