arXiv:1505.02878v2 [cs.PL] 16 May 2015

Refinement Type Inference via Horn Constraint Optimization

Kodai Hashimoto and Hiroshi Unno

University of Tsukuba
{kodai, uhiro}@logic.cs.tsukuba.ac.jp

Abstract. We propose a novel method for inferring refinement types of higher-order functional programs. The main advantage of the proposed method is that it can infer maximally preferred (i.e., Pareto optimal) refinement types with respect to a user-specified preference order. The flexible optimization of refinement types enabled by the proposed method paves the way for interesting applications, such as inferring most-general characterization of inputs for which a given program satisfies (or violates) a given safety (or termination) property. Our method reduces such a type optimization problem to a Horn constraint optimization problem by using a new refinement type system that can flexibly reason about non-determinism in programs. Our method then solves the constraint optimization problem by repeatedly improving a current solution until convergence via template-based invariant generation. We have implemented a prototype inference system based on our method, and obtained promising results in preliminary experiments.

1 Introduction

Refinement types [6, 20] have been applied to safety verification of higher-order functional programs. Some existing tools [9, 10, 15–19] enable fully automated verification by refinement type inference based on invariant generation techniques such as abstract interpretation, predicate abstraction, and CEGAR. The goal of these tools is to infer refinement types precise enough to verify a given safety specification. Therefore, types inferred by these tools are often too specific to the particular specification, and hence have limited applications.

To remedy the limitation, we propose a novel refinement type inference method that can infer maximally preferred (i.e., Pareto optimal) refinement types with respect to a user-specified preference order. For example, let us consider the following summation function (in OCaml syntax)

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

A refinement type of `sum` is of the form $(x : \{x : \texttt{int} \mid P(x)\}) \to \{y : \texttt{int} \mid Q(x, y)\}$. Here, $P(x)$ and $Q(x, y)$ respectively represent pre and post conditions of `sum`. Note that the postcondition $Q(x, y)$ can refer to the argument x as well as the return value y. Suppose that we want to infer a maximally-weak predicate for P and maximally-strong predicate for Q within a given underlying theory. Our method allows us to specify such preferences as the following constraints for type optimization

$$\mathit{maximize}(P), \qquad \mathit{minimize}(Q).$$

Here, $\mathit{maximize}(P)$ (resp. $\mathit{minimize}(Q)$) means that the set of the models of $P(x)$ (resp. $Q(x, y)$) should be maximized (resp. minimized). Our method then infers a Pareto optimal refinement type with respect to the given preferences.

In general, however, this kind of multi-objective optimization involves a trade-off among the optimization constraints. In the above example, P may not be weakened without also weakening Q. Hence, there often exist multiple optima. Actually, all the following are Pareto optimal refinement types of `sum`.[1]

$$(x : \{x : \texttt{int} \mid x = 0\}) \to \{y : \texttt{int} \mid y = x\} \tag{1}$$
$$(x : \{x : \texttt{int} \mid \texttt{true}\}) \to \{y : \texttt{int} \mid y \geq 0\} \tag{2}$$
$$(x : \{x : \texttt{int} \mid x < 0\}) \to \{y : \texttt{int} \mid \texttt{false}\} \tag{3}$$

Our method further allows us to specify a priority order on the predicate variables P and Q. If P is given a higher priority over Q (we write $P \sqsubset Q$), our method infers the type (2), whereas we obtain the type (3) if $Q \sqsubset P$. Interestingly, (3) expresses that `sum` is non-terminating for any input $x < 0$.

The flexible optimization of refinement types enabled by our method paves the way for interesting applications, such as inferring most-general characterization of inputs for which a given program satisfies (or violates) a given safety (or termination) property. Furthermore, our method can infer an upper bound of the number of recursive calls if the program is terminating, and can find a minimal-length counterexample path if the program violates a safety property.

Internally, our method reduces such a refinement type optimization problem to a constraint optimization problem where the constraints are expressed as existentially quantified Horn clauses over predicate variables [1, 11, 19]. The constraint generation here is based on a new refinement type system that can reason about (angelic and demonic) non-determinism in programs. Our method then solves the constraint optimization problem by repeatedly improving a current solution until convergence. The constraint optimization here is based on an extension of template-based invariant generation [3, 8] to existentially quantified Horn clause constraints and prioritized multi-objective optimization.

The rest of the paper is organized as follows. Sections 2 and 3 respectively formalize our target language and its refinement type system. The applications of refinement type optimization are explained in Section 4. Section 5 formalizes Horn constraint optimization problems and the reduction from type optimization problems. Section 6 proposes our Horn constraint optimization method. Section 7 reports on a prototype implementation of our method and the results of preliminary experiments. We compare our method with related work in Section 8 and conclude the paper in Section 9.

---

[1] Here, we use quantifier-free linear arithmetic as the underlying theory and consider only atomic predicates for P and Q.

$$E[op(\widetilde{v})] \longrightarrow_D E[[\![op]\!](\widetilde{v})] \quad \text{(E-OP)}$$

$$\frac{f\ \widetilde{x} = e \in D \qquad |\widetilde{x}| = |\widetilde{v}|}{E[f\ \widetilde{v}] \longrightarrow_D E[[\widetilde{v}/\widetilde{x}]e]} \quad \text{(E-APP)}$$

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

$$E[\texttt{let}\ x = *_\forall\ \texttt{in}\ e] \longrightarrow_D E[[n/x]e] \quad \text{(E-RAND}\exists)$$

$$E[\texttt{let}\ x = *_\exists\ \texttt{in}\ e] \longrightarrow_D E[[n/x]e] \quad \text{(E-RAND}\forall)$$

$$\frac{\text{if } n = 0 \text{ then } i = 1 \text{ else } i = 2}{E[\texttt{ifz}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2] \longrightarrow_D E[e_i]} \quad \text{(E-IF)}$$

Fig. 1. The operational semantics of our language L

2 Target Language *L*

This section introduces a higher-order call-by-value functional language L, which is the target of our refinement type optimization. The syntax is defined as follows.

$$\begin{array}{rl}
\text{(programs)} & D ::= \{f_i\ \widetilde{x}_i = e_i\}_{i=1}^m \\
\text{(expressions)} & e ::= x \mid e\ v \mid n \mid op(v_1, \ldots, v_{ar(op)}) \mid \texttt{ifz}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
& \quad \mid\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ x = *_\forall\ \texttt{in}\ e \mid \texttt{let}\ x = *_\exists\ \texttt{in}\ e \\
\text{(values)} & v ::= x \mid x\ \widetilde{v} \mid n \\
\text{(eval. contexts)} & E ::= [\,] \mid E\ v \mid \texttt{let}\ x\ =\ E\ \texttt{in}\ e
\end{array}$$

Here, x and f are meta-variables ranging over variables. n and op respectively represent integer constants and operations such as $+$ and $\geq$. $ar(op)$ expresses the arity of op. We write $\widetilde{x}$ (resp. $\widetilde{v}$) for a sequence of variables (resp. values). For simplicity of the presentation, the language L has integers as the only data type. We encode Boolean values `true` and `false` respectively as non-zero integers and 0. A program is expressed as a set $\{f_i\ \widetilde{x}_i = e_i\}_{i=1}^m$ of function definitions. We define $\mathrm{dom}(\{f_i\ \widetilde{x}_i = e_i\}_{i=1}^m) = \{f_1, \ldots, f_m\}$. We assume that a value $x\ \widetilde{v}$ satisfies $1 \leq |\widetilde{v}| < ar(f)$, where $|\widetilde{v}|$ represents the length of the sequence $\widetilde{v}$.

The call-by-value operational semantics of L is given in Figure 1. Here, $[\![op]\!]$ represents the integer function denoted by op. Both expressions $\texttt{let}\ x = *_\forall\ \texttt{in}\ e$ and $\texttt{let}\ x = *_\exists\ \texttt{in}\ e$ generate a random integer n, bind x to it, and evaluate e. They are, however, interpreted differently in our refinement type system (see Section 3). These expressions are used to model external functions without definitions and non-deterministic behavior caused by external inputs (e.g., user inputs, interrupts, and so on). We write $\longrightarrow_D^*$ to denote the reflexive and transitive closure of $\longrightarrow_D$.

3 Refinement Type System for *L*

In this section, we introduce a refinement type system for L that can reason about non-determinism in programs. We then formalize refinement type optimization problems (in Section 3.1), which generalize ordinary type inference problems.

The syntax of our refinement type system is defined as follows.

$$
\begin{array}{rl}
\text{(refinement types)} & \tau ::= \{x \mid \phi\} \mid (x : \tau_1) \to \tau_2 \\
\text{(type environments)} & \Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \phi \\
\text{(formulas)} & \phi ::= t_1 \le t_2 \mid \top \mid \bot \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \\
\text{(terms)} & t ::= n \mid x \mid t_1 + t_2 \mid n \cdot t \\
\text{(predicates)} & p ::= \lambda \widetilde{x}.\phi
\end{array}
$$

An integer refinement type $\{x \mid \phi\}$ equipped with a formula ϕ for type refinement represents the type of integers x that satisfy ϕ. The scope of x is within ϕ. We often abbreviate $\{x \mid \top\}$ as `int`. A function refinement type $(x : \tau_1) \to \tau_2$ represents the type of functions that take an argument x of the type τ_1 and return a value of the type τ_2. Here, τ_2 may depend on the argument x and the scope of x is within τ_2. For example, $(x : \texttt{int}) \to \{y \mid y > x\}$ is the type of functions whose return value y is always greater than the argument x. We often write $\mathit{fvs}(\tau)$ to denote the set of free variables occurring in τ. We define $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$ and $\mathrm{dom}(\Gamma) = \{x \mid x : \tau \in \Gamma\}$.

In this paper, we adopt formulas ϕ of the quantifier-free theory of linear integer arithmetic (QFLIA) for type refinement. We write $\models \phi$ if a formula ϕ is valid in QFLIA. Formulas $\top$ and $\bot$ respectively represent the tautology and the contradiction. Note that atomic formulas $t_1 < t_2$ (resp. $t_1 = t_2$) can be encoded as $t_1 + 1 \le t_2$ (resp. $t_1 \le t_2 \wedge t_2 \le t_1$) in QFLIA.

The inference rules of our refinement type system are shown in Figure 2. Here, a type judgment $\vdash D : \Gamma$ means that a program D is well-typed under a refinement type environment Γ. A type judgment $\Gamma \vdash e : \tau$ indicates that an expression e has a refinement type τ under Γ. A subtype judgment $\Gamma \vdash \tau_1 <: \tau_2$ states that τ_1 is a subtype of τ_2 under Γ. $[\![\Gamma]\!]$ occurring in the rules ISub and Rand$\exists$ is defined by $[\![\emptyset]\!] = \top$, $[\![\Gamma, x : \{\nu \mid \phi\}]\!] = [\![\Gamma]\!] \wedge [x/\nu]\phi$, $[\![\Gamma, x : (\nu : \tau_1) \to \tau_2]\!] = [\![\Gamma]\!]$, and $[\![\Gamma, \phi]\!] = [\![\Gamma]\!] \wedge \phi$. In the rule Op, $[\![op]\!]^{\mathrm{Ty}}$ represents a refinement type of op that soundly abstracts the behavior of the function $[\![op]\!]$. For example, $[\![+]\!]^{\mathrm{Ty}} = (x : \texttt{int}) \to (y : \texttt{int}) \to \{z \mid z = x + y\}$. All the rules except Rand$\forall$ and Rand$\exists$ for random integer generation are essentially the same as the previous ones [18]. The rule Rand$\forall$ requires e to have τ for *any* randomly generated integer x. Therefore, e is type-checked against τ under a type environment that assigns `int` to x. By contrast, the rule Rand$\exists$ requires e to have τ for *some* randomly generated integer x. Hence, e is type-checked against τ under a type environment that assigns a type $\{x \mid \phi\}$ to x for some ϕ such that $\mathit{fvs}(\phi) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$ and $\models [\![\Gamma]\!] \Rightarrow \exists x.\phi$. For example, $x : \texttt{int} \vdash \texttt{let}\ y = *_\exists\ \texttt{in}\ x + y : \{r \mid r = 0\}$ is derivable because we can derive $x : \texttt{int}, y : \{y \mid y = -x\} \vdash x + y : \{r \mid r = 0\}$. Thus, our new type system allows us to reason about both angelic $*_\exists$ and demonic $*_\forall$ non-determinism in higher-order functional programs.

We now discuss properties of our new refinement type system. We can prove the following progress theorem in a standard manner.

Theorem 1 (Progress). *Suppose that we have* $\vdash D : \Gamma$, $\mathrm{dom}(\Gamma) = \mathrm{dom}(D)$, *and* $\Gamma \vdash e : \tau$. *Then, either* e *is a value or* $e \longrightarrow_D e'$ *for some* e'.

$$\frac{\begin{array}{c}\Gamma \vdash \lambda \widetilde{x}_i.e_i : \Gamma(f_i)\\ (\text{for } i \in \{1,\ldots,m\})\end{array}}{\vdash \{f_i\ \widetilde{x}_i = e_i\}_{i=1}^{m} : \Gamma}\ (\textsc{Prog}) \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1\\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad x \notin \mathit{fvs}(\tau_2)\end{array}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ (\textsc{Let})$$

$$\frac{\Gamma(x) = \{\nu \mid \phi\}}{\Gamma \vdash x : \{\nu \mid \nu = x\}}\ (\textsc{IVar}) \qquad \frac{\Gamma, x : \texttt{int} \vdash e : \tau \qquad x \notin \mathit{fvs}(\tau)}{\Gamma \vdash \texttt{let}\ x = *_\forall\ \texttt{in}\ e : \tau}\ (\textsc{Rand}\forall)$$

$$\frac{\Gamma(x) = (\nu : \tau_1) \to \tau_2}{\Gamma \vdash x : (\nu : \tau_1) \to \tau_2}\ (\textsc{FVar})$$

$$\frac{\begin{array}{c}\mathit{fvs}(\phi) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}\\ \models [\![\Gamma]\!] \Rightarrow \exists x.\phi\\ \Gamma, x : \{x \mid \phi\} \vdash e : \tau \qquad x \notin \mathit{fvs}(\tau)\end{array}}{\Gamma \vdash \texttt{let}\ x = *_\exists\ \texttt{in}\ e : \tau}\ (\textsc{Rand}\exists)$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : (x : \tau_1) \to \tau_2}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma, v = 0 \vdash e_1 : \tau\\ \Gamma, v \neq 0 \vdash e_2 : \tau\end{array}}{\Gamma \vdash \texttt{ifz}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}\ (\textsc{If})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : (x : \tau_1) \to \tau_2\\ \Gamma \vdash v : \tau_1\end{array}}{\Gamma \vdash e\ v : [v/x]\tau_2}\ (\textsc{App})$$

$$\Gamma \vdash n : \{\nu \mid \nu = n\}\ (\textsc{Int}) \qquad \frac{\begin{array}{c}[\![op]\!]^{\mathrm{Ty}} = (x_1 : \tau_1) \to \cdots \to (x_m : \tau_m) \to \tau\\ \sigma_j = [v_1/x_1, \ldots, v_j/x_j]\\ \Gamma \vdash v_i : \sigma_{i-1}\tau_i \quad (\text{for } i \in \{1,\ldots,m\})\end{array}}{\Gamma \vdash op(v_1, \ldots, v_m) : \sigma_m \tau}\ (\textsc{Op})$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash e : \tau}\ (\textsc{Sub})$$

$$\frac{\models [\![\Gamma]\!] \wedge \phi_1 \Rightarrow \phi_2}{\Gamma \vdash \{\nu \mid \phi_1\} <: \{\nu \mid \phi_2\}}\ (\textsc{ISub}) \qquad \frac{\begin{array}{c}\Gamma \vdash \tau_1' <: \tau_1\\ \Gamma, \nu : \tau_1' \vdash \tau_2 <: \tau_2'\end{array}}{\Gamma \vdash (\nu : \tau_1) \to \tau_2 <: (\nu : \tau_1') \to \tau_2'}\ (\textsc{FSub})$$

Fig. 2. The inference rules of our refinement type system

We can also show the substitution lemma and the type preservation theorem in a similar manner to [18].

Lemma 1 (Substitution). *If $\Gamma \vdash v : \tau'$ and $\Gamma, x : \tau', \Gamma' \vdash e : \tau$, then $\Gamma, [v/x]\Gamma' \vdash [v/x]e : [v/x]\tau$.*

Theorem 2 (Preservation). *Suppose that we have $\vdash D : \Gamma$ and $\Gamma \vdash e : \tau$. If e is of the form $\texttt{let}\ x = *_\exists\ \texttt{in}\ e_0$, then we get $\Gamma \vdash e' : \tau$ for* some *e' such that $e \longrightarrow_D e'$. Otherwise, we get $\Gamma \vdash e' : \tau$ for* any *e' such that $e \longrightarrow_D e'$.*

Proof. We prove the theorem by induction on the derivation of $\Gamma \vdash e : \tau$. We only show the case for the rule $\textsc{Rand}\exists$ below. The other cases are similar to [18]. By $\textsc{Rand}\exists$, we have $e = \texttt{let}\ x = *_\exists\ \texttt{in}\ e_0$, $\mathit{fvs}(\phi) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$, $\models [\![\Gamma]\!] \Rightarrow \exists x.\phi$, $\Gamma, x : \{x \mid \phi\} \vdash e_0 : \tau$, and $x \notin \mathit{fvs}(\tau)$. It then follows from $\models [\![\Gamma]\!] \Rightarrow \exists x.\phi$ that there is an integer n such that $\models [\![\Gamma]\!] \wedge x = n \Rightarrow \phi$. By the rule E-$\textsc{Rand}\exists$, we get $e \longrightarrow_D [n/x]e_0 = e'$. By the rules $\textsc{Int}$ and $\textsc{Sub}$, we obtain $\Gamma \vdash n : \{x \mid \phi\}$. Thus, we get $\Gamma \vdash e' : \tau$ by Lemma 1, $\Gamma, x : \{x \mid \phi\} \vdash e_0 : \tau$, and $x \notin \mathit{fvs}(\tau)$. □

3.1 Refinement Type Optimization Problems

We now define refinement type optimization problems, which generalize refinement type inference problems addressed by previous work [9, 10, 15–19].

We first introduce the notion of *refinement type templates*. A refinement type template of a function f is the refinement type obtained from the ordinary ML-style type of f by replacing each base type `int` with an integer refinement type $\{\nu \mid P(\widetilde{x}, \nu)\}$ for some fresh predicate variable P that represents an unknown predicate to be inferred, and each function type $T_1 \to T_2$ with a (dependent) function refinement type $(x : \tau_1) \to \tau_2$. For example, from an ML-style type $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int} \to \mathtt{int}$, we obtain the following template.

$$\begin{aligned}(f :\ &(x_1 : \{x_1 \mid P_1(x_1)\}) \to \{x_2 \mid P_2(x_1, x_2)\}) \to \\ &(x_3 : \{x_3 \mid P_3(x_3)\}) \to \{x_4 \mid P_4(x_3, x_4)\}\end{aligned}$$

Note here that the first argument f is not passed as an argument to P_3 and P_4 because f is of a function type and never occurs in QFLIA formulas for type refinement. A refinement type template of a program D with $\mathrm{dom}(D) = \{f_1, \ldots, f_m\}$ is the refinement type environment $\Gamma_D = f_1 : \tau_1, \ldots, f_m : \tau_m$, where each τ_i is the refinement type template of f_i. We write $pvs(\Gamma_D)$ for the set of predicate variables that occur in Γ_D. A *predicate substitution* θ for Γ_D is a map from each $P \in pvs(\Gamma_D)$ to a closed predicate $\lambda x_1, \ldots, x_{ar(P)}.\phi$, where $ar(P)$ represents the arity of P. We write $\theta\Gamma_D$ to denote the application of a substitution θ to Γ_D. We also write $\mathrm{dom}(\theta)$ to represent the domain of θ.

We can define ordinary refinement type inference problems as follows.

Definition 1 (Refinement Type Inference). *A* refinement type inference problem *of a program D is a problem to find a predicate substitution θ such that $\vdash D : \theta\Gamma_D$.*

We now generalize refinement type inference problems to optimization problems.

Definition 2 (Refinement Type Optimization). *Let D be a program, $\prec$ be a strict partial order on predicate substitutions, and $\Theta = \{\theta \mid\ \vdash D : \theta\Gamma_D\}$. A predicate substitution $\theta \in \Theta$ is called* Pareto optimal with respect to $\prec$ *if there is no $\theta' \in \Theta$ such that $\theta' \prec \theta$. A* refinement type optimization problem $(D, \prec)$ *is a problem to find a Pareto optimal substitution $\theta \in \Theta$ with respect to $\prec$.*

In the remainder of the paper, we will often consider type optimization problems extended with user-specified constraints and/or templates for some predicate variables (see Section 4 for examples and Section 5 for formal definitions).

The above definition of type optimization problems is abstract in the sense that $\prec$ is only required to be a strict partial order on predicate substitutions. We below introduce an example concrete order, which is already explained informally in Section 1 and adopted in our prototype implementation described in Section 7. The order is defined by two kinds of optimization constraints: the optimization direction (i.e. minimize/maximize) and the priority order on predicate variables.

Definition 3. *Suppose that*

- *$\mathcal{P} = \{P_1, \ldots, P_m\}$ is a subset of $pvs(\Gamma_D)$,*

- *ρ is a map from each predicate variable in $\mathcal{P}$ to an optimization direction d that is either $\uparrow$ (for maximization) or $\downarrow$ (for minimization), and*
- *$\sqsubset$ is a strict total order on $\mathcal{P}$ that expresses the priority.*[2] *We below assume that $P_1 \sqsubset \cdots \sqsubset P_m$.*

We define a strict partial order $\prec_{(\rho,\sqsubset)}$ on predicate substitutions that respects ρ and $\sqsubset$ as the following lexicographic order:

$$\theta_1 \prec_{(\rho,\sqsubset)} \theta_2 \iff \exists i \in \{1,\ldots,m\} \,.\, \theta_1(P_i) \prec_{\rho(P_i)} \theta_2(P_i) \wedge \forall j < i.\, \theta_1(P_j) \equiv_{\rho(P_j)} \theta_2(P_j)$$

Here, a strict partial order $\prec_d$ and an equivalence relation $\equiv_d$ on predicates are defined as follows.

- $p_1 \prec_d p_2 \iff p_1 \preceq_d p_2 \wedge p_2 \not\preceq_d p_1$,
- $p_1 \equiv_d p_2 \iff p_1 \preceq_d p_2 \wedge p_2 \preceq_d p_1$,
- $\lambda\widetilde{x}.\phi_1 \preceq_\uparrow \lambda\widetilde{x}.\phi_2 \iff \models \phi_2 \Rightarrow \phi_1$, *and* $\lambda\widetilde{x}.\phi_1 \preceq_\downarrow \lambda\widetilde{x}.\phi_2 \iff \models \phi_1 \Rightarrow \phi_2$.

Example 1. Recall the function `sum` and its type template with the predicate variables P, Q in Section 1. Let us consider optimization constraints $\rho(P) = \uparrow$, $\rho(Q) = \downarrow$, and $P \sqsubset Q$, and predicate substitutions

- $\theta_1 = \{P \mapsto \lambda x.\, x = 0,\ Q \mapsto \lambda x, y.\, y = x\}$,
- $\theta_2 = \{P \mapsto \lambda x.\, \top,\ Q \mapsto \lambda x, y.\, y \geq 0\}$, and
- $\theta_3 = \{P \mapsto \lambda x.\, x < 0,\ Q \mapsto \lambda x, y.\, \bot\}$.

We then have $\theta_2 \prec_{(\rho,\sqsubset)} \theta_1$ and $\theta_2 \prec_{(\rho,\sqsubset)} \theta_3$, because $(\lambda x.\, \top) \prec_\uparrow (\lambda x.\, x = 0)$ and $(\lambda x.\, \top) \prec_\uparrow (\lambda x.\, x < 0)$. □

4 Applications of Refinement Type Optimization

In this section, we present applications of refinement type optimization to the problems of proving safety (in Section 4.1) and termination (in Section 4.3), and disproving safety (in Section 4.4) and termination (in Section 4.2) of programs in the language L. In particular, we discuss precondition inference, namely, inference of most-general characterization of inputs for which a given program satisfies (or violates) a given safety (or termination) property.

4.1 Proving Safety

We explain how to formalize, as a type optimization problem, a problem of inferring maximally-weak precondition under which a given program satisfies a given postcondition. For example, let us consider the following terminating version of `sum`.

[2] If $\sqsubset$ were partial, the relation $\prec_{(\rho,\sqsubset)}$ defined shortly would not be a strict partial order. Our implementation described in Section 7 uses topological sort to obtain a strict total order $\sqsubset$ from a user-specified partial one.

```
let rec sum' x = if x <= 0 then 0 else x + sum' (x-1)
```

In our framework, a problem to infer a maximally-weak precondition on the argument x for a postcondition $x = \mathtt{sum}'\ x$ is expressed as a type optimization problem to infer $\mathtt{sum}'$'s refinement type of the form $(x : \{x \mid P(x)\}) \to \{y \mid x = y\}$ under an optimization constraint *maximize*(P). Our type optimization method described in Sections 5.2 and 6 infers the following type.

$$(x : \{x \mid 0 \leq x \leq 1\}) \to \{y \mid x = y\}$$

This type says that the postcondition holds if the actual argument x is 0 or 1.

Example 2 (Hihger-Order Function). For an example of a higher-order function, consider the following.

```
let rec repeat f n e = if n<=0 then e else repeat f (n-1) (f e)
```

By inferring `repeat`'s refinement type of the form

$$(f : (x : \{x \mid P_1(x)\}) \to \{y \mid P_2(x, y)\}) \to (n : \mathtt{int}) \to (e : \{e \mid P_3(n, e)\}) \to \{r \mid r \geq 0\}$$

under optimization constraints $\rho(P_1) =\downarrow$, $\rho(P_2) = \rho(P_3) =\uparrow$, and $P_3 \sqsubset P_2 \sqsubset P_1$, our type optimization method obtains

$$(f : (x : \{x \mid x \geq 0\}) \to \{y \mid y \geq 0\}) \to (n : \mathtt{int}) \to (e : \{e \mid e \geq 0\}) \to \{r \mid r \geq 0\}$$

Thus, type optimization can be applied to infer maximally-weak refinement types of (possibly higher-order) arguments that are sufficient for the function to satisfy a given postcondition. □

4.2 Disproving Termination

In a similar manner to Section 4.1, we can apply type optimization to the problems of inferring maximally-weak precondition for a given program to violate the termination property. For example, consider the function `sum` in Section 1. For disproving termination of `sum`, we infer `sum`'s refinement type of the form $(x : \{x \mid P(x)\}) \to \{y \mid \bot\}$ under an optimization constraint *maximize*(P). Our type optimization method infers the following type.

$$(x : \{x \mid x < 0\}) \to \{y \mid \bot\}$$

The type expresses the fact that no value is returned by `sum` (i.e., `sum` is non-terminating) if the actual argument x satisfies $x < 0$.

Example 3 (Non-Deterministic Function). For an example of non-deterministic function, let us consider a problem of disproving termination of the following.

```
let rec f x = let n = read_int () in if n<0 then x else f x
```

Here, `read_int ()` is a function to get an integer value from the user and is modeled as $*_\exists$ in our language L. Note that the termination of `f` does not depend on the argument `x` but user inputs `n`. Actually, our type optimization method successfully disproves termination of `f` by inferring a refinement type $(x : \mathtt{int}) \to \{y \mid \bot\}$ for `f` and $\{n \mid n \geq 0\}$ for the user inputs `n`. This means that `f` is non-terminating if the user always inputs some non-negative integer. □

4.3 Proving Termination

Refinement type optimization can also be applied to bounds analysis for inferring upper bounds of the number of recursive calls. Our bounds analysis for functional programs is inspired by a program transformation approach to bounds analysis for imperative programs [7, 8]. Let us consider `sum` in Section 1. By inserting additional parameters i and c to the definition of `sum`, we obtain

```
let rec sum_t x i c = if x=0 then 0 else x + sum_t (x-1) i (c+1)
```

Here, i and c respectively represent the initial value of the argument x and the number of recursive calls so far. For proving termination of `sum`, we infer `sum_t`'s refinement type of the form

$$(x : \{x \mid P(x)\}) \to (i : \mathtt{int}) \to (c : \{c \mid \mathit{Inv}(x, i, c)\}) \to \mathtt{int}$$

under optimization constraints $\mathit{maximize}(P)$, $\mathit{minimize}(\mathit{Bnd})$, $P \sqsubset \mathit{Bnd}$, and additional constraints on the predicate variables $P, \mathit{Bnd}, \mathit{Inv}$

$$\forall x, i, c.\ (\mathit{Inv}(x, i, c) \Leftarrow c = 0 \wedge i = x) \tag{4}$$

$$\forall x, i, c.\ (\mathit{Bnd}(i, c) \Leftarrow P(x) \wedge \mathit{Inv}(x, i, c)) \tag{5}$$

Here, $\mathit{Bnd}(i, c)$ is intended to represent the bounds of the number c of recursive calls of `sum` with respect to the initial value i of the argument x. We therefore assume that $\mathit{Bnd}(i, c)$ is of the form $0 \leq c \leq k_0 + k_1 \cdot i$, where k_0, k_1 represent unknown coefficients to be inferred. The constraint (4) is necessary to express the meaning of the inserted parameters i and c. The constraint (5) is also necessary to ensure that the bounds $\mathit{Bnd}(i, c)$ is implied by a precondition $P(x)$ and an invariant $\mathit{Inv}(x, i, c)$ of `sum`. Our type optimization method then infers

$$(x : \{x \mid x \geq 0\}) \to (i : \mathtt{int}) \to (c : \{c \mid x \leq i \wedge i = x + c\}) \to \mathtt{int}$$

and $\mathit{Bnd}(i, c) \equiv 0 \leq c \leq i$. Thus, we can conclude that `sum` is terminating for any input $x \geq 0$ because the number c of recursive calls is bounded from above by the initial value i of the argument x.

Interestingly, we can infer a precondition for minimizing the number of recursive calls of `sum` by replacing the priority constraint $P \sqsubset \mathit{Bnd}$ with $\mathit{Bnd} \sqsubset P$ and adding an additional constraint $\exists x.P(x)$ (to avoid a meaningless solution $P(x) \equiv \bot$). In fact, our type optimization method obtains

$$(x : \{x \mid x = 0\}) \to (i : \mathtt{int}) \to (c : \{c \mid c = 0\}) \to \mathtt{int}$$

and $\mathit{Bnd}(i, c) \equiv c = 0$. Therefore, we can conclude that the minimum number of recursive calls is 0 when the actual argument x is 0.

We expect that our bounds analysis for functional programs can further be extended to infer non-linear upper bounds by adopting ideas from an elaborate transformation for bounds analysis of imperative programs [7].

4.4 Disproving Safety

We can use the same technique in Section 4.3 to infer maximally-weak precondition for a given program to violate a given postcondition. For example, let us consider again the function `sum`. A problem to infer a maximally-weak precondition on the argument x for violating a postcondition `sum` $x \geq 2$ can be reduced to a problem to infer `sum_t`'s refinement type of the form

$$(x : \{x \mid P(x)\}) \to (i : \texttt{int}) \to (c : \{c \mid Inv(x, i, c)\}) \to \{y \mid \neg(y \geq 2)\}$$

under the same constraints for bounds analysis in Section 4.3. The refinement type optimization method then obtains

$$(x : \{x \mid 0 \leq x \leq 1\}) \to (i : \texttt{int}) \to (c : \{c \mid 0 \leq x \wedge i = x + c\}) \to \{y \mid \neg(y \geq 2)\}$$

and $Bnd(i, c) \equiv 0 \leq c \leq i$. This result says that if the actual argument x is 0 or 1, then `sum` terminates and returns some integer y that violates $y \geq 2$. In other words, $x = 0, 1$ are counterexamples to the postcondition `sum` $x \geq 2$.

We can instead find a minimal-length counterexample path[3] to the postcondition `sum` $x \geq 2$ by just replacing the priority constraint $P \sqsubset Bnd$ with $Bnd \sqsubset P$ and adding an additional constraint $\exists x.P(x)$. Our type optimization method then infers

$$(x : \{x \mid x = 0\}) \to (i : \texttt{int}) \to (c : \{c \mid 0 \leq x \wedge i = x + c\}) \to \{y \mid \neg(y \geq 2)\}$$

and $Bnd(i, c) \equiv c = 0$. From the result, we can conclude that a minimal-length counterexample path is obtained when the actual argument x is 0.

5 Horn Constraint Optimization and Reduction from Refinement Type Optimization

We reduce refinement type optimization problems into constraint optimization problems subject to existentially-quantified Horn clauses [1, 11, 19]. We first formalize Horn constraint optimization problems (in Section 5.1) and then explain the reduction (in Section 5.2).

5.1 Horn Constraint Optimization Problems

Existentially-Quantified Horn Clause Constraint Sets ($\exists$HCCSs) over QFLIA are defined as follows.

$$\begin{array}{rl}
(\exists\text{HCCSs}) & \mathcal{H} ::= \{hc_1, \ldots, hc_m\} \\
(\text{Horn clauses}) & hc ::= h \Leftarrow b \\
(\text{heads}) & h ::= P(\widetilde{t}) \mid \phi \mid \exists \widetilde{x}.(P(\widetilde{t}) \wedge \phi) \\
(\text{bodies}) & b ::= P_1(\widetilde{t}_1) \wedge \ldots \wedge P_m(\widetilde{t}_m) \wedge \phi
\end{array}$$

[3] Here, minimality is with respect to the number of recursive calls within the path.

We write $pvs(\mathcal{H})$ for the set of predicate variables that occur in $\mathcal{H}$.

A *predicate substitution* θ for an $\exists$HCCS $\mathcal{H}$ is a map from each $P \in pvs(\mathcal{H})$ to a closed predicate $\lambda x_1, \ldots, x_{ar(P)}.\phi$. We write $\Theta_{\mathcal{H}}$ for the set of predicate substitutions for $\mathcal{H}$. We call a substitution θ is a *solution* of $\mathcal{H}$ if for each $hc \in \mathcal{H}$, $\models \theta hc$. For a subset $\Theta \subseteq \Theta_{\mathcal{H}}$, we call a substitution $\theta \in \Theta$ is a *Θ-restricted solution* if θ is a solution of $\mathcal{H}$. Our constraint optimization method described in Section 6 is designed to find a Θ-restricted solution for some Θ consisting of substitutions that map each predicate variable to a predicate with a bounded number of conjunctions and disjunctions. In particular, we often use

$$\Theta_{atom} = \left\{ P \mapsto \lambda x_1, \ldots, x_{ar(P)}.n_0 + \Sigma_{i=1}^{ar(P)} n_i \cdot x_i \geq 0 \mid P \in pvs(\mathcal{H}) \right\}$$

consisting of atomic predicate substitutions.

Example 4. Recall the function `sum` and the predicate substitutions $\theta_1, \theta_2, \theta_3$ in Example 1. Our method reduces a type optimization problem for `sum` into a constraint optimization problem for the following HCCS $\mathcal{H}_{\mathtt{sum}}$ (the explanation of the reduction is deferred to Section 5.2).

$$\left\{ \begin{array}{l} Q(x,0) \Leftarrow P(x) \wedge x = 0,\ P(x-1) \Leftarrow P(x) \wedge x \neq 0, \\ Q(x, x+y) \Leftarrow P(x) \wedge Q(x-1, y) \wedge x \neq 0 \end{array} \right\}$$

Here, θ_1 is a solution of $\mathcal{H}_{\mathtt{sum}}$, and θ_2 and θ_3 are Θ_{atom}-restricted solutions of $\mathcal{H}_{\mathtt{sum}}$. If we fix $Q(x,y) \equiv \bot$ (i.e., infer `sum`'s type of the form $(x : \{x \mid P(x)\}) \to \{y \mid \bot\}$) for disproving termination of `sum` as in Section 4.2, we obtain the following HCCS $\mathcal{H}_{\mathtt{sum}}^{\bot}$.

$$\{\bot \Leftarrow P(x) \wedge x = 0,\ P(x-1) \Leftarrow P(x) \wedge x \neq 0\}$$

$\mathcal{H}_{\mathtt{sum}}^{\bot}$ has, for example, Θ_{atom}-restricted solutions $\{P \mapsto \lambda x.x < 0\}$ and $\{P \mapsto \lambda x.x < -100\}$. $\square$

We now define Horn constraint optimization problems for $\exists$HCCSs.

Definition 4. *Let $\mathcal{H}$ be an $\exists$HCCS and $\prec$ be a strict partial order on predicate substitutions. A solution θ of $\mathcal{H}$ is called* Pareto optimal with respect to $\prec$ *if there is no solution θ' of $\mathcal{H}$ such that $\theta' \prec \theta$. A* Horn constraint optimization problem $(\mathcal{H}, \prec)$ *is a problem to find a Pareto optimal solution θ with respect to $\prec$. A* Θ-restricted Horn constraint optimization problem *is a Horn constraint optimization problem with the notion of solutions replaced by Θ-restricted solutions.*

Example 5. Recall $\mathcal{H}_{\mathtt{sum}}$ and its solutions θ_1,θ_2,θ_3 in Example 1. Let us consider a Horn constraint optimization problem $(\mathcal{H}_{\mathtt{sum}}, \prec_{(\rho,\sqsubset)})$ where $\rho(P) = \uparrow$, $\rho(Q) = \downarrow$, and $Q \sqsubset P$. We have $\theta_3 \prec_{(\rho,\sqsubset)} \theta_1$ and $\theta_3 \prec_{(\rho,\sqsubset)} \theta_2$. In fact, θ_3 is a Pareto optimal solution of $\mathcal{H}_{\mathtt{sum}}$ with respect to $\prec_{(\rho,\sqsubset)}$. $\square$

In general, an ∃HCCS $\mathcal{H}$ may not have a Pareto optimal solution with respect to $\prec_{(\rho,\sqsubset)}$ even though $\mathcal{H}$ has a solution. For example, consider a Horn constraint optimization problem $(\mathcal{H}_{\mathtt{sum}}, \prec_{(\rho,\sqsubset)})$ where $\rho(P) = \uparrow$, $\rho(Q) = \downarrow$, and $P \sqsubset Q$. Because the semantically optimal solution $Q(x,y) \equiv y = \frac{x(x+1)}{2}$ is not expressible in QFLIA, it must be approximated, for example, as $Q(x,y) \equiv y \geq 0 \wedge y \geq x \wedge y \geq 2x - 1$. The approximated solution, however, is not Pareto optimal because we can always get a better approximation like $Q(x,y) \equiv y \geq 0 \wedge y \geq x \wedge y \geq 2x - 1 \wedge y \geq 3x - 3$ if we use more conjunctions.

We can, however, show that an ∃HCCS has a Θ_{atom}-restricted Pareto optimal solution with respect to $\prec_{(\rho,\sqsubset)}$ if it has a Θ_{atom}-restricted solution. For the above example, θ_2 in Example 1 is a Θ_{atom}-restricted Pareto optimal solution.

Lemma 2. *Suppose that an ∃HCCS $\mathcal{H}$ has a Θ_{atom}-restricted solution and for any P such that $\rho(P) = \downarrow$, P is not existentially quantified in $\mathcal{H}$. It then follows that $\mathcal{H}$ has a Θ_{atom}-restricted Pareto optimal solution with respect to $\prec_{(\rho,\sqsubset)}$.*

Proof Sketch. We prove the lemma by contradiction. Suppose that $\mathcal{H}$ has a Θ_{atom}-restricted solution but no Pareto optimal one. It then follows that there exist an infinite descending chain $\theta_1 \succ_{(\rho,\sqsubset)} \theta_2 \succ_{(\rho,\sqsubset)} \dots$ of Θ_{atom}-restricted solutions and a predicate variable P such that

- $\forall i \geq 1.\ \theta_i(P) \succ_{\rho(P)} \theta_{i+1}(P) \wedge \forall Q \sqsubset P.\ \theta_i(Q) \equiv_{\rho(Q)} \theta_{i+1}(Q)$ and
- no Θ_{atom}-restricted solution is a lower bound of the chain.

The key observations here are that the half-spaces represented by $\theta_1(P), \theta_2(P), \dots$ are parallel, and for some $k > 0$ that depends on $2^{ar(P)}$ and the largest absolute value of coefficients in $\theta_1(P)$, the distance d_i between $\theta_i(P)$ and $\theta_{i+k}(P)$ are $d_i > 1$ for all $i \geq 1$, because of the strictness of $\succ_{\rho(P)}$ and the discreteness of integers. By continuity of $\mathcal{H}$, $\mathcal{H}$ has a Θ_{atom}-restricted solution θ such that $\theta(P) \equiv \lambda\widetilde{x}.\top$ if $\rho(P) = \uparrow$ and $\theta(P) \equiv \lambda\widetilde{x}.\bot$ if $\rho(P) = \downarrow$, and $\forall Q \sqsubset P.\ \theta(Q) \equiv_{\rho(Q)} \theta_1(Q)$. θ is obviously a lower bound of the chain. Thus, a contradiction is obtained. □

5.2 Reduction from Refinement Type Optimization

Our method reduces a refinement type optimization problem into an Horn constraint optimization problem in a similar manner to the previous refinement type inference method [18]. Given a program D, our method first prepares a refinement type template Γ_D of D as well as, for each expression of the form `let` $x = *_\exists$ `in` e, a refinement type template $\{x \mid P(\widetilde{y}, x)\}$ of x, where P is a fresh predicate variable and $\widetilde{y}$ is the sequence of all integer variables in the scope. Our method then generates an ∃HCCS by type-checking D against Γ_D and collecting the proof obligations of the forms $[\![\Gamma]\!] \wedge \phi_1 \Rightarrow \phi_2$ and $[\![\Gamma]\!] \Rightarrow \exists\nu.\phi$ respectively from each application of the rules ISub and Rand∃. We write $Gen(D, \Gamma_D)$ to denote the ∃HCCS thus generated from D and Γ_D.

We can show the soundness of our reduction in the same way as in [18].

Theorem 3 (Soundness of Reduction). *Let $(D, \prec)$ be a refinement type optimization problem and Γ_D be a refinement type template of D. If θ is a Pareto optimal solution of $Gen(D, \Gamma_D)$, then θ is a solution of $(D, \prec)$.*

1: **procedure** Optimize($\mathcal{H}, \prec$)
2:     **match** Solve($\mathcal{H}$) **with**
3:         *Unknown* $\rightarrow$ **return** *Unknown*
4:       | *NoSol* $\rightarrow$ **return** *NoSol*
5:       | *Sol*(θ_0) $\rightarrow$
6:           $\theta := \theta_0$;
7:           **while** `true` **do**
8:               **let** $\mathcal{H}' =$ Improve$_\prec(\theta, \mathcal{H})$ **in**
9:               **match** Solve($\mathcal{H}'$) **with**
10:                   *Unknown* $\rightarrow$ **return** *Sol*(θ)
11:                 | *NoSol* $\rightarrow$ **return** *OptSol*(θ)
12:                 | *Sol*(θ') $\rightarrow \theta := \theta'$
13:           **end**

Fig. 3. Pseudo-code of the constraint optimization method for ∃HCCSs

6 Horn Constraint Optimization Method

In this section, we describe our Horn constraint optimization method for ∃HCCSs. The method repeatedly improves a current solution until convergence. The pseudo-code of the method is shown in Figure 3. The procedure Optimize for Horn constraint optimization takes a (Θ-restricted) ∃HCCS optimization problem $(\mathcal{H}, \prec)$ and returns any of the following: *Unknown* (which means the existence of a solution is unknown), *NoSol* (which means no solution exists), *Sol*(θ) (which means θ is a possibly non-Pareto optimal solution), or *OptSol*(θ) (which means θ is a Pareto optimal solution). The sub-procedure Solve for Horn constraint solving takes an ∃HCCS $\mathcal{H}$ and returns any of *Unknown*, *NoSol*, or *Sol*(θ). The detailed description of Solve is deferred to Section 6.1.

Optimize first calls Solve to find an initial solution θ_0 of $\mathcal{H}$ (line 2). Optimize returns *Unknown* if Solve returns *Unknown* (line 3) and *NoSol* if Solve returns *NoSol* (line 4). Otherwise (line 5), Optimize repeatedly improves a current solution θ starting from θ_0 until convergence (lines 6 – 13). To improve θ, we call a sub-procedure Improve$_\prec(\theta, \mathcal{H})$ for generating an ∃HCCS $\mathcal{H}'$ from $\mathcal{H}$ by adding constraints that require any solution θ' of $\mathcal{H}'$ satisfies $\theta' \prec \theta$ (line 8). Optimize then calls Solve to find a solution of $\mathcal{H}'$. If Solve returns *Unknown*, Optimize returns *Sol*(θ) as a (possibly non-Pareto optimal) solution (line 10). If Solve returns *NoSol*, it is the case that no improvement is possible, and hence the current solution θ is Pareto optimal. Thus, Optimize returns *OptSol*(θ) (line 11). Otherwise, we obtain an improved solution $\theta' \prec \theta$, and Optimize updates the current solution θ with θ' and repeats the improvement process (line 12).

Example 6. Recall $\mathcal{H}^{\perp}_{\texttt{sum}}$ in Example 4 and consider an optimization problem $(\mathcal{H}^{\perp}_{\texttt{sum}}, \prec_{(\sqsubset,\rho)})$ where $\rho(P) =\uparrow$. We below explain how Optimize($\mathcal{H}^{\perp}_{\texttt{sum}}, \prec_{(\sqsubset,\rho)}$) proceeds. First, Optimize calls the sub-procedure Solve to find an initial solution of $\mathcal{H}^{\perp}_{\texttt{sum}}$ (e.g., $\theta_0 = \{P \mapsto \lambda x.\bot\}$). Optimize then calls the sub-procedure Improve$_\prec(\theta_0, \mathcal{H}^{\perp}_{\texttt{sum}})$ and obtains an ∃HCCS $\mathcal{H}' = \mathcal{H}^{\perp}_{\texttt{sum}} \cup \{P(x) \Leftarrow \bot,\ \exists x.P(x) \wedge \neg\bot\}$. Note that $\mathcal{H}'$ requires that for any solution θ of $\mathcal{H}'$, $\theta(P) \prec_{\rho(P)} \theta_0(P) =$

$\lambda x.\bot$. Optimize then calls Solve($\mathcal{H}'$) to find an improved solution of $\mathcal{H}$ (e.g., $\theta_1 = \{P \mapsto \lambda x.x < 0\}$). In the next iteration, Optimize returns θ_1 as a Pareto optimal solution because Improve$_\prec(\theta_1, \mathcal{H}^{\bot}_{\mathtt{sum}})$ has no solution. □

We now discuss properties of the procedure Optimize under the assumption of the correctness of the sub-procedure Solve (i.e., θ is a Θ-restricted solution of $\mathcal{H}$ if Solve($\mathcal{H}$) returns $Sol(\theta)$, and $\mathcal{H}$ has no Θ-restricted solution if Solve($\mathcal{H}$) returns *NoSol*). The following theorem states the correctness of Optimize.

Theorem 4 (Correctness of the Procedure Optimize). *Let $(\mathcal{H}, \prec)$ be a Θ-restricted Horn constraint optimization problem. If* Optimize$(\mathcal{H}, \prec)$ *returns $OptSol(\theta)$ (resp. $Sol(\theta)$), θ is a Pareto optimal (resp. possibly non-Pareto optimal) Θ-restricted solution of $\mathcal{H}$ with respect to $\prec$.*

The following theorem states the termination of Optimize for Θ_{atom}-restricted Horn constraint optimization problems.

Theorem 5 (Termination of the Procedure Optimize). *Let $(\mathcal{H}, \prec_{(\sqsubset,\rho)})$ be a Θ_{atom}-restricted Horn constraint optimization problem. It then follows that* Optimize$(\mathcal{H}, \prec_{(\sqsubset,\rho)})$ *always terminates if the sub-procedure* Solve *preferentially returns solutions having smaller absolute values of coefficients.*

Proof. Recall the proof sketch of Lemma 2. Any infinite descending chain of Θ_{atom}-restricted solutions for a predicate variable P with respect to $\prec_{(\sqsubset,\rho)}$ has a limit $\lambda\widetilde{x}.\top$ if $\rho(P) =\uparrow$ and $\lambda\widetilde{x}.\bot$ if $\rho(P) =\downarrow$. Because $\lambda\widetilde{x}.\bot$ (resp. $\lambda\widetilde{x}.\top$) is expressed as an atomic predicate $\lambda\widetilde{x}. -1 \geq 0$ (resp. $\lambda\widetilde{x}.0 \geq 0$) having absolute values of coefficients not greater than 1 and the number of such predicates is finite, the limit is guaranteed to be reached in a finite number of iterations. □

6.1 Sub-Procedure Solve for Solving ∃HCCSs

The pseudo-code of the sub-procedure Solve for solving ∃HCCSs is presented in Figure 4. Here, Solve uses existing template-based invariant generation techniques based on Farkas' lemma [3, 8] and ∃HCCS solving techniques based on Skolemization [1, 11, 19]. Solve first generates a template substitution θ that maps each predicate variable in $pvs(\mathcal{H})$ to a template atomic predicate with unknown coefficients $c_0, \ldots, c_{ar(P)}$ (line 2).[4] Solve then applies θ to $\mathcal{H}$ and obtains a verification condition of the form $\exists\widetilde{c}.\forall\widetilde{x}.\exists\widetilde{y}.\phi$ without predicate variables (line 3). Solve applies Skolemization [1, 11, 19] to the condition and obtains a simplified condition of the form $\exists\widetilde{c},\widetilde{z}.\forall\widetilde{x}.\phi'$ (line 4). Solve further applies Farkas' lemma [3, 8] to eliminate the universal quantifiers and obtains a condition of the form $\exists\widetilde{c},\widetilde{z},\widetilde{w}.\phi''$ (line 5). Solve then uses an off-the-shelf SMT solver to find a satisfying assignment to ϕ'' (line 6). If such an assignment σ is found, Solve

[4] In this way, the particular code is specialized to solve Θ_{atom}-restricted Horn constraint optimization problems. To solve Θ-restricted optimization problems for other Θ, we need here to generate templates that conform to the shape of substitutions in Θ instead. Our implementation in Section 7 iteratively increases the template size.

1: **procedure** SOLVE($\mathcal{H}$)
2: **let** $\theta = \left\{ P \mapsto \lambda \widetilde{x}.c_0 + \Sigma_{i=1}^{ar(P)} c_i \cdot x_i \geq 0 \mid P \in pvs(\mathcal{H}) \right\}$ **in**
3: **let** $\exists \widetilde{c}.\forall \widetilde{x}.\exists \widetilde{y}.\phi = \exists \widetilde{c}. \bigwedge_{hc \in \mathcal{H}} \forall fvs(hc).\theta(hc)$ **in**
4: **let** $\exists \widetilde{c}, \widetilde{z}.\forall \widetilde{x}.\phi' =$ apply Skolemization to $\exists \widetilde{c}.\forall \widetilde{x}.\exists \widetilde{y}.\phi$ **in**
5: **let** $\exists \widetilde{c}, \widetilde{z}, \widetilde{w}.\phi'' =$ apply Farkas' lemma to $\exists \widetilde{c}, \widetilde{z}.\forall \widetilde{x}.\phi'$ **in**
6: **match** SMT(ϕ'') **with**
7: *Unknown* $\rightarrow$ **return** *Unknown*
8: | $Sat(\sigma) \rightarrow$ **return** $Sol(\sigma(\theta))$
9: | *Unsat* $\rightarrow$ **match** SMT($\forall \widetilde{x}.\exists \widetilde{y}.\phi$) **with**
10: *Unknown* $\rightarrow$ **return** *Unknown*
11: | $Sat(\sigma) \rightarrow$ **return** $Sol(\sigma(\theta))$
12: | *Unsat* $\rightarrow$ **return** *NoSol*

Fig. 4. Pseudo-code of the constraint solving method for ∃HCCSs based on template-based invariant generation

returns $\sigma(\theta)$ as a solution (line 8). SOLVE returns *Unknown* if the SMT solver returns *Unknown* (line 7). Otherwise (no assignment is found),[5] SOLVE uses the SMT solver again to find a satisfying assignment σ to $\forall \widetilde{x}.\exists \widetilde{y}.\phi$ (line 9). If such a σ is found, SOLVE returns $\sigma(\theta)$ as a solution (line 11). SOLVE returns *Unknown* if *Unknown* is returned (line 10) and *NoSol* if *Unsat* is returned (line 12).

Example 7. We explain how SOLVE proceeds for $\mathcal{H}'$ in Example 6. SOLVE first generates a template substitution $\theta = \{P \mapsto \lambda x.c_0 + c_1 \cdot x \geq 0\}$ with unknown coefficients c_0, c_1 and applies θ to $\mathcal{H}'$. As a result, we get a verification condition

$$\exists c_0, c_1. \left(\begin{array}{l} \forall x. \left(\begin{array}{l} (\bot \Leftarrow c_0 + c_1 \cdot x \geq 0 \;\wedge\; x = 0) \wedge \\ (c_0 + c_1 \cdot (x - 1) \geq 0 \Leftarrow c_0 + c_1 \cdot x \geq 0 \;\wedge\; x \neq 0) \end{array} \right) \wedge \\ \exists x.\; c_0 + c_1 \cdot x \geq 0 \end{array} \right)$$

By applying Farkas' lemma, we obtain

$$\exists c_0, c_1. \left(\begin{array}{l} \exists w_1, w_2, w_3 \geq 0.\; (c_0 \cdot w_1 \leq -1 \wedge c_1 \cdot w_1 + w_2 - w_3 = 0) \wedge \\ \exists w_4, w_5, w_6 \geq 0. \left(\begin{array}{l} (-1 - c_0 + c_1) \cdot w_4 + c_0 \cdot w_5 - w_6 \leq -1 \wedge \\ c_1 \cdot (-w_4 + w_5) + w_6 = 0 \end{array} \right) \wedge \\ \exists w_7, w_8, w_9 \geq 0. \left(\begin{array}{l} (-1 - c_0 + c_1) \cdot w_7 + c_0 \cdot w_8 - w_9 \leq -1 \wedge \\ c_1 \cdot (-w_7 + w_8) - w_9 = 0 \end{array} \right) \wedge \\ \exists x.\; c_0 + c_1 \cdot x \geq 0 \end{array} \right)$$

By using an SMT solver, we obtain, for example, a satisfying assignment

$$\sigma = \left\{ \begin{array}{l} c_0 \mapsto -1, c_1 \mapsto -1, w_1 \mapsto 1, w_2 \mapsto 1, w_3 \mapsto 0, \\ w_4 \mapsto 0, w_5 \mapsto 1, w_6 \mapsto 1, w_7 \mapsto 1, w_8 \mapsto 0, w_9 \mapsto 1 \end{array} \right\}$$

Thus, SOLVE returns $\sigma(\theta) = \{P \mapsto \lambda x. -1 - x \geq 0\} \equiv \theta_1$ in Example 6. □

[5] Note here that even though no assignment is found, $\mathcal{H}$ may have a Θ_{atom}-restricted solution because Farkas' lemma is not complete for QFLIA formulas [3, 8] and Skolemization of $\exists \widetilde{c}.\forall \widetilde{x}.\exists \widetilde{y}.\phi$ into $\exists \widetilde{c}, \widetilde{z}.\forall \widetilde{x}.\phi'$ here assumes that $\widetilde{y}$ are expressed as linear expressions over $\widetilde{x}$ [1, 11, 19].

Table 1. The results of a non-termination verification benchmark set used in [2,11,13]. The results for CppInv, T2-TACAS, and TNT are according to Larraz et al. [13]. The result for MoCHi is according to [11].

	Verified	TimeOut	Other
Our tool	41	27	13
CppInv [13]	70	6	5
T2-TACAS [2]	51	0	30
TNT [5]	19	3	59
MoCHi [11]	48	26	7

The following theorem states the correctness of the sub-procedure Solve.

Lemma 3 (Correctness of the Sub-Procedure Solve). *Let $\mathcal{H}$ be an $\exists$HCCS. θ is a Θ_{atom}-restricted solution of $\mathcal{H}$ if* Solve($\mathcal{H}$) *returns Sol(θ), and $\mathcal{H}$ has no Θ_{atom}-restricted solution if* Solve($\mathcal{H}$) *returns NoSol.*

Note that the sub-procedure Solve described above does not necessarily satisfy the assumption of Theorem 5. We can, however, extend Solve to satisfy the assumption by bounding the absolute values of unknown coefficients and iteratively incrementing the bounds in SMT solving.

7 Implementation and Experiments

We have implemented a prototype refinement type optimization tool based on our method. Our tool takes OCaml programs and uses Z3 [4] as the underlying SMT solver in the sub-procedure Solve. We conducted preliminary experiments for each application presented in Section 4. All the experiments were conducted on a machine with Intel Core i7-4650U 1.70GHz, 8GB of RAM.

The experimental results are summarized in Tables 1 and 2. Table 1 shows the results of an existing first-order non-termination verification benchmark set used in [2, 11, 13]. Because the original benchmark set was written in the input language of T2 (`http://mmjb.github.io/T2/`), we used an OCaml translation of the benchmark set provided by [11]. Our tool was able to successfully disprove termination of 41 programs (out of 81) in the time limit of 100 seconds. Our prototype tool was not the best but performed well compared to the state-of-the-art tools dedicated to non-termination verification.

Table 2 shows the results of maximally-weak precondition inference for proving safety and termination, and disproving safety and termination. In the column #Iterations, > 2 represents that the 3-rd iteration timed out and possibly non-Pareto optimal solution was inferred by our tool. We used non-termination (resp. termination) verification benchmarks for higher-order programs from [11] (resp. [12]). The results show that our method is also effective for safety and non-termination verification of higher-order programs. Our prototype tool, however, could be optimized further to speed up termination and non-safety verification.

Table 2. The results of maximally-weak precondition inference for proving/disproving safety/termination.

Program	Applications	#Iterations	Time (ms)
`fixpoint_nonterm` [11]	Disprove Termination	1	2,020
`fib_CPS_nonterm` [11]	Disprove Termination	1	5,023
`indirect_e` [11]	Disprove Termination	1	1,083
`indirectHO_e` [11]	Disprove Termination	1	2,434
`loopHO` [11]	Disprove Termination	1	1,642
`foldr_nonterm` [11]	Disprove Termination	1	4,904
`repeat` (Sec. 4.1)	Prove Safety	4	948
`sum_geq3`	Prove Safety	4	2,654
`append`	Prove Safety	5	20,352
`append` [12]	Prove Termination	6	26,786
`zip` [12]	Prove Termination	13	76,641
`sum`′ (Sec. 4.1)	Prove Safety	3	1,856
`sum` (Sec. 4.2)	Disprove Termination	1	174
`sum_t` (Sec. 4.3)	Prove Termination($P \sqsubset Inv$)	4	56,042
`sum_t` (Sec. 4.3)	Prove Termination($Inv \sqsubset P$)	3	6,628
`sum_t` (Sec. 4.4)	Disprove Safety($P \sqsubset Inv$)	> 2	18,009
`sum_t` (Sec. 4.4)	Disprove Safety($Inv \sqsubset P$)	> 2	16,540

8 Related Work

Type inference problems of refinement type systems [6, 20] have been intensively studied [9, 10, 15–19]. To our knowledge, this paper is the first to address type optimization problems, which generalize ordinary type inference problems. As we saw in Sections 4 and 7, this generalization enables significantly wider applications in the verification of higher-order functional programs.

For imperative programs, Gulwani et al. have proposed a template-based method to infer maximally-weak pre and maximally-strong post conditions [8]. Their method, however, cannot directly handle higher-order functional programs, (angelic and demonic) non-determinism in programs, and prioritized multi-objective optimization, which are all handled by our new method.

Internally, our method reduces a type optimization problem to a constraint optimization problem subject to an existentially quantified Horn clause constraint set (∃HCCS). Constraint *solving* problems for ∃HCCSs have been studied by recent work [1, 11, 19]. They, however, do not address constraint *optimization* problems. The goal of our constraint optimization is to maximize/minimize the set of the models for each predicate variable occurring in the given ∃HCCS. Thus, our constraint optimization problems are different from Max-SMT [14] problems whose goal is to minimize the sum of the penalty of unsatisfied clauses.

9 Conclusion

We have generalized refinement type inference problems to type optimization problems, and presented interesting applications enabled by type optimization to inferring most-general characterization of inputs for which a given functional program satisfies (or violates) a given safety (or termination) property. We have also proposed a refinement type optimization method based on template-based invariant generation. We have implemented our method and confirmed by experiments that the proposed method is promising for the applications.

References

1. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV '13*, volume 8044 of *LNCS*, pages 869–882. Springer, 2013.
2. H. Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. Proving nontermination via safety. In *TACAS '14*, volume 8413 of *LNCS*, pages 156–171. Springer, 2014.
3. M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV '03*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.
4. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
5. F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *IJCAR '12*, volume 7364 of *LNCS*, pages 225–240. Springer, 2012.
6. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI '91*, pages 268–277. ACM, 1991.
7. S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL '09*, pages 127–139. ACM, 2009.
8. S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI '08*, pages 281–292. ACM, 2008.
9. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *CAV '11*, volume 6806 of *LNCS*, pages 470–485. Springer, 2011.
10. N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI '11*, pages 222–233. ACM, 2011.
11. T. Kuwahara, R. Sato, H. Unno, and N. Kobayashi. Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In *CAV'15*, LNCS. Springer, 2015.
12. T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *ESOP '14*, volume 8410 of *LNCS*, pages 392–411. Springer, 2014.
13. D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving non-termination using max-SMT. In *CAV '14*, volume 8559 of *LNCS*, pages 779–796. Springer, 2014.
14. R. Nieuwenhuis and A. Oliveras. On SAT modulo theories and optimization problems. In *SAT '06*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.
15. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*, pages 159–169. ACM, 2008.
16. T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130. ACM, 2010.
17. H. Unno and N. Kobayashi. On-demand refinement of dependent types. In *FLOPS '08*, volume 4989 of *LNCS*, pages 81–96. Springer, 2008.

18. H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288. ACM, 2009.
19. H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL '13*, pages 75–86. ACM, 2013.
20. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.