Data-Driven Refinement Type Optimization

Yu Gu, Hiroshi Unno, and Takeshi Tsukada

This paper presents a data-driven refinement type optimization method for higher-order functional programs that manipulate algebraic data types. Refinement type optimization is an extension of refinement type inference to multi-objective optimization of criteria such as the strength/weakness of predicates to be inferred, thereby allowing user control of types inferred, especially for modular verification. Following the previous work by Hashimoto and Unno, we reduce refinement type optimization to constraint optimization. Their constraint optimization method, however, does not scale to complex constraints over non-numerical domains because it is based on Farkas' Lemma that is limited to numerical domains and non-linear constraint solving that is of high computational complexity. This paper addresses the scalability issue by proposing a novel data-driven constraint optimization method based on counterexample guided inductive synthesis.

1 Introduction

Refinement types [1, 3, 7, 9, 13, 14, 17] are dependent types equipped with predicates that can express value-dependent behaviors of programs. They have been applied to verification of higher-order functional programs. To automate verification, researchers have proposed a variety of refinement type inference methods based on software model checking [6, 12, 13, 15, 18] and abstract interpretation techniques [5, 8].

These inference methods try to infer some refinement types that are sufficient to conclude that the given program satisfies a desired property such as assertion safety. This design is reasonable for the verification purpose because refinement type systems (with a decidable underlying theory) do not satisfy the principal typing property, and such typing (if any) is computationally hard to infer. Therefore, types inferred by these methods are often too specific to the given assertions and of limited applicability, in particular to modular verification of (a) higher-order functions that take functions as arguments and similarly (b) open programs that depend on other modules whose definitions are not available at the time of inference.

To regain user control over types to be inferred, we follow and extend an approach called *refinement type optimization* advocated by [4]. Our framework allows users to control type inference via (1) refinement type templates with predicate variables, which act as placeholders for refinement predicates to be inferred, (2) a subset of predicate substitutions, which along with the templates restricts the search space, and (3) a preference order on predicate substitutions, which expresses (possibly multi-objective) optimization criteria such as the strength/weakness of predicates.

We reduce the problem of finding a maximally-preferred (i.e., Pareto optimal) refinement type with respect to the user-specified order into a constraint optimization problem we call Optimizing predicate Constraint Satisfaction Problem (Opt-pCSP) where the set of constraints belong to the class of pCSP [10, 16] that extends the well-known class of Constraint Horn Clauses (CHCs) [2] with head-disjunctions.

This paper presents a novel data-driven con-

---

* データ駆動リファインメント型最適化.

Yu Gu, Hiroshi Unno, 筑波大学大学院情報理工学位プログラム, Dept. of Computer Science, University of Tsukuba.

Takeshi Tsukada, 千葉大学大学院理学研究院, Dept. of Mathematics and Informatics, Chiba University.

straint optimization method for Opt-pCSP based on CounterExample Guided Inductive Synthesis (CEGIS) [11] and synthesis based on stratified families of templates [16]. Our method iteratively refines the current solution of the given pCSP $\mathcal{C}$ until a Pareto optimal one is found. To this end, we solve a series $\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \ldots$ where $\mathcal{C}_0 = \mathcal{C}$ and each $\mathcal{C}_{i+1} (i \geq 0)$ is obtained from $\mathcal{C}$ by adding a constraint that specifies that any solution of $\mathcal{C}_{i+1}$ strictly improves the solution found for $\mathcal{C}_i$, with respect to the preference order. To express $\mathcal{C}_{i+1} (i \geq 0)$, we extend and use pCSP with *non-emptiness constraints*.

Our data-driven method remedies the scalability issue of the previous constraint optimization method based on Farkas' Lemma and non-linear constraint solving [4]: it does not scale to complex constraints over non-numerical domains because Farkas' Lemma is limited to numerical domains and non-linear constraint solving is of high computational complexity.

The rest of the paper is organized as follows. Section 2 briefly reviews an existing refinement type system and Section 3 formalizes refinement type optimization. Section 4 formalizes Opt-pCSP and discusses reduction from refinement type optimization to Opt-pCSP. In Section 5, we present our data-driven method for solving Opt-pCSP. We conclude the paper in Section 6.

2 Refinement Type System

This section briefly reviews an existing refinement type system [13]. The syntax of refinement types and formulas of the first-order quantifier-free theory of algebraic data types is defined as follows:

$$
\begin{array}{rll}
\sigma ::= & \{x : \delta \mid \psi\} & \text{(refinement type} \\
\mid & x : \sigma \to \sigma & \text{templates)} \\
\Pi ::= & \emptyset & \text{(type environment} \\
\mid & \Pi, x : \sigma & \text{templates)} \\
t ::= & x & \text{(terms)} \\
\mid & K(t_1, \ldots, t_{\mathrm{ar}(K)}) & \\
\mid & K\#i & \\
a ::= & P(\widetilde{t}) \mid t_1 = t_2 \mid \mathit{Is}_K(t) & \text{(atoms)} \\
\psi ::= & a \mid \top \mid \bot & \text{(formulas)} \\
\mid & \neg\psi \mid \psi_1 \wedge \psi_2 & \\
\mid & \psi_1 \vee \psi_2 \mid \psi_1 \Rightarrow \psi_2 &
\end{array}
$$

Here, x (resp. P) is a meta-variable ranging over term (resp. predicate) variables. We use ϕ as a meta-variable ranging over *pure formulas* which are formulas without predicate variables. We write $\mathit{ftv}(\phi)$ (resp. $\mathit{fpv}(\phi)$) for the set of free term (resp. predicate) variables in ϕ. We use p as a meta-variable ranging over predicates that are definable as $\lambda\widetilde{x}.\phi$ where $\mathit{ftv}(\phi) \subseteq \{\widetilde{x}\}$. We use τ (resp. Γ) as a meta-variable ranging over *refinement types* (resp. *refinement type environments*) which are refinement type templates (resp. refinement type environment templates) without predicate variables. We write $\mathit{ty}(t)$ for the type of the term t. δ is a meta-variable ranging over (monomorphic) algebraic data types (ADTs). K represents a *constructor* with $\mathit{ty}(K) = \delta_1 \times \cdots \times \delta_{\mathrm{ar}(K)} \to \delta$ for some ADTs $\delta, \delta_1, \ldots, \delta_{\mathrm{ar}(K)}$, where $\mathrm{ar}(K)$ represents the arity of K. $K\#i$ represents the *i-th selector* of the constructor K and $\mathit{ty}(K\#i(\widetilde{t})) = \delta_i$ if $\mathit{ty}(K) = \delta_1 \times \cdots \times \delta_{\mathrm{ar}(K)} \to \delta$ and $\mathit{ty}(t_i) = \delta_i$ for each i. $K\#i(t)$ is defined as t_i if t is equivalent to $K(\widetilde{t})$ for some $\widetilde{t}$ and undefined otherwise. $\mathit{Is}_K(t)$ represents the *tester* for the constructor K: $\mathit{Is}_K(t)$ is defined as $\top$ if t is equivalent to $K(\widetilde{t})$ for some $\widetilde{t}$ and $\bot$ otherwise.

We here omit the syntax of program expressions e and the inference rules for typing judgments $\Gamma \vdash e : \tau$ and subtyping judgments $\Gamma \vdash \tau_1 <: \tau_2$ (see [13] for the definitions). Intuitively, $\vdash \tau_1 <: \tau_2$ means that any program expression e that has the type τ_1 also has the type τ_2. In other words, τ_1 is logically stronger than τ_2 and τ_1 can express more precise program behaviors. The following definitions elaborate on the logical strength of refinement types:

Definition 1 (**Maximum Specification**). *The maximum specification of a closed program expression e is the strongest refinement type τ:*

$$\vdash e : \tau \wedge (\forall \tau'. \vdash e : \tau' \Rightarrow \vdash \tau <: \tau')$$

Definition 2 (**Maximal Specification**). *A maximum specification of a closed program expression e is a maximally-strong refinement type τ:*

$$\vdash e : \tau \wedge (\forall \tau'. \vdash e : \tau' \wedge \tau' \neq \tau \Rightarrow \nvdash \tau' <: \tau)$$

The maximum specification (if any) is the most accurate representation of the program's behaviors. However, it is usually not expressible as a refinement type equipped with predicates of a decidable underlying theory. This paper thus moves focus to maximal specifications. Note that they also do not exist unless we use appropriate refinement type templates. The next section introduces refinement

type optimization that allows users to control refinement type inference to obtain (but not limited to) a maximal specification.

3 Refinement Type Optimization

This section formalizes refinement type inference and generalizes it to refinement type optimization [4].

We first define some auxiliary notions. A *predicate type environment* Δ is a map from predicates variables to their types. We write $\mathrm{dom}(\Delta)$ for the domain of Δ. We write σ_Δ (resp. Π_Δ) if $\mathit{fpv}(\sigma_\Delta) \subseteq \mathrm{dom}(\Delta)$ (resp. $\forall(x:\sigma) \in \Pi_\Delta.\ \mathit{fpv}(\sigma) \subseteq \mathrm{dom}(\Delta)$). A *predicate substitution* θ is a map from predicate variables to (definable) predicates. We write Θ_Δ for the set of predicate substitutions conforming to Δ. We omit Δ and write Θ if it is clear from the context. We write $\theta\Gamma$ (resp. $\theta\sigma$) for the application of θ to Γ (resp. σ). We write $\theta{\restriction_\Delta}$ to denote the restriction of the domain of θ to $\mathrm{dom}(\Delta)$. We write $\theta \equiv \theta'$ if θ is logically equivalent to θ': $\mathrm{dom}(\theta) = \mathrm{dom}(\theta')$ and $\forall P \in \mathrm{dom}(\theta).\ \models \theta(P) \Leftrightarrow \theta'(P)$.

Definition 3 (**Refinement Type Inference**)**.** *Let e be a program expression. Let Π_Δ and σ_Δ be templates of a refinement type environment and a refinement type, respectively. A refinement type inference problem $(e, \Pi_\Delta, \sigma_\Delta)$ is the problem of finding a predicate substitution $\theta \in \Theta_\Delta$ such that $\theta\Pi_\Delta \vdash e : \theta\sigma_\Delta$.*

Definition 4 (**Refinement Type Optimization**)**.** *Let e be a program expression. Let Π_Δ and σ_Δ be templates of a refinement type environment and a refinement type, respectively. Let S_Δ and $\preceq_\Delta$ be a subset of Θ_Δ and a partial order on predicate substitutions in Θ_Δ, respectively. A refinement type optimization problem $(e, \Pi_\Delta, \sigma_\Delta, S_\Delta, \preceq_\Delta)$ is the problem of finding a predicate substitution θ such that:*

- $\theta \in S_\Delta$,
- $\theta\Pi_\Delta \vdash e : \theta\sigma_\Delta$, *and*
- $\forall\theta' \in S_\Delta.\theta'\Pi_\Delta \vdash e : \theta'\sigma_\Delta \wedge \theta' \not\equiv \theta \Rightarrow \theta' \not\preceq_\Delta \theta$.

Here, $\theta' \not\preceq_\Delta \theta$ means that either $\theta \prec_\Delta \theta'$ holds or θ and θ' are incomparable in $\prec_\Delta$. Note that $\theta_1 \prec_\Delta \theta_2$ intuitively means that θ_1 is more preferred than θ_2 and the goal of refinement type optimization is to find (if any) a maximally preferred θ (i.e., a minimal θ with respect to $\preceq_\Delta$).

4 Reduction from Refinement Type Optimization to Opt-pCSP

This section formalizes Optimizing predicate Constraint Satisfaction Problem (Opt-pCSP) and presents a reduction from refinement type optimization to Opt-pCSP.

First of all, we follow [10, 16] and define a predicate Constraint Satisfaction Problem (pCSP) $\mathcal{C}$ to be a finite set of clauses of the form

$$\phi \vee \left(\bigvee_{i=1}^{l} P_i(\widetilde{x_i})\right) \vee \left(\bigvee_{i=l+1}^{m} \neg P_i(\widetilde{x_i})\right)$$

where $0 \leq \ell \leq m$. We write $\mathcal{C}_\Delta$ if $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\Delta)$. We call a predicate substitution θ a *solution* for $\mathcal{C}$ if $\mathit{fpv}(\mathcal{C}) \subseteq \mathrm{dom}(\theta)$ and $\models \bigwedge \theta(\mathcal{C})$.

Definition 5 (pCSP **Satisfiability**)**.** *The predicate satisfiability problem of a* pCSP $\mathcal{C}$ *is the problem of finding a solution for $\mathcal{C}$.*

The language pCSP generalizes over existing languages of constraints. A pCSP $\mathcal{C}$ that satisfies $\ell \leq 1$ for all clauses in $\mathcal{C}$ is called Constraint Horn Clauses (CHCs) in the literature.

We now define Optimizing predicate Constraint Satisfaction Problem (Opt-pCSP).

Definition 6 (Opt-pCSP)**.** *Let $\mathcal{C}_\Delta$ be a* pCSP*. Let $\preceq_\Delta$ be a partial order on predicate substitutions in Θ_Δ and S_Δ be a subset of Θ_Δ. An* Opt-pCSP *$(\mathcal{C}_\Delta, \preceq_\Delta, S_\Delta)$ is the problem of finding a solution θ of $\mathcal{C}_\Delta$ such that:*

- $\theta \in S_\Delta$ *and*
- $\forall\theta' \in S_\Delta.\ \models \bigwedge \theta'(\mathcal{C}_\Delta) \wedge \theta \not\equiv \theta' \Rightarrow \theta' \not\preceq_\Delta \theta$.

The goal of Opt-pCSP is to find a maximally preferred θ (i.e., a minimal θ with respect to $\preceq_\Delta$).

We now present the reduction from refinement type optimization to Opt-pCSP.†1 A refinement type inference problem can be reduced to a pCSP by reusing an existing refinement type inference method [13]: More specifically, the procedure Gen defined in [13] inputs a refinement type inference problem $(e, \Pi_\Delta, \sigma_\Delta)$ and outputs a pCSP $\mathcal{C}_{\Delta'}$ for some Δ' such that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Delta')$. The constraint generation algorithm Gen satisfies the

†1 If the shape of refinement predicates in refinement type templates is restricted to the form $P(\widetilde{t})$, the reduction results in only CHCs optimization problems instead of Opt-pCSP. In other words, our adoption of Opt-pCSP here allows us to support expressive templates of refinement types.

soundness and the completeness as stated by the following lemmas (see [13] for the proofs):

Lemma 1. *A solution of the* pCSP $\mathcal{C}_{\Delta'} = \textsc{Gen}(e, \Pi_\Delta, \sigma_\Delta)$ *is a solution of the refinement type inference problem* $(e, \Pi_\Delta, \sigma_\Delta)$.

Lemma 2. *A solution of the refinement type inference problem* $(e, \Pi_\Delta, \sigma_\Delta)$ *can be extended to a solution of* pCSP $\mathcal{C}_{\Delta'} = \textsc{Gen}(e, \Pi_\Delta, \sigma_\Delta)$.

The following theorem shows the soundness of our reduction from refinement type optimization to Opt-pCSP:

Theorem 1 (**Soundness of Reduction**). *Suppose that* $\mathcal{C}_{\Delta'} = \textsc{Gen}(e, \Pi_\Delta, \sigma_\Delta)$. *A solution of the* Opt-pCSP $(\mathcal{C}_{\Delta'}, \mathit{ext}(\preceq_\Delta, \Delta'), \mathit{ext}(S_\Delta, \Delta'))$ *is a solution of the refinement type optimization problem* $(e, \Pi_\Delta, \sigma_\Delta, S_\Delta, \preceq_\Delta)$, *where* $\mathit{ext}(\preceq_\Delta, \Delta')$ *is the extension* $\{(\theta, \theta') \in \Theta_{\Delta'} \times \Theta_{\Delta'} \mid \theta\restriction_\Delta \preceq_\Delta \theta'\restriction_\Delta\}$ *of* $\preceq_\Delta$ *to* Δ' *and* $\mathit{ext}(S_\Delta, \Delta')$ *is the extension* $\{\theta \in \Theta_{\Delta'} \mid \theta\restriction_\Delta \in S_\Delta\}$ *of* S_Δ *to* Δ'.

Proof. Let $\preceq_{\Delta'}$ and $S_{\Delta'}$ respectively be $\mathit{ext}(\preceq_\Delta, \Delta')$ and $\mathit{ext}(S_\Delta, \Delta')$. Let θ be a solution of the Opt-pCSP $(\mathcal{C}_{\Delta'}, \preceq_{\Delta'}, S_{\Delta'})$. By Definition 6, θ is a solution of $\mathcal{C}_{\Delta'}$ such that $\theta \in S_{\Delta'}$ and $\forall \theta' \in S_{\Delta'}. \models \bigwedge \theta'(\mathcal{C}_{\Delta'}) \wedge \theta \not\equiv \theta' \Rightarrow \theta' \not\preceq_{\Delta'} \theta$. By Lemma 1, we obtain that $\theta\restriction_\Delta$ is a solution of the refinement type inference problem $(e, \Pi_\Delta, \sigma_\Delta)$. By Definition 3, we have $\theta\restriction_\Delta \Pi_\Delta \vdash e : \theta\restriction_\Delta \sigma_\Delta$. We next prove: $(\forall \theta' \in S_\Delta. \theta' \Pi_\Delta \vdash e : \theta'\sigma_\Delta \wedge \theta' \not\equiv \theta\restriction_\Delta \Rightarrow \theta' \not\preceq_\Delta \theta\restriction_\Delta)$. We assume that $\theta' \Pi_\Delta \vdash e : \theta'\sigma_\Delta \wedge \theta' \not\equiv \theta\restriction_\Delta$. By Lemma 2, there is θ'' such that $\mathrm{dom}(\theta'') = \mathrm{dom}(\Delta')$, $\models \bigwedge \theta''(\mathcal{C}_{\Delta'})$, and $\theta' \equiv \theta''\restriction_\Delta$. Thus $\theta'' \not\equiv \theta$ holds since $\theta' \not\equiv \theta\restriction_\Delta$ and $\theta' \equiv \theta''\restriction_\Delta$. We then have $\theta'' \not\preceq_{\Delta'} \theta$. By the definition of $\preceq_{\Delta'} = \mathit{ext}(\preceq_\Delta, \Delta')$, we get $\theta' \not\preceq_\Delta \theta\restriction_\Delta$. Thus $\theta\restriction_\Delta$ is a solution of the refinement type optimization problem $(e, \Pi_\Delta, \sigma_\Delta, S_\Delta, \preceq_\Delta)$. □

5 Data-Driven Approach to Opt-pCSP

This section presents a data-driven constraint optimization method for Opt-pCSP. As discussed in Section 1, we solve a series of pCSP extended with *non-emptiness constraints*. We call the extended class pnCSP. Formally, a pnCSP $(\mathcal{C}, \mathcal{K})$ consists of

- a finite set $\mathcal{C}$ of pCSP clauses over predicate variables and
- a kind function $\mathcal{K}$ that maps each predicate variable $P \in \mathit{fpv}(\mathcal{C})$ to its kind; either $\bullet$ or $\exists$ which respectively represent ordinary and non-empty predicate variables.

Here, *non-empty predicates* are predicates that are not equivalent to $\bot$.

Definition 7 (pnCSP **Satisfiability**). *The predicate satisfiability problem of a* pnCSP $(\mathcal{C}, \mathcal{K})$ *is the problem of finding a solution* θ *of* $\mathcal{C}$ *such that for all* $P \in \mathit{fpv}(\mathcal{C})$, $\models \exists \widetilde{x}.\theta(P)(\widetilde{x})$ *if* $\mathcal{K}(P) = \exists$.

Algorithm 1: Optimize

Input: An Opt-pCSP $(\mathcal{C}_\Delta, \preceq_\Delta, S_\Delta)$
Output: *Unknown* means that it is unknown whether a solution exists, *Unsat* means that there is no solution, $\mathit{Sat}(\theta)$ means that θ is a possibly non-minimal solution w.r.t. $\preceq_\Delta$, $\mathit{MinSat}(\theta)$ means that θ is a minimal solution w.r.t. $\preceq_\Delta$.

```
1  Function Optimize (C_Δ, ⪯_Δ, S_Δ) =
2      K_Δ ← {P ↦ • | P ∈ fpv(C_Δ)};
3      match Solve(C_Δ, K_Δ, S_Δ) with
4          | Unsat → return Unsat;
5          | Unknown → return Unknown;
6          | Sat(θ) →
7              while ⊤ do
8                  (C'_Δ', K'_Δ') ← Improve_⪯_Δ(θ);
9                  match Solve(C_Δ ∪ C'_Δ', K_Δ ∪
                       K'_Δ', ext(S_Δ, Δ ∪ Δ')) with
10                     | Unsat →
11                         return MinSat(θ)
12                     | Unknown →
13                         return Sat(θ)
14                     | Sat(θ') →
15                         θ ← θ'↾_Δ
16             end
17         end
18     end
19 end
```

We now explain our constraint optimization method for Opt-pCSP shown in Algorithm 1. Optimize iteratively improves the current solution θ of the pCSP $\mathcal{C}_\Delta$ until a minimal one with respect to $\preceq_\Delta$ is found. The sub-procedure $\textsc{Improve}_{\preceq_\Delta}$, which is parameterized by $\preceq_\Delta$, inputs the current solution θ and outputs a pnCSP $(\mathcal{C}'_{\Delta'}, \mathcal{K}'_{\Delta'})$ that

is used to constrain that any solution strictly improves the current solution θ with respect to $\preceq_\Delta$.

The sub-procedure SOLVE is for solving a pnCSP $(\mathcal{C}_\Delta, \mathcal{K}_\Delta, S_\Delta)$. We define SOLVE by extending an existing data-driven constraint solving method [16] with non-emptiness constraints and algebraic data types (ADTs). SOLVE is based on CounterExample Guided Inductive Synthesis (CEGIS), which iteratively accumulates example instances $\mathcal{E}$ of the original constraints $(\mathcal{C}_\Delta, \mathcal{K}_\Delta, S_\Delta)$ through the following two phases for each iteration:

- **Synthesis Phase**: find a candidate solution θ that satisfies all the examples in $\mathcal{E}$;
- **Validation Phase**: check whether the candidate θ also satisfies the original constraints $(\mathcal{C}_\Delta, \mathcal{K}_\Delta, S_\Delta)$, and if so, return θ as a genuine solution, and otherwise repeat the procedure by adding the found counterexamples to $\mathcal{E}$.

In the synthesis phase, the previous constraint solving method uses a predicate synthesizer based on stratified families of templates. The synthesizer can be extended to support non-emptiness constraints and ADTs by designing appropriate families of templates. Figure 1 presents stratified families of templates $T^\bullet_P(nd, depth)$ and $T^\exists_P(depth)$ respectively for ordinary and non-empty predicate variables over ADTs. The parameters nd and $depth$ of $T^\bullet_P(nd, depth)$ are used to control the maximum number of disjuncts and the maximum depth of selector applications. The parameter $depth$ of $T^\exists_P(depth)$ is used to control the maximum depth of constructor applications.

Stratified Template Family for Ordinary Predicate Variables:

$$
\begin{array}{l}
T^\bullet_P(nd, depth) \triangleq \\
\quad \texttt{let}\ \mathcal{T} = \{x_1, \ldots, x_{\mathrm{ar}(P)}\}\ \texttt{in} \\
\quad \texttt{let}\ \{\phi_1, \ldots, \phi_m\} = \mathit{quals}(depth, \mathcal{T})\ \texttt{in} \\
\quad \lambda\widetilde{x}.\, \bigvee_{i=1}^{nd} \bigwedge_{j=1}^{m} ((c_{i,j} > 0 \Rightarrow \phi_j) \wedge (c_{i,j} < 0 \Rightarrow \neg\phi_j))
\end{array}
$$

$$
\mathit{sel}(t) \triangleq \left\{ K\#i(t) \;\middle|\; \begin{array}{c} 1 \le i \le \mathrm{ar}(K), \delta = ty(t), \\ ty(K) = \delta_1 \to \cdots \to \delta_{\mathrm{ar}(K)} \to \delta \end{array} \right\}
$$

$$
\begin{array}{l}
\mathit{quals}(0, \mathcal{T}) \triangleq \\
\quad \{t_1 = t_2 \mid t_1, t_2 \in \mathcal{T}, t_1 \neq t_2\} \cup \\
\quad \left\{ \mathit{Is}_K(t) \;\middle|\; \begin{array}{c} t \in \mathcal{T}, ty(t) = \delta, \\ ty(K) = \delta_1 \to \cdots \to \delta_{\mathrm{ar}(K)} \to \delta \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{quals}(depth, \mathcal{T}) \triangleq \quad //depth \ge 1 \\
\quad \mathit{quals}(depth - 1, \mathcal{T} \cup \bigcup_{t \in \mathcal{T}} \mathit{sel}(t))
\end{array}
$$

Stratified Template Family for Non-Empty Predicate Variables:

$$
\begin{array}{l}
T^\exists_P(depth) \triangleq \\
\quad \texttt{let}\ \{p_1, \ldots, p_m\} = \mathit{genp}_P(depth)\ \texttt{in} \\
\quad \lambda\widetilde{x}.\, \bigvee_{i=1}^{m} b_i \wedge p_i(\widetilde{x})
\end{array}
$$

$$
\mathit{cons}(1, \mathcal{T}) \triangleq \left\{ K \begin{pmatrix} t_1, \\ \vdots \\ t_{\mathrm{ar}(K)} \end{pmatrix} \;\middle|\; \begin{array}{c} t_1, \ldots, t_{\mathrm{ar}(K)} \in \mathcal{T}, \\ ty(t_i) = \delta_i \text{ for each } i, \\ ty(K) = \delta_1 \to \cdots \to \delta_{\mathrm{ar}(K)} \to \delta \end{array} \right\}
$$

$$
\begin{array}{l}
\mathit{cons}(depth, \mathcal{T}) \triangleq \quad //depth \ge 2 \\
\quad \mathit{cons}(depth - 1,\ \mathcal{T} \cup \bigcup_{t \in \mathcal{T}} \mathit{cons}(t))
\end{array}
$$

$$
\mathit{genp}_P(depth) \triangleq \left\{ \lambda\widetilde{x}. \bigwedge_{i=1}^{\mathrm{ar}(P)} y_i = t_i \;\middle|\; \begin{array}{c} \{y_1, \ldots, y_{\mathrm{ar}(P)}\} = \{x_1, \ldots, x_{\mathrm{ar}(P)}\}, \\ t_1 \in \{y_1\} \cup \mathit{cons}(depth, \emptyset), \\ t_2 \in \{y_2\} \cup \mathit{cons}(depth, \{y_1\}), \\ \vdots \\ t_{\mathrm{ar}(P)} \in \{y_{\mathrm{ar}(P)}\} \cup \\ \quad \mathit{cons}(depth, \{y_1, \ldots, y_{\mathrm{ar}(P)-1}\}) \end{array} \right\}
$$

Fig. 1 Stratified Families of Templates

6 Conclusion

We have presented a sound reduction from refinement type optimization to a new expressive class of constraint optimization problems called Opt-pCSP, which is necessary to support expressive templates of refinement types. Furthermore, we have proposed a novel data-driven constraint optimization method for Opt-pCSP over algebraic data types, which remedies the scalability issue of the previous constraint optimization method based on Farkas' Lemma and non-linear constraint solving.

References

[1] Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A. D., and Maffeis, S.: Refinement Types for Secure Implementations, *ACM Transactions on Programming Languages and Systems*, Vol. 33, No. 2(2011), pp. 8:1–8:45.

[2] Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A.: Horn Clause Solvers for Program Verification, *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, LNCS, Vol. 9300, Springer, 2015, pp. 24–51.

[3] Flanagan, C.: Hybrid type checking, *POPL '06*, ACM, 2006, pp. 245–256.

[4] Hashimoto, K. and Unno, H.: Refinement Type Inference via Horn Constraint Optimization, *SAS '15*, LNCS, Vol. 9291, Springer, 2015, pp. 199–216.
[5] Jhala, R., Majumdar, R., and Rybalchenko, A.: Refinement Type Inference via Abstract Interpretation, arXiv:1004.2884v1, 2010.
[6] Kobayashi, N., Sato, R., and Unno, H.: Predicate abstraction and CEGAR for higher-order model checking, *PLDI '11*, ACM, 2011, pp. 222–233.
[7] Nanjo, Y., Unno, H., Koskinen, E., and Terauchi, T.: A Fixpoint Logic and Dependent Effects for Temporal Property Verification, *LICS '18*, ACM, July 2018, pp. 759–768.
[8] Pavlinovic, Z., Su, Y., and Wies, T.: Data Flow Refinement Type Inference, *Proceedings of the ACM on Programming Languages*, Vol. 5, No. POPL(2021).
[9] Rondon, P., Kawaguchi, M., and Jhala, R.: Liquid Types, *PLDI '08*, ACM, 2008, pp. 159–169.
[10] Satake, Y., Unno, H., and Yanagi, H.: Probabilistic Inference for Predicate Constraint Satisfaction, *AAAI '20*, Vol. 34, No. 02(2020), pp. 1644–1651.
[11] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V.: Combinatorial Sketching for Finite Programs, *ASPLOS XII*, ACM, 2006, pp. 404–415.
[12] Terauchi, T.: Dependent types from counterexamples, *POPL '10*, ACM, 2010, pp. 119–130.
[13] Unno, H. and Kobayashi, N.: Dependent Type Inference with Interpolants, *PPDP '09*, ACM, 2009, pp. 277–288.
[14] Unno, H., Satake, Y., and Terauchi, T.: Relatively Complete Refinement Type System for Verification of Higher-order Non-deterministic Programs, *Proceedings of the ACM on Programming Languages*, Vol. 2, No. POPL(2017), pp. 12:1–12:29.
[15] Unno, H., Terauchi, T., and Kobayashi, N.: Automating Relatively Complete Verification of Higher-order Functional Programs, *POPL '13*, ACM, 2013, pp. 75–86.
[16] Unno, H., Terauchi, T., and Koskinen, E.: Constraint-based Relational Verification, *CAV '21*, Springer, 2021.
[17] Xi, H. and Pfenning, F.: Dependent types in practical programming, *POPL '99*, ACM, 1999, pp. 214–227.
[18] Zhu, H., Nori, A. V., and Jagannathan, S.: Learning Refinement Types, *ICFP '15*, ACM, 2015, pp. 400–411.