Type-Based Parametric Analysis of Program Families

Sheng Chen
Oregon State University
chensh@eecs.oregonstate.edu

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Abstract

Previous research on static analysis for program families has focused on lifting analyses for single, plain programs to program families by employing idiosyncratic representations. The lifting effort typically involves a significant amount of work for proving the correctness of the lifted algorithm and demonstrating its scalability. In this paper, we propose a parameterized static analysis framework for program families that can automatically lift a class of type-based static analyses for plain programs to program families. The framework consists of a parametric logical specification and a parametric variational constraint solver. We prove that a lifted algorithm is correct provided that the underlying analysis algorithm is correct. An evaluation of our framework has revealed an error in a previous manually lifted analysis. Moreover, performance tests indicate that the overhead incurred by the general framework is bounded by a factor of 2.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs–Functional constructs,Type structure; D.3.2 [*Programming Languages*]: Language Classifications–Applicative (functional) languages

General Terms Languages, Theory

Keywords Variational types, constraint-based type system, static-analysis lifting, program families, choice calculus

1. Introduction

Software increasingly includes some form of variation. This can range from something as simple as having a few configuration options represented by `#ifdef` annotations to fully-fledged software product lines (SPLs) [13]. Each such variational program effectively encodes a (potentialy huge) number of programs and is thus also often called a *program family* [31].

Ensuring static properties of program families is challenging because the brute-force approach of generating and analyzing each individual program is generally infeasible due to the sheer number of programs a program family may encode.[1] Thus, the design of scalable analysis algorithms for program families has been the subject of much research, such as the variationalization of parsing [19, 23], type checking [1, 10, 17, 22, 24], dataflow analysis [5, 25], model checking [2, 11, 12, 14] and theorem proving [16, 41].

Most of these approaces employ some form of "lifting" strategy to extend a traditional analysis algorithm to deal with variational code. For instance, the variability-aware module system [24] is the result of lifting the module system introduced by Cardelli [6], the variational type inference [10] is the result of lifting the algorithm $\mathscr{W}$ [15], and the variability-aware dataflow analysis [5] is a lifted version of the traditional intraprocedural dataflow analysis [28]. All these approaches follow a similar pattern that typically involves the following steps.

(1) Add variability to data structures used in the traditional analysis. For example, variational types are represented as sets of plain types [1], value sets for dataflow analysis are extended to functions from features to values [5], or in variational type inference [9, 10], types, unifiers, and typing environment are all made variational using an explicit choice representation [18]. A similar representation has been adopted in [24, 25].

(2) Adapt analysis rules to deal with variational data structures. This applies in the obvious way to all data structure extensions mentioned in (1). Some approaches require additional machinery. For example, in [10], a type equivalence relation is required to make comparisons with choice types less rigid, and for the model checking approach described in [11, 12], the nodes and edges of transition systems are annotated with features, leading to feature transition systems.

(3) Prove that variation elimination commutes with the analysis, that is, a selection made from a program family and its analysis result should result in a plain program p and a result r such that running the traditional analysis on p would produce r. For example, the work on type checking [22] and type inference [9, 10] presented such proof. A similar proof was absent for the dataflow analysis [5], but was later presented in [4].

(4) A performance evaluation that demonstrates the efficiency gain of the lifted analysis over the brute-force approach.

This commonality indicates a general mechanism to systematically lift traditional static analyses to make them work for program families, avoiding the tedious steps of adding variability, performing proofs, and evaluating performance. So far, however, such a method is not available. The only attempt at something similar was recently made by Bodden et al. [3]. However, their approach only works for dataflow analyses formulated in the IFDS framework [35].

In this paper, we propose a type-based framework for automatically lifting plain-program static analyses to program families. Our method is applicable if (a) the analysis to be lifted can be expressed as a type system and (b) the program family is expressed using annotations (such as `#ifdef` CPP commands).

We have formalized our framework based on type analysis because it is both popular [30] and expressive [27]. Specifically, fol-

[1] For example, MySQL contains some 900 macros. Assuming these are pair-wise independent binary macros, the encoded number of program variants is close to the number of atoms in the universe to the 4th power.

ICFP'14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628155.

lowing the spirit of HM(X) [29], a parameterized Hindley-Milner type system, we design VHM(X), an extension of HM(X) with variational constructs and an annotation model, allowing a richer set of information to be tracked during type checking. The success and scope of the HM(X) has been demonstrated with a wide range of program analyses, such as overloading [39], conditional constraints [32], or flow inference [33]. It has also been extended by abstract polymorphic data types [37] and guarded algebraic data types [38].

1.1 A Simple Example: Control Flow Analysis

As one example, we consider a type-based 0CFA (context-insensitive control flow analysis), discussed by Palsberg [30], that answers the question "what is the potential set of functions an expression may be evaluated to?" Consider, for example, the following expression introduced in [30].

$$f = ((\lambda^1 h.\lambda^2 x.h\ x)\ (\lambda^3 y.y))\ (\lambda^4 z.z)$$

The type-based 0CFA developed in [21] computes $\lambda z.z$, written as $\{4\}$, as the analysis result for f, meaning that f will evaluate to the abstraction labeled by 4.

Now consider the following, related program family that is obtained by creating a choice named D between $\lambda^3 y.y$ and $\lambda^5 w.\lambda^6 y.y$.

$$e = ((\lambda^1 h.\lambda^2 x.h\ x)\ D\langle(\lambda^3 y.y), (\lambda^5 w.\lambda^6 y.y)\rangle)\ (\lambda^4 z.z)$$

Choice expressions such as $D\langle e_1, e_2\rangle$ denote variation points in the program. They require that one of the alternatives e_1 and e_2 be selected to obtain a plain program from the program family. By selecting the first alternative of D in e, we obtain f. If we instead select the second alternative, we obtain the following expression g.

$$g = ((\lambda^1 h.\lambda^2 x.h\ x)\ (\lambda^5 w.\lambda^6 y.y))\ (\lambda^4 z.z)$$

For g, 0CFA produces the result $\{6\}$. This means that for the program family e we obtain the variational result $D\langle\{4\},\{6\}\rangle$, which reflects the fact that the result of the analysis depends on the selection made for D. For the first alternative, we get the result $\{4\}$, and for the second alternative, we get $\{6\}$.

Since the brute-force approach of generating all variants and running the analysis on each variant separately doesn't scale, a 0CFA for program families has to work with the variation representation (choices, as shown, or other) directly.

Now instead of having to go through the four steps sketched above over and over again, for each new program analysis, we would rather develop a method that, given a suitable representation of a single-program analysis, would generate a provably correct program-family version of that analysis automatically.

1.2 Contributions, Perspectives, and Previous Work

This work is related to the HM(X) system developed by Odersky et al. [29] and our variational type inference algorithm [10], but extends both significantly.

We extend HM(X) in two ways. First, we make expressions, types, constraints, and also typing systems *variational*, which allows us to deal with program *families*. Second, types and also derivations are extended with annotations, which allow us to encode more type-based analyses in the framework.

There are also two major differences compared to our variational type inference [10]. First, this paper develops an analysis lifting framework while the previous work was about a type inference algorithm for variational programs. The machinery developed in this paper is more general, and the variational type inference algorithm can be expressed as an instance of the lifting framework. Second, from a technical point of view, the need for a general variational constraint solving algorithm has provided a more fundamental understanding of the problem domain. Specifically, the variational constraint solving algorithm developed here, while being more general, is also significantly simpler than our previous variational unification algorithm [10] when instantiated with the Robinson algorithm [36].

This paper presents a systematic approach to automatically lift program analyses to families of functional (and other) programs. Our work is based on a variation representation that we briefly review in Section 2. The paper then makes the following contributions.

- We introduce the idea of an automated analysis lifting framework in the form of HM(X) in Section 3.
- We present a parameterized type system that declaratively specifies the computation of analyses in Section 4. We show that the variational analysis is sound provided the underlying analysis is sound.
- In Section 5 we develop a constraint-based inference system for the type system, which is both sound and complete with respect to the type system.
- In Section 6 we construct a variational constraint solver that is parameterized by domain-specific constraint solvers. We prove that the variational constraint solver is sound/most general provided the underlying solver is sound/most general.
- In Section 7 we evaluate a prototype implementation by comparing it with manually lifted static analyses. The evaluation has revealed an error in one such manually lifted analysis, and has indicated that the runtime overhead of our approach is bounded by a factor of 2.

Beyond the theoretical lifting framework, this paper has another important impact on programming language research. With the ability to lift a class of analyses, our framework makes what we call "property-guided product derivation" feasible. In current practice, selection of programs from program families is solely based on features and functional properties. However, in many cases it would be useful to have additional selection criteria available. For example, selecting programs based on different side effects, different sets of exceptions, different kinds of security policies, and so on. Our framework enables users to express such additional non-functional requirements as part of the program selection process.

2. Variation Representation

In this paper, we focus on the static analyses for so-called "annotative" program families, that is, program families that are obtained by directly annotating program parts that vary, for example, using CPP directives. For concreteness, we use binary choices as provided by the choice calculus [18] to represent variation. The choice calculus can be viewed as a restricted, yet more disciplined version of the C preprocessor.

The main construct of the choice calculus is a named *choice* to represent alternatives in programs (and other artifacts). For the work in this paper we can assume that all choice names are globally scoped and that each choice has exactly two alternatives. For example, the expression $A\langle$`succ`$,$`odd`$\rangle$ `1` represents the two *plain* expressions `succ 1` and `odd 1`. The name A is called a *dimension*.

Choices can be eliminated through a *selection* operation, which applies a *selector* s, given by a dimension D and an index i, to an expression and replaces each of its choices named D by its ith alternative. For lambda calculus with binary choices, selection is

Term variables	x, y, z	Annotation variables	β
Type variables	α	Type constructors	T
Choices	A, B, D	Program locations	l
Constraints	C		

Expressions	e	$::=$	$x \mid \lambda^l x.e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid D\langle e, e\rangle$

Monotypes	τ	$::=$	$\alpha^\varphi \mid \tau \to^\varphi \tau$
Variational types	ϕ	$::=$	$\tau \mid D\langle\phi,\phi\rangle \mid \phi \to^\varphi \phi \mid T^\varphi\ \overline{\phi}$
Constraints	C	$::\supset$	$true \mid \phi \equiv \phi \mid \varphi \leq \varphi \mid D\langle C, C\rangle \mid C \wedge C \mid \exists\alpha.C$
Type schemas	σ	$::=$	$\forall\overline{\alpha}\overline{\beta}[C].\phi$
Annotations	φ	$::\supset$	$\varnothing \mid \{l\} \mid \beta \mid \varphi \cup \varphi \mid D\langle\varphi,\varphi\rangle$
Selectors	s	$::=$	$D.i$

Type environments	Γ	$::=$	$\varnothing \mid \Gamma, x \mapsto \sigma$
Substitutions	θ	$::=$	$\varnothing \mid \theta, \alpha \mapsto \phi$

Figure 1. Syntax of expressions, types, constraints, etc.

defined as follows.

$$\begin{aligned}
\lfloor x \rfloor_s &= x \\
\lfloor \lambda x.e \rfloor_s &= \lambda x.\lfloor e \rfloor_s \\
\lfloor e_1\ e_2 \rfloor_s &= \lfloor e_1 \rfloor_s\ \lfloor e_2 \rfloor_s \\
\lfloor D\langle e_1, e_2\rangle \rfloor_s &= \begin{cases} \lfloor e_1 \rfloor_s & \text{if } s = D.1 \\ \lfloor e_2 \rfloor_s & \text{if } s = D.2 \\ D\langle \lfloor e_1 \rfloor_s, \lfloor e_2 \rfloor_s\rangle & \text{otherwise} \end{cases}
\end{aligned}$$

The selection operation essentially traverses the AST and replaces choices along the way; it thus extends naturally to other language constructs.

For example, selecting $A.1$ from the expression $e = A\langle \texttt{succ}, \texttt{odd}\rangle\ \texttt{1}$ results in the expression $\lfloor e \rfloor_{A.1} = \texttt{succ 1}$. Selection synchronizes choices in the same dimension. For example, the expression $A\langle \texttt{succ}, \texttt{odd}\rangle\ A\langle \texttt{1}, \texttt{True}\rangle$ represents two expressions, `succ 1` and `odd True`, which can be obtained by one selection. In contrast, we need two selections, one in A and one in B, to eliminate the choices in the expression $A\langle \texttt{succ}, \texttt{odd}\rangle\ B\langle \texttt{1}, \texttt{True}\rangle$. Since the dimensions A and B are independent of one another, the choices represents four plain expressions.

Note that the choice representation is generic and can be used with any object language. It can thus represent variation in programs, types, and other software artifacts and data structures.

In the case of globally scoped choices, the semantics of a variational expression e can be expressed as a mapping from sets of selectors to plain expressions. A formal definition can be found in [10]. For illustration, here is a simple example.

$$[\![A\langle \texttt{succ}, B\langle \texttt{odd}, \texttt{even}\rangle\rangle\ \texttt{1}]\!] =$$
$$\{\{A.1\} \mapsto \texttt{succ 1}, \{A.2, B.1\} \mapsto \texttt{odd 1}, \{A.2, B.2\} \mapsto \texttt{even 1}\}$$

3. VHM(X) Syntax

We present the core syntax for VHM(X) in Section 3.1 and explain how to extend the syntax to encode static analyses in Section 3.2.

3.1 Core Syntax

We consider a parameterized type system over basic lambda calculus plus let-polymorphism and choice constructs. Figure 1 presents the syntax of various concepts used throughout this paper. The definition of expressions is conventional, except for the choice construct (see Section 2). We also attach a label l to lambda abstractions to track information about abstractions during type checking. In addition, we may make use of constants, such as `succ` and `even`, which have to appear in the initial type environment.

Another important extension of the HM(X) framework [29] is the addition of annotations (φ) to types. An annotation is a set representing information of interest for a specific analysis. Thus, the definition of φ is partly unspecified, and we adopt the notation $::\supset$ from [29] to indicate this fact. However, labels are always available as annotations, as is clear from the definition of φ. The distributive nature of choices [18] allows us to view $D\langle\varphi_1,\varphi_2\rangle \cup \varphi$ as $D\langle\varphi_1 \cup \varphi, \varphi_2 \cup \varphi\rangle$. (The case for $\varphi \cup D\langle\varphi_1,\varphi_2\rangle$ is symmetric.) This equivalence can be employed to normalize annotations such that choice constructs will not be nested within sets.

The definition of monotypes is fairly simple. However, types can carry annotations (φ). Later we will see that monotypes are the only types that are allowed to appear in the type part of a type schema. More complicated type definitions will become part of the constraints that will be discussed shortly. This design decision is motivated by Sulzmann et al. [40], who observed that in the presence of non-regular equational theories type inference may fail to compute principal types even with the help of a most general unification algorithm. They proposed as a solution to push complicated type definitions into constraints. This is what we adopt in this paper.

The definition of variational types ϕ differs from conventional type definitions in two aspects. First, we include choice types $D\langle\phi,\phi\rangle$ to represent variation in types. Second, since different instances of VHM(X) involve different type representations, we allow each instance of VHM(X) to supply extra type definitions, through the use of annotated type constructors of the form $T^\varphi\ \overline{\phi}$. Type constructors can take an arbitrary number of arguments, including 0, in which case they represent primitive types, such as `bool` and `int`. Like monotypes, variational types are also annotated.

The definition for constraints C is also open. However, we assume that the following constraints are always defined.

(i) *true* denotes a constraint that is always satisfied.

(ii) The constraint $\phi_1 \equiv \phi_2$ represents a type equivalence requirement between two types ϕ_1 and ϕ_2.

(iii) The constraint $\varphi_1 \leq \varphi_2$ imposes a partial order on annotations.

(iv) The constraint $D\langle C_1, C_2\rangle$ allows a variation in constraints.

(v) The constraints $C_1 \wedge C_2$ and $\exists\alpha.C$ have the same meaning as in first-order logic.

Each instance of VHM(X) may extend the definition of constraints.

A type schema, written as $\forall\overline{\alpha}\overline{\beta}[C].\tau$, consists of a constraint and a monotype. Note that it is polymorphic both over types and annotations. The constraint part C places a requirement on types and annotations that may be substituted for $\overline{\alpha}$ and $\overline{\beta}$ in τ: Only types and annotations that satisfy C can be used. Type schemas are considered equivalent modulo bound variable renaming.

We use θ to range over substitutions. We write $FV(C)$ for the set of free variables in C (where the $\exists$ quantifier is the only binding symbol in constraints). We also use $FV(\sigma)$ to denote the set of free variables in σ and extend its definition to type environments Γ in the usual way. The application of substitution to constraints, type schemas, and type environment, written as $\theta(C)$, $\theta(\sigma)$, and $\theta(\Gamma)$, respectively, is also defined in the usual way.

Note that the definition of ϕ and C allows us to shift complex conditions from types to constraints. For example, the expression $A\langle \texttt{succ}, \texttt{odd}\rangle$ has the type $A\langle \texttt{int} \to \texttt{int}, \texttt{int} \to \texttt{bool}\rangle$, which we can also express by saying that $A\langle \texttt{succ}, \texttt{odd}\rangle$ has the type α under the constraint $\alpha \equiv A\langle \texttt{int} \to \texttt{int}, \texttt{int} \to \texttt{bool}\rangle$.

3.2 Syntax Extensions

To instantiate VHM(X) for a specific analysis, the following information must be supplied.

- A type definition to extend ϕ, given as a type signature $\mathscr{T}$.
- An annotation definition $\mathscr{A}$ that extends φ.
- A constraint definition $\mathscr{C}$ to extend C.

These components prepare the framework for the addition of the analysis proper, which has to be provided as an extension of the type system and will be discussed in Section 4.3.

The extensibility of VHM(X) through the components $\mathscr{T}$, $\mathscr{A}$, and $\mathscr{C}$ offers much flexibility and allows a wide variety of analyses to be instantiated. In many instances, however, only a part of these components is needed, and VHM(X) can retain most of its default behavior.

As an example, consider the 0CFA. Section 1.1. The only annotations will be the labels for abstractions, which are already part of VHM(X). We thus have $\mathscr{A} = \varnothing$. Similarly, all the types and constraints for 0CFA are already present, that is, $\mathscr{T} = \mathscr{C} = \varnothing$.

For another example, consider exception analysis which tries to determine statically what kind of exceptions may be raised during the evaluation of an expression (see, for example, [28]). To instantiate VHM(X), we extend the type definition by a set of nullary exception type constructors, that is, we define $\mathscr{T} = \{EX_1, \ldots, EX_n\}$. In addition, we extend the definition of annotation with types, that is, $\varphi ::= \cdots \mid \{\phi\}$, so that exception types can be type annotations. Finally, the constraint definition $\mathscr{C}$ is extended by a predicate $Ex\ \phi$ to denote that ϕ is an exception type.

4. VHM(X) Type System

In this section we present the logical specification of the VHM(X) framework. We begin with the entailment relation in Section 4.1, which is followed by a discussion of typing rules for reasoning about programs in Section 4.2. We then show how to instantiate the logical part of VHM(X) to encode new static analyses in Section 4.3. Finally, we investigate the property of VHM(X) by comparing it to HM(X) in Section 4.4.

4.1 Constraint Entailment

The exact interpretation of constraints depends, in general, on the analysis that is being implemented. However, we require constraints to always satisfy the entailment relation defined in Figure 2. Intuitively, $C_1 \Vdash C_2$ means that the constraint C_1 is more restrictive than C_2. The top part of Figure 2 presents the relation with regard to the definition of constraints, and the bottom part defines entailment with regard to type equivalence constraints in particular.

Rule C1 expresses the monotonicity of constraints, where we use the notation $C_1 \supseteq C_2$ to denote that C_1 is more constrained than C_2 by interpreting the connective $\wedge$ as set union $\cup$ and each primitive constraint C as $\{C\}$. The rule can also be understood by viewing the constraints in a set as a conjunction, which means the set with more constraints is more restrictive. Rule C2 states transitivity, and rule C3 requires entailment to be preserved over substitution. Rules C4 and C5 deal with existential quantification. Since quantifying a constraint with $\exists$ hides part of the constraint, it becomes less restrictive [29]. Rules C6 to C8 handle choices between constraints. The validity of these rules can be seen by considering the rules that result when making an arbitrary selection for the choice D. The purpose of rule C9 is to allow the decomposition of choices into two constraints. Again, we can verify the validity of this rule by eliminating the variation in this rule, which results in two rules, with each can be verified by the rule C1. This rule offers many opportunities for optimization.

$$\text{C1}\ \frac{C_1 \supseteq C_2}{C_1 \Vdash C_2} \qquad \text{C2}\ \frac{C_1 \Vdash C_2 \qquad C_2 \Vdash C_3}{C_1 \Vdash C_3} \qquad \text{C3}\ \frac{C_1 \Vdash C_2}{\theta(C_1) \Vdash \theta(C_2)}$$

$$\text{C4}\ \ C \Vdash \exists\alpha.C \qquad \text{C5}\ \frac{C_1 \Vdash C_2}{\exists\alpha.C_1 \Vdash \exists\alpha.C_2} \qquad \text{C6}\ \frac{C \Vdash C_1 \qquad C \Vdash C_2}{C \Vdash D\langle C_1, C_2\rangle}$$

$$\text{C7}\ \frac{C_1 \Vdash C \qquad C_2 \Vdash C}{D\langle C_1, C_2\rangle \Vdash C} \qquad \text{C8}\ \frac{C_1 \Vdash C_2 \qquad C_3 \Vdash C_4}{D\langle C_1, C_3\rangle \Vdash D\langle C_2, C_4\rangle}$$

$$\text{C9}\quad D\langle C_1, true\rangle \wedge D\langle true, C_2\rangle \Vdash D\langle C_1, C_2\rangle$$

$$\text{E1}\quad \phi_1 \equiv \phi_2 \Vdash \phi_2 \equiv \phi_1 \qquad \text{E2}\quad \phi_1 \equiv \phi_2 \wedge \phi_2 \equiv \phi_3 \Vdash \phi_1 \equiv \phi_3$$

$$\text{E3}\quad \Vdash \phi \equiv \phi \qquad \text{E4}\quad \phi_1 \equiv \phi_2 \Vdash \phi[\phi_1] \equiv \phi[\phi_2]$$

$$\text{E5}\quad \Vdash D\langle \phi, \phi\rangle \equiv \phi \qquad \text{E6}\quad \Vdash D\langle \phi_1, \phi_2\rangle \equiv D\langle \lfloor \phi_1 \rfloor_{D.1}, \lfloor \phi_2 \rfloor_{D.2}\rangle$$

$$\text{E7}\quad C \Vdash D\langle T^{\varphi_1}\ \overline{\phi_1}, T^{\varphi_2}\ \overline{\phi_2}\rangle \equiv T^{D\langle \varphi_1, \varphi_2\rangle}\ \overline{D\langle \phi_1, \phi_2\rangle}$$

$$\text{A1}\ \frac{\varphi_1 \subseteq \varphi_2}{C \Vdash \varphi_1 \leq \varphi_2}$$

Figure 2. Entailment relation of constraints

For the entailment of type equivalence constraints, rules E1 to E3 express that type equivalence is reflexive, symmetric, and transitive. Rule E4 states that the type equivalence relation is a congruence, where $\phi[]$ denotes a type term with a hole (context) into which we can plug a type. Rules E5 and E6 express two fundamental invariances of choice types. Rule E7 allows the annotations attached to a type constructor T that occurs in both alternatives of a choice D to be extracted and combined with T. This rule is valid because selection commutes with term construction, that is, $\lfloor T\ \phi \rfloor_s = T\ \lfloor \phi \rfloor_s$, a fact that follows immediately from the definition of selection (see Section 2). The importance of this rule lies in the fact that it allows us to represent variational types succinctly. For instance, we can push the choice into the term and factor

$$D\langle(\alpha \xrightarrow{\{1\}} \alpha) \xrightarrow{\{2\}} \alpha \xrightarrow{\{1\}} \alpha, (\alpha \xrightarrow{\{1\}} \alpha) \xrightarrow{\{3\}} \alpha \xrightarrow{\{1\}} \alpha\rangle$$

into $(\alpha \xrightarrow{\{1\}} \alpha) \xrightarrow{D\langle\{2\},\{3\}\rangle} \alpha \xrightarrow{\{1\}} \alpha$.

Finally, we add the rule A1 for the partial order relation $\leq$ for annotations. Together with the rules C6 through C8, we define a partial ordering on variational annotations. Specific instances may extend the definition of $\leq$.

4.2 Typing Rules

The idea of type-based static analysis is to attach annotations to types and typing derivations and aggregate the information along the typing process [28]. Different analyses introduce their own specific annotations and relations between annotations that have to be integrated into the generic typing framework. To facilitate this flexibility, we parameterize the syntax-directed typing rules by a family of constraints, defining one particular constraint relationship R_e for each kind of expression e. The arity of R_e varies for different constructors of e; as a general rule of thumb, R_e needs enough parameters to relate the annotations obtained from the premises of a typing rule to the annotation provided in its conclusion. The premises of each syntax-directed typing rule can thus be partitioned into two parts: (i) a specification of the relation between types and (ii) a specification of the relation between labels and annotations (ex-

$$\textsc{Var}\quad \frac{\Gamma(x) = \forall\overline{\alpha}\overline{\beta}[C].\tau^{\varphi_1} \qquad C' \Vdash C \qquad C' \Vdash R_x\ \varphi_1\ \varphi_2}{C';\Gamma \vdash x : \tau^{\varphi_1}/\varphi_2}$$

$$\textsc{Abs}\quad \frac{C;\Gamma,(x,\tau) \vdash e : \tau'/\varphi_1 \qquad C \Vdash R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3}{C;\Gamma \vdash \lambda^l x.e : \tau \xrightarrow{\varphi_2} \tau'/\varphi_3}$$

$$\textsc{App}\quad \frac{C;\Gamma \vdash e_1 : \tau_1/\varphi_1 \qquad C;\Gamma \vdash e_2 : \tau_2/\varphi_2 \qquad C \Vdash \tau_1 \equiv \tau_2 \xrightarrow{\varphi_3} \tau \qquad C \Vdash R_{App}\ \varphi_1\ \varphi_2\ \varphi_3\ \varphi_4}{C;\Gamma \vdash e_1\ e_2 : \tau/\varphi_4}$$

$$\textsc{Let}\quad \frac{C;\Gamma \vdash e : \sigma/\varphi_1 \qquad C;\Gamma,(x,\sigma) \vdash e' : \tau/\varphi_2 \qquad C \Vdash R_{Let}\ \varphi_1\ \varphi_2\ \varphi_3}{C;\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \tau/\varphi_3}$$

$$\textsc{Choice}\quad \frac{C_1;\Gamma \vdash e_1 : \tau_1/\varphi_1 \qquad C_2;\Gamma \vdash e_2 : \tau_2/\varphi_2 \qquad D\langle C_1,C_2\rangle \Vdash D\langle\tau_1,\tau_2\rangle \equiv \tau}{D\langle C_1,C_2\rangle;\Gamma \vdash D\langle e_1,e_2\rangle : \tau/D\langle\varphi_1,\varphi_2\rangle}$$

$$\textsc{Sub}\quad \frac{C;\Gamma \vdash e : \tau/\varphi \qquad C \Vdash \tau \preceq \tau'}{C;\Gamma \vdash e : \tau'/\varphi} \qquad \textsc{Weaken}\quad \frac{C';\Gamma \vdash e : \tau/\varphi \qquad C \Vdash C'}{C;\Gamma \vdash e : \tau/\varphi}$$

$$\textsc{Gen}\quad \frac{C_1 \wedge C_2;\Gamma \vdash e : \tau/\varphi \qquad \overline{\alpha}\overline{\beta} \# FV(C_1) \cup FV(\Gamma)}{C_1 \wedge \exists\overline{\alpha}\overline{\beta}.C_2;\Gamma \vdash e : \forall\overline{\alpha}\overline{\beta}[C_2].\tau/\varphi}$$

$$\textsc{Simple}\quad \frac{C;\Gamma \vdash e : \sigma/\varphi \qquad \overline{\alpha}\overline{\beta} \# FV(\sigma) \cup FV(\Gamma)}{\exists\overline{\alpha}\overline{\beta}.C;\Gamma \vdash e : \sigma/\varphi}$$

Figure 3. Typing rules for VHM(X)

pressed by R_e). The family of constraints R_e has to be instantiated by each specific analysis instance. Whenever the second part is not needed, it can be simply turned off by using the constraint *true* as instantiation.

The typing judgment of VHM(X) has the form $C;\Gamma \vdash e : \sigma/\varphi$, which computes the type schema σ for the expression e under the typing assumptions in Γ and the constraint C that expresses the assumptions about free type variables in Γ and σ. In addition, an annotation φ is maintained by the typing relation. This annotation links the annotation constraints to the typing process and ensures that the information computed within the annotations is available as results in the type derivation. Figure 3 presents the rules for assigning types and annotations to expressions. The syntax-directed rules are shown in the upper half, and the remaining rules are collected at the bottom.

The rule VAR allows us to instantiate a type schema $\forall\overline{\alpha}\overline{\beta}[C].\tau$ with any constraint C' as long as it entails C and the parameterized constraint $R_x\ \varphi_1\ \varphi_2$ is satisfied (which expresses the relationship between the stored and returned annotation for a variable).

The rule ABS extends the traditional rule by a premise that demands the relationship R_λ hold between the annotations of the abstraction's body (φ_1), the function type (φ_2), and the whole derivation (φ_3). Note that usually φ_2 collects interesting information about the evaluation of the abstraction while φ_3 collects the information about defining the abstraction [28]. For example, in case of exception analysis (Section 4.3), the constraint is instantiated as $R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3 = \varphi_1 \le \varphi_2 \wedge \varphi_2 \le \varphi_1 \wedge \varphi_3 \le \varnothing \wedge \varnothing \le \varphi_3$, and denotes that the potential exceptions that may be raised in an abstraction are those that may be raised by evaluating the body e. Moreover, there is no exception raised for defining an abstraction.

The rule APP is more subtle. For an application $e_1\ e_2$ to be well typed, we use a more relaxed relation than requiring the argument type of e_1 to match the type of e_2. We require instead that the type of e_1 be *equivalent* to a function type whose argument type is the type of e_2. This relaxation effectively deals with the fact that choices in type expressions increase the compatibility between types. For example, the expression `odd` $D\langle 1,2\rangle$ should be considered type correct even though the argument type of `odd` (which is `int`) is *not* equal to the type of $D\langle 1,2\rangle$ (which is $D\langle$`int`$,$`int`$\rangle$).

The rule LET for introducing let-polymorphism simply aggregates the annotations obtained from the typing of the subexpressions [40].

The typing for a choice expression is obtained by combining the result for its alternatives. Specifically, we pack corresponding constraints and result types into the choice that we are typing, as expressed by the rule CHOICE.

Similar to the HM(X) framework, we use a subsumption relation to allow a type to be interpreted as some other type, as captured in rule SUB. The semantics of this relation is left unspecified, which thus allows different instances of VHM(X) to specify their interpretations as needed for their particular purposes. For example, in typing Haskell, it should be instantiated to the syntactical equality relation. On the other hand, subsumption can be interpreted as a subtyping relation when subtyping is involved. In any case, we require the relation to satisfy the standard partial ordering axioms, the contra-variance rule for function types, plus the following rules for choice types.

$$\frac{C \Vdash \phi_1 \preceq \phi \qquad C \Vdash \phi_2 \preceq \phi}{C \Vdash D\langle\phi_1,\phi_2\rangle \preceq \phi} \qquad \frac{C \Vdash \phi \preceq \phi_1 \qquad C \Vdash \phi \preceq \phi_2}{C \Vdash \phi \preceq D\langle\phi_1,\phi_2\rangle}$$

$$\frac{C \Vdash \phi_1 \preceq \phi_2 \qquad C \Vdash \phi_3 \preceq \phi_4}{C \Vdash D\langle\phi_1,\phi_3\rangle \preceq D\langle\phi_2,\phi_4\rangle}$$

The rule GEN borrowed from [40], introduces a type schema for a typing judgment. The essential idea is that the constraint is split into two parts, one part (C_2) constrains the free variables in the result type τ and the type environment Γ, while the other (C_1) doesn't. As a result, only C_2 appears as the constraint for the result type schema. (We write $S_1 \# S_2$ to express that two sets S_1 and S_2 are disjoint.) Many rules exist for generalization, but this rule has many advantages over previous ones [29]. The novel part is that $\exists\alpha.C_2$ remains in the left-hand side of the judgment, whose goal is to ensure that the constraint C_2 must be satisfiable. In other words, to generalize a type, the generalized type must at least have one instance. Further details motivating the use of $\exists\alpha.C_2$ were presented in [29].

The rule SIMPLE allows us to hide type variables and simplifies the constraint part of a typing judgment. For example, in the judgment `Eq` $\alpha_1;\Gamma \vdash e : \alpha_2 \to$ `bool`$/\varnothing$, the constraint is not needed, and we can thus transform it into $\exists\alpha_1.$`Eq` $\alpha_1;\Gamma \vdash e : \alpha_2 \to$ `bool`$/\varnothing$. We don't simply drop the constraint `Eq` α_1, because we want to maintain the fact that the constraint `Eq` α_1 must be satisfiable. The rule WEAKEN allows us to strengthen the constraint in a typing judgment while deriving the same result.

We don't have an instantiation rule to eliminate type schemas because the polymorphism is introduced through the let expressions and as a result, we only need to instantiate variable references, which is already realized in VAR.

4.2.1 Constraint Optimization

We observe that in GEN constraints are moved from the assumption into type schemas whereas in VAR constraints are moved in the

opposite order, which means that when a variable is referenced repeatedly, the constraint from the type schema will be copied many times. Since it is important to keep the constraints to a manageable size, rules like the seemingly complicated GEN are needed to split constraints in premises into two parts and move as little as possible to the resulting type schema. Achieving high performance is a main goal of the VHM(X) framework. The entailment relation defined in Figure 2 and the rule WEAKEN offer many opportunities to optimize the representation of constraints. As an example consider the following judgement.

$$D\langle C, \texttt{Eq}\ \alpha\rangle; \Gamma \vdash \texttt{elem x} : [\alpha] \to \texttt{bool}/\varnothing$$

We assume that $\alpha \notin FV(C)$ and that C is a large, complicated constraint. Generalizing this judgment directly with GEN will not leave much room for optimization since the whole constraint $D\langle C, \texttt{Eq}\ \alpha\rangle$ will be copied to the result type schema. However, since C doesn't constrain α, it shouldn't be moved to the type schema. We can achieve a better result by first applying the WEAKEN rule and then using rule C9 from Figure 2. As a result, we first obtain the following judgment.

$$D\langle C, \mathit{true}\rangle \wedge D\langle \mathit{true}, \texttt{Eq}\ \alpha\rangle; \Gamma \vdash \texttt{elem x} : [\alpha] \to \texttt{bool}/\varnothing$$

When we now apply the rule GEN, we obtain the following judgement instead, which preserves the meaning: When the first alternative of the constraint is selected, the type variable α is not constrained and can be instantiated with any type.

$$D\langle C, \mathit{true}\rangle \wedge \exists\alpha.D\langle \mathit{true}, \texttt{Eq}\ \alpha\rangle; \Gamma \vdash$$
$$\texttt{elem x} : \forall\alpha[D\langle \mathit{true}, \texttt{Eq}\ \alpha\rangle].\alpha \to \texttt{bool}/\varnothing$$

4.3 Typing Extensions

To encode a specific analysis, the entailment $\Vdash$ relation among constraints may be extended, which includes an extension of $\equiv$ and $\leq$ as well. Moreover, two components of the typing rules have to be instantiated: the subsumption relation $\preceq$ and the constraint relation $\mathscr{R}$ between annotations that is used in the syntax-directed typing rules. $\mathscr{R}$ consists of four constraint relationships R_x, R_λ, R_{Let}, and R_{App}. Finally, types for extended expression constants have to be provided through the initial environment Γ_0.

For 0CFA, no extension for $\Vdash$, $\equiv$ and $\leq$ is needed. Also, the subsumption relation is interpreted as the type equivalence relation $\equiv$. For $\mathscr{R}$, we get $R_x\ \varphi_1\ \varphi_2 = R_{App}\ \varphi_1\ \varphi_2\ \varphi_3\ \varphi_4 = R_{Let}\ \varphi_1\ \varphi_2\ \varphi_3 = \mathit{true}$, which says that for these cases the constraint always holds. For the ABS rule we have $R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3 = \{l\} \leq \varphi_2$, which records each abstraction in its resulting function type.

As an example, consider again the expression e shown in Section 1.1. We first rewrite it as $e = (e_1\ D\langle e_2, e_3\rangle)\ e_4$. Based on the typing rules in Figure 3, we can conduct a 0CFA for e. We use the following constraint C.

$$C\ =\ \alpha_2 \equiv \alpha \xrightarrow{\{4\}} \alpha\ \wedge\ \alpha_3 \equiv \alpha_1 \xrightarrow{\{6\}} \alpha_1\ \wedge\ \alpha_4 \equiv D\langle \alpha_2, \alpha_3\rangle$$

With C the following judgments are derivable.

$$C; \Gamma \vdash e_1 : (\alpha_2 \xrightarrow{D\langle\{3\},\{5\}\rangle} \alpha_4) \xrightarrow{\{1\}} \alpha_2 \xrightarrow{\{2\}} \alpha_4/\varnothing$$
$$C; \Gamma \vdash D\langle e_2, e_3\rangle : \alpha_2 \xrightarrow{D\langle\{3\},\{5\}\rangle} \alpha_4/\varnothing$$
$$C; \Gamma \vdash e_4 : \alpha_2/\varnothing$$

Now we can apply rule APP twice and obtain the judgment $C; \Gamma \vdash e : \alpha_4/\varnothing$. Based on the annotations for α_4, the result of 0CFA for e is $D\langle\{4\},\{6\}\rangle$, which is the same result we obtained in Section 1.1.

As another example consider instantiating VHM(X) for type inference of lambda calculus with let polymorphism. There is no extension needed here for $\equiv$, $\leq$ and $\Vdash$. We let $\preceq$ be type equality, which means that no extension is needed for $\preceq$. Moreover, since type inference doesn't involve the handling of annotations, we can define each constraint in $\mathscr{R}$ to be *true*.

For the exception analysis, we have to extend the type system as follows. First, for each exception type EX_i, the relation $\Vdash Ex\ EX_i$ is added to convey the fact that EX_i is an exception type. No extensions are needed for $\leq$ and $\equiv$. Next, each constraint that appears in the typing rules (Figure 3) has to be instantiated as follows [28] (we use $\varphi_1 =_\varphi \varphi_2$ to denote $\varphi_1 \leq \varphi_2 \wedge \varphi_2 \leq \varphi_1$).

$$R_x\ \varphi_1\ \varphi_2 = \varphi_2 =_\varphi \varnothing$$
$$R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3 = \varphi_2 =_\varphi \varphi_1 \wedge \varphi_3 =_\varphi \varnothing$$
$$R_{App}\ \varphi_1\ \varphi_2\ \varphi_3\ \varphi_4 = \varphi_4 =_\varphi \varphi_1 \cup \varphi_2 \cup \varphi_3$$
$$R_{Let}\ \varphi_1\ \varphi_2\ \varphi_3 = \varphi_3 =_\varphi \varphi_1 \cup \varphi_2$$

No extension for $\preceq$ is required. The initial type environment should be as follows.

$$\begin{aligned}\Gamma_0 = \{&(ex_i, EX_i),\\ &(\texttt{raise}, \forall\alpha_1\alpha_2[Ex\ \alpha_1].\alpha_1 \xrightarrow{\{\alpha_1\}} \alpha_2),\\ &(\texttt{handle}, \forall\alpha_1\alpha_2\beta_1\beta_2\beta_3[Ex\ \alpha_1 \wedge \beta_3 = \beta_2 \backslash \{\alpha_1\}].\\ &\qquad \alpha_1 \to \alpha_2^{\beta_1} \xrightarrow{\beta_1} \alpha_2^{\beta_2} \xrightarrow{\beta_3} \alpha_2)\}\end{aligned}$$

4.4 Properties

The most important property of VHM(X) is its correctness, that is, by running a lifted analysis on a program family we obtain a synchronized family of analysis results such that the original analysis would yield for any particular program, obtained through a selection with a decision δ, the same result that is obtained from the result family also by selection with δ.

The first step in establishing this result is to show that selection preserves the typing relation provided that the entailment relation is preserved over selection when the instantiated $\mathscr{R}$ constraints are involved. For example, if $C \Vdash R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3$, then $\lfloor C \rfloor_s \Vdash \lfloor R_\lambda\ l\ \varphi_1\ \varphi_2\ \varphi_3 \rfloor_s$ must hold. Moreover, we require that for each type constructor T, the relation $\Vdash \lfloor T^\varphi\ \overline{\phi} \rfloor_s \equiv T^{\lfloor\varphi\rfloor_s}\ \overline{\lfloor\phi\rfloor_s}$ holds.

LEMMA 1. *If* $C; \Gamma \vdash e : \sigma/\varphi$, *then* $\lfloor C \rfloor_s; \lfloor \Gamma \rfloor_s \vdash \lfloor e \rfloor_s : \lfloor \sigma \rfloor_s / \lfloor \varphi \rfloor_s$.

PROOF The proof is by an induction over the typing derivation. Note that for each typing rule in Figure 3, the typing relation is preserved over selection.

We show one proof case for the CHOICE rule, which is established by induction over the structure of s. We show the case $s = D.1$. (The case for $s = D.2$ is analogous, and the case when s is neither $D.1$ nor $D.2$ follows by induction from the definition of selection defined in Section 2.)

Given $C_1; \Gamma \vdash e_1 : \tau_1/\varphi_1$ and $C_2; \Gamma \vdash e_2 : \tau_2/\varphi_2$, the induction hypotheses are that

$$\lfloor C_1 \rfloor_{D.1}; \lfloor \Gamma \rfloor_{D.1} \vdash \lfloor e_1 \rfloor_{D.1} : \lfloor \tau_1 \rfloor_{D.1} / \lfloor \varphi_1 \rfloor_{D.1} \tag{1}$$

and $\lfloor C_2 \rfloor_{D.1}; \lfloor \Gamma \rfloor_{D.1} \vdash \lfloor e_2 \rfloor_{D.1} : \lfloor \tau_2 \rfloor_{D.1} / \lfloor \varphi_2 \rfloor_{D.1}$. We have to show

$$\lfloor D\langle C_1, C_2\rangle \rfloor_{D.1}; \lfloor \Gamma \rfloor_{D.1} \vdash \lfloor D\langle e_1, e_2\rangle \rfloor_{D.1} : \lfloor \tau \rfloor_{D.1} / \lfloor D\langle \varphi_1, \varphi_2\rangle \rfloor_{D.1},$$

which, by definition of selection, can be simplified to the following.

$$\lfloor C_1 \rfloor_{D.1}; \lfloor \Gamma \rfloor_{D.1} \vdash \lfloor e_1 \rfloor_{D.1} : \lfloor \tau \rfloor_{D.1} / \lfloor \varphi_1 \rfloor_{D.1} \tag{2}$$

The additional hypothesis $D\langle C_1, C_2\rangle \Vdash D\langle \tau_1, \tau_2\rangle \equiv \tau$, together with Lemma 2, gives us that $\lfloor D\langle C_1, C_2\rangle \rfloor_{D.1} \Vdash \lfloor D\langle \tau_1, \tau_2\rangle \equiv \tau \rfloor_{D.1}$, which can be simplified to

$$\lfloor C_1 \rfloor_{D.1} \Vdash \lfloor \tau_1 \rfloor_{D.1} \equiv \lfloor \tau \rfloor_{D.1} \tag{3}$$

The result (2) we want to prove follows from (1) and (3). □

LEMMA 2. *If* $C_1 \Vdash C_2$, *then* $\lfloor C_1 \rfloor_s \Vdash \lfloor C_2 \rfloor_s$.

$$\text{I-VAR}\ \frac{\Gamma(x) = \forall\overline{\alpha}\overline{\beta}[C].\tau^{\varphi_1} \qquad \alpha_1 \text{ new} \qquad \beta_1 \text{ new}}{\exists\overline{\alpha}\overline{\beta}.(C \wedge R_x\ \varphi_1\ \beta_1 \wedge \alpha_1 \equiv \tau^{\varphi_1});\Gamma \vdash_I x : \alpha_1/\beta_1}$$

$$\text{I-ABS}\ \frac{C;\Gamma,(x,\alpha_1) \vdash_I e : \alpha_2/\beta_1 \qquad \alpha_3 \text{ new} \qquad \beta_2,\beta_3 \text{ new}}{\exists\alpha_1\alpha_2\beta_1.(C \wedge R_\lambda\ l\ \beta_1\ \beta_2\ \beta_3 \wedge \alpha_3 \equiv \alpha_1 \xrightarrow{\beta_2} \alpha_2);\Gamma \vdash_I \lambda^l x.e : \alpha_3/\beta_3}$$

$$\text{I-APP}\ \frac{C_1;\Gamma \vdash_I e_1 : \alpha_1/\beta_1 \qquad C_2;\Gamma \vdash_I e_2 : \alpha_2/\beta_2 \qquad \alpha_3 \text{ new} \qquad \beta_3,\beta_4 \text{ new}}{\exists\alpha_1\alpha_2\beta_1\beta_2.(C_1 \wedge C_2 \wedge R_{App}\ \beta_1\ \beta_2\ \beta_3\ \beta_4 \wedge \alpha_1 \preceq \alpha_2 \xrightarrow{\beta_3} \alpha_3);\Gamma \vdash_I e_1\ e_2 : \alpha_3/\beta_4}$$

$$\text{I-LET}\ \frac{C_1;\Gamma \vdash_I e : \alpha/\beta \qquad C_2;\Gamma,(x,\forall\alpha\beta[C_1].\alpha) \vdash_I e' : \alpha_1/\beta_1 \qquad \beta_2 \text{ new}}{(\exists\alpha\beta\beta_1.(C_1 \wedge R_{Let}\ \beta\ \beta_1\ \beta_2)) \wedge C_2;\Gamma \vdash_I \textbf{let } x = e \textbf{ in } e' : \alpha_1/\beta_2}$$

$$\text{I-EQU}\ \frac{C_1;\Gamma \vdash_I e : \alpha/\beta \qquad C_1 \Vdash C_2 \qquad C_2 \Vdash C_1}{C_2;\Gamma \vdash_I e : \alpha/\beta}$$

$$\text{I-CHC}\ \frac{C_1;\Gamma \vdash_I e_1 : \alpha_1/\beta_1 \qquad C_2;\Gamma \vdash_I e_2 : \alpha_2/\beta_2 \qquad \alpha_3 \text{ new} \qquad \beta_3 \text{ new}}{\exists\alpha_1\alpha_2\beta_1\beta_2.(D\langle C_1,C_2\rangle \wedge D\langle\alpha_1,\alpha_2\rangle \equiv \alpha_3 \wedge \beta_3 = D\langle\beta_1,\beta_2\rangle);\Gamma \vdash_I D\langle e_1,e_2\rangle : \alpha_3/\beta_3}$$

Figure 4. Type inference for VHM(X)

PROOF By induction over the structure of C. □

Note that the reverse of Lemma 1 also holds. In expressing this result we use the notation $\Gamma_1 \uplus_D \Gamma_2$ to denote an environment Γ_3, for which $\lfloor\Gamma_3(x)\rfloor_{D.1} = \Gamma_1(x)$ and $\lfloor\Gamma_3(x)\rfloor_{D.2} = \Gamma_2(x)$ (for any variable x).

LEMMA 3. *If $C_1;\Gamma_1 \vdash e_1 : \sigma_1/\varphi_1$ and $C_2;\Gamma_2 \vdash e_2 : \sigma_2/\varphi_2$, then $D\langle C_1,C_2\rangle;\Gamma_1 \uplus_D \Gamma_2 \vdash D\langle e_1,e_2\rangle : \sigma_3/D\langle\varphi_1,\varphi_2\rangle$ is derivable and $\lfloor\sigma_3\rfloor_{D.1} = \sigma_1$ and $\lfloor\sigma_3\rfloor_{D.2} = \sigma_2$.*

PROOF Proof by contradiction with the help of Lemma 1. □

Based on Lemma 1, we can now state the following projection theorem. Any analysis for a plain program (that was selected from a program family) can be obtained from the corresponding family of results produced by the lifted analysis for the program family. We use δ to range over *complete* decisions, which are sets of selectors that eliminate all the choices in C, Γ, σ and φ.

THEOREM 1 (Projection). *Given $C;\Gamma \vdash e : \sigma/\varphi$, then $\forall(\delta,e') \in [\![e]\!]$, $\lfloor C\rfloor_\delta;\lfloor\Gamma\rfloor_\delta \vdash e' : \sigma'/\varphi'$, where $\sigma' = [\![\sigma']\!](\delta)$ and $\varphi' = [\![\varphi]\!](\delta)$.*

PROOF The proof is based on an induction over the structures of C, Γ, σ and φ, with the help of Lemma 1. □

Finally, we observe that, disregarding annotations, VHM(X) and HM(X) compute the same result in the absence of choices. Using $\vdash^{HM}$ to denote the typing relation of HM(X) [40], we have the following result.

LEMMA 4. *Given C, Γ and e plain, if $C;\Gamma \vdash e : \sigma/\varphi$, then $C;\Gamma \vdash^{HM} e : \sigma'$, where σ' can be obtained by eliminating annotations from σ.*

PROOF The proof can be established through an induction over the typing derivation based on the typing rules in Figure 3 and the rules in [40]. Moreover, observe that in absence of variation constructs, the $\equiv$ relation degenerates to type equality and the subsumption relation $\preceq$ introduced in Section 4.2 degenerates to the same subsumption relation as in [40]. □

5. Parametric Type Inference

In the specification of type-based analysis, constraints C serve as an input. However, in implementing the analysis, we need to dynamically generate and solve constraints. In this section, we discuss constraint generation; constraint solving will follow in Section 6.

In Figure 4 we present the inference rules for the judgment $C;\Gamma \vdash_I e : \alpha/\beta$, where Γ and e are the input and C, α, and β are the output. The presentation is an adaptation and extension of the type inference rules for HM(X) [40] to accommodate the handling of annotations and choices and uses the parametric constraints $\mathscr{R}$ introduced in Section 4.2.

The inference rules are derived from the corresponding typing rules (Figure 3) by moving the constraints in the premise to the left-hand side of the judgment in the conclusion. The rules gather constraints from subexpressions and add them to those of their parents to ensure correctness, that is, no conditions will be ignored. The minimality of constraints and thus the genrality of the inferred result follow from the fact that each rule only integrates constraints that are required for the particular construct under consideration.

For example, in I-VAR, the conclusion can be read as "when the constraint C is satisfied, x can have any type". Note that τ^{φ_1} may contain reference to $\overline{\alpha}\overline{\beta}$, which is why the constraint $\alpha_1 \equiv \tau^{\beta_1}$ appears inside the quantification $\exists\overline{\alpha}\overline{\beta}$. To consider another example, in rule I-APP the constraints from subexpressions, C_1 and C_2, the constraint for annotations $R_{App}\ \beta_1\ \beta_2\ \beta_3\ \beta_4$, and the constraint between the types of e_1, e_2, and the result type of the application $e_1\ e_2$, are simply collected in the constraint of the conclusion. The rules follow a similar pattern as the typing rules in Figure 3.

Note that there is only one non-syntax directed rule, I-EQU, which can be applied any time. The purpose of this rule is to keep the size of the constraint as small as possible by employing the relationships defined in Figure 2.

Given the inference rules in Figure 4, we can generate the following constraint C_1 for the expression $e_1 = \lambda^1 h.\lambda^2 x.h\ x$, which is a part of the expression e introduced in Section 1.1. The expression e_1 then has the type α_7 under the constraint C_1.

$$C_1 = \exists\alpha_1\alpha_2\alpha_3\alpha_4\alpha_5\alpha_6(\alpha_1 \equiv \alpha_3 \wedge \alpha_2 \equiv \alpha_4 \wedge \alpha_3 \equiv \alpha_4 \xrightarrow{\beta_1} \alpha_5 \wedge \{2\} \leq \beta_2 \wedge \alpha_6 \equiv \alpha_2 \xrightarrow{\beta_2} \alpha_5 \wedge \{1\} \leq \beta_3 \wedge \alpha_7 \equiv \alpha_1 \xrightarrow{\beta_3} \alpha_6)$$

The constraints for other parts of the expression e can be generated similarly and are omitted here.

We now investigate the relation between type inference rules in Figure 4 and specification rules defined in 3. First, type inference is sound, as stated in the following theorem.

THEOREM 2. *If $C;\Gamma \vdash_I e : \alpha/\beta$, then $C;\Gamma \vdash e : \alpha/\beta$.*

PROOF The proof is based on an induction over the structure of the expression e. □

Also, type inference is complete and principal and least constrained in the sense that the constraint generated by the relation $\vdash_I$ is minimal while satisfying the typing judgment discussed in Sec-

$\mathscr{V} : C \times \theta \to C \times \theta$

(a) $\mathscr{V}(\exists\alpha\beta.C, \theta) = (C_1, \theta_1 \backslash \{\alpha, \beta\})$ when $\{\alpha, \beta\}$ # $vars(\theta)$
where $(C_1, \theta_1) = \mathscr{V}(C, \theta)$

(b) $\mathscr{V}(D\langle C_1, C_2\rangle, \theta) = (D\langle C_3, C_4\rangle, \theta_3 \sqcup_D \theta_4)$
where $(C_3, \theta_3) = \mathscr{V}(C_1, \theta)$
$(C_4, \theta_4) = \mathscr{V}(C_2, \theta)$

(c) $\mathscr{V}(D\langle C, C\rangle, \theta) = \mathscr{V}(C, \theta)$

(d) $\mathscr{V}(C_1 \wedge C_2, \theta) = (C_3 \wedge C_4, \theta_4)$ when C_1 or C_2 not plain
where $(C_3, \theta_3) = \mathscr{V}(C_1, \theta)$
$(C_4, \theta_4) = \mathscr{V}(\theta_3(C_2), \theta_3)$

(e) $\mathscr{V}(D\langle \phi_1, \phi_2\rangle \equiv D\langle \phi_3, \phi_4\rangle, \theta) = \mathscr{V}(D\langle \phi_1 \equiv \phi_3, \phi_2 \equiv \phi_4\rangle, \theta)$

(f) $\mathscr{V}^\star(D\langle \phi_1, \phi_2\rangle \equiv \phi_3, \theta) = \mathscr{V}(D\langle \phi_1 \equiv \phi_3, \phi_2 \equiv \phi_3\rangle, \theta)$

(g) $\mathscr{V}^\star(\alpha \equiv \phi, \theta) = (true, \{(\alpha, \phi)\} \circ \theta)$ when $\alpha \notin FV(\phi)$

(h) $\mathscr{V}(\phi_1 \equiv \phi_2, \theta) =$ when $D \in dims(\phi_1, \phi_2)$
$\mathscr{V}(D\langle \lfloor \phi_1 \rfloor_{D.1} \equiv \lfloor \phi_2 \rfloor_{D.1}, \lfloor \phi_1 \rfloor_{D.2} \equiv \lfloor \phi_2 \rfloor_{D.2}\rangle, \theta)$

(i) $\mathscr{V}(\phi_1 \equiv \phi_2, \theta) = \mathscr{U}(\phi_1 \equiv \phi_2, \theta)$ when ϕ_1 and ϕ_2 plain

(j) $\mathscr{V}(P, \theta) = \mathscr{U}(P, \theta)$ when P analysis specific

Figure 5. Variational constraint solving

tion 4.2. We express this result in the following theorem, where we extend the definition of $\preceq$ to type schemas in the theorem, following a standard definition in [29].

THEOREM 3. *Given* $C; \Gamma \vdash e : \sigma/\varphi$, *then* $C'; \Gamma \vdash_I e : \alpha/\beta$ *such that* $C \Vdash \exists\alpha\beta.C'$, $C \Vdash \forall\alpha\beta[C'].\alpha \preceq \sigma$ *and* $C \Vdash \beta \leq \varphi$.

PROOF The proof is based on an induction of the derivation tree of $C; \Gamma \vdash e : \sigma/\varphi$. □

6. Variational Constraint Solving

To compute an analysis result, we need to solve the constraints generated in the type inference phase. Since different analyses will introduce quite different constraints, there will be no single constraint solving algorithm that solves all generated constraints. Therefore, our algorithm will defer to an application-specific solver $\mathscr{U}$ to handle constraints that are specific to a particular analysis. $\mathscr{U}$ has to be provided by the implementor of the static analysis, but there is potential for the reuse of one solver for different analyses.

6.1 A Constraint Solving Algorithm

We say a constraint is *primitive* if it doesn't contain choices or existential quantification. In this section, we will use P to range over primitive constraints.

We assume that $\mathscr{U}$ can be applied to a given set of constraints C_1 and a given substitution θ_1 and returns a residual constraint C_2 together with substitution θ_2. We require that $\theta_2(C_2) = C_2$ and $C_2 \Vdash \theta_2(C_1)$. Note that constraint solving also involves the manipulation of mappings from β to φ. These can be treated like substitutions and are thus stored in θ as well.

Based on $\mathscr{U}$, we build a variational constraint solving algorithm $\mathscr{V}$, shown in Figure 5, which solves the core constraints defined in Figure 1. The signature of $\mathscr{V}$ is the same as that for $\mathscr{U}$. In the following we will briefly discuss the different cases. When none of the cases and conditions apply, the constraint solver fails, which indicates that the analysis for the particular program family cannot produce a result. It would be nice if VHM(X) could produce partial results in the presence of unsolvable constraints for some program variants. We can envision a corresponding extension following the approach presented in [9].

Case (a) deals with existential constraints $\exists\alpha\beta.C$, which can be solved by solving C and then removing the mappings for α and β from the result substitution (indicated in Figure 5 by using the notation $\theta_1 \backslash \{\alpha, \beta\}$). The condition $\{\alpha, \beta\}$ # $vars(\theta)$ ensures that the mappings for α and β will not be removed in θ, where $vars(\theta)$ compute the domain of θ and all free variables in θ. Thus, to make this rule applicable, we may have to first rename the bound variables α and β. Note that when C is unsatisfiable, then so is $\exists\alpha\beta.C$. Note also that our rule to solve existential constraints is simpler than the one given in [40] because our constraints are only based on boolean algebra and not cylindric algebra.

Case (b) shows that solving a variational constraints $D\langle C_1, C_2\rangle$ requires solving each alternative. The potentially different residual constraints are then combined again in a choice. The potentially differing substitutions are combined with the operation $\sqcup$, which is defined as follows.

$$\theta_1 \sqcup_D \theta_2(\alpha) = \begin{cases} D\langle \theta_1(\alpha), \theta_2(\alpha)\rangle & \alpha \in dom(\theta_1) \wedge \alpha \in dom(\theta_2) \\ D\langle \theta_1(\alpha), \alpha_1\rangle & \alpha \in dom(\theta_1) \wedge \alpha_1 \text{fresh} \\ D\langle \alpha_1, \theta_2(\alpha)\rangle & \alpha \in dom(\theta_2) \wedge \alpha_1 \text{fresh} \\ \alpha & \text{otherwise} \end{cases}$$

The reason for generating fresh type variables is to reflect the fact that an alternative of a choice might not have been constrained. For example, when $\alpha \notin dom(\theta_2)$, the second alternative will be replaced with a fresh type variable, which allows the second alternative to have any type.

Case (c) allows us to save work when constraints of two alternatives of a choice are the same, and case (d) deals with constraints of the form $C_1 \wedge C_2$, where either C_1 or C_2 is not plain. (Otherwise, the constraints should be dealt with by $\mathscr{U}$.) A conjunction is solved in sequence by threading updated substitutions.

The next group of cases (e) through (i) deals with type equivalence constraints of the form $\phi_1 \equiv \phi_2$. Case (e) handles choices in the same dimension, which reduces to solving the equivalence constraint between the corresponding alternatives. In case (f) one side is a choice while the other side is not (or a choice in a different dimension). In these cases, both alternatives of D are required to be equivalent with the type on the other side. The $\star$ attached to $\mathscr{V}$ in this case (and the next) indicates that there is a dual case that can be handled correspondingly, where the two arguments to $\equiv$ are swapped. Rule (g) deals with the case that one side is a type variable α, which succeeds by extending the substitution.

The situation becomes a little more complicated when the two types are of different form and contain choices, which is addressed by the rule (h). The general idea is to eliminate choices in the types so that they become simpler, which may lead to other rules being applicable. We achieve this by making selections into the types and and create a choice constraint. (The function *dims* determines the set of dimension names contained in types and expressions.)

A potentially more efficient strategy is to inspect the structures of the types and take the appropriate actions. For example, if the constraint is of the form $T^{\varphi_1}\,\overline{\phi_1} \equiv T^{\varphi_2}\,\overline{\phi_2}$, we can instead solve the constraints $\varphi_1 = \varphi_2 \wedge \overline{\phi_1 \equiv \phi_2}$. However, this works only when the constructor T forms a free algebra, that is, has no associated equational theory; it fails, for example, when T is commutative.

For unifying two plain types, we dispatch the task to the underlying solver $\mathscr{U}$, as expressed in rule (i). Solving the constraints between two annotations of the form $\varphi_1 \leq \varphi_2$ is similar to solving the type equivalence constraint and will not be repeated here.

Finally, when a constraint is domain specific, it is solved by the underlying solver $\mathscr{U}$, as shown in case (j).

With the constraint solving algorithm, we derive the following solution θ_1 for the constraint C_1 generated in Section 5.

$$\theta_1 = \{\alpha_7 \mapsto (\alpha_4 \xrightarrow{\beta_1} \alpha_5) \xrightarrow{\{1\}} \alpha_4 \xrightarrow{\{2\}} \alpha_5, \beta_2 \mapsto \{2\}, \beta_3 \mapsto \{1\}\}$$

Note that $\theta_1(\alpha_7)$ is also the inferred type for the expression e_1 $(\lambda^1 h.\lambda^2 x.h\,x)$. With a little extra work, we can generate and solve

constraints for the other parts of the expression e (introduced in Section 1.1) and verify that the inference result is the same as the result presented in Section 4.3 for e.

6.2 Relation to Variational Unification

Even though the variational constraint solving algorithm presented in Figure 5 is more general than the variational unification algorithm developed in [10], it is also simpler. We will use the following example to illustrate the differences. Note that in [10] we use the notation $\equiv^?$, instead of $\equiv$, to denote a unification problem.

$$A\langle \texttt{Int}, \alpha \rangle \equiv A\langle \alpha, \texttt{Bool} \rangle$$

The variational unification algorithm presented in [10] consists of three steps:

(a) qualify the unification problem by attaching to each type variable a path of selectors for enclosing choices,

(b) solve the qualified unification problem with a unifier for the qualified problem, and

(c) complete the unifier obtained in step (b) to a unifier for the original unification problem.

In the given example, we get for step (a) the following unification problem.

$$A\langle \texttt{Int}, \alpha_{A.2} \rangle \equiv A\langle \alpha_{A.1}, \texttt{Bool} \rangle$$

We then obtain the following qualified unifier for step (b).

$$\{\alpha_{A.1} \mapsto \texttt{Int}, \alpha_{A.2} \mapsto \texttt{Bool}\}$$

Finally, step (c) yields the completed unifier $\{\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool} \rangle\}$. The decomposition of the algorithm into three steps poses a big challenge to both the implementation and the correctness proof of the algorithm.

Our constraint solving algorithm in Figure 5 simplifies the previous algorithm through the use of *context splitting* (cases (e) and (f)) and *context merging* (through the operation $\theta_1 \sqcup_D \theta_2$). Following this idea, we derive for the problem $\texttt{Int} \equiv \alpha$ the solution $\theta_3 = \{\alpha \mapsto \texttt{Int}\}$ in the context of the first alternative of A and $\theta_4 = \{\alpha \mapsto \texttt{Bool}\}$ in the context of the second alternative of A. The operation $\theta_3 \sqcup_D \theta_4$ produces the expected solution $\{\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool} \rangle\}$.

The constraint solving algorithm in Figure 5 has the time complexity $O(mn)$ when it is instantiated with Robinson's unification algorithm, where m and n are the size of the left and right constraint, respectively. This is the same as the time complexity of step (b) of the previous variational unification algorithm. The implementation of the new algorithm, however, is much simpler. Moreover, proving the properties of the algorithm is also easier. We'll do this next.

6.3 Properties

The solver $\mathscr{V}$ inherits desirable properties of $\mathscr{U}$. For example, $\mathscr{V}$ is sound and principal provided that $\mathscr{U}$ is. We say (C_1, θ_1) is principal for (C, θ) if $C_1 \Vdash \theta_1(C)$ and for any other (C_2, θ_2) such that $C_2 \Vdash \theta_2(C)$ implies $\theta_1 \sqsubseteq \theta_2$ and $C_2 \Vdash \theta_2(C_1)$. Here $\theta_1 \sqsubseteq \theta_2$ holds if there is some θ_3 such that $\theta_2 = \theta_3 \circ \theta_1$. The definition is similar for the case of primitive constraints.

THEOREM 4 (Soundness of $\mathscr{V}$).
If $(P', \theta'_p) = \mathscr{U}(P, \theta_p)$ *implies* $P' \Vdash \theta'_p(P)$ *and* $\theta'_p(P') = P'$, *then* $(C', \theta') = \mathscr{V}(C, \theta)$ *implies* $C' \Vdash \theta'(C)$ *and* $\theta'(C') = C'$.

THEOREM 5 (Principality of $\mathscr{V}$).
If $(P', \theta'_p) = \mathscr{U}(P, \theta_p)$ *implies* (P', θ'_p) *is principal for* (P, θ_p), *then* $(C', \theta') = \mathscr{V}(C, \theta)$ *implies* (C', θ') *is principal for* (C, θ).

PROOF The proof for both theorems is based on the induction over the structures of constraints C. □

```
-- Module F1                          -- Module F2
class A extends Object {              class A extends Object {
    D mc(Object a) {                      F mc(Object a) {
        return new D() ;                      return new F() ;
    }                                     }
    C ma(C e) {                           E ma(E e) {
        return new C() ;                      return new E() ;
    }                                     }
}                                     }

class C extends Object {}             class E extends Object {}
class D extends C {}                  class F extends E {}


-- Module F3, which requires exactly either F1 or F2
Object test2 = (new A()).ma ((new A()).mc(new Object())) ;


-- Output of FFJPL
*** Exception: Type error: Method invocation:
new A().ma(new A().mc(new Object())) is not well-formed!


-- Output of VHMX(X)
The SPL is well typed.
```

Figure 6. Discrepancy between FFJPL and VHM(X).

Thus the solver $\mathscr{V}$ has the same capability as $\mathscr{U}$. For example, if $\mathscr{U}$ is the Robinson unification algorithm, then $\mathscr{V}$ is a variational unification algorithm as developed in [10]. If $\mathscr{U}$ is the $\mathscr{U}_{\text{CFA}}$ algorithm [28] for 0CFA analysis, then $\mathscr{V}$ is the solver for variational 0CFA analysis.

We conjecture that $\mathscr{V}$ and $\mathscr{U}$ are in the same complexity class. For example, when $\mathscr{U}$ is decidable, then so is $\mathscr{V}$. Also, when $\mathscr{U}$ is semi-decidable, then so is $\mathscr{V}$. We leave this for future work.

7. Evaluation

Does our framework make true on its promise to increase the accuracy of analysis lifting under an acceptable amount of overhead? We address this question in the following two subsections.

7.1 Reliability of Analysis Lifting

To evaluate the accuracy of VHM(X), we could compare its output with our previously developed variational type inference. However, it might be more informative if the evaluation is conducted by comparing VHM(X) with tools developed by other researchers. Since most variability-aware analyses have been done for imperative languages, we have choosen the FFJPL (Feature Featherweight Java Product Lines) system[2] developed by Apel et al. [1] as our evaluation counterpart. Another advantage of using FFJPL is that it demonstrates that our lifting framework is not tied to the specific calculus presented in this paper. Thus we have adopted our framework to Featherweight Java (FJ) and refer to this version as VFJ(X).

The idea of FFJPL is to allow each module to define new classes, extend, or refine classes defined in another module. Each FFJPL program consists of a set of modules and a feature model, which describes how modules may be combined together. To derive a particular program, decisions about which modules to select have to be made. All selected modules constitute the product.

For example, Figure 6 presents a small FFJPL program, which consists of three modules `F1`, `F2`, and `F3`. The feature model (not shown here) requires that the presence of `F3` requires exactly one of `F1` or `F2`. In both modules `F1` and `F2`, we define a class `A` with the same methods `mc` and `ma`. However, note that their signatures are different in different modules. Module `F3` contains a single statement for creating new objects. This product line contains 2 valid products, one consisting of modules `F1` and `F3`, and the other

[2] http://www.fosd.de/ffj

consisting of modules `F2` and `F3`. It is easy to check that each product is well typed. According to the completeness theorem in [1], a product line is well typed if all valid products are well typed. Thus, the product line in Figure 6 should be well typed.

However, while our VFJ(TC) (VFJ(X) instantiated to type checking) correctly reports that the product line is well typed, FFJPL reports, incorrectly, a type error. We have not attempted to debug their type system or implementation. The important lesson here is that the general lifting framework VFJ(TC) gets it right while the hand-crafted analysis is erroneous.

Of course, this anecdotal piece of evidence does not prove the superiority of VFJ(X) or VHM(X) in general, but it is a reflection of the fact that the stratified approach that requires fewer definitions and much less implementation effort is more likely to be correct.[3]

For illustration, we present the typing rule for method invocations in both systems.

T-INVK-FFJPL

$$\frac{\Gamma \vdash t_0 : \overline{E} \dashv \Phi \qquad \forall E \in \overline{E} : \mathit{validref}(\Phi, E.m) \qquad \Gamma \vdash \overline{t : \overline{G}} \dashv \Phi \qquad \mathit{mtype}(\Phi, m, \overline{\mathit{last}(E)}) = \overline{\overline{H} \to F} \qquad \forall \overline{G} \in \overline{\overline{G}}, \forall \overline{H} \in \overline{\overline{H}} : \overline{G <: H}}{\Gamma \vdash t_0.m(\overline{t}) : F_{11}, \ldots, F_{n1}, \ldots, F_{1m}, \ldots, F_{nm} \vdash \Phi}$$

T-INVK-VFFJX

$$\frac{\begin{array}{c} C; \Gamma \vdash t_0 : E \dashv \Phi / \varphi_1 \\ \mathit{validrefc}(\Phi, E.m) \qquad C; \Gamma \vdash \overline{t} : G \dashv \Phi / \varphi_2 \\ G = G_1 \times \cdots \times G_n \qquad \mathit{mtypec}(\Phi, m, \mathit{lastc}(E)) = H \to^{\varphi_3} F \\ C \Vdash G \preceq H \dashv \Phi \qquad C \Vdash R_{Invk}\ \varphi_1\ \varphi_2\ \varphi_3\ \varphi_4 \end{array}}{C; \Gamma \vdash t_0.m(\overline{t}) : F \dashv \Phi / \varphi_4}$$

The rule T-INVK-FFJPL is a reproduction of the T-INVK$_{PL}$ rule in [1] except for renaming B, C, and D to F, G, and H, respectively, to avoid notational conflicts. The typing relation $\Gamma \vdash t : \overline{E} \dashv \Phi$ says that when feature Φ is chosen (similar to the notion about applying a selector to an expression), the expression t can have any potential mutually exclusive class type in $\overline{E}$. An expression can have more than one type because the same expression can have different types when different modules are selected.

We will not analyze each part of this rule. However, a notable difference is how types are represented. While FFJPL uses a flat list to represent a set of alternative types, VFJ(X) uses nested choice types to represent alternative types.

The bottom line is this. An important merit of the lifting framework is that the correctness of a variational analysis depends only on the easier to establish correctness of a single analysis (and its correct embedding in the framework).

7.2 Framework Runtime Overhead

We would like to know the overhead incurred by the general machineries of the lifting framework over manually lifted variational analysis algorithms. To this end, we have implemented VHM(X) and VFJ(X) (mentioned in Section 7.1) in Haskell and compared three instantiations with hand-crafted analyses.

First, we compared the FFJPL implementation with VFJ(TC) for 5 product lines (1 came with the FFJPL implementation and four others were created by us). The running time of VFJ(TC) is at most 32% slower than the FFJPL implementation.

Due to the limited number of available FFJPL product lines, we also compared the performance of VHM(X) with our own previously developed variational type inference algorithm for Variational Lambda Calculus (VLC) [10] and a manually created variational control-flow analysis algorithm.

[3] The definition of FFJPL is given in 26 rules, while the VFJ(X) extension requires only 6 rules. Moreover, the FFJPL implementation of the typing rules takes 250 LOC, while the VFJ(X) extension takes 30 LOC.

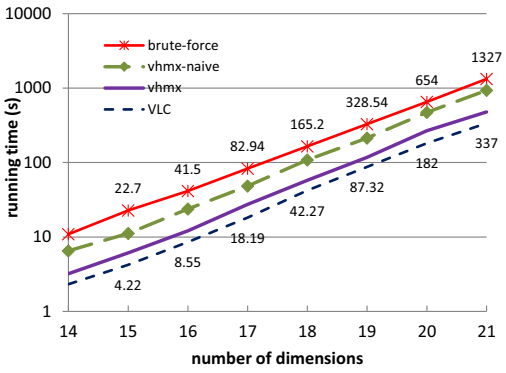

Figure 7. Efficiency comparison between VHM(X) and VLC

For the definition of VHM(TyInf), that is, VHM(X) instantiated for type inference, no extensions for $\mathscr{T}$, $\mathscr{A}$, $\mathscr{C}$ and $\preceq$ are needed; each element of $\mathscr{R}$ is set to *true*, and $\mathscr{U}$ is the Robinson unification algorithm [36]. To instantiate VHM(X) to VHM(TyInf), 9 LOC of Haskell are needed (to declare the class instance to specify $\mathscr{R}$). On the other hand, VLC has over 220 LOC of Haskell. (Here, we exclude the code for the Robinson unification algorithm that is common to both implementations.)

We show the performance for the two analyses in Figure 7. All times have been measured on a laptop with a 2.8 GHz dual core processor and 3GB memory running GHC 7.0.2 on Windows XP. We reuse the test cases from [10] that represent the worst-case performance for VLC. The X-axis denotes the number of dimensions. For these cases VLC doesn't do much better than the brute-force approach of generating and checking each variant individually. This is because there is no sharing in the expressions. For example, the expression with 21 dimensions essentially represents 2^{21} different variants by using a lot of abstractions.

The graph labeled "vhmx-naive" represents the performance of the most conservative strategy for solving type equivalence constraints using rule (h) from Figure 5. When two types are non-plain and the root of the types are not choices, the constraint is solved by splitting it into two constraints. Although this ensures correctness, it misses an opportunity for sharing. For example, given the constraint $\texttt{int} \to \texttt{bool} \equiv \texttt{int} \to D\langle \texttt{bool}, \alpha \rangle$, using (h), this will leads to two subproblems $\texttt{int} \to \texttt{bool} \equiv \texttt{int} \to \texttt{bool}$ and $\texttt{int} \to \texttt{bool} \equiv \texttt{int} \to \alpha$.

An alternative, more aggressive approach exploits the fact that two function types are equivalent if their corresponding argument and return types are equivalent. This means that we can decompose the above constraint into subproblems $\texttt{int} \equiv \texttt{int}$ and $\texttt{bool} \equiv D\langle \texttt{bool}, \alpha \rangle$, which are more efficient to solve. The graph labeled "vhmx" in Figure 7 shows the performance of an implementation of VHM(X) that employs this strategy.

We can observe that for the "vhmx" implementation the slowdown of VHM(X) over VLC is bounded by a factor of 2. This is also the case for non-worst-case expressions that offer more opportunities for sharing.

In Figure 8, we show another performance comparison between VHM(X), VLC, and the brute-force approach. The expressions used for Figure 8 are created from the expression used in Figure 7 whose number of dimensions is 21 by expanding the expressions by adding choice alternatives. Note that we only show part of the brute-force curve because the running time of it grows exponentially fast in the size of expressions, and showing its whole curve will make the difference between "vhmx" and VLC indiscernible. Again we observe that the running time of "vhmx" is within a factor of 2 of VLC, demonstrating that the price we pay for the flexibil-

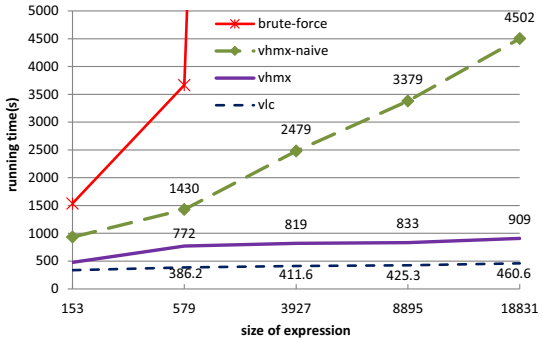

Figure 8. Efficiency comparison between VHM(X) and VLC against expression sizes.

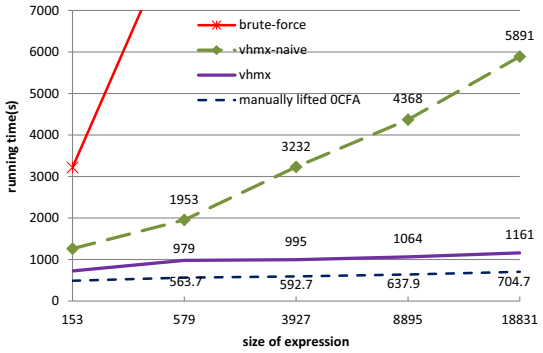

Figure 9. Efficiency comparison between VHM(X) and manually created 0CFA for variational programs

ity is acceptable. An interesting phenomenon in Figure 8 is that there is a more conspicuous discrepancy between the performance of "vhmx" and "vhmx-naive" than in Figure 7. One potential reason for this is that there are more opportunities for sharing for the expressions in Figure 8 than for those in Figure 7, and while "vhmx" takes the full advantage of these opportunities, "vhmx-naive" fails to do so.

Finally, we have also compared the performance of VHM(0CFA) and a manually lifted 0CFA. For this analysis no extension for $\mathscr{T}$, $\mathscr{A}$, $\mathscr{C}$ and $\preceq$ are needed. For $\mathscr{R}$, we only need $R_x\ l\ \varphi_1\ \varphi_2\ \varphi_3 = \{l\} \leq \varphi_2$ as discussed in Section 4.2. For $\mathscr{U}$, we use the unification algorithm presented in [28]. Again, instantiating VHM(X) for VHM(0CFA) requires 9 LOC whereas the hand-written 0CFA requires over 240 LOC in Haskell.

Figure 9 shows the running times of the brute-force approach that generates each program variant and applies 0CFA to each variant separately, VHM(0CFA) and the manually lifted 0CFA. Note that, like in Figure 8, we can't show the whole curve of the brute-force approach. To give a sense of the complexity, it takes the brute-force approach about 10 days to finish the case when the size is 18831. We observe that the slowdown of VHM(0CFA) over the manually created variational 0CFA is bounded by 1.73.

One possible reason for the smaller overhead for VHM(0CFA) compared to VHM(TyInf) is that more of the infrastructure of VHM(X) is used in 0CFA than in type inference. We conjecture that the overhead will be even lower for more complicated analyses, but we leave this question for future work to verify.

8. Related Work

The idea of this paper is inspired by the work of HM(X) [29, 40] and recent numerous efforts to verify properties of software product lines [1, 2, 5, 10–12, 14, 17, 22, 24, 25]. In this section, we compare VHM(X) with the most closely related work.

HM(X) and extensions Odersky et al. [29] formalized an extensible constraint-based type inference framework for Hindley-Milner-style type systems. Later, Sulzmann et al. [40] reformulated and improved the HM(X) framework by shifting the type descriptions from a type language to a constraint language.

We have built VHM(X) on the constraint-based HM(X) framework [40] and have extended it by adding labels to abstractions and annotations to types. The computation of annotations is expressed through relations between annotations attached to types and typing derivations, which are supplied as arguments for correspondingly parameterized syntax-directed typing rules. As a result, VHM(X) can encode more static analyses than HM(X). Second, and most importantly, we have introduced choices to all components of static analyses (types, constraints, and annotations), which allows us to lift static analyses to variational programs.

Type-based static analysis Type-based analysis, together with dataflow analysis, constraint-based analysis and abstract interpretation, are the main static analysis approaches [28].

Several type-based static analysis frameworks have been proposed. For example, the framework developed by Hankin and Métayer [20] is based on untyped lambda calculus and allows users to extend type as well as expression definitions. The biggest difference to VHM(X) is the way in which analyses are encoded. In VHM(X), we use type annotations whereas in their system this is done by interpreting types as specific sets of values (which requires users to also specify mappings from types to sets of values when encoding a static analysis). This makes it impossible to attach information to typing derivations, making the encoding of side-effect analysis and exception analysis close to impossible.

Instead of building an extensible analysis framework, Prose [34] took another approach by formalizing static analyses in a variant of System F [42]. The expressiveness of System F, together with the extensions, makes the proposed approach very expressive: some static analyses can be directly encoded without any extension needed from users. However, there are also some drawbacks. First, the problem of type checking System F is undecidable [42], which makes encoded static analyses undecidable as well. Second, the extraction of useful information is program specific and not analysis specific, that is, even for the same analysis different information has to be supplied to derive the analysis results for different programs. For example, to find dead expressions in a program, users first have to find the substitution such that the most general derivation tree for the program is preserved, which is not a simple task. Finally, with the limitation to two universes $\bot$ and $\top$, some static analyses are hard to embed. For example, it is not clear how to formalize effect and exception analysis.

The most important difference between VHM(X) and other frameworks is that instead of encoding analyses for single programs, VHM(X) lifts static analyses to program families.

Analyses for program families The software product line community has developed numerous variability-aware static analysis approaches (see citations above). A crucial difference to VHM(X) is that VHM(X) is a generic framework that can be instantiated to different analyses.

The main idea of variability-aware analysis is to take special actions when encountering variation constructs so that a whole program family can be checked directly. Liebig et al. [25] discussed that the principle for variability-aware analysis is to keep variability

local and follow the ideas of late splitting and early joining. For example, when typing the expression `id` $D\langle$`succ,odd`$\rangle$ `3`, late splitting means we shouldn't separate the typing before we encounter the choice $D\langle$`succ,odd`$\rangle$. Thus, `id` should only be type checked once. Similarly, early joining ensures `3` is also only checked once. In [10] we have identified that *sharing* and *reduction* are the main factors for improving performance in variational type inference. Sharing corresponds to the principle of keeping variability local. Reduction means to replace a choice whose alternatives have the same type with either alternative. In this paper, we have exploited the idea of sharing and reduction to make VHM(X) efficient.

Bodden [3] has described the automatic lifting of static analyses to variational programs. His approach applies to dataflow analyses that are based on the IFDS framework [35]. However, a formal relation between the lifted analyses and the original analyses is not given. In contrast, analyses that are lifted within the VHM(X) framework provably retain their correctness for program families.

Choice types The concept of named choices was introduced in [18] as a basis for a unified and principled representation for software variations. This choice representation has facilitated a method for variational type inference [9, 10]. The idea of choice types was also adopted by Kästner et al. [24] to implement an efficient type checker for C programs with compilation macros and by Liebig et al. [25, 26] for implementing scalable type checking and dataflow analysis. Recently, we have successfully employed choice types for providing better feedback when type inference fails [7, 8].

9. Conclusions

Observing a growing need for static analyses for program families, we have developed the framework VHM(X) that supports the automatic generation of such analyses from single-program analyses. The two major advantages of our approach are the ability to reuse much of the computation infrastructure for new analyses and a correctness assurance (lifted analyses work correctly if the original analyses do). The presented framework helps to separate the concerns of static analysis and program variability, that is, any new program analysis can first be developed for the single-program case and then automatically lifted to work on program families.

Acknowledgments

This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

References

[1] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Software Engineering*, pages 482–491, 2013.

[3] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 355–364, 2013.

[4] C. Brabrand, M. Ribeiro, T. Toldo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Transactions on Aspect-Oriented Software Development X*, pages 73–108. 2013.

[5] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Int. Conf. on Aspect-Oriented Software Development*, pages 13–24, 2012.

[6] L. Cardelli. Program fragments, linking, and modularization. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 266–277, 1997.

[7] S. Chen and M. Erwig. Counter-Factual Typing for Debugging Type Errors. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 583–594, 2014.

[8] S. Chen and M. Erwig. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming*, LNCS 8475, pages 35–51, 2014.

[9] S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM Int. Conf. on Functional Programming*, pages 29–40, 2012.

[10] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 36(1):1:1–1:54, 2014.

[11] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 321–330, 2011.

[12] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 335–344, 2010.

[13] P. C. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2001.

[14] M. Cordy, A. Classen, G. Perrouin, P.-Y. Schobbens, P. Heymans, and A. Legay. Simulation-based Abstractions for Software Product-Line Model Checking. In *IEEE Int. Conf. on Software Engineering*, pages 672–682, 2012.

[15] L. Damas and R. Milner. Principal Type Schemes for Functional Programming Languages. In *ACM Symp. on Principles of Programming Languages*, pages 207–208, 1982.

[16] B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 595–608, 2011.

[17] B. Delaware, W. R. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 243–252, 2009.

[18] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[19] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by Taming the Preprocessor. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 323–334, 2012.

[20] C. Hankin and D. Métayer. A type-based framework for program analysis. In *Static Analysis Symposium*, LNCS 864, pages 380–394. 1994.

[21] N. Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, LNCS 983, pages 189–206. 1995.

[22] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

[23] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 10 2011.

[24] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 773–792, 2012.

[25] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Large-Scale Variability-Aware Type Checking and Dataflow Analysis. Technical Report MIP-1212, Fakultät für Informatik und Mathematik, Universität Passau, 2012.

[26] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Foundations of Software Engineering*, pages 81–91, 2013.

[27] M. Naik and J. Palsberg. A type system equivalent to a model checker. *ACM Trans. on Programming Languages and Systems*, 30(5):29:1–29:24, 2008.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[29] M. Odersky, M. Sulzmann, and M. Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[30] J. Palsberg. Type-based analysis and applications. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 20–27, 2001.

[31] D. L. Parnas. On the design and development of program families. *IEEE Trans. on Software Engineering*, 2(1):1–9, 1976.

[32] F. Pottier. A versatile constraint-based type inference system. *Nordic J. of Computing*, 7(4):312–347, Dec. 2000.

[33] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Programming Languages and Systems*, 25(1):117–158, 2003.

[34] F. Prost. A Formalization of Static Analyses in System F. In *Automated Deduction CADE-16*, pages 252–266. 1999.

[35] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 49–61, 1995.

[36] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, Jan. 1965.

[37] V. Simonet. An extension of HM(X) with bounded existential and universal data-types. In *ACM SIGPLAN Int. Conf. on Functional Programming*, pages 39–50, 2003.

[38] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. on Programming Languages and Systems*, 29(1):1–38, 2007.

[39] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *ACM SIGPLAN Int. Conf. on Functional Programming*, pages 167–178, 2002.

[40] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC-99-009, University of South Australia, School of Computer and Information Science, 1999.

[41] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *International Conference on Generative Programming and Component Engineering*, pages 11–20, 2012.

[42] J. B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.