Dependent Type Inference with Interpolants

Hiroshi Unno
Tohoku University
uhiro@kb.ecei.tohoku.ac.jp

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

Abstract

We propose a novel type inference algorithm for a dependently-typed functional language. The novel features of our algorithm are: (i) it can iteratively refine dependent types with interpolants until the type inference succeeds or the program is found to be ill-typed, and (ii) in the latter case, it can generate a kind of counter-example as an explanation of why the program is ill-typed. We have implemented a prototype type inference system and tested it for several programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Reliability, Verification

Keywords Dependent Types, Type Inference

1. Introduction

Dependently-typed functional languages such as Cayenne [2], Dependent ML (DML) [29], and Epigram [1] can express and check detailed program specifications statically, including absence of array bounds and pattern match errors. Compared to other program verification techniques such as model checking [3, 10, 17, 23] and abstract interpretation [11], the dependently-typed languages have an advantage that they can deal with advanced programming features such as higher-order functions, polymorphic functions, and recursively defined data structures. Explicit type annotations are, however, usually required. Although there are a few recent proposals for automated type inference, there are still a number of limitations in applying them to practice; for example, for Liquid Types [25], the predicates used in dependent types must be supplied as hints for type inference, and even a typable program is rejected if the given predicates are insufficient.

In this paper, we present a novel technique of automated type inference for a dependently-typed functional language, which is essentially an "implicitly-typed" version of DML [29]. The language supports ML features such as higher-order functions, polymorphic functions, and recursively defined data structures. Note that as in DML, dependent types in our language are used in a more restricted manner than in other dependently-typed languages like Cayenne [2], Epigram [1], Coq [4], etc.: The most important restriction is that types can depend on base values but not on function values. Our type inference algorithm can iteratively refine dependent types by automatically discovering necessary predicates for verification with an interpolating prover [5, 16, 22] until the type inference succeeds or the program is found to be ill-typed; and in the latter case, it can generate a kind of counter-example as a witness of the ill-typedness of the program, which helps users to locate and fix bugs. Intuitively, a counter-example of a program is a sufficient condition for the program to be ill-typed. For example, for the program `if` $x > 0$ `then fail else` 1, $x > 0$ is a counter-example. For a subset of our language for which the type system is complete, the counter-example can also be understood as a sufficient condition for the program to fail at run-time.

In our dependent type system, the typability of a program can be reduced to the satisfiability of a constraint on predicate variables.[1] For example, let us consider the program:

```
let inc x = x + 1
let _ = assert (inc y >= y)
```

Here, the assertion `assert (inc y >= y)` checks whether the argument holds, and gets stuck if the check fails. We can prepare the type template:

$$\texttt{inc} : (\nu_1 : \texttt{int} \to \{\nu_2 : \texttt{int} \mid P(\nu_1, \nu_2)\})$$

It means that the function takes an integer ν_1 as an argument, and returns an integer ν_2 that satisfies the output specification $P(\nu_1, \nu_2)$. (We omitted the input specification for simplicity.) Then, the type inference is reduced to the problem of finding P that satisfies the following constraint:

$$\forall \nu_1, \nu_2.(\nu_2 = \nu_1 + 1 \Rightarrow P(\nu_1, \nu_2)),$$
$$\forall \nu_1, \nu_2.(P(\nu_1, \nu_2) \Rightarrow (\nu_1 = y \Rightarrow \nu_2 \geq y)).$$

The novelty of our work lies in a use of *interpolants* [16, 22] for solving constraints like the one above (see Section 4.1 and Appendix A for formal definitions). Here, an interpolant of two formulas ϕ_1 and ϕ_2 is another formula ϕ such that ϕ_1 implies ϕ, ϕ implies ϕ_2, and the free variables of ϕ must occur in both ϕ_1 and ϕ_2 In the constraint on P above, $P(\nu_1, \nu_2)$ is in fact an interpolant. Thus, we can obtain, for example, $P(\nu_1, \nu_2) \equiv \nu_2 \geq \nu_1$ as a solution, by using an interpolating prover.[2] Craig's interpolation lemma [12] states that such an interpolant always exists in the first-order predicate logic.

An advantage of the use of interpolants is that we can naturally combine information obtained from both function definitions and functions' call sites to infer general specifications. In the constraint above, $\nu_2 = \nu_1 + 1$ comes from the definition, while $\nu_1 = y \Rightarrow \nu_2 \geq y$ comes from the call site. Since the output spec-

[1] Here, the satisfiability means the existence of substitutions of predicates for the predicate variables such that the substituted constraint is valid.

[2] In general, there may be more than one interpolant of given two formulas. In the example, $\nu_2 = \nu_1 + 1$ is also an interpolant.

ification P of `inc` is an interpolant of them, P is determined by taking both sources of information into account. An interpolating prover returns a general solution such as $P(\nu_1, \nu_2) \equiv \nu_2 \geq \nu_1$ in a sense that it does not contain the variable y, which is specific to the particular call site. The advantage of interpolants discussed above helps us to obtain invariants of recursive functions, which are essential for the success of type inference. In contrast, in size inference [8, 19], a function's output specification is usually determined by taking only information from the definition, and in the on-demand dependent type refinement [26], a function's output specification is determined by taking only a part of information from the call sites.

The overall structure of our dependent type inference algorithm is shown in Figure 1. Given a source program, we generate a constraint on predicate variables that is satisfiable if and only if the program is well-typed (see Section 3). The constraint solving algorithm first expands the possibly recursive original constraint to obtain a non-recursive one whose satisfiability is a necessary condition for that of the original (see Section 4.1). Then, the algorithm uses an interpolating prover to find a solution of the expanded constraint, namely substitutions for the predicate variables in the expanded constraint (see Section 4.2). If no solution is found, we conclude that the original constraint is not satisfiable either, and report a counter-example. Otherwise, the algorithm obtains candidate solutions for the original constraint from the solution for the expanded constraint, and checks whether they are genuine (see Section 4.3). If the candidate solutions are judged to be not genuine, the algorithm expands the original constraint further and continues the constraint solving.

In the rest of this paper, before formalizing the type inference procedure sketched above, we overview the procedure in Section 2. We then sketch how the constraints are generated from source programs in Section 3. We discuss the phase for solving constraints on predicate variables using interpolants in detail in Section 4. Section 5 reports on a prototype implementation of our algorithm and experiments. Related work is presented in Section 6. We conclude the paper with some remarks about future work in Section 7.

2. Overview

We overview our algorithm with the following program:

```
let rec sum x = if x <= 0 then 0 else x + sum (x - 1)
let _ = assert (sum y >= y)
```

Here, the assertion `assert` checks whether the argument holds, and gets stuck if the check fails. Note that the assertion checking always succeeds for any run-time environment that assigns an integer value to the free variable `y`.

Constraint Generation We prepare the type template:

$$\texttt{sum} : (\nu_1 : \texttt{int} \to \{\nu_2 : \texttt{int} \mid P(\nu_1, \nu_2)\})$$

Here, $P(\nu_1, \nu_2)$ represents the output specification of `sum`. (We omitted the input specification for simplicity.) We then generate the following constraint on P, by using an algorithm similar to the one proposed in [21]:

$$\begin{aligned} C_1 &:= \forall \nu_1, \nu_2, \nu_1', \nu_2'.(\phi_1 \vee (P(\nu_1', \nu_2') \wedge \phi_2) \Rightarrow P(\nu_1, \nu_2)), \\ C_2 &:= \forall \nu_1, \nu_2, y.(P(\nu_1, \nu_2) \Rightarrow \phi_3). \end{aligned}$$

Here, ϕ_1, ϕ_2, and ϕ_3 are given by:

$$\begin{aligned} \phi_1 &:= \nu_1 \leq 0 \wedge \nu_2 = 0, \\ \phi_2 &:= \nu_1 > 0 \wedge \nu_1' = \nu_1 - 1 \wedge \nu_2 = \nu_1 + \nu_2', \\ \phi_3 &:= y = \nu_1 \Rightarrow \nu_2 \geq y. \end{aligned}$$

The constraint C_1 is generated from the definition of `sum`, and C_2 from the assertion, the call-site of `sum`. The sub-formulas ϕ_1 and $P(\nu_1', \nu_2') \wedge \phi_2$ in C_1 represent the output specifications of the then- and else- branches respectively. Note that unlike in the case of the non-recursive function `inc` in Section 1, P is no longer a mere interpolant between two formulas.

Terminology and Notation Throughout the paper, we use the term "constraint" to mean a first-order logical formula containing predicate variables. A substitution θ of predicates for the predicate variables is a *solution* of C if θC is a tautology. A constraint C is *satisfiable* if it has a solution. We often omit universal quantifiers on first-order variables; for example, we write just $P(\nu_1, \nu_2) \Rightarrow \phi_3$ for the constraint C_2 above.

We reduce the problem of solving the above constraint on P to that of computing an interpolant as follows.

Constraint Expansion We replace P in the left-hand side of the constraint C_1 with P_1, and P in the right-hand side of C_1 and P in the left-hand side of C_2 with P_0, getting the following "non-recursive" constraint:

$$\begin{aligned} C_3 &:= \phi_1 \vee (P_1(\nu_1', \nu_2') \wedge \phi_2) \Rightarrow P_0(\nu_1, \nu_2), \\ C_4 &:= P_0(\nu_1, \nu_2) \Rightarrow \phi_3. \end{aligned}$$

The constraint $C_3 \wedge C_4$ intuitively represents that of the program obtained by expanding the recursive definition of `sum` in the program of `sum` once (namely, the recursive call of `sum` is ignored). Obviously, the satisfiability of $C_3 \wedge C_4$ is a necessary condition for that of $C_1 \wedge C_2$.

Solving Expanded Constraint Now, we can obtain a solution of the constraint $C_3 \wedge C_4$ using interpolants. We first obtain P_0 as follows. Because P_1 does not occur in the right-hand sides of C_3 and C_4, we can replace $P_1(\nu_1', \nu_2')$ in C_3 with the inconsistency $\perp$ without affecting the satisfiability as follows:

$$\phi_1 \vee (\perp \wedge \phi_2) \Rightarrow P_0(\nu_1, \nu_2).$$

Thus, $P_0(\nu_1, \nu_2)$ is obtained as an interpolant of $\phi_1 \vee (\perp \wedge \phi_2)$ and ϕ_3.

Given $P_0(\nu_1, \nu_2) \equiv \psi_0$, we obtain P_1 as follows. We can replace $P_0(\nu_1, \nu_2)$ in C_3 and C_4 with ψ_0 without affecting the satisfiability as follows:

$$\begin{aligned} C_3' &:= \phi_1 \vee (P_1(\nu_1', \nu_2') \wedge \phi_2) \Rightarrow \psi_0, \\ C_4' &:= \psi_0 \Rightarrow \phi_3. \end{aligned}$$

The constraint $C_3' \wedge C_4'$ can be simplified to:

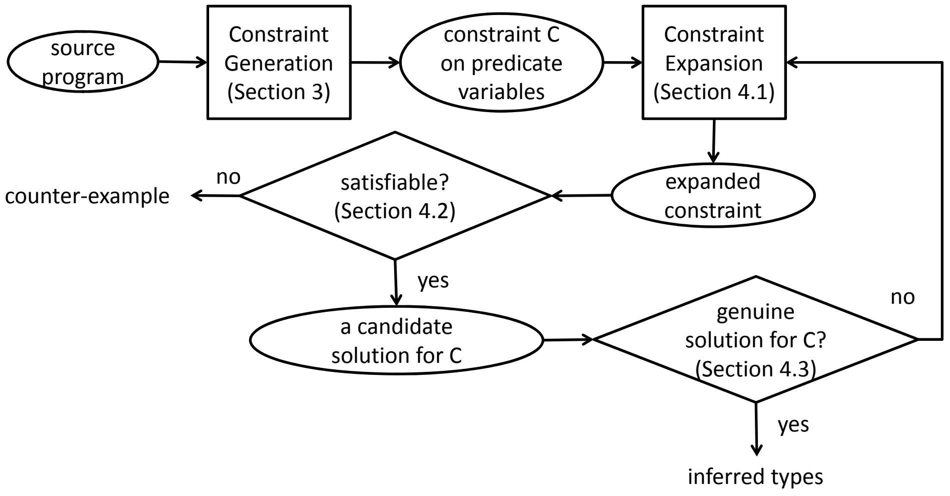
$$P_1(\nu_1', \nu_2') \Rightarrow (\phi_2 \Rightarrow \psi_0)$$

by using the fact that ψ_0 is an interpolant of $\phi_1 \vee (\perp \wedge \phi_2) (\equiv \phi_1)$ and ϕ_3. Note that $P_1(\nu_1', \nu_2')$ can then be obtained as an interpolant of $\perp$ and $\phi_2 \Rightarrow \psi_0$.

Checking Genuineness of Candidate Solutions Suppose that $P_0(\nu_1, \nu_2) \equiv \psi_0$, $P_1(\nu_1', \nu_2') \equiv \psi_1$ is a solution of $C_3 \wedge C_4$, i.e., the following formulas ϕ_A and ϕ_B hold:

$$\phi_A := \phi_1 \vee (\psi_1 \wedge \phi_2) \Rightarrow \psi_0, \qquad \phi_B := \psi_0 \Rightarrow \phi_3.$$

If the condition $\psi_0 \Rightarrow \psi_1$ holds, then $P(\nu_1, \nu_2) \equiv \psi_0$ is a solution of the original constraint $C_1 \wedge C_2$, and hence the type inference succeeds: $P(\nu_1, \nu_2) \equiv \psi_0$ satisfies C_1 because $\phi_1 \vee (\psi_0 \wedge \phi_2)$ implies $\phi_1 \vee (\psi_1 \wedge \phi_2)$ and ϕ_A holds, and satisfies C_2 because ϕ_B holds. Similarly, if another condition $\top \Rightarrow \psi_0$ holds, then $P(\nu_1, \nu_2) \equiv \top$ is a solution of the original constraint $C_1 \wedge C_2$. Thus, we call $P(\nu_1, \nu_2) \equiv \psi_0$ and $P(\nu_1, \nu_2) \equiv \top$ candidate solutions of $C_1 \wedge C_2$, and check their genuineness by using the conditions $\psi_0 \Rightarrow \psi_1$ and $\top \Rightarrow \psi_0$ respectively.



source program
Constraint Generation (Section 3)
constraint C on predicate variables
Constraint Expansion (Section 4.1)
expanded constraint
satisfiable? (Section 4.2)
no
counter-example
yes
a candidate solution for C
genuine solution for C? (Section 4.3)
no
yes
inferred types

Figure 1. Overall Structure of Type Inference Algorithm based on Interpolants

Iterative Dependent Type Refinement If neither $\psi_0 \Rightarrow \psi_1$ nor $\top \Rightarrow \psi_0$ holds, then we further expand the "recursive" constraint $C_1 \wedge C_2$ to obtain the following one:

$$\begin{array}{l}(\phi_1 \vee (P_1(\nu_1', \nu_2') \wedge \phi_2) \Rightarrow P_0(\nu_1, \nu_2)) \wedge \\ (\phi_1 \vee (P_2(\nu_1', \nu_2') \wedge \phi_2) \Rightarrow P_1(\nu_1, \nu_2)) \wedge \\ (P_0(\nu_1, \nu_2) \Rightarrow \phi_3).\end{array}$$

We again (i) solve the expanded constraint, (ii) compute candidate solutions of the original constraint, by using the solution of the expanded constraint, and (iii) check whether one of the candidate solutions is genuine. In this manner, we can iteratively refine types until the type inference succeeds.

Counter-Example Finding The above procedure is also effective for judging a program to be ill-typed (or, to contain an error), and for finding a counter-example. As mentioned above, the typability is reduced to the existence of an interpolant of certain formulas ϕ_1 and ϕ_2. No interpolant exists (hence the program is untypable) if $\phi_1 \Rightarrow \phi_2$ does not hold. In that case, the negation of $\phi_1 \Rightarrow \phi_2$ gives a condition for the program to fail.

To see how the counter-example finding works, let us replace the condition $x \leq 0$ in `sum` with $x \leq 1$. We get the following constraint instead of C_3:

$$\phi_1' \vee (P_1(\nu_1', \nu_2') \wedge \phi_2') \Rightarrow P(\nu_1, \nu_2).$$

Here, ϕ_1' and ϕ_2' are given by:

$$\begin{array}{lcl}\phi_1' & := & \nu_1 \leq 1 \wedge \nu_2 = 0, \\ \phi_2' & := & \nu_1 > 1 \wedge \nu_1' = \nu_1 - 1 \wedge \nu_2 = \nu_1 + \nu_2'.\end{array}$$

Then, an interpolant of $\phi_1' \vee (\bot \wedge \phi_2')(\equiv \phi_1')$ and $\phi_3(\equiv y = \nu_1 \Rightarrow \nu_2 \geq y)$ does not exist, as $\phi_1' \Rightarrow \phi_3$ is invalid. Thus, the refutation of $\phi_1' \Rightarrow \phi_3$ yields a counter-example: $\phi_1' \Rightarrow \phi_3$ does not hold, for example, for $y = 1$, $\nu_1 = 1$, and $\nu_2 = 0$. In fact, an evaluation of the program with the counter-example $y = 1$ indeed causes a failure: as `sum` returns 0, the assertion is violated.

3. Target Language and Constraint Generation

In this section, we first introduce a simple higher-order functional language with a special primitive `fail` that expresses a failure of a program. We then formalize a dependent type system for the language, which can ensure that `fail` is unreachable. Then, we describe our constraint generation algorithm similar to the one proposed in [21]. The language discussed here is simplified to clarify the essence of the constraint generation; our prototype inference system in Section 5 deals with an extended language with data constructors, pattern-matches, tuples, and the let-polymorphism. The constraint generation for the extended language is formalized in the full version of this paper [27].

3.1 Syntax

The syntax of expressions is defined as follows:

$$\begin{array}{lcll}e & ::= & & \textit{Expressions:} \\ & & x & \text{variable} \\ & \mid & c & \text{constant} \\ & \mid & \lambda x.e & \text{abstraction} \\ & \mid & e_1\ e_2 & \text{application} \\ & \mid & \texttt{fix}\ x.e & \text{fixed-point} \\ & \mid & \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \text{if-then-else} \\ & \mid & \texttt{fail} & \text{failure}\end{array}$$

Here, x and c are meta-variables ranging over variables and constants respectively. We write $\mathrm{FV}(e)$ to denote the set of free variables in e. Constants may include integer arithmetic operations.

The operational semantics of the language is call-by-value. An evaluation of `if` x `then` e_1 `else` e_2 proceeds to the then-branch e_1 if x has a non-zero value, and to the else-branch e_2 otherwise. An evaluation of `fail` always gets stuck. We can use `fail` to model array accesses and assertions.

The syntax of types is defined as follows:

$$\begin{array}{lcll}\psi & ::= & P(\widetilde{x}) \mid \phi & \textit{Specifications} \\ T & ::= & & \textit{Dependent Types:} \\ & & \{\nu : \texttt{int} \mid \psi\} & \text{integer refinements} \\ & \mid & \nu : T_1 \rightarrow T_2 & \text{dependent function types} \\ \Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, \phi & \textit{Type Environments}\end{array}$$

Here, P and ϕ are meta-variables ranging over predicate variables and the formulas of some first-order theory respectively. In this paper, we consider the quantifier-free theory of linear arithmetic and equalities with uninterpreted function symbols unless otherwise stated. We also use a meta-variable ν, which ranges over the variables not appearing in expressions. We write $\mathrm{FV}(\phi)$ to denote the set of free variables in ϕ.

We write T for dependent types. Our type system supports integer refinement types and dependent function types. We can use the integer refinement types to express sub-types of the ordinary integer type `int`. For example, $\{\nu : \mathtt{int} \mid \nu \geq 0\}$ denotes the type of non-negative integers. We can use the dependent function types to make the type of the return value of a function depend on its arguments. For example, $\nu_1 : \mathtt{int} \to \nu_2 : \mathtt{int} \to \{\nu_3 : \mathtt{int} \mid \nu_3 = \nu_1 + \nu_2\}$ denotes the type of functions whose return value (denoted by ν_3) is the sum of the two arguments (denoted by ν_1 and ν_2). A type environment Γ is a sequence of type bindings $x : T$, which may include guard formulas ϕ. For checking if-expressions, we use the guard formulas to express information about the value of the conditional we know in the then- and else- branches.

3.2 Type Judgment

$$\frac{x : \{\nu : \mathtt{int} \mid \phi\} \in \Gamma}{\Gamma \vdash x : \{\nu : \mathtt{int} \mid \nu = x\}}\ (\text{T-VAR-INT}) \qquad \frac{}{\Gamma \vdash c : \mathcal{TS}(c)}\ (\text{T-CON})$$

$$\frac{x : (y : T \to T') \in \Gamma}{\Gamma \vdash x : (y : T \to T')}\ (\text{T-VAR-FUN}) \qquad \frac{\Gamma, x : T \vdash e : T \qquad x \notin \mathrm{FV}(T)}{\Gamma \vdash \mathtt{fix}\ x.e : T}\ (\text{T-FIX})$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash \lambda x.e : (x : T \to T')}\ (\text{T-ABS}) \qquad \frac{\Gamma, x \neq 0 \vdash e_1 : T \qquad \Gamma, x = 0 \vdash e_2 : T}{\Gamma \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T}\ (\text{T-IF})$$

$$\frac{\Gamma \vdash e_1 : (x : T' \to T) \qquad \Gamma \vdash e_2 : T' \qquad x \notin \mathrm{FV}(T)}{\Gamma \vdash e_1\ e_2 : T}\ (\text{T-APP}) \qquad \frac{\mathrm{Valid}(\llbracket\Gamma\rrbracket \Rightarrow \bot)}{\Gamma \vdash \mathtt{fail} : T}\ (\text{T-FAIL})$$

$$\frac{\Gamma \vdash e : T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash e : T}\ (\text{T-SUB})$$

Figure 2. Typing Rules

A typing judgment is of the form $\Gamma \vdash e : T$. It reads that the expression e has the type T under the type environment Γ. The typing rules are defined in Figure 2. The function $\mathcal{TS}(c)$ returns the dependent type of c. For example, we have $\mathcal{TS}(n) = \{\nu : \mathtt{int} \mid \nu = n\}$ for an integer n, and $\mathcal{TS}(+) = \nu_1 : \mathtt{int} \to \nu_2 : \mathtt{int} \to \{\nu_3 : \mathtt{int} \mid \nu_3 = \nu_1 + \nu_2\}$.

The sub-typing relation $\Gamma \vdash T_1 <: T_2$ is defined as follows:

$$\frac{\mathrm{Valid}(\llbracket\Gamma\rrbracket \wedge \phi_1 \Rightarrow \phi_2)}{\Gamma \vdash \{\nu : \mathtt{int} \mid \phi_1\} <: \{\nu : \mathtt{int} \mid \phi_2\}}\ (\text{S-INT})$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma, \nu : T_2 \vdash T_1' <: T_2'}{\Gamma \vdash \nu : T_1 \to T_1' <: \nu : T_2 \to T_2'}\ (\text{S-FUN})$$

Here, $\llbracket\Gamma\rrbracket$ is defined by:

$$\begin{aligned}
\llbracket\Gamma, x : \{\nu : \mathtt{int} \mid \phi\}\rrbracket &= \llbracket\Gamma\rrbracket \wedge [x/\nu]\phi \\
\llbracket\Gamma, x : (\nu : T_1 \to T_2)\rrbracket &= \llbracket\Gamma\rrbracket \\
\llbracket\Gamma, \phi\rrbracket &= \llbracket\Gamma\rrbracket \wedge \phi
\end{aligned}$$

The predicate $\mathrm{Valid}(\phi)$ holds if and only if ϕ is valid.

Example 3.1. Let us consider the judgment $\Gamma \vdash \mathtt{inc}\ 1 : T_2$, where $\Gamma = \mathtt{inc} : T_{\mathtt{inc}}$, $T_{\mathtt{inc}} = x : \mathtt{int} \to \{\nu : \mathtt{int} \mid \nu = x + 1\}$, and $T_2 = \{\nu : \mathtt{int} \mid \nu = 2\}$. An example derivation of the judgment is shown in Figure 3, where $T_1 = \{\nu : \mathtt{int} \mid \nu = 1\}$.

Remark 1. Unlike in the dependently-typed languages λ_H [13, 21] and λ_L [25], we separate the language for computation (i.e. expressions) and specification (i.e. formulas), to simplify constraint solving. Thus, the rules T-APP and T-IF do not substitute expressions for variables in types or type environments to make types depend on expressions. We only allow types to depend on variables via T-VAR-INT or T-SUB. By using T-VAR-INT and T-ABS, we can derive, for example, $\vdash \lambda x.x : (x : \mathtt{int} \to \{\nu : \mathtt{int} \mid \nu = x\})$. Since we do not support refinements of function types (e.g. $\{f : \mathtt{int} \to \mathtt{int} \mid \forall x.f\ x \geq x\}$), the type of the return value of a higher-order function that takes a function argument (say, f) cannot depend on the value of f in our type system. The same restriction is applied in Dependent ML [28], λ_H, and λ_L. However, we believe that we can relax the restriction by extending our system with bounded polymorphism on predicate variables or intersection types.

3.3 Constraint Generation Algorithm

The constraint generation algorithm is shown in Figure 4. The function Gen takes Γ, e, and T as inputs, and returns a constraint C on predicate variables, such that θ is a solution of $\exists\widetilde{P}.C$ if and only if $\theta\Gamma \vdash e : \theta T$ holds, where $\widetilde{P}$ is the set of the predicate variables occurring in C but not in Γ and T. The function $\mathrm{Gen}_{<:}$ takes Γ, T_1, and T_2 as inputs, and returns a constraint C on predicate variables, such that θ is a solution of C if and only if $\theta\Gamma \vdash \theta T_1 <: \theta T_2$ holds.

In the algorithm, we assume $\mathrm{TypeOf}(\mathrm{e})$ returns the simple type of e, which can be inferred with the Hindley-Milner type inference algorithm. The following auxiliary function $\mathrm{Lift}(\widetilde{x}; \tau)$ lifts a simple type τ to a dependent type by introducing fresh predicate variables:

$$\begin{aligned}
\mathrm{Lift}(\widetilde{x}; \tau_1 \to \tau_2) &= \nu : \mathrm{Lift}(\widetilde{x}; \tau_1) \to \mathrm{Lift}(\widetilde{x}, \nu; \tau_2) \quad (\nu : \text{fresh}) \\
\mathrm{Lift}(\widetilde{x}; \mathtt{int}) &= \{\nu : \mathtt{int} \mid P(\widetilde{x}, \nu)\} \quad (\nu, P : \text{fresh})
\end{aligned}$$

Example 3.2. Let us consider the program of `inc` in Section 1. The program can be encoded as follows in our language:

$$e_{\mathtt{inc}} = (\lambda\mathtt{inc}.e_{\mathtt{as}})\ (\lambda x. + x\ 1)$$

Here, $e_{\mathtt{as}} = (\lambda b.\mathtt{if}\ b\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{fail})\ (\geq\ (\mathtt{inc}\ y)\ y)$. The Hindley-Milner algorithm infers the type `int` for $e_{\mathtt{inc}}$, and then the constraint generation for $e_{\mathtt{inc}}$ proceeds as follows:

$$\begin{aligned}
\mathrm{Gen}\,(\vdash e_{\mathtt{inc}} : \mathtt{int}) \quad = \quad & \mathrm{Gen}\,(\vdash \lambda\mathtt{inc}.e_{\mathtt{as}} : (\mathtt{inc} : T \to \mathtt{int})) \wedge \\
& \mathrm{Gen}\,(\vdash \lambda x. + x\ 1 : T)
\end{aligned}$$

Here, $T = \nu_1 : T_1 \to T_2$, $T_1 = \{\nu : \mathtt{int} \mid P(\nu)\}$, and $T_2 = \{\nu_2 : \mathtt{int} \mid Q(\nu_1, \nu_2)\}$ for unknown input $P(\nu)$ and output $Q(\nu_1, \nu_2)$ specifications of `inc`. The part $\mathrm{Gen}\,(\vdash \lambda x. + x\ 1 : T)$ is evaluated as follows:

$$\begin{aligned}
& \mathrm{Gen}\,(\vdash \lambda x. + x\ 1 : T) \\
= \quad & \mathrm{Gen}\,(x : T_1 \vdash + x\ 1 : [x/\nu_1]T_2) \\
= \quad & \mathrm{Gen}\left(x : T_1 \vdash + : \begin{array}{r}\{\nu : \mathtt{int} \mid R(\mathtt{inc}, \nu)\} \to \\ \{\nu : \mathtt{int} \mid S(\mathtt{inc}, \nu)\} \to [x/\nu_1]T_2\end{array}\right) \wedge \\
& \mathrm{Gen}\,(x : T_1 \vdash x : \{\nu : \mathtt{int} \mid R(\mathtt{inc}, \nu)\}) \wedge \\
& \mathrm{Gen}\,(x : T_1 \vdash 1 : \{\nu : \mathtt{int} \mid S(\mathtt{inc}, \nu)\}) \\
= \quad & (P(x) \wedge R(\mathtt{inc}, \nu_1) \wedge S(\mathtt{inc}, \nu_2) \wedge \\
& \nu_3 = \nu_1 + \nu_2 \Rightarrow Q(x, \nu_3)) \wedge \\
& (P(x) \wedge \nu = x \Rightarrow R(\mathtt{inc}, \nu)) \wedge \\
& (P(x) \wedge \nu = 1 \Rightarrow S(\mathtt{inc}, \nu)) \\
\equiv \quad & P(x) \wedge \nu = x + 1 \Rightarrow Q(x, \nu)
\end{aligned}$$

Here, R and S are fresh predicate variables, which represent the specifications of the sub-expressions x and 1 respectively. Similarly, the part $\mathrm{Gen}\,(\vdash \lambda\mathtt{inc}.e_{\mathtt{as}} : (\mathtt{inc} : T \to \mathtt{int}))$ is evaluated to the following constraint:

$$(\nu = y \Rightarrow P(\nu)) \wedge (Q(\nu_1, \nu_2) \Rightarrow (\nu_1 = y \Rightarrow \nu_2 \geq y)).$$

$$\dfrac{\dfrac{\dfrac{\texttt{inc} : T_{\texttt{inc}} \in \Gamma}{\Gamma \vdash \texttt{inc} : T_{\texttt{inc}}}\ \textsc{T-Var-Fun} \quad \dfrac{\dfrac{\mathrm{Valid}(\llbracket\Gamma\rrbracket \wedge \nu = 1 \Rightarrow \top)}{\Gamma \vdash T_1 <: \texttt{int}}\ \textsc{S-Int} \quad \dfrac{\mathrm{Valid}(\llbracket\Gamma, x : T_1\rrbracket \wedge \nu = x+1 \Rightarrow \nu = 2)}{\Gamma, x:T_1 \vdash \{\nu:\texttt{int} \mid \nu = x+1\} <: T_2}\ \textsc{S-Int}}{\Gamma \vdash T_{\texttt{inc}} <: (x:T_1 \to T_2)}}{\Gamma \vdash \texttt{inc} : (x : T_1 \to T_2)}\ \textsc{T-Sub} \quad \dfrac{}{\Gamma \vdash 1 : T_1}\ \textsc{T-Con}}{\Gamma \vdash \texttt{inc}\ 1 : T_2}\ \textsc{T-App}$$

Figure 3. An Example Derivation of the Typing Judgment in Example 3.1

$$\begin{aligned}
\mathrm{Gen}\,(\Gamma \vdash x : T) &= \begin{cases} \mathrm{Gen}_{<:}\,(\Gamma \vdash \{\nu : \texttt{int} \mid \nu = x\} <: T) & (\text{if } x : \{\nu : \texttt{int} \mid \phi\} \in \Gamma) \\ \mathrm{Gen}_{<:}\,(\Gamma \vdash (y : T_1 \to T_2) <: T) & (\text{if } x : (y : T_1 \to T_2) \in \Gamma) \end{cases} \\
\mathrm{Gen}\,(\Gamma \vdash c : T) &= \mathrm{Gen}_{<:}\,(\Gamma \vdash \mathcal{TS}(c) <: T) \\
\mathrm{Gen}\,(\Gamma \vdash \lambda x.e : (x : T_1 \to T_2)) &= \mathrm{Gen}\,(\Gamma, x : T_1 \vdash e : T_2) \\
\mathrm{Gen}\,(\Gamma \vdash e_1\ e_2 : T) &= \text{let } T' = \mathrm{Lift}(\mathrm{dom}(\Gamma); \mathrm{TypeOf(e_2)}) \text{ in let } x \text{ be a fresh variable in} \\
&\ \mathrm{Gen}\left(\Gamma \vdash e_1 : (x : T' \to T)\right) \wedge \mathrm{Gen}\left(\Gamma \vdash e_2 : T'\right) \\
\mathrm{Gen}\,(\Gamma \vdash \texttt{fix}\ x.e : T) &= \mathrm{Gen}\,(\Gamma, x : T \vdash e : T) \\
\mathrm{Gen}\,(\Gamma \vdash \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T) &= \mathrm{Gen}\,(\Gamma, x \neq 0 \vdash e_1 : T) \wedge \mathrm{Gen}\,(\Gamma, x = 0 \vdash e_2 : T) \\
\mathrm{Gen}\,(\Gamma \vdash \texttt{fail} : T) &= \llbracket\Gamma\rrbracket \Rightarrow \bot \\
\mathrm{Gen}_{<:}\,(\Gamma \vdash \{\nu : \texttt{int} \mid \psi_1\} <: \{\nu : \texttt{int} \mid \psi_2\}) &= \llbracket\Gamma\rrbracket \wedge \psi_1 \Rightarrow \psi_2 \\
\mathrm{Gen}_{<:}\left(\Gamma \vdash \nu : T_1 \to T_1' <: \nu : T_2 \to T_2'\right) &= \mathrm{Gen}_{<:}\,(\Gamma \vdash T_2 <: T_1) \wedge \mathrm{Gen}_{<:}\left(\Gamma, \nu : T_2 \vdash T_1' <: T_2'\right)
\end{aligned}$$

Figure 4. Constraint Generation Algorithm

4. Constraint Solving

We now describe the key part of our dependent type inference algorithm: an algorithm for solving constraints on predicate variables. To clarify the essence of the algorithm, we present an algorithm for solving constraints on one predicate variable in Sections 4.1–4.4. We discuss how to extend it to deal with constraints on multiple predicate variables in Appendix A. Appendix B discusses several optimizations of the algorithm.

Figure 5 presents the constraint solving algorithm SOLVE. We explain Expand in Section 4.1 and SolveExpanded in Section 4.2. Section 4.3 explains the lines 6–9, where SOLVE checks the genuineness of candidate solutions of the original constraint. The correctness and the termination of SOLVE are discussed in Section 4.4.

4.1 Constraint Expansion

We consider constraints of the following form in Sections 4.1-4.4:

$$(F(P) \Rightarrow \bot) \wedge (\forall \widetilde{x}. G(P)(\widetilde{x}) \Rightarrow P(\widetilde{x}))$$

Here, $\widetilde{x}$ is a sequence of variables, and $F(P)$ and $G(P)(\widetilde{x})$ are of the following form:

$$\exists \widetilde{y}. \phi_0 \vee (P(\widetilde{x_1}) \wedge \phi_1) \vee \cdots \vee (P(\widetilde{x_n}) \wedge \phi_n)$$

Here, $\exists \widetilde{y}$ binds all free variables except for $\widetilde{x}$.

Example 4.1. The following constraint $C_{\texttt{sum}}$ is obtained from the sample program for `sum` in Section 2:

$$\begin{aligned}
C_{\texttt{sum}} &:= (F(P) \Rightarrow \bot) \wedge \\
&\ (\forall \nu_1, \nu_2. G(P)(\nu_1, \nu_2) \Rightarrow P(\nu_1, \nu_2)) \\
F(P) &:= \exists \nu_1, \nu_2, y. P(\nu_1, \nu_2) \wedge \neg\phi_3 \\
G(P)(\nu_1, \nu_2) &:= \exists \nu_1', \nu_2'. \phi_1 \vee (P(\nu_1', \nu_2') \wedge \phi_2)
\end{aligned}$$

Here, ϕ_1, ϕ_2, and ϕ_3 are defined as follows:

$$\begin{aligned}
\phi_1 &:= \nu_1 \leq 0 \wedge \nu_2 = 0, \\
\phi_2 &:= \nu_1 > 0 \wedge \nu_1' = \nu_1 - 1 \wedge \nu_2 = \nu_1 + \nu_2', \\
\phi_3 &:= y = \nu_1 \Rightarrow \nu_2 \geq y.
\end{aligned}$$

We use this constraint as a running example of constraint solving.

We can expand a (possibly recursive) original constraint C to obtain (non-recursive) expanded constraints that are defined as follows:

Definition 4.1. Let C be the following constraint:

$$(F(P) \Rightarrow \bot) \wedge (\forall \widetilde{x}. G(P)(\widetilde{x}) \Rightarrow P(\widetilde{x}))$$

For each $i \geq 0$, we define an *expanded constraint* $\mathrm{Expand}(C, i)$ with the new predicate variables $\{P^{(j)} \mid 0 \leq j \leq i\}$ as follows:

$$\begin{aligned}
\mathrm{Expand}(C, i) &:= (F(P^{(0)}) \Rightarrow \bot) \wedge \\
&\ (\forall \widetilde{x}. G(P^{(1)})(\widetilde{x}) \Rightarrow P^{(0)}(\widetilde{x})) \wedge \cdots \wedge \\
&\ (\forall \widetilde{x}. G(P^{(i)})(\widetilde{x}) \Rightarrow P^{(i-1)}(\widetilde{x}))
\end{aligned}$$

The following lemma follows immediately from the construction of $\mathrm{Expand}(C, i)$ above.

Lemma 4.1. *For any constraint C and $i \geq 0$, $\mathrm{Expand}(C, i)$ has a solution if C has a solution.*

Proof. Let a substitution $\{P \mapsto \lambda \widetilde{x}. \phi\}$ be a solution for C. Then, for any $i \geq 0$, $\{P^{(0)} \mapsto \lambda \widetilde{x}. \phi, \ldots, P^{(i)} \mapsto \lambda \widetilde{x}. \phi\}$ is a solution for $\mathrm{Expand}(C, i)$. □

4.2 Solving Expanded Constraints

The sub-procedure SolveExpanded checks whether an expanded constraint $\mathrm{Expand}(C, i)$ is satisfiable, and returns a solution of $\mathrm{Expand}(C, i)$ if it is the case. The satisfiability of $\mathrm{Expand}(C, i)$

```
procedure SOLVE($C$) :
1 :   for each $i \geq 0$ :
2 :     let $C' = \mathrm{Expand}(C, i)$
3 :     match $\mathrm{SolveExpanded}(C')$ with
4 :       Unsatisfiable $\rightarrow$ abort
5 :     | Satisfiable($\theta'$) $\rightarrow$
6 :         let $\{P^{(j)} \mapsto \lambda\widetilde{x}.\phi_j \mid j \in \{0, \ldots, i\}\} = \theta'$
7 :         for each $k \in \{0, \ldots, i\}$ :
8 :           if $\phi_0 \wedge \cdots \wedge \phi_{k-1} \Rightarrow \phi_k$ then
9 :             return $\{P \mapsto \lambda\widetilde{x}.\phi_0 \wedge \cdots \wedge \phi_{k-1}\}$
```

Figure 5. Constraint Solving Algorithm based on Interpolants (Single Predicate Variable Version)

can be reduced to the validity of the formula $F(G^i(\lambda\widetilde{x}.\bot)) \Rightarrow \bot$, where $G^i(p)$ is defined as follows:

$$G^0(p) = p, \qquad G^i(p) = G^{i-1}(G(p)).$$

To see why the reduction is correct, suppose that $\mathrm{Expand}(C, i)$ is satisfiable. Namely, we have a substitution θ for the predicate variables $P^{(0)}, \ldots, P^{(i)}$ in $\mathrm{Expand}(C, i)$ such that $F(\theta P^{(0)}) \Rightarrow \bot$ and $G(\theta P^{(j)})(\widetilde{x}) \Rightarrow \theta P^{(j-1)}(\widetilde{x})$ hold for all $j \in \{1, \ldots, i\}$. Then, by the monotonicity of G, we get:

$$\begin{array}{rcl} \bot & \Leftarrow & F(\theta P^{(0)}) \\ & \Leftarrow & F(G(\theta P^{(1)})) \\ & \cdots & \\ & \Leftarrow & F(G^i(\theta P^{(i)})) \\ & \Leftarrow & F(G^i(\lambda\widetilde{x}.\bot)) \end{array}$$

Conversely, if $F(G^i(\lambda\widetilde{x}.\bot)) \Rightarrow \bot$ holds, then the following substitution satisfies $\mathrm{Expand}(C, i)$:

$$\{P^{(j)} \mapsto G^{(i-j)}(\lambda\widetilde{x}.\bot)\}_{j=0}^{i}$$

The formula $F(G^i(\lambda\widetilde{x}.\bot))$ can be always transformed to a formula of the form $\exists\widetilde{z}.\phi$ (recall the form of $F(P)$ and $G(P)(\widetilde{x})$ discussed in Section 4.1). We can check the validity of $(\exists\widetilde{z}.\phi) \Rightarrow \bot$ by using existing theorem provers including interpolating provers.

We now present an algorithm for finding a solution of a satisfiable expanded constraint $\mathrm{Expand}(C, i)$. As mentioned earlier, we reduce the problem of finding a solution of $\mathrm{Expand}(C, i)$ to that of computing interpolants.

Definition 4.2 (interpolants [12]). Given a pair of predicates $(\lambda\widetilde{x}.\phi_1, \lambda\widetilde{x}.\phi_2)$ such that ϕ_1 implies ϕ_2 and $\mathrm{FV}(\phi_1) \cap \mathrm{FV}(\phi_2) \subseteq \{\widetilde{x}\}$, we call $\lambda\widetilde{x}.\phi$ an *interpolant* of the pair if

- ϕ_1 implies ϕ,
- ϕ implies ϕ_2, and
- $\mathrm{FV}(\phi) \subseteq \mathrm{FV}(\phi_1) \cap \mathrm{FV}(\phi_2)$.

Here, we say ϕ_1 implies ϕ_2 when $\forall\widetilde{y}.\phi_1 \Rightarrow \phi_2$, where $\widetilde{y} = \mathrm{FV}(\phi_1) \cup \mathrm{FV}(\phi_2)$.

An interpolant of ϕ_1 and ϕ_2 always exists if ϕ_1 implies ϕ_2, and can be computed by using an interpolating prover in various first-order theories including the quantifier-free theory of linear arithmetic and equalities with uninterpreted function symbols [20]. For example, $\lambda x.\lambda y.x = y$ is an interpolant of the pair $(\lambda x.\lambda y.x = z \wedge y = z, \lambda x.\lambda y.x = 0 \Rightarrow y = 0)$. In general, there may be more than one interpolant of given two formulas. In the example, $\lambda x.\lambda y.x \geq y$ is also an interpolant.

We obtain a substitution for each predicate variable $P^{(0)}, \ldots, P^{(i)}$ in $\mathrm{Expand}(C, i)$ in this order as follows. As in the satisfiability reduction discussed at the beginning of this section, we can reduce the satisfiability of $\mathrm{Expand}(C, i)$ to that of the following constraint C_0 that contains only the predicate variable $P^{(0)}$:

$$(\forall\widetilde{x}.G^i(\lambda\widetilde{x}.\bot)(\widetilde{x}) \Rightarrow P^{(0)}(\widetilde{x})) \wedge (F(P^{(0)}) \Rightarrow \bot)$$

We reduce the problem of finding a solution of C_0 to that of computing interpolants. C_0 is of the following form (recall the form of $F(P)$ and $G(P)(\widetilde{x})$ discussed in Section 4.1):

$$\begin{array}{l} (\forall\widetilde{x}.(\exists\widetilde{y}.\phi) \Rightarrow P(\widetilde{x})) \wedge \\ (\forall\widetilde{x}.(\exists\widetilde{z}.\phi_0 \vee (P(\widetilde{x_1}) \wedge \phi_1) \vee \cdots \vee (P(\widetilde{x_n}) \wedge \phi_n)) \Rightarrow \phi') \end{array}$$

We can transform the second line to the following one:

$$\begin{array}{l} (\forall\widetilde{x}, \widetilde{z}.\phi_0 \Rightarrow \phi') \wedge \\ (\forall\widetilde{x}, \widetilde{z}.P(\widetilde{x_1}) \Rightarrow (\phi_1 \Rightarrow \phi')) \wedge \cdots \wedge \\ (\forall\widetilde{x}, \widetilde{z}.P(\widetilde{x_n}) \Rightarrow (\phi_n \Rightarrow \phi')) \end{array}$$

Therefore, we can obtain $P(\widetilde{x}) \equiv \phi'_1 \wedge \cdots \wedge \phi'_n$ by computing interpolants $\lambda\widetilde{x}.\phi'_k$ of the pairs $(\lambda\widetilde{x}.\phi, \lambda\widetilde{x_k}.\phi_k \Rightarrow \phi')$ for all $k \in \{1, \ldots, n\}$ with an interpolating prover.

Similarly, for each $j \in \{1, \ldots, i\}$, given solutions $P^{(0)}(\widetilde{x}) \equiv \phi^{(0)}, \ldots, P^{(j-1)}(\widetilde{x}) \equiv \phi^{(j-1)}$ to $C_0, \ldots, C_{j-1}$ respectively, the satisfiability of $\mathrm{Expand}(C, i)$ can be reduced to that of the following constraint C_j that contains only the predicate variable $P^{(j)}$:

$$(\forall\widetilde{x}.G^{i-j}(\lambda\widetilde{x}.\bot)(\widetilde{x}) \Rightarrow P^{(j)}(\widetilde{x})) \wedge (\forall\widetilde{x}.G(P^{(j)})(\widetilde{x}) \Rightarrow \phi^{(j-1)})$$

As in the case of C_0 above, the problem of finding a solution to C_j ($1 \leq j \leq i$) can be reduced to the problem of computing interpolants.

Example 4.2. Let us consider the constraint $C_{\mathtt{sum}}$ in Example 4.1. The expanded constraint $\mathrm{Expand}(C_{\mathtt{sum}}, 1)$ of $C_{\mathtt{sum}}$ on the new predicate variables $P^{(0)}$ and $P^{(1)}$ is as follows:

$$\begin{array}{rcl} \mathrm{Expand}(C_{\mathtt{sum}}, 1) & := & (F(P^{(0)}) \Rightarrow \bot) \wedge \\ & & \forall\nu_1, \nu_2.(G(P^{(1)})(\nu_1, \nu_2) \Rightarrow P^{(0)}(\nu_1, \nu_2)) \end{array}$$

Here, $F(P)$ and $G(P)(\nu_1, \nu_2)$ are defined in Example 4.1. We find a solution of $\mathrm{Expand}(C_{\mathtt{sum}}, 1)$ in this example. $P^{(0)}$ is obtained as a solution of the following constraint:

$$(\forall\nu_1, \nu_2.G(\lambda\nu_1, \nu_2.\bot)(\nu_1, \nu_2) \Rightarrow P^{(0)}(\nu_1, \nu_2)) \wedge (F(P^{(0)}) \Rightarrow \bot)$$

This is equivalent to the following constraint:

$$(\forall\nu_1, \nu_2.\phi_1 \Rightarrow P^{(0)}(\nu_1, \nu_2)) \wedge (\forall\nu_1, \nu_2, y.P^{(0)}(\nu_1, \nu_2) \Rightarrow \phi_3)$$

In this example, we can obtain $P^{(0)}(\nu_1, \nu_2) \equiv \nu_2 \geq \nu_1$ as an interpolant of $(\lambda\nu_1, \nu_2.\phi_1, \lambda\nu_1, \nu_2.\phi_3)$. Then, $P^{(1)}$ is obtained as a solution of the following constraint:

$$\begin{array}{l} (\forall\nu_1, \nu_2.G^0(\lambda\nu_1, \nu_2.\bot)(\nu_1, \nu_2) \Rightarrow P^{(1)}(\nu_1, \nu_2)) \wedge \\ (\forall\nu_1, \nu_2.G(P^{(1)})(\nu_1, \nu_2) \Rightarrow \nu_2 \geq \nu_1) \end{array}$$

This is equivalent to the following constraint:

$$\begin{array}{l} (\forall\nu_1, \nu_2.\bot \Rightarrow P^{(1)}(\nu_1, \nu_2)) \wedge \\ (\forall\nu_1, \nu_2.\phi_1 \Rightarrow \nu_2 \geq \nu_1) \wedge \\ (\forall\nu_1, \nu_2, \nu'_1, \nu'_2.P^{(1)}(\nu'_1, \nu'_2) \Rightarrow (\phi_2 \Rightarrow \nu_2 \geq \nu_1)) \end{array}$$

Thus, we can obtain $P^{(1)}(\nu_1, \nu_2) \equiv \bot$ as an interpolant of $(\lambda\nu_1, \nu_2.\bot, \lambda\nu_1, \nu_2.\phi_2 \Rightarrow \nu_2 \geq \nu_1)$. As a result, we obtain the following solution $\theta_{\mathtt{sum}}$ of $\mathrm{Expand}(C_{\mathtt{sum}}, 1)$:

$$\{P^{(0)} \mapsto \lambda\nu_1,\nu_2.\nu_2 \geq \nu_1, P^{(1)} \mapsto \lambda\nu_1,\nu_2.\bot\}.$$

If an expanded constraint $\mathrm{Expand}(C, i)$ is not satisfiable, C is not satisfiable either, and we can refute $\mathrm{Expand}(C, i)$ to obtain a counter-example for C, namely, valuations of the variables $\widetilde{z}$ that satisfy ϕ, where $\exists\widetilde{z}.\phi$ is a formula equivalent to $F(G^i(\lambda\widetilde{x}.\bot))$.

4.3 Checking Genuineness of Candidate Solutions

Given a solution $\{P^{(j)} \mapsto \lambda\widetilde{x}.\phi_j \mid j \in \{0,\dots,i\}\}$ of an expanded constraint $\mathrm{Expand}(C, i)$, SOLVE obtains the following candidate solutions of C:

$$\{\{P \mapsto \lambda\widetilde{x}.\phi_0 \wedge \cdots \wedge \phi_{k-1}\} \mid k \in \{0,\dots,i\}\}$$

For each $k \in \{0,\dots,i\}$, SOLVE judges whether the candidate solution $\{P \mapsto \lambda\widetilde{x}.\phi_0 \wedge \cdots \wedge \phi_{k-1}\}$ is genuine by checking the following sufficient condition:

$$\phi_0 \wedge \cdots \wedge \phi_{k-1} \Rightarrow \phi_k$$

The correctness of the above condition is established by the following lemma.

Lemma 4.2. *Suppose that an expanded constraint* $\mathrm{Expand}(C, i)$ *has a solution* $\{P^{(j)} \mapsto \lambda\widetilde{x}.\phi_j \mid j \in \{0,\dots,i\}\}$. *If* $\phi_0 \wedge \cdots \wedge \phi_{k-1}$ *implies* ϕ_k *for some* $k \in \{0,\dots,i\}$, *then* $\theta = \{P \mapsto \lambda\widetilde{x}.\phi_0 \wedge \cdots \wedge \phi_{k-1}\}$ *is a solution of* C.

Proof. We have $F(\lambda\widetilde{x}.\phi_0) \Rightarrow \bot$ and $G(\lambda\widetilde{x}.\phi_{j+1})(\widetilde{x}) \Rightarrow \phi_j$ for all $j \in \{0,\dots,i-1\}$. Assume that $\phi_0 \wedge \cdots \wedge \phi_{k-1}$ implies ϕ_k for some $k \in \{0,\dots,i\}$.

- If $k = 0$, we get $\theta P = \lambda\widetilde{x}.\top$, and $\phi_0 \equiv \top$ by the assumption. Thus, we have $F(\theta P) \Rightarrow \bot$ and $G(\theta P)(\widetilde{x}) \Rightarrow \theta P(\widetilde{x})$.
- Otherwise, we get:

$$\begin{array}{lll} \bot & \Leftarrow & F(\lambda\widetilde{x}.\phi_0) \\ & \Leftarrow & F(\lambda\widetilde{x}.\phi_0 \wedge \cdots \wedge \phi_{k-1}) \quad \text{(by monotonicity of } F\text{)} \\ & = & F(\theta P) \end{array}$$

  We can also show that:

$$\begin{array}{lll} \theta P(\widetilde{x}) & = & \phi_0 \wedge \cdots \wedge \phi_{k-1} \\ & \Leftarrow & G(\lambda\widetilde{x}.\phi_1)(\widetilde{x}) \wedge \cdots \wedge G(\lambda\widetilde{x}.\phi_k)(\widetilde{x}) \\ & \Leftarrow & G(\lambda\widetilde{x}.\phi_0 \wedge \ldots \wedge \phi_k)(\widetilde{x}) \quad \text{(by monotonicity of } G\text{)} \\ & \Leftarrow & G(\lambda\widetilde{x}.\phi_0 \wedge \ldots \wedge \phi_{k-1})(\widetilde{x}) \quad \text{(by the assumption)} \\ & = & G(\theta P)(\widetilde{x}) \end{array}$$

□

Example 4.3. Let us consider $\mathrm{Expand}(C_{\mathtt{sum}}, 1)$ and its solution $\theta_{\mathtt{sum}}$ in Example 4.2. We can obtain the following candidate solutions of $C_{\mathtt{sum}}$ from $\theta_{\mathtt{sum}}$:

$$P(\nu_1,\nu_2) \equiv \top, \quad P(\nu_2,\nu_2) \equiv \theta_{\mathtt{sum}} P^{(0)}(\nu_1,\nu_2) \equiv \nu_2 \geq \nu_1$$

However, the former is not genuine because $\top$ does not imply $\theta_{\mathtt{sum}} P^{(0)}(\nu_1,\nu_2) \equiv \nu_2 \geq \nu_1$, and the latter is not genuine because $\nu_2 \geq \nu_1$ does not imply $\theta_{\mathtt{sum}} P^{(1)}(\nu_1,\nu_2) \equiv \bot$. Thus, we expand $C_{\mathtt{sum}}$ further to obtain $\mathrm{Expand}(C_{\mathtt{sum}}, 2)$. Then, we may obtain the following solution $\theta'_{\mathtt{sum}}$ of $\mathrm{Expand}(C_{\mathtt{sum}}, 2)$:

$$\begin{array}{lll} \{P^{(0)} & \mapsto & \lambda\nu_1,\nu_2.\nu_2 \geq \nu_1, \\ P^{(1)} & \mapsto & \lambda\nu_1,\nu_2.\nu_1 \geq 0 \Rightarrow \nu_2 \geq 0, \\ P^{(2)} & \mapsto & \lambda\nu_1,\nu_2.\bot\}. \end{array}$$

We now obtain a genuine solution $\{P \mapsto \theta'_{\mathtt{sum}} P^{(0)}\}$ for $C_{\mathtt{sum}}$ because $\theta'_{\mathtt{sum}} P^{(0)}(\nu_1,\nu_2)$ implies $\theta'_{\mathtt{sum}} P^{(1)}(\nu_1,\nu_2)$. Thus, we inferred the following dependent type of the function `sum`:

$$\mathtt{sum} : (\nu_1 : \mathtt{int} \to \{\nu_2 : \mathtt{int} \mid \nu_2 \geq \nu_1\})$$

4.4 Properties of Constraint Solving Algorithm

Correctness The following theorem, which follows immediately from Lemmas 4.1 and 4.2, establishes the correctness of SOLVE:

Theorem 4.3 (Correctness)**.** *(a) If* $\mathrm{SOLVE}(C)$ *returns* θ, θ *is a solution for* C. *(b) If* $\mathrm{SOLVE}(C)$ *aborts,* C *is not satisfiable.*

Termination We make the following assumptions on the underlying theory of the first-order logic: (i) The validity checking is decidable; (ii) The interpolation problem is decidable. The existence of interpolants for various first-order theories is discussed in [20]. Even though these problems are decidable, the type inference problem of our dependent type system is undecidable unless we assume the strong condition on the underlying theory stated in Theorem 4.5. Therefore, our algorithm SOLVE does not terminate in all cases.

We separate the termination property of SOLVE into two: the termination for satisfiable constraints, and that for unsatisfiable constraints. The former usually depends on not only the choice of the underlying theory but also that of an interpolating prover. In contrast, the latter only depends on the choice of the underlying theory. In fact, if the underlying theory satisfies a certain condition discussed below, we can prove that $\mathrm{SOLVE}(C)$ always aborts in a finite time for any unsatisfiable constraint C. The condition guarantees that an expanded constraint $\mathrm{Expand}(C, i)$ always gets unsatisfiable for some $i \geq 0$. The following theorem formalizes the condition.

Theorem 4.4. *Let* C *be the following constraint:*

$$(F(P) \Rightarrow \bot) \wedge (\forall\widetilde{x}.G(P)(\widetilde{x}) \Rightarrow P(\widetilde{x}))$$

Suppose that the underlying theory has the least upper bounds (with respect to the implication order $\Rightarrow$*) of the following two infinite sequences:*

$$(1)\bot, G(\lambda\widetilde{x}.\bot)(\widetilde{x}), G^2(\lambda\widetilde{x}.\bot)(\widetilde{x}), \dots, G^i(\lambda\widetilde{x}.\bot)(\widetilde{x}), \dots$$
$$(2)F(\lambda\widetilde{x}.\bot), F(G(\lambda\widetilde{x}.\bot)), F(G^2(\lambda\widetilde{x}.\bot)), \dots, F(G^i(\lambda\widetilde{x}.\bot)), \dots$$

We write $\bigsqcup_i G^i(\lambda\widetilde{x}.\bot)(\widetilde{x})$ *and* $\bigsqcup_i F(G^i(\lambda\widetilde{x}.\bot))$ *to denote the least upper bounds of (1) and (2) respectively. If* C *is unsatisfiable, there exists* $i \geq 0$ *such that* $\mathrm{Expand}(C, i)$ *is not satisfiable.*

Proof. We prove the theorem by contraposition. We assume that $\mathrm{Expand}(C, i)$ is satisfiable for any $i \geq 0$, and show that $P(\widetilde{x}) \equiv \bigsqcup_i G^i(\lambda\widetilde{x}.\bot)(\widetilde{x})$ is a solution for C. Recall that $F(P)$ and $G(P)(\widetilde{x})$ are of the following form:

$$\exists\widetilde{y}.\phi_0 \vee (P(\widetilde{x_1}) \wedge \phi_1) \vee \cdots \vee (P(\widetilde{x_n}) \wedge \phi_n).$$

Thus, we have:

$$\begin{array}{lll} F(\lambda\widetilde{x}.\bigsqcup_i G^i(\lambda\widetilde{x}.\bot)(\widetilde{x})) & \equiv & \bigsqcup_i F(G^i(\lambda\widetilde{x}.\bot)), \\ G(\lambda\widetilde{x}.\bigsqcup_i G^i(\lambda\widetilde{x}.\bot)(\widetilde{x})) & \equiv & \bigsqcup_i G^{i+1}(\lambda\widetilde{x}.\bot)(\widetilde{x}) \\ & \equiv & \bigsqcup_i G^i(\lambda\widetilde{x}.\bot)(\widetilde{x}). \end{array}$$

Since $F(G^i(\lambda\widetilde{x}.\bot)) \Rightarrow \bot$ holds for any $i \geq 0$, $\bot$ is an upper bound of the infinite sequence (2). Thus, we get $\bigsqcup_i F(G^i(\lambda\widetilde{x}.\bot)) \Rightarrow \bot$ because $\bigsqcup_i F(G^i(\lambda\widetilde{x}.\bot))$ is the least upper bound of (2). □

```
let rec bs_aux key vec l u =
  if l <= u then
    let m = l + (u-l) / 2 in
    let x = elem vec m in
    if x < key then bs_aux key vec (m+1) u
    else if x > key then bs_aux key vec l (m-1)
    else Some (m)
  else None
let bsearch key vec = bs_aux key vec 0 (size vec - 1)
let _ = bsearch key vec
```

Figure 6. Part of Verified Array Programs

If the underlying theory satisfies a certain stronger condition, for any choice of an interpolating prover, we can prove the termination for both satisfiable and unsatisfiable constraints as follows:

Theorem 4.5. *A sequence of formulas $\phi_1, \ldots, \phi_n$ is said to be a finite descending chain if $\phi_i \not\Leftarrow \phi_j$ holds for all $1 \leq i < j \leq n$. We call a theory is k-bounded if any finite descending chain has the length at most k. If the underlying theory is k-bounded, $\mathrm{SOLVE}(C)$ always returns a solution or aborts for any constraint C.*

Proof. Suppose that $\mathrm{Expand}(C, k)$ has a solution $\{P^{(j)} \mapsto \lambda\widetilde{x}.\phi_j \mid j \in \{0, \ldots, k\}\}$ for some $k \geq 0$ and $\phi_0 \wedge \cdots \wedge \phi_{j-1}$ does not imply ϕ_j for any $j \in \{0, \ldots, k\}$. Then, we have a finite descending chain $\top, \phi_0, \phi_0 \wedge \phi_1, \ldots, \phi_0 \wedge \cdots \wedge \phi_k$ with the length $k+2$. This is a contradiction. Thus, either $\mathrm{Expand}(C, k)$ does not have a solution for any k or $\phi_0 \wedge \cdots \wedge \phi_{j-1}$ implies ϕ_j for some $j \in \{0, \ldots, k\}$. Consequently, $\mathrm{SOLVE}(C)$ aborts (in the former case) or returns a solution (in the latter case). $\square$

For example, given a set of n-predicates, let us consider a theory whose formula is $\perp$ or a conjunction of predicates in the set as in Liquid Types [25]. It is not at all impractical to require that interpolants always exist. Since the theory is $(2^n + 1)$-bounded, if we adopt such a theory, we can prove termination of SOLVE as in Liquid Types.

5. Experiments

We have implemented a prototype type inference system according to the formalization in the full paper [27]. We tested it for several programs to show the effectiveness of our approach.

Our type inference system takes a program written in a subset of OCaml as the input,[3] and outputs the inferred dependent types of the program if the type inference succeeds. If the program is ill-typed, the system reports a counter-example as an explanation of why the program is ill-typed. The system may not terminate for some well-typed program as we have discussed in Section 4.4. For computing interpolants, we adopted CSIsat interpolating theorem prover [5], which supports the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols.

We conducted two kinds of experiments. In the first one, we have verified that array programs never cause an array bounds error (see Section 5.1). In the second one, we have verified that sorting programs indeed return sorted lists. The source programs used in the experiments except for `isort` were originally written in DML [28, 29]. We have translated them into OCaml, by removing dependent type annotations. All the experiments were conducted on Intel Xeon CPU 5160 3.00GHz with 8GB RAM.

[3] Our system supports OCaml features such as data constructors, pattern-matches, tuples, and the let-polymorphism but does not support objects, modules, and imperative features such as reference cells and exceptions. Unlike in OCaml, our system allows users to define a recursively-defined data structure with detailed specifications by writing dependent types for the constructors.

Program	Lines	Time (sec.)
`bcopy`	15	0.077
`dotprod`	17	0.056
`bsearch`	24	0.164
`hanoi`	90	1.359
`queens`	92	18.885
`bcopy_bug`	15	0.061
`dotprod_bug`	17	0.041
`bsearch_bug`	24	0.200
`hanoi_bug`	90	0.296
`queens_bug`	92	0.322

Table 1. Experimental Results for Array Programs

5.1 Verification of Absence of Array Bounds Errors

The source programs include a solver for the towers of Hanoi problem (`hanoi`), a solver for the N-Queens problem (`queens`), the binary search algorithm (`bsearch`), vector dot product (`dotprod`), and array copy (`bcopy`).[4] The timing results are listed in the upper part of Table 1. The first column lists the names of the input programs. The second column shows the numbers of lines of the programs after desugaring and pretty-printing. The third column show the time (in seconds) taken by type inference. Our prototype system is not very time efficient for `queens` because the current naive implementation causes the size of input formulas to the interpolating prover to be large.

Let us consider the program `bsearch` in Figure 6. In the program, the functions `elem` and `size` are built-in array functions, where `elem vec m` returns the `m`-th element of the array `vec`, and `size vec` returns the size of the array `vec`. The functions have the following types:

$$\begin{aligned} \texttt{elem} &: \forall\alpha.\nu_1 : \alpha\ \texttt{array} \to \\ &\qquad \{\nu_2 : \texttt{int} \mid 0 \leq \nu_2 < \texttt{size}(\nu_1)\} \to \alpha \\ \texttt{size} &: \forall\alpha.\nu_1 : \alpha\ \texttt{array} \to \{\nu_2 : \texttt{int} \mid \nu_2 = \texttt{size}(\nu_1)\} \end{aligned}$$

We assume that $\texttt{size}(\nu) \geq 0$ holds for any ν. Our system automatically inferred the following types:

$$\begin{aligned} \texttt{bs\_aux} &: \texttt{int} \to vec : \texttt{int array} \to \{l : \texttt{int} \mid 0 \leq l\} \to \\ &\qquad \{u : \texttt{int} \mid u < \texttt{size}(vec)\} \to \texttt{int option} \\ \texttt{bsearch} &: \texttt{int} \to \texttt{int array} \to \texttt{int option} \end{aligned}$$

The programs `bcopy_bug`–`queens_bug` are buggy versions of `bcopy`–`queens`. We have intentionally inserted the bugs into them. As shown in the lower part of Table 1, counter-example finding is reasonably fast. As in this result, for most of ill-typed programs, we believe that only a small amount of constraint expansion is necessary for the counter-example finding.

We obtained the buggy program `bsearch_bug` from `bsearch` by modifying the recursive call `bs_aux key vec (m+1) u` in `bs_aux` to `bs_aux key vec (m-1) u` intentionally . Our system automatically found and reported the following counter-example for `bs_aux` in `bsearch_bug`:

$$-1 = l \leq m < 0 \leq u$$

[4] For the experiment of `hanoi`, we needed to give one type annotation to our system as a hint. In DML, eight type annotations are necessary for `hanoi`. In the other experiments, our system required no type annotations.

Program	Lines	Time (sec.)
isort	21	0.242
mergesort	66	10.113

Table 2. Experimental Results for Sorting Programs

The counter-example means that `bs_aux` can be called with, for example, the arguments $l = -1$ and $u = 1$, and then `bs_aux` causes an array bounds error. In fact, with the arguments, the then-branch is taken in `bs_aux` since $l \le u$ holds, m is bound to -1, and hence `elem vec m` fails.

5.2 Verification of Orderedness for Sorting Algorithms

The source programs include the insertion sort algorithm (`isort`) in Figure 7 and the merge sort algorithm (`mergesort`) in Figure 8. The timing results are listed in Table 2.

For the verification, we first defined a refined recursive data type `olist`, which represents increasing lists on integers. We declared the dependent types of the constructors `ONil` and `OCons` for `olist` as follows:

$$\begin{array}{rcl} \texttt{ONil} & : & \{\nu : \texttt{olist} \mid \nu = \texttt{nil}\} \\ \texttt{OCons} & : & \nu_1 : \{\nu : \texttt{int} \times \texttt{olist} \mid \nu.2 = \texttt{nil} \lor \nu.1 \le \texttt{hd}(\nu.2)\} \\ & & \to \{\nu_2 : \texttt{olist} \mid \nu_2 \neq \texttt{nil} \land \texttt{hd}(\nu_2) = \nu_1.1\} \end{array}$$

Here, `nil`, $\texttt{hd}(\nu)$, $\nu.1$, and $\nu.2$ denote the empty list, the head of the ordered list ν, the first and second elements of the tuple ν respectively. The precondition $\nu.2 = \texttt{nil} \lor \nu.1 \le \texttt{hd}(\nu.2)$ of `OCons` ensures that the constructed list is increasing. Note that the definition of `olist` is required for specifying the property to be verified in this experiment. Then, our system automatically inferred the following types for the insertion sort:

$$\begin{array}{rcl} \texttt{insert} & : & (\nu_1 : \texttt{int} \to \nu_2 : \texttt{olist} \to \\ & & \{\nu_3 : \texttt{olist} \mid \\ & & \quad \nu_2 \neq \texttt{nil} \land \texttt{hd}(\nu_2) \le \texttt{hd}(\nu_3) \lor \\ & & \quad \nu_1 \le \texttt{hd}(\nu_3)\}) \\ \texttt{isort} & : & (\texttt{int list} \to \texttt{olist}) \end{array}$$

Similarly, our system automatically inferred the following types for the merge sort:

$$\begin{array}{rcl} \texttt{merge} & : & (\nu_1 : \texttt{olist} \to \nu_2 : \texttt{olist} \to \\ & & \{\nu_3 : \texttt{olist} \mid \\ & & \quad \nu_1 = \nu_2 = \texttt{nil} \lor \\ & & \quad \nu_1 = \texttt{nil} \land \nu_2 = \nu_3 \neq \texttt{nil} \lor \\ & & \quad \nu_1 = \nu_3 \neq \texttt{nil} \land \nu_2 = \texttt{nil} \lor \\ & & \quad \nu_1 \neq \texttt{nil} \land \nu_2 \neq \texttt{nil} \land \texttt{hd}(\nu_1) \le \texttt{hd}(\nu_3) \lor \\ & & \quad \nu_1 \neq \texttt{nil} \land \nu_2 \neq \texttt{nil} \land \texttt{hd}(\nu_2) \le \texttt{hd}(\nu_3)\}) \\ \texttt{initList} & : & (\texttt{int list} \to \texttt{olist list}) \\ \texttt{mergeList} & : & (\texttt{olist list} \to \texttt{olist list}) \\ \texttt{mergeAll} & : & (\texttt{olist list} \to \texttt{olist}) \\ \texttt{mergesort} & : & (\texttt{int list} \to \texttt{olist}) \end{array}$$

In DML, users need to declare these complex specifications manually. Since these specifications are not given explicitly in the programs, Liquid Types with the simple predicate mining heuristics [25] seem unable to infer these specifications automatically.

Remark 2. The current implementation requires users to use the different sets $\{\texttt{Nil}, \texttt{Cons}\}$ and $\{\texttt{ONil}, \texttt{OCons}\}$ of constructors for the different refinement types `list` and `olist` respectively of the same data structure. However, even if the same set of constructors

```
type 'a list =
  Nil: 'a list
| Cons: 'a * 'a list -> 'a list
type olist =
  ONil: {x:olist | x = nil}
| OCons: (x1:{x:int*olist | snd(x) = nil \/
                            fst(x) <= hd(snd(x))} ->
             {x2:olist | x2 <> nil /\
                         hd(x2) = fst(x1)})
let rec insert x xs = match xs with
    ONil -> OCons(x, ONil)
  | OCons(y, ys) ->
      if x <= y then OCons(x, OCons(y, ys))
      else OCons(y, insert x ys)
let rec isort xs = match xs with
    Nil -> ONil
  | Cons(x, xs') -> insert x (isort xs')
let _ = isort xs
```

Figure 7. Verified Insertion Sorting Program

is used for `list` and `olist`, we believe that we can select an appropriate refinement type that conforms to the context for each occurrences of the constructors by using local type inference [18, 24].

6. Related Work

6.1 Dependently Typed Languages

Dependent types have been introduced to programming languages for verification of detailed specifications of programs [1, 2, 28, 29]. These languages require users to write type annotations for all functions unlike in our system, and then performs type checking.

Proof assistants support interactive development of dependently typed programs [4]. The present proof assistants seem, however, difficult to use for ordinary programmers without a knowledge of type theory and higher-order logic.

6.2 Dependent Type Inference Algorithms

There are other studies on inferring dependent types. The most distinguishing feature of our algorithm is the ability to generate a counter-example when a given program is ill-typed.

Flanagan proposed hybrid type checking, which allows users to refine data types with arbitrary program terms [13]. Knowles and Flanagan [21] proposed a constraint generation algorithm similar to the one discussed in Section 3, but did not give a constraint solving algorithm.

Rondon et al. proposed a type inference algorithm [25] based on predicate abstraction [14] for a variant of the Knowles and Flanagan's dependent type system. Compared to their algorithm, our algorithm can automatically discover predicates used in constraint solving, while their algorithm assumes given predicates for program abstraction. Another difference is that our algorithm is based on the lazy abstraction paradigm [17, 23]: we infer precise dependent types only for program fragments where complex specifications are required, and just infer simple types for the other fragments. In contrast, Liquid Types [25] do not change the predicates for abstraction depending on what is required at each program fragment.

Size inference can automatically infer size relations between arguments and return values of functions [8, 19]. Size inference tries to infer as precise dependent types as possible from functions' definitions only. Compared to size inference, an advantage of our algorithm is that it can refine recursive data types with dependent types

```
type 'a list =
  Nil: 'a list
| Cons: 'a * 'a list -> 'a list
type olist =
  ONil: {x:olist | x = nil}
| OCons: (x1:{x:int*olist | snd(x) = nil \/
                            fst(x) <= hd(snd(x))} ->
             {x2:olist | x2 <> nil
                        /\ hd(x2) = fst(x1)})
let rec merge xs ys = match xs with
    ONil -> ys
  | OCons(x, xs') ->
      (match ys with
        ONil -> xs
      | OCons(y, ys') ->
          if x <= y then OCons(x, merge xs' ys)
          else OCons(y, merge xs ys'))
let rec initList xs = match xs with
    Nil -> Nil
  | Cons(x1, xs1) ->
      (match xs1 with
        Nil -> Cons(OCons(x1, ONil), Nil)
      | Cons(x2, xs2) ->
          let l =
            if x1 <= x2 then
              OCons(x1, (OCons(x2, ONil)))
            else OCons(x2, (OCons(x1, ONil)))
          in Cons(l, initList xs2))
let rec mergeList ls = match ls with
    Nil -> Nil
  | Cons(l1, ls') ->
      (match ls' with
        Nil -> ls
      | Cons(l2, ls'') ->
          Cons(merge l1 l2, mergeList ls''))
let rec mergeAll ls = match ls with
    Nil -> ONil
  | Cons(l, ls') ->
      (match ls' with
        Nil -> l
      | Cons(_, _) -> mergeAll (mergeList ls))
let mergesort l = mergeAll (initList l)
let _ = mergesort xs
```

Figure 8. Verified Merge Sorting Program

based on the user's demand as demonstrated in the verification of the sorting programs in Section 5.2, which cannot be verified by size inference. On the other hand, an advantage of size inference is that it can infer a more precise dependent type of a function than ours from only the definition of the function.

Our previous work can use both information about functions' definitions and call-sites for refining the dependent types of the functions on demand [26]. However, a function's output specification is determined by taking only a part of information from the call sites. Our algorithm presented in this paper extends the previous work so that we can determine the output specification by taking both information from the definition and the call sites.

6.3 Other Work

The Boyer-Moore theorem provers such as ACL2 [6, 7] can automatically prove inductive theorems of Lisp functions. For example, ACL2 can verify the orderedness of the insertion sort algorithm. However, it does not directly support partial functions and functions with input specifications unlike in our type inference algorithm.

One of the important components of our algorithm is interpolating provers [5, 16, 22]. They have been applied to discovering predicates for program abstraction in model checkers [17, 23]. They iteratively refine a program abstraction with interpolants computed from a spurious error path so that the refined abstraction can correctly judge that the path is safe.

Haack and Wells proposed a technique called type error slicing for computing a slice of an ill-typed program that is sufficient and necessary for a type error to cause as an explanation of why the program is ill-typed [15].

Our use of interpolants in dependent type inference has been inspired from the use of interpolants in model checkers for imperative programs. [17, 23] The main advantage of our type-based approach over them is that we can easily support advanced programming features such as higher-order functions, polymorphic functions, and recursively defined data structures.

7. Conclusion

We proposed a novel type inference algorithm for a dependently-typed functional language, which is essentially an "implicitly-typed" version of DML [29]. Our type inference algorithm is novel because of the use of an interpolating prover. It can iteratively refine dependent types with interpolants until the type inference succeeds or the program is found to be ill-typed. In the latter case, it can generate a kind of counter-example as an explanation of why the program is ill-typed. To our knowledge, none of the usual type inference algorithms generate a counter-example. We have implemented a prototype type inference system, which supports OCaml features such as data constructors, pattern-matches, tuples, and the let-polymorphism and tested it for array and sorting programs. As a result, our system has successfully verified them. In particular, our system has automatically inferred the complex dependent type for the helper function `merge` of the merge sort defined in Figure 8, which is very hard to declare manually by ordinary programmers, and can not be inferred automatically by existing dependent type inference algorithms [8, 19, 25]. For the array programs with bugs, our system has found counter-examples in a reasonably fast time.

In general, type inference algorithms are desired to have the modularity and scalability. Our algorithm allows modular type inference. For example, when a programmer want to verify his/her module that uses a list library module, our algorithm does not require the source code of the list library if the dependent types of the exported list library functions are provided as the module interface by the library's designer. If the library source code is available, our algorithm may perform more precise type inference for the programmer's module. To make our system more scalable, we plan to improve our prototype implementation and the interpolating prover.

As future work, we also plan to support more features of OCaml such as reference cells and exceptions. To deal with reference cells, we believe that we only need to give a constraint generation rule for them. However, for exceptions, it is not clear now whether we need to extend our constraint solving algorithm to deal with constraints of the form different from the one discussed in this paper.

Another direction of future work is to extend our type inference system so that it can verify more detailed properties than those we have dealt with in this paper. For the purpose, we may extend the underlying theory in our dependent type system with the theories of lists, arrays, sets, and multi-sets. For example, if we use the theory of multi-sets, we may verify that the sorting functions always return a list whose elements are a permutation of the elements of the argument as in the collection analysis [9]. To extend our constraint solving algorithm based on interpolants with those theories, we need to extend the interpolating prover to support them.

Acknowledgments

We would like to thank anonymous referees for useful comments. This work was partially supported by Kakenhi 20240001.

References

[1] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[2] L. Augustsson. Cayenne – a language with dependent types. In *ICFP '98*, pages 239–250. ACM Press, 1998.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01*, pages 203–213. ACM Press, 2001.

[4] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[5] D. Beyer, D. Zufferey, and R. Majumdar. CSIsat : Interpolation for LA+EUF (tool paper). In *CAV '08*, volume 5123 of *Lecture Notes in Computer Science*, pages 304–308, July 2008.

[6] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, 1975.

[7] A. Bundy. The automation of proof by mathematical induction. In *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[8] W.-N. Chin and S.-C. Khoo. Calculating sized types. In *PEPM '00*, pages 62–72. ACM Press, 1999.

[9] W.-N. Chin, S.-C. Khoo, and D. N. Xu. Extending sized type with collection analysis. In *PEPM '03*, pages 75–84. ACM Press, 2003.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM Press, 1977.

[12] W. Craig. Linear reasoning. a new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22:250–268, September 1957.

[13] C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256. ACM Press, 2006.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.

[15] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP '03*, volume 2618 of *Lecture Notes in Computer Science*, pages 284–301. Springer-Verlag, April 2003.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL '04*, pages 232–244. ACM Press, 2004.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM Press, 2002.

[18] H. Hosoya and B. C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[19] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM Press, 1996.

[20] D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *SIGSOFT '06/FSE-14*, pages 105–116. ACM, 2006.

[21] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP '07*, volume 4421 of *Lecture Notes in Computer Science*, pages 505–519. Springer-Verlag, March/April 2007.

[22] K. L. McMillan. An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

```
procedure MSOLVE($C$) :
 1 :  $\Pi \leftarrow \{\epsilon\}$
 2 :  while true do
 3 :    let $C' = \mathrm{Expand}(C, \Pi)$
 4 :    match $\mathrm{SolveExpanded}(C')$ with
 5 :      Unsatisfiable $\rightarrow$ abort
 6 :    | Satisfiable($\theta'$) $\rightarrow$
 7 :        let $\{P_i^\pi \mapsto \lambda\widetilde{x_i}.\phi_i^\pi \mid \pi\in\Pi, i\in\{1,\ldots,m\}\} = \theta'$
 8 :        $\Pi' \leftarrow \{\epsilon\}$
 9 :        while true do
10 :          let $\Pi_i = \Pi' \backslash \mathrm{Leaves}(\Pi', i)$
11 :          let $\theta_{\Pi'} = \{P_i \mapsto \lambda\widetilde{x_i}.\bigwedge_{\pi\in\Pi_i}\phi_i^\pi \mid i\in\{1,\ldots,m\}\}$
12 :          if $\theta_{\Pi'}\,P_i(\widetilde{x_i})$ does not imply $\phi_i^\pi$ for some
13 :            $i\in\{1,\ldots,m\}$ and $\pi\in\mathrm{Leaves}(\Pi', i)$ then
14 :            if $\pi \cdot i \in \Pi$ then
15 :              $\Pi' \leftarrow \Pi' \cup \{\pi \cdot i\}$
16 :            else
17 :              $\Pi \leftarrow \Pi \cup \{\pi \cdot i\}$
18 :              break the inner loop
19 :          else
20 :            return $\theta_{\Pi'}$
```

Figure 9. Constraint Solving Algorithm based on Interpolants (Multiple Predicate Variable Version)

[23] K. L. McMillan. Lazy abstraction with interpolants. In *CAV '06*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag, August 2006.

[24] B. C. Pierce and D. N. Turner. Local type inference. In *POPL '98*, pages 252–265. ACM Press, 1998.

[25] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*. ACM Press, 2008.

[26] H. Unno and N. Kobayashi. On-demand refinement of dependent types. In *FLOPS '08*, volume 4989 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, April 2008.

[27] H. Unno and N. Kobayashi. Dependent type inference with interpolants (Full version), June 2009. Available from `http://www.kb.ecei.tohoku.ac.jp/~uhiro/`.

[28] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98*, pages 249–257. ACM Press, 1998.

[29] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM Press, 1999.

Appendix

A. Extension to Multiple Predicate Variables

In this section, we extend the constraint solving algorithm presented in Sections 4.1-4.4 to support multiple predicate variables.

Figure 9 presents the constraint solving algorithm MSOLVE for constraints on multiple predicate variables. In the lines 7–20, the procedure MSOLVE iteratively obtains a candidate solution $\theta_{\Pi'}$ (see the line 11) from the solution θ' of an expanded constraint $\mathrm{Expand}(C, \Pi)$, and checks whether it is genuine (see the lines 12–13). If no candidate solution is genuine, MSOLVE expands the original constraint further (see the line 17).

Constraint Expansion A constraint C generated from a program by the constraint generation algorithm described in Section 3 can always be transformed to the following form:

$$\begin{array}{l}(F(P_1,\dots,P_m)\Rightarrow\bot)\wedge\\(\forall\widetilde{x_1}.G_1(P_1,\dots,P_m)(\widetilde{x_1})\Rightarrow P_1(\widetilde{x_1}))\wedge\cdots\wedge\\(\forall\widetilde{x_m}.G_m(P_1,\dots,P_m)(\widetilde{x_m})\Rightarrow P_m(\widetilde{x_m}))\end{array}$$

Here, $P_1,\dots,P_m$ are predicate variables, and $F(P_1,\dots,P_m)$ and $G_i(P_1,\dots,P_m)(\widetilde{x})$ are of the following form:

$$\exists\widetilde{y}.\left(\begin{array}{l}(P_{(1,1)}(\widetilde{x_{(1,1)}})\wedge\cdots\wedge P_{(1,l_1)}(\widetilde{x_{(1,l_1)}})\wedge\phi_1)\vee\cdots\vee\\(P_{(n,1)}(\widetilde{x_{(n,1)}})\wedge\cdots\wedge P_{(n,l_n)}(\widetilde{x_{(n,l_n)}})\wedge\phi_n)\end{array}\right)$$

Here, $P_{(j,k)}\in\{P_1,\dots,P_m\}$ for all j,k and $\exists\widetilde{y}$ binds all free variables except for $\widetilde{x}$.

We can expand the possibly recursive original constraint C to obtain non-recursive expanded constraints that are defined as follows:

Definition A.1. Let C be the following constraint:

$$\begin{array}{l}(F(P_1,\dots,P_m)\Rightarrow\bot)\wedge\\(\forall\widetilde{x_1}.G_1(P_1,\dots,P_m)(\widetilde{x_1})\Rightarrow P_1(\widetilde{x_1}))\wedge\cdots\wedge\\(\forall\widetilde{x_m}.G_m(P_1,\dots,P_m)(\widetilde{x_m})\Rightarrow P_m(\widetilde{x_m}))\end{array}$$

Let X^* to denote the set of sequences of the elements in X. We write ϵ for the empty sequence. For $x_1,x_2\in X^*$, we write $x_1\sqsubseteq x_2$ if x_1 is a prefix of x_2. We say $Y\subseteq X^*$ is prefix-closed if for all $x_1,x_2\in X^*$ such that $x_1\sqsubseteq x_2$, $x_2\in Y$ implies $x_1\in Y$. For each prefix-closed and non-empty subset Π of $\{1,\dots,m\}^*$, we define an *expanded constraint* $\mathrm{Expand}(C,\Pi)$ with the predicate variables $\{P_1^\pi,\dots,P_m^\pi\mid\pi\in\Pi\}$ as follows:

$$\mathrm{Expand}(C,\Pi):=\bigwedge_{\pi\in\Pi}\mathrm{Expand}(C,\pi)$$

Here, $\mathrm{Expand}(C,\pi)$ is defined as follows:

$$\begin{array}{rcl}\mathrm{Expand}(C,\epsilon)&:=&F(P_1^\epsilon,\dots,P_m^\epsilon)\Rightarrow\bot\\\mathrm{Expand}(C,\pi\cdot i)&:=&\forall\widetilde{x_i}.G_i(P_1^{\pi\cdot i},\dots,P_m^{\pi\cdot i})(\widetilde{x_i})\Rightarrow P_i^\pi(\widetilde{x_i})\end{array}$$

The following lemma follows immediately from the construction of $\mathrm{Expand}(C,\Pi)$ above.

Lemma A.1. *For any constraint C and prefix-closed and non-empty subset Π of $\{1,\dots,m\}^*$, $\mathrm{Expand}(C,\Pi)$ has a solution if C has a solution.*

Proof. Solutions for $P_1,\dots,P_m$ in C are solutions for $P_1^{(\pi)},\dots,P_m^{(\pi)}$ in $\mathrm{Expand}(C,\Pi)$ for all Π. □

Solving Expanded Constraints The sub-procedure SolveExpanded checks the satisfiability and finds a solution of $\mathrm{Expand}(C,\Pi)$ in a similar manner to the algorithm for $\mathrm{Expand}(C,i)$ explained in Section 4.2. An additional technical requirement lies in solving constraints of the form:

$$\begin{array}{l}(\forall\widetilde{y_1},\mathrm{FV}(\phi_1).\phi_1\Rightarrow Q_1(\widetilde{y_1}))\wedge\cdots\wedge\\(\forall\widetilde{y_n},\mathrm{FV}(\phi_n).\phi_n\Rightarrow Q_n(\widetilde{y_n}))\wedge\\(\forall\widetilde{y_1},\dots,\widetilde{y_n},\mathrm{FV}(\phi).Q_1(\widetilde{y_1})\wedge\cdots\wedge Q_n(\widetilde{y_n})\Rightarrow\phi)\end{array}$$

For each $i=n,\dots,2,1$, we can iteratively compute a solution $\lambda\widetilde{y_i}.\phi'_i$ for Q_i as an interpolant of $(\lambda\widetilde{y_i}.\phi_i,\lambda\widetilde{y_i}.\phi_1\wedge\cdots\wedge\phi_{i-1}\wedge\phi'_{i+1}\wedge\cdots\wedge\phi'_n\Rightarrow\phi)$.

Checking Genuineness of Candidate Solutions The correctness of the genuineness checking of candidate solutions (see the lines 7–20 in Figure 9) is established by the following lemma:

Lemma A.2. *We define leaves* $\mathrm{Leaves}(\Pi,i)$ *of* Π *by* $\{\pi\in\Pi\mid\pi\cdot i\not\sqsubseteq\pi'$ *for any* $\pi'\in\Pi\}$. *Suppose that an expanded constraint* $\mathrm{Expand}(C,\Pi)$ *has a solution* $\{P_i^\pi\mapsto\lambda\widetilde{x_i}.\phi_i^\pi\mid\pi\in\Pi,i\in\{1,\dots,m\}\}$. *Let* $\theta_{\Pi'}=\{P_i\mapsto\lambda\widetilde{x_i}.\bigwedge_{\pi\in\Pi'\setminus\mathrm{Leaves}(\Pi',i)}\phi_i^\pi\mid i\in\{1,\dots,m\}\}$. *If there exists a prefix-closed and non-empty subset* Π' *of* Π *such that* $\theta_{\Pi'}P_i(\widetilde{x_i})$ *implies* ϕ_i^π *for all* $i\in\{1,\dots,m\}$ *and* $\pi\in\mathrm{Leaves}(\Pi',i)$, *then* $\theta_{\Pi'}$ *is a solution of* C.

Proof. For all $i\in\{1,\dots,m\}$ and $\pi\in\Pi$, we have:

$$\begin{array}{rcl}F(\lambda\widetilde{x_1}.\phi_1^\epsilon,\dots,\lambda\widetilde{x_m}.\phi_m^\epsilon)&\Rightarrow&\bot,\\G_i(\lambda\widetilde{x_1}.\phi_1^{\pi\cdot i},\dots,\lambda\widetilde{x_m}.\phi_m^{\pi\cdot i})(\widetilde{x_i})&\Rightarrow&\phi_i^\pi(\widetilde{x_i}).\end{array}$$

Assume that there exists a prefix-closed and non-empty subset Π' of Π such that $\theta_{\Pi'}P_i(\widetilde{x_i})$ implies ϕ_i^π for all $i\in\{1,\dots,m\}$ and $\pi\in\mathrm{Leaves}(\Pi',i)$.

- If $\Pi'=\{\epsilon\}$, we get $\theta_{\Pi'}P_i=\lambda\widetilde{x_i}.\top$, and $\phi_i^\epsilon\equiv\top$ by the assumption. Thus, we have $F(\theta_{\Pi'}P_1,\dots,\theta_{\Pi'}P_m)\Rightarrow\bot$ and $G_i(\theta_{\Pi'}P_1,\dots,\theta_{\Pi'}P_m)(\widetilde{x_i})\Rightarrow\theta_{\Pi'}P_i(\widetilde{x_i})$.
- Otherwise, we get:

  $$\bot\quad\Leftarrow\quad F(\theta_{\Pi'}P_1,\dots,\theta_{\Pi'}P_m)\quad\text{(by monotonicity of }F)$$

  We can also show that:

  $$\begin{array}{rcl}\theta_{\Pi'}P_i(\widetilde{x_i})&\Leftarrow&G_i(\lambda\widetilde{x_1}.\bigwedge_{\pi\in\Pi'}\phi_1^\pi,\dots,\lambda\widetilde{x_m}.\bigwedge_{\pi\in\Pi'}\phi_m^\pi)\\&&\text{(by monotonicity of }G)\\&\Leftarrow&G_i(\theta_{\Pi'}P_1,\dots,\theta_{\Pi'}P_m)(\widetilde{x_i})\\&&\text{(by the assumption)}\end{array}$$

□

Correctness The following theorem, which follows immediately from Lemmas A.1, A.2, establishes the correctness of MSOLVE:

Theorem A.3 (Correctness). *(a) If* $\mathrm{MSOLVE}(C)$ *returns* θ, θ *is a solution of* C. *(b) If* $\mathrm{MSOLVE}(C)$ *aborts,* C *is not satisfiable.*

B. Optimizations

The procedure MSOLVE in Figure 9 can further be optimized. After Π is updated to $\Pi\cup\{\pi\cdot i\}$ in the line 17, we recompute a solution for $\mathrm{Expand}(C,\Pi\cup\{\pi\cdot i\})$ in the line 4. This can be optimized by using information about the previous solution for $\mathrm{Expand}(C,\Pi)$. Then, we reconstruct a subset Π' of $\Pi\cup\{\pi\cdot i\}$ in the lines 7–20. This can also be optimized by using information about the subset Π' of Π constructed previously. After Π' is updated to $\Pi'\cup\{\pi\cdot i\}$ in the line 15, we recheck the conditions on $\theta_{\Pi'\cup\{\pi\cdot i\}}$ in the lines 12–13. This can be optimized by using information about the conditions on $\theta_{\Pi'}$ checked previously.

In Appendix A, we expressed the form of expanded constraints by using the functions F and G_i on predicates, and their arguments were fixed to $P_1,\dots,P_m$. This may make the constraint solving inefficient for two reasons: (1) Even though some predicate variable P_j may not actually occur in the definitions of the functions, the algorithm may wastefully expands P_j. (2) Different occurrences of the same predicate variable are not distinguished and expanded in the same way even though different solutions may be required for the different occurrences. To remedy the problem, we can express the form of constraints as follows:

$$\begin{array}{l}(F(\widetilde{P_0})\Rightarrow\bot)\wedge(\forall\widetilde{x_1}.G_1(\widetilde{P_1})(\widetilde{x_1})\Rightarrow P_1(\widetilde{x_1}))\wedge\cdots\wedge\\(\forall\widetilde{x_m}.G_m(\widetilde{P_m})(\widetilde{x_m})\Rightarrow P_m(\widetilde{x_m}))\end{array}$$

Here, $\widetilde{P_i}$ denotes a sequence of the set $\{P_1,\dots,P_m\}$. The elements of $\widetilde{P_i}$ represent the occurrences of the predicate variables $P_1,\dots,P_m$ in G_i (or F if $i=0$).