Dependent Type Inference with Interpolants

Hiroshi Unno  
Tohoku University  
hiro@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 \( \text{fail} \) 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:

\[
\begin{align*}
\text{let } \_ &= \text{assert (inc } y \geq y) \\
\text{let } x &= x + 1
\end{align*}
\]

Here, the assertion \( \text{assert (inc } y \geq y) \) checks whether the argument holds, and gets stuck if the check fails. We can prepare the type template:

\[
\text{inc : } (\nu_1 : \text{int } \rightarrow \{ \nu_2 : \text{int } | P(\nu_1, \nu_2) \})
\]

It means that the function takes an integer \( \nu_1 \) as an argument, and returns an integer \( \nu_2 \) 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 \( \phi_1 \) and \( \phi_2 \) is another formula \( \phi \) such that \( \phi_1 \) implies \( \phi \), \( \phi \) implies \( \phi_2 \), and the free variables of \( \phi \) must occur in both \( \phi_1 \) and \( \phi_2 \). 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 $\triangleq n c$ 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 $P(v_1, v_2) \equiv v_2 \geq v_1$ 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:

```plaintext
let rec sum x = if x <= 0 then 0 else 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:

```
sum : (ν1 : int → {ν2 : int | P(ν1, ν2)})
```

Here, $P(v_1, v_2)$ represents the output specification of $\text{sum}$. (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]:

$C_1 := \forall v_1, v_2, ν_1', ν_2'. (P(ν_1', ν_2') \land ψ_3) \Rightarrow P(v_1, v_2)),$

$C_2 := \forall ν_1, ν_2, y. (P(ν_1, ν_2) \Rightarrow ψ_3).$

Here, $φ_1, φ_2$, and $φ_3$ are given by:

$φ_1 := ν_1 \leq 0 \land ν_2 = 0,$

$φ_2 := ν_1 > 0 \land ν'_1 = ν_1 - 1 \land ν_2 = ν_1 + ν'_2,$

$φ_3 := y = ν_1 \lor ν_2 \geq y.$

The constraint $C_1$ is generated from the definition of $\text{sum}$, and $C_2$ from the assertion, the call-site of $\text{sum}$. The sub-formulas $φ_1$ and $P(ν'_1, ν'_2) \land ψ_2$ 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 $\triangleq n c$ 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 \emph{solution} of $C$ if $θC$ is a tautology. A constraint $C$ is \emph{satisfiable} if it has a solution. We often omit universal quantifiers on-first-order variables; for example, we write just $P(v_1, v_2) \Rightarrow φ_3$ 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 $P_2$ in the left-hand side of $C_2$ with $P_0$, getting the following “non-recursive” constraint:

$C_3 := φ_1 \lor (P_1(ν'_1, ν'_2) \land ψ_2) \Rightarrow P_0(ν_1, ν_2),$  

$C_4 := P_0(ν_1, ν_2) ⇒ φ_3.$

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

**Solving Expanded Constraint**

Now, we can obtain a solution of the constraint $C_3 \land C_4$ 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 $P_1(ν'_1, ν'_2)$ in $C_3$ with the inconsistency $\bot$ without affecting the satisfiability as follows:

$φ_1 \lor (\bot \land φ_2) ⇒ P_0(ν_1, ν_2).$

Thus, $P_0(ν_1, ν_2) \equiv ψ_0$, we obtain $P_1$ as follows. We can replace $P_0(ν_1, ν_2)$ in $C_3$ and $C_4$ with $ψ_0$ without affecting the satisfiability as follows:

$C'_3 := φ_1 \lor (P_1(ν'_1, ν'_2) \land φ_2) ⇒ ψ_0,$

$C'_4 := ψ_0 ⇒ φ_3.$

The constraint $C'_3 \land C'_4$ can be simplified to:

$P_1(ν'_1, ν'_2) ⇒ (φ_2 ⇒ ψ_0)$

by using the fact that $ψ_0$ is an interpolant of $φ_1 \lor (\bot \land φ_2)(\equiv φ_1)$ and $φ_3$. Note that $P_1(ν'_1, ν'_2)$ can then be obtained as an interpolant of $\bot$ and $φ_2 ⇒ ψ_0$.

**Checking Genuineness of Candidate Solutions**

Suppose that $P_0(ν_1, ν_2) \equiv ψ_0$, $P_1(ν'_1, ν'_2) \equiv ψ_1$ is a solution of $C_3 \land C_4$, i.e., the following formulas $φ_A$ and $φ_B$ hold:

$φ_A := φ_1 \lor (ψ_1 \land φ_2) ⇒ ψ_0,$  

$φ_B := ψ_0 ⇒ φ_3.$

If the condition $ψ_0 ⇒ ψ_1$ holds, then $P_0(ν_1, ν_2) \equiv ψ_0$ is a solution of the original constraint $C_1 \land C_2$, and hence the type inference succeeds: $P(ν_1, ν_2) \equiv ψ_0$ satisfies $C_1$ because $φ_1 \lor (ψ_0 \land φ_2)$ implies $φ_1 \lor (ψ_1 \land φ_2)$ and $φ_A$ holds, and satisfies $C_2$ because $φ_B$ holds. Similarly, if another condition $T ⇒ ψ_0$ holds, then $P(ν_1, ν_2) \equiv T$ is a solution of the original constraint $C_1 \land C_2$. Thus, we call $P(ν_1, ν_2) \equiv ψ_0$ and $P(ν_1, ν_2) \equiv T$ candidate solutions of $C_1 \land C_2$, and check their genuineness by using the conditions $ψ_0 ⇒ ψ_1$ and $T ⇒ ψ_0$ respectively.
Iterative Dependent Type Refinement If neither $\psi_0 \Rightarrow \psi_1$ nor $\top \Rightarrow \psi_0$ holds, then we further expand the “recursive” constraint $C_1 \land C_2$ to obtain the following one:

\[
\begin{align*}
& \phi_1 \lor (P_0(\nu_1, \nu_2)) \land (P_1(\nu_1, \nu_2)) \land (P_0(\nu_1, \nu_2) \Rightarrow \phi_3).
\end{align*}
\]

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 $\phi_1$ and $\phi_2$. 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 $C_3$:

\[
\phi_1' \lor (P_1(\nu_1', \nu_2') \land \phi_2') \Rightarrow P(\nu_1, \nu_2).
\]

Here, $\phi_1'$ and $\phi_2'$ are given by:

\[
\begin{align*}
& \phi_1' \quad : \quad \nu_1 \leq 1 \land \nu_2 = 0, \\
& \phi_2' \quad : \quad \nu_1' > 1 \land \nu_1' = \nu_1 - 1 \land \nu_2 = \nu_1 + \nu_2'.
\end{align*}
\]

Then, an interpolant of $\phi_1' \lor (\bot \land \phi_2')(\equiv \phi_1')$ and $\phi_2(\equiv y = \nu_1 \Rightarrow \nu_2 \geq y)$ does not exist, as $\phi_1' \Rightarrow \phi_2'$ 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 $\text{fail}$ that expresses a failure of a program. We then formalize a dependent type system for the language, which can ensure that $\text{fail}$ 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:

\[
\begin{align*}
\text{expressions:} & \quad e \quad ::= \quad x \quad \text{variable} \\
& \quad | \quad c \quad \text{constant} \\
& \quad | \quad \lambda x.e \quad \text{abstraction} \\
& \quad | \quad e_1 \cdot e_2 \quad \text{application} \\
& \quad | \quad \text{fix } x.e \quad \text{fixed-point} \\
& \quad | \quad \text{if } x \text{ then } e_1 \text{ else } e_2 \quad \text{if-then-else} \\
& \quad | \quad \text{fail} \quad \text{failure}
\end{align*}
\]

Here, $x$ and $c$ are meta-variables ranging over variables and constants respectively. We write $\text{FV}(e)$ 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 $\text{if } x \text{ then } e_1 \text{ else } e_2$ 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 $\text{fail}$ always gets stuck. We can use $\text{fail}$ to model array accesses and assertions.

The syntax of types is defined as follows:

\[
\begin{align*}
\text{specifications:} & \quad \psi \quad ::= \quad P(x) \mid \phi \quad \text{integer refinements} \\
\text{dependent types:} & \quad T \quad ::= \quad \nu : \text{int} \mid \psi \quad \text{dependent function types} \\
& \quad | \quad \nu : T_1 \rightarrow T_2 \quad \text{dependent function types} \\
\text{environments:} & \quad \Gamma \quad ::= \quad \emptyset \mid \Gamma, x : T \quad \text{Type Environments}
\end{align*}
\]

Here, $P$ and $\phi$ 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 $\nu$, which ranges over the variables not appearing in expressions. We writes $\mathcal{FV}(\phi)$ to denote the set of free variables in $\phi$.

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 \texttt{int}. For example, \{\nu : \texttt{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, \(\nu_1 : \texttt{int} \rightarrow \nu_2 : \texttt{int} \rightarrow [\nu_3 : \texttt{int} \mid \nu_3 = \nu_1 + \nu_2]\) denotes the type of functions whose return value (denoted by \(\nu_3\)) is the sum of the two arguments (denoted by \(\nu_1\) and \(\nu_2\)). A type environment \(\Gamma\) is a sequence of type bindings \(x : T\), which may include guard formulas \(\phi\). 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

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

The subtyping relation \(\Gamma \vdash T_1 <: T_2\) is defined as follows:

\[
\Gamma \vdash e : T \quad \text{and} \quad \text{Sub}(\Gamma, T_1, T_2)
\]

#### Figure 2. Typing Rules

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

### 3.3 Constraint Generation Algorithm

The constraint generation algorithm is shown in Figure 4. The function \(\text{Gen}\) takes \(\Gamma, e,\) and \(T\) as inputs, and returns a constraint \(C\) on predicate variables, such that \(\theta\) is a solution of \(\exists \overline{P}.C\) if and only if \(\Gamma \vdash e : \theta T\) holds. Here, \(\overline{P}\) is the set of the predicate variables occurring in \(C\) but not in \(\Gamma\) and \(T\). The function \(\text{Gen}_{\leq}\) takes \(\Gamma, T_1,\) and \(T_2\) as inputs, and returns a constraint \(C\) on predicate variables, such that \(\theta\) is a solution of \(C\) if and only if \(\Gamma \vdash \theta T_1 : \theta T_2\) holds.

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

\[
\text{Lift}(\overline{\tau}; \pi) = \nu : \text{Lift}(\overline{\tau}; \pi) \rightarrow \text{Lift}(\overline{\tau}; \pi) \quad (\nu : \text{fresh})
\]

\[
\text{Lift}(\overline{\tau}; \pi) = \{\nu : \text{int} \mid P(\overline{\tau}, \pi)\} \quad (\nu : P : \text{fresh})
\]

#### Example 3.2

Let us consider the program of \texttt{inc} in Section 1. The program can be encoded as follows in our language:

\[
e_{\text{inc}} = (\lambda x.e_{\text{inc}}) (\lambda x. + x 1)
\]

Here, \(e_{\text{inc}} = (\lambda b.\text{if } b\text{ then } 0\text{ else }(-1))\). The Hindley-Milner algorithm infers the type \(\texttt{int}\) for \(e_{\text{inc}}\), and then the constraint generation for \(e_{\text{inc}}\) proceeds as follows:

\[
\text{Gen}(\vdash e_{\text{inc}} : \texttt{int}) = \text{Gen}(\vdash \lambda x.e_{\text{inc}} : (\texttt{inc} : T \rightarrow \texttt{int})) \land \text{Gen}(\vdash \lambda x. + x 1 : T)
\]

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

\[
\text{Gen}(\vdash \lambda x. + x 1 : T) = \text{Gen}(x : T_1 \vdash \cdot + 1 : [x/\nu_1]T_2)
\]

\[
= \text{Gen}(x : T_1 \vdash \cdot + \{\nu : \text{int} \mid R(\lambda c, \nu)\} \rightarrow [x/\nu_1]T_2) \land
\]

\[
\text{Gen}(x : T_1 \vdash \cdot : \{\nu : \text{int} \mid S(\lambda c, \nu)\}) \land
\]

\[
\text{Gen}(x : T_1 \vdash \cdot : \{\nu : \text{int} \mid S(\lambda c, \nu)\}) \land
\]

\[
= (P(x) \rightarrow R(\lambda c, \nu) \land S(\lambda c, \nu)) \land
\]

\[
= (P(x) \land \nu = x) \rightarrow R(\lambda c, \nu) \land
\]

\[
= P(x) \land \nu = x \rightarrow S(\lambda c, \nu) \land
\]

\[
\text{Here, } R \text{ and } S \text{ are fresh predicate variables, which represent the specifications of the sub-expressions } x \text{ and } 1 \text{ respectively. Similarly, the part } \text{Gen}(\vdash \text{inc} : \text{inc} : T \rightarrow \texttt{int}) \text{ is evaluated to the following constraint:}
\]

\[
(v = y \Rightarrow P(\nu) \land Q(\nu_1, \nu_2) \Rightarrow (\nu_1 = y \Rightarrow \nu_2 \geq y))
\]
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 uniqueness 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:

\[
(F(P) \Rightarrow \bot) \land (\forall \vec{x}. G(P)(\vec{x}) \Rightarrow P(\vec{x}))
\]

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

\[
\exists y. \phi_0 \lor (P(\vec{x}_1) \land \phi_1) \lor \cdots \lor (P(\vec{x}_n) \land \phi_n)
\]

Here, \(\exists y\) binds all free variables except \(\vec{x}\).

**Example 4.1.** The following constraint \(C_{\text{sum}}\) is obtained from the sample program for \(\text{sum}\) in Section 2:

\[
C_{\text{sum}} := (F(P) \Rightarrow \bot) \land (\forall \nu_1, \nu_2. G(P)(\nu_1, \nu_2) \Rightarrow P(\nu_1, \nu_2))
\]

\[
F(P) := \exists \nu_1, \nu_2, y. P(\nu_1, \nu_2) \land \neg \phi_3
\]

\[
G(P)(\nu_1, \nu_2) := \exists \nu_1', \nu_2'. \phi_1 \lor (P(\nu_1', \nu_2') \land \phi_2)
\]

Here, \(\phi_1, \phi_2,\) and \(\phi_3\) are defined as follows:

\[
\begin{align*}
\phi_1 & := \nu_1 \leq 0 \land \nu_2 = 0, \\
\phi_2 & := \nu_1 > 0 \land \nu_1' = \nu_1 - 1 \land \nu_2 = \nu_1 + \nu_2', \\
\phi_3 & := y = \nu_1 \Rightarrow \nu_2 \geq y.
\end{align*}
\]

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:

\[
(F(P) \Rightarrow \bot) \land (\forall \vec{x}. G(P)(\vec{x}) \Rightarrow P(\vec{x}))
\]

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

\[
\text{Expand}(C, i) := (F(P^{(0)}) \Rightarrow \bot) \land (\forall \vec{x}. G(P^{(0)})(\vec{x}) \Rightarrow P^{(0)}(\vec{x})) \land \cdots \land (\forall \vec{x}. G(P^{(i)})(\vec{x}) \Rightarrow P^{(i)}(\vec{x}))
\]

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

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

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

4.2 Solving Expanded Constraints

The sub-procedure SolveExpanded checks whether an expanded constraint \(\text{Expand}(C, i)\) is satisfiable, and returns a solution of \(\text{Expand}(C, i)\) if it is the case. The satisfiability of \(\text{Expand}(C, i)\)
procedure SOLVE(C) :
1: for each i ≥ 0 :
2: let C' = Expand(C, i)
3: match SolveExpanded(C') with
4: Unsatisfiable \rightarrow abort
5: | Satisfiable(θ') \rightarrow
6: let \( P^{(i)} \mapsto λx.φ_j \mid j \in \{0, \ldots, i\} \) = θ'
7: for each k ∈ \{0, \ldots, i\} :
8: if φ_0 ∧ \cdots ∧ φ_{k-1} ⇒ φ_k then
9: return \{ P \mapsto λx.φ_0 ∧ \cdots ∧ φ_{k-1} \}

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

can be reduced to the validity of the formula \( F(G'(λx.⊥)) \) \( \Rightarrow \), where \( G'(p) \) is defined as follows:

\[
G'(p) = p, \quad G'(p) = G'^{-1}(G(p)).
\]

To see why the reduction is correct, suppose that Expand(C, i) is satisfiable. Namely, we have a substitution \( θ \) for the predicate variables \( P^{(0)}, \ldots, P^{(i)} \) in Expand(C, i) such that \( F(θP^{(i)}) \Rightarrow \perp \) and \( G(θP^{(i)}) \Rightarrow Φ^{(i)}(λx.⊥) \) hold for all \( j \in \{0, \ldots, i\} \).

Then, by the monotonicity of \( G \), we get:

\[
\perp \Leftrightarrow F(θP^{(0)}) \Leftrightarrow F(G(θP^{(1)})) \Leftrightarrow F(λx.φ_1) \Leftrightarrow F(G'(θP^{(1)})) \Leftrightarrow F(G'(λx.⊥))
\]

Conversely, if \( F(G'(λx.⊥)) \Rightarrow \perp \) holds, then the following substitution satisfies Expand(C, i):

\[
\{ P^{(i)} \mapsto G'^{(i-1)}(λx.⊥) \}_{j=0}^{i}
\]

The formula \( F(G'(λx.⊥)) \) can be always transformed to a formula of the form \( ∃x.φ \) (recall the form of \( F(P) \) and \( G(P)(λx.φ) \) discussed in Section 4.1). We can check the validity of \( (∃x.φ) \Rightarrow \perp \) by using existing theorems provers including interpolating provers.

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

**Definition 4.2 (interpolants [12]).** Given a pair of predicates \( (λx.φ_1, λx.φ_2) \) such that \( φ_1 \) implies \( φ_2 \) and \( FV(φ_1) \cap FV(φ_2) \subseteq \{ x \} \), we call \( λx.φ \) an interpolant of the pair if

- \( φ_1 \) implies \( φ \),
- \( φ \) implies \( φ_2 \), and
- \( FV(φ) \subseteq FV(φ_1) \cap FV(φ_2) \).

Here, we say \( φ_1 \) implies \( φ_2 \) when \( ∀Y.φ_1 \Rightarrow φ_2 \), where \( Y = FV(φ_1) \cup FV(φ_2) \).

An interpolant \( φ_1 \) of \( φ_2 \) 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, \( λx.λy.x = y \) is an interpolant of the pair \( (λx.λy.x = z \land y = z, λx.λy.x = 0 \Rightarrow y = 0) \). In general, there may be more than one interpolant of given two formulas. In the example, \( λx.λy.x = y \) is also an interpolant.

We obtain a substitution for each predicate variable \( P^{(0)}, \ldots, P^{(i)} \) in 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 Expand(C, i) to that of the following constraint \( C_0 \) that contains only the predicate variable \( P^{(i)} \):

\[
(∀x. G'(λx.⊥))(x) \Rightarrow P^{(i)}(x) \land (F(P^{(i)}) \Rightarrow ⊥)
\]

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 \( G(P)(λx.φ) \) discussed in Section 4.1):

\[
(∀x. (∃y. φ) \Rightarrow P(x)) \land (∀x. (∃z. φ \land (P(x) \land φ) \land \cdots \land (P(z) \land φ)) \Rightarrow φ')
\]

We can transform the second line to the following one:

\[
(∀x. (∃y. φ) \Rightarrow φ') \land (∀x. (∃z. φ \land (P(x) \land φ) \land \cdots \land (P(z) \land φ)) \Rightarrow φ')
\]

Therefore, we can obtain \( P(x) \equiv φ' \land \cdots \land P(x) \equiv φ' \) by computing interpolants \( λx.φ_k \) of the pairs \( λx.φ, λx.φ_k \) for all \( k \in \{1, \ldots, n\} \) with an interpolating prover.

Similarly, for each \( j \in \{1, \ldots, i\} \), given solutions \( P^{(0)}(x) \equiv P^{(1)}(x) \equiv \cdots \equiv P^{(i-1)}(x) \equiv P^{(i)}(x) \equiv \perp \) to \( C_0, C_1, \ldots, C_{i-1} \) respectively, the satisfiability of Expand(C, i) can be reduced to that of the following constraint \( C_j \) that contains only the predicate variable \( P^{(i)} \):

\[
(∀x. G^{(i-1)}(λx.⊥))(x) \Rightarrow P^{(j)}(x)) \land (∀x. G^{(j)}(P^{(j)})(x) \Rightarrow φ(j-1))
\]

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

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

\[
\text{Expand}(C_{saw}, 1) := \ (F(P^{(0)}) \Rightarrow ⊥) \land \forall ν_1, ν_2.(G(ν_1))(ν_1, ν_2) \Rightarrow P^{(0)}(ν_1, ν_2))
\]

Here, \( F(P) \) and \( G(P)(ν_1, ν_2) \) are defined in Example 4.1. We find a solution of \( \text{Expand}(C_{saw}, 1) \) in this example. \( P^{(0)} \) is obtained as a solution of the following constraint:

\[
(∀ν_1, ν_2.G(λν_1, ν_2, ⊥))(ν_1, ν_2) ⇒ P^{(0)}(ν_1, ν_2)) \land \neg(F(P^{(0)}) ⇒ ⊥)
\]

This is equivalent to the following constraint:

\[
(∀ν_1, ν_2.φ_1 ⇒ P^{(0)}(ν_1, ν_2)) \land (∀ν_1, ν_2, y.P^{(0)}(ν_1, ν_2) ⇒ φ_3)
\]

In this example, we can obtain \( P^{(0)}(ν_1, ν_2) \equiv ν_2 ≥ ν_1 \) as an interpolant of \( (λν_1, ν_2, φ_1, λν_1, ν_2, φ_3) \). Then, \( P^{(1)} \) is obtained as a solution of the following constraint:

\[
(∀ν_1, ν_2.G(λν_1, ν_2, ⊥))(ν_1, ν_2) ⇒ P^{(1)}(ν_1, ν_2)) \land \neg(∀ν_1, ν_2,G(ν_1, ν_2) ⇒ ν_2 ≥ ν_1)
\]

This is equivalent to the following constraint:

\[
(∀ν_1, ν_2.ν_2 ⇒ P^{(1)}(ν_1, ν_2)) \land (∀ν_1, ν_2, ν_1 ⇒ ν_2 ≥ ν_1) \land (∀ν_1, ν_2, ν_1, ν_2.P^{(1)}(ν_1, ν_2) ⇒ φ_2 ⇒ ν_2 ≥ ν_1)
\]

Thus, we can obtain \( P^{(1)}(ν_1, ν_2) \equiv ⊥ \) as an interpolant of \( (λν_1, ν_2, ⊥, λν_1, ν_2, φ_2 ⇒ ν_2 ≥ ν_1) \). As a result, we obtain the...
following solution $\theta_{\text{sun}}$ of Expand($C_{\text{sun}}$, 1):

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

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

### 4.3 Checking Genuineness of Candidate Solutions

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

$$\{ \{ P \mapsto \lambda \bar{x}, \phi_0 \land \cdots \land \phi_{k-1} \} | k \in \{0, \ldots, i\} \}$$

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

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

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

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

**Proof.** We have $F(\lambda \bar{x}, \phi_0) \vdash \bot$ and $G(\lambda \bar{x}, \phi_{j+1})(\bar{x}) \Rightarrow \phi_j$ for all $j \in \{0, \ldots, i-1\}$. Assume that $\phi_0 \land \cdots \land \phi_{k-1}$ implies $\phi_k$ for some $k \in \{0, \ldots, i\}$.

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

$$\begin{align*}
\bot \ &= \quad F(\lambda \bar{x}, \phi_0) \\
\ &= \quad F(\lambda \bar{x}, \phi_0 \land \cdots \land \phi_{k-1}) \quad \text{(by monotonicity of $F$)} \\
\ &= \quad F(\theta P)
\end{align*}$$

We can also show that:

$$\begin{align*}
\theta P(\bar{x}) &\equiv \phi_0 \land \cdots \land \phi_{k-1} \\
&\equiv G(\lambda \bar{x}, \phi_1)(\bar{x}) \land \cdots \land G(\lambda \bar{x}, \phi_k)(\bar{x}) \\
&\equiv G(\lambda \bar{x}, \phi_0 \land \cdots \land \phi_1)(\bar{x}) \quad \text{(by monotonicity of $G$)} \\
&\equiv G(\lambda \bar{x}, \phi_0 \land \cdots \land \phi_{k-1})(\bar{x}) \quad \text{(by the assumption)} \\
&\equiv G(\theta P)(\bar{x})
\end{align*}$$

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

$$\text{sum} : \{ \nu_1 : \text{int} \mapsto \{ \nu_2 : \text{int} | \nu_2 \geq \nu_1 \} \}$$

### 4.4 Properties of Constraint Solving Algorithm

#### Correctness

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

**Theorem 4.3 (Correctness).** (a) If SOLVE($C$) returns $\theta$, $\theta$ is a solution for $C$. (b) If 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 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 SOLVE($C$) always aborts in a finite time for any unsatisfiable constraint $C$. The condition guarantees that an expanded constraint 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:

$$\begin{align*}
(1) &\Rightarrow G(\lambda \bar{x}, \bot)(\bar{x}) \quad \text{and} \\
(2) &\Rightarrow F(\lambda \bar{x}, \bot), F(G(\lambda \bar{x}, \bot)), F(G(\lambda \bar{x}, \bot)), \ldots, F(G'(\lambda \bar{x}, \bot))
\end{align*}$$

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

$$\begin{align*}
(1) &\Rightarrow, G(\lambda \bar{x}, \bot)(\bar{x}), G^2(\lambda \bar{x}, \bot)(\bar{x}), \ldots, G^i(\lambda \bar{x}, \bot)(\bar{x}), \ldots \\
(2) &\Rightarrow F(\lambda \bar{x}, \bot), F(G(\lambda \bar{x}, \bot)), F(G(\lambda \bar{x}, \bot)), \ldots, F(G'(\lambda \bar{x}, \bot))
\end{align*}$$

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

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

$$\exists \phi_0 \land \cdots \land \phi_n$$

Thus, we have:

$$\begin{align*}
F(\lambda \bar{x}) \quad &\Rightarrow \quad G^i(\lambda \bar{x}, \bot)(\bar{x}) \quad \Rightarrow \quad F(\lambda \bar{x}, \bot) \\
F(\lambda \bar{x}, \bot) \\
&\Rightarrow \quad G^i(\lambda \bar{x}, \bot)(\bar{x}) \quad \Rightarrow \quad F(\lambda \bar{x}, \bot)
\end{align*}$$

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

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 $\phi_i \not\models \phi_j$ 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, $\text{SOLVE}(C)$ always returns a solution or aborts for any constraint $C$.

**Proof.** Suppose that $\text{Expand}(C, k)$ has a solution $\{P^{(j)} \to \exists x \phi_j | j \in \{0, \ldots, k\}\}$ for some $k \geq 0$ and $\phi_0 \land \cdots \land \phi_{k-1}$ does not imply $\phi_j$ for any $j \in \{0, \ldots, k\}$. Then, we have a finite descending chain $T, \phi_0, \phi_1, \ldots, \phi_{k-1}$ with the length $k+2$. This is a contradiction. Thus, either $\text{Expand}(C, k)$ does not have a solution for any $k$ or $\phi_0 \land \cdots \land \phi_{k-1}$ implies $\phi_j$ for some $j \in \{0, \ldots, k\}$. Consequently, $\text{SOLVE}(C)$ aborts (in the former case) or returns a solution (in the latter case).

For example, given a set of $n$-predicates, let us consider a theory whose formula is $\bot$ 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 $\text{SOLVE}$ 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, 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 interpreted 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 OCaml.

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). 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 shows 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 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:

\[
\begin{align*}
\text{elem} & : \forall \alpha, \nu_1 : \alpha \to \{\nu_2 : \text{int} | 0 \leq \nu_2 < \text{size}(\nu_1)\} \to \alpha \\
\text{size} & : \forall \alpha, \nu_1 : \alpha \to \{\nu_2 : \text{int} | \nu_2 = \text{size}(\nu_1)\}
\end{align*}
\]

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

\[
\begin{align*}
\text{bs_aux} & : \text{int} \to \text{vec} : \text{int array} \to \{l : \text{int} | 0 \leq l\} \to \{u : \text{int} | u < \text{size}(\text{vec})\} \to \text{int option} \\
\text{bsearch} & : \text{int array} \to \text{int option}
\end{align*}
\]

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 bs_aux 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:

\[
-1 = l \leq m < 0 \leq u
\]

Table 1. Experimental Results for Array Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcopy</td>
<td>15</td>
<td>0.077</td>
</tr>
<tr>
<td>dotprod</td>
<td>17</td>
<td>0.056</td>
</tr>
<tr>
<td>bsearch</td>
<td>24</td>
<td>0.164</td>
</tr>
<tr>
<td>hanoi</td>
<td>90</td>
<td>1.359</td>
</tr>
<tr>
<td>queens</td>
<td>92</td>
<td>18.885</td>
</tr>
<tr>
<td>bcopy_bug</td>
<td>15</td>
<td>0.361</td>
</tr>
<tr>
<td>dotprod_bug</td>
<td>17</td>
<td>0.041</td>
</tr>
<tr>
<td>bsearch_bug</td>
<td>24</td>
<td>0.200</td>
</tr>
<tr>
<td>hanoi_bug</td>
<td>90</td>
<td>0.296</td>
</tr>
<tr>
<td>queens_bug</td>
<td>92</td>
<td>0.322</td>
</tr>
</tbody>
</table>

\footnote{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.}

\footnote{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.}
Similarly, our system automatically inferred the following types for \texttt{olist}, which represents increasing lists on integers. We declared the empty list, the head of the ordered list \( \nu \), the first and second elements of the tuple \( \nu \) respectively. The precondition \( \nu_2 = \text{nil} \lor \nu.1 \leq \text{hd}(\nu.2) \) of \texttt{OCons} ensures that the constructed list is increasing. Note that the definition of \texttt{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{verbatim}
ONil : \{ \nu : olist | \nu = \text{nil} \}
OCons : \{ \nu_1 : \text{int} \times olist | \nu_2 = \text{nil} \lor \nu.1 \leq \text{hd}(\nu.2) \}
\rightarrow \{ \nu_2 : olist | \nu_2 \neq \text{nil} \land \text{hd}(\nu_2) = \nu.1 \}
\end{verbatim}

Here, \texttt{nil}, \texttt{hd}(\nu), \nu.1, and \nu.2 denote the empty list, the head of the ordered list \( \nu \), the first and second elements of the tuple \( \nu \) respectively. The precondition \( \nu_2 = \text{nil} \lor \nu.1 \leq \text{hd}(\nu.2) \) of \texttt{OCons} ensures that the constructed list is increasing. Note that the definition of \texttt{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{verbatim}
insert : \{ \nu_1 : \text{int} \rightarrow \nu_2 : olist \rightarrow
\{ \nu_3 : olist | \nu_2 \neq \text{nil} \land \text{hd}(\nu_2) \leq \text{hd}(\nu_3) \lor 
\nu_1 \leq \text{hd}(\nu_3) \}\}
\end{verbatim}

and

\begin{verbatim}
isort : \{ \text{int list} \rightarrow olist \}
\end{verbatim}

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

\begin{verbatim}
merge : \{ \nu_1 : olist \rightarrow \nu_2 : olist \rightarrow
\{ \nu_3 : olist | 
\nu_1 = \nu_2 = \text{nil} \lor
\nu_1 = \text{nil} \land \nu_2 = \nu_3 \neq \text{nil} \lor
\nu_1 = \nu_3 \neq \text{nil} \land \nu_2 = \text{nil} \lor
\nu_1 \neq \text{nil} \land \nu_2 \neq \text{nil} \land \text{hd}(\nu_1) \leq \text{hd}(\nu_3) \lor
\nu_1 \neq \text{nil} \land \nu_2 \neq \text{nil} \land \text{hd}(\nu_2) \leq \text{hd}(\nu_3) \}\}
\end{verbatim}

\begin{verbatim}
initList : \{ \text{int list} \rightarrow olist \}
\end{verbatim}

\begin{verbatim}
mergeList : \{ olist list \rightarrow olist list \}
\end{verbatim}

\begin{verbatim}
mergeAll : \{ olist list \rightarrow olist \}
\end{verbatim}

\begin{verbatim}
mergesort : \{ \text{int list} \rightarrow olist \}
\end{verbatim}

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.

\textbf{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 \texttt{list} and \texttt{olist} respectively of the same data structure. However, even if the same set of constructors was used for \texttt{list} and \texttt{olist}, we believe that we can select an appropriate refinement type that conforms to the context for each occurrence of the constructors by using local type inference [18, 24].

\section{Related Work}

\subsection{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.

\subsection{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.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
Program & Lines & Time (sec.) \\
\hline
isort & 21 & 0.242 \\
mergesort & 66 & 10.113 \\
\hline
\end{tabular}
\caption{Experimental Results for Sorting Programs}
\end{table}

The counter-example means that \texttt{bs} aux can be called with, for example, the arguments \( l = -1 \) and \( u = 1 \), and then \texttt{bs} aux causes an array bounds error. In fact, with the arguments, the then-branch is taken in \texttt{bs} aux since \( l \leq u \) holds, \( m \) is bound to \( -1 \), and hence \texttt{elem vec m} fails.

The verification results are listed in Table 2.

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

\begin{verbatim}
ONil : \{ \nu : olist | \nu = \text{nil} \}
OCons : \{ \nu_1 : \text{int} \times olist | \nu_2 = \text{nil} \lor \nu.1 \leq \text{hd}(\nu.2) \}
\rightarrow \{ \nu_2 : olist | \nu_2 \neq \text{nil} \land \text{hd}(\nu_2) = \nu.1 \}
\end{verbatim}

\begin{verbatim}
let rec isort xs = match xs with
\{ x:olist | x = \text{nil} \rightarrow \text{ONil} \}
| OCons: \{ x1: \text{int} \times olist | \text{snd}(x) = \text{nil} \lor 
\text{fst}(x) \neq \text{hd}(\text{snd}(x)) \rightarrow
\{ x2: olist | x2 = \text{nil} \lor
\text{hd}(x2) = \text{fst}(\text{snd}(x)) \}
\}
\end{verbatim}

\begin{verbatim}
let rec insert x xs = match xs with
\text{ONil} \rightarrow OCons(x, \text{ONil})
| OCons(y, ys) \rightarrow
\{ if x <= y then OCons(x, OCons(y, ys)) else OCons(y, insert x ys) \}
\end{verbatim}

\begin{verbatim}
let _ = isort xs
\end{verbatim}
type 'a list = 
| Nil: 'a list 
| Cons: 'a * 'a list -> 'a list 

Let's use the mergesort function as an example:

```
let _ = mergesort xs
```

The mergesort function is defined as follows:

```
let mergesort l = mergeAll (initList l)
```

The `mergeAll` function is recursively defined as:

```
let rec mergeAll ls = match ls with 
  | (Cons(x1, Nil)) -> Nil 
  | Nil -> Nil 
  | Cons(x1, l1) -> Cons(OCons(x1, ONil), Nil)
```

Similarly, the `initList` function is defined as:

```
let rec initList xs = match xs with 
  | Nil -> Nil 
  | Cons(x1, x1) -> (match x1 with 
      | Nil -> Cons(x1, Nil)
```

The `merge` function is defined as:

```
let merge xs ys = match xs with 
  | Nil -> ys 
  | Cons(y, ys') -> 
      if x <= y then Cons(x, merge xs' ys) 
      else Cons(y, merge xs ys')
```

And the `mergeList` function is defined recursively as:

```
let rec mergeList ls = match ls with 
  | Cons(l1, ls') -> Cons(merge l1 l2, mergeList ls'')
```

The `mergeSort` function is defined similarly:

```
let mergeSort l = mergeAll (initList l)
```

### 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


procedure MSOLVE(C):

1: $\Pi \leftarrow \{\}$
2: while true do
3: let $C' = \text{Expand}(C, \Pi)$
4: match $\text{SolveExpanded}(C')$ with
5: Unsatisfiable $\rightarrow$ abort
6: $\Pi' \leftarrow \{\}$
7: while true do
8: let $\{P_i \mapsto \lambda x_i.\phi^i_n \mid \pi \in \Pi, i \in \{1, \ldots, m\}\}$ = $\theta'$
9: $\Pi' \leftarrow \{\}$
10: let $\Pi = \Pi' \setminus \text{Leaves}(\Pi', i)$
11: let $\theta_{\Pi'} = \{P_i \mapsto \lambda x_i.\phi^i_{\Pi'} \mid i \in \{1, \ldots, m\}\}$
12: if $\theta_{\Pi'} P_i(x_i)$ does not imply $\phi^i_n$ for some
13: if $\pi \cdot i \in \Pi$ then
14: $\Pi' \leftarrow \Pi' \cup \{\pi \cdot i\}$
15: else
16: $\Pi \leftarrow \Pi \cup \{\pi \cdot i\}$
17: break the inner loop
18: else
19: return $\theta_{\Pi'}$

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


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 $\theta'$ of an expanded constraint $\text{Expand}(C, \Pi)$, 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:

\[(\forall x_1, G_1(P_1, \ldots, P_m)(x_1) \Rightarrow P_1(x_1)) \land \cdots \land (\forall x_m, G_m(P_1, \ldots, P_m)(x_m) \Rightarrow P_m(x_m))\]

Here, \(P_1, \ldots, P_m\) are predicate variables, and \(F(P_1, \ldots, P_m)\) and \(G_1(P_1, \ldots, P_m)(x)\) are of the following form:

\[\exists y_j \left( (P_{1,1}(x_{1,1})) \land \cdots \land (P_{1,i}(x_{1,i})) \land (P_{n,1}(x_{n,1})) \land \cdots \land (P_{n,k}(x_{n,k})) \land y_1 \right) \land \cdots \land (\exists x_j, G_{i}(P_1, \ldots, P_m)(x_j) \Rightarrow P_i(x_j))\]

Here, \(P_{j,k} \in \{P_1, \ldots, P_m\}\) for all \(j, k\) and \(\exists y_j\) binds all free variables except \(x\).

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:

\[(F(P_1, \ldots, P_m) \Rightarrow \bot) \land (\forall x_1, G_1(P_1, \ldots, P_m)(x_1) \Rightarrow P_1(x_1)) \land \cdots \land (\forall x_m, G_m(P_1, \ldots, P_m)(x_m) \Rightarrow P_m(x_m))\]

Let \(X^*\) denote the set of sequences of the elements in \(X\). We write \(\epsilon\) for the empty sequence. For \(x_1, x_2 \in X^*\), we write \(x_1 \sqsubset x_2\) 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 \(x_1 \sqsubset x_2, x_2 \in Y\) implies \(x_1 \in Y\). For each prefix-closed and non-empty subset \(\Pi\) of \([1, \ldots, m]^*,\) we define an expanded constraint \(\text{Expand}(C, \Pi)\) with the predicate variables \(\{P_1^{\pi}, \ldots, P_m^{\pi}\} \in \Pi\) as follows:

\[\text{Expand}(C, \Pi) := \bigwedge_{\pi \in \Pi} \text{Expand}(C, \pi)\]

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

\[\text{Expand}(C, \pi) := F(P_1^{\pi}, \ldots, P_m^{\pi}) \Rightarrow \bot\]

\[\text{Expand}(C, \pi) := \forall x_1, G_1(P_1^{\pi} \ldots, P_m^{\pi})(x_1) \Rightarrow P_1^{\pi}(x_1)\]

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

**Lemma A.1.** For any constraint \(C\) and prefix-closed and non-empty subset \(\Pi\) of \([1, \ldots, m]^*\), \(\text{Expand}(C, \Pi)\) has a solution if \(C\) has a solution.

**Proof.** Solutions for \(P_1, \ldots, P_m\) in \(C\) are solutions for \(P_1^{\pi}, \ldots, P_m^{\pi}\) in \(\text{Expand}(C, \Pi)\) for all \(\Pi\).

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

\[(\forall y_i, \text{FV}(\phi_1), \phi_1 \Rightarrow Q_1(y_i)) \land \cdots \land (\forall y_n, \text{FV}(\phi_n), \phi_n \Rightarrow Q_n(y_n)) \land (\forall y_1, \ldots, y_n, \text{FV}(\phi), Q_1(y_1) \land \cdots \land Q_n(y_n) \Rightarrow \phi)\]

For each \(i = n, \ldots, 2, 1\), we can iteratively compute a solution \(\lambda y_i, \phi_i\) for \(Q_i\) as an interpolant of \((\lambda y_i, \phi_i, \lambda y_{i+1}, \phi_{i+1}) \land \cdots \land \lambda y_n, \phi_n)\)

**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 \(Leaves(\Pi, i)\) of \(\Pi\) by \(|\pi| \in \Pi \mid \pi \cdot i \not\subseteq \pi' \text{ for any } \pi' \in \Pi\). Suppose that an expanded constraint \(\text{Expand}(C, \Pi)\) has a solution \(\{P_1^{\pi} \leftrightarrow \lambda x_1, \phi_1^{\pi}\} \in \Pi, i \in \{1, \ldots, m\}\). Let \(\theta_{\Pi} = \{P_1 \mapsto \lambda x_1, \forall x \in \Pi', \exists x \in \Pi', \phi_1^{\pi} \mid i \in \{1, \ldots, m\}\}. \text{If there exists a prefix-closed and non-empty subset } \Pi' \text{ of } \Pi \text{ such that } \theta_{\Pi'}(P_1(x)) \text{ implies } \phi_1^{\pi} \text{ for all } i \in \{1, \ldots, m\} \text{ and } \pi \in \text{Leaves}(\Pi')(i), \text{ then } \theta_{\Pi'} \text{ is a solution of } C.\]

**Proof.** For all \(i \in \{1, \ldots, m\}\) and \(\pi \in \Pi\), we have:

\[F(\lambda x, \phi_1^{\pi}, \ldots, \lambda x_m, \phi_m^{\pi}) \Rightarrow \bot, \quad G_i(\lambda x, \phi_1^{\pi}, \ldots, \lambda x_m, \phi_m^{\pi})(x_i) \Rightarrow \phi_1^{\pi}(x_i)\]

Assume that there exists a prefix-closed and non-empty subset \(\Pi'\) of \(\Pi\) such that \(\theta_{\Pi'}(P_1(x)) \text{ implies } \phi_1^{\pi}\) for all \(i \in \{1, \ldots, m\}\) and \(\pi \in \text{Leaves}(\Pi')(i)\).

- If \(\Pi' = \{\}\), we get \(\theta_{\Pi'} P_i = \lambda x, T\), and \(\phi_i^{\pi} \equiv T\) by the assumption. Thus, we have \(F(\theta_{\Pi'} P_1, \ldots, \theta_{\Pi'} P_m) \Rightarrow \bot\) and \(G_i(\theta_{\Pi'} P_1, \ldots, \theta_{\Pi'} P_m)(x_i) \Rightarrow \phi_1^{\pi}(x_i)\).
- Otherwise, we get:

\[F(\theta_{\Pi'} P_1, \ldots, \theta_{\Pi'} P_m) \Rightarrow \phi_1^{\pi}(x_i)\]

(by monotonicity of \(F\))

We can also show that:

\[\forall x_i, G_i(\lambda x, \phi_1^{\pi}, \ldots, \lambda x_m, \phi_m^{\pi})(x_i) \Rightarrow \phi_1^{\pi}(x_i)\]

(by monotonicity of \(G\))

\[\forall x_i, G_i(\theta_{\Pi'} P_1, \ldots, \theta_{\Pi'} P_m)(x_i) \Rightarrow \phi_1^{\pi}(x_i)\]

(by the assumption)

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

**Theorem A.3 (Correctness).** (a) If MSOLVE(C) returns \(\theta, \theta\) is a solution of \(C\). (b) If MSOLVE(C) aborts, \(C\) is not satisfiable.

**B. Optimizations**

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