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) [28], 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 [28]. 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:

\[
\text{let inc} \ x = x + 1 \\
\text{let } y = \text{assert} \ (\text{inc} \ y >= y)
\]

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

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

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:

\[
\forall \nu_1, \nu_2, (\nu_2 = \nu_1 + 1 \Rightarrow P(\nu_1, \nu_2)), \\
\forall \nu_1, \nu_2, (P(\nu_1, \nu_2) \Rightarrow (\nu_1 = y \Rightarrow \nu_2 >= y)).
\]

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, \( P(\nu_1, \nu_2) \equiv \nu_2 >= \nu_1 \) 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 \( \nu_1 = y \Rightarrow \nu_2 >= y \) 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 \( \mathtt{inc} \) 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 \( P(v_1, v_2) \equiv v_2 \geq v_1 \) 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, 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:

```ml
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:

```ml
sum : (\nu_1 : \text{int} \to \{\nu_2 : \text{int} | P(\nu_1, \nu_2)\})
```

Here, \( P(\nu_1, \nu_2) \) 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 \([21]\):

\[
C_1 := \forall \nu_1, \nu_2, \nu_1', \nu_2'. (P(\nu_1', \nu_2') \land \phi_3) \Rightarrow P(\nu_1, \nu_2),
C_2 := \forall \nu_1, \nu_2, y. (P(\nu_1, \nu_2) \Rightarrow \phi_3).
\]

Here, \( \phi_1, \phi_2, \) and \( \phi_3 \) are given by:

\[
\phi_1 := \nu_1 \leq 0 \land \nu_2 = 0,
\phi_2 := \nu_1 > 0 \land \nu_1' = \nu_1 - 1 \land \nu_2 = \nu_1 + \nu_2',
\phi_3 := y = \nu_1 \Rightarrow \nu_2 \geq y.
\]

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 \( \phi_1 \) and \( P(\nu_1', \nu_2') \land \phi_2 \) 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 \( \text{inc} \) 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 \( \theta \) of predicates for the predicate variables is a solution of \( C \) if \( \theta C \) 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:

\[
C_3 := \phi_1 \lor (P_1(\nu_1', \nu_2') \land \phi_2) \Rightarrow P_0(\nu_1, \nu_2),
C_4 := P_0(\nu_1, \nu_2) \Rightarrow \phi_3.
\]

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 \( P_1(\nu_1', \nu_2') \) in \( C_3 \) with the inconsistency \( \bot \) without affecting the satisfiability as follows:

\[
\phi_1 \lor (\bot \land \phi_2) \Rightarrow P_0(\nu_1, \nu_2).
\]

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

\[
C_3' := \phi_1 \lor (P_1(\nu_1', \nu_2') \land \phi_2) \Rightarrow \psi_0,
C_4' := \psi_0 \Rightarrow \phi_3.
\]

The constraint \( C_3' \land C_4' \) can be simplified to:

\[
P_1(\nu_1', \nu_2') \Rightarrow (\phi_2 \Rightarrow \psi_0)
\]

by using the fact that \( \psi_0 \) is an interpolant of \( \phi_1 \lor (\bot \land \phi_2) \) \((\equiv \phi_1)\) and \( \phi_3 \). Note that \( P_1(\nu_1', \nu_2') \) can then be obtained as an interpolant of \( \bot \) 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 \( C_3 \land C_4 \), i.e., the following formulas \( \phi_0 \) and \( \phi_B \) hold:

\[
\phi_A := \phi_1 \lor (\psi_1 \land \phi_2) \Rightarrow \psi_0,
\phi_B := \psi_0 \Rightarrow \phi_3.
\]

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 \( C_1 \land C_2 \), and hence the type inference succeeds: \( P(\nu_1, \nu_2) \equiv \psi_0 \) satisfies \( C_1 \) because \( \phi_1 \lor (\psi_0 \land \phi_2) \) implies \( \phi_1 \lor (\psi_1 \land \phi_2) \) and \( \phi_0 \) holds, and satisfies \( C_2 \) because \( \phi_B \) holds. Similarly, if another condition \( T \Rightarrow \psi_0 \) holds, then \( P(\nu_1, \nu_2) \equiv T \) is a solution of the original constraint \( C_1 \land C_2 \). Thus, we call \( P(\nu_1, \nu_2) \equiv \psi_0 \) and \( P(\nu_1, \nu_2) \equiv T \) candidate solutions of \( C_1 \land C_2 \), and check their genuineness by using the conditions \( \psi_0 \Rightarrow \psi_1 \) and \( T \Rightarrow \psi_0 \) 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:

\[
(\phi_1 \lor (P_1(\nu_1, \nu_2) \land \phi_2)) \land \\
(\phi_1 \lor (P_2(\nu_1, \nu_2) \land \phi_2)) \land \\
(P_0(\nu_1, \nu_2) \Rightarrow \phi_3) .
\]

We again (i) solve the expanded constraint, (ii) compute candidate solutions of the original constraint, and 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 $C_3$:

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

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

\[
\phi_1' := \nu_1 \leq 1 \land \nu_2 = 0,
\]

\[
\phi_2' := \nu_1 > 1 \land \nu_1' = \nu_1 - 1 \land \nu_2 = \nu_1 + \nu_2' .
\]

Then, an interpolant of $\phi_1' \lor (\top \land \phi_2') \land \phi_3(\nu_1 \Rightarrow \nu_2 \geq y)$ 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 $\text{fail}$ 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 Appendix C.

3.1 Syntax

The syntax of expressions is defined as follows:

\[
e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \text{fix } x.e \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \mid \text{fail}
\]

Expressions:

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

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:

\[
\begin{align*}
\psi & ::= P(\bar{x}) \mid \phi \\
\Gamma & ::= \{ \nu : \text{int} \mid \psi \} \\
\text{Type Environments:} & \\
T & ::= \nu : T_1 \rightarrow T_2 \\
\end{align*}
\]

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 write $\text{FV}(\phi)$ 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 int. For example, \(\{\nu : \text{int} \mid \nu \geq 0\}\) 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, \(\nu_1 : \text{int} \rightarrow \nu_2 : \text{int} \rightarrow \{\nu_3 : \text{int} \mid \nu_3 = \nu_1 + \nu_2\}\) 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

\[
\begin{align*}
\Gamma & \vdash x : \{\nu : \text{int} \mid \nu = \alpha\} \\
(\text{T-INT}) & \quad \Gamma, x : T \vdash e : T \\
\Gamma, x : T \vdash e : T & \quad \text{if } x \not\in \text{FV}(T) \\
(\text{T-FIX}) & \quad \Gamma, x : T \vdash \text{fix} x.e : T \\
(\text{T-APPL}) & \quad \Gamma, \lambda x.e : (x : T \rightarrow T') \\
(\text{T-APP}) & \quad \Gamma, x : T \vdash e : T' \\
(\text{T-CON}) & \quad \Gamma, e : \mathcal{T}(\mathcal{S}(e)) \\
(\text{T-FAIL}) & \quad \Gamma, \text{fail} : T \\
(\text{T-TERM}) & \quad \Gamma, x : \text{int} \\
(\text{T-SUB}) & \quad \text{Lift}(\tau) \vdash \sigma \\
(\text{T-CASE}) & \quad \Gamma, \text{case} e \text{ of } x \rightarrow \sigma \\
(\text{T-COND}) & \quad \Gamma, x : T \vdash e : T \\
\end{align*}
\]

**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 \(\Gamma\). The typing rules are defined in Figure 2. The function \(\mathcal{T}(\mathcal{S}(\cdot))\) returns the dependendent type of \(e\). For example, if we have \(\mathcal{T}(\mathcal{S}(\text{int})) = \{\nu : \text{int} \mid \nu = \alpha\}\), then \(\mathcal{T}(\mathcal{S}(\text{int}))\) returns the type \(\text{int}\) under the type environment \(\Gamma\). The subtyping relation \(\Gamma \vdash T_1 <: T_2\) is defined as follows:

\[
\begin{align*}
\Gamma & \vdash \phi_1 \Rightarrow \phi_2 \\
(\text{S-INT}) & \quad \Gamma, \nu : \text{int} \vdash \phi_1 <: \nu : \text{int} \vdash \phi_2 \\
\Gamma & \vdash T_1 <: T_2 \\
(\text{S-FUN}) & \quad \Gamma, \nu : T_1 \vdash T_1' <: T_2' \\
\end{align*}
\]

**Example 3.1.** Let us consider the judgment \(\Gamma \vdash \text{inc} : T_2\), where \(\Gamma = \{\text{inc} : T_2, T_2 = \{\nu : \text{int} \mid P(\nu)\}\}\) and \(T_2 = \{\nu : \text{int} \mid \nu = \alpha\}\). An example derivation of the judgment is shown in Figure 3, where \(T_1 = \{\nu : \text{int} \mid \nu = \alpha\}\).

**Remark 1.** Unlike in the dependently-typed languages \(\lambda H\) [13, 21] and \(\lambda L\) [25], we separate the language for computation (i.e. expressions) and specification (i.e. formulas), to simplify constraint solving. Thus, the rules T-APPL 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, \(\nu : \text{int} \vdash x : \{\nu : \text{int} \mid \nu = \alpha\}\). Since we do not support refinements of function types (e.g. \(\{\nu : \text{int} \rightarrow \nu = \alpha\}\)), the type of the return value of a higher-order function that takes a function argument \(\nu\) cannot depend on the value of \(\nu\) in our type system. The same restriction is applied in Dependent ML [27], \(\lambda\nu\), and \(\lambda L\). 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 \(\text{Gen} \vdash \varphi : \mathcal{T}(\mathcal{S}(\varphi))\) returns the simple type of \(\varphi\), which can be inferred with the Hindley-Milner type inference algorithm. The function \(\text{Gen} \vdash \varphi : \mathcal{T}(\mathcal{S}(\varphi))\) is defined as follows:

\[
\begin{align*}
\varphi \vdash \text{inc} : \mathcal{T}(\mathcal{S}(\text{inc})) \\
\varphi \vdash \lambda x.\varphi : \mathcal{T}(\mathcal{S}(\lambda x.\varphi)) \\
\end{align*}
\]

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

\[
\begin{align*}
\varphi & \vdash \text{inc} : \mathcal{T}(\mathcal{S}(\text{inc})) \\
\varphi & \vdash \lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})) \\
\varphi & \vdash x : \text{int} \\
\end{align*}
\]

**Example 3.2.** Let us consider the program of \(\text{inc}\) in Section 1. The program can be encoded as follows in our language:

\[
\begin{align*}
\text{inc} & = (\lambda x.\text{inc} : \mathcal{T}(\mathcal{S}(\text{inc}))) \\
\lambda x.\text{int} & = (\lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int}))) \\
\end{align*}
\]

Here, \(\text{inc} = (\lambda x.\text{int})\). The Hindley-Milner algorithm infers the type \(\text{int}\) for \(\text{inc}\), and then the constraint generation for \(\varphi\) proceeds as follows:

\[
\begin{align*}
\varphi & \vdash \text{inc} : \mathcal{T}(\mathcal{S}(\text{inc})) \\
\varphi & \vdash \lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})) \\
\varphi & \vdash x : \text{int} \\
\end{align*}
\]

**Example 3.3.** Let us consider the program of \(\text{inc}\) in Section 1. The program can be encoded as follows in our language:

\[
\begin{align*}
\text{inc} & = (\lambda x.\text{inc} : \mathcal{T}(\mathcal{S}(\text{inc}))) \\
\lambda x.\text{int} & = (\lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int}))) \\
\end{align*}
\]

Here, \(\varphi = (\lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})))\). The Hindley-Milner algorithm infers the type \(\text{int}\) for \(\text{inc}\), and then the constraint generation for \(\varphi\) proceeds as follows:

\[
\begin{align*}
\varphi & \vdash \text{inc} : \mathcal{T}(\mathcal{S}(\text{inc})) \\
\varphi & \vdash \lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})) \\
\varphi & \vdash x : \text{int} \\
\end{align*}
\]

**Example 3.4.** Let us consider the program of \(\text{inc}\) in Section 1. The program can be encoded as follows in our language:

\[
\begin{align*}
\text{inc} & = (\lambda x.\text{inc} : \mathcal{T}(\mathcal{S}(\text{inc}))) \\
\lambda x.\text{int} & = (\lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int}))) \\
\end{align*}
\]

Here, \(\varphi = (\lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})))\). The Hindley-Milner algorithm infers the type \(\text{int}\) for \(\text{inc}\), and then the constraint generation for \(\varphi\) proceeds as follows:

\[
\begin{align*}
\varphi & \vdash \text{inc} : \mathcal{T}(\mathcal{S}(\text{inc})) \\
\varphi & \vdash \lambda x.\text{int} : \mathcal{T}(\mathcal{S}(\lambda x.\text{int})) \\
\varphi & \vdash x : \text{int} \\
\end{align*}
\]
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:

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

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

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

\[
\begin{align*}
C_{\text{sum}} & := (F(P) \Rightarrow \bot) \land (\forall \nu_1, \nu_2, G(P)(\nu_1, \nu_2) \Rightarrow P(\nu_1, \nu_2)) \\
F(P) & := \exists \nu_1, \nu_2, y, P(\nu_1, \nu_2) \land \neg \phi_3 \\
G(P)(\nu_1, \nu_2) & := \exists \nu'_1, \nu'_2, \phi_1 \lor (P(\nu'_1, \nu'_2) \land \phi_2)
\end{align*}
\]

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

\[
\begin{align*}
\phi_1 & := \nu_1 \leq 0 \land \nu_2 = 0, \\
\phi_2 & := \nu_1 > 0 \land \nu'_1 = \nu_1 - 1 \land \nu_2 = \nu_1 + \nu'_2, \\
\phi_3 & := y = \nu_1 \lor \nu_2 \geq y.
\end{align*}
\]

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 \(\{P(j)\mid 0 \leq j \leq i\}\) as follows:

\[
\text{Expand}(C, i) := (F(P(0)) \Rightarrow \bot) \land (\forall \vec{x}. G(P(1)(\vec{x}) \Rightarrow P(0)(\vec{x}))) \land \cdots \land (\forall \vec{x}. G(P(i)(\vec{x}) \Rightarrow P(i-1)(\vec{x})))
\]

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 \(\text{Expand}(C, i)\).

\(\square\)

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

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

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

$$G'(p) = p, \quad G'(x) = G'(x'; G(p)).$$

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

$$\bot \Leftarrow F(\theta P^{(0)}) \Leftarrow F(G(\theta P^{(1)})) \quad \ldots \quad \Leftarrow F(G'(\theta P^{(i)})) \Leftarrow F(G'(\lambda x.\bot))$$

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

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

The formula $F(G'(\lambda x.\bot))$ can be always transformed to a formula of the form $\exists \overline{x}.\phi$ (recall the form of $F(P)$ and $G(P)$ discussed in Section 4.1) by using existing theorem provers including interpolating provers.

We now present an algorithm for finding a solution of a satisfiable extended 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 $\{\lambda x.\phi_1, \lambda x.\phi_2\}$ such that $\phi_1$ implies $\phi_2$ and $\text{FV}(\phi_1) \cap \text{FV}(\phi_2) \subseteq \{\overline{x}\}$, we call $\lambda x.\phi$ an interpolant of the pair if

- $\phi_1$ implies $\phi$,
- $\phi$ implies $\phi_2$, and
- $\text{FV}(\phi) \subseteq \text{FV}(\phi_1) \cap \text{FV}(\phi_2)$.

Here, we say $\phi$ implies $\phi_2$ when $\forall \overline{y}.\phi_1 \Rightarrow \phi_2$, where $\overline{y} = \text{FV}(\phi_1) \cup \text{FV}(\phi_2)$.

An interpolant of $\phi_1$ and $\phi_2$ always exists if $\phi_1$ implies $\phi_2$, 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 $(\lambda x.\lambda y.x = z \land y = z, \lambda x.\lambda y.x = 0 \Rightarrow y = 0)$. In general, there may be more than one interpolant of given two formulas. In the example, $\lambda x.\lambda y.x = y$ 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 $P^{(0)}$:

$$\langle \forall \overline{x}.G'(\lambda x.\bot) \Rightarrow P^{(0)}(\overline{x}) \land \left( F(P^{(0)}) \Rightarrow \bot \right) \rangle$$

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 $G(P)$ discussed in Section 4.1):

$$\langle \forall \overline{x}.(\exists \overline{y}.\phi) \Rightarrow P(\overline{x}) \rangle \land \langle \forall \overline{x}.(\exists \overline{z}.\phi \lor (P(\overline{x}) \land \phi_1) \lor \cdots \lor (P(\overline{x_n}) \land \phi_n)) \Rightarrow \phi' \rangle$$

We can transform the second line to the following one:

$$\langle \forall \overline{x}.(\overline{x} \Rightarrow \phi) \rangle \land \langle \forall \overline{x}.(\overline{x} \Rightarrow \phi_1) \rangle \land \cdots \land \langle \forall \overline{x}.(\overline{x} \Rightarrow \phi_n) \rangle$$

Therefore, we can obtain $P(\overline{x}) \equiv \phi'_1 \land \cdots \land \phi'_n$ by computing interpolants $\lambda x.\phi'_k$ of the pairs $\lambda x.\phi_1, \lambda x.\phi_2, \ldots, \lambda x.\phi_k$ for all $k \in \{1, \ldots, n\}$ with an interpolating prover.

Similarly, for each $j \in \{1, \ldots, i\}$, given solutions $P^{(j)}(\overline{x}) \equiv \phi'_1 \land \cdots \land \phi'_i$ to $C_0, \ldots, C_{j-1}$ respectively, the satisfiability of Expand($C, i$) can be reduced to that of the following constraint $C_j$ that contains only the predicate variable $P^{(j)}$:

$$\langle \forall \overline{x}.(\exists \overline{y}.\phi) \Rightarrow P^{(j)}(\overline{x}) \rangle \land \langle \forall \overline{x}.G(\lambda x.P^{(j)})(\overline{x}) \Rightarrow \phi^{(j-1)} \rangle$$

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 $C_{\text{con}}$ in Example 4.1. The expanded constraint $\text{Expand}(C_{\text{con}}, 1)$ of $C_{\text{con}}$ on the new predicate variables $P^{(0)}$ and $P^{(1)}$ is as follows:

$$\text{Expand}(C_{\text{con}}, 1) := (F(P^{(0)}) \Rightarrow \bot) \land \forall \nu_1, \nu_2, (G(P^{(1)})(\nu_1, \nu_2) \Rightarrow P^{(0)}(\nu_1, \nu_2))$$

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

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

This is equivalent to the following constraint:

$$\langle \forall \nu_1, \nu_2, \phi_1 \Rightarrow P^{(0)}(\nu_1, \nu_2) \rangle \land \langle \forall \nu_1, \nu_2, \nu. P^{(0)}(\nu_1, \nu_2) \Rightarrow \phi_3 \rangle$$

In this example, we can obtain $P^{(0)}(\nu_1, \nu_2) \equiv \nu_2 \geq \nu_1$ as an interpolant of $\langle \lambda \nu_1, \nu_2.\phi_1, \lambda \nu_1, \nu_2.\phi_2 \rangle$. Then, $P^{(1)}$ is obtained as a solution of the following constraint:

$$\langle \forall \nu_1, \nu_2, G(\lambda \nu_1, \nu_2.\bot)(\nu_1, \nu_2) \Rightarrow P^{(1)}(\nu_1, \nu_2) \rangle \land \langle \forall \nu_1, \nu_2, G(P^{(1)})(\nu_1, \nu_2) \Rightarrow \nu_2 \geq \nu_1 \rangle$$

This is equivalent to the following constraint:

$$\langle \forall \nu_1, \nu_2, \bot \Rightarrow P^{(1)}(\nu_1, \nu_2) \rangle \land \langle \forall \nu_1, \nu_2, \phi_1 \Rightarrow \nu_2 \geq \nu_1 \rangle$$

$$\langle \forall \nu_1, \nu_2, \nu_1.\nu_2.\bot, P^{(1)}(\nu_1, \nu_2) \Rightarrow \phi_2 \Rightarrow \nu_2 \geq \nu_1 \rangle$$

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

$$
\{ P^{(0)} \mapsto \lambda \nu_1, \nu_2.2 \geq \nu_1, P^{(1)} \mapsto \lambda \nu_1, \nu_2. \}. 
$$

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

4.3 Checking Genuineness of Candidate Solutions

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

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

For each $k \in \{0, \ldots, i\}$, $\text{SOLVE}$ 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 $\text{Expand}(C, i)$ has a solution $\{ P^{(j)} \mapsto \lambda \bar{x}. \phi_j \mid j \in \{0, \ldots, i\} \}$. 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 $F(\lambda \bar{x}. \phi_0) \Rightarrow \bot$ and $G(\lambda \bar{x}. \phi_{j+1}) \Rightarrow \phi_j$ 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 $F(\theta P) = \top \land \bot$ and $G(\theta P(\bar{x})) = \theta P(\bar{x})$.
- Otherwise, we get:

  $\bot \leftarrow F(\lambda \bar{x}. \phi_0)$

  $\leftarrow F(\lambda \bar{x}. \phi_0 \land \cdots \land \phi_{k-1})$ (by monotonicity of $F$)

  $= F(\theta P)$

  We can also show that:

  $\theta P(\bar{x}) = \phi_0 \land \cdots \land \phi_{k-1}$

  $\leftarrow G(\lambda \bar{x}. \phi_1(\bar{x}) \land \cdots \land G(\lambda \bar{x}. \phi_k(\bar{x})$)

  $\leftarrow G(\lambda \bar{x}. \phi_0 \land \cdots \land \phi_k(\bar{x})$ (by monotonicity of $G$)

  $\leftarrow G(\lambda \bar{x}. \phi_0 \land \cdots \land \phi_{k-1}(\bar{x})$ (by the assumption)

  $= G(\theta P(\bar{x}))$

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

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

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 $\text{SOLVE}$:

**Theorem 4.3** (Correctness). (a) If $\text{SOLVE}(C)$ returns $\theta$, $\theta$ is a solution for $C$. (b) If $\text{SOLVE}(C)$ 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 $\text{SOLVE}$ does not terminate in all cases.

We separate the termination property of $\text{SOLVE}$ 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 $\text{SOLVE}(C)$ always aborts in a finite time for any unsatisfiable constraint $C$. The condition guarantees that an expanded constraint $\text{Expand}(C, i)$ always gets unsatisfiable for some $i \geq 0$. The following theorem formalizes the condition.

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

$$(F(P) \Rightarrow \bot) \land (\forall \bar{x}. G(P)(\bar{x}) \Rightarrow P(\bar{x}))$$

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

$$(1) \bot, G(\lambda \bar{x}. \bot)(\bar{x}), G^2(\lambda \bar{x}. \bot)(\bar{x}), \ldots, G^i(\lambda \bar{x}. \bot)(\bar{x}), \ldots$$

$$(2) F(\lambda \bar{x}. \bot), F(G(\lambda \bar{x}. \bot)), F(G^2(\lambda \bar{x}. \bot)), \ldots, F(G^i(\lambda \bar{x}. \bot)), \ldots$$

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 $\text{Expand}(C, i)$ is not satisfiable.

**Proof.** We prove the theorem by contraposition. We assume that $\text{Expand}(C, i)$ 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 $F(P)$ and $G(P)(\bar{x})$ are of the following form:

$$
\exists \bar{y}. \phi_0 \lor (P(\bar{x}) \land \phi_1) \lor \cdots \lor (P(\bar{x}_n) \land \phi_n).
$$

Thus, we have:

$$
F(\lambda \bar{x}. \bigcup_i G^i(\lambda \bar{x}. \bot)(\bar{x})) \equiv \bigcup_i F(G^i(\lambda \bar{x}. \bot)),$$

$$
G(\lambda \bar{x}. \bigcup_i G^i(\lambda \bar{x}. \bot)(\bar{x})) \equiv \bigcup_i G^{i+1}(\lambda \bar{x}. \bot)(\bar{x}) \equiv \bigcup_i G^i(\lambda \bar{x}. \bot)(\bar{x}).
$$

Since $F(G^i(\lambda \bar{x}. \bot)) \Rightarrow \bot$ holds for any $i \geq 0$, $\bot$ is an upper bound of the infinite sequence (2). Thus, we get $\bigcup_i F(G^i(\lambda \bar{x}. \bot)) \Rightarrow \bot$ because $\bigcup_i F(G^i(\lambda \bar{x}. \bot))$ is the least upper bound of (2).
let rec bs_aux key vec 1 u =
    if l <= u then
        let m = l + (u-1) / 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 1 (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, SOLVE(\( C \)) always returns a solution or aborts for any constraint \( C \).

**Proof.** Suppose that Expand(\( C,k \)) has a solution \( \{ P(i) \rightarrow \lambda x.\phi_i \mid j \in \{0, \ldots, k\} \} \) 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 \( \top, \phi_0, \phi_0 \land \phi_1, \ldots, \phi_0 \land \cdots \land \phi_{k-1} \) with the length \( k+2 \). This is a contradiction. Thus, either 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, 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 SOLVE as in Liquid Types.

### 5. Experiments

We have implemented a prototype type inference system according to the formalization in Appendix C, A, and B. 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 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 sources used in the experiments except for isort were originally written in 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.

### 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 desugarung 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 naïve implementation causes the size of input formulas to the interpolating prover to be large.

<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>0.061</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>

**Table 1. Experimental Results for Array Programs**

DML [27, 28]. 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 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.

---

4 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.

---
### 5.2 Verification of Orderedness for Sorting Algorithms

The source programs include the insertion sort algorithm (\texttt{isort}) and the merge sort: (\texttt{mergesort}) in Figure 8. The timing 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 dependent types of the constructors \texttt{ONil} and \texttt{OCons} for \texttt{olist} as follows:

\[
\begin{align*}
\texttt{ONil} : & \{ \nu : \texttt{olist} | \nu = \texttt{nil} \} \\
\texttt{OCons} : & \{ \nu : \texttt{int} \times \texttt{olist} | \nu.2 = \texttt{nil} \lor \nu.1 \leq \text{hd}(\nu.2) \} \\
& \rightarrow \{ \nu_2 : \texttt{olist} | \nu_2 = \texttt{nil} \land \text{hd}(\nu_2) = \nu.1 \}
\end{align*}
\]

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 = \texttt{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:

\[
\begin{align*}
\text{insert} : & (\nu_1 : \texttt{int} \rightarrow \nu_2 : \texttt{olist} \rightarrow \\
& \{ \nu_3 : \texttt{olist} | \\
& \quad \nu_2 = \texttt{nil} \land \text{hd}(\nu_2) \leq \text{hd}(\nu_3) \lor \\
& \quad \nu_1 \leq \text{hd}(\nu_3) \}) \\
\text{isort} : & (\texttt{int list} \rightarrow \texttt{olist})
\end{align*}
\]

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

\[
\begin{align*}
\text{merge} : & (\nu_1 : \texttt{olist} \rightarrow \nu_2 : \texttt{olist} \rightarrow \\
& \{ \nu_3 : \texttt{olist} | \\
& \quad \nu_1 = \nu_2 = \texttt{nil} \lor \\
& \quad \nu_1 = \texttt{nil} \land \nu_2 = \nu_3 \neq \texttt{nil} \lor \\
& \quad \nu_1 = \nu_3 \neq \texttt{nil} \land \nu_2 = \texttt{nil} \lor \\
& \quad \nu_1 \neq \texttt{nil} \land \nu_2 \neq \texttt{nil} \land \text{hd}(\nu_1) \leq \text{hd}(\nu_3) \lor \\
& \quad \nu_1 \neq \texttt{nil} \land \nu_2 \neq \texttt{nil} \land \text{hd}(\nu_2) \leq \text{hd}(\nu_3) \}) \\
\text{initList} : & (\texttt{int list} \rightarrow \texttt{olist}) \\
\text{mergeList} : & (\texttt{olist list} \rightarrow \texttt{olist list}) \\
\text{mergeAll} : & (\texttt{olist list} \rightarrow \texttt{olist}) \\
\text{mergesort} : & (\texttt{int list} \rightarrow \texttt{olist})
\end{align*}
\]

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, Cons}\} and \{\texttt{ONil, 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 is used for list and 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].

### 6. Related Work

#### 6.1 Dependent Typed Languages

Dependent types have been introduced to programming languages for verification of detailed specifications of programs [1, 2, 27, 28]. 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isort</td>
<td>21</td>
<td>0.242</td>
</tr>
<tr>
<td>mergesort</td>
<td>66</td>
<td>10.113</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isort</td>
<td>21</td>
<td>0.242</td>
</tr>
<tr>
<td>mergesort</td>
<td>66</td>
<td>10.113</td>
</tr>
</tbody>
</table>

Table 2. Experimental Results for Sorting Programs

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.
let _ = mergesort xs
| OCons: (x1: 'a * 'a list | x = nil)
| OCons(x, xs') ->
  | OCons(y, ys') ->
  | OCons(y, ys') ->
| Cons(l, initList xs2))
| Cons(x1, xs1) ->
  | Cons(x2, xs2) ->
  | Cons(l1, l2, mergeList ls''))
| Cons(1, ls') ->
  | Cons(2, ls'') ->
  | Cons(l, ls) ->
| mergeAll ls = match ls with
| mergeAll ls = match ls with
| mergeAll ls = match ls with
| merge = mergeAll (initList l)
| merge = mergeAll (initList l)
| mergesort l = mergeAll (initList l)

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 [28]. 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

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 θI′ (see the line 11) from the solution θ′ of an expanded constraint Expand(C, Π), 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 and
Appendix C can always be transformed to the following form:

\[
(F(P_{1}, \ldots, P_{m}) \Rightarrow \bot) \land \\
(\forall x, y, G_{1}(P_{1}, \ldots, P_{m})(x, y) \Rightarrow P_{1}(x, y)) \land \\
(\forall \lambda \in \Lambda, G_{m}(P_{1}, \ldots, P_{m})(\lambda) \Rightarrow P_{m}(\lambda))
\]

Here, \(P_{1}, \ldots, P_{m}\) are predicate variables, and \(F(P_{1}, \ldots, P_{m})\) and \(G_{i}(P_{1}, \ldots, P_{m})(\lambda)\) are of the following form:

\[
\exists y, (P_{1}(x_{1}, y_{1}) \land \cdots \land P_{i}(x_{i}, y_{i}) \land \phi_{i}) \lor \cdots \lor (P_{n}(x_{n}, y_{n}) \land \cdots \land P_{m}(x_{m}, y_{m}) \land \phi_{m})
\]

Here, \(F(k, \lambda) \in \{P_{1}, \ldots, P_{m}\}\) for all \(k, \lambda\) and \(\exists y\) binds all free variables except for \(x\).

**Example A.1.** Let us consider the following program:

\[
\text{izip } : \{v_{1} : \text{int } | P_{1}(v_{1}) \} \rightarrow \\
v_{2} : \{v_{2} : \text{int } | P_{2}(v_{1}, v_{2}) \} \rightarrow \text{int}
\]

Here, we assume that \(z \geq 0\). We prepare the type template:

\[
\text{izip } (v_{1} : \{v_{1} : \text{int } | P_{1}(v_{1}) \}) \rightarrow \\
v_{2} : \{v_{2} : \text{int } | P_{2}(v_{1}, v_{2}) \} \rightarrow \text{int}
\]

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 \(\Pi_{1}\) 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 \(P_{1}^{p_{1}}, \ldots, P_{m}^{p_{m}}\) in \(\text{Expand}(C, \Pi)\) for all \(\Pi\).

**Example A.2.** Let us consider the constraint \(C_{\text{izip}}\) in Example A.1. The expanded constraint \(\text{Expand}(C_{\text{izip}}, \{\epsilon, 1, 2\})\) of \(C_{\text{izip}}\) on the new predicate variables \(P_{1}, P_{2}, P_{1}^{\epsilon}, P_{2}^{\epsilon}, P_{1}^{1}, P_{2}^{1}\), and \(P_{2}^{2}\) is as follows:

\[
\text{Expand}(C_{\text{izip}}, \{\epsilon, 1, 2\}) = (F(P_{1}, P_{2}) \Rightarrow \bot) \land \\
(\forall v_{1}, (G_{1}(P_{1}, P_{2})(v_{1}) \Rightarrow P_{1}(v_{1})) \land \\
(\forall v_{2}, (G_{2}(P_{1}, P_{2})(v_{1}, v_{2}) \Rightarrow P_{2}(v_{1}, v_{2})))
\]

Here, \(F(P_{1}, P_{2})\) is defined in Appendix A.1.

**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 \(\text{Expand}(C, \epsilon)\) explained in Section 4.2. An additional technical requirement lies in solving constraints of the form:

\[
(\forall y, F_{1}(\phi_{1}, \phi_{2}) \Rightarrow Q_{1}(\phi_{2})) \land \\
(\forall y, F_{2}(\phi_{1}, \phi_{2}, \phi_{3}) \Rightarrow Q_{2}(\phi_{3}))
\]

For each \(i = m, \ldots, 2, 1\), we can iteratively compute a solution \(\phi_{3}^{i+1}, \phi_{2}^{i+1}, \phi_{1}^{i+1}\) for \(Q_{i}\) as an interpolant of \((\lambda y_{i}, \phi_{1}, \lambda y_{i}, \phi_{1} \land \cdots \land \phi_{i-1} \land \cdots \land \phi_{0}) \Rightarrow \phi\).

**Example A.3.** In this example, we consider the expanded constraint \(\text{Expand}(C_{\text{izip}}, \{\epsilon, 1, 2\})\) in Example A.2. A solution for \(P_{1}^{\epsilon}\) and \(P_{2}^{\epsilon}\) in \(\text{Expand}(C_{\text{izip}}, \{\epsilon, 1, 2\})\) is obtained as the one for the following constraint:

\[
(\forall v_{1}, G_{1}(\lambda v_{1}, \lambda v_{1}, \lambda v_{2}, \lambda v_{2})(v_{1}) \Rightarrow P_{1}^{\epsilon}(v_{1})) \land \\
(\forall v_{2}, (G_{2}(\lambda v_{1}, \lambda v_{2})(v_{1}, v_{2}) \Rightarrow P_{2}^{\epsilon}(v_{1}, v_{2})) \land \\
(F(P_{1}^{\epsilon}, P_{2}^{\epsilon}) \Rightarrow \bot)
\]

This can be reduced to the following constraint without \(P_{1}^{\epsilon}\):

\[
(\forall v_{2}, G_{2}(\lambda v_{1}, \lambda v_{2})(v_{1}, v_{2}) \Rightarrow P_{2}^{\epsilon}(v_{1}, v_{2})) \land \\
(F(G_{1}(\lambda v_{1}, \lambda v_{2}, \lambda v_{2}, \lambda v_{2}), P_{2}^{\epsilon}) \Rightarrow \bot)
\]

We can now obtain solutions for \(P_{1}^{\epsilon}\) and then \(P_{1}^{\epsilon}\) by computing interpolants. We then compute those for \(P_{1}^{\epsilon}, P_{2}^{\epsilon}, P_{1}^{2}, P_{2}^{2}\) similarly. As a result, we may obtain the following solution \(\theta_{\text{izip}}\) for \(\text{Expand}(C_{\text{izip}}, \{\epsilon, 1, 2\})\):

\[
\theta_{\text{izip}} = \{ P_{1}^{\epsilon} \leftarrow \lambda v_{1}, \nu_{1} \geq 0, P_{2}^{\epsilon} \leftarrow \lambda v_{1}, v_{2}, v_{1} = v_{2}, \\
P_{1}^{2} \leftarrow \lambda v_{1}, v_{1}, P_{2}^{2} \leftarrow \lambda v_{1}, v_{2}, P_{1}^{2} \leftarrow \lambda v_{1}, v_{1}, P_{2}^{2} \leftarrow \lambda v_{1}, v_{2}, \}
\]

**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 lemmas.

**Lemma A.2.** We define leaves \(\text{Leaves}(\Pi, i)\) of \(\Pi\) by \(\{\pi \in \Pi \mid \pi \cdot i \notin \pi \neq \pi^{'} \in \Pi\}\). Suppose that an expanded constraint \(\text{Expand}(C, \Pi)\) has a solution \(P_{i}^{\epsilon} \leftarrow \lambda x_{i}^{\epsilon}, \phi_{1}^{i} \mid \phi_{1} = 0 \in \Pi, i \in \{1, \ldots, m\}\). Let \(\theta_{i} = \{ P_{i} \leftarrow \lambda x_{i}^{\epsilon}, \nabla_{i, \Pi_{i}, i, i} \text{Leaves}(\Pi, i) \phi_{1}^{i} \mid i \in \{1, \ldots, m\}\}\). If there exists a prefix-closed and non-empty subset \(\Pi_{i}\) of \(\Pi\)
II' of II such that $\theta_II', P_i(x_i)$ implies $\phi^n_i$ for all $i \in \{1, \ldots, m\}$ and $\pi \in \text{Leaves}(\Pi', i)$, then $\theta_II'$ is a solution of $C$.

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

\[ F(\lambda x_i. \phi^n_1, \ldots, \lambda x_m. \phi^n_m) \Rightarrow \perp, \]

\[ G_i(\lambda x_i. \phi^n_1, \ldots, \lambda x_m. \phi^n_m)(x_i) \Rightarrow \phi^n_i(x_i). \]

Assume that there exists a prefix-closed and non-empty subset II' of II such that $\theta_II', P_i(x_i)$ implies $\phi^n_i$ for all $i \in \{1, \ldots, m\}$ and $\pi \in \text{Leaves}(\Pi', i)$.

- If $I' = \{e\}$, we get $\theta_II', P_i = \lambda x_i. \top$, and $\phi^n_i \equiv \top$ by the assumption. Thus, we have $F(\theta_II', P_i, \ldots, \theta_II', P_m) \Rightarrow \perp$ and $G_i(\theta_II', P_1, \ldots, \theta_II', P_m)(x_i) \Rightarrow \theta_II', P_i(x_i)$.

- Otherwise, we get:

\[ \perp \Leftarrow F(\theta_II', P_1, \ldots, \theta_II', P_m) \text{ (by monotonicity of } F) \]

We can also show that:

\[ \theta_II', P_i(x_i) \Leftarrow G_i(\lambda x_i. \bigwedge_{\pi \in I'} \phi^n_1, \ldots, \lambda x_m. \bigwedge_{\pi \in I'} \phi^n_m) \]

(by monotonicity of $G$)

\[ \Leftarrow G_i(\theta_II', P_1, \ldots, \theta_II', P_m)(x_i) \]

(by the assumption)

\[ \square \]

**Example A.4.** In this example, we consider the expanded constraint $\text{Expand}([C_{\otimes}P_1 \{x_1, x_2\}])$ in Example A.2 and its solution $\theta_{\otimes}P_1$ in Example A.3. We regard $\theta_{\otimes}P_1$ as a candidate solution for $C_{\otimes}P_1$, and it is actually so because $\theta_{\otimes}P_1 P_2(x_1)$ implies $\theta_{\otimes}P_2 P_1(x_1)$ and $\theta_{\otimes}P_2 P_1(x_1, x_2)$ implies $\theta_{\otimes}P_1 P_2(x_1, x_2)$. We then obtain a genuine solution $\{P_1 \mapsto \theta_{\otimes}P_1, P_2 \mapsto \theta_{\otimes}P_1 P_2\}$ for $C_{\otimes}P_1$. Thus, we inferred the following dependent type function $\text{izip}:

\text{izip}(\nu_i : \{v_i : \text{int} \mid v_i \geq 0\} \rightarrow (v_i : \text{int} \mid v_i = v_2) \rightarrow \text{int})

**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 II 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 reconstruct 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 II constructed previously. After $\Pi'$ is updated to $\Pi' \cup \{\pi \cdot i\}$ in the line 15, we recheck the conditions on $\theta_{\Pi'I'}(\xi)$ 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 $G$, on predicates, and their arguments were fixed to $P_1, \ldots, P_m$. This may make the constraint solving inefficient for two reasons: (1) Even though some predicate variable $P_i$ may not actually occur in the definitions of the functions, the algorithm may wastefully expands $P_i$. (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:

\( (F(P) \Rightarrow \perp) \land (\forall x_i. G_1(\pi_1)(x_i) \Rightarrow P_1(x_i)) \land \cdots \land (\forall x_m. G_m(\pi_m)(x_m) \Rightarrow P_m(x_m)) \)

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

**C. Constraint Generation for Full Language**

In this section, we first introduce the higher-order functional language $\lambda\ell$ and its dependent type system. Then, we describe the constraint generation algorithm based on Knowles and Flanagan's work [21], and prove its soundness and completeness with respect to the type system.

**C.1 Syntax**

The syntax of expressions and types in $\lambda\ell$ is defined as follows:

\[
\begin{align*}
\text{expressions:} & \\
\text{variable} & := x \\
\text{constant} & := e \\
\text{tuple} & := (e_1, \ldots, e_m) \\
\text{abstraction} & := \lambda x.e \\
\text{application} & := e_1 e_2 \\
\text{fixed-point} & := \text{fix } x.e \\
\text{let-binding} & := \text{let } x = e_1 \text{ in } e_2 \\
\text{arity-0 constructor} & := K \\
\text{arity-1 constructor} & := K e \\
\text{pattern-match} & := \text{match } x \text{ with } \{p_i \mapsto e_i\}_{i=1}^m \\
\text{failure} & := \text{fail} \\
\text{values:} & \\
\text{pattern} & := e | (v_1, \ldots, v_m) | \lambda x.e \\
\text{pattern-match} & := K x \\
\text{failure} & := K v
\end{align*}
\]

\[
\begin{align*}
\text{types:} & \\
\text{boxed} & := P(x) \\
\text{boxed type} & := \alpha \\
\text{boxed type variable} & := \nu : T(B) \\
\text{boxed generalization} & := \nu : T_1(B) \times \cdots \\
\text{boxed dependent type} & := T_n(B) \\
\text{boxed dependent function} & := \nu : T_1(B) \rightarrow T_2(B)
\end{align*}
\]

Here, $x, e, K, P,$ and $\alpha$ are meta-variables ranging over variables, constants, constructors, predicate variables, and type variables respectively. We also use a meta-variable $\nu$ which ranges over the variables not appearing in expressions. We write $\text{FV}(e)$ to denote the set of free variables in $e$. Constants may include integer arithmetic operations. The language supports data constructors, pattern-matches, tuples, and a failure expression, which are partially or not supported by the existing languages $\lambda\ell$ [25] and $\lambda\ell$ [13, 21] with dependent type systems.

The syntax of pattern-matches and that of patterns is restricted without loss of generality for simplicity of constraint generation. We assume the variables in the pattern $(x_1, \ldots, x_m)$ are distinct each other. We can express if-then-else expressions if $e_1$ then $e_2$ else $e_3$ by using the pattern-matches and data con-
structors True and False as follows:

\[ \text{let } b = e_1 \text{ in match } b \text{ with } True \rightarrow e_2 \mid False \rightarrow e_3. \]

We can use the fixed-point operator \( \text{fix } x.e \) to define recursive functions. We abbreviate \( \text{let } f = \text{ fix } f.\lambda x.e \) as \( \text{let rec } f = e_1 \text{ in } e_2 \). For example, the function \( \text{sum} \) discussed in Section 2 can be defined as follows:

\[ \text{let rec } \text{sum} = \lambda x.\text{if } x \leq 0 \text{ then } 0 \text{ else } x + \text{sum}(x-1) \text{ in } \ldots \]

The meta-variable \( \phi \) ranges over the formulas of some first-order theory. In this paper, we especially consider the quantifier-free theory of linear arithmetic and equalities with uninterpreted function symbols.

Unlike \( \lambda_L \) and \( \lambda_H \), we separate the language for computation (i.e. expressions) and specification (i.e. formulas) for simplicity of constraint solving. We write \( \tau, \sigma \) for ordinary ML types and schemas, \( T, S \) for dependent types and schemas, which are used in our dependent type system, and \( T, S \) for dependent type and schema templates, which are used in constraint generation. We write \( B \) to denote base types which consist of user-defined and built-in types such as bool and int. Base types can take a sequence of type parameters, which is denoted by \( B(T) \). For example, in the type \( T \text{list of } T \), the type parameter \( T \) represents the type of the list elements. Our type system supports let-polymorphism. Thus, we can use type variables denoted by \( \alpha \) in types. We write \( TV(T) \) to denote the set of free type variables in \( T \). The main advantage of our type system is that it supports general refinement types \([13]\), dependent type and, dependent function types. We can use the general refinement types to express refinements of base types. For example, \( \{ \nu : \text{int } | \nu \geq 0 \} \) denotes the type of non-negative integers. The dependent type and can express relations among the tuple elements. For example, \( \{ \nu : \text{int } \times \text{int } | \nu.1 \geq \nu.2 \} \) denotes the type of integer pairs whose first element (denoted by \( \nu.1 \)) is greater than or equal to the second element (denoted by \( \nu.2 \)). We can use the dependent function types to make the type of the return value of a function depend on its arguments. For example, the type \( \nu \vdash (\nu_1 : \text{int } \rightarrow \nu_2 : \text{int } | \nu_3 = \nu_1 + \nu_2) \) expresses that the return value of the integer addition operator \( + \) is the sum of the two arguments (denoted by \( \nu_1 \) and \( \nu_2 \)). Type environments \( \Gamma \) and environment templates \( \Gamma \) are sequences of type bindings \( x : S \) and \( x : \hat{S} \) respectively, which may include guard formulas \( \phi \).

C.2 Operational Semantics

The call-by-value operational semantics of the language is given in Figure 10. Evaluation of the failure expression \( \text{fail} \) always gets stuck. We can use \( \text{fail} \) to model an array access \( a[e] \) as follows:

\[ \text{let } i = e \text{ in if } 0 \leq i < n \text{ then } a_i \text{ else fail.} \]

Here, \( n \) and \( a_i \) represent the size and the \( i \)-th element of the array \( a \) respectively. Similarly, an assertion \( \text{assert } e \) can be modeled as follows:

\[ \text{if } e \text{ then Unit else fail.} \]

In E-CON, the constant \( e \) is interpreted as a function written \([e]\). For example \( + 1 2 \rightarrow [+](1)2 \rightarrow [+](1)(2) = 3. \) In E-MATCH, we use the following auxiliary function \( \text{Unify}(p, v) \) which returns a substitution \( \rho \) for the variables in \( p \) such that \( \rho(p) = v \) (if any):

\[ \text{Unify}(x, v) = \{ x \mapsto v \} \]
\[ \text{Unify}(K, K) = \emptyset \]
\[ \text{Unify}(K, x) = \{ x \mapsto v \} \]
\[ \text{Unify}(x_1, \ldots, x_m, (v_1, \ldots, v_m)) = \{ x_1 \mapsto v_1, \ldots, x_m \mapsto v_m \} \]

We nondeterministically select a clause \( p \rightarrow e \) such that \( p \) matches to \( v \). We assume such a clause always exists; Note that we can always make a non-exhaustive pattern-match exhaustive by adding a clause \( p \rightarrow \text{fail} \) to the pattern-match for each missing pattern \( p \).

C.3 Type Judgment

The typing judgment is of the form \( \Gamma \vdash e : T \). It reads that the expression \( e \) has the type \( T \) under the type environment \( \Gamma \). The typing rules are defined in Figure 11. The helper functions \( TS(c) \) and \( TS(K) \) return the type schemas of \( c \) and \( K \) respectively. For example, we have \( TS(n) = \{ \nu : \text{int } | \nu = n \} \) for an integer \( n \), \( TS(\text{True}) = \{ \nu : \text{bool } | \nu = \text{true} \} \), and \( TS(\leq) = \nu_1 : \text{int } \rightarrow \nu_2 : \text{int } \rightarrow \{ \nu_3 : \text{bool } | (\nu_1 \leq \nu_2 \Rightarrow \nu_3 = \text{true}) \land (\nu_1 > \nu_2 \Rightarrow \nu_3 = \text{false}) \} \).

In T-VAR, given the type \( T \) of \( x \), the following helper function \( \text{TY}(x; T) \) returns as precise a type of \( x \) as possible:

\[ \text{TY}(x; T) = \begin{cases} \{ \nu : T \text{B } | \nu = x \} & \text{(if } T = \{ \nu : T \text{B } | \phi \} \text{)} \\ \{ \nu : T_1 \times \cdots \times T_m | \nu = x \} & \text{(if } T = \{ \nu : T_1 \times \cdots \times T_m | \phi \} \text{)} \\ T & \text{(otherwise)} \end{cases} \]

Note that it returns a more precise type than \( T \) if \( T \) is not a function type. In that case, information about \( x \) not included in \( T \) may be added later in our type system. In other words, the helper function can make types depend on program variables. For example, we can derive \( \lambda x : x : x : x \rightarrow \{ \nu : \text{int } | \nu = x \} \) by using T-VAR and T-ABS.

In T-FAIL, the following function \( \text{T} \) translates a type environment \( \Gamma \) to a logical formula that is implied by \( \Gamma \):

\[ \Gamma, x : \{ \nu : \text{bool } | \phi \} \]
\[ \Gamma, \nu \phi \]
\[ \Gamma, \phi \]

In our specification language, tuple selections \( \nu_i \) and constructors \( K \) are encoded by uninterpreted function symbols. The predicate \( \text{Valid}(\phi) \) holds if and only if \( \phi \) is valid.

The relation \( p \vdash x : T \) \( \Gamma \) reads that \( \Gamma \) holds if the pattern \( p \) matches to \( x \) with the type \( T \), and is defined as follows:

\[ \frac{\Gamma = x_1 : T_1, \ldots, x_m : T_m, x.1 = x_1 \land \cdots \land x.m = x_m}{x \vdash x' : T \vdash x : T} \]  
(P-VAR)

\[ \frac{\Gamma ; x_1 : T_1, \ldots, x_m : T_m \vdash x : T \rightarrow x.1 = \text{true}}{\Gamma \vdash x : T \times \text{int } | \nu \phi \rightarrow x.1 = \text{true} \land \nu \phi} \]  
(P-TUPLE)

\[ \frac{\Gamma \vdash \text{bool } | \phi \rightarrow \nu \phi \rightarrow \nu \phi}{\Gamma \vdash \nu : \text{bool } | \phi} \]  
(P-KON0)

\[ \frac{\Gamma \vdash \nu : \text{bool } | \phi, x : x.1 = \text{true}}{\Gamma \vdash x_2 : \{ \nu : \text{bool } | \phi \} \rightarrow x_2 : T_1 : T_2, x.1 = \text{true} \rightarrow x.2 = \text{true} \rightarrow x.1 = \text{true} \rightarrow x.2 = \text{true} \phi} \]  
(P-KON1)

Example C.1. Let us consider the following judgment:

\[ b : \{ \nu : \text{bool } | \nu = \text{true} \} \vdash \text{match } b \text{ with } True \rightarrow \text{Unit } | \text{False } \rightarrow \text{fail : unit.} \]
Run-time Expressions

\[ e ::= \cdots | \text{match } v \text{ with } \{p_i \rightarrow e_i\}_{i=1}^m \]

Evaluation Contexts

\[ E ::= [\star] | (v_1, \ldots, v_{i-1}, E_i, e_{i+1}, \ldots, e_m) | E \ e \ | v \ E \ | \text{let } x = E \text{ in } e | KE \]

Evaluation Rules

\[
\begin{align*}
\frac{c \ v \rightarrow [c](v)}{(E-\text{CON})} \\
\frac{(\lambda x.e) \ v \rightarrow [v/x]e}{(E-\text{APP})} \\
\frac{\text{fix } x.e \rightarrow \text{fix } x.e/x}{(E-\text{FIX})} \\
\frac{\text{let } x = \nu \text{ in } c \rightarrow [v/x]e}{(E-\text{LET})} \\
\frac{p \rightarrow e \in \{p_i \rightarrow e_i\}_{i=1}^m \ \text{Unify}(p_i, v) = \rho}{(E-\text{MATCH})} \\
\frac{\nu \ e = \nu}{(E-\text{CONTEXT})} \\
\frac{e \rightarrow e'}{e^{\ast} \rightarrow e' \quad (E-\text{REFL})} \\
\frac{e \rightarrow e'' \quad e'' \rightarrow e'}{e \rightarrow e' \quad (E-\text{TRANS})}
\end{align*}
\]

We can derive this judgment by using T-MATCH, T-KON0, and T-FAIL. In T-MATCH, we derive the following relation for the False-case:

\[
\text{False} \vdash b : \{\nu : \text{bool} \mid \nu = \text{true}\} \triangleright \Gamma_{\text{false}}.
\]

Here, we get \(\Gamma_{\text{false}} \equiv b = \text{false}\) since \(TS(\text{false}) = \{\nu : \text{bool} \mid \nu = \text{false}\}\). Thus, for checking the False-case, we need to derive the following judgment:

\[
b : \{\nu : \text{bool} \mid \nu = \text{true}\}, b = \text{false} \vdash \text{fail} : \text{unit}.
\]

We can derive this by T-FAIL, since we have:

\[
[b : \{\nu : \text{bool} \mid \nu = \text{true}\}, b = \text{false}] \equiv b = \text{true} \land b = \text{false} \equiv \bot
\]

and therefore \([b : \{\nu : \text{bool} \mid \nu = \text{true}\}, b = \text{false}] \vdash \bot\) is valid.

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

\[
\frac{\text{Valid}(\Gamma, \nu : \{\nu : T_1 \times \cdots \times T_m \mid \phi_1\}) \Rightarrow \phi_2}{\Gamma \vdash T_i <: T'_i \quad (i = 1, \ldots, m)}
\]

\[
\frac{\Gamma \vdash \nu : T_1 \times \cdots \times T_m \mid \phi_1}{\Gamma \vdash \nu : T_1 \times \cdots \times T_m \mid \phi_2 \quad (S-\text{BASE})}
\]

\[
\frac{\Gamma \vdash \nu : T_1 \times \cdots \times T_m \mid \phi_1}{\Gamma \vdash \nu : T_1 \times \cdots \times T_m \mid \phi_2 \quad (S-\text{TUPLE})}
\]

\[
\frac{\Gamma \vdash \nu : T_1 \rightarrow T_2 \mid \phi_1}{\Gamma \vdash \nu : T_1 \rightarrow T'_2 \mid \phi_2 \quad (S-\text{FUN})}
\]

Here, \(T_1, R, T_2\) is interpreted as \(T_1 <: T_2, T_2 <: T_1,\) or \(T_1 <: T_2 \land T_2 <: T_1\) depending on the variance of the \(i\)-th parameter of the base type \(B\).

**Example C.2.** Let us consider the following judgment:

\[
\Gamma \vdash \text{id} : \{\nu : \text{int} \mid \nu = 1\}
\]

Here, \(\Gamma = \text{id} : (x : \text{int} \rightarrow \{\nu : \text{int} \mid \nu = x\})\). We can derive the judgment by using T-APP, T-VAR, T-CON, and T-SUB. In T-APP, we need to derive the following judgment:

\[
\Gamma \vdash \text{id} : (x : \{\nu : \text{int} \mid \nu = x\}) \rightarrow \{\nu : \text{int} \mid \nu = 1\}.
\]

This is derived by T-SUB. In T-SUB, we need to derive the following relation:

\[
\Gamma \vdash x : \{\nu : \text{int} \mid \nu = x\} <: x : \{\nu : \text{int} \mid \nu = 1\} \rightarrow \{\nu : \text{int} \mid \nu = 1\}.
\]
This is broken down into derivations of the following two sub-relations:

\[ \Gamma \vdash \nu : \text{int} \mid \nu = 1 \]  
\[ \Gamma, x : \nu : \text{int} \mid \nu = 1 \vdash \nu : \text{int} \mid \nu = 1 \].

The former is derivable since \([\Gamma, \nu = 1] \Rightarrow \top\) is valid. The latter is also derivable since \([\Gamma, x : \nu : \text{int} \mid \nu = 1, \nu = x] \Rightarrow \nu = 1\) is valid, where \([\Gamma, x : \nu : \text{int} \mid \nu = 1, \nu = x] \equiv x = 1 \land \nu = x\).

Unlike in \(\lambda_L\) and \(\lambda_H\), the typing rules T-App and T-Match 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 the helper function \(T\mathcal{Y}(x; T)\) or the sub-typing rules.

### C.4 Type Soundness

The type system ensures that well-typed programs never get stuck. Formally, the following theorem holds:

**Theorem C.1 (Type Soundness).** If \( \Gamma \vdash e : T \) is derivable and \( \text{FV}(e) = \emptyset \), then \( e \) either evaluates to a value or diverges.

**Proof.** It follows from type preservation and progress lemmas (see the Appendix D for details). \(\square\)

### C.5 Constraint Generation Algorithm

The constraint generation algorithm is defined in Figure 12. The algorithm\( \text{Gen}\) takes a dependent type environment template \(\hat{\Gamma}\), an expression \(e\), and a dependent type template \(\hat{T}\) and generates a constraint on the predicate variables in the templates and newly generated predicate variables for the sub-expressions of \(e\). It has a solution, namely a substitution for the predicate variables if and only if \(\hat{\Gamma} \vdash e : \hat{T}\) is derivable. Basically, the algorithm tries to derive \(\hat{\Gamma} \vdash e : \hat{T}\) without checking the validity of the formulas of the form \([\hat{\Gamma}] \Rightarrow \phi\) in T-Fail, S-BASE, and S-TUPLE. Instead, it gathers these formulas and returns the conjunction of them as a constraint.

The auxiliary algorithm \(\text{Gen}_{\prec}\) is also defined in Figure 12. \(\text{Gen}_{\prec}\) takes a dependent type environment template \(\hat{\Gamma}\), a pair \((\hat{T}_1, \hat{T}_2)\) of dependent type templates and generates a constraint on the predicate variables in the templates. It has a solution if and only if \(\hat{\Gamma} \vdash \hat{T}_1 \prec \hat{T}_2\) is derivable.
In the algorithm, the following auxiliary function \( \text{Lift}(\bullet; \bullet) \) lifts a ML type (schema) to a dependent type (schema) template by introducing fresh predicate variables:

\[
\begin{align*}
\text{Lift}(\vec{x}; \alpha) &= \alpha \\
\text{Lift}(\vec{x}; T_1 \times \cdots \times T_m) &= \text{let } P \text{ be a fresh predicate variable} \\
&\quad \{ \nu : \text{Lift}(\vec{x}; T_1) \times \cdots \times \text{Lift}(\vec{x}; T_m) | P(\vec{x}, \nu) \} \\
\text{Lift}(\vec{x}; \tau \rightarrow \tau_2) &= \text{let } \nu \text{ be a fresh variable} \\
&\quad \nu : \text{Lift}(\vec{x}; \tau) \rightarrow \text{Lift}(\vec{x}, \nu; \tau_2) \\
\text{Lift}(\vec{x}; B) &= \text{let } P \text{ be a fresh predicate variable} \\
&\quad \text{let } \nu \text{ be a fresh variable} \\
&\quad \{ \nu : \text{Lift}(\vec{x}; \tau) B | P(\vec{x}, \nu) \} \\
\text{Lift}(\vec{x}; \forall \alpha. \tau) &= \forall \alpha. \text{Lift}(\vec{x}; \tau)
\end{align*}
\]
Note that some expression is annotated with ML types and schemas for simplicity of constraint generation. \(x[\tau], e[\tau], \text{ and } K[\tau]\) mean that a type schema of \(x\), \(e\), and \(K\) is instantiated with \(\tau\) respectively, \(e : \tau\) and \(x : \sigma\) mean that \(e\) and \(x\) have a ML type \(\tau\) and schema \(\sigma\) respectively. To obtain these annotations, we assume that ML types and schemas of a program are inferred a priori with Hindley-Milner type inference algorithm.

**Example C.3.** Let us consider the function `sum` presented in Section C.1. After Hindley-Milner type inference, the program is desugared and annotated with ML types as follows:

\[
\begin{align*}
e_{\text{sum}} & \quad \text{let } \text{sum} : (\text{int} \rightarrow \text{int}) = \text{fix } \text{sum}, \lambda x. \notag \\
& \quad \text{let } y : \text{bool} = (\leq) (x : \text{int}) (0 : \text{int}) \text{ in } \notag \\
& \quad \text{match } y \text{ with True } \rightarrow 0 \notag \\
& \quad | \text{False } \rightarrow \notag \\
& \quad \text{(+)} (x : \text{int}) (\sum (- (x : \text{int}) (1 : \text{int}) : \text{int}) \notag \\
& \quad \text{...}
\end{align*}
\]

We explain how the constraint \(\text{Gen} \left( \emptyset \vdash e_{\text{sum}} : T \right)\) is computed below. First, the ML type \(\text{int} \rightarrow \text{int}\) of \(\text{sum}\) is lifted to the following dependent type template:

\[
\begin{align*}
T_{\text{sum}} & = \text{Lift}(e_{\text{sum}}) \rightarrow \text{int} \rightarrow \text{int} \notag \\
& = \nu_1 : \{\nu_1 : \text{int} \mid P(\nu_1)\} \rightarrow \{\nu_2 : \text{int} \mid Q(\nu_1, \nu_2)\}
\end{align*}
\]

Then, we compute the following constraint:

\[
\begin{align*}
\text{Gen} \left( \emptyset \vdash \text{fix } \text{sum}, \lambda x. \text{let } y : \text{bool} = \text{... in ... : } T_{\text{sum}} \right).
\end{align*}
\]

The computation is reduced to that of the following constraint for \(T = \text{sum} : T_{\text{sum}}, x : \nu : \text{int} \mid P(\nu)\):

\[
\begin{align*}
\text{Gen} \left( \Gamma \vdash \text{let } y : \text{bool} = \text{... in ... : } \{\nu : \text{int} \mid Q(\nu, x)\} \right).
\end{align*}
\]

We then lift the ML type bood of \(y\) to \(\text{Lift}(\text{sum}, x : \text{bool}) = T_y = \{\nu : \text{bool} \mid P_y(\sum(x, \nu))\},\) the type of \(x\) to \(\text{Lift}(\text{sum}, x : \text{int}) = \{\nu : \text{int} \mid P_0(\sum(x, \nu))\},\) and the type of \(0\) to \(\text{Lift}(\text{sum}, x : \text{int}) = \{\nu : \text{int} \mid P_0(\sum(x, \nu))\}\) by introducing fresh predicative variables \(P_y, P_x,\) and \(P_0,\) and compute the following constraints:

\[
\begin{align*}
\text{Gen} \left( \Gamma \vdash (\leq) (x : \text{int}) (0 : \text{int}) : T_y \right),
\text{Gen} \left( \Gamma, y : T_y \vdash \text{match ... : } \{\nu : \text{int} \mid Q(\nu, x)\} \right).
\end{align*}
\]

The computation of the former is reduced to that of the following constraints:

\[
\begin{align*}
\text{Gen}_{<}\left( \Gamma, T S(\leq) : \leq : T^t \right),
\text{Gen}_{<}\left( \Gamma \vdash \{\nu : \text{int} \mid \nu = x\} : \{\nu : \text{int} \mid P_y(\sum(x, \nu))\} \right),
\text{Gen}_{<}\left( \Gamma \vdash \{\nu : \text{int} \mid \nu = 0\} : \{\nu : \text{int} \mid P_0(\sum(x, \nu))\} \right).
\end{align*}
\]

Here, \(T^t = \nu_1 : \{\nu : \text{int} \mid P_y(\sum(x, \nu))\} \rightarrow \nu_2 : \{\nu : \text{int} \mid P_0(\sum(x, \nu))\} \rightarrow \nu_1 : \{\nu : \text{bool} \mid P_0(\sum(x, \nu))\}.\) The conjunction of the results is equivalent to the following constraint without \(P_y\) and \(P_0:\)

\[
(x \leq 0 \Rightarrow \nu = \text{true} \land x > 0 \Rightarrow \nu = \text{false}) \Rightarrow P_y(\sum(x, \nu)).
\]

The computation of the latter proceeds similarly, and we obtain the following constraints:

\[
\begin{align*}
P(x) \land P_y(\sum(x, \nu) \land \nu = \text{true} & \Rightarrow Q(x, 0),
\Gamma \vdash (P(x) \land P_y(\sum(x, \nu) \land \nu = \text{false} : P(x - 1)), \notag \\
P(x) \land P_y(\sum(x, \nu) \land \nu = \text{false} \land Q(x - 1, \nu') & \Rightarrow Q(x, x + \nu')).
\end{align*}
\]

The conjunction of the constraints is equivalent to the following one without \(P_y:\)

\[
\begin{align*}
(P(x) \land x \leq 0 & \Rightarrow Q(x, 0)) \land 
\Gamma \vdash (P(x) \land x > 0 : P(x - 1)), \notag \\
(P(x) \land x > 0 \land Q(x - 1, \nu') & \Rightarrow Q(x, x + \nu')).
\end{align*}
\]

**C.6 Algorithm Soundness and Completeness**

The following theorem establishes soundness and completeness of the constraint generation algorithm with respect to the type system:

**Theorem C.2 (Algorithm Soundness and Completeness).** (a) If \(\emptyset(\text{Gen} \left( \Gamma, T^t : \leq : T^t \right))\) is valid, then \(\emptyset(\Gamma) \vdash e : \emptyset(T) \) is derivable.

(b) If \(\emptyset(\Gamma) \vdash e : \emptyset(T) \) is derivable, then there exists \(\emptyset'\) such that \(\emptyset \leq \emptyset'\) and \(\emptyset'(\text{Gen} \left( \Gamma, T^t : \leq : T^t \right))\) is valid.

**Proof.** (a) Induction on the structure of \(e\). (b) Induction on the derivation of \(\emptyset(\Gamma) \vdash e : \emptyset(T)\). See the Appendix E for details. □

**D. Proof of Type Soundness**

This section proves Theorem C.1. The theorem is a corollary of the type preservation (Lemma D.4) and the progress (Lemma D.5) lemmas, which are proved in Sections D.1 and D.2 respectively.

We assume that if \(\Gamma, e : \emptyset(T_1) \rightarrow \emptyset(T_2), \Gamma \vdash \nu : \emptyset(T_1) \) and \(\Gamma \vdash \emptyset(T) \rightarrow \emptyset(T_1)\), then \(\emptyset(T)\) is defined and \(\Gamma, \nu : \emptyset(T) \vdash \emptyset(T)\) is defined.

We define the typing rule for the run-time expression \(\text{match } v \text{ with } \{p_1 \rightarrow e_1\}_{i=1}^m\) as follows:

\[
\begin{align*}
\Gamma \vdash v : T', p_i \downarrow v : T'_i \rightarrow \Gamma_i \rightarrow \Gamma, \Gamma_i \vdash e_1 : T \notag \\
\Gamma \vdash \text{match } v \text{ with } \{p_i \rightarrow e_1\}_{i=1}^m : T \notag \\
\end{align*}
\]

In \(\text{T-Match}'\), the relation \(p \downarrow v : T' \rightarrow \Gamma\) reads that \(\Gamma\) holds if the pattern \(p\) matches \(v\) with the type \(T\), and is defined as follows:

\[
\begin{align*}
x \downarrow v : T \rightarrow x : T \notag \\
\end{align*}
\]

\[
\begin{align*}
\Gamma = x_1 : T_1, \ldots, x_m : T_m, v, 1 = x_1 \land \cdots \land v, m = x_m \notag \\
\text{dom}(\Gamma_i) \cap \text{FV}(T) = \emptyset \notag \\
\end{align*}
\]

\[
\begin{align*}
\text{K} \downarrow v : \{v : T B | \phi\} \notag \\
\text{K} \downarrow v : \{v : T B | \phi\} \notag \\
\end{align*}
\]

\[
\begin{align*}
\text{K} \downarrow v : \{v : T B | \phi\} \notag \\
\text{K} \downarrow v : \{v : T B | \phi\} \notag \\
\end{align*}
\]
D.1 Preservation

Lemma D.1. If $\Gamma \vdash v : T$ and $\Gamma, x : T', \Gamma' \vdash T_1 : T_2$, then $\Gamma, [v/x]T' \vdash [v/x]T_1 : [v/x]T_2$ is derivable.

Proof. By induction on the derivation of $\Gamma, x : T', \Gamma' \vdash T_1 : T_2$.

Lemma D.2 (Substitution). If $\Gamma \vdash v : T'$ and $\Gamma, x : T', \Gamma' \vdash e : T$, then $\Gamma, [v/x]T' \vdash [v/x]e : [v/x]T'$ is derivable.

Proof. We prove the lemma by induction on the derivation of $\Gamma, x : T', \Gamma' \vdash e : T$.

T-Var We have $e = y$. If $x = y$, then we get $[v/x]e = v$. Thus, we have $\Gamma \vdash [v/x]e : T$. We prove $\Gamma \vdash [v/x]e : [v/x]T$ by case analysis on $v$:

- If $v = c$, then $T' = \{v : \overrightarrow{T B} | \emptyset\}$ and $[v/x]T = \{v : \overrightarrow{T B} | v = v\}$. By T-CON, we obtain $\Gamma \vdash [v/x]e : [v/x]T$.
- If $v = K$, then $T' = \{v : \overrightarrow{T B} | \emptyset\}$ and $[v/x]T = \{v : \overrightarrow{T B} | v = v\}$. By T-KON0, we obtain $\Gamma \vdash [v/x]e : [v/x]T$.

By I.H., we have $\Gamma \vdash [v/x]T' \vdash [v/x]e' : [v/x]T'$. By Lemma D.1, we get $\Gamma, [v/x]T' \vdash [v/x]e' : [v/x]T'$. By T-KON1, we obtain $\Gamma \vdash T', [v/x]T' \vdash [v/x]e' : [v/x]T'$.

T-match We have $e = match y \text{ with } \{p_i \to e_i\}_{i=1}^{m}$, where

- $y : \forall \overline{\nu T}_y \in (\Gamma, x : T', \Gamma')$.
- $\Gamma \vdash y : \overrightarrow{T B} | \emptyset$.
- $\Gamma, x : T', \Gamma' \vdash e_i : T_i$.
- $\Gamma \vdash \exists \overline{\alpha B} \gamma \in \bigwedge_i \{T_i \}$.

By I.H., we have $\Gamma \vdash T', [v/x]T' \vdash [v/x]e' : [v/x]T'$. By Lemma D.1, we get $\Gamma, [v/x]T' \vdash [v/x]e' : [v/x]T'$. By T-KON1, we obtain $\Gamma \vdash T', [v/x]T' \vdash [v/x]e' : [v/x]T'$. Since $\Gamma \vdash [v/x]e = fix_y [v/x]e'$, we obtain $\Gamma \vdash [v/x]e' : [v/x]T'$.
We have $e = \text{fail}$. Since $[v/x]e = \text{fail}$, we have $\Gamma, [v/x]e \vdash [v/x]T$. By Lemma D.1, we obtain $\Gamma, [v/x]e \vdash [v/x]T$.

By T-SUB, we obtain $\Gamma, [v/x]e \vdash [v/x]T$.

If $e_1$ is not a value, only E-CONTEXT applies to $e$ and thus we get $e' = e_1 e_2$ for some $e_2$ such that $e_2 \rightarrow e_2'$.

By I.H., we get $\Gamma \vdash e_1' : (x : T' \rightarrow T)$, and hence we get $e_1' e_2 : T$. By T-APP, we get $\Gamma \vdash e_1' e_2 : T$.

If $e_1$ is a value and $e_2$ is not a value, only E-CONTEXT applies to $e$ and thus we get $e' = e_1 e_2$ for some $e_2$ such that $e_2 \rightarrow e_2'$.

By I.H., we get $\Gamma \vdash e_2' : T$. By T-APP, we get $\Gamma \vdash e_1' e_2' : T$.

By Lemma D.3, we have $\Gamma \vdash e' = e_1' e_2'$.

Therefore, $\Gamma \vdash e' : T$ is derivable.

\[ \square \]

Lemma D.3. Suppose that

- $\Gamma \vdash v : T'$,
- $\rho \vdash x : T$,
- $\Gamma' \vdash \rho(v) : \rho(T)$.

Then, $\Gamma \vdash \rho(e) : \rho(T)$ is derivable.

Proof. We prove the lemma by case analysis of the structure of $\rho$.

Case $\rho = x$. We have $\rho = (x \mapsto v)$. By P-VAR', we get $\Gamma' \vdash x : T'$.

By Lemma D.2, we obtain $\Gamma \vdash \rho(e) : \rho(T)$.

Case $\rho = K$. We have $v = k$ and $\rho = 0$. By P-KON0', we get $\Gamma' = [v/\phi]T'$. Then, $\rho = 0$.

We have $\Gamma' \vdash v' : [\overline{T} / \overline{\alpha}]T''$, and $\rho = 0$.

We have $\Gamma' \vdash v' : [\overline{T} / \overline{\alpha}]T''$. By Lemma D.2, we obtain $\Gamma, [v'/v_1, v/v_2] \vdash \rho(e) : \rho(T)$. Since $[v'/v_1, v/v_2]T = T$.

We have $\Gamma \vdash \rho(e) : \rho(T)$.

Case $\rho = (x_1, \ldots, x_m)$. We have $v = (v_1, \ldots, v_m)$ and $\rho = (x_1 \mapsto v_1, \ldots, x_m \mapsto v_m)$. By P-TUPLE', we get $\Gamma' = x_1 : T_1, \ldots, x_m : T_m$.

$\rho = (x_1 \mapsto v_1, \ldots, x_m \mapsto v_m)$. We have $\Gamma \vdash v_i : T_i$ for all $i \in \{1, \ldots, m\}$.

By Lemma D.2, we obtain $\Gamma, v_1 \land \cdots \land v_m \vdash \rho(e) : \rho(T)$. Since $v_1 \land \cdots \land v_m = v_m$, we obtain $\Gamma \vdash \rho(e) : \rho(T)$.

Therefore, $\Gamma \vdash e : T$ is derivable.

\[ \square \]

Lemma D.4 (Preservation). Suppose that $\Gamma \vdash e : T$ and $e \rightarrow e'$. Then, $\Gamma \vdash e' : T$ is derivable.

Proof. We prove the lemma by induction on the derivation of $\Gamma \vdash e : T$.

T-Var We have $e = x$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-Con We have $e = c$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-Tuple We have $e = (v_1, \ldots, v_m)$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-Abs We have $e = x. e_1$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-App We have $e = e_1 e_2$, where $\Gamma \vdash e_1 : (x : T' \rightarrow T)$, $\Gamma \vdash e_2 : T'$, and $x \notin \text{FV}(T)$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-Fail We have $e = \text{fail}$.

This case is impossible since there is no $e'$ such that $e \rightarrow e'$.

T-Sub We have $\Gamma \vdash e : T'$ and $\Gamma' \vdash v : T' \rightarrow T$. By I.H., we obtain $\Gamma \vdash e' : T'$.

Thus, $\Gamma' \vdash v : T'$. By T-LET, we get $\Gamma \vdash e'' : T'$.

Therefore, $\Gamma \vdash e' : T$ is derivable.

\[ \square \]

D.2 Progress

Lemma D.5 (Progress). Suppose that $\vdash e : T$ and $\text{FV}(e) = \emptyset$. Then, either $e$ is a value or there exist $e'$ such that $e \rightarrow e'$.
Proof. We prove the lemma by induction on the derivation of \( e : T \).

T-Var This case is impossible since \( \text{FV}(e) = \emptyset \).

T-Con We have \( e = c \).

\( c \) is a value.

T-Pair We have \( e = (e_1, \ldots, e_m) \), where \( \vdash e_i : T_i \) for \( i = 1, \ldots, m \).

By I.H., either \( e \) is a value or there exists \( i \in \{1, \ldots, m\} \) such that \( e_i \longrightarrow e'_i \).

In the latter case, we can apply E-CONTEXT.

T-Abs We have \( e = \lambda x.e_1 \).

\( \lambda x.e_1 \) is a value.

T-App We have \( e = e_1 e_2 \), where \( \vdash e_1 : T' \rightarrow T, \vdash e_2 : T' \).

By I.H., \( (a) e_1 \longrightarrow e'_1 \) or \( (b) e_2 \longrightarrow e'_2 \) or \( (c) e_1 \) and \( e_2 \) are values.

In the cases (a) and (b), we can apply E-CONTEXT.

In the case (c), \( e_1 = \lambda x.e' \) for some \( x \) and \( e' \). Thus, we can apply E-APP.

T-Let We have \( e = \text{let } x = e_1 \text{ in } e_2 \), where \( \vdash e_1 : T' \).

By I.H., either \( (a) e_1 \longrightarrow e'_1 \) or \( (b) e_2 \) is a value.

In the case (a), \( K \) is a value.

In the case (b), we can apply E-CONTEXT.

T-Match This case is impossible since \( \text{FV}(e) = \emptyset \).

T-Match’ We have \( e = \text{match } v \) with \( \{p_1 \rightarrow e_1\} \).

We can apply E-MATCH.

T-Fail We have \( e = \text{fail} \), where Valid([\( \emptyset \] \( \rightarrow \) \( \perp \) \)]).

This case is impossible since Valid([\( \emptyset \] \( \rightarrow \) \( \perp \) \)] is inconsistent.

T-Sub We have \( \vdash e : T' \).

By I.H., either \( e \) is a value or there exists \( e' \) such that \( e \longrightarrow e' \).

\[ \square \]

E. Proof of Soundness and Completeness of Constraint Generation

This section proves Theorem C.2, which consists of algorithm soundness and completeness. The algorithm soundness (Lemma E.2) is proved in Section E.1. The algorithm completeness (Lemma E.5) is proved in Section E.2.

E.1 Algorithm Soundness

Lemma E.1. If \( \theta(\Gamma, \psi) \vdash \theta(\Gamma, \psi) \vdash \Gamma < : T \).

Proof. We prove the lemma by induction on the structure of \( \Gamma \).

Case \( \Gamma_1 = \alpha \). We have \( \Gamma_2 = \alpha \).

By S-VAR, we obtain \( \theta(\Gamma) \vdash \psi \) and \( \theta(\Gamma) \vdash \psi \).

We have \( \Gamma_2 \) is valid.

By I.H., we obtain \( \theta(\Gamma) \vdash \theta(\Gamma) \vdash \theta(\Gamma) \).

\[ \square \]

Lemma E.2 (Algorithm Soundness). If \( \theta(\Gamma, \psi) \vdash \theta(\Gamma, \psi) \vdash \Gamma < : T \).

Proof. We prove the lemma by induction on the structure of \( e \).

Case \( e = x \). We have \( e \vdash \text{valid} \).

We know \( \theta(\Gamma) \vdash \psi \) and \( \theta(\Gamma) \vdash \psi \).

By I.H., we obtain \( \theta(\Gamma) \vdash \theta(\Gamma) \vdash \theta(\Gamma) \).

\[ \square \]
Since we get \(\theta([\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau})/\alpha]T') = [\theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))]/\alpha T'\), we obtain \(\theta(\widehat{\Gamma}) \vdash [\theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))]/\alpha \theta(T')\). We get \(\theta(\widehat{\Gamma}) \vdash [\theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))]/\alpha \theta(T') <: \theta(\widehat{T})\).

By T-SUB, we get \(\theta(\widehat{\Gamma}) \vdash c : \theta(\widehat{T})\).

**Case** \(e = (v_1, \ldots, v_m)\) We have \(\widehat{T}_1 = \text{Lift}(\text{dom}(\widehat{\Gamma}); \tau_i)\) and \(\theta(\text{Gen}(\widehat{T} \vdash v_i : \widehat{T}_i))\) for all \(i \in \{1, \ldots, m\}\), and \(\theta(\text{Gen}_< (\widehat{T} \vdash \{v : \widehat{T}_1 \times \cdots \times \widehat{T}_m | v = (v_1, \ldots, v_m)\}) <: \theta(\widehat{T})\).

By I.H., we get \(\theta(\widehat{\Gamma}) \vdash v_i : \theta(\widehat{T}_i)\) for all \(i \in \{1, \ldots, m\}\).

By T-TUPLE, we have \(\theta(\widehat{\Gamma}) \vdash \langle v_1, \ldots, v_m \rangle : \{\nu : \langle \theta(\widehat{T}) \rangle \times \cdots \times \theta(\widehat{T}_m) | \nu = (v_1, \ldots, v_m)\}\).

By Lemma E.1, we get \(\theta(\widehat{T}) \vdash \theta(\{\nu : \langle \theta(\widehat{T}) \rangle \times \cdots \times \theta(\widehat{T}_m) | \nu = (v_1, \ldots, v_m)\}) <: \theta(\widehat{T})\).

Since we have \(\theta(\{\nu : \langle \theta(\widehat{T}) \rangle \times \cdots \times \theta(\widehat{T}_m) | \nu = (v_1, \ldots, v_m)\}) = \{\nu_1 : \theta(\widehat{T}_1) \times \cdots \times \nu_m : \theta(\widehat{T}_m) | \nu = (v_1, \ldots, v_m)\}\), we have \(\theta(\widehat{\Gamma}) \vdash \langle v_1, \ldots, v_m \rangle : \theta(\widehat{T})\).

By T-SUB, we get \(\theta(\widehat{\Gamma}) \vdash \langle e_1, \ldots, e_m \rangle : \theta(\widehat{T})\).

**Case** \(e = \lambda x.e'\) We have \(\theta(\text{Gen}(\widehat{T}, x : \widehat{T}_1 \vdash e' : \widehat{T}_2))\) and \(\widehat{T} = \langle x : \hat{T}_1 \rightarrow \hat{T}_2 \rangle\).

By I.H., we get \(\theta(\widehat{T}, x : \hat{T}_1) \vdash e' : \theta(\widehat{T}_2)\).

Since \(\theta(\langle x : \hat{T}_1 \rangle, x : \theta(\hat{T}_1))\), we obtain \(\theta(\hat{T}_1), x : \hat{T} \vdash e' : \theta(\hat{T}_2)\).

By T-ABS, we obtain \(\theta(\hat{T}) \vdash \lambda x.e' : \theta(\text{Gen}(\hat{T}, x : \hat{T}_1 \vdash e' : \hat{T}_2))\).

Since \(\theta(\hat{T}) = \langle x : \hat{T}_1 \rightarrow \hat{T}_2 \rangle\), we get \(\theta(\hat{T}) \vdash \lambda x.e' : \theta(\hat{T})\).

**Case** \(e = e_1 e_2\) We have \(\widehat{T} = \text{Lift}(\text{dom}(\widehat{\Gamma}); \tau), \theta(\text{Gen}(\hat{T} \vdash e_1 : \hat{T} \rightarrow \hat{T}))\), \(\alpha \subseteq \text{PTV}(\hat{T})\).

By I.H., we get \(\theta(\hat{T}) \vdash e_1 : \theta(\hat{T} \rightarrow \hat{T})\). We get \(\theta(\hat{T}) \vdash e_2 : \theta(\hat{T})\).

Since \(\theta(\hat{T} \rightarrow \hat{T}) = \langle \hat{T} \rightarrow \hat{T} \rangle\), we have \(\theta(\hat{T}) \vdash e_1 : \hat{T} \rightarrow \hat{T} \Rightarrow \theta(\hat{T})\).

By T-APP, we obtain \(\theta(\hat{T}) \vdash e_1 e_2 : \theta(\hat{T})\).

**Case** \(e = \eta\) We have \(\forall (\hat{T}, \tau) = \text{Lift}(\text{dom}(\widehat{\Gamma}); \sigma), \hat{T} \subseteq \hat{T}_1 \vdash \hat{T} \vdash \hat{T}_2\).

By I.H., we get \(\theta(\hat{T}, x : \hat{T} \rightarrow \hat{T}) \vdash e : \theta(\hat{T})\).

Since \(\theta(\hat{T}, x : \hat{T}) = \langle \hat{T} \rightarrow \hat{T} \rangle\), we have \(\theta(\hat{T}), x \vdash e : \theta(\hat{T})\).

By T-LET, we get \(\theta(\hat{T}) \vdash \eta = e_1 \text{ in } e_2 \vdash : \theta(\hat{T})\).

**Case** \(e = \mathbf{f} x.e'\) We have \(\theta(\hat{T}, x : \hat{T} \vdash e' : \hat{T})\).

By I.H., we get \(\theta(\hat{T}, x : \hat{T}) \vdash e : \theta(\hat{T})\).

Since \(\theta(\hat{T}, x : \hat{T}) = \langle \hat{T}, x : \theta(\hat{T})\rangle\), we have \(\theta(\hat{T}), x \vdash e : \theta(\hat{T})\).

By T-FIX, we get \(\theta(\hat{T}) \vdash \mathbf{f} x.e' : \theta(\hat{T})\).

**Case** \(e = K\) We have \(\forall (\hat{T}, \tau) = T.S(\hat{\Gamma})\) and \(\theta(\text{Gen}_<(\hat{T} \vdash [\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))/\alpha]T' <: \tau])\).

By T-KoN0, we get \(\theta(\hat{T}) \vdash K : [\theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))/\alpha] \theta(T')\).

By Lemma E.1, we get \(\theta(\hat{T}) \vdash \langle \theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))/\alpha \rangle T' <: \theta(\hat{T})\).

Since it is the case that

\[\theta(\hat{T}) \vdash [\theta(\text{Lift}(\text{dom}(\widehat{\Gamma}); \widehat{\tau}))/\alpha]T' <: \theta(\hat{T})\]
Proof. We can prove the lemma by induction on the structure of $e$.

Lemma E.5 (Algorithm Completeness). If $\theta(\overline{\Gamma}) \vdash e : \theta(\overline{\Gamma})$ is derivable, then there exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'(\text{Gen} \; \overline{\Gamma} \vdash e : \theta(\overline{\Gamma}))$ is valid.

Proof. We prove the lemma by induction on the derivation of $\theta(\overline{\Gamma}) \vdash e : \theta(\overline{\Gamma})$.

T-Var We have $e = x$ and $\theta(\overline{\Gamma}) = \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow$, where $x : \forall \overline{a} \, . \, T \in \theta(\overline{\Gamma})$.

We have $\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow \rightarrow \overline{T}$.\n
We have $\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow \rightarrow \overline{a} \, T < \overline{T}$, where $x : \forall \overline{a} \, . \, T \in \overline{a}$.

There exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'((\overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow) = \theta'(\overline{T})$.

By Lemma E.3, there exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'(\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow < \overline{T})$.

T-Con We have $e = c$ and $\theta(\overline{\Gamma}) = [\overline{T}/\overline{a}] \downarrow$, where $\text{TS}(e) = \forall \overline{a} \, . \, T$.

We have $\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow \rightarrow \overline{T}$.

There exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'((\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow) = \theta'(\overline{T})$.

By Lemma E.3, there exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'(\text{Gen}_{\overline{c}} \; \overline{T} \, y(x; \overline{T}/\overline{a}) \downarrow < \overline{T})$.

T-Tuple We have $e = (v_1, \ldots, v_m)$ and $\theta(\overline{\Gamma}) = \{ \nu : T_i \times \cdots \times T_m : \nu = (v_1, \ldots, v_m) \}$, where $\theta(\overline{\Gamma}) = v_i$ for all $i \in \{1, \ldots, m\}$.

We have $\text{Gen}_{\overline{c}} \; (v_1 : T_1 \times \cdots \times T_m : \nu = (v_1, \ldots, v_m) : \overline{T})$.

\[
\bigwedge_{i=1}^{m} \text{Gen}_{\overline{c}} \; (v_i : T_i) \wedge
\]

\[
\text{Gen}_{\overline{c}} \; (\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T}).
\]

where $\overline{T} = \text{Lift}(\text{dom}(\overline{\Gamma}) : \nu)$ for all $i \in \{1, \ldots, m\}$. There exists $\theta'$ such that $\theta \subseteq \theta'$ and $\theta'(\overline{T})$ for all $i \in \{1, \ldots, m\}$.

By I.H., we have $\theta'((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.

By I.H., there exists $\theta''$ such that $\theta \subseteq \theta''$ and $\theta''(\text{Gen} \; \overline{\Gamma} \vdash e : \theta(\overline{\Gamma}))$ for all $i \in \{1, \ldots, m\}$.

Since, $\theta''((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.

By I.H., we have $\theta''((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.

By I.H., we have $\theta''((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.

By I.H., we have $\theta''((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.

By I.H., we have $\theta''((\overline{T} \downarrow : \nu = (v_1, \ldots, v_m) < \overline{T})$.
By I.H., there exists \( \theta' \) such that \( \theta \subseteq \theta' \) and \( \theta' (\text{Gen}(\overline{\Gamma}, x : \overline{T} \vdash e' : \overline{T})) \).

**T-Kon0** We have \( e = K \) and \( \theta(\overline{T}) = [\overline{T}/\overline{\alpha}]T' \), where \( T\Sigma(K) = \forall \overline{\alpha}.T' \).

We have \( \text{Gen}(\overline{\Gamma} \vdash K[\overline{\alpha}] : \overline{T}) = \text{Gen}_c : (\overline{\Gamma} \vdash \text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}[T'] < : \overline{T}) \).

where \( \forall \overline{\alpha}.T' = T\Sigma(K) \).

There exists \( \theta' \) such that \( \theta \subseteq \theta' \) and \( \theta' (\text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}[T']) = \theta'(\overline{T}) \).

By Lemma E.3, there exists \( \theta'' \) such that \( \theta' \subseteq \theta'' \) and
\[ \theta''(\text{Gen}_c : (\overline{\Gamma} \vdash \text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}[T'] < : \overline{T})). \]

**T-Kon1** We have \( e = K e' \) and \( \theta(\overline{T}) = T \), where \( T\Sigma(K) = \forall \overline{\alpha}.T', \theta(\overline{T}) \vdash e' : T'' \), \( \theta(\overline{T}) \vdash [\overline{T}/\overline{\alpha}]T' < : x : T'' \rightarrow T \), and \( x \notin \text{FV}(T) \).

We have \( \text{Gen}(\overline{\Gamma} \vdash K[e'] : \overline{T}) = \text{Gen}_c : (\overline{\Gamma} \vdash \text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}[T'] < : \overline{T''} \rightarrow \overline{T}) \land \text{Gen} (\overline{\Gamma} \vdash e' : \overline{T''}) \), where \( \forall \overline{\alpha}.T'' = T\Sigma(K) \), \( \overline{T''} = \text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha}), \) and \( x \notin \text{FV}(\overline{T}) \).

There exists \( \theta' \) such that \( \theta \subseteq \theta' \), \( \theta' (\text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})) = \overline{T} \), \( \theta'(\overline{T''}) = T'' \), and \( \theta'(\overline{T}) = T \).

By Lemma E.3, there exists \( \theta'' \) such that \( \theta' \subseteq \theta'' \) and
\[ \theta''(\text{Gen}_c : (\overline{\Gamma} \vdash \text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}[T'] < : \overline{T''} \rightarrow \overline{T})). \]

By I.H., there exists \( \theta'' \) such that \( \theta'' \subseteq \theta'' \) and
\[ \theta''(\text{Gen}_c : (\overline{\Gamma} \vdash e' : \overline{T''})). \]

**T-Match** We have \( e = \text{match } x \text{ with } \{ p_i \rightarrow e_i \}_{i=1}^m \) and \( \theta(\overline{T}) = T \), where \( x : \forall \overline{\alpha}.T' \in \theta(\overline{\Gamma}), p_i \downarrow x : [\overline{T}/\overline{\alpha}]T' > \overline{\Gamma}_i \), \( \theta(\overline{\Gamma}), \overline{\Gamma}_i \vdash e_i : T \), and \( \text{dom}(\overline{\Gamma}_i) \cap \text{FV}(T) = \emptyset \) for all \( i \in \{1, \ldots, m\} \).

We have \( \text{Gen}(\overline{\Gamma} \vdash \text{match } x[\overline{x}] \text{ with } \{ p_i \rightarrow e_i \}_{i=1}^m : \overline{T}) = \bigwedge_{i=1}^m \text{Gen}(\overline{\Gamma}, \overline{\Gamma}_i \vdash e_i : \overline{T}) \), where \( x : \forall \overline{\alpha}.\overline{T} \in \overline{\Gamma} \) and \( p_i \downarrow x : [\text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})/\overline{\alpha}]\overline{T} > \overline{\Gamma}_i \) for all \( i \in \{1, \ldots, m\} \).

There exists \( \theta' \) such that \( \theta \subseteq \theta' \), \( \theta' (\text{Lift}(\text{dom}(\overline{\Gamma}); \overline{\alpha})) = \overline{T} \).

Since \( \theta(\overline{\Gamma}), \overline{\Gamma}_i = \theta'(\overline{\Gamma}, \overline{\Gamma}_i) \), we get \( \theta'(\overline{\Gamma}, \overline{\Gamma}_i) \vdash e_i : \theta'(\overline{T}) \) for all \( i \in \{1, \ldots, m\} \).

By I.H., there exists \( \theta'' \) such that \( \theta' \subseteq \theta'' \) and
\[ \theta''(\text{Gen}_c : (\overline{\Gamma}, \overline{\Gamma}_i \vdash e_i : \overline{T})). \]

**T-Fail** We have \( e = \text{fail} \) and \( \theta(\overline{T}) = T \), where \( \text{Valid}([\theta(\overline{\Gamma})] \Rightarrow \bot) \).

We have \( \text{Gen}(\overline{\Gamma} \vdash \text{fail} : \overline{T}) = \overline{\Gamma} \Rightarrow \bot \).

\( \theta(\overline{T}) \Rightarrow \bot \) follows immediately.

**T-Sub** We have \( \theta(\overline{T}) = T \), where \( \theta(\overline{T}) \vdash e : T' \) and \( \theta(\overline{T}) \vdash T' < : T \).

By I.H., there exists \( \theta' \) such that \( \theta \subseteq \theta' \) and \( \theta' (\text{Gen}(\overline{\Gamma} \vdash e : T')) \).

By Lemma E.3, there exists \( \theta'' \) such that \( \theta' \subseteq \theta'' \) and
\[ \theta''(\text{Gen}_c : (\overline{\Gamma} \vdash T' < : T)). \]

By Lemma E.4, we get \( \theta''(\text{Gen}(\overline{\Gamma} \vdash e : T)) \).

\( \blacksquare \)