On-Demand Refinement of Dependent Types

Hiroshi Unno[1] and Naoki Kobayashi[2]

[1] University of Tokyo, `uhiro@yl.is.s.u-tokyo.ac.jp`
[2] Tohoku University, `koba@ecei.tohoku.ac.jp`

Abstract. Dependent types are useful for statically checking detailed specifications of programs and detecting pattern match or array bounds errors. We propose a novel approach to applications of dependent types to practical programming languages: Instead of requiring programmers' declaration of dependent function types (as in Dependent ML) or trying to infer them from function *definitions* only (as in size inference), we *mine* the output specification of a dependent function from the function's *call sites*, and then propagate that specification *backward* to infer the input specification. We have implemented a prototype type inference system which supports higher-order functions, parametric polymorphism, and algebraic data types based on our approach, and obtained promising experimental results.

1 Introduction

Dependent types are useful for statically verifying that programs satisfy detailed specifications and for detecting data-dependent errors such as pattern match or array bounds errors. For example, the function $\lambda x.x + 1$ is given a type $\texttt{int} \to \texttt{int}$ in the simple type system, but with dependent types, it is given a type $\Pi x : \texttt{int}.\{y : \texttt{int} \mid y = x + 1\}$, so that we can conclude that the array access $a[(\lambda x.x + 1)0]$ is safe (if the size of array a is more than 1).

There are several approaches to introducing dependent types into programming languages. Size inference [1–3] fixes the shape of dependent types *a priori* (e.g., a list type is of the form $\tau\ \texttt{list}^n$ where n is the length of a list), and tries to infer a dependent type of a function automatically from the function's definition. Shortcomings of that approach are inflexibility and inefficiency; for example, it would be hard to infer that a sorting function indeed returns a sorted list. Dependent ML (DML) [4, 5] lets users declare the dependent type of each function manually, and checks whether the declaration is correct. A shortcoming of that approach is that it is often cumbersome for programmers to declare dependent types for *all* functions. For example, consider the following function `isort` for insertion sort, and suppose that one wants to verify that `isort` returns a sorted list.

```
fun insert (x, xs) = match xs with
    Nil _ -> Cons (x, Nil ())
  | Cons (y, ys) -> if x <= y then Cons(x, xs) else Cons(y, insert (x, ys))
fun isort xs = match xs with
```

```
    Nil _ -> Nil ()
  | Cons (x, xs’) -> insert (x, isort xs’)
```

It would be fine to declare that `isort` returns a sorted list (because that is indeed the property to be verified). It is, however, cumbersome to declare a dependent type of the auxiliary function `insert` as well. Knowles and Flanagan [6] proposes a complete type reconstruction algorithm for a certain dependent type system, but the inferred types include fixed-point operators on predicates, so that the inferred types alone cannot be used for actual verification or bug finding (without a reasonable algorithm for computing fixed-points).

We propose an alternative, complementary approach to the previous approaches discussed above. Instead of requiring programmers’ declaration of dependent function types or trying to infer them from function *definitions* only, we infer a function’s type using information about not only the function’s definition but also the function’s *call sites*. Another related, distinguishing feature of our approach is that types are refined *on-demand*; we start with the simplest type for each function, and refine the type gradually, when it turns out that more precise type information is required by a call site of the function. For example, the function $f \overset{\triangle}{=} \lambda x.x + 1$ is first given a type $\texttt{int} \rightarrow \texttt{int}$, but if a calling context $a[f\ y]$ is encountered, the type is refined to $\Pi x : \texttt{int}.\{y : \texttt{int} \mid y = x + 1\}$ (since from the calling context, we know that the actual return value of f is important for the whole program to be typed). For another example, consider the sorting function `isort` above. The auxiliary function `insert` is first given a type `int list` $\rightarrow$ `int list`. If the type of `sort` is declared as `int list` $\rightarrow$ `int ordlist` (where `int ordlist` denotes the type of sorted lists), however, we can find from the call site `insert (x, isort xs’)` that the type of the output of `insert` should be `int ordlist`. We can then propagate that information backward to infer the type of an argument of `insert` (see Section 5 for a more detailed description of this refinement step). In this manner, we expect that our approach can deal with more flexible dependent types (without losing efficiency) than the size inference. Indeed, we have already implemented the prototype inference system and succeeded in verifying the sorting function above.

The idea of on-demand type refinement mentioned above, so called *type-error-guided type refinement*, has been inspired from that of counter-example-guided abstraction refinement (CEGAR) in abstract model checking [7]. In CEGAR, the coarsest abstraction is first used for model checking; the predicates used for abstraction are gradually refined when a false counter-example is encountered. In our approach, simple types are first used for type-checking. If the type-checking fails, types are gradually refined by inspecting a fragment of the program which causes the failure (until no further refinement is possible, when a type error is reported).

To formalize the idea mentioned above, Section 2 introduces a simple first-order functional language with assert expressions and a dependent type system for it. The assert expressions are used to model array bound checks and user-supplied specifications. Section 3 formalizes our type inference algorithm, and proves its soundness. In Section 4, we discuss extension of the type inference algorithm to deal with higher-order functions, parametric polymorphism, and algebraic data types. Section 5 reports on a prototype implementation of our algorithm (for the full language, including higher-order functions, parametric polymorphism, and algebraic data types) and experiments. Section 6 discusses related work and Section 7 concludes.

2 Language and Dependent Type System

We use a call-by-value, first-order functional language to present our type inference algorithm. We extend the language with higher-order functions in Section 4. The language is essentially an "implicitly-typed" version of a subset of DML [4, 5] extended with assert expressions.

The syntax of the language is defined as follows:

$$\begin{aligned}
\text{(expressions)}\ e &::= x \mid n \mid (e_1, e_2) \mid \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2 \mid f\ e \\
&\mid\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 \\
&\mid\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{assert}\ e_1\ \texttt{in}\ e_2 \\
\text{(values)}\ v &::= n \mid (v_1, v_2) \mid \texttt{fun}\ f\ x = e
\end{aligned}$$

Here, x, n, and f are meta-variables ranging over a set of variables, integer constants, and function names respectively. We write $\mathrm{FV}(e)$ for the set of free variables in e. We assume given primitive operators such as $+$, $\times$, $=$ and $\leq$ on integers, and $\neg$, $\wedge$, and $\Rightarrow$ on booleans. Actually, booleans are represented by integers (the truth $\top$ by a non-zero integer, and the false $\bot$ by zero). Thus, $e_1 \leq e_2$ returns 1 if the value of e_1 is less than or equal to that of e_2, and returns 0 otherwise. In the function definition $\texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2$, f may appear in e_1 for recursive calls. However, we do not allow mutually recursive functions in the language for the sake of simplicity. Our framework can be easily extended to deal with mutually recursive functions. An assertion $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ evaluates to e_2 only if the conditional e_1 holds. Otherwise, it gets stuck. Assertions are used for modeling array bounds errors and user-supplied specifications. For example, the array access $a[x]$ is modeled as $\texttt{assert}\ 0 \leq x < h\ \texttt{in}\ \cdots$, where h is the size of a. See Appendix B for the operational semantics.

We introduce a dependent type system, which ensures that well-typed programs never get stuck. In particular, an assertion $\texttt{assert}\ e_1\ \texttt{in}\ e_2$ is accepted only if e_1 is statically guaranteed to have a non-zero integer.

The syntax of types is defined as follows:

$$\begin{aligned}
&\text{(base types)}\ t ::= \texttt{int}^{\rho} \mid t_1 \times t_2 \quad \text{(function types)}\ \sigma ::= \forall \widetilde{\rho}. \langle \phi \mid t \rightarrow \tau \rangle \\
&\text{(expression types)}\ \tau ::= \{t \mid \phi\} \qquad \text{(constraints)}\ \phi ::= \rho \mid n \mid \texttt{op}(\widetilde{\phi}) \mid \forall \rho.\phi \mid \exists \rho.\phi \\
&\text{(type environments)}\ \mathit{\Gamma} ::= \emptyset \mid \mathit{\Gamma}, x : t \mid \mathit{\Gamma}, f : \sigma
\end{aligned}$$

A constraint, denoted by ϕ, is an *index variable* ρ, a constant n, an operator expression $\texttt{op}(\widetilde{\phi})$, or quantifier expressions. We often write $\top$ for 1 and $\bot$ for 0. Note that the set of operators contains standard logical operators like $\wedge$ and $\neg$.

The base type $\texttt{int}^{\rho}$ is the type of an integer whose value is denoted by ρ. The base type $t_1 \times t_2$ is the type of pairs consisting of values with the types t_1 and t_2. The expression type $\{t \mid \phi\}$ is a subtype of t whose index variables are constrained by ϕ. For example, $\{\texttt{int}^{\rho_1} \times \texttt{int}^{\rho_2} \mid \rho_1 > \rho_2\}$ is the type of integer pairs whose first element is greater than the second element. The index variables in t are bound in $\{t \mid \phi\}$. The function type $\forall \widetilde{\rho}.\langle \phi \mid t \to \tau \rangle$ is the type of functions that take an argument of the type $\{t \mid \phi\}$ and return a value of the type τ. For example, $\langle \rho_1 > 0 \wedge \rho_2 > 0 \mid \texttt{int}^{\rho_1} \times \texttt{int}^{\rho_2} \to \{\texttt{int}^{\rho_3} \mid \rho_3 = \rho_1 + \rho_2\}\rangle$ is the type of functions that take a pair of positive integers as an argument, and return the sum of the integers. The index variables in t and $\widetilde{\rho}$ are bound in $\forall \widetilde{\rho}.\langle \phi \mid t \to \tau \rangle$. We often abbreviate $\forall \widetilde{\rho}.\langle \phi \mid t \to \tau \rangle$ as $\forall \widetilde{\rho}.\{t \mid \phi\} \to \tau$ if the index variables in t do not occur in τ and as $\forall \widetilde{\rho}.t \to \tau$ if $\phi \equiv \top$. We assume that α-conversion is implicitly performed so that bound variables are different from each other and free variables.

A typing judgment is of the form $\phi; \Gamma \vdash e : \tau$. It reads that on the assumption that index variables satisfy ϕ, the expression has type τ under the type environment Γ. For example, $\rho > 0; x : \texttt{int}^{\rho} \vdash x + 1 : \{\texttt{int}^{\rho'} \mid \rho' > 1\}$.

The typing rules are given in Figure 1. In the figure, $\mathrm{FIV}(o)$ is the set of free index variables in some object o. $\eta \models \phi$ means that an index environment η (a function from index variables to integers) satisfies a constraint ϕ. We write $\models \phi$ if $\emptyset \models \forall \widetilde{\rho}.\phi$, where $\{\widetilde{\rho}\} = \mathrm{FIV}(\phi)$.

The subtyping relation $\phi \vdash \sigma \leqslant \sigma'$ on function types is defined by:

$$\frac{\models \phi \Rightarrow \forall \widetilde{\rho'}, \mathrm{FIV}(t_1).(\phi'_1 \Rightarrow \exists \widetilde{\rho}.(\phi_1 \wedge \forall \mathrm{FIV}(t_2).(\phi_2 \Rightarrow \phi'_2)))}{\phi \vdash \forall \widetilde{\rho}.\langle \phi_1 \mid t_1 \to \{t_2 \mid \phi_2\}\rangle \leqslant \forall \widetilde{\rho'}.\langle \phi'_1 \mid t_1 \to \{t_2 \mid \phi'_2\}\rangle}$$

We say that Γ is valid if and only if for any $f : \sigma \in \Gamma$ and v, if $\phi; \Gamma \vdash f\ v : \tau$ then, $[\![f]\!](v)$ is defined (i.e., f is a primitive operator) and $\phi; \Gamma \vdash [\![f]\!](v) : \tau$ is derivable (i.e., the type $\Gamma(f)$ captures the behavior of the primitive operator f correctly).

The refinement relation $\phi \vdash \Gamma \leqslant \Gamma'$ on type environments is defined as follows:

$$\frac{}{\phi \vdash \emptyset \leqslant \emptyset} \qquad \frac{\phi \vdash \Gamma \leqslant \Gamma'}{\phi \vdash \Gamma, x : t \leqslant \Gamma', x : t} \qquad \frac{\phi \vdash \Gamma \leqslant \Gamma' \qquad \phi \vdash \sigma \leqslant \sigma'}{\phi \vdash \Gamma, f : \sigma \leqslant \Gamma', f : \sigma'}$$

The type system ensures that evaluation of a well-typed program never gets stuck. Formally, the following theorem holds (see the Appendix C for the proof).

Theorem 1 (Soundness). *If* $\top; \Gamma \vdash e : \tau$ *is derivable,* $\mathrm{FV}(e) = \emptyset$*, and* Γ *is valid, then* e *either evaluates to a value or diverges.*

3 Type Inference Algorithm

This section formalizes our type inference algorithm and proves its soundness. First, we extend the syntax of constraints with predicate variables to denote

$$\frac{\begin{array}{c} x : t \in \Gamma \qquad \widetilde{\rho}' = \mathrm{FIV}(t) \\ \widetilde{\rho} \cap \mathrm{FIV}(\phi, \Gamma) = \emptyset \end{array}}{\phi; \Gamma \vdash x : \{[\widetilde{\rho}/\widetilde{\rho}']t \mid \widetilde{\rho} = \widetilde{\rho}'\}}\ (\textsc{T-Var})$$

$$\frac{\begin{array}{c} \phi; \Gamma \vdash e_1 : \{t \mid \phi'\} \\ \phi \wedge \phi'; \Gamma, x : t \vdash e_2 : \tau \\ \mathrm{FIV}(t) \cap \mathrm{FIV}(\tau) = \emptyset \end{array}}{\phi; \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau}\ (\textsc{T-Let})$$

$$\frac{}{\phi; \Gamma \vdash n : \{\mathtt{int}^{\rho} \mid \rho = n\}}\ (\textsc{T-Int})$$

$$\frac{\begin{array}{c} \phi; \Gamma \vdash e_1 : \{t_1 \times t_2 \mid \phi'\} \\ \phi \wedge \phi'; \Gamma, x_1 : t_1, x_2 : t_2 \vdash e_2 : \tau \\ \mathrm{FIV}(t_1, t_2) \cap \mathrm{FIV}(\tau) = \emptyset \end{array}}{\phi; \Gamma \vdash \mathtt{let}\ (x_1, x_2) = e_1\ \mathtt{in}\ e_2 : \tau}\ (\textsc{T-Let-Pair})$$

$$\frac{\begin{array}{c} \phi; \Gamma \vdash e_1 : \{t_1 \mid \phi_1\} \\ \phi; \Gamma \vdash e_2 : \{t_2 \mid \phi_2\} \end{array}}{\phi; \Gamma \vdash (e_1, e_2) : \{t_1 \times t_2 \mid \phi_1 \wedge \phi_2\}}\ (\textsc{T-Pair})$$

$$\frac{\begin{array}{c} \phi; \Gamma \vdash e_1 : \{\mathtt{int}^{\rho} \mid \phi'\} \\ \phi \wedge \exists\rho.(\phi' \wedge \rho \neq 0); \Gamma \vdash e_2 : \tau \\ \phi \wedge \exists\rho.(\phi' \wedge \rho = 0); \Gamma \vdash e_3 : \tau \end{array}}{\phi; \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}\ (\textsc{T-If})$$

$$\frac{\begin{array}{c} \phi \wedge \phi_1; \Gamma, f : \sigma, x : t_1 \vdash e_1 : \tau_1 \\ \widetilde{\rho} \cap \mathrm{FIV}(\Gamma, \phi) = \emptyset \\ \sigma = \forall\widetilde{\rho}.\langle \phi_1 \mid t_1 \to \tau_1 \rangle \\ \phi; \Gamma, f : \sigma \vdash e_2 : \tau_2 \end{array}}{\phi; \Gamma \vdash \mathtt{fun}\ f\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}\ (\textsc{T-Let-Fun})$$

$$\frac{\begin{array}{c} \phi; \Gamma \vdash e_1 : \{\mathtt{int}^{\rho} \mid \rho \neq 0\} \\ \phi; \Gamma \vdash e_2 : \tau \end{array}}{\phi; \Gamma \vdash \mathtt{assert}\ e_1\ \mathtt{in}\ e_2 : \tau}\ (\textsc{T-Assert})$$

$$\frac{\begin{array}{c} f : \sigma \in \Gamma \qquad \phi \vdash \sigma \leqslant \tau_1 \to \tau_2 \\ \phi; \Gamma \vdash e : \tau_1 \end{array}}{\phi; \Gamma \vdash f\ e : \tau_2}\ (\textsc{T-App})$$

$$\frac{\begin{array}{c} \phi_1'; \Gamma \vdash e : \{t \mid \phi_2'\} \\ \models \phi_1 \Rightarrow (\phi_1' \wedge (\phi_2' \Rightarrow \phi_2)) \end{array}}{\phi_1; \Gamma \vdash e : \{t \mid \phi_2\}}\ (\textsc{T-Sub})$$

Fig. 1. Typing Rules

unknown predicates. We also introduce *extended type environments* to model an intermediate state for on-demand type refinement.

$$\begin{array}{rl} \text{(type constraints)} & \phi ::= \cdots \mid P(\widetilde{\phi}) \\ \text{(constraint substitutions)} & S ::= \emptyset \mid S, P \mapsto \lambda\widetilde{\rho}.\phi \\ \text{(extended function types)} & T ::= (\sigma; \phi; \widetilde{S}) \\ \text{(extended type environments)} & \Delta ::= \emptyset \mid \Delta, x : t \mid \Delta, f : T \end{array}$$

Here, P is a meta-variable ranging over the set of predicate variables, which are used to express unknown specifications of functions. We write $\mathrm{FPV}(o)$ for the set of free predicate variables in some object o. Constraint substitutions map predicate variables to predicates (i.e., functions from index variables to constraints). An extended type environment Δ maps a function name f to an extended function type which is a triple of the form $(\sigma; \phi; \widetilde{S})$. Here, σ is a *template* for the type of f, which may contain predicate variables. For example, a template for a function from integers to integers is $\langle P(\rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\rho_x, \rho_y)\}\rangle$. The second element ϕ is a constraint that records a sufficient condition on predicate variables for the definition of f to be well-typed; this is used to avoid re-checking the function's definition when the function's type needs to be refined. The third element $\widetilde{S}$ records solutions for ϕ (which are substitutions for predicate variables) found so far.

The type inference algorithm is specified as inference rules for the 5-tuple relation $\mathit{\Delta} \rhd e : \tau \dashv \phi; \mathit{\Delta}'$. Here, $\mathit{\Delta}$, e, and τ should be regarded as inputs of the algorithm, and ϕ and $\mathit{\Delta}'$ as outputs of the algorithm. Intuitively, ϕ is a sufficient condition for e to have type τ, and $\mathit{\Delta}'$ describes types refined during the inference. For example, let e, τ, and $\mathit{\Delta}$ be $f(z)$, $\{\mathtt{int}^{\rho} \mid \rho > 1\}$, and $z : \mathtt{int}^{\rho_z}, f : (\sigma; \phi_1; \{S\})$, where:

$$\begin{aligned}
&\sigma = \langle P(\rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\rho_x, \rho_y)\}\rangle \\
&\phi_1 = \forall \rho_x, \rho_y . P(\rho_x) \Rightarrow (\rho_y = \rho_x + 1 \Rightarrow Q(\rho_x, \rho_y)) \\
&S = \{P \mapsto \lambda \rho_x . \top, Q \mapsto \lambda(\rho_x, \rho_y). \top\}
\end{aligned}$$

Then, ϕ and $\mathit{\Delta}'$ would be $\rho_z > 0$ and $z : \mathtt{int}^{\rho_z}, f : (\sigma; \phi; \{S, S'\})$, where S' is $\{P \mapsto \lambda \rho_x . \rho_x > 0, Q \mapsto \lambda(\rho_x, \rho_y) . \rho_y > 1\}$.

The inference rules for the relation $\mathit{\Delta} \rhd e : \tau \dashv \phi; \mathit{\Delta}'$ (which are a declarative description of our type inference algorithm) are given in Figures 2 and 3. Figure 3 shows the rules for function definitions and applications, and Figure 2 shows the rules for other expressions. In the figures, TypeOf$(\mathit{\Delta}, e)$ is a *template* for the type of e, obtained from the simple type of e by decorating it with fresh index variables and predicate variables. For example, if the simple type of e is $\mathtt{int}$, then TypeOf$(\mathit{\Delta}, e)$ returns $\mathtt{int}^{\rho}$; if the simple type of e is $\mathtt{int} \to \mathtt{int}$, TypeOf$(\mathit{\Delta}, e)$ returns $\langle P(\rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\rho_x, \rho_y)\}\rangle$.

In the rules in Figure 2, type inference proceeds in a backward manner: For example, in B-Var, given the required type $\{t \mid \phi\}$ of the variable x, if $x : t' \in \mathit{\Delta}$, we check whether $|t| = |t'|$ (where $|t|$ is the simple type obtained from t by removing index variables and constraints). If the check succeeds, we produce the constraint $[t'/t]\phi$, which is the constraint obtained from ϕ by replacing each occurrence of an index variable of t with the corresponding index variable of t'.

In B-Pair, given the required type $\{t_1 \times t_2 \mid \phi\}$ of the pair (e_1, e_2), we compute the constraint ϕ_2 which is sufficient for e_2 to have $\{t_2 \mid \phi\}$. Then, we compute the constraint ϕ_1 which is sufficient for e_1 to have $\{t_1 \mid \phi_2\}$. The remaining rules in Figure 2 can be read in a similar manner.

We now explain the rules for functions in Figure 3. In B-Let-Fun, a template for the function's type is first prepared (see the first line). We then check the function's definition, and compute a sufficient condition ψ on predicate variables for the definition to be well-typed (see the second line). Then, we find a solution S for ψ (i.e., a substitution such that $\models S(\psi)$) by using an auxiliary algorithm Solve(FPV(σ); ψ), which is explained later. As a result, we obtain the input specification of f which is sufficient for no assertion violation to occur in f. At this stage, there is no requirement for the output of f, so that the inferred return type of f is of the form $\{t \mid \top\}$. Finally, we check e_2 and produce ϕ_2 and $\mathit{\Delta}'$. Note that f's type may be refined during the type inference for e_2.

B-App is the rule for applications. From the type τ of $f\,e$ and the simple type of e, we prepare a template of f's type: $\{t \mid P(\widetilde{\rho})\} \to \tau$. The value of the predicate variable P is computed by a sub-algorithm, expressed by using the relation $\mathit{\Delta} \rhd f : \sigma \dashv_{\{P\}} S; \mathit{\Delta}'$ (which is defined using B-Reuse and B-Refine: see below). Finally, we check that the function's argument e has the required type $\{t \mid S(P(\widetilde{\rho}))\}$.

We have two rules B-Reuse and B-Refine for the auxiliary judgment $\Delta \rhd f : \sigma \dashv_{\widetilde{P}} S; \Delta'$. The rule B-Reuse supports the case where the type of f in Δ is precise enough to be a subtype of σ, while B-Refine supports the case where the type of f needs to be refined. The rules are non-deterministic, in the sense that both rules may be applied. In the actual implementation, B-Reuse is given a higher priority, so that B-Refine is used only when applications of B-Reuse fail. For recursive calls and primitive operators, B-Refine is not used.

In B-Reuse, we pick up an already inferred type $S_k(\sigma')$, and match it with the required type σ. (Since the argument type of σ is a predicate variable, we actually match the return types of σ and σ' here.) The constraint ψ, computed by using B-Sub, is a sufficient condition for $S_k(\sigma')$ to be a subtype of σ. We then solve ψ by using Solve.

In B-Refine, we match the template σ' of the function's type with the required type σ, and compute a sufficient condition ψ for σ' to be a subtype of σ. We then compute a solution for $\psi \wedge \phi$ by using Solve. The key point here is that both information about the function's definition (expressed by ϕ) and that about the call site (expressed by ψ) are used to compute the function's type. Solve can use predicates occurring in ψ as hints for computing a solution of $\psi \wedge \phi$.

$$\frac{x : t' \in \Delta \qquad |t| = |t'|}{\Delta \rhd x : \{t \mid \phi\} \dashv [t'/t]\phi; \Delta}\ \text{(B-Var)}$$

$$\frac{}{\Delta \rhd n : \{\texttt{int}^{\rho} \mid \phi\} \dashv [n/\rho]\phi; \Delta}\ \text{(B-Int)}$$

$$\frac{\begin{array}{c}\Delta \rhd e_2 : \{t_2 \mid \phi\} \dashv \phi_2; \Delta_2 \\ \Delta_2 \rhd e_1 : \{t_1 \mid \phi_2\} \dashv \phi_1; \Delta_1\end{array}}{\Delta \rhd (e_1, e_2) : \{t_1 \times t_2 \mid \phi\} \dashv \phi_1; \Delta_1}\ \text{(B-Pair)}$$

$$\frac{\begin{array}{c} t = \mathrm{TypeOf}(\Delta, e_1) \\ \Delta, x : t \rhd e_2 : \tau \dashv \phi_2; \Delta_2 \\ \Delta_2 \setminus x \rhd e_1 : \{t \mid \phi_2\} \dashv \phi_1; \Delta_1\end{array}}{\Delta \rhd \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \dashv \phi_1; \Delta_1}\ \text{(B-Let)}$$

$$\frac{\begin{array}{c} t_1 \times t_2 = \mathrm{TypeOf}(\Delta, e_1) \\ \Delta, x_1 : t_1, x_2 : t_2 \rhd e_2 : \tau \dashv \phi_2; \Delta_2 \\ \Delta_2 \setminus \{x_1, x_2\} \rhd e_1 : \{t_1 \times t_2 \mid \phi_2\} \dashv \phi_1; \Delta_1\end{array}}{\Delta \rhd \texttt{let}\ (x_1, x_2) = e_1\ \texttt{in}\ e_2 : \tau \dashv \phi_1; \Delta_1}\ \text{(B-Let-Pair)}$$

$$\frac{\begin{array}{c} \Delta \rhd e_2 : \tau \dashv \phi_2; \Delta_2 \qquad \Delta_2 \rhd e_3 : \tau \dashv \phi_3; \Delta_3 \\ \rho : \text{fresh} \qquad \phi = (\rho \neq 0 \wedge \phi_2) \vee (\rho = 0 \wedge \phi_3) \\ \Delta_3 \rhd e_1 : \{\texttt{int}^{\rho} \mid \phi\} \dashv \phi_1; \Delta_1\end{array}}{\Delta \rhd \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \dashv \phi_1; \Delta_1}\ \text{(B-If)}$$

$$\frac{\begin{array}{c} \rho : \text{fresh} \qquad \Delta \rhd e_1 : \{\texttt{int}^{\rho} \mid \rho \neq 0\} \dashv \phi_1; \Delta_1 \\ \Delta_1 \rhd e_2 : \tau \dashv \phi_2; \Delta_2\end{array}}{\Delta \rhd \texttt{assert}\ e_1\ \texttt{in}\ e_2 : \tau \dashv \phi_1 \wedge \phi_2; \Delta_2}\ \text{(B-Assert)}$$

Fig. 2. Type inference rules (for basic expressions)

Constraint Solving We now describe a heuristic algorithm $\mathrm{Solve}(\widetilde{P}; \varphi)$ to obtain a solution for φ (i.e., a substitution for the predicate variables $\widetilde{P}$ that satisfy φ).

If φ contains a subformula of the form $\forall\widetilde{\rho}.(P(\widetilde{\rho}) \Rightarrow \psi(\widetilde{\rho}, P))$, and $\psi(\widetilde{\rho}, P)$ does not contain negative occurrences of P, then the algorithm tries to compute the greatest fixed-point of $F = \lambda P.\lambda\widetilde{\rho}.\psi(\widetilde{\rho}, P)$ by iterations from $\lambda\rho_x.\top$ (i.e., by computing $F^n(\lambda\rho_x.\top)$ for $n = 1, 2, \ldots$). (As a special case, if $\psi(\widetilde{\rho}, P)$ does

$$\frac{\begin{array}{c}\sigma = \forall\widetilde{\rho}.\langle \phi \mid t \to \tau_1\rangle = \mathrm{TypeOf}(\varDelta, \mathtt{fun}\ f\ x = e_1)\\ \varDelta, f:\sigma, x:t \rhd e_1 : \tau_1 \dashv \phi_1; \varDelta_1, f:\sigma, x:t \qquad \psi = \forall\widetilde{\rho}, \mathrm{FIV}(t).(\phi \Rightarrow \phi_1)\\ S = \mathrm{Solve}(\mathrm{FPV}(\sigma);\psi) \qquad \varDelta_1, f:(\sigma;\psi;\{S\}) \rhd e_2 : \tau \dashv \phi_2; \varDelta_2\end{array}}{\varDelta \rhd \mathtt{fun}\ f\ x = e_1\ \mathtt{in}\ e_2 : \tau \dashv \phi_2; \varDelta_2 \setminus f} \quad (\text{B-Let-Fun})$$

$$\frac{\begin{array}{c} t = \mathrm{TypeOf}(\varDelta, e) \qquad \widetilde{\rho} = \mathrm{FIV}(t) \qquad P : \mathrm{fresh}\\ \varDelta \rhd f : \{t \mid P(\widetilde{\rho})\} \to \tau \dashv_{\{P\}} S; \varDelta_1 \qquad \varDelta_1 \rhd e : \{t \mid S(P(\widetilde{\rho}))\} \dashv \phi_2; \varDelta_2\end{array}}{\varDelta \rhd f\ e : \tau \dashv \phi_2; \varDelta_2} \quad (\text{B-App})$$

$$\frac{f : (\sigma'; \phi; \{S_j\}_{j=1}^{m}) \in \varDelta \qquad 1 \le k \le m \qquad S_k(\sigma') \leqslant \sigma \dashv \psi \qquad S = \mathrm{Solve}(\widetilde{P};\psi)}{\varDelta \rhd f : \sigma \dashv_{\widetilde{P}} S; \varDelta} \quad (\text{B-Reuse})$$

$$\frac{\begin{array}{c} \varDelta = \varDelta_b, f : (\sigma'; \phi; \{S_j\}_{j=1}^{m}), \varDelta_a \qquad \sigma' \leqslant \sigma \dashv \psi\\ \mathrm{dom}(S) = \widetilde{P} \qquad \mathrm{dom}(S_{m+1}) = \mathrm{FPV}(\sigma') \qquad S, S_{m+1} = \mathrm{Solve}(\widetilde{P} \cup \mathrm{FPV}(\sigma'); \psi \wedge \phi)\end{array}}{\varDelta \rhd f : \sigma \dashv_{\widetilde{P}} S; \varDelta_b, f : (\sigma'; \phi; \{S_j\}_{j=1}^{m+1}), \varDelta_a} \quad (\text{B-Refine})$$

$$\frac{\phi = \forall\widetilde{\rho'}, \mathrm{FIV}(t_1).(\phi'_1 \Rightarrow \exists\widetilde{\rho}.(\phi_1 \wedge \forall\mathrm{FIV}(t_2).(\phi_2 \Rightarrow \phi'_2)))}{\forall\widetilde{\rho}.\langle \phi_1 \mid t_1 \to \{t_2 \mid \phi_2\}\rangle \leqslant \forall\widetilde{\rho'}.\langle \phi'_1 \mid t_1 \to \{t_2 \mid \phi'_2\}\rangle \dashv \phi} \quad (\text{B-Sub})$$

Fig. 3. Type inference rules (for functions)

not contain P, then the iteration immediately converges with the solution $P = \lambda\rho_x.\psi(\widetilde{\rho}, P)$.) The algorithm also use widening [8] to accelerate convergence.

If the above iteration does not converge, the algorithm chooses a new starting point of iterations by extracting a sub-formula of $\psi(\widetilde{\rho}, P)$ which does not contain P and generalizing its constants. (This phase roughly corresponds to predicate discovery in abstract model checking. Unlike model checking, however, we do not repeat the whole verification process; we just redo the fixed-point computation.)

Example 1. $\mathtt{fun\ pred}\ x = \mathtt{assert}\ x > 0\ \mathtt{in}\ x - 1\ \mathtt{in\ assert}\ y = \mathtt{pred}\ z\ \mathtt{in}\ ()$

By B-Let-Fun, we first check the definition of `pred`. We prepare the template $\sigma = \forall\widetilde{\rho}.\langle P(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\}\rangle$ for the type of `pred`, where $\widetilde{\rho}$ denotes a sequence of index variables (whose length is unknown). Then we check $\varDelta \rhd \mathtt{assert}\ x > 0\ \mathtt{in}\ x - 1 : \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\} \dashv \phi'; \varDelta'$ for $\varDelta = \varDelta_0, \mathtt{pred} : \sigma$, and obtain $\phi' = \rho_x > 0 \wedge Q(\widetilde{\rho}, \rho_x, \rho_x - 1)$. Here, $\varDelta_0 = + : \langle \top \mid \mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \to \{\mathtt{int}^{\rho_3} \mid \rho_3 = \rho_1 + \rho_2\}\rangle, \ldots, \le : \langle \top \mid \mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \to \{\mathtt{int}^{\rho_3} \mid \rho_3 = \rho_1 \le \rho_2\}\rangle, \ldots$ is the extended type environment for primitive operators. Thus, we obtain the constraint $\phi = \forall\widetilde{\rho}, \rho_x.P(\widetilde{\rho}, \rho_x) \Rightarrow \phi'$ on P and Q. We then check $\mathtt{assert}\ y = \mathtt{pred}\ z\ \mathtt{in}\ ()$ under $\varDelta_1 = \varDelta_0, \mathtt{pred} : (\sigma; \phi; \{P \mapsto \lambda\widetilde{\rho}, \rho_x.\rho_x > 0, Q \mapsto \lambda\widetilde{\rho}, \rho_x, \rho_y.\top\})$.
To check $\mathtt{pred}\ z$ against the type $\{\mathtt{int}^{\rho_y} \mid \rho = \rho_y\}$, the rule B-Refine is used. From $\sigma \leqslant \forall\widetilde{\rho}.\langle P_1(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid \rho = \rho_y\}\rangle \dashv \psi$, we get

$$\psi = \forall\widetilde{\rho}, \rho_x.P_1(\widetilde{\rho}, \rho_x) \Rightarrow \exists\widetilde{\rho'}.(P(\widetilde{\rho'}, \rho_x) \wedge \forall\rho_y.(Q(\widetilde{\rho'}, \rho_x, \rho_y) \Rightarrow \rho = \rho_y)).$$

Then, $\psi \wedge \phi$ is passed to Solve as an input. From the subformula $Q(\widetilde{\rho'}, \rho_x, \rho_y) \Rightarrow \rho = \rho_y)$, Solve infers that $Q(\rho, \rho_x, \rho_y) \equiv \rho = \rho_y$. From the subformula ϕ, $P(\rho, \rho_x)$ is inferred to be $\rho_x > 0 \wedge \rho = \rho_x - 1$. Thus, we obtain the refined type $\forall \rho.\langle \rho_x > 0 \wedge \rho = \rho_x - 1 \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid \rho = \rho_y\}\rangle$ of `pred`.

Example 2.

$\mathtt{fun}\ \mathit{fact}\ x = \mathtt{if}\ x \leq 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * \mathit{fact}\ (x-1)\ \mathtt{in}\ \mathtt{assert}\ \mathit{fact}\ y > 0\ \mathtt{in}\ ()$

By B-Let-Fun, we first check the definition of `fact`. We prepare the template $\sigma = \forall \widetilde{\rho}.\langle P(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\}\rangle$ for the type of `fact`, where $\widetilde{\rho}$ denotes a sequence of index variables (whose length is unknown). Then we check $\Delta \rhd \mathtt{if}\ x \leq 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * \mathit{fact}\ (x-1) : \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\} \dashv \phi'; \Delta'$ for $\Delta = \Delta_0, \mathtt{fact} : \sigma$, and obtain $\phi' = (\rho_x \leq 0 \wedge \phi_1) \vee (\rho_x > 0 \wedge \phi_2)$. Here, $\phi_1 = Q(\widetilde{\rho}, \rho_x, 1)$ and $\phi_2 = \exists \widetilde{\rho'}.(P(\widetilde{\rho'}, \rho_x - 1) \wedge \forall \rho_y.(Q(\widetilde{\rho'}, \rho_x - 1, \rho_y) \Rightarrow Q(\widetilde{\rho}, \rho_x, \rho_x * \rho_y)))$ are respectively obtained from the then- and else- branches of the if-expression by B-If. B-App and B-Reuse are used to generate the subformula ϕ_2 for the else-branch. Thus, we obtain the constraint $\phi = \forall \widetilde{\rho}, \rho_x.P(\widetilde{\rho}, \rho_x) \Rightarrow \phi'$ on P and Q. We then check $\mathtt{assert\ fact}\ y > 0\ \mathtt{in}\ ()$ under $\Delta_1 = \Delta_0, \mathtt{fact} : (\sigma; \phi; \{P \mapsto \lambda \widetilde{\rho}, \rho_x.\top, Q \mapsto \lambda \widetilde{\rho}, \rho_x, \rho_y.\top\})$.
To check $\mathtt{fact}\ y$ against the type $\{\mathtt{int}^{\rho_y} \mid \rho_y > 0\}$, the rule B-Refine is used. From $\sigma \leqslant \forall \widetilde{\rho}.\langle P_1(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid \rho_y > 0\}\rangle \dashv \psi$, we get

$$\psi = \forall \widetilde{\rho}, \rho_x.P_1(\widetilde{\rho}, \rho_x) \Rightarrow \exists \widetilde{\rho'}.(P(\widetilde{\rho'}, \rho_x) \wedge \forall \rho_y.(Q(\widetilde{\rho'}, \rho_x, \rho_y) \Rightarrow \rho_y > 0)).$$

Then, $\psi \wedge \phi$ is passed to Solve as an input. From the subformula $Q(\widetilde{\rho'}, \rho_x, \rho_y) \Rightarrow \rho_y > 0)$, Solve infers that $Q(\rho_x, \rho_y) \equiv \rho_y > 0$. From the subformula ϕ, $P(\rho_x)$ is inferred to be $\top$ as the result of the greatest fixed-point computation of the function $F = \lambda P.\lambda \rho_x.(\rho_x \leq 0 \wedge 1 > 0) \vee (\rho_x > 0 \wedge P(\rho_x - 1) \wedge \forall \rho_y.(\rho_y > 0 \Rightarrow \rho_x * \rho_y > 0)) \equiv \lambda P.\lambda \rho_x.\rho_x \leq 0 \vee (\rho_x > 0 \wedge P(\rho_x - 1))$ by iterations from $\lambda \rho_x.\top$, which converge immediately since $F(\lambda \rho_x.\top) \equiv \lambda \rho_x.\rho_x \leq 0 \vee \rho_x > 0 \equiv \lambda \rho_x.\top$. Thus, we obtain the refined type $\langle \top \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid \rho_y > 0\}\rangle$ of `fact`.

3.1 Soundness

We say that Δ is valid if and only if for any $f : (\sigma; \phi; \{S_j\}_{j=1}^m) \in \Delta$, $\models S_k(\phi)$ holds for any $k \in \{1, \ldots, m\}$.

Let us define the function $(\!|\Delta|\!)$, which maps an extended type environment Δ to an ordinary type environment, as follows:

$$(\!|\emptyset|\!) = \emptyset \qquad (\!|\Delta, x : t|\!) = (\!|\Delta|\!), x : t$$
$$(\!|\Delta, f : (\sigma; \phi; \{S_j\}_{j=1}^m)|\!) \ = \ (\!|\Delta|\!), f : \mathrm{merge}(\{S_j(\sigma)\}_{j=1}^m).$$

Here, $\mathrm{merge}(\{\sigma_j\}_{j=1}^m) = \langle \phi_1 \vee \cdots \vee \phi_m \mid t \to \{t' \mid (\phi_1 \Rightarrow \phi'_1) \wedge \cdots \wedge (\phi_m \Rightarrow \phi'_m)\}\rangle$ if $\sigma_j = \langle \phi_j \mid t \to \{t' \mid \phi'_j\}\rangle$ for any $j \in \{1, \ldots, m\}$. The following theorem states that the type inference algorithm is sound with respect to the dependent type system presented in Section 2. (We assume the soundness of Solve here; see Appendix D for the proof).

Theorem 2 (Soundness). *If $\Delta \rhd e : \tau \dashv \phi; \Delta'$ is derivable and Δ is valid then, Δ' is valid, $\vdash (\!|\Delta'|\!) \leqslant (\!|\Delta|\!)$, and $\phi; (\!|\Delta'|\!) \vdash e : \tau$ is derivable.*

Theorems 1 and 2 imply that if the type inference algorithm returns a type of an expression and a refined type environment Δ such that $(\!|\Delta|\!)$ is valid, then the evaluation of the expression never gets stuck (in particular, assertions in the expression are never violated).

Note that the type inference algorithm is *not* complete with respect to the type system because of the incompleteness of Solve.

4 Extensions

In this section, we discuss how to extend our type inference algorithm formalized in Section 3 with higher-order functions, parametric polymorphism, and algebraic data types.

Higher-Order Functions A main new issue in handling higher-order functions is what kind of template is prepared for higher-order functions. For example, for a function of type ($\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$, one may be tempted to consider a template of the form: $\langle R_1(P_1, Q_1) \mid \langle P_1(\rho_1) \mid \mathtt{int}^{\rho_1} \to \{\mathtt{int}^{\rho_2} \mid Q_1(\rho_1, \rho_2)\}\rangle \to \{\mathtt{int}^{\rho_3} \mid R_2(P_1, Q_1, \rho_3)\}\rangle$, which is the type of a function that takes a function whose precondition P_1 and postcondition Q_1 satisfy $R_1(P_1, Q_1)$, and returns an integer that satisfies $R_2(P_1, Q_1, \rho_3)$. This allows us to express a higher-order function that is polymorphic on the property of a function argument, but requires a significant extension of the constraint solving algorithm due to the presence of higher-order predicates.

Instead, we consider only first-order predicate variables, and use a template $\langle P_1(\rho_1) \mid \mathtt{int}^{\rho_1} \to \{\mathtt{int}^{\rho_2} \mid Q_1(\rho_1, \rho_2)\}\rangle \to \{\mathtt{int}^{\rho_3} \mid Q_2(\rho_3)\}$ for $(\mathtt{int} \to \mathtt{int}) \to \mathtt{int}$. This allows us to extend the algorithm in Section 3 in a fairly straightforward manner. A shortcoming of the approach is that a higher-order function is monomorphic on the property of function arguments; we use parametric polymorphism to overcome that disadvantage to some extent.

Parametric Polymorphism The above treatment of higher-order functions sometimes results in too specific types. For example, from the calling context $(\mathtt{map}\ (\lambda x.x+1)\ l) : \{\mathtt{int}^w\ \mathtt{list} \mid w \geq 0\}$, the following type of map would be inferred:

$$(\{\mathtt{int}^x \mid x \geq -1\} \to \{\mathtt{int}^y \mid y \geq 0\}) \to \{\mathtt{int}^z\ \mathtt{list} \mid z \geq -1\} \to \{\mathtt{int}^w\ \mathtt{list} \mid w \geq 0\}.$$

This is too specific to be used in other calling contexts of map. To remedy the problem, we use parametric polymorphism. In the case of map function, the polymorphic type $\forall \alpha, \beta.(\alpha \to \beta) \to \alpha\ \mathtt{list} \to \beta\ \mathtt{list}$ is assigned to map, which can be instantiated to $(\{\mathtt{int}^x \mid P(x)\} \to \{\mathtt{int}^y \mid Q(y)\}) \to \{\mathtt{int}^z\ \mathtt{list} \mid P(z)\} \to \{\mathtt{int}^w\ \mathtt{list} \mid Q(w)\}$ for any P and Q.

Algebraic Data Types We require users to declare a data type invariant and dependent types for constructors of each user-defined algebraic data type as in DML. Then, our algorithm infers dependent types of functions automatically unlike in DML. We allow users to declare *multiple* types for each data constructor; for example, for lists, users may declare `Nil` as $\forall\alpha.\mathtt{unit} \rightarrow \{\alpha\ \mathtt{list}^{\rho} \mid \rho = 0\}$ and $\forall\rho.\mathtt{unit} \rightarrow \{\mathtt{ordlist}^{\rho_1} \mid \rho_1 = \rho\}$ (see Section 5.1). This allows users to specify multiple properties like the list length and sortedness.

The main new difficulty in type inference is how to handle multiple types declared for each constructor as mentioned above. An extended type environment Δ now maps each function name to *a set of* extended function types, instead of a single extended function type. For example, for a list function, the following four templates may be generated: $\{\ \langle P_1(\rho_x) \mid \mathtt{int\ list}^{\rho_x} \rightarrow \{\mathtt{int\ list}^{\rho_y} \mid Q_1(\rho_x, \rho_y)\}\rangle, \langle P_2(\rho_x) \mid \mathtt{int\ list}^{\rho_x} \rightarrow \{\mathtt{ordlist}^{\rho_y} \mid Q_2(\rho_x, \rho_y)\}\rangle, \langle P_3(\rho_x) \mid \mathtt{ordlist}^{\rho_x} \rightarrow \{\mathtt{int\ list}^{\rho_y} \mid Q_3(\rho_x, \rho_y)\}\rangle, \langle P_4(\rho_x) \mid \mathtt{ordlist}^{\rho_x} \rightarrow \{\mathtt{ordlist}^{\rho_y} \mid Q_4(\rho_x, \rho_y)\}\rangle\}$. These templates are generated on-demand (based on calling contexts), in order to avoid a combinatorial explosion of the number of templates. Once an appropriate template is chosen, the rest of the algorithm is basically the same as the one described in Section 3: constraints on predicate variables are generated and solved.

4.1 Formalization

We formalize the extensions described above in this section.

The syntax of expressions, base types, function types, and extended type environments are extended as follows:

$$\begin{aligned}
\text{(expressions) } e &::= \cdots \mid \mathtt{fun}\ f\ \widetilde{f}\ x = e_1\ \mathtt{in}\ e_2 \mid f\ \widetilde{f}\ e \\
&\quad \mid\ () \mid c\ e \mid \mathtt{match}\ e\ \mathtt{with}\ \{c_j\ x_j \rightarrow e_j\}_{j=1}^{m} \\
\text{(base types) } t &::= \cdots \mid \alpha \mid \mathtt{unit} \mid \widetilde{t}\ d^{\widetilde{\rho}} \\
\text{(function types) } \sigma &::= \cdots \mid \forall\widetilde{\alpha}.\widetilde{\sigma} \rightarrow \forall\widetilde{\rho}.\langle\phi \mid t \rightarrow \tau\rangle \\
\text{(extended type environments) } \Delta &::= \emptyset \mid \Delta, x : t \mid \Delta, f : \widetilde{T}
\end{aligned}$$

Here, c, α, and d are meta-variables ranging over a set of names of user-defined data constructors, a set of type variables, and a set of names of user-defined algebraic data types respectively.

The function definition $\mathtt{fun}\ f\ \widetilde{f}\ x = e_1\ \mathtt{in}\ e_2$ defines a function f which takes zero or more function arguments $\widetilde{f}$ and an expression argument x. The function application $f\ \widetilde{f}\ e$ applies the function f to the actual function arguments $\widetilde{f}$ and the actual expression argument e. We do not allow partial applications in our language for the sake of simplicity. The restriction causes no loss of generality since they can be encoded in our language. For example, the expression $\mathtt{let}\ \mathit{add} = \lambda x.\lambda y.x + y\ \mathtt{in}\ (\mathit{add}\ 1)\ 2$ can be encoded as $\mathtt{fun}\ \mathit{add}\ (x, y) = x + y\ \mathtt{in}\ \mathtt{fun}\ \mathit{add_1}\ y = \mathit{add}\ (1, y)\ \mathtt{in}\ \mathit{add_1}\ 2$ in our language. Actually, it is not difficult to formalize our method for a language without the restriction. The syntax of expressions is also extended with a unit primitive, and introduction and elimination forms for algebraic data types. The algebraic data type $\widetilde{t}\ d^{\widetilde{\rho}}$ is parametrized by the base types $\widetilde{t}$ and the index variables $\widetilde{\rho}$. We write $\mathrm{CN}(d)$ for the set of the constructor names and $\iota(d^{\widetilde{\rho}})$ for the data type invariant for d expressed as a constraint on $\widetilde{\rho}$. We define the type invariant $\iota(t)$ of t as follows:

$$\begin{aligned}
\iota(t) &= \top \quad (\text{if } t = \alpha, \mathtt{unit}, \mathtt{int}^{\rho}) \\
\iota(t_1 \times t_2) &= \iota(t_1) \wedge \iota(t_2) \\
\iota(\widetilde{t}\ d^{\widetilde{\rho}}) &= \iota(\widetilde{t}) \wedge \iota(d^{\widetilde{\rho}}) \\
\iota(t_1, \dots, t_m) &= \iota(t_1) \wedge \cdots \wedge \iota(t_m)
\end{aligned}$$

We assume that type assignments for constructors of user-defined algebraic data types are added to the initial type environment $\varDelta_0$. Namely, we regard constructors as built-in functions. For example, we may define the algebraic data type $\alpha\ \mathtt{list}^{\rho}$ of polymorphic lists whose lengths are referred to by ρ as follows. The algebraic data type has the two constructors $\mathtt{Nil} : \forall\alpha.\mathtt{unit} \to \{\alpha\ \mathtt{list}^{\rho} \mid \rho = 0\}$ and $\mathtt{Cons} : \forall\alpha.\alpha \times \alpha\ \mathtt{list}^{\rho_1} \to \{\alpha\ \mathtt{list}^{\rho_2} \mid \rho_2 = \rho_1 + 1\}$, and has the data type invariant $\iota(\mathtt{list}^{\rho}) = \rho \geq 0$, which is obvious from the types of the constructors. The function type $\forall\widetilde{\alpha}.\widetilde{\sigma} \to \forall\widetilde{\rho}.\langle\phi \mid t \to \tau\rangle$ with the type parameters $\widetilde{\alpha}$ is the type of polymorphic functions which take function arguments of the types $\widetilde{\sigma}$ and an expression argument of the type $\{t \mid \phi\}$, and returns a value of the type τ.

The extended type inference rules are presented in Figure 4. In B-Let-Fun', the auxiliary function $\mathrm{TypeOf}(\varDelta, \mathtt{fun}\ f\ \widetilde{f}\ x = e_1)$ returns a set of all possible templates for f with only first-order predicate variables. Note that it is wasteful to generate all possible templates and corresponding sufficient conditions on predicate variables at function definition sites as in B-Let-Fun'. It is not difficult to generate them on-demand (based on calling contexts).

In B-App', we obtain the type t of the actual expression argument e and the function types $\widetilde{\sigma}$ of the actual function arguments $\widetilde{f}$ whose predicate variables are replaced with fresh ones by using auxiliary functions $\mathrm{TypeOf}(\varDelta, e)$ and $\mathrm{TypeOf}(\varDelta, \widetilde{f})$ respectively. Since $\mathrm{TypeOf}(\varDelta, o)$ returns all possible types of o, we try all possible combination of t and $\widetilde{\sigma}$ in order.[3] We then guess that f has the type $\widetilde{\sigma} \to \{t \mid P(\widetilde{\rho})\} \to \tau$. We then generate and solve the sufficient condition for P and $\mathrm{FPV}(\widetilde{\sigma})$ by the auxiliary judgement $\varDelta \rhd \{f : \widetilde{\sigma} \to \{t \mid P(\widetilde{\rho})\} \to \tau, \widetilde{f} : \widetilde{\sigma}\} \dashv_{\{P\} \cup \mathrm{FPV}(\widetilde{\sigma})} S; \varDelta_1$ which internally applies B-Reuse' and B-Refine' for backward propagation and works as follows. We first propagate the output specification τ of the higher-order function f backward to mine output specifications of function arguments in $\widetilde{f}$. We then propagate them backward to obtain the input specifications of the functions. We mine output specifications of other function arguments in $\widetilde{f}$ by using the propagated input specifications, and repeat the above procedure until we obtain the input and output specifications of all the function arguments. Finally, we infer the input specification of f by using

[3] The number of all possible combination is not so large in our experience; For example, we did not confront to this kind of non-determinism for verification of sorting algorithms in Section 5.1.

$$\frac{\begin{array}{c}\{\sigma_j\}_{j=1}^{m} = \{\forall\widetilde{\alpha_j}.\widetilde{\sigma_j} \to \forall\widetilde{\rho_j}.\langle\phi_j \mid t_j \to \tau_j\rangle\}_{j=1}^{m} = \mathrm{TypeOf}(\mathit{\Delta}, \mathtt{fun}\ f\ \widetilde{f}\ x = e_1) \qquad \mathit{\Delta}_1 = \mathit{\Delta} \\ \mathit{\Delta}_j, f : \sigma_j, \widetilde{f} : \widetilde{\sigma}_j, x : t_j \rhd e_1 : \tau_j \dashv \phi'_j; \mathit{\Delta}_{j+1}, f : \sigma_j, \widetilde{f} : \widetilde{\sigma}_j, x : t_j \qquad \psi_j = \forall\widetilde{\rho_j}, \mathrm{FIV}(t_j).(\phi_j \Rightarrow \phi'_j) \\ S_j = \mathrm{Solve}(\mathrm{FPV}(\sigma_j); \psi_j) \qquad (j = 1, \ldots, m) \qquad \mathit{\Delta}_{m+1}, f : \{(\sigma_j; \psi_j; \{S_j\})\}_{j=1}^{m} \rhd e_2 : \tau \dashv \phi'; \mathit{\Delta}'\end{array}}{\mathit{\Delta} \rhd \mathtt{fun}\ f\ \widetilde{f}\ x = e_1\ \mathtt{in}\ e_2 : \tau \dashv \phi'; \mathit{\Delta}' \setminus f}$$
(B-Let-Fun')

$$\frac{\begin{array}{c}\widetilde{\sigma} \in \mathrm{TypeOf}(\mathit{\Delta}, \widetilde{f}) \qquad t \in \mathrm{TypeOf}(\mathit{\Delta}, e) \qquad \widetilde{\rho} = \mathrm{FIV}(t) \qquad P : \text{fresh} \\ \mathit{\Delta} \rhd \{f : \widetilde{\sigma} \to \{t \mid P(\widetilde{\rho})\} \to \tau, \widetilde{f} : \widetilde{\sigma}\} \dashv_{\{P\}\cup\mathrm{FPV}(\widetilde{\sigma})} S; \mathit{\Delta}_1 \\ \mathit{\Delta}_1 \rhd e : \{t \mid S(P(\widetilde{\rho}))\} \dashv \phi_2; \mathit{\Delta}_2\end{array}}{\mathit{\Delta} \rhd f\ \widetilde{f}\ e : \tau \dashv \phi_2; \mathit{\Delta}_2} \text{(B-App')}$$

$$\frac{\begin{array}{c}f : \{\widetilde{T}, (\sigma'; \phi; \{S_j\}_{j=1}^{m})\} \in \mathit{\Delta} \qquad 1 \le k \le m \qquad S_k(\sigma') \leqslant \sigma \dashv_{\widetilde{P'}} \psi \\ \mathrm{dom}(S) = \widetilde{P} \qquad \mathrm{dom}(S') = \widetilde{P'} \qquad S, S' = \mathrm{Solve}(\widetilde{P} \cup \widetilde{P'}; \psi)\end{array}}{\mathit{\Delta} \rhd f : \sigma \dashv_{\widetilde{P}} S; \mathit{\Delta}} \text{(B-Reuse')}$$

$$\frac{\begin{array}{c}\mathit{\Delta} = \mathit{\Delta}_b, f : \{\widetilde{T}, (\sigma'; \phi; \{S_j\}_{j=1}^{m})\}, \mathit{\Delta}_a \qquad \sigma' \leqslant \sigma \dashv_{\widetilde{P'}} \psi \\ \mathrm{dom}(S) = \widetilde{P} \qquad \mathrm{dom}(S') = \widetilde{P'} \qquad \mathrm{dom}(S_{m+1}) = \mathrm{FPV}(\sigma') \\ S, S', S_{m+1} = \mathrm{Solve}(\widetilde{P} \cup \widetilde{P'} \cup \mathrm{FPV}(\sigma'); \psi \wedge \phi)\end{array}}{\mathit{\Delta} \rhd f : \sigma \dashv_{\widetilde{P}} S; \mathit{\Delta}_b, f : \{\widetilde{T}, (\sigma'; \phi; \{S_j\}_{j=1}^{m+1})\}, \mathit{\Delta}_a} \text{(B-Refine')}$$

$$\frac{\sigma \succ_{\widetilde{P_1}} \widetilde{\sigma} \to \forall\widetilde{\rho}.\{t_1 \mid \phi_1\} \to \{t_2 \mid \phi_2\} \qquad \sigma' = \forall\widetilde{\alpha'}.\widetilde{\sigma'} \to \forall\widetilde{\rho'}.\{t_1 \mid \phi'_1\} \to \{t_2 \mid \phi'_2\} \qquad \widetilde{\sigma'} \leqslant \widetilde{\sigma} \dashv_{\widetilde{P_2}} \phi}{\sigma \leqslant \sigma' \dashv_{\widetilde{P_1}\cup\widetilde{P_2}} \phi \wedge \forall\widetilde{\rho'}, \mathrm{FIV}(t_1).((\phi'_1 \wedge \iota(t_1)) \Rightarrow \exists\widetilde{\rho}.(\phi_1 \wedge \forall\mathrm{FIV}(t_2).((\phi_2 \wedge \iota(t_2)) \Rightarrow \phi'_2)))}$$
(B-Sub')

$$\frac{}{\mathit{\Delta} \rhd () : \{\mathtt{unit} \mid \phi\} \dashv \phi; \mathit{\Delta}} \text{(B-Unit)}$$

$$\frac{\begin{array}{c}\widetilde{t}\ d^{\widetilde{\rho}} \in \mathrm{TypeOf}(\mathit{\Delta}, e) \qquad \mathrm{CN}(d) = \{c_j\}_{j=1}^{m} \cup \{c_j\}_{j=m+1}^{m'} \\ \mathit{\Delta}_1 = \mathit{\Delta} \qquad c_j : \{\widetilde{T_j}, \sigma_j\} \in \mathit{\Delta}_j \qquad \sigma_j \succ_{\emptyset} \langle\phi_j^1 \mid t_j \to \{\widetilde{t}\ d^{\widetilde{\rho}} \mid \phi_j^2\}\rangle \\ \text{if } j \le m \text{ then } \mathit{\Delta}_j, x_j : t_j \rhd e_j : \tau \dashv \phi_j; \mathit{\Delta}_{j+1}, x_j : t_j \\ \text{else } \phi_j = \bot \text{ and } \mathit{\Delta}_{j+1} = \mathit{\Delta}_j \\ \phi'_j = \forall\mathrm{FIV}(t_j).((\phi_j^1 \wedge \iota(t_j) \wedge \phi_j^2) \Rightarrow \phi_j) \qquad (j = 1, \ldots, m') \\ \mathit{\Delta}_{m'+1} \rhd e : \{\widetilde{t}\ d^{\widetilde{\rho}} \mid \phi'_1 \wedge \cdots \wedge \phi'_{m'}\} \dashv \phi; \mathit{\Delta}'\end{array}}{\mathit{\Delta} \rhd \mathtt{match}\ e\ \mathtt{with}\ \{c_j\ x_j \to e_j\}_{j=1}^{m} : \tau \dashv \phi; \mathit{\Delta}'} \text{(B-Match)}$$

Fig. 4. The Extended Type Inference Rules

the output specification $\underset{\sim}{\tau}$ of f and the mined- or propagated- specifications of the function arguments $\widetilde{f}$.

In B-Reuse' and B-Refine', we pick up an extended function type $(\sigma'; \phi; \{S_j\}_{j=1}^m)$ for f from the type environment Δ which has the same shape as the function type σ required by the context.

In B-Sub', we first instantiate the possibly polymorphic type σ to the type of the form $\widetilde{\sigma} \to \forall\widetilde{\rho}.\{t_1 \mid \phi_1\} \to \{t_2 \mid \phi_2\}$ with parametricity in mind by a type instantiation relation $\sigma \succ_{\widetilde{P}} \widetilde{\sigma} \to \forall\widetilde{\rho}.\{t_1 \mid \phi_1\} \to \{t_2 \mid \phi_2\}$. The type instantiation may introduce fresh predicate variables $\widetilde{P}$, which are concretized later in B-Reuse' or B-Refine'. For example let us find ϕ and $\widetilde{P}$ such that $\forall\alpha.\alpha \to \alpha \leqslant \{\mathtt{int}^{\rho_1} \mid Q(\rho_1)\} \to \{\mathtt{int}^{\rho_2} \mid \rho_2 \geq 0\} \dashv_{\widetilde{P}} \phi$. We get $\forall\alpha.\alpha \to \alpha \succ_{\{P\}} \{\mathtt{int}^{\rho_1} \mid P(\rho_1)\} \to \{\mathtt{int}^{\rho_2} \mid P(\rho_2)\}$ for the fresh predicate variable P. Thus, we have $\widetilde{P} = \{P\}$ and $\phi \equiv \forall\rho_1.(Q(\rho_1) \Rightarrow (P(\rho_1) \wedge \forall\rho_2.(P(\rho_2) \Rightarrow \rho_2 \geq 0)))$. In B-Reuse' or B-Refine', we obtain $P \equiv \lambda\rho_2.\rho_2 \geq 0$ since ϕ contains the subformula $P(\rho_2) \Rightarrow \rho_2 \geq 0$. Thus, we get $(P \mapsto \lambda\rho_2.\rho_2 \geq 0)(\phi) \equiv \forall\rho_1.(Q(\rho_1) \Rightarrow \rho_1 \geq 0)$, and then $Q \equiv \lambda\rho_1.\rho_1 \geq 0$.

The type inference rule B-Unit for the unit primitive is straightforward. The rule for the introduction form $c\ e$ is the same as B-App'. The rule B-Match for the elimination form `match` e `with` $\{c_j\ x_j \to e_j\}_{j=1}^m$ requires that $\{c_j\}_{j=1}^m$ is a subset of $\mathrm{CN}(d)$, where d is the algebraic data type of e.

5 Implementation and Experiments

We have implemented a prototype type inference system (available from `http://web.yl.is.s.u-tokyo.ac.jp/~uhiro/depinf/`) according to the formalization in Section 3. It supports higher-order functions, parametric polymorphism, and algebraic data types as described in Section 4. We adopted Cooper's algorithm for checking satisfiability of integer constraints. We report two kinds of experiments to show the effectiveness of our approach. All the experiments were performed on Intel Xeon CPU 3GHz with 3GB RAM.

5.1 Verification of sorting algorithms

This experiment shows an application of our system to infer the specifications for auxiliary functions from the specification of the top-level function. The programs used in the experiment are the insertion sort defined in Section 1, and a merge sort. We discuss below the experiment for the insertion sort. The experiment for the merge sort is similar: The merge sort program consists of a main function `msort` and two auxiliary functions `merge` and `msplit`. The types of `merge` and `msplit` have been automatically inferred from only the type specification that `msort` should return a sorted list.

In the experiment, `Nil` is defined as a constructor having two types: $\forall\alpha.\mathtt{unit} \to \{\alpha\ \mathtt{list}^{\rho} \mid \rho = 0\}$ and $\forall\rho.\mathtt{unit} \to \{\mathtt{ordlist}^{\rho_1} \mid \rho_1 = \rho\}$. `Cons` is defined as a constructor having two types: $\forall\alpha.\alpha \times \alpha\ \mathtt{list}^{\rho_1} \to \{\alpha\ \mathtt{list}^{\rho_2} \mid \rho_2 = \rho_1 + 1\}$ and $\langle\rho_1 \leq \rho_2 \mid \mathtt{int}^{\rho_1} \times \mathtt{ordlist}^{\rho_2} \to \{\mathtt{ordlist}^{\rho_3} \mid \rho_3 = \rho_1\}\rangle$. Here, $\alpha\ \mathtt{list}^n$ is the type of lists of length n, whose elements have the type α. $\texttt{ordlist}^n$ is the type of ordered lists, whose elements are integers greater than or equal to n. As in this example, multiple types can be declared for each constructor in our system, and an appropriate type is chosen depending on each context. We also added a type declaration that `isort` should return a value of type $\{\texttt{ordlist}^\rho \mid \top\}$. Appendix A shows the whole code used in the experiment.

Our system succeeded in verifying the program, and inferred the following types in 0.912 seconds:

$$\begin{aligned}&\texttt{insert} : \forall\rho.\langle \rho \leq \rho_1 \land \rho \leq \rho_2 \mid \texttt{int}^{\rho_1} \times \texttt{ordlist}^{\rho_2} \rightarrow \{\texttt{ordlist}^{\rho_3} \mid \rho \leq \rho_3\}\rangle,\\ &\texttt{isort} : \texttt{int list} \rightarrow \texttt{ordlist}.\end{aligned}$$

The type of `insert` means that `insert` returns a sorted list whose head is greater than or equal to the first argument and the head of the second argument if a sorted list is given as the second argument.

We describe below how the type of the auxiliary function `insert` is refined. From the definition of `insert`, the initial type assigned to `insert` is $\texttt{int} \times \texttt{int list} \rightarrow \texttt{int list}$. When the call site $\texttt{insert}\ (x, \texttt{isort}\ xs')$ (on the last line of the definition of `isort`) is checked (with the required output specification $\{\texttt{ordlist}^\rho \mid \top\}$), the following new template for the type of `insert` is prepared:

$$\forall\rho.\langle P(\rho, \rho_1, \rho_2) \mid \texttt{int}^{\rho_1} \times \texttt{ordlist}^{\rho_2} \rightarrow \{\texttt{ordlist}^{\rho_3} \mid Q(\rho, \rho_1, \rho_2, \rho_3)\}\rangle,$$

Since the required type for $\texttt{insert}\ (x, \texttt{isort}\ xs')$ is $\{\texttt{ordlist}^\rho \mid \top\}$, the system first tries to let $Q(\rho, \rho_1, \rho_2, \rho_3)$ be $\top$, and checks the constraint extracted from the definition of `isort`. That type is, however, not precise enough to check the recursive call $\texttt{insert}(x, ys)$ (on the last line of the definition of `insert`), which requires that $\forall\rho_{ret}.Q(\rho', \rho_x, \rho_{ys}, \rho_{ret}) \Rightarrow \rho_y \leq \rho_{ret}$ holds. Thus, $Q(\rho, \rho_1, \rho_2, \rho_3)$ is strengthened to $\rho \leq \rho_3$. Then, the system successfully propagates the output specification backward to infer the input specification $P(\rho, \rho_1, \rho_2) \equiv \rho \leq \rho_1 \land \rho \leq \rho_2$ of `insert`.

5.2 Experiment with functions from the OCaml list module

In this experiment, we demonstrate an application of our system to learn specifications of library functions. We use the list module of the OCaml programming language (`http://caml.inria.fr/`) as the target of the experiment.

The experiment proceeded as follows.

1. We manually translated the source code of the list module into our language. We have also added the definition of list constructors $\texttt{Nil} : \forall\alpha.\texttt{unit} \rightarrow \{\alpha\ \texttt{list}^\rho \mid \rho = 0\}$ and $\texttt{Cons} : \forall\alpha.\alpha \times \alpha\ \texttt{list}^{\rho_1} \rightarrow \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho_1 + 1\}$.
2. We executed our system for the translated code above. No call site information was used in this phase (except for the calls inside libraries).
3. Let f be a function whose argument type constraint inferred in the previous step is not $\top$. (For example, the argument type of `combine` was inferred to be $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\}$ in Step 2.) Let g be another library

function. Then, we searched for code fragments of the form $f\ (\ldots g\ (\ldots)\ldots)$ from various application programs. (Here, we have used Google Code Search, `http://www.google.com/codesearch/`.)
4. We executed our system to the code fragments collected in the above step, to refine the types of library functions.

The first and third steps of the experiment have been conducted manually, but automation of those steps would not be difficult.

The result of the experiment is summarized in Table 1. A manually simplified version of the result is also presented in Table 2. This simplification process can be automated easily. Table 3 shows some of the call sites used in the final step. The filed "time" indicates the time spent in the second and fourth steps.

For most of the library functions, the inferred types are the same as the expected types (modulo simplification of some constraints). For some functions, the inferred types were less precise than expected: For example, the type of `rev_map2` in Table 1 does not capture the property that the length of the returned list is the same as that of the second argument. We expect that those types can be refined by using more appropriate call sites.

As for the efficiency, our system was slow for `length`, `map2`, and `combine`. We think that this is due to the present naive implementation of the fixed-point computation algorithm, and that we can remedy the problem by using convex-hull or selective hull operator [9] to keep the size of the constraints small.

As already mentioned, we have collected the call sites manually in step 3. To confirm that our choice of call sites did not much affect the quality of the inferred types, we have tested our system also with call sites other than those shown in Table 3, and confirmed that similar types are inferred from them.

We explain some of the results in Table 1. The types of `append` and `split` were refined at the call sites:

$$\begin{aligned}&\texttt{combine}\ (\texttt{append}\ (\texttt{fst}\ \ (\texttt{split}\ a), \texttt{fst}\ \ (\texttt{split}\ b)),\\ &\qquad\qquad \texttt{append}\ (\texttt{snd}\ \ (\texttt{split}\ a), \texttt{snd}\ \ (\texttt{split}\ b)))\end{aligned}$$

Since `combine` takes a pair of lists with the same length, the output specification $\{\alpha\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho\}$ of `append` for some polymorphic index variables ρ was mined from the call site $\texttt{append}\ (\texttt{snd}\ \ (\texttt{split}\ a), \texttt{snd}\ \ (\texttt{split}\ b))$. Then, our system propagated the output specification backward to obtain the input specification $\{\alpha\ \texttt{list}^{\rho_1} \times \alpha\ \texttt{list}^{\rho_2} \mid \rho_1 + \rho_2 = \rho\}$ of `append`. Our system propagated the input specification of `append`, and mined the output specification $\{\alpha\ \texttt{list}^{\rho_2} \times \beta\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho\}$ of `split` for some polymorphic index variables ρ from the call site $\texttt{snd}\ \ (\texttt{split}\ b)$. Here, our system reused the polymorphic type $\forall\alpha,\beta.\alpha \times \beta \rightarrow \beta$ of `snd` to mine the output specification of `split`. Then, our system propagated the output specification backward to obtain the input specification $\{(\alpha\times\beta)\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\}$ of `split`. The input specification of `split` is reused for the call site $\texttt{snd}\ \ (\texttt{split}\ a)$. Our system analyzed the other call sites $\texttt{append}\ (\texttt{fst}\ \ (\texttt{split}\ a), \texttt{fst}\ \ (\texttt{split}\ b))$ similarly, and obtained the type $\forall\alpha,\beta.\forall\rho.\{(\alpha\times\beta)\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\} \rightarrow \{\alpha\ \texttt{list}^{\rho_2} \times \beta\ \texttt{list}^{\rho_3} \mid \rho_2 = \rho\}$ of `split`.

function	inferred specifications	time (sec.)
`length`	$\forall\alpha.\forall\rho,\rho'.\{\alpha\ \texttt{list}^{\rho_1} \mid \rho \geq \rho_1 \geq \rho'\} \to \{\texttt{int}^{\rho_2} \mid \rho \geq \rho_2 \geq \rho'\}$	27.773
`hd`	$\forall\alpha.\{\alpha\ \texttt{list}^{\rho} \mid \rho > 0\} \to \alpha$	0.004
`tl`	$\forall\alpha.\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \mid \rho_1 > 0 \land \rho_1 = \rho + 1\} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho\}$	0.064
`nth`	$\forall\alpha.\{\alpha\ \texttt{list}^{\rho_1} \times \texttt{int}^{\rho_2} \mid \rho_1 > \rho_2 \geq 0\} \to \alpha$	0.268
`rev`	$\forall\alpha.\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho\}$	0.540
`append`	$\forall\alpha.\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \times \alpha\ \texttt{list}^{\rho_2} \mid \rho_1 + \rho_2 = \rho\} \to$ $\{\alpha\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho\}$	2.892
`map`	$\forall\alpha,\beta.(\alpha \to \beta) \to \forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\} \to \{\beta\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho\}$	0.292
`iter2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{unit}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{unit}$	0.276
`map2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \to \gamma) \to$ $\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2 = \rho\} \to \{\gamma\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho\}$	14.236
`rev_map2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \to \gamma) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \gamma\ \texttt{list}$	0.448
`fold_left2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \times \gamma \to \alpha) \to$ $\{\alpha \times (\beta\ \texttt{list}^{\rho_1} \times \gamma\ \texttt{list}^{\rho_2}) \mid \rho_1 = \rho_2\} \to \alpha$	0.276
`fold_right2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \times \gamma \to \gamma) \to$ $\{(\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2}) \times \gamma \mid \rho_1 = \rho_2\} \to \gamma$	0.276
`for_all2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{bool}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{bool}$	0.276
`exists2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{bool}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{bool}$	0.276
`split`	$\forall\alpha,\beta.\forall\rho.\{(\alpha \times \beta)\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\} \to$ $\{\alpha\ \texttt{list}^{\rho_2} \times \beta\ \texttt{list}^{\rho_3} \mid \rho_2 = \rho_3 = \rho\}$	0.340
`combine`	$\forall\alpha,\beta.\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2 = \rho\} \to$ $\{(\alpha \times \beta)\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho\}$	15.576

Table 1. The specifications of the library functions from the OCaml list module. Our system automatically inferred them from the call sites of the functions in Table 3.

By merging the inferred types, we obtained the type $\forall\alpha,\beta.\forall\rho.\{(\alpha \times \beta)\ \texttt{list}^{\rho_1} \mid \rho_1 = \rho\} \to \{\alpha\ \texttt{list}^{\rho_2} \times \beta\ \texttt{list}^{\rho_3} \mid \rho_2 = \rho_3 = \rho\}$ of `split`.

6 Related Work

As already mentioned in Section 1, closely related to ours is the work on DML [4, 5] and size inference [1–3].

DML [4, 5] is an extension of ML with a restricted form of dependent types. DML requires users to declare function types, and then automatically performs implicit argument inference and type checking. An advantage of our approach is that users need not always declare function types, as demonstrated in the verification of sorting functions. On the other hand, advantages of DML are that the type checking algorithm is complete. In practice, therefore, combination of DML's approach and our approach seems useful.

Size inference can automatically infer size relations between arguments and return values of functions [1–3]. A main difference is that the size inference tries

function	inferred specifications	time (sec.)
`length`	$\forall\alpha.\alpha\ \texttt{list}^{\rho_1} \to \{\texttt{int}^{\rho_2} \mid \rho_2 = \rho_1\}$	27.773
`hd`	$\forall\alpha.\{\alpha\ \texttt{list}^{\rho} \mid \rho > 0\} \to \alpha$	0.004
`tl`	$\forall\alpha.\langle\rho_1 > 0 \mid \alpha\ \texttt{list}^{\rho_1} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho_1 - 1\}\rangle$	0.064
`nth`	$\forall\alpha.\{\alpha\ \texttt{list}^{\rho_1} \times \texttt{int}^{\rho_2} \mid \rho_1 > \rho_2 \geq 0\} \to \alpha$	0.268
`rev`	$\forall\alpha.\alpha\ \texttt{list}^{\rho_1} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho_1\}$	0.540
`append`	$\forall\alpha.\alpha\ \texttt{list}^{\rho_1} \times \alpha\ \texttt{list}^{\rho_2} \to \{\alpha\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho_1 + \rho_2\}$	2.892
`map`	$\forall\alpha,\beta.(\alpha \to \beta) \to \alpha\ \texttt{list}^{\rho_1} \to \{\beta\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho_1\}$	0.292
`iter2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{unit}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{unit}$	0.276
`map2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \to \gamma) \to$ $\langle\rho_1 = \rho_2 \mid \alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \to \{\gamma\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho_1\}\rangle$	14.236
`rev_map2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \to \gamma) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \gamma\ \texttt{list}$	0.448
`fold_left2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \times \gamma \to \alpha) \to$ $\{\alpha \times (\beta\ \texttt{list}^{\rho_1} \times \gamma\ \texttt{list}^{\rho_2}) \mid \rho_1 = \rho_2\} \to \alpha$	0.276
`fold_right2`	$\forall\alpha,\beta,\gamma.(\alpha \times \beta \times \gamma \to \gamma) \to$ $\{(\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2}) \times \gamma \mid \rho_1 = \rho_2\} \to \gamma$	0.276
`for_all2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{bool}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{bool}$	0.276
`exists2`	$\forall\alpha,\beta.(\alpha \times \beta \to \texttt{bool}) \to$ $\{\alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \mid \rho_1 = \rho_2\} \to \texttt{bool}$	0.276
`split`	$\forall\alpha,\beta.(\alpha \times \beta)\ \texttt{list}^{\rho_1} \to \{\alpha\ \texttt{list}^{\rho_2} \times \beta\ \texttt{list}^{\rho_3} \mid \rho_2 = \rho_3 = \rho_1\}$	0.340
`combine`	$\forall\alpha,\beta.\langle\rho_1 = \rho_2 \mid \alpha\ \texttt{list}^{\rho_1} \times \beta\ \texttt{list}^{\rho_2} \to \{(\alpha \times \beta)\ \texttt{list}^{\rho_3} \mid \rho_3 = \rho_1\}\rangle$	15.576

Table 2. A manually simplified version of the specifications in Table 3.

to infer as specification as possible from the definition of a function, while our algorithm starts with simple types, and gradually refines the types based on information about functions' call sites. A main advantage of our approach is that we can allow more flexible dependent types based on the user's demand (as demonstrated in the verification of sorting functions, where two kinds of list types were declared). Another possible advantage of our approach (that has yet to be confirmed by more experiments) is that the on-demand inference can be more efficient, especially when precise specification is not required for most functions. On the other hand, an advantage of size inference is that it can find more precise specification than ours, and that it needs to infer the specification of a function just once.

Rich type systems have been introduced to practical programming languages so that non-trivial program invariants can be expressed as types. Datasort refinements (often called refinement types) as well as type reconstruction algorithm for a finite set of user-defined refinements were introduced by Freeman et al. [10]. The algorithm infers refinement types of functions without requiring type annotations. However, the datasort refinements cannot express linear constraints (e.g. those on the lengths of lists) unlike in the index refinements of DML. Datasort refinements were also introduced to programming languages with computational effects [11]. Dunfield et al. combined the datasort refinements and the index

file	code	refined funcs.
`predabst.ml`	`combine (a1, (tl a2))`	`tl`
`completion.ml`	`nth (a3, (length a3 - 1))`	`length`
`xdr.ml`	`let (a4, a5) = split a6 in`	`map, split`
	`combine (a4, map f1 a5)`	
`pmlize.ml`	`combine (rev a7, a8)`	`rev`
`ass.ml`	`combine (append (fst (split a9), fst (split a10)),`	`append, split`
	`append (snd (split a9), snd (split a10)))`	
`printtyp.ml`	`map2 f2 (a11, map2 f3 (a12, a13))`	`map2`
`ctype.ml`	`fold_left2 f4 (a14, a15, combine (a16, a17))`	`combine`

Table 3. The call sites used to infer the specifications of the functions in Table 1. We collected them from the existing programs written in OCaml.

refinements, and presented type reconstruction algorithms for them [12, 13]. Recently, generalized algebraic data types (GADT) have been introduced to practical functional programming languages such as Haskell. Several researchers proposed GADT inference algorithms [14–16]. *Partial* type inference in the spirit of local type inference [17] is employed in those type systems, to reduce type annotations. Type information can, however, be propagated locally, so that the types of recursive functions cannot be inferred automatically.

Flanagan proposed hybrid type checking which allows users to refine data types with arbitrary program terms [18]. A type reconstruction algorithm for that type system has been proposed by Knowles and Flanagan [6]. The result of their type inference algorithm, however contains fixed-point operators on predicates, so that their algorithm alone can neither statically detect errors, nor produce useful documentations for the program. Their algorithm does not support compound data structures and parametric polymorphism.

There are other approaches to applying dependent types to practical programming languages [19–21]. They require either more type annotations than DML or dynamic checks.

As mentioned in Section 1, the idea of our approach has been inspired by automatic predicate discovery and loop invariant inference in other verification techniques, such as predicate abstraction [7, 22–24], the induction-iteration method [25, 26], on-demand loop invariant refinement by Leino [27], and constraint-based invariant generation which solves unknown parameters in invariant templates [28, 29]. Our main contribution in this respect is to bring those techniques into the context of dependently-typed functional languages; The advantage of using the type-based setting is that the verification technique can be smoothly extended to support algebraic data types, higher-order functions, etc.

7 Conclusion

We have proposed a novel approach to applying dependent types to practical programming languages: Our type inference system first assigns simple types to functions, and refine them *on demand*, using information about both the functions' definitions and call sites. A prototype type inference system has been already implemented and tested for non-trivial programs.

References

1. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL '96, ACM Press (1996) 410–423
2. Chin, W.N., Khoo, S.C.: Calculating sized types. In: PEPM '00, ACM Press (1999) 62–72
3. Chin, W.N., Khoo, S.C., Xu, D.N.: Extending sized type with collection analysis. In: PEPM '03, ACM Press (2003) 75–84
4. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI '98, ACM Press (1998) 249–257
5. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL '99, ACM Press (1999) 214–227
6. Knowles, K., Flanagan, C.: Type reconstruction for general refinement types. In: ESOP '07. (2007)
7. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI '01, ACM Press (2001) 203–213
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL '78, ACM Press (1978) 84–96
9. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: ASIAN '06. LNCS, Springer-Verlag (December 2006)
10. Freeman, T., Pfenning, F.: Refinement types for ML. In: PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, ACM Press (1991) 268–277
11. Davies, R., Pfenning, F.: Intersection types and computational effects. In: ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, ACM Press (2000) 198–208
12. Dunfield, J.: Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University (September 2002)
13. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL '04, ACM Press (2004) 281–292
14. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL '03, ACM Press (2003) 224–235
15. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: POPL '06, ACM Press (2006) 232–244
16. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP '06, ACM Press (2006) 50–61
17. Pierce, B.C., Turner, D.N.: Local type inference. In: POPL '98, ACM Press (1998) 252–265
18. Flanagan, C.: Hybrid type checking. In: POPL '06, ACM Press (2006) 245–256
19. Augustsson, L.: Cayenne – a language with dependent types. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, ACM Press (1998) 239–250
20. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Manuscript, available online (April 2005)

21. Ou, X., Tan, G., Mandelbaum, Y., Walker, D.: Dynamic typing with dependent types. In: Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science. (August 2004) 437–450
22. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV '97, Springer-Verlag (1997) 72–83
23. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL '02, ACM Press (2002) 191–202
24. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL '02, ACM Press (2002) 58–70
25. Suzuki, N., Ishihata, K.: Implementation of an array bound checker. In: POPL '77, ACM Press (1977) 132–143
26. Xu, Z., Miller, B.P., Reps, T.: Safety checking of machine code. In: PLDI '00, ACM Press (2000) 70–82
27. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: APLAS '05. Volume 3780 of LNCS., Springer-Verlag (November 2005) 119–134
28. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: SAS '04. Volume 3148 of LNCS., Springer-Verlag (August 2004) 53–68
29. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI '07, ACM Press (2007) 300–309

Appendix

A Insertion Sort and Merge Sort Programs

The following is the actual code used in the experiment described in Section 5.1.

```
type 'a list(n) with n>=0 =
  Nil of unit where n=0
| Cons of 'a * 'a list(r) where n=r+1

type ordlist(h) with tt =
  { r : tt } Nil of unit where h=r
| { r1, r2 : r1 <= r2 }
  Cons of int(r1) * ordlist(r2) where h=r1

fun isort xs =
  fun insert (x, xs) =
    match xs with
      Nil _ -> Cons (x, Nil ())
    | Cons (y, ys) ->
        if x <= y then  Cons(x, Cons(y, ys))
        else Cons(y, insert (x, ys))
  in
  match xs with
    Nil _ -> Nil ()
  | Cons (x, xs') -> insert (x, isort xs')
in
  (isort xs : ordlist)

fun merge (l1, l2) =
  match l1 with
    Nil _ -> l2
  | Cons (h1, t1) ->
      (match l2 with
        Nil _ -> l1
      | Cons (h2, t2) ->
          if h1 <= h2 then Cons (h1, merge (t1, l2))
          else Cons (h2, merge (l1, t2)))
in
fun msort xs =
  fun msplit xs =
    match xs with
      Nil _ -> (Nil (), Nil ())
    | Cons (x, xs') ->
        (match xs' with
          Nil _ -> (Cons (x, Nil ()), Nil ())
        | Cons (y, ys) ->
            let (zs1, zs2) = msplit ys in
            (Cons (x, zs1), Cons (y, zs2)))
  in
  let (ys1, ys2) = msplit xs in
    merge (msort ys1, msort ys2)
in
  (msort xs : ordlist)
```

B Operational Semantics

The operational semantics of the language is given in Figure 5.

Evaluation Contexts

$$\begin{aligned} E ::= {} & [\bullet] \mid (E, e) \mid (v, E) \mid (\texttt{fun}\ f\ x = e)\ E \mid f\ E \\ \mid {} & \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{let}\ (x_1, x_2) = E\ \texttt{in}\ e \\ \mid {} & \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{assert}\ E\ \texttt{in}\ e \end{aligned}$$

Evaluation Rules

$$\frac{}{\texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2 \longrightarrow [\texttt{fun}\ f\ x = e_1/f]e_2}\ (\text{E-Let-Fun})$$

$$\frac{n \neq 0}{\texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1}\ (\text{E-If-True})$$

$$\frac{n = 0}{\texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2}\ (\text{E-If-False})$$

$$\frac{}{(\texttt{fun}\ f\ x = e)\ v \longrightarrow [v/x, \texttt{fun}\ f\ x = e/f]e}\ (\text{E-Fun-App})$$

$$\frac{n \neq 0}{\texttt{assert}\ n\ \texttt{in}\ e \longrightarrow e}\ (\text{E-Assert})$$

$$\frac{}{f\ v \longrightarrow [\![f]\!](v)}\ (\text{E-Ext-Fun-App})$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}\ (\text{E-Context})$$

$$\frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow [v/x]e}\ (\text{E-Let})$$

$$\frac{}{e \longrightarrow^* e}$$

$$\frac{}{\texttt{let}\ (x_1, x_2) = (v_1, v_2)\ \texttt{in}\ e \longrightarrow [v_1/x_1, v_2/x_2]e}\ (\text{E-Let-Pair})$$

$$\frac{e \longrightarrow^* e'' \qquad e'' \longrightarrow e'}{e \longrightarrow^* e'}$$

Fig. 5. The Operational Semantics

C Proof of Soundness of the Type System

We introduce typing rules T-Fun and T-Fun-App for run-time function closures.

$$\frac{\begin{array}{c}\phi \wedge \phi'; \mathit{\Gamma}, f : \sigma, x : t \vdash e : \tau \\ \widetilde{\rho} \cap \mathrm{FIV}(\mathit{\Gamma}, \phi) = \emptyset \qquad \sigma = \forall \widetilde{\rho}. \langle \phi' \mid t \to \tau \rangle\end{array}}{\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e : \sigma} \quad (\text{T-Fun})$$

$$\frac{\begin{array}{c}\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e_1 : \sigma \qquad \phi \vdash \sigma \leqslant \tau_1 \to \tau_2 \\ \phi; \mathit{\Gamma} \vdash e_2 : \tau_1\end{array}}{\phi; \mathit{\Gamma} \vdash (\texttt{fun}\ f\ x = e_1)\ e_2 : \tau_2} \quad (\text{T-Fun-App})$$

Theorem 1 (Soundness). *If $\top; \mathit{\Gamma} \vdash e : \tau$ is derivable, $\mathrm{FV}(e) = \emptyset$, and $\mathit{\Gamma}$ is valid, then e either evaluates to a value or diverges.*

Proof. A corollary of the type preservation (Lemma 3) and the progress (Lemma 5), which are proved later in this section.

C.1 Preservation

Lemma 1 (Substitution).

1. *If $\phi; \mathit{\Gamma} \vdash v : \{t \mid \phi'\}$, $\phi \wedge \phi'; \mathit{\Gamma}, x : t \vdash e : \tau$, and $\mathrm{FIV}(t) \cap \mathrm{FIV}(\tau) = \emptyset$ then, $\phi; \mathit{\Gamma} \vdash [v/x]e : \tau$ is derivable.*
2. *If $\phi; \mathit{\Gamma} \vdash v : \sigma$ and $\phi; \mathit{\Gamma}, f : \sigma \vdash e : \tau$ then, $\phi; \mathit{\Gamma} \vdash [v/f]e : \tau$ is derivable.*

Lemma 2 (Subtyping). *If $\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e : \sigma$ and $\phi \vdash \sigma \leqslant \sigma'$ then $\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e : \sigma'$.*

Lemma 3 (Preservation). *Suppose that $\phi; \mathit{\Gamma} \vdash e : \tau$, $e \longrightarrow e'$, and $\mathit{\Gamma}$ is valid. Then, $\phi; \mathit{\Gamma} \vdash e' : \tau$ is derivable.*

Proof. We prove the theorem by induction on the derivation of $\phi; \mathit{\Gamma} \vdash e : \tau$.

T-Sub we have $\phi = \phi_1$ and $\tau = \{t \mid \phi_2\}$ where $\phi_1'; \mathit{\Gamma} \vdash e : \{t \mid \phi_2'\}$ and $\models \phi_1 \Rightarrow (\phi_1' \wedge (\phi_2' \Rightarrow \phi_2))$.
By I.H., we get $\phi_1'; \mathit{\Gamma} \vdash e' : \{t \mid \phi_2'\}$. By T-Sub, we obtain $\phi_1; \mathit{\Gamma} \vdash e' : \{t \mid \phi_2'\}$.

Otherwise By induction on the derivation of $e \longrightarrow e'$.

E-Let-Fun We have $e = \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2$ and $e' = [\texttt{fun}\ f\ x = e_1/f]e_2$.
By T-Let-Fun, we have $\phi \wedge \phi_1; \mathit{\Gamma}, f : \sigma, x : t_1 \vdash e_1 : \tau_1$, $\widetilde{\rho} \cap \mathrm{FIV}(\mathit{\Gamma}, \phi) = \emptyset$, $\sigma = \forall \widetilde{\rho}. \langle \phi_1 \mid t_1 \to \tau_1 \rangle$, and $\phi; \mathit{\Gamma}, f : \sigma \vdash e_2 : \tau$.
By T-Fun, we obtain $\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e_1 : \sigma$.
By Lemma 1, we have $\phi; \mathit{\Gamma} \vdash [\texttt{fun}\ f\ x = e_1/f]e_2 : \tau$.

E-Fun-App We have $e' = [v/x, \texttt{fun}\ f\ x = e_1/f]e_1$ and $e = (\texttt{fun}\ f\ x = e_1)\ v$.
By T-Fun-App, we get $\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e_1 : \sigma$, $\phi \vdash \sigma \leqslant \tau' \to \tau$, and $\phi; \mathit{\Gamma} \vdash v : \tau'$.
By Lemma 2, we get $\phi; \mathit{\Gamma} \vdash \texttt{fun}\ f\ x = e_1 : \tau' \to \tau$,
By T-Fun, we obtain $\phi \wedge \phi'; \mathit{\Gamma}, f : \tau' \to \tau, x : t \vdash e_1 : \tau$.
By Lemma 1, we have $\phi; \mathit{\Gamma} \vdash [v/x, \texttt{fun}\ f\ x = e_1/f]e_1 : \tau$.

E-Ext-Fun-App We have $e = f\ v$ and $e' = [\![f]\!](v)$. Because $\mathit{\Gamma}$ is valid, $[\![f]\!](v)$ is defined and $\phi; \mathit{\Gamma} \vdash [\![f]\!](v) : \tau$.

E-Let We have $e = \texttt{let}\ x = v\ \texttt{in}\ e_1$ and $e' = [v/x]e_1$.
By T-Let, we have $\phi; \mathit{\Gamma} \vdash v : \{t \mid \phi'\}$, $\phi \wedge \phi'; \mathit{\Gamma}, x : t \vdash e_1 : \tau$, and $\mathrm{FIV}(t) \cap \mathrm{FIV}(\tau) = \emptyset$.
By Lemma 1, we obtain $\phi; \mathit{\Gamma} \vdash [v/x]e_1 : \tau$.

E-Let-Pair Similar to the case E-Let

E-If-True We have $e = \texttt{if}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ and $e' = e_1$ where $n \neq 0$.
By T-If, we have $\phi; \mathit{\Gamma} \vdash n : \{\texttt{int}^\rho \mid \phi'\}$ and $\phi \wedge (\exists \rho.(\phi' \wedge \rho \neq 0)); \mathit{\Gamma} \vdash e_1 : \tau$.
By T-Int (and T-Sub), we get $\models \phi \Rightarrow (\rho = n \Rightarrow \phi')$. Since $\rho \notin \mathrm{FIV}(\phi)$, we have $\models \phi \Rightarrow (\exists \rho.(\phi' \wedge \rho = n))$. Then, we get $\models \phi \Rightarrow (\exists \rho.(\phi' \wedge \rho \neq 0))$.
By T-Sub, we obtain $\phi; \mathit{\Gamma} \vdash e_1 : \tau$.

E-If-False Similar to the case E-If-True

E-Assert We have $e = \texttt{assert}\ n\ \texttt{in}\ e_1$ and $e' = e_1$.
By T-Assert, we have $\phi; \mathit{\Gamma} \vdash e_1 : \tau$.

E-Context We have $e = E[e_1]$ and $e' = E[e_1']$ where $e_1 \longrightarrow e_1'$. We can obtain $\phi; \mathit{\Gamma} \vdash E[e_1'] : \tau$ by case analysis of the structure of $E[e_1]$.

C.2 Progress

Lemma 4 (Canonical Form). *If v is a value such that $\phi; \mathit{\Gamma} \vdash v : \{\texttt{int}^\rho \mid \phi'\}$, then $v = n$ for some integer n such that $\models \phi \Rightarrow [n/\rho]\phi'$.*

Lemma 5 (Progress). *Suppose that $\phi; \mathit{\Gamma} \vdash e : \tau$, ϕ is satisfiable,* $\mathrm{FV}(e) = \emptyset$, *and $\mathit{\Gamma}$ is valid. Then, either e is a value or there exist e' such that $e \longrightarrow e'$.*

Proof. We prove the theorem by induction on the derivation of $\phi; \mathit{\Gamma} \vdash e : \tau$.

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

T-Int We have $e = n$. n is a value.

T-Pair We have $e = (e_1, e_2)$ where $\phi; \mathit{\Gamma} \vdash e_1 : \{t_1 \mid \phi_1\}$ and $\phi; \mathit{\Gamma} \vdash e_2 : \{t_2 \mid \phi_2\}$.
By I.H., either
- e_1 is a value. By I.H., either
  - e_2 is a value. Then, (e_1, e_2) is a value.
  - There exists e_2' such that $e_2 \longrightarrow e_2'$. E-Context applies.
- There exists e_1' such that $e_1 \longrightarrow e_1'$. E-Context applies.

T-Let-Fun We have $e = \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2$.
We can apply E-Let-Fun.

T-App We have $e = f\ e_1$ where $\phi; \mathit{\Gamma} \vdash e_1 : \tau_1$.
By I.H., either
- e_1 is a value v. Since $\mathit{\Gamma}$ is valud, $[\![f]\!](v)$ is defined. Therefore, E-Ext-Fun-App applies.
- There exists e_1' such that $e_1 \longrightarrow e_1'$. Therefore, E-Context applies.

T-Let We have $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ where $\phi; \mathit{\Gamma} \vdash e_1 : \{t \mid \phi'\}$.
By I.H., either
- e_1 is a value. E-Let applies.

  – There exists e_1' such that $e_1 \longrightarrow e_1'$. E-CONTEXT applies.

T-Let-Pair Similar to the case T-LET

T-If We have $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ where $\phi; \Gamma \vdash e_1 : \{\texttt{int}^\rho \mid \phi'\}$.
By I.H., either
  – e_1 is a value. By Lemma 4, $e_1 = n$ for some integer n. Therefore, E-IF-TRUE or E-IF-FALSE applies.
  – There exists e_1' such that $e_1 \longrightarrow e_1'$. E-CONTEXT applies.

T-Assert We have $e = \texttt{assert}\ e_1\ \texttt{in}\ e_2$ where $\phi; \Gamma \vdash e_1 : \{\texttt{int}^\rho \mid \rho \neq 0\}$.
By I.H., either
  – e_1 is a value. By Lemma 4, $e_1 = n$ for some integer n such that $\models \phi \Rightarrow n \neq 0$. We get $n \neq 0$ since ϕ is satisfiable. Therefore, E-ASSERT applies.
  – There exists e_1' such that $e_1 \longrightarrow e_1'$. E-CONTEXT applies.

T-Sub We have $\phi = \phi_1$ and $\tau = \{t \mid \phi_2\}$ where $\phi_1'; \Gamma \vdash e : \{t \mid \phi_2'\}$ and $\models \phi_1 \Rightarrow (\phi_1' \wedge (\phi_2' \Rightarrow \phi_2))$.
ϕ_1' is satisfiable since ϕ_1 is satisfiable. By I.H., either e is a value or there exists e' such that $e \longrightarrow e'$.

T-Fun We have $e = \texttt{fun}\ f\ x = e_1$.
$\texttt{fun}\ f\ x = e_1$ is a value.

T-Fun-App Similar to the case T-APP

D Proof of Soundness of Type Inference Rules

Lemma 6. *If $\vdash \Gamma' \leqslant \Gamma$ and $\Gamma; e \vdash \tau :$ is derivable then, $\Gamma'; e \vdash \tau :$ is derivable.*

Lemma 7. *If $\phi; \Gamma \vdash e_1 : \{t_1 \mid \phi_1\}$ and $\phi_1; \Gamma \vdash e_2 : \{t_2 \mid \phi_2\}$ and derivable then, $\phi; \Gamma \vdash e_2 : \{t_2 \mid \phi_2\}$ is derivable.*

Lemma 8. *If $\Delta \rhd f : \sigma \dashv_{\widetilde{P}} S; \Delta'$ and Δ is valid, then Δ' is valid, $\vdash (\!|\Delta'|\!) \leqslant (\!|\Delta|\!)$, and $f : \sigma' \in (\!|\Delta'|\!)$ for some σ' such that $\vdash \sigma' \leqslant S(\sigma)$.*

Theorem 2 (Soundness). *If $\Delta \rhd e : \tau \dashv \phi; \Delta'$ is derivable and Δ is valid then, Δ' is valid, $\vdash (\!|\Delta'|\!) \leqslant (\!|\Delta|\!)$, and $\phi; (\!|\Delta'|\!) \vdash e : \tau$ is derivable.*

Proof. We prove the theorem by induction on the derivation of $\Delta \rhd e : \tau \dashv \phi; \Delta'$.

B-Var We have $e = x$, $\tau = \{t \mid \phi'\}$, $\phi = [t'/t]\phi'$, and $\Delta' = \Delta$ where $x : t' \in \Delta$ and $|t| = |t'|$. $\widetilde{\rho'} = \mathrm{FIV}(t')$.
By T-VAR, we get $[t'/t]\phi'; (\!|\Delta|\!) \vdash x : \{t \mid \widetilde{\rho} = \widetilde{\rho'}\}$ and $\widetilde{\rho} = \mathrm{FIV}(t)$. We have $\models [t'/t]\phi' \Rightarrow (\widetilde{\rho} = \widetilde{\rho'} \Rightarrow \phi')$. By T-SUB, we obtain $[t'/t]\phi'; (\!|\Delta|\!) \vdash x : \{t \mid \phi'\}$.

B-Int We have $e = n$, $\tau = \{\texttt{int}^\rho \mid \phi'\}$, $\phi = [n/\rho]\phi'$, and $\Delta' = \Delta$.
By T-INT, we get $[n/\rho]\phi'; (\!|\Delta|\!) \vdash n : \{\texttt{int}^\rho \mid \rho = n\}$. We have $\models [n/\rho]\phi' \Rightarrow (\rho = n \Rightarrow \phi')$. By T-SUB, we obtain $[n/\rho]\phi'; (\!|\Delta|\!) \vdash n : \{\texttt{int}^\rho \mid \phi'\}$.

B-Pair We have $e = (e_1, e_2)$, $\tau = \{t_1 \times t_2 \mid \phi'\}$, $\phi = \phi_1$, and $\Delta' = \Delta_1$ where $\Delta \rhd e_2 : \{t_2 \mid \phi'\} \dashv \phi_2; \Delta_2$ and $\Delta_2 \rhd e_1 : \{t_1 \mid \phi_2\} \dashv \phi_1; \Delta_1$.
By I.H., we obtain $\phi_2; (\!|\Delta_2|\!) \vdash e_2 : \{t_2 \mid \phi'\}$ and $\phi_1; (\!|\Delta_1|\!) \vdash e_1 : \{t_1 \mid \phi_2\}$.
We have $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_2|\!)$.
By Lemma 6 and Lemma 7, we get $\phi_1; (\!|\Delta_1|\!) \vdash e_2 : \{t_2 \mid \phi'\}$.
By T-PAIR, we get $\phi_1; (\!|\Delta_1|\!) \vdash (e_1, e_2) : \{t_1 \times t_2 \mid \phi'\}$.

B-Let-Fun We have $e = \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2$ and $\Delta' = \Delta_2 \setminus f$ where $\sigma = \forall\widetilde{\rho}.\langle \phi' \mid t \to \tau_1\rangle = \mathrm{TypeOf}(\Delta, \texttt{fun}\ f\ x = e_1)$, $\Delta, f : \sigma, x : t \rhd e_1 : \tau_1 \dashv \phi_1; \Delta_1, f : \sigma, x : t$, $\psi = \forall\widetilde{\rho}, \mathrm{FIV}(t).(\phi' \Rightarrow \phi_1)$, $\mathrm{dom}(S) = \mathrm{FPV}(\sigma)$, $\models S(\psi)$, and $\Delta_1, f : (\sigma; \psi; \{S\}) \rhd e_2 : \tau \dashv \phi; \Delta_2$.
By I.H., we obtain $\phi_1; (\!|\Delta_1, f : \sigma, x : t|\!) \vdash e_1 : \tau_1$ and $\vdash (\!|\Delta_1, f : \sigma, x : t|\!) \leqslant (\!|\Delta, f : \sigma, x : t|\!)$, and $\phi; (\!|\Delta_2|\!) \vdash e_2 : \tau$ and $\vdash (\!|\Delta_2|\!) \leqslant (\!|\Delta_1, f : (\sigma; \psi; \{S\})|\!)$.
Suppose that $(\!|\Delta_2|\!) = \Gamma, f : S'(\sigma)$.
We have $\vdash \Gamma, f : S'(\sigma), x : t \leqslant (\!|\Delta_1, f : S'(\sigma), x : t|\!)$ and $\models S'(\psi)$.
By T-Sub and Lemma 6, we obtain $\phi \wedge S'(\phi'); \Gamma, f : S'(\sigma), x : t \vdash e_1 : S'(\tau_1)$.
By T-Let-Fun, we have $\phi; (\!|\Delta_2 \setminus f|\!) \vdash \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau$.

B-App We have $e = f\ e'$ and $\Delta' = \Delta_2$ where $t = \mathrm{TypeOf}(\Delta, e')$, $\widetilde{\rho} = \mathrm{FIV}(t)$, P : fresh, $\Delta \rhd f : \{t \mid P(\widetilde{\rho})\} \to \tau \dashv_{\{P\}} S; \Delta_1$, and $\Delta_1 \rhd e' : \{t \mid S(P(\widetilde{\rho}))\} \dashv \phi_2; \Delta_2$.
By I.H., we obtain $\phi_2; (\!|\Delta_2|\!) \vdash e' : \{t \mid S(P(\widetilde{\rho}))\}$ and $\vdash (\!|\Delta_2|\!) \leqslant (\!|\Delta_1|\!)$.
By Lemma 8, we get $f : \sigma' \in (\!|\Delta_1|\!)$ for some σ' such that $\vdash \sigma' \leqslant \{t \mid S(P(\widetilde{\rho}))\} \to \tau$.
By T-App, we get $\phi; (\!|\Delta_1|\!) \vdash f\ e' : \tau$.
By Lemma 6, we obtain $\phi; (\!|\Delta_2|\!) \vdash f\ e' : \tau$.

B-Let We have $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$, $\phi = \phi_1$, and $\Delta' = \Delta_1$ where $t = \mathrm{TypeOf}(\Delta, e_1)$, $\Delta, x : t \rhd e_2 : \tau \dashv \phi_2; \Delta_2$, and $\Delta_2 \setminus x \rhd e_1 : \{t \mid \phi_2\} \dashv \phi_1; \Delta_1$.
By I.H., we get $\phi_2; (\!|\Delta_2|\!) \vdash e_2 : \tau$ and $\vdash (\!|\Delta_2|\!) \leqslant (\!|\Delta, x : t|\!)$, and $\phi_1; (\!|\Delta_1|\!) \vdash e_1 : \{t \mid \phi_2\}$ and $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_2 \setminus x|\!)$.
By T-Sub, we obtain $\phi_1 \wedge \phi_2; (\!|\Delta_2|\!) \vdash e_2 : \tau$.
By Lemma 6 and $\vdash (\!|\Delta_1|\!), x : t \leqslant (\!|\Delta_2 \setminus x|\!), x : t = (\!|\Delta_2|\!)$, we get $\phi_1 \wedge \phi_2; (\!|\Delta_1|\!), x : t \vdash e_2 : \tau$.
We have $\mathrm{FIV}(t) \cap \mathrm{FIV}(\tau) = \emptyset$.
By T-Let, we get $\phi_1; (\!|\Delta_1|\!) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau$.
We obtain $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_2 \setminus x|\!) \leqslant (\!|\Delta, x : t \setminus x|\!) = (\!|\Delta|\!)$.

B-Let-Pair Similar to the case B-Let

B-If We have $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$, $\phi = \phi_1$, and $\Delta' = \Delta_1$ where $\Delta \rhd e_2 : \tau \dashv \phi_2; \Delta_2$, $\Delta_2 \rhd e_3 : \tau \dashv \phi_3; \Delta_3$, ρ : fresh, $\Delta_3 \rhd e_1 : \{\texttt{int}^\rho \mid \phi'\} \dashv \phi_1; \Delta_1$, and $\phi' = (\rho \neq 0 \wedge \phi_2) \vee (\rho = 0 \wedge \phi_3)$.
By I.H., we get $\phi_2; (\!|\Delta_2|\!) \vdash e_2 : \tau$ and $\vdash (\!|\Delta_2|\!) \leqslant (\!|\Delta|\!)$, $\phi_3; (\!|\Delta_3|\!) \vdash e_3 : \tau$ and $\vdash (\!|\Delta_3|\!) \leqslant (\!|\Delta_2|\!)$, and $\phi_1; (\!|\Delta_1|\!) \vdash e_1 : \{\texttt{int}^\rho \mid \phi'\}$ and $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_3|\!)$.
We have $\models (\phi_1 \wedge (\exists\rho.(\phi' \wedge \rho \neq 0))) \Rightarrow \phi_2$, $\models (\phi_1 \wedge (\exists\rho.(\phi' \wedge \rho = 0))) \Rightarrow \phi_3$, and $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_2|\!)$.
By T-Sub and Lemma 6, we obtain $\phi_1 \wedge (\exists\rho.(\phi' \wedge \rho \neq 0)); (\!|\Delta_1|\!) \vdash e_2 : \tau$.
By T-Sub and Lemma 6, we have $\phi_1 \wedge (\exists\rho.(\phi' \wedge \rho = 0)); (\!|\Delta_1|\!) \vdash e_3 : \tau$.
By T-If, we get $\phi_1; (\!|\Delta_1|\!) \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau$.

B-Assert We have $e = \texttt{assert}\ e_1\ \texttt{in}\ e_2$, $\phi = \phi_1 \wedge \phi_2$, and $\Delta' = \Delta_2$ where ρ : fresh, $\Delta \rhd e_1 : \{\texttt{int}^\rho \mid \rho \neq 0\} \dashv \phi_1; \Delta_1$, and $\Delta_1 \rhd e_2 : \tau \dashv \phi_2; \Delta_2$.
By I.H., we get $\phi_1; (\!|\Delta_1|\!) \vdash e_1 : \{\texttt{int}^\rho \mid \rho \neq 0\}$ and $\vdash (\!|\Delta|\!) \leqslant (\!|\Delta_1|\!)$, $\phi_2; (\!|\Delta_2|\!) \vdash e_2 : \tau$ and $\vdash (\!|\Delta_1|\!) \leqslant (\!|\Delta_2|\!)$.
We have $\vdash (\!|\Delta|\!) \leqslant (\!|\Delta_2|\!)$.
By T-Sub and Lemma 6, we obtain $\phi_1 \wedge \phi_2; (\!|\Delta|\!) \vdash e_1 : \{\texttt{int}^\rho \mid \rho \neq 0\}$.
By T-Sub and Lemma 6, we have $\phi_1 \wedge \phi_2; (\!|\Delta|\!) \vdash e_2 : \tau$.
By T-Assert, we get $\phi_1 \wedge \phi_2; (\!|\Delta|\!) \vdash \texttt{assert}\ e_1\ \texttt{in}\ e_2 : \tau$.