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 [1,2,3] 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 users to declare 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))
```

J. Garrigue and M. Hermenegildo (Eds.): FLOPS 2008, LNCS 4989, pp. 81–96, 2008.

```
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] propose 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 $f \stackrel{\triangle}{=} \lambda x.x+1$ is first given a type $\texttt{int} \to \texttt{int}$, 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 $\texttt{int list} \to \texttt{int list}$. If the type of `isort` is declared as $\texttt{int list} \to \texttt{ordlist}$ (where `ordlist` denotes the type of sorted lists), however, we can find from the call site $\texttt{insert}\ (x, \texttt{isort}\ xs')$ that the type of the output of `insert` should be `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 briefly 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 the full paper [8] 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 be non-zero. The type system is used to state properties of our type inference algorithm in Section 3. The type system is undecidable, since the constraint language includes integer addition and multiplication.

The syntax of types is defined as follows:

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

A constraint, denoted by ϕ, is an *index variable* ρ, a constant n, an operator expression $\mathtt{op}(\widetilde{\phi})$, or a quantifier expression. $\widetilde{o}$ signifies a list of objects $o_1, \ldots, o_m$ for some $m \geq 0$. 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 $\mathtt{int}^{\rho}$ 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, $\{\mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \mid \rho_1 > \rho_2\}$ 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, $\langle \rho_1 > 0 \wedge \rho_2 > 0 \mid \mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \to \{\mathtt{int}^{\rho_3} \mid \rho_3 = \rho_1 + \rho_2\}\rangle$ 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 the type τ under the type environment Γ. For example, $\rho > 0; x : \mathtt{int}^{\rho} \vdash x + 1 : \{\mathtt{int}^{\rho'} \mid \rho' > 1\}$.

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

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

$$\frac{\phi; \Gamma \vdash e_1 : \{t_1 \mid \phi_1\} \qquad \phi; \Gamma \vdash e_2 : \{t_2 \mid \phi_2\}}{\phi; \Gamma \vdash (e_1, e_2) : \{t_1 \times t_2 \mid \phi_1 \wedge \phi_2\}} \text{(T-Pair)}$$

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

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

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

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

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

$$\frac{\phi; \Gamma \vdash e_1 : \{\mathtt{int}^{\rho} \mid \rho \neq 0\} \qquad \phi; \Gamma \vdash e_2 : \tau}{\phi; \Gamma \vdash \mathtt{assert}\ e_1\ \mathtt{in}\ e_2 : \tau} \text{(T-Assert)}$$

$$\frac{\phi_1'; \Gamma \vdash e : \{t \mid \phi_2'\} \qquad \models \phi_1 \Rightarrow (\phi_1' \wedge (\phi_2' \Rightarrow \phi_2))}{\phi_1; \Gamma \vdash e : \{t \mid \phi_2\}} \text{(T-Sub)}$$

Fig. 1. Typing Rules

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. The relation $\eta \models \phi$ 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:

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

The type system ensures that evaluation of a well-typed, closed expression (i.e., an expression e such that $\top; \Gamma_0 \vdash e : \tau$, where Γ_0 is the type environment for primitive operators) never gets stuck: See [8] for a formal discussion.

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

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

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 $\forall\widetilde{\rho}.\langle P(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\}\rangle$, where $\widetilde{\rho}$ denotes a sequence of index variables (whose length is unknown). 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 $\Delta \rhd e : \tau \dashv \phi; \Delta'$. Here, Δ, e, and τ should be regarded as inputs of the algorithm, and ϕ and Δ' as outputs of the algorithm. Intuitively, ϕ is a sufficient condition for e to have type τ, and Δ' describes types refined during the inference. For example, let e, τ, and Δ be $f(z)$, $\{\mathtt{int}^{\rho} \mid \rho > 1\}$, and $z{:}\mathtt{int}^{\rho_z}$,

$f : (\sigma; \phi_1; \{S\})$, where $\sigma = \forall\widetilde{\rho}.\langle P(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\}\rangle$, $\phi_1 = \forall\widetilde{\rho}, \rho_x.P(\widetilde{\rho}, \rho_x) \Rightarrow \forall\rho_y.(\rho_y = \rho_x + 1 \Rightarrow Q(\widetilde{\rho}, \rho_x, \rho_y))$, and $S = \{P \mapsto \lambda\rho_x.\top, Q \mapsto \lambda(\rho_x, \rho_y).\top\}$. Then, ϕ and Δ' would be $\rho_z > 0$ and $z : \mathtt{int}^{\rho_z}, f : (\sigma; \phi_1; \{S, S'\})$, 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 $\Delta \rhd e : \tau \dashv \phi; \Delta'$ (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. The sub-algorithm $\sigma \leqslant \sigma' \dashv \phi$ for computing a sufficient condition ϕ for σ to be a subtype of σ' is also defined in Figure 3. In the figures, TypeOf(Δ, o) is a *template* for the type of some object o, obtained from the simple type of o by decorating it with fresh index variables and predicate variables. For example, if the simple type of o is `int`, then TypeOf(Δ, o) returns $\mathtt{int}^{\rho}$; if the simple type of o is $\mathtt{int} \to \mathtt{int}$, TypeOf$(\Delta, o)$ returns $\forall\widetilde{\rho}.\langle P(\widetilde{\rho}, \rho_x) \mid \mathtt{int}^{\rho_x} \to \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\}\rangle$.

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 $x : t' \in \Delta$, we check whether $|t| = |t'|$ (where $|t|$ is the simple type obtained from t by removing index variables). 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$(\sigma); \psi)$, 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 Δ'. 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 $\Delta \rhd f : \sigma \dashv_{\{P\}} S; \Delta'$ (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 $\psi \wedge \phi$.

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

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

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

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

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

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

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

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

Constraint Solving. We now describe a heuristic algorithm 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 $\lambda\widetilde{\rho}.\top$ (i.e., by computing $F^n(\lambda\widetilde{\rho}.\top)$ for $n = 1, 2, \ldots$). (As a special case, if $\psi(\widetilde{\rho}, P)$ does not contain P, then the iteration immediately converges with the solution $P = \lambda\widetilde{\rho}.\psi(\widetilde{\rho}, P)$.) The algorithm also uses widening [9] 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

$$\frac{\begin{array}{c}\sigma = \forall\widetilde{\rho}.\langle\phi \mid t \to \tau_1\rangle = \mathrm{TypeOf}(\Delta, \mathtt{fun}\ f\ x = e_1) \\ \Delta, f:\sigma, x:t \rhd e_1 : \tau_1 \dashv \phi_1; \Delta_1, f:\sigma, x:t \qquad \psi = \forall\widetilde{\rho}, \mathrm{FIV}(t).(\phi \Rightarrow \phi_1) \\ S = \mathrm{Solve}(\mathrm{FPV}(\sigma); \psi) \qquad \Delta_1, f:(\sigma;\psi;\{S\}) \rhd e_2 : \tau \dashv \phi_2; \Delta_2\end{array}}{\Delta \rhd \mathtt{fun}\ f\ x = e_1\ \mathtt{in}\ e_2 : \tau \dashv \phi_2; \Delta_2 \setminus f} \text{ (B-Let-Fun)}$$

$$\frac{\begin{array}{c}t = \mathrm{TypeOf}(\Delta, e) \qquad \widetilde{\rho} = \mathrm{FIV}(t) \qquad P : \text{fresh} \\ \Delta \rhd f : \{t \mid P(\widetilde{\rho})\} \to \tau \dashv_{\{P\}} S; \Delta_1 \qquad \Delta_1 \rhd e : \{t \mid S(P(\widetilde{\rho}))\} \dashv \phi_2; \Delta_2\end{array}}{\Delta \rhd f\ e : \tau \dashv \phi_2; \Delta_2} \text{ (B-App)}$$

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

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

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

Fig. 3. Type inference rules (for functions)

discovery in abstract model checking. Unlike model checking, however, we do not repeat the whole verification process; we just redo the fixed-point computation.

We use the following examples to illustrate how type inference works.

Example 1. $\mathtt{fun\ pred}\ x = \mathtt{assert}\ x > 0\ \mathtt{in}\ x - 1\ \mathtt{in\ assert}\ e_1 = \mathtt{pred}\ e_2\ \mathtt{in}\ ()$ 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`. Then we check $\Delta \rhd \mathtt{assert}\ x > 0\ \mathtt{in}\ x - 1 : \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\} \dashv \phi'; \Delta'$ for $\Delta = \Delta_0, \mathtt{pred}{:}\sigma, x{:}\mathtt{int}^{\rho_x}$, and obtain $\phi' = \rho_x > 0 \wedge Q(\widetilde{\rho}, \rho_x, \rho_x - 1)$. Here, $\Delta_0 = +{:}\langle\top \mid \mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \to \{\mathtt{int}^{\rho_3} \mid \rho_3 = \rho_1 + \rho_2\}\rangle, \ldots, \le{:}\langle\top \mid \mathtt{int}^{\rho_1} \times \mathtt{int}^{\rho_2} \to \{\mathtt{int}^{\rho_3} \mid \rho_3 = \rho_1 \le \rho_2\}\rangle, \ldots$ 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 $\mathtt{assert}\ e_1 = \mathtt{pred}\ e_2\ \mathtt{in}\ ()$ under $\Delta_1 = \Delta_0, \mathtt{pred} : (\sigma; \phi; \{P \mapsto \lambda\rho_x.\rho_x > 0, Q \mapsto \lambda(\rho_x, \rho_y).\top\})$. To check $\mathtt{pred}\ e_2$ against the type $\{\mathtt{int}^{\rho_y} \mid \rho = \rho_y\}$, the rule B-Refine is used. From $\sigma \leqslant \{\mathtt{int}^{\rho_x} \mid R(\rho_x)\} \to \{\mathtt{int}^{\rho_y} \mid \rho = \rho_y\} \dashv \psi$, we get $\psi = \forall\rho_x.R(\rho_x) \Rightarrow \exists\widetilde{\rho}.(P(\widetilde{\rho}, \rho_x) \wedge \forall\rho_y.(Q(\widetilde{\rho}, \rho_x, \rho_y) \Rightarrow \rho = \rho_y))$. 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

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

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`. Then we check $\Delta \rhd \mathtt{if}\ x \leq 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * fact\ (x-1) : \{\mathtt{int}^{\rho_y} \mid Q(\widetilde{\rho}, \rho_x, \rho_y)\} \dashv \phi'; \Delta'$ for $\Delta = \Delta_0, \mathtt{fact} : \sigma, x : \mathtt{int}^{\rho_x}$, 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. 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 $\mathtt{assert}\ \mathtt{fact}\ e > 0\ \mathtt{in}\ ()$ under $\Delta_1 = \Delta_0, \mathtt{fact} : (\sigma; \phi; \{P \mapsto \lambda\rho_x.\top, Q \mapsto \lambda(\rho_x, \rho_y).\top\})$. To check `fact` e against the type $\{\mathtt{int}^{\rho_y} \mid \rho_y > 0\}$, the rule B-Refine is used. From $\sigma \leqslant \{\mathtt{int}^{\rho_x} \mid R(\rho_x)\} \to \{\mathtt{int}^{\rho_y} \mid \rho_y > 0\} \dashv \psi$, we get $\psi = \forall\rho_x.R(\rho_x) \Rightarrow \exists\widetilde{\rho}.(P(\widetilde{\rho}, \rho_x) \wedge \forall\rho_y.(Q(\widetilde{\rho}, \rho_x, \rho_y) \Rightarrow \rho_y > 0))$. 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 the full paper [8] for the proof).

Theorem 1 (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.*

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 briefly discuss how to extend our type inference algorithm formalized in Section 3 with higher-order functions, parametric polymorphism, and algebraic data types. Interested readers are referred to the full paper [8] for the formalization of the extended algorithm.

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 $(\texttt{int} \to \texttt{int}) \to \texttt{int}$, one may be tempted to consider a template of the form: $\langle R_1(P_1, Q_1) \mid \langle P_1(\rho_1) \mid \texttt{int}^{\rho_1} \to \{\texttt{int}^{\rho_2} \mid Q_1(\rho_1, \rho_2)\}\rangle \to \{\texttt{int}^{\rho_3} \mid R_2(P_1, Q_1, \rho_3)\}\rangle$, 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 $\langle P_1(\rho_1) \mid \texttt{int}^{\rho_1} \to \{\texttt{int}^{\rho_2} \mid Q_1(\rho_1, \rho_2)\}\rangle \to \{\texttt{int}^{\rho_3} \mid Q_2(\rho_3)\}$ for $(\texttt{int} \to \texttt{int}) \to \texttt{int}$. 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, the following type of `map` would be inferred from the calling context $(\texttt{map}\ (\lambda x.x + 1)\ l) : \{\texttt{int}^w\ \texttt{list} \mid w \geq 0\}$:

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

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 $\forall\alpha, \beta.(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$ is assigned to `map`, which can be instantiated to $(\{\texttt{int}^x \mid P(x)\} \to \{\texttt{int}^y \mid Q(y)\}) \to \{\texttt{int}^z\ \texttt{list} \mid P(z)\} \to \{\texttt{int}^w\ \texttt{list} \mid Q(w)\}$ for any P and Q.

Algebraic Data Types. We require users to declare data type invariants 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 $\forall\alpha.\texttt{unit} \to \{\alpha\ \texttt{list}^\rho \mid \rho = 0\}$ and $\forall\rho.\texttt{unit} \to \{\texttt{ordlist}^{\rho_1} \mid \rho_1 = \rho\}$ (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, a list function may have the following four templates: $\{\ \langle P_1(\rho_x) \mid \texttt{int list}^{\rho_x} \to \{\texttt{int list}^{\rho_y} \mid Q_1(\rho_x, \rho_y)\}\rangle$, $\langle P_2(\rho_x) \mid \texttt{int list}^{\rho_x} \to \{\texttt{ordlist}^{\rho_y} \mid Q_2(\rho_x, \rho_y)\}\rangle$, $\langle P_3(\rho_x) \mid \texttt{ordlist}^{\rho_x} \to \{\texttt{int list}^{\rho_y} \mid Q_3(\rho_x, \rho_y)\}\rangle$, $\langle P_4(\rho_x) \mid \texttt{ordlist}^{\rho_x} \to \{\texttt{ordlist}^{\rho_y} \mid Q_4(\rho_x, \rho_y)\}\rangle\}$, which 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.

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 the type specification that `msort` should return a sorted list only.

In the experiment, `Nil` is defined as a constructor having two types: $\forall\alpha.\texttt{unit} \to \{\alpha\ \texttt{list}^{\rho} \mid \rho = 0\}$ and $\forall\rho.\texttt{unit} \to \{\texttt{ordlist}^{\rho_1} \mid \rho_1 = \rho\}$. `Cons` is defined as a constructor having two types: $\forall\alpha.\alpha \times \alpha\ \texttt{list}^{\rho_1} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho_1 + 1\}$ and $\langle \rho_1 \leq \rho_2 \mid \texttt{int}^{\rho_1} \times \texttt{ordlist}^{\rho_2} \to \{\texttt{ordlist}^{\rho_3} \mid \rho_3 = \rho_1\}\rangle$. Here, $\alpha\ \texttt{list}^{\rho}$ is the type of lists of length ρ, whose elements have the type α. $\texttt{ordlist}^{\rho}$ is the type of ordered lists, whose elements are integers greater than or equal to ρ. 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 $\{\texttt{ordlist}^{\rho} \mid \top\}$. The full paper [8] shows the whole code used in the experiment.

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

$$\texttt{insert} : \forall\rho.\langle \rho \leq \rho_1 \wedge \rho \leq \rho_2 \mid \texttt{int}^{\rho_1} \times \texttt{ordlist}^{\rho_2} \to \{\texttt{ordlist}^{\rho_3} \mid \rho \leq \rho_3\}\rangle,$$
$$\texttt{isort} : \texttt{int list} \to \texttt{ordlist}.$$

The type of `insert` means that `insert` returns a sorted list whose head is greater than or equal to the first argument or 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 $\texttt{int} \times \texttt{int list} \to \texttt{int list}$. 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:

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

Since the required type for `insert` $(x, \texttt{isort}\ xs')$ is $\{\texttt{ordlist}^{\rho} \mid \top\}$, the system first infers that $Q(\rho_1, \rho_2, \rho_3) \equiv \top$, 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 $\forall \rho_{ret}.Q(\widetilde{\rho'}, \rho_x, \rho_{ys}, \rho_{ret}) \Rightarrow \rho_y \leq \rho_{ret}$ holds. Thus, $Q(\rho, \rho_1, \rho_2, \rho_3)$ is strengthened to $\rho \leq \rho_3$. Then, the system successfully infers the input specification $P(\rho, \rho_1, \rho_2) \equiv \rho \leq \rho_1 \wedge \rho \leq \rho_2$ by propagating the output specification.

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 on 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. Table 2 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 [10] 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 2, and confirmed that similar types are inferred from them.

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 2.

function name	inferred specification	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`	$\forall\alpha.\forall\rho.\{\alpha\ \texttt{list}^{\rho_1} \mid \rho_1 > 0 \wedge \rho_1 = \rho + 1\} \to \{\alpha\ \texttt{list}^{\rho_2} \mid \rho_2 = \rho\}$	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 2. The call sites used to infer the specifications of the functions in Table 1. We collected them from existing programs written in OCaml.

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

6 Related Work

As already mentioned in Section 1, closely related to ours is the work on DML [4,5] and size inference [1,2,3].

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.

Size inference can automatically infer size relations between arguments and return values of functions [1,2,3]. A main difference is that the size inference tries to infer as precise 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 which include dependent types with datasort and index refinements [11,12], and generalized algebraic data types [13,14,15] have been introduced to practical programming languages so that non-trivial program invariants can be expressed as types [16,17]. *Partial* type inference in the spirit of local type inference [18] 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 [19]. 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.

Theorem provers such as Coq [20] can also be used for writing dependently typed programs [21,22]. Epigram [23] and Cayenne [24] support interactive development of dependently typed programs: a program template and sub-goals are automatically generated from a type. These systems greatly reduce users' burden of writing programs and types. However, these systems currently seem to be difficult to master for ordinary programmers without a knowledge of formal logic.

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 [25,26,7,27], the induction-iteration

method [28], on-demand loop invariant refinement by Leino [29], and constraint-based invariant generation which solves unknown parameters in invariant templates [30,31]. 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 refines 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.

Future work includes an extension of our system for producing better error messages. With the current system, when type inference fails, it is difficult for the user to judge whether the failure is due to a bug of the program, or the incompleteness of our type inference algorithm. Finding minimal unsatisfiable constraints as in [16] would be useful for producing better error messages.

Our type inference algorithm presented in this paper assumes that all the function definitions are available. To support separate type inference for each module, we have to let users declare module interface (i.e., dependent types of the exported functions). Some module interface may be, however, automatically generated as shown in the experiments in Section 5.2.

Acknowledgments

We thank anonymous reviewers for their comments.

References

1. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL 1996, pp. 410–423. ACM Press, New York (1996)
2. Chin, W.N., Khoo, S.C.: Calculating sized types. In: PEPM 2000, pp. 62–72. ACM Press, New York (1999)
3. Chin, W.N., Khoo, S.C., Xu, D.N.: Extending sized type with collection analysis. In: PEPM 2003, pp. 75–84. ACM Press, New York (2003)
4. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI 1998, pp. 249–257. ACM Press, New York (1998)
5. Xi, H., Pfenning, F.: Dependent types in practical programming. In: POPL 1999, pp. 214–227. ACM Press, New York (1999)
6. Knowles, K., Flanagan, C.: Type Reconstruction for General Refinement Types. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 505–519. Springer, Heidelberg (2007)
7. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: PLDI 2001, pp. 203–213. ACM Press, New York (2001)

8. Unno, H., Kobayashi, N.: On-demand refinement of dependent types (Full version) (January, 2008), http://web.yl.is.s.u-tokyo.ac.jp/~uhiro/
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, pp. 84–96. ACM Press, New York (1978)
10. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, Springer, Heidelberg (2008)
11. Dunfield, J.: Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University (September, 2002)
12. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL 2004, pp. 281–292. ACM Press, New York (2004)
13. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL 2003, pp. 224–235. ACM Press, New York (2003)
14. Pottier, F., Régis-Gianas, Y.: Stratified type inference for generalized algebraic data types. In: POPL 2006, pp. 232–244. ACM Press, New York (2006)
15. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP 2006, pp. 50–61. ACM Press, New York (2006)
16. Sulzmann, M., Voicu, R.: Language-based program verification via expressive types. Electronic Notes in Theoretical Computer Science 174(7), 129–147 (2007)
17. Kiselyov, O., Shan, C.c.: Lightweight static capabilities. Electronic Notes in Theoretical Computer Science 174(7), 79–104 (2007)
18. Pierce, B.C., Turner, D.N.: Local type inference. In: POPL 1998, pp. 252–265. ACM Press, New York (1998)
19. Flanagan, C.: Hybrid type checking. In: POPL 2006, pp. 245–256. ACM Press, New York (2006)
20. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development. Springer, Heidelberg (2004)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54. ACM Press, New York (2006)
22. Chlipala, A.: Modular development of certified program verifiers with a proof assistant. In: ICFP 2006, pp. 160–171. ACM Press, New York (2006)
23. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Manuscript (April, 2005)
24. Augustsson, L.: Cayenne – a language with dependent types. In: ICFP 1998: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 239–250. ACM Press, New York (1998)
25. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
26. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL 2002, pp. 191–202. ACM Press, New York (2002)
27. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002, pp. 58–70. ACM Press, New York (2002)
28. Suzuki, N., Ishihata, K.: Implementation of an array bound checker. In: POPL 1977, pp. 132–143. ACM Press, New York (1977)
29. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
30. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)
31. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI 2007, pp. 300–309. ACM Press, New York (2007)