Decidable Subclassing-Bounded Quantification

Juan Chen
Microsoft Research
One Microsoft Way
Redmond, WA 98052
juanchen@microsoft.com

ABSTRACT

Bounded quantification allows quantified types to specify subtyping bounds for the type variables they introduce. It has undecidable subtyping and type checking. This paper shows that subclassing-bounded quantification—type variables have subclassing bounds—has decidable type checking. The main difficulty is that, type variables can have either upper bounds *or lower bounds*, which complicates the minimal type property.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms: Languages

Keywords: Typed intermediate language, decidability, class and object encoding, bounded quantification

1. INTRODUCTION

Bounded quantification (System $F_{<:}$), where quantified types may specify subtyping bounds for the type variables they introduce (such as $\forall\alpha \leq \tau_1.\tau_2$), has been used extensively as the theoretical foundation for object-oriented languages [2]. However, difficulties, including decidability of the type checking [8], affect its practicality, especially as a base for typed intermediate languages in compilers of object-oriented languages.

Subtyping and thus type-checking of $F_{<:}$ have been proved undecidable [8]. The main problem results from the following subtyping rule:

$$\frac{\Delta \vdash \tau_1' \leq \tau_1 \quad \Delta, \alpha \leq \tau_1' \vdash \tau_2 \leq \tau_2'}{\Delta \vdash (\forall\alpha \leq \tau_1.\tau_2) \leq (\forall\alpha \leq \tau_1'.\tau_2')}$$

The bounds of type variables are in terms of subtyping. Bounds can be any types, even quantified types. Checking subtyping between τ_2 and τ_2' uses the assumption $\alpha \leq \tau_1'$, which may cause infinite loops [8].

This paper proves that a language LIL_C with subclassing-bounded quantification—quantified types give subclassing bounds to type variables—has decidable type checking. LIL_C (Low-level Intermediate Language with Classes) is a typed intermediate language for compiling Java-like object-oriented languages. It is lower level than Java bytecode: it can express object layouts and it decomposes primitives such as virtual method invocation and down cast. LIL_C aims to faithfully model standard implementation techniques of object-oriented features, using only simple constructs that compiler writers who are not type theorists can understand. Unlike traditional class and object encodings that compile classes away, LIL_C preserves class names and subclassing. The bounds of type variables in quantified types are in terms of subclassing, and can only be class names or other type variables. A previous paper explained LIL_C in details [4].

Subclassing-bounded quantification enables LIL_C to express covariant source arrays with invariant arrays, and to address displays of super classes (for casting) and even tables that store interface information for classes (*itables*).

This paper focuses on the decidability of type checking. For clarity, it describes only a simplified subset of LIL_C. The detailed proof for the full language is in the Appendix of [4]. The decidability proof follows the standard approach: we first prove the decidability of subtyping, then prove the minimal type property, that is, any well-typed expression has a minimal type such that all valid types for the expression are super types of the minimal type. The first step is trivial, because LIL_C constrains the bounds of type variables to only class names or type variables.

The challenge of proving the minimal type property lies in expressions that introduce new local type variables. The type variables cannot occur in the types of the expressions. LIL_C defines a type lifting operation to compute the "least" super type that does not have free occurrences of a type variable. The existence of type variables with lower bounds further complicates the definition of the lifting operation.

The main contributions of this paper include:

- It proves the decidability of subclassing-bounded quantification, including the decidability of subtyping, the minimal type property, and bounded joins and meets.
- LIL_C simplifies the type lifting operation, by explicitly specifying the shape of the kind environment (which contains in-scope type variables).
- The proof techniques can be extended to deal with interfaces. Interfaces bring multiple inheritance between classes/interfaces.

The rest of the paper is organized as follows: Section 2 briefly describes LIL_C. Section 3 shows details of the de-

TLDI'05, January 10, 2005, Long Beach, California, USA.

cidability proof. Section 4 explains how interfaces would change the proof. Section 5 discusses related work and Section 6 concludes.

2. INTRODUCTION OF LIL$_C$

LIL$_C$ preserves notions of classes and objects. Each class has a corresponding record type that describes the object layout of the class, including the runtime tag and the vtable. Objects are first coerced to records before field fetching and virtual method invocation. The coercion has no runtime overhead.

The notation "$C_1 \ll C_2$" represents that C_1 is a subclass of C_2. "$\ll$" preserves the subclassing relation defined in the source program. Unlike in source languages like Java, in LIL$_C$ a class name C represents only instances of exact C, not including C's subclasses. LIL$_C$ uses an existential type $\exists\alpha \ll C.\ \alpha$ to represent instances of C or C's subclasses. The dynamic types of those objects are abstracted by type variable α. The layout of an object of type $\exists\alpha \ll C.\ \alpha$ can be approximated by a record that contains all fields and methods declared in C. The key point is that, a type variable connects the object's dynamic type with the types of the tag and the "this" pointers in the approximation record. This way, type cast and dynamic dispatch are guaranteed safe.

Preserving class names and subclassing simplifies the type system. First, structural recursive types are not necessary because each record type can refer to any class name, including the class to which the record type corresponds. Second, the bounds of type variables must be classes or type variables, not arbitrary types.

The subclassing hierarchy consists of class names and type variables (Figure 1). The head of each arrow is a super class of the tail. The base structure is a tree formed by class names[1]. The introduction of a new type variable specifies either an upper or a lower bound. No new arrows connecting two existing nodes are allowed. Therefore, any two nodes have at most one path connecting them.

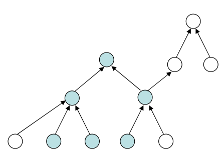

Figure 1: Subclassing Hierarchy (shaded nodes are class names, and unshaded are type variables.)

2.1 Syntax

Selected types and expressions of LIL$_C$ are as follows.

$$
\begin{array}{lcl}
\kappa & ::= & \Omega \mid \Omega_C \\
\tau & ::= & \ldots \mid C \mid \alpha \mid Tag(\tau) \mid \forall\alpha \ll \tau_1.\ \tau_2 \mid \exists\alpha \ll \tau_1.\ \tau_2 \\
 & & \mid (\tau_1, \ldots, \tau_n) \rightarrow \tau \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\
e & ::= & \ldots \mid x \mid \mathrm{tag}(C) \mid C(e) \mid \mathrm{c2r}(e) \mid \{l_i = e_i\}_{i=1}^{n} \mid e.l \\
 & & \mid e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) \mid (\alpha, x) = \mathrm{open}(e_1) \text{ in } e_2 \\
 & & \mid \text{pack } \tau_1 \text{ as } \alpha \ll u \text{ in } (e : \tau_2) \\
 & & \mid \mathrm{ifParent}(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2
\end{array}
$$

Kind Ω_C specifies class names and type variables that will be instantiated with class names. Kind Ω_C is a subkind of Ω, meaning that each type of kind Ω_C has kind Ω.

[1] We consider single inheritance between classes and discuss interfaces in Section 4.

We use B, C, and D to range over class names. Top$_C$ is a special class name, like *System.Object* in Microsoft .NET Framework. Each class is a subclass of Top$_C$. Type $Tag(\tau)$ represents the tag of type τ. Each tag uniquely identifies a class. LIL$_C$ treats tags as abstract.

LIL$_C$ has bounded universal types $\forall\alpha \ll \tau.\ \tau'$ and bounded existential types $\exists\alpha \ll \tau.\ \tau'$. Both give upper bounds to the type variables. The bound Top$_C$ is often omitted. Function types $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ and record types $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ are standard.

Non-standard expressions include: "tag(C)" represents the tag of class C; "$C(e)$" coerces a record e to an object of class C; "c2r(e)" coerces an object e to a record; "ifParent(e) then bind (α, x) in e_1 else e_2" tests whether tag e has a parent tag, and if so, it transfers the control to e_1 with x bound to the parent tag and α bound to the type the parent tag identifies, and to e_2 otherwise. The "ifParent" expression is used for casting. Each tag has a pointer to its parent tag. The pointers form a tag chain. LIL$_C$ implements downward cast by comparing each tag in the chain with the target tag.

The syntax of functions, class declarations and programs is omitted.

2.2 Static Semantics

A table Θ contains class declarations. A kind environment Δ collects all in-scope type variables and their bounds. An entry in Δ is either $\alpha \ll \tau$ or $\alpha \gg \tau$, introducing a new type variable α and its upper or lower bound τ. The upper bound Top$_C$ is often omitted. A type environment Γ maps variables to types. The well-formedness rules of types are straightforward. Selected kinding rules are as follows.

$$
\frac{C \in \mathrm{domain}(\Theta)}{\Theta; \Delta \vdash C : \Omega_C}
\quad
\frac{\alpha \in \mathrm{domain}(\Delta)}{\Theta; \Delta \vdash \alpha : \Omega_C}
\quad
\frac{\Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash Tag(\tau) : \Omega}
$$

$$
\frac{\Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash \tau : \Omega}
\quad
\frac{\Theta; \Delta \vdash \tau_i : \Omega\ \forall 1 \le i \le n}{\Theta; \Delta \vdash \{l_i : \tau_i\}_{i=1} : \Omega}
$$

$$
\frac{\Theta; \Delta \vdash \tau_i : \Omega\ \forall 1 \le i \le n \quad \Theta; \Delta \vdash \tau : \Omega}{\Theta; \Delta \vdash (\tau_1, \ldots, \tau_n) \rightarrow \tau : \Omega}
$$

$$
\frac{\Theta; \Delta \vdash \tau_1 : \Omega_C \quad \Theta; \Delta, \alpha \ll \tau_1 \vdash \tau_2 : \Omega}{\Theta; \Delta \vdash \forall\alpha \ll \tau_1.\ \tau_2 : \Omega}
\quad
\frac{\Theta; \Delta \vdash \tau_1 : \Omega_C \quad \Theta; \Delta, \alpha \ll \tau_1 \vdash \tau_2 : \Omega}{\Theta; \Delta \vdash \exists\alpha \ll \tau_1.\ \tau_2 : \Omega}
$$

The subclassing judgment $\Theta; \Delta \vdash \tau_1 \ll \tau_2$ means that, under environments Θ and Δ, τ_1 is a subclass of τ_2.

$$
\frac{\Theta(C) = C : B\{\ldots\}}{\Theta; \Delta \vdash C \ll B}
\quad
\frac{\Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash \tau \ll \mathrm{Top}_C}
\quad
\frac{\Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash \tau \ll \tau}
$$

$$
\frac{\Theta; \Delta \vdash \tau_1 \ll \tau_2 \quad \Theta; \Delta \vdash \tau_2 \ll \tau_3}{\Theta; \Delta \vdash \tau_1 \ll \tau_3}
\quad
\frac{\alpha \ll \tau \in \Delta \quad \Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash \alpha \ll \tau}
\quad
\frac{\alpha \gg \tau \in \Delta \quad \Theta; \Delta \vdash \tau : \Omega_C}{\Theta; \Delta \vdash \tau \ll \alpha}
$$

The subtyping judgment $\Theta; \Delta \vdash \tau_1 \le \tau_2$ means that, under environments Θ and Δ, τ_1 is a subtype of τ_2. Standard subtyping rules include: prefix and depth subtyping for record types, subtyping for function types, and reflexivity and transitivity rules.

Subtyping between quantified types are similar to the ones in Castagna and Pierce's work [3], but the latter lacks the minimal type property. LIL$_C$ has the minimal type property

$$\frac{}{\Theta;\Delta;\Gamma \vdash x : \Gamma(x)}\ \textbf{var} \qquad \frac{C \in \text{domain}(\Theta)}{\Theta;\Delta;\Gamma \vdash \text{tag}(C) : Tag(C)}\ \textbf{tag} \qquad \frac{\Theta;\Delta;\Gamma \vdash e_i : \tau_i\ \forall 1 \le i \le n}{\Theta;\Delta;\Gamma \vdash \{l_i = e_i\}_{i=1}^{n} : \{l_i : \tau_i\}_{i=1}^{n}}\ \textbf{record}$$

$$\frac{\Theta;\Delta;\Gamma \vdash e : \{l_i : \tau_i\}_{i=1}^{n} \quad 1 \le j \le n}{\Theta;\Delta;\Gamma \vdash e.l_j : \tau_j}\ \textbf{field} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash e : \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau \quad \sigma = t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m \\ \Theta;\Delta;\Gamma \vdash e_i : \tau_i[\sigma]\ \forall 1 \le i \le n \quad \Theta;\Delta \vdash t_i \ll u_i[\sigma] \forall 1 \le i \le m\end{array}}{\Theta;\Delta;\Gamma \vdash e[t_1, \ldots, t_m](e_1, \ldots, e_n) : \tau[\sigma]}\ \textbf{call}$$

$$\frac{\Theta;\Delta \vdash \tau_1 \ll u \quad \alpha \notin \text{domain}(\Delta) \quad \Theta;\Delta;\Gamma \vdash e : \tau_2[\tau_1/\alpha]}{\Theta;\Delta;\Gamma \vdash \text{pack } \tau_1 \text{ as } \alpha \ll u \text{ in } (e : \tau_2) : \exists \alpha \ll u.\ \tau_2}\ \textbf{pack} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash e_1 : \exists \beta \ll u.\ \tau_1 \quad \alpha \notin \text{domain}(\Delta) \\ \Theta;\Delta, \alpha \ll u;\Gamma, x : \tau_1[\alpha/\beta] \vdash e_2 : \tau_2 \quad \alpha \notin free(\tau_2)\end{array}}{\Theta;\Delta;\Gamma \vdash (\alpha, x) = \text{open}(e_1) \text{ in } e_2 : \tau_2}\ \textbf{open}$$

$$\frac{\Theta;\Delta;\Gamma \vdash e : R(C)}{\Theta;\Delta;\Gamma \vdash C(e) : C}\ \textbf{obj} \qquad \frac{\Theta;\Delta;\Gamma \vdash e : C}{\Theta;\Delta;\Gamma \vdash \text{c2r}(e) : R(C)}\ \textbf{c2r\_c} \qquad \frac{\Theta;\Delta;\Gamma \vdash e : \alpha \quad \Theta;\Delta \vdash \alpha \ll C}{\Theta;\Delta;\Gamma \vdash \text{c2r}(e) : \text{Approx}R(\alpha, C)}\ \textbf{c2r\_tv}$$

$$\frac{\Theta;\Delta;\Gamma \vdash e : \tau_1 \quad \Theta;\Delta \vdash \tau_1 \le \tau_2}{\Theta;\Delta;\Gamma \vdash e : \tau_2}\ \textbf{sub} \qquad \frac{\Theta;\Delta;\Gamma \vdash e : Tag(s) \quad \alpha \notin \text{domain}(\Delta) \quad \Theta;\Delta, \alpha \gg s;\Gamma, x : Tag(\alpha) \vdash e_1 : \tau \quad \Theta;\Delta;\Gamma \vdash e_2 : \tau}{\Theta;\Delta;\Gamma \vdash \text{ifParent}(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2 : \tau}\ \textbf{ifParent}$$

Figure 2: Selected Expression Typing Rules

because it uses subclassing-bounded quantification and subclassing does not imply subtyping: $C_1 \ll C_2$ does *not* imply $C_1 \le C_2$. LIL$_C$ does not generalize the subtyping rules as in Full F$_{<:}$, to avoid complications with joins/meets of types.

$$\frac{\Theta;\Delta \vdash u \ll u' \quad \Theta;\Delta, \alpha \ll \text{Top}_C \vdash \tau \le \tau'}{\Theta;\Delta \vdash (\exists \alpha \ll u.\ \tau) \le (\exists \alpha \ll u'.\ \tau')}$$

$$\frac{\Theta;\Delta \vdash u' \ll u \quad \Theta;\Delta, \alpha \ll \text{Top}_C \vdash \tau \le \tau'}{\Theta;\Delta \vdash (\forall \alpha \ll u.\ \tau) \le (\forall \alpha \ll u'.\ \tau')}$$

The subclassing polymorphism says that, an object of C_1 and C_1's subclasses can be used wherever an object of C_2 and C_2's subclasses is needed. It is expressed by: $(\exists \alpha \ll C_1.\ \alpha) \le (\exists \alpha \ll C_2.\ \alpha)$. Note that $\exists \alpha \ll C_1.\ \alpha$ represents objects of C_1 and C_1's subclasses, and $\exists \alpha \ll C_2.\ \alpha$ represents objects of C_2 and C_2's subclasses.

Figure 2 shows selected expression typing rules. In **ifParent**, expression e must be a tag of some type s. In checking the true branch e_1 where s has a parent tag, a new type variable α with lower bound s is introduced, and a new variable x indicates the parent tag.

R and $ApproxR$ in rules **obj**, **c2r_c** and **c2r_tv** are macros used by the type checker. $R(C)$ is a record type describing the object layout of class C. $ApproxR(\alpha, C)$ is a record type approximating the layout of α, which is a subclass of C.

The decidability proof requires the following property of the approximation:

Lemma 1. *Property of approximation: if* $\Theta;\Delta \vdash C_1 \ll C_2$ *and* $\Theta;\Delta \vdash \alpha : \Omega_C$, *then* $\Theta;\Delta \vdash ApproxR(\alpha, C_1) \le ApproxR(\alpha, C_2)$.

The definition of Approx$R$ in LIL$_C$ satisfies the property. Suppose a class C has fields $f_1 : s_1, \ldots, f_n : s_n$ and methods $m_1 : t_1, \ldots, m_k : t_k$:

$$\begin{aligned}\text{Approx}R(\alpha, C) = \{&vtable : \{tag : Tag(\alpha),\\ &\quad m_1 : addThis(\exists \gamma \ll \alpha.\ \gamma, t_1),\\ &\quad \ldots,\\ &\quad m_k : addThis(\exists \gamma \ll \alpha.\ \gamma, t_k)\},\\ &f_1^M : s_1, \ldots, f_n^M : s_n\}\\ \text{Approx}R(\alpha, \text{Top}_C) = \{&vtable : \{tag : Tag(\alpha)\}\}\end{aligned}$$

The function $addThis(\tau_{this}, \tau)$ adds τ_{this}, the type for the "this" pointer, to a function type τ:

$$\begin{aligned}&addThis(\tau_{this}, \forall\ tvs\ (\tau_1, \ldots, \tau_n) \to \tau)\\ &= \forall\ tvs(\tau_{this}, \tau_1, \ldots, \tau_n) \to \tau\end{aligned}$$

In Approx$R(\alpha, C)$, the tag has type $Tag(\alpha)$ and the "this" pointer has type $\exists \gamma \ll \alpha.\ \gamma$. Approx$R(\alpha, \text{Top}_C)$ contains only a vtable and the vtable contains only the tag of α.

3. DECIDABILITY OF TYPE CHECKING

Section 3.1 proves the decidability of subtyping. Section 3.2 proves the minimal type property. The main decidability theorem (Theorem 25) follows.

We first present some subclassing properties. Basically, these properties specify the super classes or subclasses of the last type variable in Δ:

Lemma 2. *Properties of subclassing:*

1. *If* $\Theta;\Delta, \gamma \ll u \vdash \tau \ll \gamma$, *then* $\tau = \gamma$.
2. *If* $\Theta;\Delta, \gamma \gg u \vdash \gamma \ll \tau$, *then* $\tau = \gamma$ *or* Top_C.
3. *If* $\Theta;\Delta' \vdash \tau_1 \ll \tau_2$, $\Delta' = \Delta, \gamma \ll \tau$ *or* $\Delta' = \Delta, \gamma \gg \tau$, *and* $\tau_1 \ne \gamma$, $\tau_2 \ne \gamma$, *then* $\Theta;\Delta \vdash \tau_1 \ll \tau_2$.
4. *If* $\Theta;\Delta, \alpha \ll u \vdash \alpha \ll \tau$, $\tau \ne \alpha$, *then* $\Theta;\Delta \vdash u \ll \tau$.
5. *If* $\Theta;\Delta, \alpha \gg u \vdash \tau \ll \alpha$, $\tau \ne \alpha$, *then* $\Theta;\Delta \vdash \tau \ll u$.

Proof: by induction on subclassing rules. □

Lemma 3. *Subclassing is decidable.*

Proof: Subclassing is a partial order on a finite set of class names and type variables. □

3.1 Decidability of Subtyping

To prove that subtyping is decidable, we develop a set of algorithmic (syntax-directed) subtyping rules $\Theta;\Delta \vDash \tau_1 \le \tau_2$ (Figure 3), and prove that the new rules are equivalent to the original subtyping rules. The new rules push transitivity into individual cases, so that the transitivity rule is no longer necessary.

$$\frac{m \geq n \quad \Theta;\Delta \vDash \tau_i \leq \tau_i' \ \forall 1 \leq i \leq n \quad \Theta;\Delta \vdash \tau_i : \Omega \ \forall 1 \leq i \leq m}{\Theta;\Delta \vDash \{l_i : \tau_i\}_{i=1}^m \leq \{l_i : \tau_i'\}_{i=1}^n} \ \textbf{ast\_rec} \qquad \frac{\Theta;\Delta \vDash \tau_i' \leq \tau_i \forall 1 \leq i \leq n \quad \Theta;\Delta \vDash \tau \leq \tau'}{\Theta;\Delta \vDash (\tau_1,\ldots,\tau_n) \rightarrow \tau \leq (\tau_1',\ldots,\tau_n') \rightarrow \tau'} \ \textbf{ast\_fun}$$

$$\frac{\Theta;\Delta \vdash \tau : \Omega}{\Theta;\Delta \vDash \tau \leq \tau} \ \textbf{ast\_ref} \qquad \frac{\Theta;\Delta \vdash u \ll u' \quad \Theta;\Delta,\alpha \vDash \tau \leq \tau'}{\Theta;\Delta \vDash (\exists \alpha \ll u.\ \tau) \leq (\exists \alpha \ll u'.\ \tau')} \ \textbf{ast\_}\exists \qquad \frac{\Theta;\Delta \vdash u' \ll u \quad \Theta;\Delta,\alpha \vDash \tau \leq \tau'}{\Theta;\Delta \vDash (\forall \alpha \ll u.\ \tau) \leq (\forall \alpha \ll u'.\ \tau')} \ \textbf{ast\_}\forall$$

Figure 3: Algorithmic Subtyping Rules

Lemma 4 (Soundness). *If* $\Theta;\Delta \vDash T_1 \leq T_2$*, then* $\Theta;\Delta \vdash T_1 \leq T_2$.

Proof: by induction on the algorithmic subtyping rules. □

Lemma 5 (Transitivity). *If* $\Theta;\Delta \vDash T_1 \leq T_2$ *and* $\Theta;\Delta \vDash T_2 \leq T_3$*, then* $\Theta;\Delta \vDash T_1 \leq T_3$.

Proof: by induction on the sum of the sizes of the derivations $\Theta;\Delta \vDash T_1 \leq T_2$ and $\Theta;\Delta \vDash T_2 \leq T_3$. If either derivation ends with **ast_ref**, then $\Theta;\Delta \vDash T_1 \leq T_3$ is simply the conclusion of the other derivation. Cases where both derivations end with **ast_rec**, **ast_∃**, **ast_∀**, or **ast_fun** can be proved by induction. There are no other cases. □

Lemma 6 (Completeness). *If* $\Theta;\Delta \vdash T_1 \leq T_2$*, then* $\Theta;\Delta \vDash T_1 \leq T_2$.

Proof: by induction on the subtyping rules. The transitivity case is proved by Lemma 5. □

Lemma 7. *The algorithmic subtyping rules terminate.*

Proof: the decreasing metric is the sum of sizes of the types in the subtyping judgment. Other side conditions (subclassing) is decidable too. □

Corollary 8. *It is decidable to check whether* $\Theta;\Delta \vdash \tau_1 \leq \tau_2$ *holds.*

Proof: by Lemmas 4, 6 and 7. □

3.2 Minimal Types

We define a set of algorithmic typing rules $\Theta;\Delta;\Gamma \vDash e : \tau$ to compute the minimal type of each expression (Figure 4), and prove that the new rules are equivalent to the rules in Figure 2.

The key difficulties are the "open" and "ifParent" expressions. The open expression "$(\alpha, x) = \text{open}(e_1)$ in e_2" introduces a new type variable α, whose scope is e_2. In the original typing rule, if e_2 has a type τ where α is not free, then the open expression has type τ. Naturally, the minimal type of the open expression should come from the minimal type of e_2. But the minimal type of e_2 might contain free occurrences of α. Because α is invisible outside e_2, α should not appear free in the minimal type of the open expression. We define the minimal type of the open expression to be the least super type of the minimal type of e_2 that contains no free occurrences of α. For this purpose, a type lifting operation $(\tau) \Uparrow_\Delta^\alpha$ computes the least super type of τ under environment Δ that contains no free occurrences of α. The minimal type of the "open" expression is the result of lifting α out of the minimal type of e_2 under the new environment with α.

When computing $(\exists \beta \ll \alpha.\ \tau) \Uparrow_\Delta^\alpha$, we need to compute the least *super class* of α that contains no free occurrences of α, according to the subtyping rule on quantified existential types. This is challenging for arbitrary Δ, because α might have more than one super classes in Δ. For example, environment "$\alpha \ll C, \gamma \gg \alpha$" introduces two variables α and γ. α has two super classes, C and γ. The least super class of α should be like a union type $C \cup \gamma$. Similarly, intersection types for bounded universal types are necessary. General notions of union and intersection types on both class names and type variables complicate subclassing, type lifting and joins/meets operations, thus are undesirable.

We observe that $(\tau) \Uparrow_\Delta^\alpha$ is used only when α appears last in Δ. This means that α can have only a super class or a subclass in Δ. Therefore, union types or intersection types are unnecessary. We define $(\tau) \Uparrow_\Delta^\alpha$ only when α is the last entry in Δ.

Expression "ifParent(e) then bind (α, x) in e_1 else e_2" introduces a new type variable α whose scope is e_1. The algorithmic typing rule needs type lifting for the type of e_1, as in open expressions. Also, the minimal type of the whole expression is the least upper bound (join) of the minimal types of the two branches.

We also define an operation $\tau \uparrow_\Delta$ to compute the least super class name for type τ in environment Δ. It is used in **a_c2r_tv**, because the approximation needs a concrete class name. The type variable lifting is defined as follows.

Definition 9. *Class and type variable lifting*

$$\begin{array}{rcl} C \uparrow_\Delta & = & C \\ \alpha \uparrow_\Delta & = & \begin{cases} \tau \uparrow_\Delta & \alpha \ll \tau \in \Delta \\ Top_C & \alpha \gg \tau \in \Delta \end{cases} \end{array}$$

By definition, $\tau \uparrow_\Delta$ is a class name. The process of computing $\tau \uparrow_\Delta$ always terminates because there is no loop in Δ. Also it has the following properties:

Lemma 10. *Properties of class and type variable lifting*

1. $\Theta;\Delta \vdash \gamma \ll \gamma \uparrow_\Delta$.
2. *If* $\Theta;\Delta \vdash \tau_1 \ll \tau_2$*, then* $\Theta;\Delta \vdash \tau_1 \uparrow_\Delta \ll \tau_2 \uparrow_\Delta$.

Proof: *1*: by induction on depth where depth is defined as: depth$(C) = 0$, depth$(\alpha) =$ depth$(\tau) + 1$ if $\alpha \ll \tau \in \Delta$, depth$(\alpha) = 0$ if $\alpha \gg \tau \in \Delta$.

2: by induction on subclassing rules. □

In the rest of this section, Section 3.2.1 defines type lifting. Section 3.2.2 defines joins and meets. Section 3.2.3 proves the minimal type property.

3.2.1 Type Lifting and Lowering

Figures 5 defines type lifting and type lowering. Lowering is necessary because of contravariant function argument types. Lifting and lowering are mutually recursively defined. They have the following properties:

$$\frac{}{\Theta;\Delta;\Gamma \vDash x : \Gamma(x)}\ \textbf{a\_var} \qquad \frac{C \in \mathrm{domain}(\Theta)}{\Theta;\Delta;\Gamma \vDash \mathrm{tag}(C) : Tag(C)}\ \textbf{a\_tag} \qquad \frac{\Theta;\Delta;\Gamma \vDash e_i : \tau_i\ \forall 1 \le i \le n}{\Theta;\Delta;\Gamma \vDash \{l_i = e_i\}_{i=1}^{n} : \{l_i : \tau_i\}_{i=1}^{n}}\ \textbf{a\_record}$$

$$\frac{\Theta;\Delta;\Gamma \vDash e : \{l_i : \tau_i\}_{i=1}^{n} \quad 1 \le i \le n}{\Theta;\Delta;\Gamma \vDash e.l_i : \tau_i}\ \textbf{a\_field} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Gamma \vDash e : \forall\alpha_1 \ll u_1,\ldots,\alpha_m \ll u_m.(\tau_1,\ldots,\tau_n) \to \tau \quad \sigma = t_1,\ldots,t_m/\alpha_1,\ldots,\alpha_m \\ \Theta;\Delta;\Gamma \vDash e_i : \tau_{mi} \quad \Theta;\Delta \vdash \tau_{mi} \le \tau_i[\sigma]\ \forall 1 \le i \le n \quad \Theta;\Delta \vdash t_i \ll u_i[\sigma] \forall 1 \le i \le m\end{array}}{\Theta;\Delta;\Gamma \vDash e[t_1,\ldots,t_m](e_1,\ldots,e_n) : \tau[\sigma]}\ \textbf{a\_call}$$

$$\frac{\begin{array}{c}\Theta;\Delta \vdash \tau_1 \ll u \qquad \alpha \notin \mathrm{domain}(\Delta) \\ \Theta;\Delta;\Gamma \vDash e : \tau_m \qquad \Theta;\Delta \vdash \tau_m \le \tau_2[\tau_1/\alpha]\end{array}}{\Theta;\Delta;\Gamma \vDash \text{pack } \tau_1 \text{ as } \alpha \ll u \text{ in } (e : \tau_2) : \exists\alpha \ll u.\ \tau_2}\ \textbf{a\_pack} \qquad \frac{\begin{array}{c}\Theta;\Delta;\Gamma \vDash e_1 : \exists\beta \ll u.\ \tau_1 \qquad \alpha \notin \mathrm{domain}(\Delta) \\ \Theta;\Delta,\alpha \ll u;\Gamma, x : \tau_1[\alpha/\beta] \vDash e_2 : \tau_m\end{array}}{\Theta;\Delta;\Gamma \vDash (\alpha, x) = \mathrm{open}(e_1) \text{ in } e_2 : (\tau_m) \Uparrow^{\alpha}_{\Delta,\alpha \ll u}}\ \textbf{a\_open}$$

$$\frac{\Theta;\Delta;\Gamma \vDash e : \tau \quad \Theta;\Delta \vdash \tau \le R(C)}{\Theta;\Delta;\Gamma \vDash C(e) : C}\ \textbf{a\_obj} \qquad \frac{\Theta;\Delta;\Gamma \vDash e : C}{\Theta;\Delta;\Gamma \vDash \mathrm{c2r}(e) : R(C)}\ \textbf{a\_c2r\_c} \qquad \frac{\Theta;\Delta;\Gamma \vDash e : \alpha}{\Theta;\Delta;\Gamma \vDash \mathrm{c2r}(e) : \mathrm{Approx}R(\alpha, \alpha \uparrow_{\Delta})}\ \textbf{a\_c2r\_tv}$$

$$\frac{\Theta;\Delta;\Gamma \vDash e : Tag(s) \quad \alpha \notin \mathrm{domain}(\Delta) \quad \Theta;\Delta,\alpha \gg s;\Gamma, x : Tag(\alpha) \vDash e_1 : \tau_1 \quad \Theta;\Delta;\Gamma \vDash e_2 : \tau_2}{\Theta;\Delta;\Gamma \vDash \text{ifParent}(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2 : (\tau_1) \Uparrow^{\alpha}_{\Delta,\alpha \gg s} \vee_{\Delta} \tau_2}\ \textbf{a\_ifParent}$$

Figure 4: Algorithmic Expression Typing Rules

Suppose $\Delta = \Delta_1, P(\alpha)$ where $P(\alpha) = \alpha \ll u_\alpha$ or $P(\alpha) = \alpha \gg u_\alpha$:

$$\begin{array}{lcll}
(\tau) \Uparrow^{\alpha}_{\Delta} & = & \tau & \alpha \notin free(\tau) \\
(\exists\beta \ll u.\ \tau) \Uparrow^{\alpha}_{\Delta} & = & \exists\beta \ll u.\ \tau' & u \ne \alpha \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\exists\beta \ll \alpha.\ \tau) \Uparrow^{\alpha}_{\Delta} & = & \exists\beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\exists\beta \ll \alpha.\ \tau) \Uparrow^{\alpha}_{\Delta} & = & \exists\beta \ll \mathrm{Top}_C.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\forall\beta \ll u.\ \tau) \Uparrow^{\alpha}_{\Delta} & = & \forall\beta \ll u.\ \tau' & u \ne \alpha \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\forall\beta \ll \alpha.\ \tau) \Uparrow^{\alpha}_{\Delta} & = & \forall\beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
((\tau_1,\ldots,\tau_n) \to \tau) \Uparrow^{\alpha}_{\Delta} & = & (\tau'_1,\ldots,\tau'_n) \to \tau' & \tau'_i = (\tau_i) \Downarrow^{\alpha}_{\Delta} \text{ and } \tau' = (\tau) \Uparrow^{\alpha}_{\Delta} \\
(\{l_i : \tau_i\}_{i=1}^{n}) \Uparrow^{\alpha}_{\Delta} & = & \{l_i : \tau'_i\}_{i=1}^{p} & p = max\{i \mid (\tau_j) \Uparrow^{\alpha}_{\Delta} \text{ exists } \forall 1 \le j \le i\} \quad \tau'_i = (\tau_i) \Uparrow^{\alpha}_{\Delta}\ \forall 1 \le i \le p
\end{array}$$

$$\begin{array}{lcll}
(\tau) \Downarrow^{\alpha}_{\Delta} & = & \tau & \alpha \notin free(\tau) \\
(\exists\beta \ll u.\ \tau) \Downarrow^{\alpha}_{\Delta} & = & \exists\beta \ll u.\ \tau' & u \ne \alpha \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\exists\beta \ll \alpha.\ \tau) \Downarrow^{\alpha}_{\Delta} & = & \exists\beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\forall\beta \ll u.\ \tau) \Downarrow^{\alpha}_{\Delta} & = & \forall\beta \ll u.\ \tau' & u \ne \alpha \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\forall\beta \ll \alpha.\ \tau) \Downarrow^{\alpha}_{\Delta} & = & \forall\beta \ll \mathrm{Top}_C.\ \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
(\forall\beta \ll \alpha.\ \tau) \Downarrow^{\alpha}_{\Delta} & = & \forall\beta \ll u_\alpha.\ \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta_1,\beta,P(\alpha)} \\
((\tau_1,\ldots,\tau_n) \to \tau) \Downarrow^{\alpha}_{\Delta} & = & (\tau'_1,\ldots,\tau'_n) \to \tau' & \tau'_i = (\tau_i) \Uparrow^{\alpha}_{\Delta} \text{ and } \tau' = (\tau) \Downarrow^{\alpha}_{\Delta} \\
(\{l_i : \tau_i\}_{i=1}^{n}) \Downarrow^{\alpha}_{\Delta} & = & \{l_i : \tau'_i\}_{i=1}^{n} & \tau'_i = (\tau_i) \Downarrow^{\alpha}_{\Delta}\ \forall 1 \le i \le n
\end{array}$$

Figure 5: Definitions of Type Lifting and Lowering

LEMMA 11. *let* $\Delta' = \Delta, \alpha \ll u_\alpha$ *or* $\Delta, \alpha \gg u_\alpha$.

1. *If* $\exists \tau'$, $\alpha \notin free(\tau')$ *and* $\Theta; \Delta' \vdash \tau \leq \tau'$, *then* $(\tau) \Uparrow^{\alpha}_{\Delta'}$ *exists, and* $\Theta; \Delta' \vdash \tau \leq (\tau) \Uparrow^{\alpha}_{\Delta'}$, *and* $\forall s$ *such that* $\Theta; \Delta' \vdash \tau \leq s$ *and* $\alpha \notin free(s)$, $\Theta; \Delta \vdash (\tau) \Uparrow^{\alpha}_{\Delta'} \leq s$.

2. *If* $\exists \tau'$, $\alpha \notin free(\tau')$ *and* $\Theta; \Delta' \vdash \tau' \leq \tau$, *then* $(\tau) \Downarrow^{\alpha}_{\Delta'}$ *exists and* $\Theta; \Delta' \vdash (\tau) \Downarrow^{\alpha}_{\Delta'} \leq \tau$, *and* $\forall s$ *such that* $\Theta; \Delta' \vdash s \leq \tau$ *and* $\alpha \notin free(s)$, $\Theta; \Delta \vdash s \leq (\tau) \Downarrow^{\alpha}_{\Delta'}$.

Proof: prove the two parts simultaneously by induction on structure of types. □

3.2.2 Joins and Meets

The (subtyping) joins/meets of quantified types require (subclassing) joins/meets of class names and type variables. We first define the latter.

3.2.2.1 Classes and Type Variables.

LEMMA 12. *If* C, C_1 *and* C_2 *are class names,* $\Theta; \bullet \vdash C \ll C_1$ *and* $\Theta; \bullet \vdash C \ll C_2$, *then either* $\Theta; \bullet \vdash C_1 \ll C_2$ *or* $\Theta; \bullet \vdash C_2 \ll C_1$.

Proof: by induction on the sum of the sizes of the left derivation $\Theta; \bullet \vdash C \ll C_1$ and the right derivation $\Theta; \bullet \vdash C \ll C_2$. Examine the last rule in the left derivation.

We only list one case here when the left derivation ends with the transitivity rule. That is, $\exists C_1'$ such that $\Theta; \bullet \vdash C \ll C_1'$ and $\Theta; \bullet \vdash C_1' \ll C_1$. By induction hypothesis, either $\Theta; \bullet \vdash C_1' \ll C_2$ or $\Theta; \bullet \vdash C_2 \ll C_1'$ because C_1' and C_2 have common lower bound C.

If $\Theta; \bullet \vdash C_1' \ll C_2$, then by induction hypothesis again, either $\Theta; \bullet \vdash C_1 \ll C_2$ or $\Theta; \bullet \vdash C_2 \ll C_1$ because C_1 and C_2 have common lower bound C_1'. If $\Theta; \bullet \vdash C_2 \ll C_1'$, then by the transitivity of subclassing $\Theta; \bullet \vdash C_2 \ll C_1$. □

We now define the subclassing joins $(\tau_1 \curlyvee_\Delta \tau_2)$ and meets $(\tau_1 \curlywedge_\Delta \tau_2)$ of τ_1 and τ_2 under environment Δ:

DEFINITION 13. $\tau_1 \curlyvee_\Delta \tau_2$ *and* $\tau_1 \curlywedge_\Delta \tau_2$ *are defined by induction on the kind environment.*

- *If* Δ *is empty, then* τ_1 *and* τ_2 *are both concrete class names.* $\tau_1 \curlyvee_\Delta \tau_2$ *is the least common ancestor of* τ_1 *and* τ_2, *which always exists because we allow only single inheritance on class names and each class name is a subclass of* Top_C. $\tau_1 \curlywedge_\Delta \tau_2$ *exists only when* τ_1 *and* τ_2 *have a common lower bound. Lemma 12 guarantees that if* τ_1 *and* τ_2 *have a common lower bound, then* $\tau_1 \curlywedge_\Delta \tau_2$ *exists. If* $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, *then* $\tau_1 \curlywedge_\Delta \tau_2 = \tau_1$.

- *If* $\Delta' = \Delta, \alpha \ll \tau$, $\Theta; \Delta \vdash \tau_1 : \Omega_C$ *and* $\Theta; \Delta \vdash \tau_2 : \Omega_C$, *then*

$$
\begin{array}{lcl}
\alpha \curlyvee_{\Delta'} \alpha & = & \alpha \\
\alpha \curlyvee_{\Delta'} \tau_2 & = & \tau \curlyvee_\Delta \tau_2 \\
\tau_1 \curlyvee_{\Delta'} \alpha & = & \tau_1 \curlyvee_\Delta \tau \\
\tau_1 \curlyvee_{\Delta'} \tau_2 & = & \tau_1 \curlyvee_\Delta \tau_2 \\
\\
\alpha \curlywedge_{\Delta'} \alpha & = & \alpha \\
\alpha \curlywedge_{\Delta'} \tau_2 & = & \alpha \quad \quad if\ \Theta; \Delta \vdash \tau \ll \tau_2 \\
\tau_1 \curlywedge_{\Delta'} \alpha & = & \alpha \quad \quad if\ \Theta; \Delta \vdash \tau \ll \tau_1 \\
\tau_1 \curlywedge_{\Delta'} \tau_2 & = & \tau_1 \curlywedge_\Delta \tau_2
\end{array}
$$

- *If* $\Delta' = \Delta, \alpha \gg \tau$, $\Theta; \Delta \vdash \tau_1 : \Omega_C$ *and* $\Theta; \Delta \vdash \tau_2 : \Omega_C$, *then*

$$
\begin{array}{lcl}
\alpha \curlyvee_{\Delta'} \alpha & = & \alpha \\
\alpha \curlyvee_{\Delta'} \tau_2 & = & \begin{cases} \alpha & if\ \Theta; \Delta \vdash \tau_2 \ll \tau \\ Top_C & otherwise \end{cases} \\
\tau_1 \curlyvee_{\Delta'} \alpha & = & \begin{cases} \alpha & if\ \Theta; \Delta \vdash \tau_1 \ll \tau \\ Top_C & otherwise \end{cases} \\
\tau_1 \curlyvee_{\Delta'} \tau_2 & = & \tau_1 \curlyvee_\Delta \tau_2 \\
\\
\alpha \curlywedge_{\Delta'} \tau_1 & = & \alpha \quad \quad if\ \tau_1 = \alpha\ or\ Top_C \\
Top_C \curlywedge_{\Delta'} \alpha & = & \alpha \\
\alpha \curlywedge_{\Delta'} \tau_2 & = & \tau \curlywedge_\Delta \tau_2 \\
\tau_1 \curlywedge_{\Delta'} \alpha & = & \tau \curlywedge_\Delta \tau_1 \\
\tau_1 \curlywedge_{\Delta'} \tau_2 & = & \tau_1 \curlywedge_\Delta \tau_2
\end{array}
$$

$\tau_1 \curlyvee_\Delta \tau_2$ and $\tau_1 \curlywedge_\Delta \tau_2$ have the following properties:

LEMMA 14. $\Theta; \Delta \vdash \tau_1 \ll \tau_1 \curlyvee_\Delta \tau_2$ *and* $\Theta; \Delta \vdash \tau_2 \ll \tau_1 \curlyvee_\Delta \tau_2$, *and* $\forall s$ *such that* $\Theta; \Delta \vdash \tau_1 \ll s$ *and* $\Theta; \Delta \vdash \tau_2 \ll s$, $\Theta; \Delta \vdash \tau_1 \curlyvee_\Delta \tau_2 \ll s$.

If $\Theta; \Delta \vdash \tau \ll \tau_1$ *and* $\Theta; \Delta \vdash \tau \ll \tau_2$, *then* $\tau_1 \curlywedge_\Delta \tau_2$ *is defined and* $\Theta; \Delta \vdash \tau_1 \curlywedge_\Delta \tau_2 \ll \tau_1$ *and* $\Theta; \Delta \vdash \tau_1 \curlywedge_\Delta \tau_2 \ll \tau_2$, *and* $\forall s$ *such that* $\Theta; \Delta \vdash s \ll \tau_1$ *and* $\Theta; \Delta \vdash s \ll \tau_2$, $\Theta; \Delta \vdash s \ll \tau_1 \curlywedge_\Delta \tau_2$.

Proof: by induction on kind environment and by properties of subclassing Lemma 2. □

3.2.2.2 Types.

We define the subtyping joins/meets of two types in Figure 6. They have the following properties:

LEMMA 15. *If* $\Theta; \Delta \vdash \tau_1 : \Omega$ *and* $\Theta; \Delta \vdash \tau_2 : \Omega$, *then:*

If $\exists \tau$ *such that* $\Theta; \Delta \vdash \tau_1 \leq \tau$ *and* $\Theta; \Delta \vdash \tau_2 \leq \tau$, *then* $\tau_1 \vee_\Delta \tau_2$ *exists,, and* $\Theta; \Delta \vdash \tau_1 \leq \tau_1 \vee_\Delta \tau_2$, *and* $\Theta; \Delta \vdash \tau_2 \leq \tau_1 \vee_\Delta \tau_2$, *and* $\forall s$ *such that* $\Theta; \Delta \vdash \tau_1 \leq s$ *and* $\Theta; \Delta \vdash \tau_2 \leq s$, $\Theta; \Delta \vdash \tau_1 \vee_\Delta \tau_2 \leq s$.

If $\Theta; \Delta \vdash \tau \leq \tau_1$ *and* $\Theta; \Delta \vdash \tau \leq \tau_2$, *then* $\tau_1 \wedge_\Delta \tau_2$ *exists, and* $\Theta; \Delta \vdash \tau_1 \wedge_\Delta \tau_2 \leq \tau_1$ *and* $\Theta; \Delta \vdash \tau_1 \wedge_\Delta \tau_2 \leq \tau_2$, *and* $\forall s$ *such that* $\Theta; \Delta \vdash s \leq \tau_1$ *and* $\Theta; \Delta \vdash s \leq \tau_2$, $\Theta; \Delta \vdash s \leq \tau_1 \wedge_\Delta \tau_2$.

Proof: by simultaneous induction on the definitions of joins and meets. □

3.2.3 The Minimal Type Property

We prove the soundness and completeness of the algorithmic typing rules. Appendix B lists details of interesting cases.

LEMMA 16 (SOUNDNESS). *If* $\Theta; \Delta; \Gamma \vDash E : T$, *then* $\Theta; \Delta; \Gamma \vdash E : T$.

Proof sketch: by induction on algorithmic typing rules. □

LEMMA 17 (COMPLETENESS). *If* $\Theta; \Delta; \Gamma \vdash E : T$, *then* $\exists T_m$ *such that* $\Theta; \Delta; \Gamma \vDash E : T_m$ *and* $\Theta; \Delta \vdash T_m \leq T$.

Proof sketch: by induction on the typing rules. □

The algorithmic typing rules terminate because the sum of the sizes of the expressions in the typing judgments is decreasing. Also the side conditions are decidable.

The proof of Lemmas 16 and 17 use the following standard lemmas:

$$
\begin{array}{lcll}
\tau \vee_\Delta \tau & = & \tau & \\
\{l_i : s_i\}_{i=1}^n \vee_\Delta \{k_j : t_j\}_{j=1}^m & = & \{l_i : \tau_i\}_{i=1}^p & \begin{cases} p = max\{i \mid l_j = k_j \ \forall 1 \leq j \leq i\} \\ \tau_i = s_i \vee_\Delta t_i \ \forall 1 \leq i \leq p \end{cases} \\
(\exists\alpha \ll u_1.\ \tau_1) \vee_\Delta (\exists\alpha \ll u_2.\ \tau_2) & = & \exists\alpha \ll u.\ \tau & u = u_1 \curlyvee_\Delta u_2 \text{ and } t = \tau_1 \vee_{\Delta,\alpha} \tau_2 \\
(\forall\alpha \ll u_1.\ \tau_1) \vee_\Delta (\forall\alpha \ll u_2.\ \tau_2) & = & \forall\alpha \ll u.\ \tau & u = u_1 \curlywedge_\Delta u_2 \text{ and } t = \tau_1 \vee_{\Delta,\alpha} \tau_2 \\
((s_1,\ldots,s_n) \to s) \vee_\Delta ((t_1,\ldots,t_n) \to t) & = & (t'_1,\ldots,t'_n) \to t' & t'_i = s_i \wedge_\Delta t_i \ \forall 1 \leq i \leq n \text{ and } t' = s \vee_\Delta t \\
\tau \wedge_\Delta \tau & = & \tau & \\
\{l_i : s_i\}_{i=1}^n \wedge_\Delta \{l_j : t_j\}_{j=1}^m & = & \{l_i : \tau_i\}_{i=1}^p & \begin{cases} p = max(m,n), \tau_i = s_i \wedge_\Delta t_i \ \forall 1 \leq i \leq min(m,n) \\ if\ m \leq n, then\ \tau_i = s_i \ \forall m+1 \leq i \leq n \\ otherwise, \tau_i = t_i \ \forall n+1 \leq i \leq m \end{cases} \\
(\exists\alpha \ll u_1.\ \tau_1) \wedge_\Delta (\exists\alpha \ll u_2.\ \tau_2) & = & \exists\alpha \ll u.\ \tau & u = u_1 \curlywedge_\Delta u_2 \text{ and } t = \tau_1 \wedge_{\Delta,\alpha} \tau_2 \\
(\forall\alpha \ll u_1.\ \tau_1) \wedge_\Delta (\forall\alpha \ll u_2.\ \tau_2) & = & \forall\alpha \ll u.\ \tau & u = u_1 \curlyvee_\Delta u_2 \text{ and } t = \tau_1 \wedge_{\Delta,\alpha} \tau_2 \\
((s_1,\ldots,s_n) \to s) \wedge_\Delta ((t_1,\ldots,t_n) \to t) & = & (t'_1,\ldots,t'_n) \to t' & t'_i = s_i \vee_\Delta t_i \ \forall 1 \leq i \leq n \text{ and } t' = s \wedge_\Delta t
\end{array}
$$

Figure 6: Definitions of subtyping joins/meets

LEMMA 18. *Permutation of kind environment entries does not affect kinding, subclassing or subtyping. If $\Delta = \Delta_1, \Delta_2, \alpha \ll \tau, \Delta_3$ and $\Theta; \Delta_1 \vdash \tau : \Omega_C$, $\Delta' = \Delta_1, \alpha \ll \tau, \Delta_2, \Delta_3$,*

- *if $\Theta; \Delta \vdash \tau : \kappa$, then $\Theta; \Delta' \vdash \tau : \kappa$;*
- *if $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, then $\Theta; \Delta' \vdash \tau_1 \ll \tau_2$;*
- *if $\Theta; \Delta \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta' \vdash \tau_1 \leq \tau_2$.*

Proof: by induction on kinding, subclassing and subtyping rules respectively. □

LEMMA 19. *Weakening of kind environment: if $domain(\Delta) \cap domain(\Delta') = \emptyset$, then*

1. *If $\Theta; \Delta \vdash T : \kappa$, then $\Theta; \Delta, \Delta' \vdash T : \kappa$*
2. *If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, then $\Theta; \Delta, \Delta' \vdash \tau_1 \ll \tau_2$*
3. *If $\Theta; \Delta \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta, \Delta' \vdash \tau_1 \leq \tau_2$.*

Proof: by induction on kinding, subclassing, subtyping rules respectively. Quantified type cases use Lemma 18. □

LEMMA 20. *Type substitution preserves kinding, subclassing and subtyping. Suppose $\Delta = \Delta_1, \eta \ll \tau, \Delta_2$ and $\Theta; \Delta_1 \vdash s \ll \tau$ (or $\Delta = \Delta_1, \eta \gg \tau, \Delta_2$ and $\Theta; \Delta_1 \vdash \tau \ll s$), and $\delta = s/\eta$ and $\Delta' = \Delta_1, \Delta_2[\delta]$.*

- *If $\Theta; \Delta \vdash \tau : \kappa$, then $\Theta; \Delta' \vdash \tau[\delta] : \kappa$.*
- *If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, then $\Theta; \Delta' \vdash \tau_1[\delta] \ll \tau_2[\delta]$.*
- *If $\Theta; \Delta \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta' \vdash \tau_1[\delta] \leq \tau_2[\delta]$.*

Proof: by induction on kinding, subclassing and subtyping rules respectively. □

LEMMA 21. *Inversion of subtyping*

1. *If $\Theta; \Delta \vdash S \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, then $S = \{l_1 : \tau'_1, \ldots, l_n : \tau'_n, \ldots\}$ and $\Theta; \Delta \vdash \tau'_i \leq \tau_i \ \forall 1 \leq i \leq n$.*
2. *If $\Theta; \Delta \vdash S \leq \forall\alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$, then $S = \forall\alpha_1 \ll u'_1, \ldots, \alpha_m \ll u'_m.(\tau'_1, \ldots, \tau'_n) \to \tau'$. If $\exists t_1, \ldots, t_m$ such that $\Theta; \Delta \vdash t_i \ll u_i[\sigma] \ \forall 1 \leq i \leq m$ where $\sigma = t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m$, then $\Theta; \Delta \vdash t_i \ll u'_i[\sigma] \ \forall 1 \leq i \leq m$, and $\Theta; \Delta \vdash \tau_i[\sigma] \leq \tau'_i[\sigma] \ \forall 1 \leq i \leq n$, and $\Theta; \Delta \vdash \tau'[\sigma] \leq \tau[\sigma]$.*
3. *If $\Theta; \Delta \vdash s \leq \exists\alpha \ll \tau_1.\ \tau_2$, then $s = \exists\alpha \ll \tau'_1.\ \tau'_2$, $\Theta; \Delta \vdash \tau'_1 \ll \tau_1$ and $\Theta; \Delta, \alpha \vdash \tau'_2 \leq \tau_2$.*
4. *If $\Theta; \Delta \vdash s \leq C$, then $s = C$.*
5. *If $\Theta; \Delta \vdash s \leq \alpha$, then $s = \alpha$.*
6. *If $\Theta; \Delta \vdash s \leq Tag(\tau)$, then $s = Tag(\tau)$.*

proof: by inspection of algorithmic subtyping rules. Case *2* is proved by $\Theta; \Delta, \alpha_1, \ldots, \alpha_{i-1} \vdash u_i \ll u'_i \ \forall 1 \leq i \leq m$ and $\Theta; \Delta' \vdash \tau_i \leq \tau'_i \ \forall 1 \leq i \leq n$ and $\Theta; \Delta' \vdash \tau' \leq \tau$ ($\Delta' = \Delta, \alpha_1, \ldots, \alpha_m$). By Lemma 20, $\Theta; \Delta \vdash u_i[\sigma] \ll u'_i[\sigma] \ \forall 1 \leq i \leq m$ and $\Theta; \Delta \vdash \tau_i[\sigma] \leq \tau'_i[\sigma] \ \forall 1 \leq i \leq n$ and $\Theta; \Delta \vdash \tau'[\sigma] \leq \tau[\sigma]$. By the transitivity of subclassing, $\Theta; \Delta \vdash t_i \ll u'_i[\sigma] \ \forall 1 \leq i \leq m$. □

A "narrowing" lemma is necessary to prove the completeness of the typing rule for the open expression (Appendix C, case **open**). It says that, in environments that have stronger constraints on type variables and variables, expressions have more specialized minimal types.

LEMMA 22. *Narrowing of Environments. If $\Theta; \Delta; \Gamma \vDash E : T$, $\Theta \vdash \Delta' \ll \Delta$ and $\Theta; \Delta' \vdash \Gamma' \leq \Gamma$, then $\Theta; \Delta'; \Gamma' \vDash E : T'$ and $\Theta; \Delta' \vdash T' \leq T$.*

Proof sketch: by induction on algorithmic typing rules. Environment narrowing is defined as:

- If domain(Γ') = domain(Γ), and $\Theta; \Delta \vdash \Gamma'(x) \leq \Gamma(x)$ $\forall x \in$ domain(Γ), then we define $\Theta; \Delta \vdash \Gamma' \leq \Gamma$.
- If domain(Δ') = domain(Δ), and $\forall\alpha \ll u \in \Delta$, $\Delta' = \Delta'_1, \alpha \ll u', \Delta'_2$, $\Delta = \Delta_1, \alpha \ll u, \Delta_2$, and $\Theta; \Delta'_1 \vdash u' \ll u$, then we define $\Theta \vdash \Delta' \ll \Delta$.

It has the following properties:

LEMMA 23.

1. *If $\Theta; \Delta \vdash \tau : \kappa$ and $\Theta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau : \kappa$.*
2. *If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$ and $\Theta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau_1 \ll \tau_2$.*
3. *If $\Theta; \Delta \vdash \tau_1 \leq \tau_2$ and $\Theta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau_1 \leq \tau_2$.*
4. *If $\alpha \in domain(\Delta)$ and $\Theta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \alpha \uparrow_{\Delta'} \ll \alpha \uparrow_\Delta$.*

Proof: prove (1), (2) and (3) by induction on kinding, subclassing and subtyping rules respectively.

(4): by the first part of Lemma 10 $\Theta; \Delta \vdash \alpha \ll \alpha \uparrow_\Delta$. By (2) $\Theta; \Delta' \vdash \alpha \ll \alpha \uparrow_\Delta$. By the second part of Lemma 10, $\Theta; \Delta' \vdash \alpha \uparrow_{\Delta'} \ll \alpha \uparrow_\Delta$. □

LEMMA 24. • *If* $\Theta; \Delta \vdash \tau : \kappa$*, then* $free(\tau) \in \Delta$*.*

- *If* $\Theta; \Delta \vdash \tau_1 \ll \tau_2$*, then* $\Theta; \Delta \vdash \tau_1 : \Omega$ *and* $\Theta; \Delta \vdash \tau_2 : \Omega$*.*

- *If* $\Theta; \Delta \vdash \tau_1 \leq \tau_2$*, then* $\Theta; \Delta \vdash \tau_1 : \Omega$ *and* $\Theta; \Delta \vdash \tau_2 : \Omega$*.*

- *If* $\Theta; \Delta; \Gamma \vdash e : \tau$*, then* $\Theta; \Delta \vdash \tau : \Omega$*.*

Proof: by induction on kinding, subclassing, subtyping and typing rules respectively. □

With decidability of the subtyping and the minimal type property, we can prove the main theorem:

THEOREM 25. *Type checking of* LIL_C *is decidable.*

Proof: To decide whether $\Theta; \Delta; \Gamma \vdash e : \tau$ holds, we can first get the minimal type τ_m of e such that $\Theta; \Delta; \Gamma \vDash e : \tau_m$, then test whether $\Theta; \Delta \vdash \tau_m \leq \tau$. Because both the algorithmic typing rules and the subtyping rules are decidable, type checking is decidable.

4. INTERFACES

Interfaces change the structure of the subclassing hierarchy (Figure 7). Each class can implement zero or more interfaces. For example, in Figure 7, both classes C_1 and C_2 implement both interfaces C and I. There is no join of C_1 and C_2, nor meet of C and I.

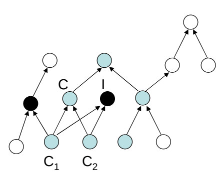

Figure 7: Subclassing Hierarchy with Interfaces (Shaded nodes are class names, black nodes are interfaces, and unshaded are type variables.)

The proof techniques in this paper can be extended to handle interfaces, with a restricted form of union types and intersection types defined only on class names and interface names, not on type variables. Therefore, we can avoid the complexities with general union and intersection types as described in Section 3.

LIL_C needs new subclassing rules related to union and intersection types. The subclassing properties (Lemma 2) still hold. Subclassing is decidable even with the new types, because the set of class names, interfaces, union/intersection types and type variables is finite. Subtyping and expression typing rules remain the same.

The only change in the decidability proof are Lemma 12 and Definition 13. Lemma 12 no longer holds and is no longer necessary. The base case in Defintion 13—the subclassing joins/meets of class names and interface names—will change to use union and intersection types. All other definitions/lemmas are the same.

5. RELATED WORK

In the original bounded quantification paper [2], Cardelli *et al.* described two systems with bounded quantification: Kernel $F_{<:}$ and Full $F_{<:}$. The key differences are in the subtyping between quantified types. Kernel $F_{<:}$ has rule **kernel_∀** and Full $F_{<:}$ has rule **full_∀**:

$$\frac{\Delta, \alpha \leq \tau \vdash \tau_2 \leq \tau_2'}{\Delta \vdash (\forall \alpha \leq \tau.\tau_2) \leq (\forall \alpha \leq \tau.\tau_2')} \text{ kernel\_}\forall$$

$$\frac{\Delta \vdash \tau_1' \leq \tau_1 \quad \Delta, \alpha \leq \tau_1' \vdash \tau_2 \leq \tau_2'}{\Delta \vdash (\forall \alpha \leq \tau_1.\tau_2) \leq (\forall \alpha \leq \tau_1'.\tau_2')} \text{ full\_}\forall$$

Kernel $F_{<:}$ has decidable type checking, but is not expressive enough for features such as inheritance. Type-checking of Full $F_{<:}$ has been proved undecidable [8]. The bounds of type variables can be quantified types, which leads to undecidable subtyping, thus undecidable type checking.

Castagna and Pierce proposed the following subtyping rule [3]:

$$\frac{\Delta \vdash \tau_1' \leq \tau_1 \quad \Delta, \alpha \leq Top \vdash \tau_2 \leq \tau_2'}{\Delta \vdash (\forall \alpha \leq \tau_1.\tau_2) \leq (\forall \alpha \leq \tau_1'.\tau_2')}$$

The new system has decidable subtyping, with the bound of the new type variable forgotten (relaxed to Top, a super type of all types) in deciding subtyping of the body types. But the system does not have the minimal typing property [3]. Expression $\Lambda \alpha \leq \tau.\lambda x : \alpha.x$ can be typed with both $\forall \alpha \leq \tau.\alpha \rightarrow \tau$ and $\forall \alpha \leq \tau.\alpha \rightarrow \alpha$, but it has no minimal type. The decidability of this system remains an open problem. In LIL_C, subclassing between class names does not imply subtyping. Expression $\Lambda \alpha \ll \tau.\lambda x : \alpha.x$ will not have type $\forall \alpha \ll \tau.\alpha \rightarrow \tau$.

Ghelli and Pierce explained an algorithm for computing the minimal type of Kernel $F_{<:}$ extended with existential types [5]. They defined a type lifting operation for open expressions. Because they use **kernel_∀**, their type lifting operation is simpler than the one in LIL_C. Lifting α out of $\exists \beta \leq \alpha.\tau$ results in type *Top*.

Bruce proposed a safe and decidable language TOOPLE for object-oriented languages [1]. Unlike LIL_C, TOOPLE is designed as a source-level language. The denotational semantics of TOOPLE is based on F-bounded quantification. The language separates subclassing and inheritance, but does not preserve class names. Inheritance is expressed as a special "matching" relation.

League *et al.* used row polymorphism [9] for object and class encodings, instead of bounded quantification [7]. Inheritance is expressed by existentially bounded row variables. Their language has decidable type checking, with explicit coercions (no explicit subtyping).

As an application of guarded recursive datatypes, Xi *et al.* proposed an object encoding [11]. The encoding interprets objects as functions that dispatch messages. The language has decidable type-checking, only with extensive type annotations and with hints provided by the programmers.

Katiyar and Sankar proved that the subtyping of $F_{<:}$ is decidable if the bounds in quantified types do not contain *Top* [6]. Vorobyov proved the decidability of subtyping in some extensions of $F_{<:}$, based on interpretations in monadic second-order theories [10].

6. CONCLUSION

This paper proves the decidability of type-checking LIL_C, a typed intermediate language with subclassing-bounded quantification. The proof follows the standard two-step approach:

the decidability of subtyping and the minimal type property. Subtyping in LIL_C is decidable because of the separation of subclassing and subtyping. With expressions that introduce new local type variables (with lower bounds), the minimal type property requires a special type lifting operation. The proof techniques can be extended to handle interfaces with minor changes.

7. REFERENCES

[1] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *the 8th annual conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 29–46. ACM Press, 1993.

[2] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[3] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *21st ACM Symposium on Principles of Programming Languages*, pages 151–162. ACM Press, 1994. Corrigendum: Decidable Bounded Quantification, http://www.cis.upenn.edu/∼bcpierce/papers/fsubnew-corrigendum.ps.

[4] Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. Technical report, Microsoft Corporation, 2004. http://www.research.microsoft.com/∼juanchen/tr.pdf. A shorter paper to appear in the ACM Symposium on Principles of Programming Languages 2005.

[5] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1–2):75–96, 1998.

[6] Dinesh Katiyar and Sriram Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[7] Christopher League, Zhong Shao, and Valery Trifonov. Type-preserving compilation of Featherweight Java. *ACM Trans. on Programming Languages and Systems*, 24(2), March 2002.

[8] Benjamin C. Pierce. Bounded quantification is undecidable. In *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 427–459. The MIT Press, MA, 1994.

[9] Didier Rémy. Programming objects with ML-ART, an extension to ML with abstract and record types. In *International Conference on Theoretical Aspects of Computer Software*, pages 321–346. Springer-Verlag, 1994.

[10] Sergei G. Vorobyov. Structural decidable extensions of bounded quantification. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 164–175, California, 1995.

[11] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

APPENDIX

A. PROOF OF LEMMA 16

By induction on algorithmic expression typing rules.

Case a_open $E = (\alpha, x) = \text{open}(e_1)$ in e_2, $T = (\tau_m) \Uparrow^{\alpha}_{\Delta'}$ $(\Delta' = \Delta, \alpha \ll u)$, with derivations $\Theta; \Delta; \Gamma \vDash e_1 : \exists \beta \ll u.\, \tau_1$, $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vDash e_2 : \tau_m$ $(\alpha \notin \text{domain}(\Delta))$.

By induction hypothesis, $\Theta; \Delta; \Gamma \vdash e_1 : \exists \beta \ll u.\, \tau_1$ and $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vdash e_2 : \tau_m$. By Lemma 11, $\Theta; \Delta, \alpha \ll u \vdash \tau_m \leq T$. By subsumption **sub** $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vdash e_2 : T$. By the definition of type lifting, $\alpha \notin free(T)$. Therefore, $\Theta; \Delta; \Gamma \vdash E : T$ by **open**.

Case a_c2r_tv $E = \text{c2r}(e)$, $T = \text{Approx}R(\alpha, \alpha \uparrow_\Delta)$ with derivation $\Theta; \Delta; \Gamma \vDash e : \alpha$.

By induction hypotheis, $\Theta; \Delta; \Gamma \vdash e : \alpha$. By Lemma 10, $\Theta; \Delta \vdash \alpha \ll \alpha \uparrow_\Delta$ and $\alpha \uparrow_\Delta$ is a concrete class name. By **c2r_tv** $\Theta; \Delta; \Gamma \vdash E : T$.

Case a_ifParent $E = \text{ifParent}(e)$ then bind (α, x) in e_1 else e_2, $T = (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s} \vee_\Delta \tau_2$ with derivations $\Theta; \Delta; \Gamma \vDash e : Tag(s)$, $\Theta; \Delta, \alpha \gg s; \Gamma, x : Tag(\alpha) \vDash e_1 : \tau_1$ and $\Theta; \Delta; \Gamma \vDash e_2 : \tau_2$.

By induction hypothesis, $\Theta; \Delta; \Gamma \vdash e : Tag(s)$, $\Theta; \Delta, \alpha \gg s; \Gamma, x : Tag(\alpha) \vdash e_1 : \tau_1$ and $\Theta; \Delta; \Gamma \vdash e_2 : \tau_2$. By properties of type lifting Lemma 11, $\Theta; \Delta, \alpha \gg s \vdash \tau_1 \leq (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s}$. By properties of subtyping joins/meets 15, $\Theta; \Delta \vdash (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s} \leq T$. By weakening of environments (Lemma 19), $\Theta; \Delta, \alpha \gg s \vdash (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s} \leq T$. By transitivity of subtyping and **sub**, $\Theta; \Delta, \alpha \gg s; \Gamma, x : Tag(\alpha) \vdash e_1 : T$. By Lemma 15, $\Theta; \Delta \vdash \tau_2 \leq T$. By **sub**, $\Theta; \Delta; \Gamma \vdash e_2 : T$. By **ifParent** $\Theta; \Delta; \Gamma \vdash E : T$. □

B. PROOF OF LEMMA 22

By induction on algorithmic typing rules.

Case a_var: $E = x$, $T = \Gamma(x)$.

Let $T' = \Gamma'(x)$. By **a_var**, $\Theta; \Delta'; \Gamma' \vDash E : T'$. And by the definition of $\Theta; \Delta' \vdash \Gamma' \leq \Gamma$, we have $\Theta; \Delta' \vdash T' \leq T$.

Case a_obj: $E = C(e)$, $T = C$ with derivation $\Theta; \Delta; \Gamma \vDash e : \tau_m$ and $\Theta; \Delta \vdash \tau_m \leq R(C)$.

By induction hypothesis, $\Theta; \Delta'; \Gamma' \vDash e : \tau'_m$ and $\Theta; \Delta' \vdash \tau'_m \leq \tau_m$. By Lemma 23, $\Theta; \Delta' \vdash \tau_m \leq R(C)$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau'_m \leq R(C)$. Let $T' = T$. By reflexivity of subtyping $\Theta; \Delta' \vdash T' \leq T$. By **a_obj** $\Theta; \Delta'; \Gamma' \vDash E : T'$.

Case a_c2r_tv: $E = \text{c2r}(e)$, $T = \text{Approx}R(\alpha, \alpha \uparrow_\Delta)$ with derivations $\Theta; \Delta; \Gamma \vDash e : \alpha$.

By induction hypothesis $\Theta; \Delta'; \Gamma' \vDash e : \tau'$ and $\Theta; \Delta' \vdash \tau' \leq \alpha$. By Lemma 21, $\tau' = \alpha$. Let $T' = \text{Approx}R(\alpha, \alpha \uparrow_{\Delta'})$. By Lemma 23 $\Theta; \Delta' \vdash \alpha \uparrow_{\Delta'} \ll \alpha \uparrow_\Delta$. By properties of approximation Lemma 1, $\Theta; \Delta' \vdash T' \leq T$. By **a_c2r_tv** $\Theta; \Delta'; \Gamma' \vDash E : T'$.

Case a_call: $E = e[t_1, \ldots, t_m](e_1, \ldots, e_n)$, $T = \tau[\sigma]$ with $\Theta; \Delta; \Gamma \vDash e : \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m(\tau_1, \ldots, \tau_n) \to \tau$, and $\Theta; \Delta; \Gamma \vDash e_i : \tau_{mi}$, $\Theta; \Delta \vdash \tau_{mi} \leq \tau_i[\sigma]$ $\forall 1 \leq i \leq n$ where $\sigma = t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m$ and $\Theta; \Delta \vdash t_i \ll u_i[\sigma]$ $\forall 1 \leq i \leq m$.

By Lemma 23, $\Theta; \Delta' \vdash t_i \ll u_i[\sigma]$ $\forall 1 \leq i \leq m$. By induction hypothesis, $\Theta; \Delta'; \Gamma' \vDash e : S$ and $\Theta; \Delta' \vdash S \leq \forall \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$. By subtyping inversion Lemma 21 $S = \forall \alpha_1 \ll u'_1, \ldots, \alpha_m \ll u'_m.(\tau'_1, \ldots, \tau'_n) \to \tau'$, and $\Theta; \Delta' \vdash t_i \ll u'_i[\sigma]$ $\forall 1 \leq i \leq m$, and $\Theta; \Delta' \vdash \tau_i[\sigma] \leq \tau'_i[\sigma]$ $\forall 1 \leq i \leq n$, and $\Theta; \Delta' \vdash \tau'[\sigma] \leq \tau[\sigma]$. By induction hypothesis, $\forall 1 \leq i \leq n$, $\Theta; \Delta'; \Gamma' \vDash e_i : \tau'_{mi}$ and $\Theta; \Delta' \vdash \tau'_{mi} \leq \tau_{mi}$. By lemma 23 $\Theta; \Delta' \vdash \tau_{mi} \leq \tau_i[\sigma]$ $\forall 1 \leq i \leq n$. By the transitivity of subtyping, $\Theta; \Delta' \vdash \tau'_{mi} \leq$

$\tau_i'[\sigma]$. Let $T' = \tau'[\sigma]$, then $\Theta; \Delta' \vdash T' \leq T$. By **a_call** $\Theta; \Delta'; \Gamma' \vDash E : T'$.

Case a_open: $E = (\alpha, x) = \text{open}(e_1)$ in e_2, $T = (\tau_m) \Uparrow^{\alpha}_{\Delta'}$ $(\Delta' = \Delta, \alpha \ll u)$, with derivations $\Theta; \Delta; \Gamma \vDash e_1 : \exists\beta \ll u.\ \tau_1$ and $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vDash e_2 : \tau_m$ $(\alpha \notin \text{domain}(\Delta))$.

By induction hypothesis, $\Theta; \Delta'; \Gamma' \vDash e_1 : \tau_{m1}$ and $\Theta; \Delta' \vdash \tau_{m1} \leq \exists\beta \ll u.\ \tau_1$. By subtyping inversion Lemma 21, (1) $\tau_{m1} = \exists\beta \ll u'.\ \tau_1'$, (2) $\Theta; \Delta' \vdash u' \ll u$ and (3) $\Theta; \Delta', \beta \vdash \tau_1' \leq \tau_1$. By alpha-equivalence and $\alpha \notin \text{domain}(\Delta')$, we have $\Theta; \Delta', \alpha \vdash \tau_1'[\alpha/\beta] \leq \tau_1[\alpha/\beta]$. By Lemma 23 and $\Theta \vdash (\Delta', \alpha \ll u') \ll (\Delta', \alpha)$, $\Theta; \Delta', \alpha \ll u' \vdash \tau_1'[\alpha/\beta] \leq \tau_1[\alpha/\beta]$. By definition, $\Theta \vdash (\Delta', \alpha \ll u') \ll (\Delta, \alpha \ll u)$, and $\Theta; \Delta', \alpha \ll u' \vdash (\Gamma', x : \tau_1'[\alpha/\beta]) \leq (\Gamma, x : \tau_1[\alpha/\beta])$. Then by induction hypothesis, $\Theta; \Delta', \alpha \ll u'; \Gamma', x : \tau_1'[\alpha/\beta] \vDash e_2 : \tau_m'$ and $\Theta; \Delta', \alpha \ll u' \vdash \tau_m' \leq \tau_m$. Let $T' = (\tau_m') \Uparrow^{\alpha}_{\Delta', \alpha \ll u'}$. By **a_open**, $\Theta; \Delta'; \Gamma' \vDash E : T'$. By properties of type lifting Lemma 11, $\Theta; \Delta, \alpha \ll u \vdash \tau_m \leq T$. By Lemma 23 $\Theta; \Delta', \alpha \ll u' \vdash \tau_m \leq T$. By the transitivity of subtyping and $\Theta; \Delta', \alpha \ll u' \vdash \tau_m' \leq \tau_m$, we have $\Theta; \Delta', \alpha \ll u' \vdash \tau_m' \leq T$. By Lemma 11, $\Theta; \Delta' \vdash T' \leq T$ because $\alpha \notin free(T)$.

Case a_ifParent $E = \text{ifParent}(e)$ then bind (α, x) in e_1 else e_2, $T = (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s} \vee_{\Delta} \tau_2$ with derivations $\Theta; \Delta; \Gamma \vDash e : Tag(s)$, $\Theta; \Delta, \alpha \gg s; \Gamma, x : Tag(\alpha) \vDash e_1 : \tau_1$ and $\Theta; \Delta; \Gamma \vDash e_2 : \tau_2$.

By induction hypothesis $\Theta; \Delta'; \Gamma' \vDash e : \tau'$, $\Theta; \Delta' \vdash \tau' \leq Tag(s)$, $\Theta; \Delta', \alpha \gg s; \Gamma', x : Tag(\alpha) \vDash e_1 : \tau_1'$, $\Theta; \Delta', \alpha \gg s \vdash \tau_1' \leq \tau_1$, $\Theta; \Delta'; \Gamma' \vDash e_2 : \tau_2'$ and $\Theta; \Delta' \vdash \tau_2' \leq \tau_2$. By inversion of subtyping Lemma 21 $\tau' = Tag(s)$. Let $T_1 = (\tau_1) \Uparrow^{\alpha}_{\Delta, \alpha \gg s}$, $T_1' = (\tau_1') \Uparrow^{\alpha}_{\Delta', \alpha \gg s}$ and $T' = T_1' \vee_{\Delta'} \tau_2'$. By **a_ifParent** $\Theta; \Delta'; \Gamma' \vDash E : T'$. From properties of type lifting Lemma 11, $\Theta; \Delta, \alpha \gg s \vdash \tau_1 \leq T_1$. By properties of subtyping joins/meets 15 $\Theta; \Delta \vdash T_1 \leq T$ and $\Theta; \Delta \vdash \tau_2 \leq T$. By weakening of kind environment Lemma 19 $\Theta; \Delta, \alpha \gg s \vdash T_1 \leq T$. By transitivity of subtyping $\Theta; \Delta, \alpha \gg s \vdash \tau_1 \leq T$. By Lemma 23 $\Theta; \Delta', \alpha \gg s \vdash \tau_1 \leq T$ and $\Theta; \Delta' \vdash \tau_2 \leq T$. By transitivity of subtyping $\Theta; \Delta', \alpha \gg s \vdash \tau_1' \leq T$ and $\Theta; \Delta' \vdash \tau_2' \leq T$. Because $\alpha \notin free(T)$, $\Theta; \Delta' \vdash T_1' \leq T$. By Lemma 15 $\Theta; \Delta' \vdash T' \leq T$ because T is a common upper bounds of both T_1' and τ_2'. □

C. PROOF OF LEMMA 17

By induction on the typing rules.

Case call $E = e[t_1, \ldots, t_m](e_1, \ldots, e_n)$, $T = \tau[\sigma]$ with $\Theta; \Delta; \Gamma \vdash e : \forall\alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$, $\Theta; \Delta; \Gamma \vdash e_i : \tau_i[\sigma]$ $\forall 1 \leq i \leq n$, $\sigma = t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m$ and $\Theta; \Delta \vdash t_i \ll u_i[\sigma]$ $\forall 1 \leq i \leq m$.

By induction hypothesis, $\Theta; \Delta; \Gamma \vDash e : S$, $\Theta; \Delta \vdash S \leq \forall\alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m.(\tau_1, \ldots, \tau_n) \to \tau$, and $\Theta; \Delta; \Gamma \vDash e_i : \tau_{mi}$, $\Theta; \Delta \vdash \tau_{mi} \leq \tau_i[\sigma]$ $\forall 1 \leq i \leq n$. By subtyping inversion Lemma 21, $S = \forall\alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m'.(\tau_1', \ldots, \tau_n') \to \tau'$, $\Theta; \Delta \vdash t_i \ll u_i'[\sigma]$ $\forall 1 \leq i \leq m$, $\Theta; \Delta \vdash \tau_i[\sigma] \leq \tau_i'[\sigma]$ $\forall 1 \leq i \leq n$, and $\Theta; \Delta \vdash \tau'[\sigma] \leq \tau[\sigma]$. By the transitivity of subtyping $\Theta; \Delta \vdash \tau_{mi} \leq \tau_i'[\sigma]$. Let $T_m = \tau'[\sigma]$, then $\Theta; \Delta \vdash T_m \leq T$. By **a_call**, $\Theta; \Delta; \Gamma \vDash E : T_m$.

Case open $E = (\alpha, x) = \text{open}(e_1)$ in e_2, $T = \tau_2$ with derivations $\Theta; \Delta; \Gamma \vdash e_1 : \exists\beta \ll u.\ \tau_1$ and $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vdash e_2 : \tau_2$, where $\alpha \notin \text{domain}(\Delta)$ and $\alpha \notin free(\tau_2)$.

By induction hypothesis, $\exists\tau_{m1}$ and τ_{m2} such that $\Theta; \Delta; \Gamma \vDash e_1 : \tau_{m1}$ and $\Theta; \Delta \vdash \tau_{m1} \leq \exists\beta \ll u.\ \tau_1$, $\Theta; \Delta, \alpha \ll u; \Gamma, x : \tau_1[\alpha/\beta] \vDash e_2 : \tau_{m2}$ and $\Theta; \Delta, \alpha \ll u \vdash \tau_{m2} \leq \tau_2$.

By inversion of subtyping Lemma 21, $\tau_{m1} = \exists\beta \ll u'.\ \tau_1'$, and $\Theta; \Delta \vdash u' \ll u$ and $\Theta; \Delta, \beta \vdash \tau_1' \leq \tau_1$. By α-conversion, $\Theta; \Delta, \alpha \vdash \tau_1'[\alpha/\beta] \leq \tau_1[\alpha/\beta]$ because α is a fresh type variable. By Lemma 23 and $\Theta \vdash (\Delta, \alpha \ll u') \ll (\Delta, \alpha \ll \text{Top}_C)$, $\Theta; \Delta, \alpha \ll u' \vdash \tau_1'[\alpha/\beta] \leq \tau_1[\alpha/\beta]$. By definition, $\Theta \vdash (\Delta, \alpha \ll u') \ll (\Delta, \alpha \ll u)$ and $\Theta; \Delta, \alpha \ll u' \vdash (\Gamma, x : \tau_1'[\alpha/\beta]) \leq (\Gamma, x : \tau_1[\alpha/\beta])$. By narrowing of environments Lemma 22, $\exists\tau_{m2}'$ such that $\Theta; \Delta, \alpha \ll u'; \Gamma, x : \tau_1'[\alpha/\beta] \vDash e_2 : \tau_{m2}'$ and $\Theta; \Delta, \alpha \ll u' \vdash \tau_{m2}' \leq \tau_{m2}$. By Lemma 23 and $\Theta \vdash (\Delta, \alpha \ll u') \ll (\Delta, \alpha \ll u)$ and $\Theta; \Delta, \alpha \ll u \vdash \tau_{m2} \leq \tau_2$, $\Theta; \Delta, \alpha \ll u' \vdash \tau_{m2} \leq T$. By transitivity of subtyping $\Theta; \Delta, \alpha \ll u' \vdash \tau_{m2}' \leq T$. Let $T_m = (\tau_{m2}') \Uparrow^{\alpha}_{\Delta, \alpha \ll u'}$. By properties of type lifting Lemma 11 T_m exists and $\Theta; \Delta \vdash T_m \leq T$ because $\alpha \notin free(T)$. By **a_open**, $\Theta; \Delta; \Gamma \vDash E : T_m$.

Case c2r_tv $E = \text{c2r}(e)$, $T = \text{Approx}R(\alpha, C)$ with derivation $\Theta; \Delta; \Gamma \vdash e : \alpha$ and $\Theta; \Delta \vdash \alpha \ll C$.

By induction hypothesis $\exists\tau_m$ such that $\Theta; \Delta; \Gamma \vDash e : \tau_m$ and $\Theta; \Delta \vdash \tau_m \leq \alpha$. By Lemma 21, $\tau_m = \alpha$. By Lemma 10, $\Theta; \Delta \vdash \alpha \uparrow_{\Delta} \ll C$. Let $T_m = \text{Approx}R(\alpha, \alpha \uparrow_{\Delta})$. By Lemma 1, $\Theta; \Delta \vdash T_m \leq T$. By **a_c2r_tv** $\Theta; \Delta; \Gamma \vDash E : T_m$.

Case ifParent $E = \text{ifParent}(e)$ then bind (α, x) in e_1 else e_2, $T = \tau$ with derivations $\Theta; \Delta; \Gamma \vdash e : Tag(s)$, $\Theta; \Delta, \alpha \gg s; \Gamma, x : Tag(\alpha) \vdash e_1 : \tau$ and $\Theta; \Delta; \Gamma \vdash e_2 : \tau$.

Let $\Delta' = \Delta, \alpha \gg s$. By induction hypothesis, $\exists\tau_m$, τ_{m1} and τ_{m2} such that $\Theta; \Delta; \Gamma \vDash e : \tau_m$, $\Theta; \Delta'; \Gamma, x : Tag(\alpha) \vDash e_1 : \tau_{m1}$ and $\Theta; \Delta; \Gamma \vDash e_2 : \tau_{m2}$, and $\Theta; \Delta \vdash \tau_m \leq Tag(s)$, $\Theta; \Delta' \vdash \tau_{m1} \leq \tau$ and $\Theta; \Delta \vdash \tau_{m2} \leq \tau$. By Lemma 21, $\tau_m = Tag(s)$. By Lemma 11, $(\tau_{m1}) \Uparrow^{\alpha}_{\Delta'}$ exists, and $\Theta; \Delta \vdash (\tau_{m1}) \Uparrow^{\alpha}_{\Delta'} \leq \tau$ because by Lemma 24 $\alpha \notin free(\tau)$. Let $T_m = (\tau_{m1}) \Uparrow^{\alpha}_{\Delta'} \vee_{\Delta} \tau_{m2}$. By Lemma 15 T_m exists and $\Theta; \Delta \vdash T_m \leq T$, because T is a common upper bound of both $(\tau_{m1}) \Uparrow^{\alpha}_{\Delta'}$ and τ_{m2}. By **a_ifParent** $\Theta; \Delta; \Gamma \vDash E : T_m$. □