Some object-oriented languages, such as Java and C#, provide interfaces as a feature for supporting a restricted form of multiple inheritance. Most typed intermediate languages for compiling object-oriented languages do not support interfaces or multiple inheritance. This paper describes a typed intermediate language that supports interface implementation strategies based on interface tables (itables). The language can faithfully model itables, the standard itable-based interface method invocation, and interface cast. The language extends a typed intermediate language that supports classes and single inheritance, and it has a sound and decidable type system. We believe that the underlying ideas can be applied to address other interface implementation techniques.

1The work by Chiyan Chen was done during an internship at Microsoft Research
1 Introduction

Interfaces in object-oriented languages provide a convenient mechanism to express contracts that classes expose to the outer world. Languages that allow only single inheritance between classes, such as Java, use interfaces to support a restricted form of multiple inheritance. However, most low-level typed intermediate languages for compiling object-oriented languages do not support interfaces or multiple inheritance.

This paper explains LILCI (Low-level Intermediate Language with Classes and Interfaces), a typed intermediate language that supports both classes and interfaces. LILCI is based on a language LILC that supports only classes and single inheritance between classes [6]. Both languages aim to faithfully support standard techniques for object-oriented features, without extra runtime overhead. LILC is low level in the sense that it can express object layout including vtables, and that it breaks down primitives such as virtual method invocation, downward type cast, and array store checks. LILC differs from most class and object encodings in that it preserves class names and name-based subclassing.

This paper focuses on interface implementation techniques based on interface tables (itables). Each class stores in its itable information on interfaces the class implements. Each itable entry contains a unique identifier (called tag) and a method table for an interface. At interface method invocation time, the tag of each itable entry is compared with the tag of the target interface. If there is a match, the entry must correspond to the target interface, and the desired method can be fetched from the entry’s method table. Many compilers use this strategy [11].

The main contributions of this work include:

• providing a typed framework that faithfully models itables, itable-based interface cast and interface method invocation. To the best of our knowledge, it is the first language that is able to express these details.

• extending a typed intermediate language that supports only single inheritance between classes to support multiple inheritance via interfaces.

Multiple inheritance complicates object layout representations. As shown in Section 2, the representation of itables is the key difficulty in supporting itable-based strategies.

Multiple inheritance also complicates type checking. Because of the existence of conditional branch expressions, the proof of the minimal type property needs the least upper subtyping bounds of types, which requires the least upper subclassing bounds of types. There might not exist the least upper subclassing bounds for any two classes because of multiple inheritance. To simplify the type checking, LILCI adds type annotations on conditional branch expressions so that the least upper bounds are not needed.

Many other techniques can improve the performance of interface method invocation [1, 12, 9, 2, 16, 14, 10]. It remains an open problem how to address these techniques in a typed intermediate language. We discuss in Section 6 some extensions of LILCI that might be applied to express the techniques.

The rest of the paper is organized as follows. Section 2 gives an informal overview of how LILCI extends LILC to support interfaces. It focuses on how to represent itables with array types and existential types and how to model interface method invocation and interface cast with polymorphic functions. The next two sections explain the syntax and semantics of LILCI. Section 5 describes a source language and the translation from the source language to LILCI. Sections 6 and 7 discuss related and future work respectively. Section 8 concludes.

2 Overview

This section first summarizes the basic ideas of LILC (Section 2.1). We refer readers to the previous paper [6] for details of virtual method invocation, class cast, and arrays. Section 2.2 explains how LILCI extends LILC to support interfaces. Section 2.3 explains object layout. The rest of the section explains interface method invocation and interface cast.

2.1 Basic Ideas of LILC

Most typed intermediate languages compile classes and objects away. LILC differs from those languages in that it preserves class names and name-based subclassing.
LIL\textsubscript{C} has an “exact” notion of class types. A class name in LIL\textsubscript{C} represents only objects of exact that class, not including objects of subclasses. Each class \( C \) has a corresponding record type \( R(C) \) that describes the object layout of \( C \). Objects can be coerced to or from records with appropriate types.

LIL\textsubscript{C} has subclassing-bounded quantification: quantified types can specify subclassing bounds for type variables they introduce. The notation \( \exists a \ll B \) represents subclassing. Type \( \exists a \ll \tau \cdot \tau' \) introduces a type variable \( \alpha \) and \( \alpha \)'s upper subclassing bound \( \tau \). Those subclassing bounds can only be class names or type variables that will be instantiated with class names. The subclassing-bounded quantification makes the type system decidable and expressive.

A source class name \( C \) is translated to an LIL\textsubscript{C} type \( \exists a \ll C \cdot \alpha \), which represents objects of \( C \) and \( C \)'s subclasses. If \( C \ll B \), then \( \exists a \ll C \cdot \alpha \) is a subtype of \( \exists a \ll B \cdot \alpha \), which describes the inheritance subsumption: an object of class \( C \) or \( C \)'s subclasses can be used as an object of class \( B \) or \( B \)'s subclasses.

The layouts of objects whose actual runtime (“exact”) types are statically unknown can be approximated by a record type: \( \text{ApproxR}(\alpha, C) \) represents objects with exact type \( \alpha \), where \( \alpha \) is a subclass of \( C \). Record types can refer to any class names in the program. Therefore, no structural recursive types are necessary.

\( R \) and \text{ApproxR} are not type constructors, but macros used by the type checker. Therefore, the layout strategy is part of the type system. However, these two macros are parameters of the type system. The compiler has the freedom to choose layout strategies as long as certain structural subtyping requirements with \( R \) and \text{ApproxR} are satisfied.

The coercions between objects and records have no runtime overhead. To create an object, we create a record and coerce it to an object. To fetch a field or invoke a method, we coerce the object to a record and then fetch the field or the method pointer from the record.

Each class has an abstract tag to uniquely identify the class at run time. Type \( \text{Tag}(\tau) \) describes the tag of \( \tau \). “Tag” is an abstract type constructor. LIL\textsubscript{C} uses tags for dynamic type tests.

### 2.2 Extending LIL\textsubscript{C} to Support Interfaces

This section gives an informal explanation of how LIL\textsubscript{CI} extends LIL\textsubscript{C}. Formal details are in Sections 3 and 4.

The ideas of LIL\textsubscript{C} can be extended to interfaces naturally. LIL\textsubscript{CI} preserves interface names (ranged over \( I \) and \( J \)). Interface names are treated almost the same as class names: (1) interface names can appear in object types, array element types, and cast target types; (2) subclassing in LIL\textsubscript{CI} includes interfaces. \( C \ll I \) represents that class \( C \) implements interface \( I \). \( I \ll J \) represents that interface \( I \) is a sub-interface of interface \( J \); (3) the bounds in quantified types can be interface names. Type variables can be instantiated with interface names; (4) a source interface name \( I \) is translated to \( \exists a \ll I \cdot \alpha \), representing objects of classes that implement \( I \); (5) interfaces have tags as identifiers: \( \text{Tag}(I) \) represents the tag of interface \( I \).

The key structure of the itable-based interface implementation strategy is the itable, which collects the information for all interfaces a class implements. Each interface has a corresponding entry that contains the tag and a method table for the interface. For each method declared in the interface, the method table contains a function pointer that implements the method in the class. The order of function pointers in the method table is the same in all itable entries that correspond to the same interface. This allows the compiler to use the same offset for a method in the method table of an interface, no matter in which class.

The standard approach places no requirement on the ordering of itable entries in an itable, and searches the target interface at run time.

We use array types to represent itables. Each element has an existential type that abstracts the corresponding interface. The existentially quantified type variable has a lower bound \( C \), because each entry in the itable of class \( C \) represents an interface \( I \) such that \( C \ll I \). The body of the existential type is a record of the tag and the method table. As a result, we are able to express itables in objects whose “exact” types are unknown at compile time.

There remains one problem: method table types. There is no information about methods in the method table of each itable entry. But during the itable searching process, once there is an entry with the same tag as the target interface, the method table in that entry must contain all methods of the target interface. LIL\textsubscript{CI} has two views of method tables: an abstract view with unknown methods, and a concrete view with known ones. For the former, LIL\textsubscript{CI} uses type \( \text{Imty}(\tau_1, \tau_2) \) for the method table of interface \( \tau_1 \) in class \( \tau_2 \).
For the latter a record type describes all methods. An abstract view of type $Imty(I, C)$ can be coerced to and from a record with the corresponding record type. Again, the coercions are runtime no-ops.

The itable of class $C$ includes an array of element type $\exists \alpha \ni I \ni C$. \{tag : Tag(\alpha), mtable : Imty(\alpha, C)\}. The type variable $\alpha$ hides the actual interface name in each itable entry. For brevity, we use $ITY(\tau)$ for $\exists \alpha \ni I \ni C$. \{tag : Tag(\alpha), mtable : Imty(\alpha, \tau)\}. The itable also keeps the number of interfaces (the array length), for the purpose of itable search.

The translation from the source language to LIL $CI$ creates itables for each class. For each interface $I$ a class $C$ implements, the translation builds an itable entry. It (1) creates a record that contains $C$’s implementations of all methods in $I$ (concrete method table); (2) coerces the record in (1) to type $Imty(I, C)$ (abstract method table); (3) creates a record of the tag of $I$ and the method table in (2); and (4) coerces the record in (3) to type $ITY(\tau)$. After creating all entries, the translation phase builds an array that contains all itable entries.

At interface invocation time, each entry in the array is fetched and the tag in that entry is compared with the target interface’s tag. In case of a match, the abstract method table can be coerced back to the concrete view, which contains all methods of the target interface.

### 2.3 Object Layout

The object layout in LIL $CI$ is standard. Each object of $C$ contains a pointer to the vtable of $C$ and the fields. The vtable of $C$ is also a record, containing the tag of $C$, the itable, and virtual method pointers.

Suppose a class $C$ has fields $f_1, \ldots, f_n$ of type $s_1, \ldots, s_n$ respectively, and methods $m_1, \ldots, m_k$ of type $t_1, \ldots, t_k$ respectively.

$$R(C) = \{vtable : \{tag : Tag(C),
  itable : \{\text{length} : \text{int}, \text{table} : ITY(C)\},
  m_1 : addThis(\exists \gamma < C. \gamma, t_1),
  \ldots,
  m_k : addThis(\exists \gamma < C. \gamma, t_k)\},
  f_1 : s_1, \ldots, f_n : s_n\}$$

The tag in the vtable of $C$ has type $Tag(C)$. The itable is a record of an array and an integer (the array length). Each virtual method in the vtable has an explicit “this” pointer as the first parameter. The “this” pointer has type $\exists \gamma < C. \gamma$, meaning that only objects of $C$ or $C$’s subclasses can be passed as the “this” parameter. The “this” pointer can be objects of $C$’s subclasses because $C$’s subclasses can inherit $C$’s method implementations.

The function $addThis(\tau_{\text{this}}, \tau)$ adds $\tau_{\text{this}}$, the type for the “this” pointer, to a function type $\tau$.

$$addThis(\tau_{\text{this}}, \forall tvs (\tau_1, \ldots, \tau_n) \rightarrow \tau) = \forall tvs(\tau_{\text{this}}, \tau_1, \ldots, \tau_n) \rightarrow \tau$$

#### 2.3.1 Approximation of Object Layout

$ApproxR(\alpha, C)$ approximates the layout of objects with exact type $\alpha$, where $\alpha$ is a subclass of $C$.

Suppose class $C$ has the same fields $f_1, \ldots, f_n$ and methods $m_1, \ldots, m_k$ as in the definition of $R$, and $\alpha < C$.

$$ApproxR(\alpha, C) = \{vtable : \{tag : \text{Tag}(\alpha),
  itable : \{\text{length} : \text{int}, \text{table} : ITY(\alpha)\},
  m_1 : addThis(\exists \gamma < \alpha. \gamma, t_1),
  \ldots,
  m_k : addThis(\exists \gamma < \alpha. \gamma, t_k)\},
  f_1 : s_1, \ldots, f_n : s_n\}$$

$ApproxR(\alpha, C) = \{vtable : \{tag : \text{Tag}(\alpha),
  itable : \{\text{length} : \text{int}, \text{table} : ITY(\alpha)\}\}$$

Objects of $\alpha$ must have fields $f_1, \ldots, f_n$ and methods $m_1, \ldots, m_k$, although they have their own tag (of type $\text{Tag}(\alpha)$), itable, and “this” pointer type $\exists \gamma < \alpha. \gamma$. $ApproxR(\alpha, C)$ contains only a vtable and the vtable contains only the tag and the itable of $\alpha$. 

3


\[
\text{fix } ILookup(\alpha, \beta)(t_\alpha : \text{Tag}(\alpha), \text{obj} : \beta) : \text{Imty}(\alpha, \beta) = \begin{cases}
\text{if } (i \geq \text{len}) \text{ then error[Imty}(\alpha, \beta)] \text{ else } & \text{/not found} \\
\text{entryi} : \exists \gamma \gg \beta. \{\text{tag} : \text{Tag}(\gamma), \text{mtable} : \text{Imty}(\gamma, \beta)\} = \text{it}[i] \text{ in } & \text{/get ith entry} \\
(\gamma', \text{entryi}') = \text{open(entryi)} \text{ in } & \text{/entryi} \\
\text{ifEqTag}^{\text{Imty}(\alpha, \beta)}(\text{entryi}'.\text{tag}, t_\alpha) \text{ then } \text{entryi}'.\text{mtable} \text{ else } & \text{/\gamma' = \alpha} \\
\text{loop}[\alpha, \beta](t_\alpha, \text{obj}, \text{it}, i + 1, \text{len}) & \text{/next iteration}
\end{cases}
\]

Figure 1: Interfaces Lookup

2.4 Interface Method Invocation

Itable-based interface method invocation searches an object’s itable for the target interface, if the object’s exact type is unknown at compile time. The searching process is expressed as a polymorphic function applicable to any object and any target interface. It is a loop that iterates over each entry in the itable. Figure 1 shows the function. Suppose an object has type \(\alpha\) and the target interface is \(I\). Each entry in \(\beta\)’s itable has type \(\exists \gamma \gg \beta\). \(\{\text{tag} : \text{Tag}(\gamma), \text{mtable} : \text{Imty}(\gamma, \beta)\}\). Type variable \(\gamma\) hides the actual interface to which the entry corresponds. At each iteration, an entry is first opened, then the tag of the entry is fetched and compared with the tag of the target interface \(I\). If not the same, then the loop continues to the next iteration. Otherwise, the type checker refines the type of the method table to \(\text{Imty}(I, \beta)\), which can be coerced to a record that contains all \(I\)’s methods.

The lookup function returns the abstract method table (of type \(\text{Imty}(\alpha, \beta)\)) for the target interface \(\alpha\) in class \(\beta\). Each call site instantiates \(\alpha\) with the concrete target interface name.

LIL\(_{CI}\) can express optimizations such as caching and moving-to-front without difficulties. The cached entry can have the same type as the entries in the itable. The tag comparison and type refinement are still applicable. The moving-to-front strategy can also be expressed by LIL\(_{CI}\), because the array representation of itables does not care about the orders of itable entries and LIL\(_{CI}\) supports array mutation.

2.5 Interface Cast

Interface downward casts may cast an object of an arbitrary type to an interface. These casts are checked at run time. Errors are raised in case of a type mismatch. LIL\(_{CI}\) expresses the cast with a polymorphic function similar to the one for interface method invocation. The only difference is the return value. The cast function returns an object of the target interface, whereas the method search function returns the method table of the target interface. Figure 2 describes the cast routine. If in an object of type \(\beta\), the itable contains an entry for the target interface \(\alpha\), then \(\beta \ll \alpha\) according to the itable entry type. Therefore, the object can be coerced to type \(\exists \delta \ll \alpha, \delta\), meaning that the exact type of the object implements interface \(\alpha\).

3 Syntax of LIL\(_{CI}\)

This section explains the syntax of LIL\(_{CI}\) (Figures 3, 4 and 5). To emphasize the difference between LIL\(_{CI}\) and LIL\(_C\), we focus on the new constructs in LIL\(_{CI}\) (underlined in the figures). In each syntax category, we first summarize the constructs already in LIL\(_C\), and then introduce the new ones.
3.1 Kinds and Types

\[
\kappa ::= \Omega | \Omega_c \\
\tau ::= \text{int} | C | \bar{\text{I}} | \text{Imty}(\tau_1, \tau_2) | \text{Tag}(\tau) | \alpha | \text{array}(\tau) \\
| \forall \alpha \ll \tau, \tau' | \exists \alpha \ll \tau, \tau' | \exists \alpha \gg \tau, \tau' | (\tau_1, \ldots, \tau_n) \rightarrow \tau \\
| \{l_1^{\alpha_1} : \tau_1, \ldots, l_n^{\alpha_n} : \tau_n\} | \{(l_1^{\alpha_1} : \tau_1, \ldots, l_n^{\alpha_n} : \tau_n)\}
\]

\text{tvs} ::= \bullet | \alpha \ll \tau, \text{tvs} \\
\phi ::= \text{I} | \text{M}
\]

Figure 3: Kinds and Types

The extensions needed to support interfaces are limited: interface names, abstract interface method table types, existential types with lower subclassing bounds, and coercions between concrete and abstract interface method tables.

3.2 Values and Expressions

Figure 4 shows the values and the expressions of LIL_{CI}. Word-size values include integer “n”, heap label “ℓ”, object of class C “C(v)” (coerced from record v), tag of class C “tag(C)”, and packed word-size values “pack τ as α \ll \tau_n in (v : \tau)^\prime”.

```haskell
fix InterfaceCast(α, β)(tα : Tag(α), obj : β : ITY(β)) =
// tα : tag of target interface α, obj : object to cast
itable : \{length : int, table : ITY(β)\} = c2r(obj).itable in
it : ITY(β) = itable.table in
len : int = itable.length in
loop(α, β)(tα, obj, it, 0, len)

fix loop(α, β)(tα : Tag(α), obj : β, it : ITY(β), i : int, len : int) : \exists δ \ll α. δ =
if(i \geq len) then error[δ \ll α. δ] else
entryi : \exists γ \gg β. \{tag : Tag(γ), mttable : Imty(γ, β)\} = it[i] in
(γ', entryi') = open(entryi) in
ifE QT(δ \ll α. δ) then pack β as δ \ll γ' in (obj : δ) else // γ' = α
loop(α, β)(tα, obj, it, i + 1, len)
```

Figure 2: Cast to Interfaces
\[ v ::= n \mid \ell \mid C(v) \mid \text{tag}(C) \mid \text{tag}(I) \mid r2im[I, C](v) \mid \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (v : \tau') \mid \text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (v : \tau') \]

\[ hv ::= \{ l_i = \nu_i \}_{i=1}^n \mid \{ v_0, \ldots, v_{n-1} \}^* \mid \text{fix } g(tvs)(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \epsilon_m \]

\[ e ::= x \mid n \mid \ell \mid \text{tag}(C) \mid \text{tag}(I) \mid C(e) \mid e2r(e) \mid r2im[\tau_1, \tau_2](e) \mid \text{im2r}(e) \mid \text{error}[\tau] \mid \text{new}[\tau]\{ l_i = e_{i,1} \}_{i=1}^n \mid e.l \mid e_1.d_i := e_2 \text{ in } e_3 \mid \text{new}[e_0, \ldots, e_{n-1}]^* \mid e_1(e_2) \mid e_1(e_2) := e_3 \text{ in } e_4 \mid x : \tau = e_1 \text{ in } e_2 \mid x := e_1 \text{ in } e_2 \mid e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n) | (\alpha, x) = \text{open}(e_1) \text{ in } e_2 \mid \text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (e : \tau') \mid \text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (e : \tau') \mid \text{ifParent}(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2 \mid \text{ifEqTag}(e_1, e_2) \text{ then } e_1 \text{ else } e_2 \]

Figure 4: Values and Expressions

New word-size values include: tag of interface I “\text{tag}(I)”, abstract method table for interface I in class C “\text{r2im}[I, C](v)” (coerced from concrete table v), and packed values with lower bounds “\text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (v : \tau’)”.

Heap values remain the same as in LIL_C: record “\{l_i = \nu_i \}_{i=1}^n”, array “[\nu_0, \ldots, \nu_{n-1}]^*”, and function “\text{fix } g(tvs)(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \epsilon_m”. Arrays are annotated with their element types. Functions may be polymorphic. The return type is also specified in each function.

Besides values, LIL_CJ has the following expressions: variable “\(x\)”, coercions between objects and records “C(e)” and “C2R(e)”, runtime error expression “\text{error}[\tau]” (an expression of type \(\tau\) is expected in normal execution), record creation “\text{new}[\tau]\{ l_i = e_{i,1} \}_{i=1}^n”, field access “e.l”, field assignment “e_1.d_i := e_2 \text{ in } e_3”, array creation “\text{new}[e_0, \ldots, e_{n-1}]^*”, array subscript “e_1[e_2]”, array element assignment “e_1[e_2] := e_3 \text{ in } e_4”, let binding “x : \tau = e_1 \text{ in } e_2” (\(\tau\) specifies \(x\)’s type), variable assignment “x := e_1 \text{ in } e_2”, function call “e[\tau_1, \ldots, \tau_m](e_1, \ldots, e_n)”, open expression “\((\alpha, x) = \text{open}(e_1) \text{ in } e_2\)”, pack expression “\text{pack } \tau \text{ as } \alpha \ll \tau_u \text{ in } (e : \tau’)”, fetching-parent-tag “\text{ifParent}(e) \text{ then bind } (\alpha, x) \text{ in } e_1 \text{ else } e_2”, and tag comparison “\text{ifEqTag}(e_1, e_2) \text{ then } e_1 \text{ else } e_2”. The “\text{ifParent}” expression is used in casting objects to classes. The “\text{ifEqTag}” expression compares two tags. If the two tags are equal, the types identified by the tags are the same. The true branch uses this result to refine types.

LIL_CJ changes the syntax of the “\text{ifParent}” expression in LIL_C: it annotates the result type of the whole expression. The annotation simplifies the type checking because there is no need to compute the least upper bounds of the types of the two branches.

Two expressions in LIL_CJ describe coercions between abstract and concrete interface method tables: expression “\text{r2im}[\tau_1, \tau_2](e)” coerces the concrete table e for interface \(\tau_1\) in class \(\tau_2\) to the abstract one, and expression “\text{im2r}(e)” coerces the abstract table e to the concrete one.

To create values of existential types with lower subclassing bounds, a second pack expression “\text{pack } \tau \text{ as } \alpha \gg \tau_u \text{ in } (e : \tau’)” coerces an expression e to type “\exists \alpha \gg \tau_u. \tau’”, by hiding type \(\tau\) with type variable \(\alpha\).

### 3.3 Declarations and Programs

Figure 5 shows the declarations in LIL_CJ. Unchanged declarations include: field declaration “f : \tau”, method declaration “m : \forall tvs (\tau_1, \ldots, \tau_n) \rightarrow \tau” which specifies only method signatures, heap declaration “H” which maps labels to heap values, and variable-value mapping “\(V\)”. Method implementations are functions allocated on the heap.

Class declarations in LIL_CJ specify super interfaces, besides super classes. Programs declare both classes and interfaces.

An interface declaration “I \{\text{method}\}” specifies an interface name I, all its super interfaces \(\overline{I}\), and declarations of all the methods in the interface and its super interfaces \text{method}. \(\overline{\rho}\) means a sequence of items in \(\rho\).
As in Section 3, this section summarizes rules that already exist in LIL environment ∆ tracks type variables in scope and their bounds. Each entry in ∆ introduces a new type variable in quantified types must have kind Ω.

The subclassing judgment Θ; ∆ ⊩ τ₁ ≪ τ₂ requires that both τ₁ and τ₂ have kind Ω.κ.

The subtyping judgment Θ; ∆ ⊩ τ₁ ≪ τ₂ means that, under environments Θ and ∆, τ₁ is a subclass of τ₂. Subtyping is reflexive and transitive. The only change that interfaces bring to subtyping is the

### 4 LIL<sub>CI</sub> Semantics

As in Section 3, this section summarizes rules that already exist in LIL<sub>CI</sub>, and focuses on the new ones.

#### 4.1 Static Semantics

LIL<sub>CI</sub> maintains a declaration table Θ that maps class and interface names to their declarations. A kind environment ∆ tracks type variables in scope and their bounds. Each entry in ∆ introduces a new type variable and an upper or lower bound of the type variable. The bound is a class name, an interface name, or another type variable introduced previously in ∆. A heap environment Σ maps labels to types. A type environment Π maps variables to types. Substitution τ/α means replacing α with τ.

#### 4.1.1 Types

The kinding judgment Θ; ∆ ⊩ τ : κ means that, under environments Θ and ∆, type τ has kind κ. Figure 6 shows the kinding rules.

Non-standard kinding rules are those related to Ω, the kind that classifies class and interface names. All type variables have kind Ω<sub>c</sub>. The tag constructor can be applied only to types with kind Ω<sub>c</sub>. Bounds of type variables in quantified types must have kind Ω<sub>c</sub>. Interface names have kind Ω<sub>i</sub>. Type Imty(τ₁, τ₂) requires that both τ₁ and τ₂ have kind Ω<sub>c</sub>.

The subtyping judgment Θ; ∆ ⊩ τ₁ ≪ τ₂ means that, under environments Θ and ∆, τ₁ is a subclass of τ₂. Subtyping is reflexive and transitive. The only change that interfaces bring to subtyping is the

---

<sup>1</sup>LIL<sub>CI</sub> can easily add type variables of kind Ω, but there is no need at present.
subtyping between interfaces or between classes and interfaces. Figure 7 shows the subtyping rules.

The subtyping judgment $\Theta; \Delta \vdash \tau_1 \leq \tau_2$ means that, under environments $\Theta$ and $\Delta$, $\tau_1$ is a subtype of $\tau_2$. Figure 8 shows the subtyping rules. As expected, there are standard structural record prefix and depth subtyping, function subtyping, reflexivity and transitivity. Exact record types are subtypes of normal record subtyping between the body types. Contrary to LIL has an additional typing rule of the open expression, for elimination of existential types with lower quantified types.

The subtyping rule for existential types with lower subclassing bounds is in the same spirit as other quantified types.

4.1.2 Expressions

The typing judgment $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ means that, under environments $\Theta$, $\Delta$, $\Sigma$ and $\Gamma$, expression $e$ has type $\tau$. Figure 9 shows new expression typing rules.

Most typing rules are straightforward, except for coercions between records and objects and for the tag comparison expression. We omit the details here.

$R_I$ is another macro used by the type checker besides $R$ and $ApproxR$. It represents concrete method table types. $R_I$ is also a parameter of the type system and can be easily replaced.

Suppose interface $I$ declares methods $m_1, \ldots, m_k$ of type $\tau_1, \ldots, \tau_k$ respectively. $R_I(I, \tau)$ is defined as follows.

$$R_I(I, \tau) = \{m_1 : addThis(\exists \gamma \ll \tau. \gamma, \tau_1), \ldots, m_k : addThis(\exists \gamma \ll \tau. \gamma, \tau_k)\}$$

$LIL_{CI}$ has an additional typing rule of the open expression, for elimination of existential types with lower subclassing bounds. Also, it has a typing rule for the new pack expression. Both rules are straightforward.

In "ifParent"($e$) then bind ($\alpha$, $x$) in $e_1$ else $e_2$, tag $e$ has type Tag($\tau'$) for some type $\tau'$. If $e$ has a parent tag, a new type variable $\alpha$ is introduced for the super class of $\tau'$ and a new value variable $x$ (of type Tag($\alpha$)) is introduced for the parent tag. Both branches $e_1$ and $e_2$ should have specified type $\tau$. 

<table>
<thead>
<tr>
<th>$\Theta(I) = I : I_1, \ldots, I_n$ for $1 \leq i \leq n$</th>
<th>$\Theta(C) = C : B, I_1, \ldots, I_n$ for $\tau = B$ or $I_i(1 \leq i \leq n)$</th>
<th>$\Theta; \Delta \vdash \tau : \Omega$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta; \Delta \vdash I \ll I_i$</td>
<td>$\Theta; \Delta \vdash C \ll \tau$</td>
<td>$\Theta; \Delta \vdash \tau \ll \Omega$</td>
</tr>
</tbody>
</table>

Figure 7: Subtyping Rules

$$m \geq n$$

$\Theta; \Delta \vdash \{t_i^{\phi_i} : \tau_i\}_{i=1}^m \leq \{t_i^{\phi_i} : \tau_i\}_{i=1}^n$ st_breadth

$\forall 1 \leq i \leq m, \begin{cases} 
\Theta; \Delta \vdash \tau_1 \leq \tau_i' & \text{if } \phi_i = 1 \\
\Theta; \Delta \vdash \tau_1 \leq \tau_1' & \text{if } \phi_i = M
\end{cases}$ st_depth

$\Theta; \Delta \vdash \{t_i^{\phi_i} : \tau_i\}_{i=1}^m \leq \{t_i^{\phi_i} : \tau_i\}_{i=1}^n$ st_exact

$\Theta; \Delta \vdash \{\exists \alpha < u_1, \tau_1\} \leq \{\exists \alpha < u_2, \tau_2\}$ st_\exists

$\Theta; \Delta \vdash u_1 \ll u_2 \quad \Theta; \Delta, \alpha \ll \text{Topc} \quad \tau_1 \leq \tau_2$ st_\exists′

$\Theta; \Delta \vdash u_2 \ll u_1 \quad \Theta; \Delta, \alpha \ll \text{Topc} \quad \tau_1 \leq \tau_2$ st_\exists′

$\Theta; \Delta \vdash \forall \alpha < u_1, \tau_1 \leq \forall \alpha < u_2, \tau_2$ st_\forall

$\Theta; \Delta \vdash \tau_1 \leq \tau_2$ st_trans

$\Theta; \Delta \vdash \tau \ll \tau$ st_ref

Figure 8: Subtyping Rules

<table>
<thead>
<tr>
<th>$\Theta; \Delta \vdash t_i \ll s_i \quad \forall 1 \leq i \leq n$</th>
<th>$\Theta; \Delta \vdash s \ll t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta; \Delta \vdash (s_1, \ldots, s_n) \rightarrow s \ll (t_1, \ldots, t_n) \rightarrow t$</td>
<td>st_depth</td>
</tr>
</tbody>
</table>

The subtyping rule for existential types with lower subclassing bounds is in the same spirit as other quantified types.
Figure 9: Expression Typing Rules
The typing rules for heap values are unchanged (see Figure 10).

\[
\begin{align*}
\Theta;\bullet;\Sigma; \vdash v_i : \tau & \quad \forall 0 \leq i \leq n-1 \quad \text{hv\_array} \\
\Theta;\Sigma; \vdash [v_0, \ldots, v_{n-1}]^T : \text{array}(\tau) & \\
\Theta;\Sigma; \vdash \{l_i = v_i\}_{i=1}^n : \{t_i^0 : \tau_i\}_{i=1}^n & \quad \text{hv\_record} \\
\end{align*}
\]

\[
\tau_f = \forall \text{tvs} (\tau_1, \ldots, \tau_n) \rightarrow \tau \quad \Theta;\text{tvs};\Sigma; g : \tau_f, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_m : \tau \\
\Theta;\Sigma; \vdash \text{fix } g(\text{tvs})(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e_m : \tau_f
\]

Figure 10: Heap Value Typing Rules

### 4.1.3 Declarations and Programs

\[
\begin{align*}
\Theta; \vdash \text{Topc}\{\} \\
\Theta; \vdash B : \{ f_1 : s_1, \ldots, f_p : s_p, m_1 : t_1, \ldots, m_q : t_q \} & \quad B \text{ declared before } C \\
p \leq n & \quad q \leq j & \Theta; \vdash s_i : \Omega \quad \forall 1 \leq i \leq n & \Theta; \vdash t_i : \Omega \quad \forall q + 1 \leq i \leq j \\
\forall 1 \leq i \leq k, \Theta; \vdash I_i : \{\} & \quad \forall \text{methods } m \text{ in } I_i, m \in m_1, \ldots, m_j \\
\end{align*}
\]

\[
\Theta; \vdash C : B, I_1, \ldots, I_k \{ f_1 : s_1, \ldots, f_n : s_n, m_1 : t_1, \ldots, m_j : t_j \} \\
\forall 1 \leq i \leq n, I_i \text{ declared before } I & \quad \Theta; \vdash I_i : \{\} \\
\forall \text{methods } m \text{ in } I_i, m \in m_1, \ldots, m_j & \quad \Theta; \vdash I : I_1, \ldots, I_n\{m_1 : t_1, \ldots, m_j : t_j\}
\]

Figure 11: Well-formedness of Class and Interface Declarations

Figure 11 shows the well-formedness rules for class and interface declarations. Well-formedness of class declarations in LILCI is similar to the one in LILC, except for super interfaces. A well-formed class contains all the members inherited from its parent class and super interfaces, which is solely an implementation strategy to simplify looking up members. The parent class must be declared before the child class.

A well-formed interface contains all the methods of its super interfaces, with same types. The super interfaces must be declared before the sub-interface.

\[
\begin{align*}
\text{domain}(V) = \text{domain}(\Gamma) & \quad \Theta;\bullet;\Sigma; \vdash V(x) : \Gamma(x) \quad \forall x \in \text{domain}(V) \\
\Theta;\Sigma; \vdash V : \Gamma & \quad \text{domain}(H) = \text{domain}(\Sigma) \\
\Theta;\Sigma; \vdash H(\ell) : \Sigma(\ell) \quad \forall \ell \in \text{domain}(H) & \quad \Theta; H : \Sigma \\
\Theta = \text{decl}_1, \ldots, \text{decl}_n & \quad \Theta \vdash \text{decl}_i \quad \forall 1 \leq i \leq n \\
\Theta; H : \Sigma & \quad \Theta; \Sigma; \vdash V : \Gamma \\
\Theta;\bullet;\Sigma; \vdash e : \tau & \quad \Theta;\Sigma; \vdash (\Theta; H; V; e) : \tau
\end{align*}
\]

Figure 12: Well-formedness of Programs

A program \((\Theta; H; V; e)\) is well-formed with respect to a heap environment if all its declarations in \(\Theta\) are well-formed (including interface declarations), the heap \(H\) respects the heap environment, the variable-value binding \(V\) is well-formed and the main expression \(e\) is well-typed (see Figure 12). The main expression \(e\) has no free type variables, and refers only to labels in \(H\) and variables in \(V\).
4.2 Dynamic Semantics

Figure 13 shows the evaluation rules for the LILC1 expressions. The programs in the first column evaluate one step to the corresponding ones in the second column, if the side conditions in the third column hold. Judgment $P \rightarrow P'$ means that $P$ steps to $P'$. Expressions $r2im[\tau_1, \tau_2](v)$ and $im2r(v)$ coerce between concrete and abstract interface method tables. The open expression can eliminate new packed values of existential types with lower subclassing bounds.

4.3 Properties of LILC1

Both soundness and decidability of LILC are preserved by LILC1. Both properties are proved in standard ways: the former is proved by progress and preservation, and the latter is proved by the minimal typing property and decidability of subtyping. The proofs can be found in Appendix C.

Theorem 1 (Preservation) If $\Sigma \vdash P : \tau$ and $P \rightarrow P'$, then $\exists \Sigma' \text{ such that } \Sigma' \vdash P' : \tau$.

Theorem 2 (Progress) If $\Sigma \vdash P : \tau$, then the main expression in $P$ is a value, or $\exists P'$ such that $P \rightarrow P'$.

Theorem 3 (Decidability) It is decidable whether $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ holds.

LILC1 no longer needs the bounded join and meet property of LILC, because it adds type annotations to conditional branch expressions. We speculate that, we can eliminate the type annotation on the “ifParent” expression and add a restricted form of union and intersection types to make the type system still decidable.

5 Source Language and Translation to LILC1

This section describes the source language and formalizes the translation from the source language to LILC1. Figure 14 shows the syntax of the source language.

The source language is roughly Featherweight Java (FJ) [13] enhanced with interfaces, assignments, and one dimensional arrays of objects. All local variables and record fields are mutable. Variables are renamed such that no two variables have the same name. A “new” expression is used to create objects instead
of constructors as in FJ. Figure 14 shows the syntax. Compared with the source language of LIL C, this
language has interface names, expression “imthd[I.m]e(e1, . . . , en)” which invokes an interface method m
(in interface I) on e with arguments e1, . . . , en, and interface cast expression cast[I](e). Interface array cast
is similar to class array cast and is omitted. An interface declaration contains a sequence of super interfaces
and a sequence of method signatures. For simplicity, the source language requires that methods declared
in each interface be different from those in the super interfaces. The semantics of the source language is
straightforward and is in Appendix A.

The source-LIL C I translation performs several tasks, besides lowering types and expressions. First, it
collects members for each class and each interface, including those from super classes or interfaces. Second,
it lifts method definitions to global functions, after adding the “this” parameter. Therefore after translation,
method declarations have only method signatures, not method bodies. Third, it creates itables and vtables for
classes.

Type Translation Class and interface names are translated to existential types. Covariant array types
are translated to existential types that hide the actual element types. The type translation is shown as follows:

\[
\begin{align*}
\tau & \quad := \quad \text{int} \mid C \mid I \mid \text{array}(C) \mid \text{array}(I) \\
e & \quad := \quad n \mid \ell \mid x : \tau = e_1 \text{ in } e_2 \mid x := e_1 \text{ in } e_2 \\
& \quad \quad | \text{new } C\{f_1 = e_1, . . . , f_n = e_n\} \mid e.f_i \\
& \quad \quad | e_1.f_i := e_2 \text{ in } e_3 \mid e.m(e_1, . . . , e_n) \mid \text{imthd}[I.m]e(e_1, . . . , e_n) \\
& \quad \quad | \text{new}[e_0, . . . , e_{n-1}]^\tau \mid e_1[e_2] := e_3 \text{ in } e_4 \mid e_1[e_2] \\
& \quad \quad | \text{cast}[C](e) \mid \text{cast}[C][\text{[}](e) \mid \text{cast}[I](e) \\
v & \quad := \quad n \mid \ell \\
hv & \quad := \quad C\{f_1 = v_1, . . . , f_n = v_n\} \mid [v_0, . . . , v_{n-1}]^\tau \\
\text{mdecl} & \quad := \quad \tau \ m(x_1 : \tau_1, . . . , x_n : \tau_n) = e \\
\text{msig} & \quad := \quad \tau \ m(x_1 : \tau_1, . . . , x_n : \tau_n) \\
\text{idcl} & \quad := \quad I : J_1, . . . , J_n\{\text{msig}_1, . . . , \text{msig}_k\} \\
\text{cdecl} & \quad := \quad C : B, I_1, . . . , I_p\{f_1 : \tau_1, . . . , f_n : \tau_n, \text{mdecl}_1, . . . , \text{mdecl}_k\} \\
\text{decl} & \quad := \quad \text{cdecl} \mid \text{idcl} \\
\text{prog} & \quad := \quad \text{decl}_1, . . . , \text{decl}_n \text{ in } e
\end{align*}
\]

Type Translation

Figure 14: Syntax of the Source Language

Method Translation A method declaration is translated to a pair of a function label and a global
function definition with an explicit “this”. The translation of a method needs the enclosing class name to
specify the “this” pointer’s type. Suppose class C implements a method m. The translation generates a new
label \(l_m\) and translates the method as follows:

\[
\begin{align*}
|\tau \ m(x_1 : \tau_1, . . . , x_n : \tau_n) = e, C| & = \\
(l_m, \text{fix } m\{\text{this} : \exists \alpha \ll C. \alpha, x_1 : |\tau_1|, . . . , x_n : |\tau_n|\} : |\tau| = |e|)
\end{align*}
\]

A method signature in the source language is translated to an \text{LIL} C I method declaration:

\[
|\tau \ m(x_1 : \tau_1, . . . , x_n : \tau_n)| = m : (|\tau_1|, . . . , |\tau_n|) \rightarrow |\tau|
\]

Expression Translation Figure 15 shows the translation of expressions. Each new variable introduced
during translation is unique. The translation of expression new \(C\{f_i = e_i\}_{i=1}^n\) creates a record with \text{vtable}_C
(vtable of C) and fields \(f_1, . . . , f_n\) and then coerces the record to an object. The translation of array store
instructions inserts store checks: the object to be stored is first cast to the target element type.

Figure 14: Syntax of the Source Language
The “ILookup” function defined in Section 2 is used to search itables in the translation of the interface method invocation expression “imthd[I.m]e₁,...,eₙ]”. The polymorphic function “InterfaceCast” is instantiated with the target interface I and the dynamic type α of the object to cast, during the translation of the interface cast expression “cast[I](e)”.

**Interface Translation** A source interface declaration is translated to an LILₖₜ interface declaration. The translation collects all super interfaces and all methods including those in the super interfaces. Function iclosure computes the interface closure of a class or an interface. Definitions of iclosure on classes are in the explanation of class declaration translation. Function mclosure computes method closures. The notation “⊕” represents sets, and “+” represents set union operation. Duplicated elements are removed after set union. Interfaces are translated as follows:

\[
\begin{align*}
iclosure(I : I₁, \ldots, Iₙ\{\omega\}) &= \niclosure(I₁) + \cdots + \niclosure(Iₙ) + <I > \\
mclosure(I : I₁, \ldots, Iₙ\{msig₁, \ldots, msigₖ\}) &= \\
&= \mclosure(I₁) + \cdots + \mclosure(Iₙ) + <msig₁, \ldots, msigₖ > \\
mclosure(I :\{msig₁, \ldots, msigₖ\}) &= msig₁', \ldots, msigₖ' \\
iclosure(I : J₁, \ldots, Jₙ\{\omega\}) &= J₁', \ldots, Jₙ' |msigᵢ' = mdeclᵢ' ∀1 ≤ i ≤ q \\
\{J₁', \ldots, Jₙ'\{mdecl₁', \ldots, mdeclₗ\} &= I : J₁', \ldots, Jₙ'\{mdecl₁', \ldots, mdeclₗ\}
\end{align*}
\]

**Itable Entry Creation** When the translation creates the itable for class C, it first creates an entry for each interface in the interface closure of C. Each pair of (interface, class) is translated to an LILₖₜ interface declaration. The first record contains all translated methods in the method closure of the interface. The first two records are static heap values. The last is used to create the itable. The creation of itable entries is as follows:

\[
\begin{align*}
mclosure(I) &= msig₁', \ldots, msigₖ' \\
mdecl₁' \text{ declares } mᵢ, \forall 1 ≤ i ≤ q \\
mdecl₁' \text{ defines } mᵢ \in C \text{ for } mdecl₁', C = (l₁, \ldots) \forall 1 ≤ i ≤ q \\
|I, C| &= (|I|, θ₀, |C|) ; θ₀ = |mdecl₁', C| = (l₁, \ldots) \forall 1 ≤ i ≤ q) \\
\end{align*}
\]
Class Translation A class declaration is translated to a triple: a class declaration in LIL_{CI}, function definitions that correspond to method declarations, and the vtable. The translation creates an array that contains all itable entries. The itable is a record of the array and the array length. The operator $\oplus$ stands for concatenation with masking: if $S$ is a sequence of bindings, then $S \oplus b$ means “$S, b$” if $S$ does not have the item bound in $b$. Otherwise, $S \oplus b$ means replacing with $b$ the corresponding entry in $S$. Auxiliary function $\text{mtype}(\Theta, m, C)$ returns the type of method $m$ in class $C$. The following shows the class translation:

$$\text{iclosure}(\text{Topc}) = []$$
$$\text{iclosure}(C : B, I_1, \ldots, I_n\{\ldots\}) = \text{iclosure}(B) + \text{iclosure}(I_1) + \ldots + \text{iclosure}(I_n)$$
$$|\text{Topc}| = (\text{Topc}\{\}, <+>, \text{vtable}_{\text{Topc}} \leadsto \{ \text{tag} = \text{tag}('\text{Topc}'), \text{itable} = \{\text{length} = 0, \text{table} = []\} \})$$

$$|B| = (B : \varnothing\{\text{fields}_B, \text{mtype}_B\}, \varnothing\text{vtable}_{B} \leadsto \{ \text{tag} = \varnothing, \text{itable} = \varnothing, \text{fps}_B \})$$

$\text{mdecl}_i$ defines $m_i$ | $\text{mdecl}_i, C| = (l_i, \text{mbody}_i) \forall 1 \leq i \leq k$

$\text{iclosure}(C) = I'_1, \ldots, I'_p$ | $|I'_j, C| = (l'_m = r_{m_j}, l'_j = r_j, \text{entry}_j) \forall 1 \leq j \leq p$

$\text{labels} = \left( l'_1 \leadsto \text{mbody}_1, \ldots, l'_k \leadsto \text{mbody}_k, \right.$

$a \leadsto \{\text{length} = p, \text{table} = (\text{entry}_1, \ldots, \text{entry}_p)\}$

$\text{newms} = \text{mtype}_B \oplus m_1 : \text{mtype}(\Theta, m_1, C) \oplus \ldots \oplus m_k : \text{mtype}(\Theta, m_k, C)$

$\text{vtable} = \{ \text{tag} = \text{tag}(C), \text{itable} = a, \text{fps}_B \oplus m_1 = l_1 \oplus \ldots \oplus m_k = l_k \}$

$\text{newc} = C : B, I'_1, \ldots, I'_p \{ \text{fields}_B, f_1 : \tau_1, \ldots, f_n : \tau_n, \text{newms} \}$

$$\{C : B, \varnothing\{f_1 : \tau_1, \ldots, f_n : \tau_n, \text{mdecl}_1, \ldots, \text{mdecl}_k\}\} = \text{(newc, labels, vtable}_{C} \leadsto \text{vtable})$$

Program Translation A program is translated to a set of new declarations, an initial heap (which contains function definitions, itables, and vtables), and a new main expression.

**Theorem 4** The translation from the source language to LIL_{CI} preserves types. If a source program $P$ has type $\tau$, then $\Sigma_0 \vdash |P| : \tau$ holds in LIL_{CI}, where $\Sigma_0$ is the environment for the initial heap.

6 Related Work

League et al. [15] and Chen et al. [5] have studied supporting interfaces and/or multiple inheritance in typed intermediate languages.

League et al. proposed a typed intermediate language JFlint for compiling Java-like languages [15]. JFlint uses existential quantification over row variables for object encodings. It is able to express vtables and itables. Two approaches were proposed to support interfaces. The first approach uses unordered records for itable entries, which does not model itable search. The second approach pairs a specialized itable with an object when casting the object to an interface. The specialized itable contains only information of the target interface. The second approach is not faithful to the standard implementation: upcasting is no longer free.

Chen et al. proposed a model based on guarded recursive datatypes that supports multiple inheritance between classes [5]. The model preserves class names. Every method invocation is associated with a sequence of class names, for identifying methods. This approach encodes objects with functions, which differs from the standard implementation of objects as records. As a result, object layouts, including itables and vtables, cannot be addressed.

7 Future Work

As future work, we discuss a few other interface implementation techniques.

The inline cache-based strategies cache one or more most recently invoked interface method [8, 12]. Interface method invocation is replaced with a stub that first compares the target interface method with the cached ones. If there is a match, then the stub invokes the cached function pointer. Otherwise, it resorts to
the slow search. For this approach, method tags and type refinement based on method tag comparison, like tags and tag comparison, seem applicable.

Selector-based strategies assign unique ids (selectors) to identify method signatures [7]. Each class contains a table of method pointers indexed by selectors. The table could be modeled by records. Optimizations such as sparse arrays [10] or graph-coloring [9] further complicate the representation, and we leave those as future work.

8 Conclusion

This paper describes a typed intermediate language that supports interfaces, a restricted form of multiple inheritance. The language models itables with array types and existential types. It expresses the standard itable-based interface method invocation and interface cast with polymorphic functions, without extra runtime overhead. The type system is sound and decidable.

References


A Semantics of the Source Language

Figures 16–21 show the semantics of the source language.

\[
\Theta(I) = I: J_1, \ldots, J_n[\_] \quad 1 \leq i \leq n
\]

\[
\Theta(C) = C: B, I_1, \ldots, I_p[\ldots] \quad \tau = B \text{ or } I_j \quad \forall 1 \leq j \leq p
\]

\[
\Theta \vdash I \leq J_i
\]

\[
\Theta \vdash \tau_1 \leq \tau_2
\]

\[
\Theta \vdash \text{array}(\tau_1) \leq \text{array}(\tau_2)
\]

\[
\Theta \vdash \tau \leq \tau
\]

\[
\Theta \vdash \tau_1 \leq \tau_2 \quad \Theta \vdash \tau_2 \leq \tau_3
\]

\[
\Theta \vdash \tau_1 \leq \tau_3
\]

Figure 16: Subtyping of the Source Language

B Type-preserving Translation

Lemma 5  
If \( \Theta \vdash \tau_1 \leq \tau_2 \) in the source language, then \( \Theta; \bullet \vdash \tau_1 \preccurlyeq \tau_2 \) in \( \text{LIL}_{C1} \) (\( \tau_1 \) and \( \tau_2 \) are class or interface names).

- If \( \Theta \vdash \tau_1 \leq \tau_2 \) in the source language, then \( \Theta; \bullet \vdash |\tau_1| \leq |\tau_2| \) in \( \text{LIL}_{C1} \).

Proof: by induction on source language subtyping rules.

In the rest of the section, we assume a source language program \( P = \Theta \) in \( e \), and \( |P| = (|\Theta|; H_0; \bullet; |e|) \), and \( |\Theta| \vdash H_0 : \Sigma_0 \) holds in \( \text{LIL}_{C1} \):

Lemma 6  
If \( \text{fields}(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \) in the source language, then \( |\Theta|(C) = \{ f_1 : |\tau_1|, \ldots, f_n : |\tau_n|, \text{mdecls} \} \).

- If \( \text{mtype}(\Theta, m, C) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \) in the source language, then \( |\Theta|(C) = \{ \ldots, m : (|\tau_1|, \ldots, |\tau_n|) \rightarrow |\tau|, \ldots \} \).

- If \( \text{mtype}(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \) in the source language, then \( |\Theta|(I) = \{ \ldots, m : (|\tau_1|, \ldots, |\tau_n|) \rightarrow |\tau|, \ldots \} \).

- \( \forall \text{ class } C \in \Theta, \text{ if } R(C) = \{ \text{vtable} : \text{vtype}_C, \ldots \} \), then \( \Sigma_0(\text{vtable}_C) = \text{vtype}_C \).

Proof: the first three are proved by induction on the definition of \( \text{fields}(\Theta, C) \), \( \text{mtype}(\Theta, m, C) \) and \( \text{mtype}(\Theta, m, I) \) respectively. The third is proved by induction on class declaration translation rules and by that method translation preserves types.

Lemma 7  
The expression translation from the source language to \( \text{LIL}_{C1} \) preserves types. If \( \Theta; \Sigma; \Gamma \vdash E : T \) in the source language, then \( \Theta; \bullet; \Sigma_0; \Sigma; |\Gamma| \vdash |E| : |T| \) in \( \text{LIL}_{C1} \), where \( |f_1 : \tau_1, \ldots, f_n : \tau_n| = \ell_1 : |\tau_1|, \ldots, \ell_n : |\tau_n| \), and \( |x_1 : \tau_1, \ldots, x_m : \tau_m| = x_1 : M |\tau_1|, \ldots, x_m : M |\tau_m| \).

Proof: by induction over the expression typing rules in the source language.
\[ \text{mbody}(\Theta, m, C) = (\tau \ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e) \]
\[ \Theta; \bullet; this : C, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \]
\[ \Theta \vdash (C, \tau \ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e) \]
\[ \Theta \vdash \text{Topic}() \]
\[ \forall 1 \leq i \leq p, I_i \text{ declared before } I, \quad \Theta \vdash I_i, \quad \text{and } \forall 1 \leq q \leq k, \text{msig}_q \notin \text{mclosure}(I_i) \]
\[ \Theta \vdash I = I_1, \ldots, I_p \{\text{msig}_1, \ldots, \text{msig}_k\} \]
\[ \Theta \vdash B : \{\_\} \quad f_i \notin \text{fields}(\Theta, B) \forall 1 \leq i \leq n \quad \Theta \vdash (C, \text{mdecl}_i) \forall 1 \leq i \leq k \]
If \( \text{mdecl}_i \) redefines a method \( m \) in \( B \), then \( \text{mtype}(\Theta, m, C) = \text{mtype}(\Theta, m, B) \)
\[ \forall 1 \leq i \leq p, \Theta \vdash I_i : \{\_\}, \forall m \in \text{mclosure}(I_i), \text{mtype}(\Theta, m, C) = \text{mtype}(\Theta, m, I_i) \]
\[ \Theta \vdash C : B, I_1, \ldots, I_p \{f_1 : \tau_1, \ldots, f_n : \tau_n, \text{mdecl}_1, \ldots, \text{mdecl}_k\} \]
\[ \text{decl}_1, \ldots, \text{decl}_n \vdash \text{decl}_i \quad \forall 1 \leq i \leq n \quad \text{decl}_1, \ldots, \text{decl}_n; \bullet \vdash e : \tau \]
\[ \vdash \text{decl}_1, \ldots, \text{decl}_n \text{ in } e : \tau \]

Figure 17: Well-formedness of Source Programs

\[ F = f_1 : \tau_1, \ldots, f_n : \tau_n \quad \overline{M} = \text{mdecl}_1, \ldots, \text{mdecl}_k \quad \overline{MS} = \text{msig}_1, \ldots, \text{msig}_p \]
\[ \text{fields}(\Theta, \text{Topc}) = \bullet \]
\[ \Theta(C) = C : B, I_1, \ldots, I_p \{F, \overline{M}\} \quad \text{fields}(\Theta, B) = f'_1 : \tau'_1, \ldots, f'_j : \tau'_j \]
\[ \text{fields}(\Theta, C) = f'_1 : \tau'_1, \ldots, f'_j : \tau'_j, F \]
\[ \Theta(C) = C : B, I_1, \ldots, I_p \{F, \overline{M}\} \quad \tau \ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \in \overline{M} \]
\[ \text{mtype}(\Theta, m, C) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \]
\[ \text{mbody}(\Theta, m, C) = \tau \ m(x_1 : \tau_1, \ldots, x_n : \tau_n) = e \]
\[ \Theta(C) = C : B, I_1, \ldots, I_p \{F, \overline{M}\} \quad m \text{ is not declared in any of } \overline{M} \]
\[ \text{mtype}(\Theta, m, C) = \text{mtype}(\Theta, m, B) \]
\[ \text{mbody}(\Theta, m, C) = \text{mbody}(\Theta, m, B) \]
\[ \Theta(I) = I : \{\_\} \overline{MS} \quad \tau \ m(x_1 : \tau_1, \ldots, x_n : \tau_n) \in \overline{MS} \]
\[ \text{mtype}(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \]
\[ \Theta(I) = I : I_1, \ldots, I_n \{\_\} \quad \exists 1 \leq j \leq n \text{ mtype}(\Theta, m, I_j) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \]
\[ \text{mtype}(\Theta, m, I) = (\tau_1, \ldots, \tau_n) \rightarrow \tau \]

Figure 18: Helper Functions for the Source Language Typing

\[ \text{fields}(\Theta; C) = f_1 : \tau_1, \ldots, f_n : \tau_n \]
\[ \Theta; \Sigma; \Gamma \vdash v_i : \tau_i \quad \forall 1 \leq i \leq n \]
\[ \Theta; \Sigma \vdash C\{f_i = v_i\}_{i=1}^n : C \]
\[ \Theta; \Sigma; \Gamma \vdash v_i : \tau \quad \forall 0 \leq i \leq n-1 \]
\[ \Theta; \Sigma \vdash [v_0, \ldots, v_{n-1}] : \text{array}(\tau) \]

Figure 19: Heap Value Typing of the Source Language
\[ \Theta; \Sigma; \Gamma \vdash n : \text{int} \quad \Theta; \Sigma; \Gamma \vdash x : \Gamma(x) \quad \Theta; \Sigma; \Gamma \vdash \ell : \Sigma(\ell) \quad \Theta; \Sigma; \Gamma \vdash e : \tau \quad \Theta; \Sigma; \Gamma \vdash \tau \leq \tau' \]

\[ \Theta; \Sigma; \Gamma \vdash e_1 : \tau \quad \Theta; \Sigma; \Gamma, x : \tau \vdash e_2 : \tau' \quad \Theta; \Sigma; \Gamma \vdash x : \tau = e_1 \text{ in } e_2 : \tau' \]

\[ \text{fields}(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \]

\[ \Theta; \Sigma; \Gamma \vdash e_1 : \tau_i \forall i \leq i \leq n \quad \Theta; \Sigma; \Gamma \vdash e_2 : \tau_i \]

\[ \Theta; \Sigma; \Gamma \vdash \text{new } C\{f_1 = e_1, \ldots, f_n = e_n\} : C \]

\[ \Theta; \Sigma; \Gamma \vdash e_1 : C \quad \text{fields}(\Theta, C) = f_1 : \tau_1, \ldots, f_n : \tau_n \quad 1 \leq i \leq n \quad \Theta; \Sigma; \Gamma \vdash e_2 : \tau_i \]

\[ \Theta; \Sigma; \Gamma \vdash e_1 : \tau \quad \forall 0 \leq i \leq n - 1 \quad \Theta; \Sigma; \Gamma \vdash e_2 : \tau \]

\[ \Theta; \Sigma; \Gamma \vdash \text{new}[e_0, \ldots, e_{n-1}] : \text{array}(\tau) \]

\[ \Theta; \Sigma; \Gamma \vdash e_1 : \tau \quad \Theta; \Sigma; \Gamma \vdash e_2 : \text{int} \quad \Theta; \Sigma; \Gamma \vdash e_1[e_2] : \tau \]

\[ \Theta; \Sigma; \Gamma \vdash e : C \quad \Theta; \Sigma; \Gamma \vdash e : I \quad \Theta; \Sigma; \Gamma \vdash \text{cast}[\tau](e) : \tau \quad \Theta; \Sigma; \Gamma \vdash \text{cast}[C]\{e\} : \text{array}(C) \]

\[ \Theta; \Sigma; \Gamma \vdash e : C \quad \Theta; \Sigma; \Gamma \vdash e : \tau_1 \quad \Theta; \Sigma; \Gamma \vdash e : \tau_2 \]

**Figure 20: Expression Typing of the Source Language**

<table>
<thead>
<tr>
<th>Original State</th>
<th>New State</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((H; V; x))</td>
<td>((H; V; V(x)))</td>
<td>(V' = V, x = v)</td>
</tr>
<tr>
<td>((H; V; x : \tau = v \in e))</td>
<td>((H; V'; e))</td>
<td>(V = V_1, x = v, V_2 ) and (V = V_1, x = v', V_2)</td>
</tr>
<tr>
<td>((H; V; x := v' \in e))</td>
<td>((H; V'; e))</td>
<td>(H' = H, t \leadsto C(f_1 = v_1, \ldots, f_n = v_n))</td>
</tr>
<tr>
<td>((H; V; \text{new } C{f_i = v_i}_{i=1}^n))</td>
<td>((H'; V; \ell))</td>
<td>(H(\ell) = C{f_i = v_1, \ldots, f_n = v_n} \quad 1 \leq i \leq n)</td>
</tr>
<tr>
<td>((H; V; \ell, f_i))</td>
<td>((H; V; v_i))</td>
<td>(\forall \ell' \neq \ell \quad H'(\ell') = H(\ell'))</td>
</tr>
<tr>
<td>((H; V; \ell, f_i := v'_i \in e))</td>
<td>((H'; V; e))</td>
<td>(H(\ell) = C{f_i = v'_1, \ldots, f_n = v_n})</td>
</tr>
<tr>
<td>((H; V; \ell, m(v_1, \ldots, v_p)))</td>
<td>((H'; V; e))</td>
<td>(mbody(\Theta, m, C) = \tau \quad m(x_1 : \tau_1, \ldots, x_p : \tau_p) = e)</td>
</tr>
<tr>
<td>((H; V; \text{imthd}[m]e(e_1, \ldots, e_n)))</td>
<td>((H'; V; e))</td>
<td>(H(\ell) = C{f_i = v_i}_{i=1}^n \quad \ell \notin \text{domain}(H))</td>
</tr>
<tr>
<td>((H; V; \text{new}[v_0, \ldots, v_{n-1}]^\tau))</td>
<td>((H'; V; \ell))</td>
<td>(H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad \ell \notin \text{domain}(H))</td>
</tr>
<tr>
<td>((H; V; \ell[i]))</td>
<td>((H; V; v_i))</td>
<td>(H(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad 0 \leq i \leq n - 1)</td>
</tr>
<tr>
<td>((H; V; \ell[i] := v'_i \in e))</td>
<td>((H'; V; e))</td>
<td>(\forall \ell' \neq \ell \quad H'(\ell') = H(\ell'))</td>
</tr>
<tr>
<td>((H; V; \text{cast}<a href="e">\ell</a>))</td>
<td>((H'; V; e))</td>
<td>(H'(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad 0 \leq i \leq n - 1)</td>
</tr>
<tr>
<td>((H; V; \text{cast}[C]{e}))</td>
<td>((H'; V; e))</td>
<td>(H'(\ell) = [v_0, \ldots, v_{n-1}]^\tau \quad 0 \leq i \leq n - 1)</td>
</tr>
<tr>
<td>((H; V; \cdot))</td>
<td>((H'; V; e))</td>
<td>(H(\ell) = D{f_i = v_i}_{i=1}^n \quad \Theta \vdash D \leq C)</td>
</tr>
<tr>
<td>((H; V; \text{cast}[\ell]{e}))</td>
<td>((H'; V; e))</td>
<td>(H(\ell) = [v_0, \ldots, v_{n-1}]^D \quad \Theta \vdash D \leq C)</td>
</tr>
<tr>
<td>((H; V; \text{cast}[C]{\ell}))</td>
<td>((H; V; e))</td>
<td>(H(\ell) = C{f_i = v_i}_{i=1}^n \quad I \in \text{closure}(C))</td>
</tr>
</tbody>
</table>

**Figure 21: Dynamic Semantics of the Source Language**
C  Proofs of Decidability and Soundness

LILCI’s decidability and soundness are proved similar to those of LILC. LILCI does not need joins and meets of types because LILCI has type annotations on conditional branch expressions.

C.1 Supporting Lemmas for Decidability and Soundness

Lemma 8  Weakening of kind environment:
Suppose \( \alpha \not\in \text{domain}(\Delta) \), and \( \Delta = \Delta; \alpha \ll u \) or \( \Delta' = \Delta; \alpha \gg u \):

1. If \( \Theta; \Delta \vdash \tau : \kappa \), then \( \Theta; \Delta' \vdash \tau : \kappa \).
2. If \( \Theta; \Delta \vdash \tau_1 \ll \tau_2 \), then \( \Theta; \Delta' \vdash \tau_1 \ll \tau_2 \).
3. 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. \( \square \)

Lemma 9  Weakening of heap environment. If \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \( \Sigma' = \Sigma, \ell : \tau \) where \( \ell \) is a fresh label, then \( \Theta; \Delta; \Sigma'; \Gamma \vdash e : \tau \).

proof: by induction on expression typing rules. \( \square \)

Lemma 10  Weakening of type environment. If \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \( \Gamma' = \Gamma, x : \tau \) or \( \Gamma' = \Gamma, x : \eta \) where \( x \) is a fresh variable, then \( \Theta; \Delta; \Sigma; \Gamma' \vdash e : \tau \).

proof: by induction on expression typing rules. \( \square \)

Lemma 11  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 \( \delta = \eta / \eta \) 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. \( \square \)

Lemma 12  Strengthening of environments for values. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \tau \), then \( \Theta; \bullet; \Sigma; \bullet \vdash v : \tau \).

Proof: by induction on structure of values. \( \square \)

Lemma 13  If \( \text{freetvs}(\Sigma) = \emptyset \) and \( \Theta; \Sigma \vdash V : \Gamma \), and \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \tau \), then \( \tau \) contains no type variables.

Proof: by induction on value typing rules, \( \text{freetvs}(\Gamma) = \emptyset \). By induction on expression typing rules, \( \tau \) contains no type variables. \( \square \)

C.1.1 Subclassing

We define an operation \( \tau \uparrow_\Delta \) to compute the least super class name for type \( \tau \) in environment \( \Delta \). It is used in \( \text{a_c2r_tv} \), because the approximation needs a concrete class name. The type variable lifting is defined as follows.

Definition 14  Class/interface and type variable lifting

\[
\begin{align*}
C \uparrow_\Delta &= C \\
I \uparrow_\Delta &= \text{Topc} \\
\alpha \uparrow_\Delta &= \begin{cases} 
\tau \uparrow_\Delta & \alpha \ll \tau \in \Delta \\
\text{Topc} & \alpha \gg \tau \in \Delta
\end{cases}
\end{align*}
\]
Definition 16 Type lifting and lowering. Suppose $\Delta = \Delta_1, P(\alpha)$ where $P(\alpha) = \alpha \ll u_\alpha$ or $P(\alpha) = \alpha \gg u_\alpha$:

\[
\begin{align*}
(\tau) \uparrow^\alpha_\Delta &= \tau & \alpha \notin \text{free}(\tau) \\
(\exists \beta \ll u. \tau) \uparrow^\alpha_\Delta &= \exists \beta \ll u. \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
(\exists \beta \ll \alpha. \tau) \uparrow^\alpha_\Delta &= \exists \beta \ll \alpha. \tau' & P(\alpha) = \alpha \ll u_\alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
(\exists \beta \gg u. \tau) \uparrow^\alpha_\Delta &= \exists \beta \gg u. \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
(\exists \beta \gg \alpha. \tau) \uparrow^\alpha_\Delta &= \exists \beta \gg \alpha. \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
(\forall \beta \ll u. \tau) \uparrow^\alpha_\Delta &= \forall \beta \ll u. \tau' & u \neq \alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
(\forall \beta \ll \alpha. \tau) \uparrow^\alpha_\Delta &= \forall \beta \ll \alpha. \tau' & P(\alpha) = \alpha \gg u_\alpha \text{ and } \tau' = (\tau) \uparrow^\alpha_{\Delta_1, \beta \ll \text{Topc}.P(\alpha)} \\
((\tau_1, \ldots, \tau_n) \rightarrow \tau) \uparrow^\alpha_\Delta &= (\tau_1', \ldots, \tau_n') \rightarrow \tau' & \tau_i' = (\tau_i) \uparrow^\alpha_\Delta \text{ and } \tau' = (\tau) \uparrow^\alpha_\Delta \\
\{(\ell_i^\phi : \tau_i)_{i=1}^n\} \uparrow^\alpha_\Delta &= \{(\ell_i^\phi : \tau_i')_{i=1}^n\} \uparrow^\alpha_\Delta & \begin{cases} 
 m = \max \{ j \mid \forall 1 \leq i \leq j \text{ if } \phi_i = M \text{ then } \alpha \notin \text{free}(\tau_i) \} \\
 \text{otherwise } (\tau_i) \uparrow^\alpha_\Delta \text{ exists} \end{cases} \\
\{(\ell_i^\phi : \tau_i)_{i=1}^n\} \downarrow^\alpha_\Delta &= \{(\ell_i^\phi : \tau_i')_{i=1}^n\} \downarrow^\alpha_\Delta & \begin{cases} 
 \forall 1 \leq i \leq n \text{ if } \phi_i = M \text{ then } \alpha \notin \text{free}(\tau_i) \text{ otherwise } (\tau_i) \downarrow^\alpha_\Delta \text{ exists} \end{cases}
\end{align*}
\]

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

Lemma 15 Properties of class/interface and type variable lifting

1. $\Theta; \Delta \vdash \gamma \ll \gamma \uparrow^\alpha_\Delta$.
2. If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$, then $\Theta; \Delta \vdash \tau_1 \uparrow^\alpha_\Delta \ll \tau_2 \uparrow^\alpha_\Delta$.

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.

C.1.2 Subtyping

We define type lifting and type lowering in Figure 22. $(\tau) \uparrow^\alpha_\Delta$ is defined to be the least super type of $\tau$ under environment $\Delta$ that contains no free occurrence of $\alpha$. Similarly, $(\tau) \downarrow^\alpha_\Delta$ is defined to be the greatest subtype of $\tau$ under environment $\Delta$ that contains no free occurrence of $\alpha$. Note that $\alpha$ appears as the last entry in $\Delta$. The definitions of lifting and lowering are mutually recursive.
Lemma 17 Suppose $\Delta = \Delta_1, P(\alpha)$ where $P(\alpha) = \alpha \ll u_\alpha$ or $P(\alpha) = \alpha \gg u_\alpha$, and $\beta \notin \text{domain}(\Delta)$. If $\Theta; \Delta, \beta \ll \text{Top} \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta_1, \beta \ll \text{Top} \vdash P(\alpha) \vdash \tau_1 \leq \tau_2$. Also, if $\Theta; \Delta_1, \beta \ll \text{Top} \vdash P(\alpha) \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta, \beta \ll \text{Top} \vdash \tau_1 \leq \tau_2$.

Proof: by induction on subtyping rules. 

Type lifting and lowering have the following properties:

Lemma 18 Properties of Type Lifting.
Let $\Delta' = \Delta, \alpha \ll u_\alpha$ or $\Delta' = \Delta, \alpha \gg u_\alpha$.

1. If $\exists s$ such that $\alpha \notin \text{free}(s)$ and $\Theta; \Delta' \vdash \tau \leq s$, then $(\tau) \uparrow^m_{\Delta'}$ exists, and $\Theta; \Delta' \vdash \tau \leq (\tau) \uparrow^m_{\Delta'}$, and $\forall t$ such that $\Theta; \Delta' \vdash \tau \leq t$ and $\alpha \notin \text{free}(t)$, then $\Theta; \Delta \vdash (\tau) \uparrow^m_{\Delta'} \leq t$.

2. If $\Theta; \Delta' \vdash s \leq \tau$ and $\alpha \notin \text{free}(s)$, then $(\tau) \downarrow^m_{\Delta'}$ exists and $\Theta; \Delta \vdash (\tau) \downarrow^m_{\Delta'}$, and $\forall t$ such that $\Theta; \Delta' \vdash t \leq \tau$ and $\alpha \notin \text{free}(t)$, $\Theta; \Delta \vdash (\tau) \downarrow^m_{\Delta'} \leq t$.

3. If $\Theta; \Delta' \vdash \tau_1 \leq \tau_2$, then $\Theta; \Delta \vdash (\tau_1) \uparrow^m_{\Delta'} \leq (\tau_2) \uparrow^m_{\Delta'}$.

Proof: 

1, 2: by mutual induction on the definition of lifting and lowering.

3: let $T_1 = (\tau_1) \uparrow^m_{\Delta'}$ and $T_2 = (\tau_2) \uparrow^m_{\Delta'}$. By 1, $\Theta; \Delta' \vdash t \leq T_2$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau_1 \leq T_2$. By 1 and $\alpha \notin \text{free}(T_2)$, $\Theta; \Delta \vdash \tau_1 \leq T_2$.

Lemma 19 Properties of Approximation

- $R(C)$ contains no free type variables. $\text{ApproxR}(\alpha, C)$ contains no free type variables other than $\alpha$.
- If $\Theta; \Delta \vdash C_1 \ll C_2$ and $\Theta; \Delta \vdash \alpha : \Omega$, then $\Theta; \Delta \vdash \text{ApproxR}(\alpha, C_1) \leq \text{ApproxR}(\alpha, C_2)$.
- If $\Theta; \Delta \vdash C_1 \ll C_2$, then $\Theta; \Delta \vdash R(C_1) \leq \text{ApproxR}(\alpha, C_2)[C_1/\alpha]$.

Proof: by the definitions of $R$, $\text{ApproxR}$ and record subtyping rules. 

C.2 Decidability of Type Checking

Lemma 20 Subclassing is decidable.

Proof: Subclassing is a partial order. Class names, interface names, and type variables constitute a finite partial order set. One simple way to decide whether a type $\tau_1$ is a subclass of type $\tau_2$ is by brute-forth search: first get all the types $\tau$ such that $\tau_1 \ll \tau$ and then test whether $\tau_2$ is one of the super types. 

C.2.1 Decidable Subtyping

The set of subtyping rules (shown in Figure 8) is not suitable for implementation: in the transitivity rule, the premises use a new type $\tau_2$ that does not appear in the conclusion, which means that the type checker needs to guess what $\tau_2$ to use.

To prove the decidability of subtyping, we develop a new set of subtyping rules that is equivalent to the existing set but does not have the transitivity rule, called algorithmic subtyping rules (shown in Figure 23). This new set is syntax-directed except for $\text{ast}_\text{record}$, thus can be used as an algorithm for checking subtyping.

Lemma 21 The algorithmic subtyping rules are sound. 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.

Case $\text{ast}_\text{record}$: $T_1 = \{t^0_1 : \tau_1, \ldots, t^n_1 : \tau_n, \ldots\}$, $T_2 = \{t'_1 : \tau'_1, \ldots, t'_n : \tau'_n\}$ with subderivations $\Theta; \Delta \vdash \tau_i \leq \tau'_{i'}$ if such that $\phi_i = I$, and $\tau_i = \tau'_{i'}$ otherwise.

By induction hypothesis, $\Theta; \Delta \vdash \tau_i \leq \tau'_{i'}$ if such that $\phi_i = I$. By breadth subtyping ($\text{st}_\text{breadth}$) followed by depth subtyping ($\text{st}_\text{depth}$), $\Theta; \Delta \vdash T_1 \leq T_2$.

Case $\text{ast}_\text{exact}$: $T_1 = \{t^0_1 : \tau_1, \ldots, t^n_1 : \tau_n, \ldots\}$, $T_2 = \{t^0_1 : \tau'_1, \ldots, t^n_1 : \tau'_n\}$ with subderivations $\Theta; \Delta \vdash \tau_i \leq \tau'_{i'}$ if such that $\phi_i = I$, and $\tau_i = \tau'_{i'}$ otherwise.
The algorithmic subtyping is transitive. If $\Theta; \Delta \vdash \tau_i \leq \tau_i'$ if $\phi_i = I$  
$\tau_i = \tau_i'$ if $\phi_i = M$

\[ \forall i \leq i \leq n \{ \Theta; \Delta \models \tau_i \leq \tau_i' \text{ if } \phi_i = I \] 
\[ \tau_i = \tau_i' \text{ if } \phi_i = M \]

\[ \Theta; \Delta \models \{ l^{\phi_1}_{i_1} : \tau_1, \ldots, l^{\phi_n}_{i_n} : \tau_n, \ldots \} \leq \{ l^{\phi_1'}_{i_1'} : \tau_1', \ldots, l^{\phi_n'}_{i_n'} : \tau_n', \ldots \} \]

\[ \forall n \leq i \leq n \{ \Theta; \Delta \models \tau_i \leq \tau_i' \text{ if } \phi_i = I \] 
\[ \tau_i = \tau_i' \text{ if } \phi_i = M \]

\[ \Theta; \Delta \models \{ l^{\phi_1}_{i_1} : \tau_1, \ldots, l^{\phi_n}_{i_n} : \tau_n, \ldots \} \leq \{ l^{\phi_1'}_{i_1'} : \tau_1', \ldots, l^{\phi_n'}_{i_n'} : \tau_n', \ldots \} \]

By induction hypothesis, $\Theta; \Delta \vdash \tau_i \leq \tau_i'$ for each $i$ such that $\phi_i = I$. By $\text{st_exact}$ followed by $\text{st_breadth}$ and $\text{st_depth}$, $\Theta; \Delta \vdash T_1 \leq T_2$.

**Case ast:** $T_1 = \exists \alpha \ll u_1, \tau_1, T_2 = \exists \alpha \ll u_2, \tau_2$ with subderivations $\Theta; \Delta \vdash u_1 \ll u_2$ and $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$.

By induction hypothesis $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$. By $\text{st} \exists \Theta; \Delta \vdash T_1 \leq T_2$.

**Case ast.\(\exists\)**: $T_1 = \exists \alpha \gg u_1, \tau_1, T_2 = \exists \alpha \gg u_2, \tau_2$ with subderivations $\Theta; \Delta \vdash u_2 \ll u_1$ and $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$.

By induction hypothesis $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$. By $\text{st} \exists \gg \Theta; \Delta \vdash T_1 \leq T_2$.

**Case ast.\(\forall\)**: $T_1 = \forall \alpha \ll u_1, \tau_1, T_2 = \forall \alpha \ll u_2, \tau_2$ with subderivations $\Theta; \Delta \vdash u_2 \ll u_1$ and $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$.

By induction hypothesis $\Theta; \Delta, \alpha \ll \text{Topc} \vdash \tau_1 \leq \tau_2$. By $\text{st} \forall \Theta; \Delta \vdash T_1 \leq T_2$.

**Case ast.\(\text{fun}\)**: $T_1 = (s_1, \ldots, s_n) \rightarrow s, T_2 = (t_1, \ldots, t_n) \rightarrow t$ with subderivations $\Theta; \Delta \vdash t_i \leq s_i$ for all $1 \leq i \leq n$ and $\Theta; \Delta \vdash s \leq t$.

By induction hypothesis $\Theta; \Delta \vdash t_i \leq s_i$ for all $1 \leq i \leq n$ and $\Theta; \Delta \vdash s \leq t$. By $\text{st} \text{fun} \Theta; \Delta \vdash T_1 \leq T_2$.

**Case ast.\(\text{ref}\)**: by $\text{st} \text{ref}$.

**Lemma 22** The algorithmic subtyping is transitive. 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 left derivation $\Theta; \Delta \vdash T_1 \leq T_2$ and the right derivation $\Theta; \Delta \vdash T_2 \leq T_3$.

Case 1: if either the left derivation or the right one ends with $\text{ast} \text{ref}$, then the proof obligation is simply the conclusion of the other subderivation.

In the following cases, we will not consider $\text{ast} \text{ref}$.

Case 2: both the left and the right derivations use $\text{ast} \text{record}$ as the last rule. We have $T_1 = \{ l^{\phi_1}_{i_1} : \tau_1, \ldots, l^{\phi_n}_{i_n} : \tau_n, \ldots \}$, $T_2 = \{ l^{\phi_1'}_{i_1'} : \tau_1', \ldots, l^{\phi_n'}_{i_n'} : \tau_n', \ldots \}$, $T_3 = \{ l^{\phi_1''}_{i_1''} : \tau_1'', \ldots, l^{\phi_n''}_{i_n''} : \tau_n'', \ldots \}$ for some $1 \leq p \leq n$, and $1 \leq i \leq n$, if $\phi_i = I$ then $\Theta; \Delta \vdash \tau_i \leq \tau_i'$ otherwise $\tau_i = \tau_i'$, and $1 \leq i \leq p$, if $\phi_i = I$ then $\Theta; \Delta \vdash \tau_i' \leq \tau_i''$ otherwise $\tau_i' = \tau_i''$. By induction hypothesis, $1 \leq i \leq p$, if $\phi_i = I$ then $\Theta; \Delta \vdash \tau_i \leq \tau_i''$ otherwise $\tau_i = \tau_i''$. By $\text{ast} \text{record}$, $\Theta; \Delta \vdash T_1 \leq T_3$. 

22
Case 3: the left derivation ends with $\text{ast.exact}$ and the right ends with $\text{ast.record}$. We have $T_1 = \{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n, \ldots\}$, $T_2 = \{i_1^{\phi_1} : \tau_1', \ldots, i_n^{\phi_n} : \tau_n'\}$, $T_3 = \{i_1^{\phi_1} : \tau_1'', \ldots, i_p^{\phi_p} : \tau_p''\}$ $1 \leq p \leq n,$ and $\forall 1 \leq i \leq n,$ if $\phi_i = I$ then $\Theta; \Delta \models \tau_i \leq \tau_i'$ otherwise $\tau_i = \tau_i'$, and $\forall 1 \leq i \leq p,$ if $\phi_i = I$ then $\Theta; \Delta \models \tau_i'' \leq \tau_i'''$ otherwise $\tau_i'' = \tau_i'''$. By induction hypothesis, $\forall 1 \leq i \leq p,$ if $\phi_i = I$ then $\Theta; \Delta \models \tau_i \leq \tau_i'$ otherwise $\tau_i = \tau_i''$. By $\text{ast.exact}$, $\Theta; \Delta \models T_1 \leq T_3$.

Case 4: both the left and the right derivations end with $\text{ast.}\exists$. We have $T_1 = \exists \alpha \ll u_1$, $\tau_1$, $T_2 = \exists \alpha \ll u_2$, $\tau_2$, $T_3 = \exists \alpha \ll u_3$, $\tau_3$, and subderivations $\Theta; \Delta \models u_1 \ll u_2$, $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_2$, $\Theta; \Delta \models u_2 \ll u_3$ and $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_3 \leq \tau_2$. By the transitivity of subclassing $\Theta; \Delta \models u_1 \ll u_3$. By induction hypothesis $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_3$. By $\text{ast.}\exists \Theta; \Delta \models T_1 \leq T_3$.

Case 5: both the left and the right derivations end with $\text{ast.}\exists \gg$. Similar to case 4.

Case 6: both the left and the right derivations end with $\text{ast.}\forall$. Similar to case 4.

Case 7: both the left and the right derivations end with $\text{ast.fun}$. We have $T_1 = (\tau_1, \ldots, \tau_n) \rightarrow T$, $T_2 = (\tau_1', \ldots, \tau_n') \rightarrow T'$, $T_3 = (\tau_1'', \ldots, \tau_n'') \rightarrow T''$, and subderivations $\Theta; \Delta \models \tau_i' \leq \tau_i \forall 1 \leq i \leq n$, $\Theta; \Delta \models \tau \leq \tau'$, $\Theta; \Delta \models \tau_i'' \leq \tau_i' \forall 1 \leq i \leq n$ and $\Theta; \Delta \models \tau'' \leq \tau''$. By induction hypothesis, $\Theta; \Delta \models \tau_i'' \leq \tau_i \forall 1 \leq i \leq n$ and $\Theta; \Delta \models \tau \leq \tau''$. By $\text{ast.fun} \Theta; \Delta \models T_1 \leq T_3$.

There are no other cases.

Lemma 23 The algorithmic subtyping is complete. If $\Theta; \Delta \models T_1 \leq T_2$, then $\Theta; \Delta \models T_1 \leq T_2$.

Proof: By induction on the subtyping rules.

Case $\text{st.breadth}$: $T_1 = \{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n, \ldots\}$, $T_2 = \{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n\}$

By $\text{ast.ref}$, $\Theta; \Delta \models \tau_i \leq \tau_i$, $\forall \phi_i = I$. By $\text{ast.record}$, $\Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.depth}$: $T_1 = \{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n\}$, $T_2 = \{i_1^{\phi_1} : \tau_1', \ldots, i_n^{\phi_n} : \tau_n'\}$, and $\forall 1 \leq i \leq n$ if $\phi_i = I$ then $\Theta; \Delta \models \tau_i \leq \tau_i'$, otherwise $\tau_i = \tau_i'$. By induction hypothesis $\forall 1 \leq i \leq n$ if $\phi_i = I$ then $\Theta; \Delta \models \tau_i \leq \tau_i'$, otherwise $\tau_i = \tau_i'$. By $\text{ast.record}$, $\Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.exact}$: $T_1 = \{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n\}$, $T_2 = \{i_1^{\phi_1} : \tau_1', \ldots, i_n^{\phi_n} : \tau_n'\}$

By $\text{ast.ref}$, $\Theta; \Delta \models \tau_i \leq \tau_i$, $\forall \phi_i = I$. By $\text{ast.exact}$, $\Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.}\exists$: $T_1 = \exists \alpha \ll u_1$, $\tau_1$, $T_2 = \exists \alpha \ll u_2$, $\tau_2$ with subderivations $\Theta; \Delta \models u_1 \ll u_2$ and $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_2$.

By induction hypothesis $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_2$. By $\text{ast.}\exists \Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.}\forall$: $T_1 = \forall \alpha \ll u_1$, $\tau_1$, $T_2 = \forall \alpha \ll u_2$, $\tau_2$ with subderivations $\Theta; \Delta \models u_2 \ll u_1$ and $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_2$.

By induction hypothesis $\Theta; \Delta; \alpha \ll \text{Topc} \models \tau_1 \leq \tau_2$. By $\text{ast.}\forall \Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.fun}$: $T_1 = (s_1, \ldots, s_n) \rightarrow s$, $T_2 = (t_1, \ldots, t_n) \rightarrow t$ with subderivations $\Theta; \Delta \models t_i \leq s_i \forall 1 \leq i \leq n$ and $\Theta; \Delta \models s \leq t$.

By induction hypothesis $\Theta; \Delta \models t_i \leq s_i \forall 1 \leq i \leq n$ and $\Theta; \Delta \models s \leq t$. By $\text{ast.fun} \Theta; \Delta \models T_1 \leq T_2$.

Case $\text{st.ref}$: by $\text{ast.ref}$.

Case $\text{st.trans}$: $T_1 = \tau_1$ and $T_2 = \tau_3$, with subderivations $\Theta; \Delta \models \tau_1 \leq \tau_2$ and $\Theta; \Delta \models \tau_2 \leq \tau_3$.

By induction hypothesis, $\Theta; \Delta \models \tau_1 \leq \tau_2$ and $\Theta; \Delta \models \tau_2 \leq \tau_3$. By the transitivity of the algorithmic subtyping, $\Theta; \Delta \models T_1 \leq T_2$.

Definition 24 size of types.

\begin{align*}
\text{size}(\forall \alpha \ll \tau; \tau') & = \text{size}(\tau') + 1 \\
\text{size}(\exists \alpha \ll \tau; \tau') & = \text{size}(\tau') + 1 \\
\text{size}(\exists \alpha \gg \tau; \tau') & = \text{size}(\tau') + 1 \\
\text{size}(\tau_1, \ldots, \tau_n \rightarrow \tau) & = \sum_{i=1}^{n} \text{size}(\tau_i) + \text{size}(\tau) + 1 \\
\text{size}(\{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n\}) & = \sum_{i=1}^{n} \text{size}(\tau_i) + 1 \\
\text{size}(\{i_1^{\phi_1} : \tau_1, \ldots, i_n^{\phi_n} : \tau_n\}) & = \sum_{i=1}^{n} \text{size}(\tau_i) + 1 \\
\text{size}(\tau) & = 1 \text{ otherwise }
\end{align*}
Lemma 25 The algorithmic subtyping terminates.

Proof: In all algorithmic subtyping rules, the sum of the sizes of the types in the subtyping judgments of the premises is strictly smaller than that of the conclusion. Also, the side conditions such as subclassing are decidable too.

We can easily get the decidability of subtyping from Lemmas 21, 23 and 25.

Corollary 26 Subtyping is decidable. It is decidable to check whether \( \Theta; \Delta \vdash \tau_1 \leq \tau_2 \) holds.

We will use the subtyping rules in Figure 8 and the algorithmic subtyping rules in Figure 23 interchangeably in the rest of the paper.

C.2.2 Minimal Typing

The expression typing rules are not syntax-directed because of the subsumption rule sub (see Figure ??). Similar to the transitivity rule for subtyping, sub has a “free” type \( \tau_1 \) that shows in the premises but not in the conclusion.

We design a set of algorithmic expression typing rules (shown in Figure 24). This new set of rules is syntax-directed, and we prove that it is sound and complete with respect to the original expression typing rules.

Lemma 27 Minimal typing is sound. If \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T \), then \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T \).

Proof: by induction on algorithmic expression typing rules.

Case a_var \( E = x, T = \Gamma(x) \). By var \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T \).

Case a_int, a_error, a_label and a_tag: trivial.

Case a_object \( E = C(e), T = C \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e : R(C) \). By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e : R(C) \).

Case a_c2r, a_object \( E = c2r(e), T = R(C) \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e : C \). By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e : C \).

Case a_c2r, a_object \( E = c2r(e), T = \text{ApproxR}(\alpha, \alpha \uparrow \Delta) \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \alpha \).

By induction hypothesis, \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \alpha \).

By Lemma 15, \( \Theta; \Delta \vdash \alpha \ll \alpha \uparrow \Delta \) and \( \alpha \uparrow \Delta \) is a concrete class name.

Case a_r2im \( E = r2im[I, C](e), T = \text{Imty}(I, C) \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \( \Theta; \Delta \vdash \tau \leq R(I, C) \).

By induction hypothesis, \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \tau \).

By sub \( \Theta; \Delta; \Sigma; \Gamma \vdash e : R(I, C) \).

By r2im \( \Theta; \Delta; \Sigma; \Gamma \vdash e : T \).

Case a_im2r \( E = \text{im}2r(e), T = R(I, \tau) \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \text{Imty}(I, \tau) \).

By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \text{Imty}(I, \tau) \).

By \text{im}2r \( \Theta; \Delta; \Sigma; \Gamma \vdash e : T \).

Case a_record \( E = \text{new}[\tau]\{l_1 = e_1, \ldots, l_n = e_n\}, T = \tau = \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \phi_i \).

By induction hypothesis, \( \Theta; \Delta; \Sigma; \Gamma \vdash e_i ; \tau_{mi} \).

By \text{new} \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \tau \).

By \phi \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \tau \).

By sub and \( \Theta; \Delta \vdash \tau_{mi} \leq \tau_i \), we have \( \Theta; \Delta; \Sigma; \Gamma \vdash e_i ; \tau_i \).

By record \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; T \).

Case a_field_E \( E = e_i, I, T = \tau_i, 1 \leq i \leq n \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \).

By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \phi_i \).

By field \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; T \).

Case a_field_R \( E = e_i, I, T = \tau_i, 1 \leq i \leq n \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \).

By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; \phi_i \).

By field \( \Theta; \Delta; \Sigma; \Gamma \vdash e ; T \).

Case a_assignR_E \( E = e_1, I, \tau_i = e_2 \) in \( e_3, T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 ; \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 ; \tau_{mi} \).

By induction hypothesis \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 ; \phi_i \).

By field \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 ; \tau_{mi} \).

By assignR \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 ; \{l_1^{\phi_1} : \tau_1, \ldots, l_n^{\phi_n} : \tau_n\} \).

Also by sub \( \Theta; \Delta \vdash \tau_{mi} \leq \tau_i \).

By assignR \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 ; \tau_i \).
Figure 24: Algorithmic Expression Typing Rules
Case a\_assignR\_R \quad E = e_1,l_i := e_2 \text{ in } e_3, \quad T = \tau \text{ with subderivations } \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \{l_1^{\phi_1} : \tau_1, \ldots, l_i^{\phi_i} : \tau_i, \ldots, l_n^{\phi_n} : \tau_n\}, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_m, \Delta; \Sigma; \Gamma \vdash l_i \equiv \tau_i, \text{ and } \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau.

By induction hypothesis, \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \{l_1^{\phi_1} : \tau_1, \ldots, l_i^{\phi_i} : \tau_i, \ldots, l_n^{\phi_n} : \tau_n\}, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_m \text{ and } \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau. \text{ By } \text{sub and } \Theta; \Delta; \Sigma; \Gamma \vdash \tau_m \leq \tau_i, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_i. \text{ By } \text{assignR } \Theta; \Delta; \Sigma; \Gamma \vdash E : T.$

Case a\_array \quad E = \text{new}[e_0, \ldots, e_{n-1}]^T \text{ and } T = \text{array}(\tau) \text{ with subderivations } \Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau_m, \text{ and } \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau. \text{ By induction hypothesis, } \Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau, \forall 0 \leq i \leq n - 1.

By induction hypothesis, \Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau_m, \forall 0 \leq i \leq n - 1. \text{ By sub and } \Theta; \Delta; \Sigma; \Gamma \vdash \tau_m \leq \tau_i, \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau. \text{ By } \text{array, } \Theta; \Delta; \Sigma; \Gamma \vdash E : T.$

Case a\_subscript \quad E = e_1[e_2], \text{ } T = \tau \text{ with subderivations } \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau) \text{ and } \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \text{int}. \text{ By } \text{subscript, } \Theta; \Delta; \Sigma; \Gamma \vdash E : T.$
Corollary 27 Inversion of Subtyping.

- If $\Theta; \Delta \vdash s \leq \{\ell_1^{\phi_1} : \tau_1 \ldots, \ell_n^{\phi_n} : \tau_n, \ldots\}$, then either $s = \{\ell_1^{\phi_1} : \tau_1 \ldots, \ell_n^{\phi_n} : \tau_n, \ldots\}$ or $s = \{\ell_1^{\phi_1} : \tau_1 \ldots, \ell_n^{\phi_n} : \tau_n, \ldots\}$. In either case $\Theta; \Delta \vdash s_1 \leq \tau_1 \forall \phi_1 = I$, and $s_2 = r_2 \forall \phi_2 = M$.
- If $\Theta; \Delta \vdash s \leq \{\ell_1^{\phi_1} : \tau_1 \ldots, \ell_n^{\phi_n} : \tau_n\}$, then $s = \{\ell_1^{\phi_1} : \tau_1 \ldots, \ell_n^{\phi_n} : \tau_n\}$.
- If $\Theta; \Delta \vdash s \leq \text{array}(\tau)$, then $s = \text{array}(\tau)$.
- If $\Theta; \Delta \vdash s \leq \text{int},$ then $s = \text{int}$.
- If $\Theta; \Delta \vdash S \leq \forall \alpha_1 \ll u_1, \ldots, \forall \alpha_m \ll u_m, (\tau_1 \ldots, \tau_n) \rightarrow \tau$, then (1) $S = \forall \alpha_1 \ll u_1', \ldots, \forall \alpha_m \ll u_m', (s_1 \ldots, s_n) \rightarrow s_2$, (2) $\Theta; \Delta \vdash \alpha_1 \ll \text{Topc}, \ldots, \forall \alpha_m \ll \text{Topc} \vdash \tau_1 \ll \tau_i \forall i \leq i \leq n$, (3) $\Theta; \Delta \vdash \alpha_1 \ll \text{Topc}, \ldots, \forall \alpha_m \ll \text{Topc} \vdash s \leq \tau$, and (4) if $\exists t_1, \ldots, t_m$ such that $\forall i \leq i \leq m, \Theta; \Delta \vdash t_i \ll u_i[1 \ldots, t_m/\alpha_1, \ldots, \alpha_m]$, then $\forall i \leq i \leq m, \Theta; \Delta \vdash t_i \ll u_i'[1 \ldots, t_m/\alpha_1, \ldots, \alpha_m]$.}

Proof: by inspection of algorithmic subtyping rules. Case $\forall$, part 4: $\Theta; \Delta; \alpha_1 \ll \text{Topc}, \ldots, \forall \alpha_m \ll \text{Topc} \vdash u_i \ll u_i'$ $\forall i \leq i \leq m$. Let $\sigma_i = t_1, \ldots, t_i/\alpha_1, \ldots, \alpha_i$, $\forall i \leq i \leq m$. By substitution Lemma 11 $\Theta; \Delta \vdash u_i[\sigma_1 \ldots, \sigma_m] 

Definition 29 If $\text{domain}(\Gamma') = \text{domain}(\Gamma)$, and $\Gamma'(x) = \Gamma(x) \forall x : M \Gamma(x) \in \Gamma$, and $\Theta; \Delta \vdash \Gamma'(y) \leq \Gamma(y) \forall y : \tau \in \Gamma$, then $\Theta; \Delta \vdash \Gamma' \leq \Gamma$.

We define $\Theta; \Delta \vdash \Delta' \ll \Delta$ if: (1) $\Delta = P_1(\alpha_1), \ldots, P_n(\alpha_n)$, $\Delta' = P'_1(\alpha_1), \ldots, P'_n(\alpha_n)$ (2) $\forall i \leq i \leq n$, either $P_i(\alpha_i) = \alpha_i \ll u_i, P'_i(\alpha_i) = \alpha_i \ll u'_i$ and $\Theta; \Delta_i \vdash u_i \ll u_i$, or $P_i(\alpha_i) = \alpha_i \ll u_i, P'_i(\alpha_i) = \alpha_i \ll u_i$, and $\Theta; \Delta_i \vdash u_i \ll u_i'$, or $P_i(\alpha_i) = \alpha_i \ll \text{Topc}$. $\Delta_i = P_i(\alpha_i), P'_i(\alpha_i)$.

Lemma 30 1. If $\Theta; \Delta \vdash \tau : \kappa$ and $\Theta; \Delta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau : \kappa$.

2. If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$ and $\Theta; \Delta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau_1 \ll \tau_2$.

3. If $\Theta; \Delta \vdash \tau_1 \ll \tau_2$ and $\Theta; \Delta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \tau_1 \ll \tau_2$.

4. If $\alpha \in \text{domain}(\Delta)$ and $\Theta; \Delta \vdash \Delta' \ll \Delta$, then $\Theta; \Delta' \vdash \alpha \ll \Delta' \ll \alpha \ll \Delta$.

Proof: prove (1), (2) and (3) by induction on kinding, subclassing and subtyping rules respectively. (4) by the first part of Lemma 15 $\Theta; \Delta \vdash \alpha \ll \Delta$. By (2) $\Theta; \Delta' \vdash \Delta' \ll \alpha \ll \Delta$. By the second part of Lemma 15, $\Theta; \Delta' \vdash \alpha \ll \Delta$.

Lemma 31 Narrowing of Environments.

If $\Theta; \Delta; \Sigma; \Gamma \vdash E : T$, $\Theta; \Delta' \vdash \Gamma \leq \Gamma' \leq T$, then $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$ and $\Theta; \Delta' \vdash T' \leq T$. 

27
Proof: by induction on algorithmic typing rules.

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

Case **a_int, a_error, a_label** and **a_tag**: trivial.

Case **a_object**: $E = C(e)$, $T = C$ with subderivation $\Theta; \Delta'; \Sigma; \Gamma \vdash e : R(C)$. By induction hypothesis, $\Theta; \Delta'; \Sigma; \Gamma' \vdash e : \tau'$ and $\Theta'; \Delta' \vdash \tau' \leq R(C)$. By inversion of subtyping Lemma 28 $\tau' = R(C)$. Let $T' = T$. By reflexivity of subtyping $\Theta'; \Delta' \vdash T' \leq T$. By **a_object** $\Theta'; \Delta'; \Sigma; \Gamma' \vdash e : T'$.

Case **a_c2r-c**: $E = c2r(e)$, $T = R(C)$ with subderivation $\Theta; \Delta'; \Sigma; \Gamma \vdash e : C$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash e : \tau'$ and $\Theta'; \Delta' \vdash \tau' \leq C$. By inversion of subtyping Lemma 28, $\tau' = C$. Let $T' = T$. By reflexivity of subtyping $\Theta'; \Delta' \vdash T' \leq T$. By **a_c2r-c** $\Theta'; \Delta'; \Sigma; \Gamma' \vdash e : T'$.

Case **a_c2r-tv**: $E = c2r(e)$, $T = \text{ApproXR}(\alpha, \alpha \uparrow \Delta)$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e : \alpha$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash e : \tau'$ and $\Theta'; \Delta' \vdash \tau' \leq \alpha$. By Lemma 28, $\tau' = \alpha$. Let $T' = \text{ApproXR}(\alpha, \alpha \uparrow \Delta)$. By Lemma 30 $\Theta; \Delta' \vdash \alpha \uparrow \Delta \leq \alpha \uparrow \Delta$. By properties of approximation Lemma 19, $\Theta; \Delta' \vdash T' \leq T$. By **a_c2r-tv** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.

Case **a_r2im**: $E = r2im[I, C|(e), T = \text{Imty}(I, C)$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau$ and $\Theta; \Delta' \vdash \tau \leq R_I(I, C)$. By transitivity of subtyping, $\Theta; \Delta' \vdash T' \leq R_I(I, C)$. Let $T' = T$. By reflexivity of subtyping, $\Theta; \Delta' \vdash T' \leq T$. By **a_r2im** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.

Case **a_im2r**: $E = im2r(e), T = R_I(I, \tau)$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \text{Imty}(I, \tau)$. By induction of subtyping Lemma 28, $\tau' = \text{Imty}(I, \tau)$. Let $T' = T$. By reflexivity of subtyping, $\Theta; \Delta' \vdash T' \leq T$. By **a_im2r** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.

Case **a_record**: $E = \text{new}[\tau][\{i_1 = e_1, \ldots, i_n = e_n\}], T = \tau = \{\{l_{i_1}^0 : \tau_1, \ldots, l_{i_n}^0 : \tau_n\} \mid 1 \leq i \leq n\}$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash i_1 : \tau_{j_1}, \ldots, i_n : \tau_{j_n}$ and $\Theta; \Delta' \vdash \tau_{j_i} \leq \tau_i \forall 1 \leq i \leq n$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash e_i : \tau_{j_i}' \forall 1 \leq i \leq n$. By Lemma 30 and $\Theta; \Delta' \vdash \tau_{j_i}' \leq \tau_{j_i} \forall 1 \leq i \leq n$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau' \leq \{\{l_{i_1}^0 : \tau_1, \ldots, l_{i_n}^0 : \tau_n\} \mid 1 \leq i \leq n\}$. Let $T' = T$. By **a_record** $\Theta; \Delta'; \Sigma; \Gamma' \vdash \tau' \leq T'$. By reflexivity of subtyping $\Theta; \Delta' \vdash T' \leq T$. By **a_record** $\Theta; \Delta'; \Sigma; \Gamma' \vdash \tau' \leq T'$.

Case **a_field_E**: $E = e.i_i$, $T = \tau_i$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \{\{l_{i_1}^0 : \tau_1, \ldots, l_{i_n}^0 : \tau_n\} \mid 1 \leq i \leq n\}$. Let $T' = T$, then by reflexivity of subtyping $\Theta; \Delta' \vdash T' \leq T$. By **a_field_E** $\Theta; \Delta'; \Sigma; \Gamma' \vdash \tau_i \leq T'$.

Case **a_field_R**: $E = e.i_i$, $T = \tau_i$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \{\{l_{i_1}^0 : \tau_1, \ldots, l_{i_n}^0 : \tau_n\} \mid 1 \leq i \leq n\}$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash i_i : \tau_i \forall 1 \leq i \leq n$. By transitivity of subtyping $\Theta; \Delta' \vdash T' \leq \{\{l_{i_1}^0 : \tau_1, \ldots, l_{i_n}^0 : \tau_n\} \mid 1 \leq i \leq n\}$. In both cases $\Theta; \Delta' \vdash T' \leq T$. In case (1) by **a_field_R** $\Theta; \Delta'; \Sigma; \Gamma' \vdash \tau_i \leq T'$. In case (2) by **a_field_E** $\Theta; \Delta'; \Sigma; \Gamma' \vdash \tau_i \leq T'$.

Case **a_assignR_E**: $E = e_i.i_i := e_2$ in e_3, $T = \tau$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \{\{l_{i_1}^0 : \tau_1\} \mid 1 \leq i \leq n\}, \phi_i = M, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_i}, \Theta; \Delta' \vdash \tau_{m_i} \leq \tau_i$ and $\Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : s, \Theta; \Delta' \vdash s \leq \{\{l_{i_1}^0 : \tau_1\} \mid 1 \leq i \leq n\}, \Theta; \Delta'; \Sigma; \Gamma' \vdash e_2 : \tau_{m_i}, \Theta; \Delta' \vdash T' \leq \tau_i$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau' \leq \tau_i$. Let $T' = \tau'$, then $\Theta; \Delta' \vdash T' \leq T$. By **a_assignR_E** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$. In case (2) by **a_assignR_E** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.

Case **a_assignR_R**: $E = e_i.i_i := e_2$ in e_3, $T = \tau$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \{\{l_{i_1}^0 : \tau_1\} \mid 1 \leq i \leq n\}, \phi_i = M, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_i}, \Theta; \Delta' \vdash \tau_{m_i} \leq \tau_i$ and $\Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau$. By induction hypothesis $\Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : s, \Theta; \Delta' \vdash s \leq \{\{l_{i_1}^0 : \tau_1\} \mid 1 \leq i \leq n\}, \Theta; \Delta'; \Sigma; \Gamma' \vdash e_2 : \tau_{m_i}, \Theta; \Delta' \vdash T' \leq \tau_i$. By transitivity of subtyping $\Theta; \Delta' \vdash \tau' \leq \tau_i$. Let $T' = \tau'$, then $\Theta; \Delta' \vdash T' \leq T$. In case (1) by **a_assignR_R** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$. In case (2) by **a_assignR_R** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.

Case **a_array**: $E = \text{new}[e_0, \ldots, e_{n-1}], T = \text{array}(\tau)$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau_{m_i}$ and $\Theta; \Delta \vdash \tau_{m_i} \leq \tau \forall 0 \leq i \leq n - 1$. By properties of approximation Lemma 19, $\Theta; \Delta' \vdash T' \leq T$. In case (1) by **a_array** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$. In case (2) by **a_array** $\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'$.
By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_{m_i}$ and $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau_{m_j}$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_i} \leq \tau_{m_j} \forall 0 \leq i \leq n - 1$. By Lemma 30 $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_i} \leq \tau \forall 0 \leq i \leq n - 1$. By transitivity of subtyping $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_i} \leq \tau \forall 0 \leq i \leq n - 1$. Let $T' = T$. By reflexivity of subtyping $\Theta;\Delta;\Sigma;\Gamma \vdash T' \leq T$. By a\textit{assign} $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau \wedge \Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau$.

**Case a\textit{subscript}** $E = e_1[e_2]$, $T = \tau$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau(\sigma)$ and $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \int$.

By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_1'$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau(\sigma)$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau$. By inversion of subtyping Lemma 28 $\tau_1' \equiv \tau_2'$ and $\tau_2' \equiv \tau$. By reflexivity of subtyping $\Theta;\Delta;\Sigma;\Gamma \vdash T' \leq T$. By a\textit{subscript} $\Theta;\Delta;\Sigma;\Gamma \vdash T' : E$.

**Case a\textit{assign}** $E = e_1[e_2] := e_3$ in $e_4$, $T = \tau'$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau(\sigma)$, $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \int$, and $\Theta;\Delta;\Sigma;\Gamma \vdash e_3 : \tau' \leq \tau$ and $\Theta;\Delta;\Sigma;\Gamma \vdash e_4 : \tau'$.

By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_1'$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau_1$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau_2'$. By Lemma 30 $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau$.

**Case a\textit{let}** $E = x : \tau = e_1$ in $e_2$, $T = \tau_2$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_1$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau$ and $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \int$.

By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_1'$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau_1$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_2' \leq \tau_2$. By Lemma 30 $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau_2'$. Let $T' = \tau_2'$. By reflexivity of subtyping $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' \leq \tau_2'$. By a\textit{let} $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1' : T'$. 

**Case a\textit{assign}** $E = x := e_1$ in $e_2$, $T = \tau$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_{m_i}$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_1} \leq \Gamma(x)$, $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau$.

By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_{m_1}'$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_1}' \leq \tau_{m_1}$, $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau'$ and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau' \leq \tau$. By the definition of $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_1}' \leq \Gamma(x)$. Let $T' = \tau'$, then $\Theta;\Delta;\Sigma;\Gamma \vdash T' \leq T$. By Lemma 30 $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_1} \leq \Gamma(x)$. By transitivity of subtyping $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_{m_1}' \leq \tau_{m_1}$.

**Case a\textit{call}** $E = e(t_1, \ldots, t_n)$, $T = \tau|$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e : \Gamma \in E$.

By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e : \Gamma$. By Lemma 11 $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau_2$. By the definition of $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau_2$. Let $T' = \tau_1$. By reflexivity of subtyping, $\Theta;\Delta;\Sigma;\Gamma \vdash T' \leq T$. By a\textit{call} $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 : T'$. 

**Case a\textit{ifpack}** $E = \operatorname{pack} \tau as \alpha \ll \tau \in \Gamma (e : \tau')$, $T = \exists \alpha \ll \tau_1 \wedge \tau' \wedge \tau$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_1$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau_2$ and $\alpha \not\in \text{domain}(\Delta)$. By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e : \tau_1$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau_2$ and $\alpha \not\in \text{domain}(\Delta)$. By Lemma 30 and $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \tau_2$. Let $T' = \tau_2$. By reflexivity of subtyping, $\Theta;\Delta;\Sigma;\Gamma \vdash T' \leq T$. 

**Case a\textit{open}** $E = (a, x) = \operatorname{open}(e_1)$ in $e_2$, $T = (\tau_m)^{\alpha}_{\Delta, \alpha \ll \tau_1}$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \exists \beta \ll \tau_1$, $\Theta;\Delta;\alpha \ll \tau_1$, $\Theta;\Delta;\Sigma;\Gamma \vdash e_2 : \tau_m (\alpha \not\in \text{domain}(\Delta))$. By induction hypothesis $\Theta;\Delta;\Sigma;\Gamma \vdash e_1 : \tau_1$, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau_1 \leq \exists \beta \ll \tau_1$. By inversion of subtyping Lemma 28, (1) $\tau'_1 = \exists \beta \ll \tau_1$, (2) $\Theta;\Delta;\Sigma;\Gamma \vdash \tau'_1 \ll \tau_1$, and (3) $\Theta;\Delta;\Sigma;\Gamma \vdash \tau'_1 \not\ll \tau_1$. By alpha-equivalence and $\alpha \not\in \text{domain}(\Delta)$, $\Theta;\Delta;\alpha \ll \tau_1 \wedge \alpha \not\in \text{domain}(\Delta)$. By Lemma 30 and $\Theta;\Delta;\Sigma;\Gamma \vdash \Delta;\alpha \ll \tau'_1 \ll \tau_1 \not\ll \alpha$. By definition, $\Theta;\Delta;\alpha \ll \tau'_1 \ll \tau_1$. Let $T' = (\tau'_1)^{\alpha}_{\Delta, \alpha \ll \tau_1'}$. By a\textit{open} $\Theta;\Delta;\Sigma;\Gamma \vdash e : E$.

By properties of type lifting Lemma 18, $\Theta;\Delta;\alpha \ll \tau_1 \wedge \tau_2 \leq T$. By Lemma 30 $\Theta;\Delta;\alpha \ll \tau'_1 \wedge \tau_2 \leq T$. By the transitivity of subtyping and $\Theta;\Delta;\alpha \ll \tau'_1 \wedge \tau_2 \leq T$. By Lemma 18, $\Theta;\Delta;\Sigma;\Gamma \vdash \tau'_1 \leq \tau_2$ because $\alpha \not\in \text{free}(T)$.
\(\Theta; \Delta \vdash \tau_2 \leq \tau.\)

By induction hypothesis \(\Theta; \Delta'; \Sigma; \Gamma' \vdash e : s, \Theta; \Delta' \vdash s \leq \text{Tag}(\tau'), \Theta; \Delta', \alpha \gg \tau'; \Sigma; \Gamma', x : \text{Tag}(\alpha) = e_1 : \tau'_1, \Theta; \Delta', \alpha \gg \tau' \vdash \tau_1' \leq \tau_1, \Theta; \Delta'; \Sigma; \Gamma' \vdash e_2 : \tau'_2 \) and \(\Theta; \Delta' \vdash \tau'_2 \leq \tau_2.\) By inversion of subtyping Lemma 28 \(s = \text{Tag}(\tau').\) By Lemma 30, \(\Theta; \Delta', \alpha \gg \tau' \vdash \tau_1' \leq \tau \) and \(\Theta; \Delta' \vdash \tau'_2 \leq \tau.\) By transitivity of subtyping, \(\Theta; \Delta', \alpha \gg \tau' \vdash \tau_1' \leq \tau \) and \(\Theta; \Delta' \vdash \tau'_2 \leq \tau.\) Let \(T' = T.\) By \textbf{a_ifParent} \(\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'.\) By reflexivity of subtyping, \(\Theta; \Delta' \vdash T' \leq T.\)

**Case a_ifTag_eq** \(E = \text{ifEqTag}''(e_1, e_2)\) then \(e_1\) else \(e_2, T = \tau\) with subderivations \(\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{Tag}(C_1), \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \text{Tag}(C_2), \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_1 \) and \(\Theta; \Delta \vdash \tau_1 \leq \tau_2 \) such that \(C_1 \neq C_2.\)

By induction hypothesis \(\Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : \tau'_1, \Theta; \Delta' \vdash \tau'_1 \leq \text{Tag}(C_1), \Theta; \Delta'; \Sigma; \Gamma' \vdash e_2 : \tau'_2, \Theta; \Delta' \vdash \tau'_2 \leq \text{Tag}(C_2), \Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : \tau_1 \) and \(\Theta; \Delta' \vdash \tau_2 \leq \tau_1.\) By inversion of subtyping Lemma 28, \(\tau'_1 = \text{Tag}(C_1)\) and \(\tau'_2 = \text{Tag}(C_2).\) Let \(T = T,\) then by reflexivity of subtyping \(\Theta; \Delta' \vdash T' \leq T.\) By Lemma 30 \(\Theta; \Delta' \vdash \tau_1 \leq \tau.\) By transitivity of subtyping \(\Theta; \Delta' \vdash \tau'_2 \leq \tau.\) By \textbf{a_ifTag_eq} \(\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'.\)

**Case a_ifTag_neq** \(E = \text{ifEqTag}''(e_1, e_2)\) then \(e_1\) else \(e_2, T = \tau\) with subderivations \(\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{Tag}(\gamma), \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \text{Tag}(\tau_2), \Theta; \Delta; \Sigma; \Gamma \vdash \tau_1 \leq \tau_2 \) such that \(\gamma = \tau_1, \Theta; \Delta \vdash \tau_2 \leq \tau(C_1 \neq C_2).\)

By induction hypothesis \(\Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : \tau'_1, \Theta; \Delta' \vdash \tau'_1 \leq \text{Tag}(\gamma), \Theta; \Delta'; \Sigma; \Gamma' \vdash e_2 : \tau'_2, \Theta; \Delta' \vdash \tau'_2 \leq \text{Tag}(\tau_2), \Theta; \Delta'; \Sigma; \Gamma' \vdash e_1 : \tau_1 \) and \(\Theta; \Delta' \vdash \tau_2 \leq \tau_1.\) By inversion of subtyping Lemma 28, \(\tau'_1 = \text{Tag}(\gamma)\) and \(\tau'_2 = \text{Tag}(\tau_2).\) Let \(T = T,\) then by reflexivity of subtyping \(\Theta; \Delta' \vdash T' \leq T.\) By Lemma 30 \(\Theta; \Delta' \vdash \tau_1 \leq \tau_1.\) By transitivity of subtyping \(\Theta; \Delta' \vdash \tau'_2 \leq \tau_1.\) By the definition of \(\Theta; \Delta' \vdash \tau'_1 \leq \tau_1, \Theta; \Delta' \vdash \tau'_2 \leq \tau_2.\) By Lemma 30 \(\Theta; \Delta' \vdash \tau'_1 \leq \tau_1.\) By \textbf{a_ifTag_neq} \(\Theta; \Delta'; \Sigma; \Gamma' \vdash E : T'.\)

**Lemma 32** Minimal typing is complete. If \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T,\) then \(\exists T_m \text{ such that } \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \and \Theta; \Delta \vdash T_m \leq T.\)

Proof: by induction on the expression typing rules.

**Case var** \(E = x, T = \Gamma(x)\)

Let \(T_m = T.\) By \textbf{a_var} \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T_m,\) and by reflexivity of subtyping \(\Theta; \Delta \vdash T_m \leq T.\)

**Case int, error, label and tag:** trivial.

**Case object** \(E = C(e), T = C \) with subderivation \(\Theta; \Delta; \Sigma; \Gamma \vdash e : R(C).\)

By induction hypothesis \(\exists \tau_m \text{ such that } \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_m.\) By inversion of subtyping Lemma 28, \(\tau_m = R(C).\) Let \(T_m = T.\) By \textbf{a_object} \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T_m,\) by reflexivity of subtyping \(\Theta; \Delta \vdash T_m \leq T.\)

**Case c2r_c** \(E = c2r(e), T = R(C) \) with subderivation \(\Theta; \Delta; \Sigma; \Gamma \vdash e : C.\)

By induction hypothesis \(\exists \tau_m \text{ such that } \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_m \) and \(\Theta; \Delta \vdash \tau_m \leq C.\) By Lemma 28, \(\tau_m = C.\) Let \(T_m = T.\) By \textbf{a_c2r_c} \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T_m,\) by reflexivity of subtyping, \(\Theta; \Delta \vdash T_m \leq T.\)

**Case c2r_tv** \(E = c2r(e), T = \text{ApproxR}(\alpha, C) \) with subderivation \(\Theta; \Delta; \Sigma; \Gamma \vdash e : \alpha \and \Theta; \Delta \vdash \alpha \leq C.\)

By induction hypothesis \(\exists \tau_m \text{ such that } \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_m \) and \(\Theta; \Delta \vdash \tau_m \leq \alpha.\) By Lemma 28, \(\tau_m = \alpha.\) By Lemma 15, \(\Theta; \Delta \vdash \alpha \vdash C.\) Let \(T_m = \text{ApproxR}(\alpha, \alpha \vdash \Delta).\) By Lemma 19, \(\Theta; \Delta \vdash T_m \leq T.\) By \textbf{a_c2r_tv} \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T_m.\)

**Case r2im** \(E = r2im[I, C](e), T = \text{Imty}(I, C) \) with subderivations \(\Theta; \Delta; \Sigma; \Gamma \vdash e : R_I(I, C).\)

By induction hypothesis \(\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \(\Theta; \Delta \vdash \tau \leq R_I(I, C).\) Let \(T_m = \tau,\) then \(\Theta; \Delta \vdash T_m \leq T.\)

**Case im2r** \(E = \text{im2r}(e), T = R_I(I, \tau) \) with subderivation \(\Theta; \Delta; \Sigma; \Gamma \vdash e : \text{Imty}(I, \tau).\)

By induction hypothesis \(\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \(\Theta; \Delta \vdash \tau \leq \text{Imty}(I, \tau).\) By inversion of subtyping Lemma 28 \(\tau' = \text{Imty}(I, \tau).\) Let \(T_m = T.\) By reflexivity of subtyping \(\Theta; \Delta \vdash T_m \leq T.\) By \textbf{a_im2r} \(\Theta; \Delta; \Sigma; \Gamma \vdash E : T_m.\)

**Case record** \(E = \text{new}[\tau](l_1 = e_1, \ldots, l_n = e_n), T = \tau = \{l_1^{\Theta_1 : \tau_1}, \ldots, l_n^{\Theta_n : \tau_n}\} \) with subderivations \(\Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau_i, \forall 1 \leq i \leq n.\)
By induction hypothesis, \( \exists m_1, \ldots, m_n \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau_{m_i} \) and \( \Theta; \Delta \vdash \tau_{m_i} \leq \tau_i, \forall 1 \leq i \leq n \). Let \( T_m = T \). By \textbf{a_record}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). By reflexivity of subtyping \( \Theta; \Delta \vdash T_m \leq T \).

**Case field** \( E = e_1 . l_1, \ T = \tau_1 \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \{ l_1^{\Theta} : \tau_1, \ldots , l_n^{\Theta} : \tau_n \} \) \((1 \leq i \leq n)\).

By induction hypothesis, \( \exists m_1, m_2, m_3 \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_{m_1} \) and \( \Theta; \Delta \vdash \tau_{m_1} \leq \tau_{m_2} \leq \tau_{m_3} \). By Lemma 28, either (1) \( \tau_{m_1} = \{ l_1^{\Theta} : \tau_1, \ldots , l_n^{\Theta} : \tau_n \} \) or (2) \( \tau_{m_2} = \{ l_1^{\Theta} : \tau_1, \ldots , l_n^{\Theta} : \tau_n \} \).

In either case \( \Theta; \Delta \vdash \tau_{m_j} \leq \tau_i, \forall 1 \leq j \leq n \). In particular, \( \Theta; \Delta \vdash \tau_{m_3} \leq \tau_i \). Let \( T_m = \tau_{m_3} \), then we have \( \Theta; \Delta \vdash T_m \leq T \).

In case (1), by \textbf{a_field R}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). In case (2) by \textbf{a_field E}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

**Case assignR** \( E = e_1 . l_1 \), \( T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \{ l_1^{\Theta} : \tau_1, \ldots , l_n^{\Theta} : \tau_n \}, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_1 \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau \).

By induction hypothesis, \( \exists m_1, m_2, m_3 \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \). Let \( T_m = \tau_{m_3} \), then \( \Theta; \Delta \vdash T_m \leq T \). By Lemma 28, either (1) \( \tau_{m_1} = \{ l_1^{\Theta} : \tau_1^1, \ldots , l_n^{\Theta} : \tau_n^1 \} \) or (2) \( \tau_{m_2} = \{ l_1^{\Theta} : \tau_1^2, \ldots , l_n^{\Theta} : \tau_n^2 \} \).

In case (1), by \textbf{a_assignR E}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). In case (2), by \textbf{a_assignR R}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

**Case let** \( E = e : \tau \) in \( e_2, \ T = \tau' \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau' \).

By induction hypothesis, \( \exists m_1, m_2 \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \), and \( \Theta; \Delta \vdash \tau_{m_1} \leq \tau _{m_2} \). Let \( T_m = \tau_{m_3} \), then \( \Theta; \Delta \vdash T_m \leq T \). By \textbf{a let}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

**Case assign** \( E = x := e_1 \) in \( e_2, \ T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau \) and \( \Theta; \Delta \vdash \tau_{m_1} \leq \tau _{m_2} \). Let \( T_m = \tau_{m_2} \), then \( \Theta; \Delta \vdash T_m \leq T \). By \textbf{a assign}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

**Case array** \( E = \text{new}[e_0, \ldots , e_{n-1}]^T \) and \( T = \text{array}(\tau) \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_i : \tau \) \( \forall 0 \leq i \leq n - 1 \). Let \( T_m = T \). By \textbf{a array}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). By reflexivity of subtyping, \( \Theta; \Delta \vdash T_m \leq T \).

**Case subscript** \( E = e_1[e_2], \ T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau) \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \text{int} \). By Lemma 28, \( \tau_{m_1} = \text{array}(\tau) \), and \( \tau_{m_2} = \text{int} \). Let \( T_m = T \). By \textbf{a subscript}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). By reflexivity of subtyping, \( \Theta; \Delta \vdash T_m \leq T \).

**Case assignA** \( E = e_1[e_2] \), \( T = \tau' \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau) \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \text{int} \). By induction hypothesis, \( \exists m_1, m_2, m_3, m_4 \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \), \( \Theta; \Delta \vdash \tau_{m_1} \leq \text{array}(\tau) \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \), \( \Theta; \Delta \vdash \tau_{m_2} \leq \text{int} \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_3 : \tau_{m_3} \), \( \Theta; \Delta \vdash \tau_{m_3} \leq \text{tau} \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_4 : \tau_{m_4} \), \( \Theta; \Delta \vdash \tau_{m_4} \leq \tau' \). By Lemma 28, \( \tau_{m_1} = \text{array}(\tau) \), and \( \tau_{m_2} = \text{int} \). Let \( T_m = \tau_{m_4} \), then \( \Theta; \Delta \vdash T_m \leq T \). By \textbf{a assignA}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

**Case call** \( E = \text{call}(e_1[t_1, \ldots , t_m], e_1, e_n), \ T = \text{tau}[\sigma] \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \text{array}(\tau), \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \text{tau}[\sigma], \forall 1 \leq i \leq n \). By \textbf{a call}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

By induction hypothesis, \( \exists m_1, m_2, m_3, m_4 \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_{m_1} \), \( \Theta; \Delta \vdash \tau_{m_1} \leq \text{array}(\tau) \). Let \( T_m = \tau_{m_4} \), then \( \Theta; \Delta \vdash T_m \leq T \). By \textbf{a call}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \).

By induction hypothesis, \( \forall \alpha \in \text{domain}(\Delta) \).

**Case pack** \( E = \text{pack}(\tau, \alpha) \), \( T = \exists \alpha \in \tau, \tau' \) with subderivation \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau'[\sigma/\alpha] \). By induction hypothesis, \( \forall \alpha \in \text{domain}(\Delta) \).

**Case let** \( E = \text{let} \), \( T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) and \( \Theta; \Delta \vdash \tau \leq \tau_\alpha \). Let \( T_m = T \). By \textbf{a let}, \( \Theta; \Delta; \Sigma; \Gamma \vdash E : T_m \). By reflexivity of subtyping, \( \Theta; \Delta \vdash T_m \leq T \).
Case open \( E = (\alpha, x) = \text{open}(e_1) \) in \( e_2, T = \tau' \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \exists \beta \ll \tau_u, \tau \) and \( \Theta; \Delta, \alpha \ll \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_2 : \tau', \) where \( \alpha \notin \text{domain}(\Delta) \) and \( \alpha \notin \text{free}(\tau') \).

By induction hypothesis, \( \exists \tau_{m_1} \land \tau_{m_2} \) similar to \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \) and \( \Theta; \Delta \vdash \tau_{m_1} \ll \exists \beta \ll \tau_u, \tau \).

By Lemma 28, \( \tau_{m_1} \ll \exists \beta \ll \tau_{u}, s, \Theta; \Delta \vdash s \ll \tau_u \) and \( \Theta; \Delta, \beta \ll \text{Topc} \vdash \exists s \ll \tau_u \). By Lemma 30 and \( \Theta; \Delta, \alpha \ll \text{Topc} \vdash \exists \beta \ll \tau_u \). Therefore, by definition of type environment typing, \( \Theta; \Delta, \alpha \ll \exists s_u \vdash (\Gamma, x : \tau[\alpha/\beta], \Delta) \ll \text{Topc} \ll \exists s \ll \tau_u \). By narrowing of environments Lemma 31, \( \exists \tau'_{m_2} \) such that \( \Theta; \Delta, \alpha \ll \exists s_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_2 : \tau'_{m_2} \) and \( \Theta; \Delta, \alpha \ll \exists s_u \vdash \tau'_{m_2} \ll \tau_{m_2} \). Let \( T_m = (\tau'_{m_2}) \#_{\text{au}} \Theta; \Delta, \alpha \ll \exists s_u \vdash T_{m_2} \). By transitivity of subtyping \( \Theta; \Delta, \alpha \ll \exists s_u \vdash \tau'_{m_2} \ll \tau_{m_2} \). By properties of type lifting Lemma 18 \( \Theta; \Delta \vdash T_m \ll T \) because \( \alpha \notin \text{free}(T) \).

Case pack and Case open are similar to Case pack and Case open.

Case ifParent \( E = \text{ifParent}(e) \) then bind \((\alpha, x)\) in \( e_1 \) else \( e_2, T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash \exists \beta \ll \tau_{u}, \tau \) and \( \Theta; \Delta, \alpha \ll \tau_u; \Sigma; \Gamma, x : \tau[\alpha/\beta] \vdash e_1 : \tau \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau \).

Let \( \Delta' = \Delta, \alpha \gg \tau' \) and \( \exists \tau_{m_1} \land \tau_{m_2} \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau', \Theta; \Delta; \Sigma; \Gamma, x : \text{Tag}(\alpha) \vdash e_1 : \tau_{m_1} \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \).

By induction hypothesis, \( \exists \tau_{m_1}, \tau_{m_2} \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \), and \( \Theta; \Delta \vdash \tau_{m_1} \ll \text{Tag}(\alpha), \Theta; \Delta \vdash \tau_{m_2} \ll \text{Tag}(\alpha), \Theta; \Delta \vdash \tau_{m_1} \ll \tau_{m_2} \). By transitivity of subtyping \( \Theta; \Delta, \alpha \ll \exists s_u \vdash \tau'_{m_2} \ll \tau_{m_2} \).

Case ifTag.eq \( E = \text{ifEqTag}(e_1, e_2) \) then \( e_1 \) else \( e_2, T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash \exists \beta \ll \tau_{u}, \tau \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{C_1}, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{C_2} \).

By induction hypothesis, \( \exists \tau_{m_1}, \tau_{m_2} \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \), and \( \Theta; \Delta \vdash \tau_{m_1} \ll \text{Tag}(C_1), \Theta; \Delta \vdash \tau_{m_2} \ll \text{Tag}(C_2) \).

Let \( T_m = T \), then by reflexivity of subtyping \( \Theta; \Delta \vdash T_m \ll T \).

Case ifTag.neq \( E = \text{ifEqTag}(e_1, e_2) \) then \( e_1 \) else \( e_2, T = \tau \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash \exists \beta \ll \tau_{u}, \tau \) and \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{C_1}, \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{C_2} \).

By induction hypothesis, \( \exists \tau_{m_1}, \tau_{m_2} \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau_{m_1} \), \( \Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau_{m_2} \), and \( \Theta; \Delta \vdash \tau_{m_1} \ll \text{Tag}(C_1), \Theta; \Delta \vdash \tau_{m_2} \ll \text{Tag}(C_2) \).

Let \( T_m = T \), then by reflexivity of subtyping \( \Theta; \Delta \vdash T_m \ll T \).

Case sub \( E = e, T = \tau_2 \) with subderivations \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_1 \) and \( \Theta; \Delta \vdash \tau_1 \ll \tau_2 \).

By induction hypothesis, \( \exists \tau_{m_1}, \tau_{m_2} \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_{m_1} \) and \( \Theta; \Delta \vdash \tau_{m_1} \ll \tau_{m_2} \). Let \( T_m = \tau_{m_2} \), then \( \Theta; \Delta; \Sigma; \Gamma \vdash T_m \ll T \).

We define the size of expressions in Figure 25.

Lemma 34 Minimal Typing terminates.

Proof: in each inference rule, the premises refer to only subexpressions of the expression in the conclusion. The size of the expression in the conclusion is always larger than the sum of the subexpression sizes in the premises. Also the side conditions are decidable.

Theorem 35 Type checking of LLL<sub>C1</sub> is decidable.

Proof: To decide whether \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau \) holds, we can first get the minimal type \( \tau_m \) of \( e \) such that \( \Theta; \Delta; \Sigma; \Gamma \vdash e : \tau_m \), then test whether \( \Theta; \Delta \vdash \tau_m \ll \tau \). Because both minimal typing and subtyping are decidable, type checking is decidable.
Lemma 33. Permutation of substitution. If $\delta = s/\eta$ and $s$ has no free type variables, then:

1. $\tau[\tau'/\alpha][\delta] = \tau[\delta][\tau'/\alpha]$.
2. $\tau[\eta/\tau_\alpha][\delta] = \tau[\delta][\eta/(\tau_\alpha/\delta)]$, if $\gamma \neq \eta$ and $\tau_\alpha$ is a class name or a type variable other than $\gamma$.

Lemma 37. It preserves typing to substitute an appropriate type for the first type variable in the kind environment. If $\Theta; \Delta; \Sigma; \Gamma \vdash e : T$, $\Delta = \eta \ll \tau, \Delta'$ and $\Theta; \bullet \vdash s \ll \tau$ (or $\Delta = \eta \gg \tau, \Delta'$ and $\Theta; \bullet \vdash \tau \ll s$), and $\delta = s/\eta$, then $\Theta; \Delta' / \delta; \Sigma / \delta; \Gamma / \delta \vdash E[\delta] : T[\delta]$.

Proof: by induction on expression typing rules.

Case $\text{var\ } e = x$, $T = \Gamma(x)$.

By $\text{var\ } \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash x : (\Gamma[\delta])(x)$. Because $(\Gamma[\delta])(x) = (\Gamma(x))[\delta]$ and $E[\delta] = x$, we have $\Theta; \Delta'[\delta]; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.

Case $\text{int\ } e$, $T = C$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : R(C)$.

By induction hypothesis $\Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash e[\delta] : R(C)[\delta]$. $R(C)[\delta] = R(C)$ because $R(C)$ contains no free type variables. By $\text{object\ } \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.

Case $\text{c2r}_{\alpha\cdot} E = c2r(e)$, $T = R(C)$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : C$.

By induction hypothesis, $\Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash e[\delta] : C[\delta]$. $C[\delta] = C$ and $R(C)[\delta] = R(C)$. By $\text{c2r}_{\alpha\cdot} \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.

Case $\text{c2r}_{\alpha\cdot \tau\cdot v\cdot} E = c2r(e)$, $T = \text{ApproxR}(\alpha, C)$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \alpha$ $(\Theta; \Delta \vdash \alpha \ll C)$.

(1) $\eta \neq \alpha$. By induction hypothesis, $\Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash e[\delta] : \alpha[\delta]$. $\alpha[\delta] = \alpha$. And $T[\delta] = T$ since the only free type variable $T$ contains is $\alpha$. By $\text{c2r}_{\alpha\cdot \tau\cdot v\cdot} \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.

(2) $\eta = \alpha$. Then by Lemma 11, $\Theta; \Delta'/\delta \vdash s \ll C$. By induction hypothesis $\Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash e : s$.

By $\text{c2r\ } \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : R(s)$. By Lemma 19, $\Theta; \Delta'/\delta \vdash R(s) \leq \text{ApproxR}(\alpha, C)[s/\alpha]$. By $\text{sub\ } \Theta; \Delta'/\delta; \Sigma[\delta]; \Gamma[\delta] \vdash E[\delta] : T[\delta]$.
Case r2im $E = r2im[1, C](\epsilon), \ T = Imty(I, C)$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e : R_I(I, C)$.

By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e[\delta] : R_I(I, C)[\delta]$. $R_I(I, C)[\sigma] = R_I(I, C)$ because $R_I(I, C)$ has no free type variable. By r2im $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case im2r $E = im2r(\epsilon), \ T = R_I(I, \tau)$.

By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e[\delta] : Imty(I, \tau)[\delta]$. $Imty(I, \tau)[\sigma] = Imty(I, \tau[\sigma])$. By im2r and $R_I(I, \tau)[\sigma] = R_I(I, \tau)[\sigma]$. By induction hypothesis $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case record $E = \{\tau_1 = t_1, \ldots, \tau_n = t_n\} \subset T = \{\tau_1^1 = \tau_1, \ldots, \tau_n^1 = \tau_n\}$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1[i] : \tau_i[\delta] \forall 1 \leq i \leq n$.

By induction hypothesis $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_1[i] : \tau_i[\delta] \forall 1 \leq i \leq n$. By record $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case field $E = e.\pi_i, \ T = \tau_i$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : S \subset T = \{\tau_1^1 = \tau_1, \ldots, \tau_n^1 = \tau_n\}$ and $1 \leq i \leq n$.

By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e[i] : S[\delta]$. By field $S[\delta] = \{\tau_1^1 = \tau_1, \ldots, \tau_n^1 = \tau_n\}$, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case assignR $E = e_1.e_2 := e_2$ in $e_3, \ T = \tau$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : S \subset T = \{\tau_1^1 = \tau_1, \ldots, \tau_n^1 = \tau_n\}$. By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_2[\delta] : \tau[i_1]$. By induction hypothesis we have $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_2[\delta] : \tau[i_1]$. By assignR, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case assignA $E = e_1[e_2] := e_2$ in $e_4, \ T = \tau'$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau$, $\Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \int$. By induction hypothesis we have $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_1[\delta] : (array(\tau))(\delta)$. By assignA, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case let $E = x : \tau = e_1$ in $e_2, \ T = \tau'$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \tau$ and $\Theta; \Delta; \Sigma; \Gamma \vdash x : \tau$, $\Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau'$. By induction hypothesis $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_1[\delta] : \tau[\delta]$. By assignA, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case assign $E = x := e_1$ in $e_2, \ T = \tau$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \Gamma(x)$ $(x \vdash M \Gamma(x) \in \Gamma)$ and $\Theta; \Delta; \Sigma; \Gamma \vdash e_2 : \tau$.

By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e_1[\delta] : (\Gamma(x))[\delta]$. By assignA, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case call $E = e[t_1, \ldots, t_m](e_1, \ldots, e_n), \ T = \tau[\sigma]$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau[\sigma]$ and $\Theta; \Delta; \Sigma; \Gamma \vdash e_1[i] \forall 1 \leq i \leq n$. By induction hypothesis $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e[\delta] : \tau[\delta]$. By assignA, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case pack $E = \text{pack } \tau$ as $\alpha \ll \tau_\alpha$ in $e : \tau'$. $T = \exists \alpha \ll \tau_\alpha, \tau'$ with subderivation $\Theta; \Delta; \Sigma; \Gamma \vdash e : \tau'[\tau/\alpha]$ $(\Theta; \Delta \vdash \tau' \ll \tau_\alpha, \alpha \notin \text{domain}(\Delta))$. By induction hypothesis, $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash e[\delta] : \tau'[\tau/\alpha][\delta]$. $T[\delta] = \exists \alpha \ll \tau_\alpha[\delta], \tau'[\delta][\alpha]$ and by permutation of substitution Lemma 36 we have $\tau'[\tau/\alpha][\delta] = \tau'[\tau/\alpha][\delta]$. By Lemma 11, $\Theta; \Delta'[\delta] \vdash t_1[i]$$ \ll \alpha[\sigma][\delta]$. By pack $\Theta; \Delta'[\delta]; \Sigma'[\delta]; \Gamma'[\delta] \vdash E[\delta] : T[\delta]$.

Case open $E = (\alpha, x) = \text{open}(e_1)$ in $e_2, \ T = \tau'$ with subderivations $\Theta; \Delta; \Sigma; \Gamma \vdash e_1 : \exists \beta \ll \tau_\alpha, \tau'$ and $\Theta; \Delta, \alpha \ll \tau_\alpha; \Sigma; \Gamma, x : \tau[\alpha/\beta] \ll e_2 : \tau'$ $(\alpha \notin \text{domain}(\Delta), \alpha \notin \text{free}(\tau'))$. 34
By induction hypothesis, Θ; Δ'[δ]; Σ[δ]; Α[δ] ⊢ e1[δ]; (3β ≪ τu. τ)[δ] and Θ; Δ'[δ], α ≪ τu[δ]; Σ[δ]; Α[δ], x : τ[α/β][δ] ⊢ e2[δ]; τ'[δ]. We know (3β ≪ τu. τ)[δ] = 3β ≪ τu[δ]. τ[δ] and τ[α/β][δ] = τ[δ][α/δ][β] = τ[δ][α/β]. By open Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].

Cases pack >> and open >> are similar to pack and open.
Case ifParent E = ifParent′(e) then bind (α, x) in e1 else e2, T = τ with subderivations Θ; Δ; Σ; Γ ⊢ e : Tag(τ′), Θ; Δ, α ⊨ τ′ : Σ; Γ, x : Tag(α) ⊢ e1 : τ and Θ; Δ; Σ; Γ ⊢ e2 : τ.
By induction hypothesis, Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e : Tag(τ'[δ]), Θ; Δ'[δ], α ⊨ τ'[δ]; Σ[δ]; Γ[δ], x : Tag(α)[δ] ⊢ e1[δ]; τ[δ] and Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e2[δ]; τ[δ]. Tag(τ'[δ]) = Tag(τ'[δ]) and Tag(α)[δ] = Tag(α) because α is a fresh type variable. By ifParent′ Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].
Case ifTag_eq E = ifTag′(e1, e2) then e1 else e2, T = τ with subderivations Θ; Δ; Σ; Γ ⊢ e1 : Tag(C1), Θ; Δ; Σ; Γ ⊢ e2 : Tag(C2) and Θ; Δ; Σ; Γ ⊢ e1 : τ (C1 = C2).
By induction hypothesis, Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e1[δ] : Tag(C1)[δ], Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e2[δ] : Tag(C2)[δ] and Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e1[δ] : τ[δ]. Tag(C1)[δ] = Tag(C1). Tag(C2)[δ] = Tag(C2). By ifTag_eq Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].
Case ifTag_neq E = ifTag′(e1, e2) then e1 else e2, T = τ with subderivations Θ; Δ; Σ; Γ ⊢ e1 : Tag(γ), Θ; Δ; Σ; Γ ⊢ e2 : Tag(τ′), Δ = Δ′, P(γ), Δ2 where P(γ) = γ ≪ u or P(γ) = γ ⊨ u, Θ; Δ1 ⊨ τγ : Ωc, Θ; Δ; Σ; Γ ⊢ e1 : τ[γ/τγ] and Θ; Δ; Σ; Γ ⊢ e2 : τ.
By induction hypothesis, Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e1[δ] : Tag(γ)[δ], Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e2[δ] : Tag(τ′)[δ], Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e1[δ] : τ[γ/τγ][δ] and Θ; Δ'[δ]; Σ[δ]; Γ[δ] ⊢ e2[δ] : τ[δ].
(1) η ≨ γ. Let Δ1 = (Δ1 − η)[δ], then Δ′[δ] = Δ′[δ], P(γ)[δ], Δ2[δ]. By the property that type substitution preserves kinds Lemma 11 Θ; Δ1 ⊨ τγ : Ωc. We have Tag(γ)[δ] = Tag(γ), Tag(τ′)[δ] = Tag(τ′)[δ]. By permutation of substitutions Lemma 36 τ[γ/τγ][δ] = τ[δ][γ/τγ][δ]. By ifTag_eq Θ; Δ′[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].
(2) η = γ. Then Δ1 is empty and τγ must be a class name. Δ′ = Δ2. We have Tag(γ)[δ] = Tag(s), Tag(τ′)[δ] = Tag(τ′).
If s = τγ, then τ[γ/τγ][δ] = τ[δ]. By ifTag_eq Θ; Δ′[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ]. If s ≠ τγ, by ifTag_neq Θ; Δ′[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].
Case sub E = e and T = τ2 with subderivation Θ; Δ; Σ; Γ ⊢ e : τ1 (Θ; Δ ⊨ τ1 ⊨ τ2).
By induction hypothesis, Θ; Δ′[δ]; Σ[δ]; Γ[δ] ⊢ e[δ] : τ1[δ]. Because type substitution preserves subtyping (Lemma 11), Θ; Δ′[δ] ⊨ τ1[δ] ⊨ τ2[δ]. By sub Θ; Δ′[δ]; Σ[δ]; Γ[δ] ⊢ E[δ] : T[δ].

Corollary 38 If τus = α1 ≪ u1, ..., αm ≪ um, σ = τ1, ..., τm/α1, ..., αm, and ∀1 ≤ i ≤ m, Θ; • ⊢ ti ≪ u1[σ], and Θ; τus; Σ; Γ ⊨ e : τ, then Θ; •; Σ[σ]; Γ[σ] ⊢ e[σ] : τ[σ].

C.3.2 Preservation

Lemma 39 Inversion of Values

• If H(ℓ) = {k1 = v1, ..., kn = vn} and Θ; •; Σ; Γ ⊢ ℓ : {l1, ..., ln : τn}, then m ≥ n, k1 = l1 and Θ; •; Σ; Γ ⊢ vi : τi ∀1 ≤ i ≤ n.
• If H(ℓ) = [v0, ..., vn−1]′ and Θ; •; Σ; Γ ⊢ ℓ : array(τ′), then τ′ = τ and Θ; •; Σ; Γ ⊢ vi : τ′ ∀0 ≤ i ≤ n−1.
• If Θ; •; Σ; Γ ⊢ ℓ : α1 ≪ u1, ..., αm ≪ um (τ1, ..., τn) → τ, and Σ(ℓ) = fun < α1 ≪ u′1, ..., αm ≪ w′m > (x1 : s1, ..., xn : sn) : s = e0, then (1) let τγ = ∀1 ≤ u1, ..., αm ≪ um (τ1, ..., τn) → τ and τ′f = ∀1 ≤ u1, ..., αm ≪ w′m (s1, ..., sn) → s, then Σ(ℓ) = τ′f and Θ; • ⊢ τ′f ≤ τf. (2) Θ; τus; Σ; g ⊢ τ′f, x1 : s1, ..., xn : sn ⊢ e0 : s.
• If Θ; •; Σ; Γ ⊢ v : 3β ≪ τv, τ and v = pack s0 as β ≪ sv in (v′ : s), then Θ; • ⊢ s0 ≪ sv, Θ; • ⊢ sv ≪ τv, Θ; β ≪ Topc ⊨ s ≤ τ, and Θ; •; Σ; Γ ⊢ v′ : s[s0/β].

35
• If $\Theta;\bullet;\Sigma;\Gamma \vdash v : \exists \beta \gg u$, and $v = \text{pack } s_0$ as $\beta \gg s_u$ in $(v' : s)$, then $\Theta;\bullet \vdash s_u \ll s_0$, $\Theta;\bullet \vdash \tau_u \ll s_u$, $\Theta;\beta \ll \text{Topc} \vdash s \leq \tau$, and $\Theta;\bullet;\Sigma;\Gamma \vdash v' : s[s_0/\beta]$.

• If $v = \text{tag}(C)$ and $\Theta;\bullet;\Sigma;\Gamma \vdash \text{Tag}(\tau)$, then $\tau = C$.

• If $\Theta;\bullet;\Sigma;\Gamma \vdash C(v) : C'$ then $C = C'$ and $\Theta;\bullet;\Sigma;\Gamma \vdash v : R(C)$.

• If $\Theta;\bullet;\Sigma;\Gamma \vdash \text{r2im}[I,C](v) : \text{Inty}(I',C')$ then $I = I'$, $C = C'$ and $\Theta;\bullet;\Sigma;\Gamma \vdash v : R_I(I,C)$.

Definition 40 Extension of Environments. $\Sigma'$ extends $\Sigma$ means that $\Sigma' = \Sigma, \Sigma_{\text{new}}$ for some $\Sigma_{\text{new}}$ such that $\text{domain}(\Sigma) \cap \text{domain}(\Sigma_{\text{new}}) = \emptyset$. Similarly, $\Gamma'$ extends $\Gamma$ means that $\Gamma' = \Gamma, \Gamma_{\text{new}}$ for some $\Gamma_{\text{new}}$ such that $\text{domain}(\Gamma) \cap \text{domain}(\Gamma_{\text{new}}) = \emptyset$.

Theorem 41 Evaluation of LILC_1 expressions preserves types. If

• $\text{freetvs}(\Sigma) = \emptyset$ $\Theta \vdash H : \Sigma$ $\Theta;\Sigma \vdash V : \Gamma$

• $\Theta;\bullet;\Sigma;\Gamma \vdash E : T$

• $(H;V;E) \mapsto (H';V';E')$,

then $\exists \Sigma'$ and $\Gamma'$ such that

1. $\text{freetvs}(\Sigma') = \emptyset$ $\Theta \vdash H' : \Sigma'$ $\Theta;\Sigma' \vdash V' : \Gamma'$

2. $\Theta;\bullet;\Sigma';\Gamma' \vdash E' : T$

3. $\Sigma'$ extends $\Sigma$ and $\Gamma'$ extends $\Gamma$.

Proof: by induction on expression typing rules.

Case int, Case label, Case tag: $E$ is a value in each case. No evaluation rules apply.

The cases that apply congruence rules are trivial. We will use only record as an example to illustrate the proofs in those cases, and omit the rest for simplicity.

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

The only applicable evaluation rule is $\text{ev}_{\text{var}}$, where $H' = H$, $V' = V$, $E' = V(x)$. By $\Theta;\Sigma \vdash V : \Gamma$, we have $\Theta;\bullet;\Sigma;\Gamma \vdash V(x) : \Gamma(x)$. By weakening of type environments Lemma 10, $\Theta;\bullet;\Sigma;\Gamma \vdash V(x) : \Gamma(x)$. Let $\Sigma' = \Sigma$, $\Gamma' = \Gamma$, then $\Theta;\bullet;\Sigma';\Gamma' \vdash V' : T$.

Case error: $E = \text{error}[\tau]$, $T = \tau$: trivial.

Case object: $E = C(e)$, $T = C$ with subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : R(C)$: only the congruence rule can apply.

Case c2r.c: $E = c2r(e)$, $T = R(C)$ and subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : C$.

If rule $\text{ev}_{\text{c2r}}$ applies, $e = C'(v)$, $H' = H$, $V' = V$ and $E' = v$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. By Lemma 39 $\Sigma' = \Sigma$, $\Gamma' = \Gamma$ and $\Theta;\bullet;\Sigma;\Gamma \vdash v : R(C)$, which is exactly $\Theta;\bullet;\Sigma';\Gamma' \vdash E' : T$.

Case c2r.jv: $E = c2r(e)$, $T = \text{ApproxR}(\alpha,C)$ and subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : \alpha$: not applicable because by Lemma 13, the subderivation $\Theta;\bullet;\Sigma;\Gamma \vdash e : \alpha$ is invalid.

Case r2im: $E = r2im[I,C](e)$, $T = \text{Inty}(I,C)$ with subderivations $\Theta;\Delta;\Sigma;\Gamma \vdash e : R_I(I,C)$.

If rule $\text{ev}_{\text{r2im}}$ applies, $e = r2im[I',C'](v)$, $H' = H$, $V' = V$ and $E' = v$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. By Lemma 39 $\Sigma' = \Sigma$, $\Gamma' = \Gamma$ and $\Theta;\bullet;\Sigma;\Gamma \vdash v : R_I(I,C)$, which is exactly $\Theta;\bullet;\Sigma';\Gamma' \vdash E' : T$.

Case im2r: $E = \text{im2r}(e)$, $T = R_I(I,\tau)$ with subderivation $\Theta;\Delta;\Sigma;\Gamma \vdash e : \text{Inty}(I,\tau)$.

Only the congruence rule can apply.

Case record: $E = \text{new}[\tau]\{t_1 = e_1, \ldots, t_n = e_n\}$, $T = \tau = \{t_1'^{\tau_1}, \ldots, t_n'^{\tau_n}\}$ with subderivations $\Theta;\bullet;\Sigma;\Gamma \vdash e_i : \tau_i$ $\forall 1 \leq i \leq n$.

If the congruence rule applies, then $\exists H', V'$ and $e'_i$ such that $e_1, \ldots, e_{i-1}$ are all values, $(H;V;e_i) \mapsto (H';V';e'_i)$ and $E' = \{e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n\}$. By induction hypothesis, $\exists \Sigma'$ and $\Gamma'$ such that

• $\text{freetvs}(\Sigma) = \emptyset$ $\Theta \vdash H' : \Sigma'$ $\Theta;\Sigma' \vdash V' : \Gamma'$

• $\Theta;\bullet;\Sigma';\Gamma' \vdash e'_i : \tau_i$
Because $\Sigma'$ extends $\Sigma$ and $\Gamma'$ extends $\Gamma$, by Lemmas 9 and 10 $\Theta; \cdot; \Sigma'; \Gamma' \vdash e_j : \tau_j \forall 1 \leq j \leq n$. By record, $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

If ev_record applies, then all $e_1, \ldots, e_n$ are values, $H' = H, \ell \rightsquigarrow \{l_1 = e_1, \ldots, l_n = e_n\}$ where $\ell$ is a fresh label, $V' = V$ and $E' = \ell$. Let $\Sigma' = \Sigma, \ell : \tau$ and $\Gamma' = \Gamma$. Each $e_j (1 \leq i \leq n)$ is a value with no free variables. By strengthening of environments for values Lemma 12 and $\Theta; \cdot; \Sigma; \Gamma \vdash e_i : \tau_i$ we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash e_j : \tau_j$.

By hv_record $\Theta; \Sigma ; \{l_1 = e_1, \ldots, l_n = e_n\} : \tau$. By the well-formedness of heaps (Figure 12), $\Theta \vdash H' : \Sigma'$. By weakening of heap environment Lemma 9, $\Theta; \Sigma' \vdash V' : \Gamma'$. By label, $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

Case field: $E = e.l_i, T = \tau_i$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e : \{l_1^{\tau_1}, \ldots, l_n^{\tau_n} : \tau_n\} (1 \leq i \leq n)$. If ev_field applies, then $e$ is a label, $H(e) = \{k_1 = v_1, \ldots, k_m = v_m\}, H' = H, V' = V$ and $E' = v_i$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. By inversion of value Lemma 39 $m \geq n, k_j = l_j$ and $\Theta; \cdot; \Sigma; \Gamma \vdash v_j : \tau_j \forall 1 \leq j \leq n$. Therefore $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

Case assignR $E = e_1.l_i := e_2 \in e_3, T = \tau$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \{l_1^{\tau_1}, \ldots, l_n^{\tau_n} : \tau_i, \ldots, l_m^{\tau_m} : \tau_m\}, \Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau_i$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_3 : \tau$.

If assignR applies, then $e_1$ is a label, and $H = H_1, e_1 \rightsquigarrow \{k_1 = v_1, \ldots, k_m = v_m\}, H_2$, and $H' = H_1, e_1 \rightsquigarrow \{k_1 = v_1, \ldots, k_{i-1} = v_{i-1}, k_i = e_2, k_{i+1} = v_{i+1}, \ldots, k_m = v_m\}, H_2$, and $V' = V$, and $E' = e_3$. By inversion of values Lemma 39, $m \geq n, k_j = l_j$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_j : \tau_j \forall 1 \leq j \leq n$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. Because $e_2$ is a value and $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau_i$, by Lemma 12, $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau_2$. Then by hv_record $\Theta; \cdot; \Sigma; \Gamma \vdash \{k_1 = v_1, \ldots, k_{i-1} = v_{i-1}, k_i = e_2, k_{i+1} = v_{i+1}, \ldots, k_m = v_m\} : \{l_1^{\tau_1}, \ldots, l_i^{\tau_i}, \ldots, l_m^{\tau_m} : \tau_m\}$.

By the well-formedness of the heap, $H' : \Sigma'$. From $\Theta; \cdot; \Sigma; \Gamma \vdash e_3 : \tau$, we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

Case array $E = \text{new}[e_0, \ldots, e_{n-1}] : T$ and $T = \text{array}(\tau)$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \tau \forall 0 \leq i \leq n - 1$.

If array applies, then all $e_0, \ldots, e_{n-1}$ are values, $H' = H, \ell \rightsquigarrow \{e_0, \ldots, e_{n-1}\} : \tau, V' = V$ and $E' = \ell$. Let $\Sigma' = \Sigma, \ell : \tau$ and $\Gamma' = \Gamma$. By hv_array and $\Theta; \cdot; \Sigma; \Gamma \vdash e_i : \tau(\forall 0 \leq i \leq n - 1)$ and Lemma 12, $\Theta; \cdot; \Sigma; \Gamma \vdash \{e_0, \ldots, e_{n-1}\} : \tau : \text{array}(\tau)$. Then $\Theta \vdash H' : \Sigma'$. By weakening of heap environment Lemma 9, $\Theta; \Sigma' \vdash V' : \Gamma'$. By label, $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

Case subscript $E = e_1[e_2], T = \tau'$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau')$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \text{int}$.

If subscript applies, then $e_1$ is a label, $H(e_1) = \{v_0, \ldots, v_{n-1}\} : \tau$, $e_2 = i (0 \leq i \leq n - 1)$, $H' = H, V' = V$ and $E' = v_i$. By Lemma 39, $\tau' = \tau$ and $\Theta; \cdot; \Sigma; \Gamma \vdash v_j : \tau \forall 0 \leq j \leq n - 1$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. By $\Theta; \cdot; \Sigma; \Gamma \vdash e_3 : \tau$ and Lemma 12, $\Theta \vdash H' : \Sigma'$. From $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau_2$, $\Theta; \cdot; \Sigma' ; \Gamma' \vdash E' : T$.

Case let: $E = x : \tau = e_1 \in e_2, T = \tau'$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \tau$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau'$. If let applies, then $e_1$ is a value, and $H' = H$, and $V' = V, x = e_1$ and $E' = e_2$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma, x : \Sigma, \Gamma$. By $\Theta; \cdot; \Sigma; \Gamma, x : \Sigma, \Gamma \vdash e_2 : \tau'$, we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$. By $\Theta; \cdot; \Sigma; \Gamma \vdash V : \Gamma$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \tau$, we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash V' : \tau'$.

Case assign $E = x := e_1 \in e_2, T = \tau$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \Gamma(x) : \tau (x := \Gamma(x) \in \Gamma)$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau$.

If assign applies, then $e_1$ is a value, and $H' = H$, and $V' = V, x = e_1$ and $E' = e_2$. Let $\Sigma' = \Sigma$ and $\Gamma' = \Gamma$. By $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 \in \Gamma(x)$, we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash V' : \tau'$. By $\Theta; \cdot; \Sigma; \Gamma \vdash V : \Gamma$ and $\Theta; \cdot; \Sigma; \Gamma \vdash e_2 : \tau$, we have $\Theta; \cdot; \Sigma'; \Gamma' \vdash E' : T$.

Case call: $E = e[t_1, \ldots, t_m]([e_1, \ldots, e_n]), T = \tau[\sigma]$ with subderivations $\Theta; \cdot; \Sigma; \Gamma \vdash e : \tau_1$ where $\tau_1 = \forall\text{fun}(\tau_1, \ldots, \tau_n) \rightarrow \tau$ and $\text{fun} = \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m$. By $\Theta; \cdot; \Sigma; \Gamma \vdash e_1 : \tau_1[\sigma] \forall 1 \leq i \leq n$ and $\sigma = t_1, \ldots, t_m/\alpha_1, \ldots, \alpha_m$ and $1 \leq i \leq m$, $\Theta; \cdot; \Gamma \vdash t_1 \ll u_1[\sigma]$.

If rule call applies, then $(1) e$ is a label and $H(e) = \text{fix} g < \text{tvs} > (x_1 : s_1, \ldots, x_n : s_n) : s = s_0$ where $\text{tvs} = \alpha_1 \ll u_1', \ldots, \alpha_m \ll u_m', (2)$ all $e_1, \ldots, e_n$ are values, $(3) H' = H, (4) V' = V, g = e, x_1 = e_1, \ldots, x_n = e_n$ and $(5) E' = e_0[\sigma]$. By inversion of value Lemma 39, $(1) t_i \ll \forall\sigma_1 \subseteq u_1, \ldots, \alpha_m \ll u_m, (s_1, \ldots, s_n) \rightarrow s$, then $\Sigma(e) = \tau_f$ and $\Theta; \cdot; \tau_f \leq \tau_f$ and $(2) \Theta; \text{tvs} ; \Sigma; g : \tau_f : x_1 : s_1, \ldots, x_n : s_n \vdash $
Corollary 42 (Preservation) If $\Sigma \vdash P : \tau$ and $P \rightarrow P'$, then $\exists \Sigma'$ such that $\Sigma' \vdash P' : \tau$. 

C.3.3 Progress

Lemma 43 Canonical forms

1. If $\Theta; \bullet; \Sigma; \Gamma \vdash v : \{l_1^{e_1} : \tau_1, \ldots, l_n^{e_n} : \tau_n\}$ and $\Theta; H : \Sigma$, then $v$ is a label and $H(v) = \{l_1 = v_1, \ldots, l_n = v_n, \ldots\}$. 

2. If $\Theta; \bullet; \Sigma; \Gamma \vdash v : array(\tau)$ and $\Theta; H : \Sigma$, then $v$ is a label and $H(v) = [v_0, \ldots, v_{n-1}]^\tau$. 

3. If $\Theta; \bullet; \Sigma; \Gamma \vdash v : \forall \mathbf{t} \in \mathbf{t}(\tau_1, \ldots, \tau_n) \rightarrow \tau \ (\mathbf{t} = \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m)$ and $\Theta; H : \Sigma$, then $v$ is a label and $H(v) = \{\mathbf{t} : \alpha_1 \ll u_1, \ldots, \alpha_m \ll u_m\}$. 

38
4. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \alpha \ll \tau_\alpha. \tau \), then \( v = \text{pack} \, \tau_0 \) as \( \alpha \ll \tau'_0 \) in \((v' : \tau')\).

5. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \exists \alpha \gg \tau_\alpha. \tau \), then \( v = \text{pack} \, \tau_0 \) as \( \alpha \gg \tau'_0 \) in \((v' : \tau')\).

6. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \text{Tag}(\tau) \), then \( v = \text{tag}(C) \) for some \( C \).

7. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \text{Tag}(C) \), then \( v = \text{tag}(C) \).

8. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : C \), then \( v = C(v') \) for some value \( v' \).

9. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \text{Imty}(I, \tau) \), then \( v = r2\text{im}[I, C](v') \) for some value \( v' \) and class \( C \).

10. If \( \Theta; \bullet; \Sigma; \Gamma \vdash v : \text{int} \), then \( v \) is an integer.


**Theorem 44**
If \( \text{freetys}(\Sigma) = \emptyset \), and \( \Theta; H : \Sigma, \) and \( \Theta; \Sigma \vdash V : \Gamma \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash E : T \), then either \( E \) is a value, or \( E \) can evaluate one step, that is, \( \exists H', V' \) and \( E' \) such that \((H; V; E) \Rightarrow (H'; V'; E')\).

Proof: by induction on expression typing rules.

**Case int**, **Case label**, **Case tag**: all the expressions are values already.

**Case var**: \( E = x \)

By \( \Theta; \Sigma \vdash V : \Gamma \), domain(\( \Gamma \)) = domain(\( V \)). From \( x \in \text{domain}(\( \Gamma \)), \) we know \( x \in \text{domain}(V) \). Let \( H' = H, V' = V \) and \( E' = V(x) \). By \textbf{ev-var} \((H; V; E) \Rightarrow (H'; V'; E')\).

**Case error**: \( E = \text{error}[\{\cdot\}] \) by \textbf{ev-error}, \( E \) steps to itself.

**Case object**: \( E = C(e) \) with subderivation \( \Theta; \bullet; \Sigma; \Gamma \vdash e : R(C) \).

By induction hypothesis, either \( e \) is a value or \( \exists H', V' \) and \( e' \) such that \((H; V; e) \Rightarrow (H'; V'; e')\). If \( e \) is a value, then \( E \) is a value. Otherwise, let \( E' = C(e') \). By the congruence rule, \((H; V; E) \Rightarrow (H'; V'; E')\).

**Case c2r-c**: \( E = c2r(e) \) with subderivation \( \Theta; \bullet; \Sigma; \Gamma \vdash e : C \).

By induction hypothesis, either \( e \) is a value or \( \exists H', V' \) and \( e' \) such that \((H; V; e) \Rightarrow (H'; V'; e')\). If \( e \) is a value, then by Lemma 43 \( e = C(v) \). Let \( E' = v \). By \textbf{ev-c2r} \((H; V; E) \Rightarrow (H'; V'; E')\). Otherwise let \( E' = e2r(e') \). By the congruence rule \((H; V; E) \Rightarrow (H'; V'; E')\).

**Case c2r-tv**: not applicable because by Lemma 13, the subderivation \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \alpha \) is invalid.

**Case r2im**: \( E = r2im[I, C](e) \) with subderivations \( \Theta; \bullet; \Sigma \vdash e : R(I, C) \).

By induction hypothesis, either \( e \) is a value or \( \exists H', V' \) and \( e' \) such that \((H; V; e) \Rightarrow (H'; V'; e')\). If \( e \) is a value, then \( E \) is a value. Otherwise let \( E' = c2r(e') \). By the congruence rule \((H; V; E) \Rightarrow (H'; V'; E')\).

**Case im2r**: \( E = im2r(e) \) with subderivations \( \Theta; \bullet; \Sigma \vdash e : \text{Imty}(I, \tau) \).

By induction hypothesis, either \( e \) is a value or \( \exists H', V' \) and \( e' \) such that \((H; V; e) \Rightarrow (H'; V'; e')\). If \( e \) is a value, then by Lemma 43 \( e = r2im[I, C](v) \). Let \( E' = v \). By \textbf{ev-im2r} \((H; V; E) \Rightarrow (H'; V'; E')\). Otherwise let \( E' = c2r(e') \). By the congruence rule \((H; V; E) \Rightarrow (H'; V'; E')\).

**Case record**: \( E = \text{new}[\{\cdot\}]\{l_1 = e_1, \ldots, l_n = e_n\} \) with subderivations \( \Theta; \bullet; \Sigma \vdash e_i : \tau_i \) \( \forall 1 \leq i \leq n \).

By induction hypothesis, each subexpression \( e_i \) either is a value or can evaluate one more step.

If all \( e_i \) are values, then let \( H' = H, \ell \mapsto \{l_1 = e_1, \ldots, l_n = e_n\} \) (\( \ell \) is a fresh label), \( V' = V \) and \( E' = \ell \). By \textbf{ev-record} \((H; V; E) \Rightarrow (H'; V'; E')\).

If \( \exists e_i \) such that \( e_1, \ldots, e_{i-1} \) are values and \( \exists H', V', e'_i \) such that \((H; V; e_i) \Rightarrow (H'; V'; e'_i)\). Let \( E' = \text{new}[\{\cdot\}]\{l_1 = e_1, \ldots, l_i = e'_i, \ldots, l_n = e_n\} \).

**Case field**: \( E = e.L_i \) with subderivation \( \Theta; \bullet; \Sigma \vdash e : \{l_0^{\tau_1}, \ldots, l_0^{\tau_n} : \tau_n\} \) and \( 1 \leq i \leq n \).

By induction hypothesis, either \( e \) is a value or \( e \) can evaluate one step.

If \( e \) is a value, by canonical form Lemma 43, \( e \) is a label and \( H(e) = \{l_1 = v_1, \ldots, l_n = v_n, \ldots\} \). Let \( H' = H, V' = V \) and \( E' = v_i \). By \textbf{ev-field} \((H; V; E) \Rightarrow (H'; V'; E')\).

If \( \exists H', V', e' \) such that \((H; V; e) \Rightarrow (H'; V'; e')\), then let \( E' = e'.L_i \) and by the congruence rule \((H; V; E) \Rightarrow (H'; V'; E')\).

In the rest cases, we state only which evaluation rule to apply, but omit the new \( H', V' \) or \( E' \).

**Case assignR**: \( E = e_1.L_i := e_2 \) in \( e_3 \) with subderivations \( \Theta; \bullet; \Sigma \vdash e_1 : \{l_1^{\tau_1}, \ldots, l_1^{\tau_n} : \tau_1\} \) and \( \Theta; \bullet; \Sigma \vdash e_2 : \tau_i \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. Similarly, either \( e_2 \) is a value or \( e_2 \) can evaluate one step.
If both \( e_1 \) and \( e_2 \) are values, then by canonical form Lemma 43 \( e_1 \) is a label and \( H(e_1) = \{ l_1 = v_1, \ldots, l_n = v_n, \ldots \} \), and by \texttt{ev_assignR} \( E \) can evaluate one step.

If either \( e_1 \) or \( e_2 \) can evaluate one step, by the congruence rule, \( E \) can evaluate one step.

**Case array** \( E = \text{new}[e_0, \ldots, e_{n-1}]^\tau \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_i : \tau \forall 0 \leq i \leq n-1 \)

By induction hypothesis, each subexpression \( e_i \) either is a value or can evaluate one step. If all \( e_0, \ldots, e_{n-1} \) are values, by \texttt{ev_array} \( E \) can evaluate one step. Otherwise, \( \exists i \) such that \( e_i \) can evaluate one step and \( e_0, \ldots, e_{i-1} \) are all values. By the congruence rule, \( E \) can evaluate one step.

**Case subscript** \( E = e_1[e_2] \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau) \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_2 : \text{int} \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. So \( e_2 \) can.

If \( e_1 \) and \( e_2 \) are both values, by canonical form Lemma 43, \( e_1 \) is a label and \( H(v) = [v_0, \ldots, v_{n-1}]^\tau \) and \( e_2 \) is an integer. The runtime array bounds check guarantees that the index \( e_2 \) is within range, that is, \( 0 \leq e_2 \leq n-1 \). By \texttt{ev_sub} \( E \) can evaluate one step.

If \( e_1 \) or \( e_2 \) can evaluate one step, by the congruence rule \( E \) can evaluate one step.

**Case assign** \( E = e_1[e_2] := e_3 \) in \( e_4 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \text{array}(\tau) \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_2 : \text{int} \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_3 : \tau \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. So do \( e_2 \) and \( e_3 \).

If all \( e_1, e_2 \) and \( e_3 \) are values, by canonical form Lemma 43, \( e_1 \) is a label and \( H(v) = [v_0, \ldots, v_{n-1}]^\tau \) and \( e_2 \) is an integer. The runtime array bounds check guarantees that the index \( e_2 \) is within range, that is, \( 0 \leq e_2 \leq n-1 \). By \texttt{ev_assignA} \( E \) can evaluate one step.

If \( e_1 \) or \( e_2 \) or \( e_3 \) can evaluate one step, by the congruence rule \( E \) can evaluate one step.

**Case let** \( E = x := e_1 \) in \( e_2 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \Gamma(x) \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. By \( \Theta; \Sigma \vdash V : \Gamma, \text{domain}(V) = \text{domain}(\Gamma) \). Since \( x \in \text{domain}(\Gamma) \), \( x \in \text{domain}(V) \). If \( e_1 \) is a value, by \texttt{ev_assign} \( E \) can evaluate one step. Otherwise, by the congruence rule \( E \) can evaluate one step.

**Case call** \( E = e[t_1, \ldots, t_m]|[e_1, \ldots, e_n] \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \forall \text{tvs}(\tau_1, \ldots, \tau_n) \rightarrow \tau \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \tau_i[\sigma] \forall 1 \leq i \leq n \), where \( \sigma = t_1, \ldots, t_m/\text{tvs} \).

By induction hypothesis, either \( e \) is a value or \( e \) can evaluate one step. So does each expression \( e_i \) \( \forall 1 \leq i \leq n \).

If all \( e \) and \( e_1, \ldots, e_n \) are values, then by canonical form Lemma 43, \( e \) is a label and \( H(e) = \text{fix } g(\text{tvs}')(x_1 : \tau_1', \ldots, x_m : \tau_m') : \tau' = e_m \). By \texttt{ev_call} \( E \) can evaluate one step.

If \( e \) or any of \( e_i \) can evaluate one step, by the congruence rule \( E \) can evaluate one step.

**Case pack** \( E = \text{pack} \tau \) as \( \alpha \ll \tau_0 \) in \( (e : \tau)' \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \tau'[\tau/\alpha] \).

By induction hypothesis, either \( e \) is a value or \( e \) can evaluate one step. If \( e \) is a value, then \( E \) is a value. If \( e \) can evaluate one step, then by the congruence rule \( E \) can evaluate one step.

**Case open** \( E = (\alpha, x) = \text{open}(e_1) \) in \( e_2 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \exists \beta \ll \tau_0, \tau \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. If \( e_1 \) is a value, then by canonical form Lemma 43, \( e_1 = \text{pack} \tau_0 \) as \( \beta \ll \tau_0 \) in \( (v : \tau') \). By \texttt{ev_open} \( E \) can evaluate one step. If \( e_1 \) can evaluate one step, then by the congruence rule \( E \) can evaluate one step.

Cases \texttt{pack} and \texttt{open} are similar to \texttt{pack} and \texttt{open}.

**Case ifParent** : \( E = \text{ifParent}(e) \) then bind \((\alpha, x)\) in \( e_1 \) else \( e_2 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \text{Tag}(\tau) \).

By induction hypothesis either \( e \) is a value or \( e \) can evaluate one step.

If \( e \) is a value, then by canonical form Lemma 43 \( e = \text{Tag}(C) \) for some class \( C \). If \( C \) extends some class \( B \), then by \texttt{ev_ifParent1} \( E \) can evaluate one step. If \( C \) does not extend any class, then by \texttt{ev_ifParent2} \( E \) can evaluate one step.

If \( e \) can evaluate one step, then by the congruence rule \( E \) can evaluate one step.

**Case ifTag_eq** : \( E = \text{ifEqTag}(e_1, e_2) \) then \( e_1 \) else \( e_2 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_1 : \text{Tag}(C_1) \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_2 : \text{Tag}(C_2) \) \( (C_1 = C_2) \).

By induction hypothesis, either \( e_1 \) is a value or \( e_1 \) can evaluate one step. So does \( e_2 \).

If \( e_1, e_2 \) are both values, then by canonical form Lemma 43, \( e_1 = e_2 = \text{tag}(C_1) \). By \texttt{ev_eqTag} \( E \) can evaluate one step.
If \( e_{11} \) or \( e_{12} \) can evaluate one step, then by the congruence rule \( E \) can evaluate one step.

**Case ifTag_neq:** \( E = \text{if} E \text{eq} \text{Tag}(e_{11}, e_{12}) \) then \( e_1 \) else \( e_2 \) with subderivations \( \Theta; \bullet; \Sigma; \Gamma \vdash e_{11} : \text{Tag}(C_1) \) and \( \Theta; \bullet; \Sigma; \Gamma \vdash e_{12} : \text{Tag}(C_2) \) (\( C_1 \neq C_2 \)).

By induction hypothesis, either \( e_{11} \) is a value or \( e_{11} \) can evaluate one step. So does \( e_{12} \).

If \( e_{11}, e_{12} \) are both values, then by canonical form Lemma 43, \( e_{11} = \text{tag}(C_1) \) and \( e_{12} = \text{tag}(C_2) \). By \( \text{ev}_\text{neqTag} \) \( E \) can evaluate one step.

If \( e_{11} \) or \( e_{12} \) can evaluate one step, then by the congruence rule \( E \) can evaluate one step.

**Case ifTag_tv:** \( E = \text{if} E \text{eq} \text{Tag}(e_{11}, e_{12}) \) then \( e_1 \) else \( e_2 \) with subderivation \( \Theta; \bullet; \Sigma; \Gamma \vdash e_{11} : \text{Tag}(\gamma) \).

Not applicable because the subderivation is invalid by Lemma 13.

**Case sub:** \( E = e \) with subderivation \( \Theta; \bullet; \Sigma; \Gamma \vdash e : \tau_1 \).

By induction hypothesis, either \( e \) is a value or \( e \) can evaluate one step. That is, either \( E \) is a value or \( E \) can evaluate one step. \( \square \)

**Corollary 45 (progress)** If \( \Sigma \vdash P : \tau \), then either the main expression in \( P \) is a value, or \( \exists P' \) such that \( P \rightarrow P' \).

**Theorem 46** \( \text{LIL}_{C1} \) is Sound. Well-typed \( \text{LIL}_{C1} \) programs do not get stuck.

Proof: by progress and preservation. \( \square \)