Appendix
A. Explicitly-Typed Calculus
In this section, we define our explicitly-typed λ-calculus with sets of type-substitutions that we outlined in Section A.3.

A.1 Types

Definition A.1 (Types). Let \( \mathcal{V} \) be a countable set of type variables ranged over by Greek letter \( \alpha, \beta, \gamma, \ldots \), and \( \mathcal{B} \) a finite set of basic (or constant) types ranged over by \( b \). A type is a term co-inductively produced by the following grammar:

\[
\text{Types} \quad t ::= \quad \alpha \quad \text{type variable} \\
| b \quad \text{basic} \\
| t \times t \quad \text{product} \\
| t \rightarrow t \quad \text{arrow} \\
| t \lor t \quad \text{union} \\
| \lnot t \quad \text{negation} \\
| \emptyset \quad \text{empty}
\]

that satisfies two additional requirements:

- (regularity) the term must have a finite number of different sub-terms.
- (contractivity) every infinite branch must contain an infinite number of occurrences of atoms (i.e., either a type variable or the immediate application of a type constructor: basic, product, arrow).

We use \( \mathcal{T} \) to denote the set of all types.

We write \( t_1 \land t_2 \), \( t_1 \lor t_2 \), and \( \bot \) respectively as an abbreviation for \( t_1 \land \lnot t_2 \), \( \lnot(t_1 \lor \lnot t_2) \), and \( \lnot 0 \).

The condition on infinite branches bars out ill-formed types such as \( t \), \( \bot \) respectively as an abbreviation for \( t, \lnot t \), \( \lnot 1 \)

\( t \in \mathcal{T} \) if and only if \( \var{t} \) is Noetherian (that is, strongly normalizing).

Since types are infinite then the accessory definitions on them will be given either by using memoization (e.g., the definition of \( \var{} \)), the variables occurring in a type: Definition A.2, of by co-inductive techniques (e.g., the definition of \( \text{val}() \)), the variables occurring at top-level of a type: Definition A.3.

Definition A.2 (Type variables). Let \( \var{0} \) and \( \var{1} \) be two functions from \( \mathcal{T} \times \mathcal{P} \mathcal{V} \) to \( \mathcal{P} \mathcal{V} \) defined as:

\[
\begin{align*}
\var{0}(t, \emptyset) &= \begin{cases} 
\emptyset & \text{if } t \in \emptyset \\
\var{1}(t, \emptyset) & \text{otherwise}
\end{cases} \\
\var{1}(\alpha, \emptyset) &= \{ \alpha \} \\
\var{1}(b, \emptyset) &= \emptyset \\
\var{1}(t_1 \times t_2, \emptyset) &= \var{0}(t_1, \emptyset) \cup \var{0}(t_2, \emptyset) \\
\var{1}(t_1 \rightarrow t_2, \emptyset) &= \var{0}(t_1, \emptyset) \cup \var{0}(t_2, \emptyset) \\
\var{1}(t_1 \lor t_2, \emptyset) &= \var{1}(t_1, \emptyset) \cup \var{1}(t_2, \emptyset) \\
\var{1}(\lnot t_1, \emptyset) &= \var{1}(t_1, \emptyset) \\
\var{1}(\emptyset, \emptyset) &= \emptyset
\end{align*}
\]

The set of type variables occurring in a type \( t \), written \( \var{t} \), is defined as \( \var{0}(t, 0) \). A type \( t \) is said to be ground or closed if and only if \( \var{t} \) is empty. We write \( \mathcal{T}_0 \) to denote the set of all the ground types.

Definition A.3 (Top-level variables). Let \( t \) be a type. The set \( \text{thv}(t) \) of type variables that occur at top level in \( t \), that is, all the variables of \( t \) that have at least one occurrence not under a constructor, is defined as:

\[
\begin{align*}
\text{thv}(\alpha) &= \{ \alpha \} \\
\text{thv}(b) &= \emptyset \\
\text{thv}(t_1 \times t_2) &= \emptyset \\
\text{thv}(t_1 \rightarrow t_2) &= \emptyset \\
\text{thv}(t_1 \lor t_2) &= \text{thv}(t_1) \cup \text{thv}(t_2) \\
\text{thv}(\lnot t_1) &= \text{thv}(t_1) \\
\text{thv}(\emptyset) &= \emptyset
\end{align*}
\]

Definition A.4 (Type Substitution). A type-substitution \( \sigma \) is a total mapping of type variables to types that is the identity everywhere but on a finite subset of \( \mathcal{V} \), which is called the domain of \( \sigma \) and denoted by \( \text{dom}(\sigma) \). We use the notation \( \{ t_{1/\alpha_1}, \ldots, t_{n/\alpha_n} \} \) to denote the type-substitution that maps \( \alpha_i \) to \( t_i \), for \( i = 1, \ldots, n \). Given a substitution \( \sigma \), the range of \( \sigma \) is defined as the set of types \( \text{ran}(\sigma) = \{ \sigma(\alpha) \mid \alpha \in \text{dom}(\sigma) \} \), and the set of type variables occurring in the range is defined as \( \text{tvran}(\sigma) = \bigcup_{\alpha \in \text{dom}(\sigma)} \var{\sigma(\alpha)} \).
Definition A.5. Given a type \( t \in T \) and a type-substitution \( \sigma \), the application of \( \sigma \) to \( t \) is co-inductively defined as follows:

\[
\begin{align*}
\beta \sigma &= b \\
(t_1 \times t_2)\sigma &= (t_1\sigma) \times (t_2\sigma) \\
(t_1 \rightarrow t_2)\sigma &= (t_1\sigma) \rightarrow (t_2\sigma) \\
(t_1 \lor t_2)\sigma &= (t_1\sigma) \lor (t_2\sigma) \\
(-t)\sigma &= \neg(t\sigma) \\
0\sigma &= 0 \\
\alpha \sigma &= \sigma(\alpha) & \text{if } \alpha \in \text{dom}(\sigma) \\
\alpha \sigma &= \alpha & \text{if } \alpha \notin \text{dom}(\sigma)
\end{align*}
\]

Definition A.6. Let \( \sigma_1 \) and \( \sigma_2 \) be two substitutions such that \( \text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \emptyset \) (\( \sigma_1 \not\equiv \sigma_2 \) for short). Their union \( \sigma_1 \cup \sigma_2 \) is defined as

\[
(\sigma_1 \cup \sigma_2)(\alpha) = \begin{cases} 
\sigma_1(\alpha) & \alpha \in \text{dom}(\sigma_1) \\
\sigma_2(\alpha) & \alpha \in \text{dom}(\sigma_2) \\
\alpha & \text{otherwise}
\end{cases}
\]

Definition A.7. Given two sets of type-substitutions \([\sigma_i]_{i \in I}\) and \([\sigma_j]_{j \in J}\), we define their composition as

\[
[\sigma_i]_{i \in I} \circ [\sigma_j]_{j \in J} = [\sigma_i \circ \sigma_j]_{i \in I, j \in J}
\]

where

\[
\sigma_i \circ \sigma_j(\alpha) = \begin{cases} 
(\sigma_i(\alpha))\sigma_j & \alpha \in \text{dom}(\sigma_i) \\
\sigma_j(\alpha) & \alpha \in \text{dom}(\sigma_j) \setminus \text{dom}(\sigma_i) \\
\alpha & \text{otherwise}
\end{cases}
\]

A.2 Expressions

Definition A.8 (Expressions). Let \( \mathcal{C} \) be a set of constants ranged over by \( c \) and \( X \) a countable set of expression variables ranged over by \( x, y, z, \ldots \). An expression \( e \) is a term inductively generated by the following grammar:

\[
\text{Expressions} \quad e ::= \quad c \quad \text{constant}\\
| x \quad \text{expression variable}\\
| (e, e) \quad \text{pair}\\
| \pi_i(e) \quad \text{projection}(i \in \{1, 2\})\\
| \lambda^{\sigma_{i \in I} t_i \rightarrow s_i}_{\{j \in J\}} x.e \quad \text{abstraction}\\
| e \quad e \quad \text{application}\\
| e \in \mathcal{E} \quad e : e \quad \text{type case}\\
| e[\sigma_j]_{j \in J} \quad \text{instantiation}
\]

where \( t_i, s_i \) range over types, \( t \in T_0 \) is a ground type and \( \sigma_j \) ranges over type-substitutions. We write \( \mathcal{E} \) to denote the set of all expressions.

A \( \lambda \)-abstraction comes with a non-empty sequence of arrow types (called its interface) and a possibly empty set of type-substitutions (called its decorations). We write \( \lambda^{\sigma_{i \in I} t_i \rightarrow s_i}_{\{j \in J\}} x.e \) for short when the decoration is an empty set.

Since expressions are finite, then the accessory definitions on them can be given by induction.

Given a set of type variables \( \Delta \) and a set of type-substitutions \([\sigma_j]_{j \in J}\), we use the following notation:

\[
\Delta[\sigma_j]_{j \in J} = \bigcup_{j \in J} \bigcup_{\alpha \in \Delta} \text{var}(\sigma_j(\alpha))
\]

Definition A.9. Let \( e \) be an expression. The set \( \text{fv}(e) \) of free variables of the expression \( e \) is defined by induction as:

\[
\begin{align*}
\text{fv}(x) &= \{x\} \\
\text{fv}(c) &= \emptyset \\
\text{fv}(\lambda^{\sigma_{i \in I} t_i \rightarrow s_i}_{\{j \in J\}} x.e) &= \text{fv}(c) \setminus \{x\} \\
\text{fv}(\pi_i(e)) &= \text{fv}(e) \\
\text{fv}(e_1 e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(\lambda^{\sigma_{i \in I} t_i \rightarrow s_i}_{\{j \in J\}} x.e) &= \text{fv}(e) \cup \{x\} \\
\text{fv}(e[\sigma_j]_{j \in J}) &= \text{fv}(e) \cup \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(\pi_i(e)) &= \text{fv}(e) \cup \text{fv}(e_1) \cup \text{fv}(e_2) \\
\text{fv}(e[\sigma_j]_{j \in J}) &= \text{fv}(e)
\end{align*}
\]
The set \( \text{bv}(e) \) of bound variables of the expression \( e \) is defined by induction as:

\[
\begin{align*}
\text{bv}(x) &= \emptyset \\
\text{bv}(c) &= \emptyset \\
\text{bv}(e_1, e_2) &= \text{bv}(e_1) \cup \text{bv}(e_2) \\
\text{bv}(\lambda x. e) &= \text{bv}(e) \\
\text{bv}(\lambda x \in t_i . s_i . x. e) &= \text{bv}(e) \cup \{x\} \\
\text{bv}(e_1, e_2) &= \text{bv}(e_1) \cup \text{bv}(e_2) \\
\text{bv}(e \circ t ? e_1 : e_2) &= \text{bv}(e) \cup \text{bv}(e_1) \cup \text{bv}(e_2) \\
\text{bv}(e(\sigma_j)_{i \in J}) &= \text{bv}(e)
\end{align*}
\]

The set \( \text{tv}(e) \) of type variables occurring in \( e \) is defined by induction as:

\[
\begin{align*}
\text{tv}(x) &= \emptyset \\
\text{tv}(c) &= \emptyset \\
\text{tv}(e_1, e_2) &= \text{tv}(e_1) \cup \text{tv}(e_2) \\
\text{tv}(\pi(e)) &= \text{tv}(e) \\
\text{tv}(\lambda x \in t_i . s_i . x. e) &= \text{tv}(\lambda x \in t_i . s_i . x.e) \cup \text{var}(\Lambda_i \in J, t_i . s_i \rightarrow \sigma_j) \\
\text{tv}(e_1, e_2) &= \text{tv}(e_1) \cup \text{tv}(e_2) \\
\text{tv}(e \circ t ? e_1 : e_2) &= \text{tv}(e) \cup \text{tv}(e_1) \cup \text{tv}(e_2) \\
\text{tv}(e[\sigma_j]_{i \in J}) &= \text{tv}(e[\sigma_j]_{i \in J})
\end{align*}
\]

An expression \( e \) is closed if \( \text{tv}(e) \) is empty.

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. We equate expressions up to the \( \alpha \)-renaming of their bound expression variables. In particular, when substituting an expression \( e \) for a variable \( y \) in an expression \( e' \) (see Definition A.10), we assume that the bound variables of \( e' \) are distinct from the bound and free variables of \( e \), to avoid unwanted captures. For example, \( (\lambda x \rightarrow \beta . x.x)y \) is \( \alpha \)-equivalent to \( (\lambda x \rightarrow \beta . x.x)y \).

The situation is a bit more complex for type variables, as we do not have an explicit binder for them. Intuitively, a type variable can be \( \alpha \)-converted if it is a polymorphic one, that is, if it can be instantiated. For example, \( (\lambda x \rightarrow \beta . x.x)y \) is \( \alpha \)-equivalent to \( (\lambda x \rightarrow \beta . x.x)y \) and \( (\lambda x \rightarrow \alpha . x.x)y \) is \( \alpha \)-equivalent to \( (\lambda x \rightarrow \beta . x.x)y \). Polymorphic variables can be bound by interfaces, but also by decorations: for example, in \( \lambda x \rightarrow \lambda y . (\lambda x \rightarrow \lambda y . x.y)x \). The \( \alpha \) occurring in the interface of the inner abstraction is bound by the decoration \( [\text{Int}_{\alpha}] \), and the whole expression is \( \alpha \)-equivalent to \( (\lambda x \rightarrow \lambda y . (\lambda x \rightarrow \beta . y.y)x) \). If a type variable is bound by an outer abstraction, it cannot be instantiated; such a variable is called monomorphic. For example, the expression \( \lambda (\alpha \rightarrow \alpha . x.x)[\text{Int}_{\alpha}] \) is not sound (i.e., it cannot be typed), because \( \alpha \) is bound at the level of the outer abstraction, not at level of the inner one. Consequently, in this expression, \( \alpha \) is monomorphic for the inner abstraction, but polymorphic for the outer one. Monomorphic type variables cannot be \( \alpha \)-converted: \( (\alpha (\alpha \rightarrow \alpha . x.x)y \) is not \( \alpha \)-equivalent to \( (\lambda x \rightarrow \beta . x.x)y \). Note that the scope of polymorphic variables may include some type-substitutions \( [\sigma_j]_{i \in J} \); for example, \( (\lambda (\alpha \rightarrow \beta . x.x)y)[\text{Int}_{\beta}] \) is \( \alpha \)-equivalent to \( (\lambda x \rightarrow \beta . x.x)y[\text{Int}_{\beta}] \). Finally, we have to be careful when performing expression and type-substitutions to avoid clashes of polymorphic variables namespaces. For example, substituting \( \lambda x \rightarrow \alpha . z.z \) for \( y \) in \( (\alpha \rightarrow \alpha . x.x y \) would lead to an unwanted capture of \( \alpha \) (assuming \( \alpha \) is polymorphic, i.e., not bound by a \( \lambda \)-abstraction placed above this two expressions), so we have to \( \alpha \)-convert one of them, so that the result of the substitution is, for instance, \( \alpha \rightarrow \alpha . x.x (\lambda \beta \rightarrow \beta . x.x) \).

To resume, we adopt the following conventions on \( \alpha \)-conversion for type variables. We assume that polymorphic variables are pairwise distinct and distinct from any monomorphic variables in the expressions under consideration. We equate expressions up to \( \alpha \)-renaming of their polymorphic variables. In particular, when substituting an expression \( e \) for a variable \( y \) in an expression \( e' \) (see Definition A.10), we suppose the polymorphic type variables of \( e' \) to be distinct from the monomorphic and polymorphic type variables of \( e \) to avoid unwanted capture\(^6\).

**Definition A.10 (Expression Substitution).** An expression substitution \( \varrho \) is a total mapping of expression variables to expressions that is the identity everywhere but on a finite subset of \( X \), which is called the domain of \( \varrho \) and denoted by \( \text{dom}(\varrho) \). We use the notation \( \{e_1/x_1, \ldots, e_n/x_n\} \) to denote the expression substitution that maps \( x_i \), into \( e_i \), for \( i = 1 \ldots n \).

\(^6\)In this discussion, the definitions of the notions of polymorphic and monomorphic variables remain informal. To make them more formal, we would have to distinguish between the two by carrying around a set of type variables \( \Delta \) which would contain the monomorphic variables that cannot be \( \alpha \)-converted. Then all definitions (such as expression substitutions, for example) would have to be parametrized with \( \Delta \), making the definitions and technical developments difficult to read just because of \( \alpha \)-conversion. Therefore, for the sake of readability, we decided to keep the distinction between polymorphic and monomorphic variables informal.
The definitions of free variables, bound variables and type variables are extended to expression substitutions as follows.

\[ \text{fv}(\varrho) = \bigcup_{x \in \text{dom}(\varrho)} \text{fv}(\varrho(x)), \quad \text{bv}(\varrho) = \bigcup_{x \in \text{dom}(\varrho)} \text{bv}(\varrho(x)), \quad \text{tv}(\varrho) = \bigcup_{x \in \text{dom}(\varrho)} \text{tv}(\varrho(x)) \]

We now define the application of an expression substitution \( \varrho \) to an expression \( e \). To avoid unwanted captures, we remind that we assume that the bound variables of \( e \) do not occur in the domain of \( \varrho \) and that the polymorphic type variables of \( e \) are distinct from the type variables occurring in \( \varrho \) (using \( \alpha \)-conversion if necessary).

**Definition A.11.** Given an expression \( e \in \mathcal{E} \) and an expression substitution \( \varrho \), the application of \( \varrho \) to \( e \) is defined as follows:

\[
\begin{align*}
(\varepsilon_1, \varepsilon_2) \varrho & = \varepsilon\quad (\pi_i(e)) \varrho = \pi_i(\varepsilon \varrho) \\
(\lambda x \in t \mapsto s \cdot x) \varrho & = \lambda x \in t \mapsto s \cdot (\varepsilon x) \\
(e \in t \mid \varepsilon_1 : \varepsilon_2) \varrho & = (\varepsilon_1 \varrho) \cdot (\varepsilon_2 \varrho) \\
(x \varrho) & = \varepsilon(x) \quad \text{if } x \in \text{dom}(\varrho) \\
(x \varrho) & = x \quad \text{if } x \not\in \text{dom}(\varrho)
\end{align*}
\]

In an instantiation \( e[\sigma_j]_{j \in J} \), the \( \sigma_j \) operate on the polymorphic type variables, which we assume distinct from the variables in \( \varrho \) (using \( \alpha \)-conversion if necessary). As a result, we have \( tv(\varrho) \cap \bigcup_{j \in J} \text{dom}(\sigma_j) = \emptyset \). Similarly, in the abstraction case, we have \( x \not\in \text{dom}(\varrho) \).

We now define the relabeling of an expression \( e \) with a set of type substitutions \( [\sigma_j]_{j \in J} \), which consists in propagating the \( \sigma_j \) to the \( \lambda \)-abstractions in \( e \) if needed. We suppose that the polymorphic type variables in \( e \) are distinct from the type variables in the range of \( \sigma_j \) (this is always possible using \( \alpha \)-conversion).

**Definition A.12 (Relabeling).** Given an expression \( e \in \mathcal{E} \) and a set of type-substitutions \( [\sigma_j]_{j \in J} \), we define the relabeling of \( e \) with \( [\sigma_j]_{j \in J} \), written \( e \oplus [\sigma_j]_{j \in J} \), as \( e \) if \( tv(e) \cap \bigcup_{j \in J} \text{dom}(\sigma_j) = \emptyset \), and otherwise as follows:

\[
\begin{align*}
(\varepsilon_1, \varepsilon_2) \oplus [\sigma_j]_{j \in J} & = (\varepsilon_1 \oplus [\sigma_j]_{j \in J}, \varepsilon_2 \oplus [\sigma_j]_{j \in J}) \\
(\pi_i(e)) \oplus [\sigma_j]_{j \in J} & = \pi_i(\varepsilon \oplus [\sigma_j]_{j \in J}) \\
(\lambda x \in t \mapsto s \cdot x) \oplus [\sigma_j]_{j \in J} & = \lambda x \in t \mapsto (s \oplus [\sigma_j]_{j \in J}) \\
(e \in t \mid \varepsilon_1 : \varepsilon_2) \oplus [\sigma_j]_{j \in J} & = e \oplus [\sigma_j]_{j \in J} \mid e_1 \oplus [\sigma_j]_{j \in J} : e_2 \oplus [\sigma_j]_{j \in J} \\
(x \oplus [\sigma_j]_{j \in J}) & = \varepsilon(x) \oplus [\sigma_j]_{j \in J} \quad \text{if } x \in \text{dom}(\varrho) \\
(x \oplus [\sigma_j]_{j \in J}) & = x \quad \text{if } x \not\in \text{dom}(\varrho)
\end{align*}
\]

The substitutions are not propagated if they do not affect the variables of \( e \) (i.e., if \( tv(e) \cap \bigcup_{j \in J} \text{dom}(\sigma_j) = \emptyset \)). In particular, constants and variables are left unchanged, as they do not contain any type variable. Suppose now that \( tv(e) \cap \bigcup_{j \in J} \text{dom}(\sigma_j) \neq \emptyset \). In the abstraction case, the propagated substitutions are composed with the decorations of the abstraction, without propagating them further down in the body. Propagation in the body occurs, whenever is needed, that is, during either reduction (see (Rappl) in Section A.4) or type-checking (see (abstr) in Section A.3). In the instantiation case \( e[\sigma_j]_{j \in J} \), we propagate the result of the composition of \( [\sigma_j]_{j \in J} \) with \( [\sigma_j]_{j \in J} \) in \( e \). The remaining cases are simple inductive cases. Finally notice that in a type case \( e \in t \mid e_1 : e_2 \), we do not apply the \( [\sigma_j]_{j \in J} \) to \( t \), simply because \( t \) is ground.

### A.3 Type System

Because of the type directed nature of our calculus, its dynamic semantics is only defined for well-typed expressions. We therefore introduce now the type system before giving the reduction rules.

**Definition A.13 (Typing Environment).** A typing environment \( \Gamma \) is a finite mapping from expression variables \( \chi \) to types \( \mathcal{T} \), and written as a finite set of pairs \( \{ (x_1 : t_1), \ldots, (x_n : t_n) \} \). The set of expression variables which is defined in \( \Gamma \) is called the domain of \( \Gamma \), denoted by \( \text{dom}(\Gamma) \). The set of types occurring in \( \Gamma \), that is \( \bigcup_{t \in \mathcal{T}} \text{var}(t) \), is denoted by \( \text{var}(\Gamma) \). If \( \Gamma \) is a type environment, then \( \text{\Gamma}, \langle x : t \rangle \) is the type environment defined as

\[
(\Gamma, \langle x : t \rangle)(y) = \begin{cases} t & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}
\]

The definition of type-substitution application can be extended to type environments by applying the type-substitution to each type in the type environment, namely,

\[
\Gamma \sigma = \{ (x : t \sigma) \mid (x : t) \in \Gamma \}
\]

The typing judgement for expressions has the form \( \Delta \vdash e : t \), which states that under the set \( \Delta \) of (monomorphic) type variables and the typing environment \( \Gamma \) the expression \( e \) has type \( t \). When \( \Delta \) and \( \Gamma \)
are both empty, we write \( \vdash e : t \) for short. We assume that there is a basic type \( b_c \) for each constant \( c \). We write \( \sigma \nsubseteq \Delta \) as abbreviation for \( \text{dom} (\sigma) \cap \Delta = \emptyset \). The typing rules are given in Figure 5. Some of these rules deserve a few comments.

\[
\begin{array}{c}
\Delta \frac{\vdash e : b_c \quad (\text{const})}{\vdash \Delta \Gamma \vdash e : \Gamma(x)} \\
\Delta \frac{\vdash e : t_1 \times t_2 \quad (\text{proj})}{\vdash \Delta \Gamma \vdash \pi_1(e) : t_1} \\
\Delta \frac{\vdash e : t_1 \rightarrow t_2 \quad (\text{var})}{\vdash \Delta \Gamma \vdash e : t_1 \rightarrow t_2} \\
\Delta \frac{\vdash e : t_1 \times t_2 \quad (\text{pair})}{\vdash \Delta \Gamma \vdash e_1 : t_1, e_2 : t_2} \\
\Delta \frac{\vdash e : t \quad (\text{appl})}{\vdash \Delta \Gamma \vdash e : t} \\
\Delta \frac{\vdash \lambda^{s_1 \rightarrow s_2} e : s \quad (\text{abstr})}{\vdash \Delta \Gamma \vdash \lambda^{s_1 \rightarrow s_2} e : s} \\
\Delta \frac{\forall i \in I, \exists j \in J, \Delta \vdash e_i : s \quad (\text{case})}{\vdash \Delta \Gamma \vdash e : s} \\
\Delta \frac{\forall j \in J, \Delta \vdash e[s_j]_j : t_j \quad (\text{inter})}{\vdash \Delta \Gamma \vdash e : |J| > 1} \\
\Delta \frac{\vdash e : s \quad (\text{subsum})}{\vdash \Delta \Gamma \vdash e : t} \\
\end{array}
\]

**Figure 5. Typing Rules**

The rule (abstr) is a little bit tricky. Consider a \( \lambda \)-abstraction \( \lambda^{s_1 \rightarrow s_2} e \). Since the type variables introduced in the (relabelled) interface are bound by the abstraction, they cannot be instantiated in the body \( e \), so we add them to the set \( \Delta \) when type-checking \( e \). We have to verify that the abstraction can be typed with each arrow type \( t_i \rightarrow s_i \) in the interface to which we apply each decoration \( s_j \). That is, for each type \( t_i s_i \rightarrow s_i s_j \), we check the abstraction once: the variable \( x \) is assumed to have type \( t_i s_j \), and the relabeled body \( e[s_i]_j \) is checked against the type \( s_i s_j \). Note that relabeling \( e \) with \( s_j \) (i.e., propagating \( s_j \) down to the outermost \( \lambda \)-abstractions in \( e \)) is necessary for the type-checking to succeed, as explained in Section 3.

For a type case \( e \in \{ ? \} e_1 : e_2 \) (rule (case)), we first infer the type \( t' \) of the expression \( e \) which is dynamically tested against \( t \). Then we check the type of each branch \( e_i \) only if there is a possibility that the branch can be selected. For example, consider the first branch \( e_1 \). If \( t' \) has a non-empty intersection with \( t \) (i.e., \( t' \not\subseteq t \)), then \( e_1 \) might be selected. In this case, in order for the whole expression to have type \( s \), we need to check that \( e_1 \) has also type \( s \). Otherwise (i.e., \( t' \subseteq t \)), \( e_1 \) cannot be selected, and there is no need to type-check it. The reasoning is the same for the second branch \( e_2 \) where \( \neg t \) is replaced by \( t \). Note that the ability to ignore the first branch \( e_1 \) and/or the second one \( e_2 \) when computing the type for a type case \( e \in \{ ? \} e_1 : e_2 \) is important to type-check overloaded functions. For example, consider the abstraction \( \lambda^\text{Int \rightarrow Int} (\text{\text{Bool} \rightarrow \text{Bool}}) x . (x \in \text{Int} \ ? \ 42 : \text{false}) \). According to the rule (abstr), the abstraction will be checked against \( \text{Int} \rightarrow \text{Int} \), that is, the body is checked against type \( \text{Int} \) assuming that \( x \) has type \( \text{Int} \). Since \( \text{Int} \subseteq \text{Int} \), the second branch would not be type-checked. Otherwise, the type of the body contains \( \text{\text{Bool}} \), which is not a subtype of \( \text{Int} \) and thus it is not well-typed.

In the rule (inst), we require that only the polymorphic type variables (i.e., those not in \( \Delta \)) can be instantiated. Otherwise, an expression such as \( \lambda^{\alpha \rightarrow \beta} x . x[\beta_1]_\alpha \) could be typed as follows:

\[
\begin{align*}
\{\alpha, \beta\} \vdash (x : \alpha) : \alpha \\
\{\alpha, \beta\} \vdash (x : \alpha) : \beta \\
\vdash \lambda^{\alpha \rightarrow \beta} x . x[\beta_1]_\alpha : \alpha \rightarrow \beta
\end{align*}
\]

which is unsound (by applying the above functions it is possible to create polymorphic values of type \( \beta \), for every \( \beta \)). Moreover, in a judgment \( \Delta \vdash e : t \), the type variables occurring in \( \Gamma \) are to be considered monomorphic, and therefore we require them to be contained in \( \Delta \). Without such a requirement it would be
possible to have a derivation such as the following one:

\[
\frac{\{\alpha\}; \{x : \beta\} \vdash x : \beta}{\{\alpha\}; \{x : \beta\} \vdash \{\gamma/\beta\} : \{\alpha\}}
\]

which states that assuming \(x\) has (any) type \(\beta\), we can infer that it has also (any other) type \(\gamma\). This is unsound. We next prove that this condition \(\text{var}(\Gamma) \subseteq \Delta\) is preserved by the typing rules (see Lemma B.1). When we type a closed expression, typically in the proof of the progress property, we have \(\Delta = \Gamma \supseteq \emptyset\), which satisfies the condition. This implies that all the judgments used in the proofs used for soundness satisfy it. Therefore, henceforth, we implicitly assume the condition \(\text{var}(\Gamma) \subseteq \Delta\) to hold.

The rule \textit{(subsum)} makes the type system depend on the subtyping relation defined in \[\leq\]. It is important not to confuse the subtyping relation \(\leq\) of our system, which denotes semantic subtyping \(\text{ie. set-theoretic inclusion of denotations}\), with the ML one, which stands for type variable instantiation. For example, in ML we have \(\alpha \to \alpha \leq \text{Int} \to \text{Int}\) (because \(\text{Int} \to \text{Int}\) is an instance of \(\alpha \to \alpha\)). But this is not true in our system, as the relation would have to hold for any possible instantiation of \(\alpha\), thus in particular for \(\alpha\) equal to \(\text{Bool}\). Notice that the preorder \(\sqsubseteq\) defined in Section 4 includes the type variable instantiation of the preorder typically used for ML, so any direct comparison with constraint systems for ML types should focus on \(\sqsubseteq\) rather than \(\leq\).

Note that there is not any typing rule for intersection elimination, as it can be encoded by subsumption. Indeed, we have \(t \land s \leq t\) and \(t \land s \leq s\), so from \(\Delta \vdash s : t \land s\), we can deduce \(\Delta \vdash t : t\) (or \(s\)). The rule \textit{(inter)} introduces intersection only to combine different instances of the same type. It does not restrict the expressiveness of the type system, as we prove that the usual rule for intersection introduction is admissible in our system (see Lemma B.2).

### A.4 Operational Semantics

In this section, we define the semantics of the calculus, which depends on the type system given in Section A.3.

**Definition A.14 (Values).** An expression \(e\) is a value if it is closed, well-typed \((\text{ie. } \vdash e : t \text{ for some type } t)\), and produced by the following grammar:

\[
\text{Values} \quad v ::= c \mid (v, v) \mid \lambda_{[\sigma_j] \in J}^{t_1 \to s_1} x.e
\]

We write \(\forall\) to denote the set of all values.

**Definition A.15 (Context).** Let the symbol \([\_]\) denote a hole. A context \(C[\_]\) is an expression with a hole:

\[
\text{Contexts} \quad C[\_] \ ::= \ [\_]
\]

\[
\mid (C[\_], e) \mid (e, C[\_])
\]

\[
\mid C[\_] \mid e \mid C[\_]
\]

\[
\mid C[\_] \mid e \mid C[\_]
\]

\[
\mid \pi_i(C[\_])
\]

\[
\mid \lambda_{[\sigma_j] \in J}^{t_1 \to s_1} x.C[\_]
\]

An evaluation context \(E[\_]\) is a context that implements outermost leftmost reduction:

\[
\text{Evaluation Contexts} \quad E[\_] \ ::= \ [\_]
\]

\[
\mid (E[\_], e) \mid (v, E[\_])
\]

\[
\mid E[\_] \mid e \mid vE[\_]
\]

\[
\mid E[\_] \mid e \mid E[\_]
\]

\[
\mid \pi_i(E[\_])
\]

We use \(C[e]\) and \(E[e]\) to denote the expressions obtained by replacing \(e\) for the hole in \(C[\_]\) and \(E[\_]\), respectively.

We define a small-step operational call-by-value semantics for the calculus. The semantics is given by the relation \(\Rightarrow\), which is shown in Figure 6. There are four notions of reduction: one for projections, one for applications, one for type cases, and one for instantiations, plus context closure. Henceforth we will establish all the properties for the reduction using generic contexts but, of course, these holds also when the more restrictive evaluation contexts are used. The latter will only be used in Section 5 in order to simplify the setting.

The \(\text{(Rproj)}\) rule is the standard projection rule. The \(\text{(Rappl)}\) rule states the semantics of applications: this is standard call-by-value \(\beta\)-reduction, with the difference that the substitution of the argument for the parameter is performed on the relabeled body of the function. Notice that relabeling depends on the type of the argument and keeps only the substitutions that make the type of the argument \(v\) match (at least one of the input types defined in the interface of the function (formally, we select the set \(P\) of substitutions \(\sigma_j\) such that the argument \(v\) has type \(t_i\sigma_j\) for some \(i\)). The \(\text{(Rcase)}\) rule checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly. Finally, the
Proof. By induction on the derivation of $e$.

By syntactic meta-theory as the BDC intersection type system defined by Barendregt, Coppo, and Dezani [1], the typing judgements as the BDC intersection type system defined by Barendregt, Coppo, and Dezani [1].

Finally, we prove that the explicitly-typed calculus is able to derive the same soundness property that substitutions preserve typing, and so on. These are functional to the proof of our type system, in particular the admissibility of the intersection rule, the generation lemma for values, the type of the tail are all is needed. Studying weaker conditions for the reduction rules is an interesting topic.

Branches $\vdash$ preservation. For example, assume that $\vdash t_1 \times 0$ to $(e_1, e_2)$. In our system, a product type with an empty component is itself empty, and thus $e$ has type 0. Therefore the type of the projection as well has type 0 (since $0 \times 0 = 0$, then by subsumption $(e_1, e_2) : 0 \times 0$ and the result follows from the (proj) typing rule). If it were possible to reduce a projection when the argument is not a value, then $e$ could be reduced to $e_1$, which has type $t_1$: type preservation would be violated.

Likewise, the reduction rule for applications requires the argument to be a value. Let us consider the application $(\lambda(x : t \to t \times s)(x, x))(e)$, where $\Delta \vdash e : t \lor s$. The type system assigns to the abstraction the type $(\forall x(x : t \lor s))\Delta \vdash e : t \lor s$, which is a subtype of $(\forall x(x : t \lor s))\Delta \vdash \lambda(x : t \lor s)(x, x)(e) : t \lor s$. If the semantics permits to reduce an application when the argument is not a value, then this application could be reduced to the expression $(e, e)$, which is typed by $(\forall x(x : t \lor s))\Delta \vdash \lambda(x : t \lor s)(x, x)(e) = \lambda(x : t \lor s)(x, x)(e) : t \lor s$.

Finally, if we allow $(\langle e \, t \, e \rangle : e_1)$ to reduce to $e_1$ when $\vdash e : t$ but $e$ is not a value, we could break type preservation. For example, assume that $\vdash e : 0$. Then the type system would not check anything about the branches $e_1$ and $e_2$ (see the typing rule (case) in Figure [5]) and so $e_1$ could be ill-typed.

Notice that in all these cases the usage of values ensure subject-reduction but it is not a necessary condition: in some cases weaker constraints could be used. For instance, in order to check whether an expression is a list of integers, in general it is not necessary to fully evaluate the whole list: the head and the type of the tail are all is needed. Studying weaker conditions for the reduction rules is an interesting topic we leave for future work, in particular in the view of adapting our framework to lazy languages.

B. Properties of the Type System

In this section we present some properties of our type system. First, we study the syntactic meta-theory of our type system, in particular the admissibility of the intersection rule, the generation lemma for values, the property that substitutions preserve typing, and so on. These are functional to the proof of soundness, the fundamental property that link any type system of a calculus with its operational counterpart: well-typed expressions do not go wrong. Finally, we prove that the explicitly-typed calculus is able to derive the same typing judgements as as the BDC intersection type system defined by Barendregt, Coppo, and Dezani [1].

B.1 Syntactic Meta-Theory

Lemma B.1. If $\Delta \vdash e : t$ and $\text{var}(\Gamma) \subseteq \Delta$, then $\text{var}(\Gamma') \subseteq \Delta'$ holds for each judgment $\Delta' \vdash e' : t'$ in the derivation of $\Delta' \vdash e : t$.

Proof. By induction on the derivation of $\Delta' \vdash e : t$.

Lemma B.2 (Admissibility of intersection introduction). Let $e$ be an expression. If $\Delta \vdash e : t$ and $\Delta \vdash e : t'$, then $\Delta \vdash e : t \land t'$.
Proof. The proof proceeds by induction on the two typing derivations. First, assume that these two derivations end with an instance of the same rule corresponding to the top-level constructor of e.

(const): both derivations end with an instance of (const):

\[ \Delta \Gamma \vdash e : b_e \quad (\text{const}) \quad \Delta \Gamma \vdash e : b_e \quad (\text{const}) \]

Trivially, we have \( b_e \land b_e \simeq b_e \), by subsumption, the result follows.

(var): both derivations end with an instance of (var):

\[ \Delta \Gamma \vdash x : \Gamma(x) \quad (\text{var}) \quad \Delta \Gamma \vdash x : \Gamma(x) \quad (\text{var}) \]

Trivially, we have \( \Gamma(x) \land \Gamma(x) \simeq \Gamma(x) \), by subsumption, the result follows.

(pair): both derivations end with an instance of (pair):

\[ \Delta \Gamma \vdash e_1 : t_1 \quad \Delta \Gamma \vdash e_2 : t_2 \quad (\text{pair}) \]

By induction, we have \( \Delta \Gamma \vdash e_1 : (t_1 \times t_2) \). Then the rule (pair) gives us \( \Delta \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \). Moreover, because intersection distributes over products, we have \( (t_1 \times t_2) \times (t_1 \times t_2) \simeq (t_1 \times t_2) \times (t_1 \times t_2) \). Then by (subsum), we have \( \Delta \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \times (t_1 \times t_2) \). Then the rule (pair) gives us \( \Delta \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \). Finally, by applying (pair), we get \( \Delta \Gamma \vdash e_1, e_2 : t_1 \times t_2 \) as expected.

(appl): both derivations end with an instance of (appl):

\[ \Delta \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Delta \Gamma \vdash e_2 : t_1 \quad (\text{appl}) \]

By induction, we have \( \Delta \Gamma \vdash e_1 : (t_1 \rightarrow t_2) \land (t_1 \rightarrow t_2) \) and \( \Delta \Gamma \vdash e_2 : t_1 \). Because intersection distributes over arrows, we have \( (t_1 \rightarrow t_2) \land (t_1 \rightarrow t_2) \leq (t_1 \rightarrow t_1) \rightarrow (t_2 \rightarrow t_2) \). Then by the rule (subsum), we get \( \Delta \Gamma \vdash (e_1, e_2) : (t_1 \rightarrow t_1) \rightarrow (t_2 \rightarrow t_2) \). Finally, by applying (appl), we get \( \Delta \Gamma \vdash e_1, e_2 : t_1 \rightarrow t_2 \) as expected.

(abstr): both derivations end with an instance of (abstr):

\[ \forall i \in I, j \in J. \Delta' \vdash \lambda_{i,j} e : t_i \rightarrow s_{i,j} \]

It is clear that

\[ ( \bigwedge_{i \in I, j \in J} t_i \sigma_j \rightarrow s_{i,j} ) \land ( \bigwedge_{i \in I, j \in J} t_i \sigma_j \rightarrow s_{i,j} ) \simeq \bigwedge_{i \in I, j \in J} t_i \sigma_j \rightarrow s_{i,j} \]

By subsumption, the result follows.

(case): both derivations end with an instance of (case):

\[ \Delta \Gamma \vdash e_0 : t_0 \]

\[ \begin{cases} t_0 \not\leq t \Rightarrow \Delta \Gamma \vdash e_1 : s \quad (\text{case}) \\ t_0 \leq t \Rightarrow \Delta \Gamma \vdash e_2 : s \end{cases} \]
Proof. ...
Lemma B.5. Let $e$ be an expression, $\varrho$ an expression substitution and $[\sigma_j]_{j\in J}$ a set of type substitutions such that $tv(\varrho) \cap \bigcup_{j\in J} \text{dom}(\sigma_j) = \emptyset$. Then $(e\varrho)@[\sigma_j]_{j\in J} = (e@[\sigma_j]_{j\in J})\varrho$.

Proof. By induction on the structure of $e$.

$e = c$: 

\[
\begin{align*}
  c & = (c@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} \\
  & = c@[\sigma_k]_{k\in K} \\
  & = c
\end{align*}
\]

$e = x$: if $x \notin \text{dom}(\varrho)$, then 

\[
\begin{align*}
x & = (x@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} \\
  & = x@[\sigma_k]_{k\in K} \\
  & = x \\
  & = x@\varrho \\
  & = (x@[\sigma_j]_{j\in J})\varrho
\end{align*}
\]
Otherwise, let \( g(x) = e' \). As \( \text{tv}(g) \cap \bigcup_{j \in J} \text{dom}(\sigma_j) = \emptyset \), we have \( \text{tv}(e') \cap \bigcup_{j \in J} \text{dom}(\sigma_j) = \emptyset \). Then

\[
(xg)[\sigma_j]_{j \in J} = e'[\sigma_j]_{j \in J} = e' = \pi g = (x0[\sigma_j]_{j \in J})g
\]

\( e = (e_1, e_2) \):

\[
((e_1, e_2)g)\sigma_j = (e_1g, e_2g)\sigma_j = ((e_1g)\sigma_j, (e_2g)\sigma_j) = ((e_1\sigma_j)g, (e_2\sigma_j)g) \quad \text{(by induction)}
\]

\[
(e_1 \pi e_2)g = (e_1 \pi e_2)[\sigma_j]_{j \in J} = ((e_1 \pi e_2)\sigma_j)_{j \in J} = ((e_1 \sigma_j) (e_2 \sigma_j))_{j \in J} (by \ induction)
\]

\( e = \lambda^{\chi_{e_1, e_2}}_{\sigma_k} \) using \( \alpha \)-conversion on the polymorphic type variables of \( e \), we can assume \( \text{tv}(g) \cap \bigcup_{k \in K} \text{dom}(\sigma_k) = \emptyset \). Consequently we have \( \text{tv}(g) \cap \bigcup_{k \in K} \text{dom}(\sigma_k) = \emptyset \), and we deduce

\[
((e'[\sigma_k]_{k \in K})\sigma_j)_{j \in J} = (e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J} = (\lambda^{\chi_{e_1, e_2}}_{\sigma_k} \circ \sigma_j)_{j \in J} (by \ induction)
\]

\[
((e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J})g = ((e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J})g = (e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J} (by \ induction)
\]

\[
((e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J})g = ((e'[\sigma_k]_{k \in K} \circ \sigma_j)_{j \in J})g
\]

\[
\square
\]

**Lemma B.6 ([Expression] Substitution Lemma).** Let \( e, e_1, \ldots , e_n \) be expressions, \( x_1, \ldots , x_n \) distinct variables, and \( t, t_1, \ldots , t_n \) types. If \( \Delta \Gamma, (x_1 : t_1), \ldots , (x_n : t_n) \vdash e : t \) and \( \Delta \Gamma, \Gamma \vdash e_1 : t_i \) for all \( i \), then \( \Delta \Gamma, \Gamma \vdash e[e_1/x_1, \ldots , e_n/x_n] : t \).

**Proof.** By induction on the typing derivations for \( \Delta \Gamma, (x_1 : t_1), \ldots , (x_n : t_n) \vdash e : t \). We simply “plug” a copy of the derivation for \( \Delta \Gamma, \Gamma \vdash e_1 : t_i \) wherever the rule (var) is used for variable \( x_i \). For simplicity,
in what follows, we write Γ′ for Γ, (x₁: t₁), ..., (xₙ: tₙ) and ϱ for \{e₁/ x₁, ..., eₙ/ xₙ\}. We proceed by a case analysis on the last applied rule.

(const): straightforward.
(var): e = x and ΔΓ′ ⊢ x: Γ′(x).

If x = xᵢ, then Γ′(x) = tᵢ and xᵢ = eᵢ. From the premise, we have ΔΓ ⊢ eᵢ: tᵢ. The result follows.

Otherwise, Γ′(x) = Γ(x) and xᵢ = x. Clearly, we have ΔΓ ⊢ x: Γ(x). Thus the result follows as well.

(pair): consider the following derivation:

\[
\begin{array}{l}
\Delta Γ′ ⊢ e₁ : t₁ \\
\Delta Γ′ ⊢ e₂ : (t₁, t₂) \\
\hline
\Delta Γ′ ⊢ (e₁, e₂) : (t₁ × t₂)
\end{array}
\]

By applying the induction hypothesis twice, we have ΔΓ ⊢ e₁ : tᵢ. By (pair), we get ΔΓ ⊢ (e₁, e₂) : (t₁, t₂), that is, ΔΓ ⊢ (e₁, e₂) : (t₁, t₂).

(proj): consider the following derivation:

\[
\begin{array}{l}
\Delta Γ′ ⊢ e′ : t₁ × t₂ \\
\Delta Γ′ ⊢ πᵢ(e′) : tᵢ
\end{array}
\]

By induction, we have ΔΓ ⊢ e′ : t₁ × t₂. Then the rule (proj) gives us ΔΓ ⊢ πᵢ(e′) : tᵢ, that is ΔΓ ⊢ πᵢ(e′) : tᵢ.

(abstr): consider the following derivation:

\[
\begin{align*}
\forall i \in I, j \in J & \quad \Delta Γ′, (x : tᵢσ_j) ⊢ e′[σ_j] : sᵢσ_j \\
\Delta′ &= Δ \cup \text{var} (\bigwedge_{i \in I, j \in J} tᵢσ_j \rightarrow sᵢσ_j) \\
\Delta Γ′ ⊢ \lambda_{[σ_j]_{i \in I, j \in J}} x.e′ : \bigwedge_{i \in I, j \in J} tᵢσ_j \rightarrow sᵢσ_j
\end{align*}
\]

By α-conversion, we can ensure that tv(ϱ) ∩ \bigcup_{j \in J} dom(σ_j) = \emptyset. By induction, we have Δ′Γ, (x : tᵢσ_j) ⊢ e′[σ_j] : sᵢσ_j for all i ∈ I and j ∈ J. Because tv(ϱ) ∩ dom(σ_j) = \emptyset, by Lemma B.5, we get Δ′Γ, (x : tᵢσ_j) ⊢ (e′[σ_j]) : sᵢσ_j. Then by applying (abstr), we obtain ΔΓ ⊢ \lambda_{[σ_j]_{i \in I, j \in J}} x.e′ : \bigwedge_{i \in I, j \in J} tᵢσ_j → sᵢσ_j. That is, ΔΓ ⊢ (\lambda_{[σ_j]_{i \in I, j \in J}} x.e′) : \bigwedge_{i \in I, j \in J} tᵢσ_j → sᵢσ_j (because tv(ϱ) ∩ \bigcup_{j \in J} dom(σ_j) = \emptyset).

(case): consider the following derivation:

\[
\begin{array}{l}
\Delta Γ′ ⊢ e₀ : t' \\
\hline
\begin{array}{l}
t' ≤t \Rightarrow Δ Γ′ ⊢ e₁ : s \\
t' ≤t \Rightarrow Δ Γ′ ⊢ e₂ : s
\end{array}
\end{array}
\]

ΔΓ′ ⊢ (e₀ \in \? e₁ : e₂) : s.

By induction, we have ΔΓ ⊢ e₀ : t' and ΔΓ ⊢ e₁ : s (for i such that ΔΓ′ ⊢ eᵢ : s has been type-checked in the original derivation). Then the rule (case) gives us ΔΓ ⊢ (e₀ \in \? e₁ : e₂) : s that is ΔΓ ⊢ (e₀ \in \? e₁ : e₂) : s.

(inst):

\[
\begin{array}{l}
\Delta Γ′ ⊢ e′ : s \\
\hline
\Delta Γ′ ⊢ e'[σ] : s \quad \sigma \notin Δ
\end{array}
\]

Using α-conversion on the polymorphic type variables of e, we can assume tv(ϱ) \cap dom(σ) = \emptyset. By induction, we have ΔΓ ⊢ e′ : s. Since σ \notin Δ, by applying (inst) we obtain ΔΓ ⊢ (e′[σ]) : sσ, that is, ΔΓ ⊢ (e′[σ]) : sσ because tv(ϱ) \cap dom(σ) = \emptyset.

(inter):

\[
\begin{array}{l}
\forall j \in J, Δ Γ′ ⊢ e′[σ_j] : t_j \\
\hline
\Delta Γ′ ⊢ e′[σ_j]_{i \in I, j \in J} : \bigwedge_{i \in I, j \in J} t_j
\end{array}
\]

By induction, for all j ∈ J we have ΔΓ ⊢ e′[σ_j]: t_j, that is ΔΓ ⊢ (e′[σ_j]): t_j. Then by applying (inter) we get ΔΓ ⊢ (e′[σ_j])_{i \in I, j \in J}: \bigwedge_{i \in I, j \in J} t_j, that is ΔΓ ⊢ (e′[σ_j])_{i \in I, j \in J} : \bigwedge_{i \in I, j \in J} t_j.

(subsum): consider the following derivation:

\[
\begin{array}{l}
\Delta Γ′ ⊢ e′ : s \\
\hline
\Delta Γ′ ⊢ e′ : t
\end{array}
\]

By induction, we have ΔΓ ⊢ e′ : s. Then the rule (subsum) gives us ΔΓ ⊢ e′ : t.
Definition B.7. Given two typing environments \( \Gamma_1, \Gamma_2 \), we define their intersection as

\[
(\Gamma_1 \cap \Gamma_2)(x) = \begin{cases} 
\Gamma_1(x) \cap \Gamma_2(x) & \text{if } x \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

We define \( \Gamma_2 \leq \Gamma_1 \) if \( \Gamma_2(x) \leq \Gamma_1(x) \) for all \( x \in \text{dom}(\Gamma_1) \), and \( \Gamma_1 \simeq \Gamma_2 \) if \( \Gamma_1 \leq \Gamma_2 \) and \( \Gamma_2 \leq \Gamma_1 \).

Given an expression \( e \) and a set \( \Delta \) of (monomorphic) type variables, we write \( e \in \Delta \) if \( \sigma \in \Delta \) for all the type substitution \( \sigma \) that occur in a subterm of \( e \) of the form \( e'[\sigma_{j}]_{j \in J} \) (in other terms, we do not consider the substitutions that occur in the decorations of \( \lambda \)-abstractions).

Lemma B.8 (Weakening). Let \( e \) be an expression, \( \Gamma, \Gamma' \) two typing environments and \( \Delta' \) a set of type variables. If \( \Delta \Gamma \vdash e : t, \Gamma' \leq \Gamma \) and \( e \not\in \Delta' \), then \( \Delta \cup \Delta' \Gamma' \vdash e : t \).

Proof. By induction on the derivation of \( \Delta \Gamma \vdash e : t \). We perform a case analysis on the last applied rule.

(const): straightforward.

(var): \( \Delta \Gamma \vdash x : \Gamma(x) \). It is clear that \( \Delta \cup \Delta' \Gamma' \vdash x : \Gamma'(x) \) by (var). Since \( \Gamma'(x) \leq \Gamma(x) \), by (subsum), we get \( \Delta \cup \Delta' \Gamma' \vdash e : \Gamma(x) \).

(pair): consider the following derivation:

\[
\frac{\Delta \Gamma \vdash e_1 : t_1 \quad \Delta \Gamma \vdash e_2 : t_2}{\Delta \Gamma \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{(pair)}
\]

By applying the induction hypothesis twice, we have \( \Delta \cup \Delta' \Gamma' \vdash e_1 : t_1 \). Then by (pair), we get \( \Delta \cup \Delta' \Gamma' \vdash (e_1, e_2) : t_1 \times t_2 \).

(proj): consider the following derivation:

\[
\frac{\Delta \Gamma \vdash e' : t_1 \times t_2}{\Delta \Gamma \vdash \pi_i(e') : t_i} \quad \text{(proj)}
\]

By the induction hypothesis, we have \( \Delta \cup \Delta' \Gamma' \vdash e' : t_1 \times t_2 \). Then by (proj), we get \( \Delta \cup \Delta' \Gamma' \vdash \pi_i(e') : t_i \).

(abstr): consider the following derivation:

\[
\frac{\forall i \in I, j \in J. \Delta'' \Gamma, (x : t_i, \sigma_j) \vdash e' \sigma_j}{\Delta'' = \Delta \cup \text{var}(\bigwedge_{i \in I, j \in J} t_i, \sigma_j \rightarrow s_i, \sigma_j)} \quad \text{(abstr)}
\]

By induction, we have \( \Delta'' \cup \Delta' \Gamma' \vdash e' \sigma_j : s_i, \sigma_j \) for all \( i \in I \) and \( j \in J \). Then by (abstr), we get \( \Delta \cup \Delta' \Gamma' \vdash \lambda_{\sigma_j \in J} x.e' : \bigwedge_{i \in I, j \in J} t_i, \sigma_j \rightarrow s_i, \sigma_j \).

(case): consider the following derivation:

\[
\frac{\Delta \Gamma \vdash e_0 : t'}{\Delta \Gamma \vdash (e_0 \in t ? e_1 : e_2) : s} \quad \text{(case)}
\]

By induction, we have \( \Delta \cup \Delta' \Gamma' \vdash e_0 : t_0 \) and \( \Delta \cup \Delta' \Gamma' \vdash e_1 : s \) (for \( i \) such that \( e_i \) has been type-checked in the original derivation). Then by (case), we get \( \Delta \cup \Delta' \Gamma' \vdash (e_0 \in t ? e_1 : e_2) : s \).

(inst): consider the following derivation:

\[
\frac{\Delta \Gamma \vdash e' : s \quad \sigma \notin \Delta}{\Delta \Gamma \vdash e' \sigma : s} \quad \text{(inst)}
\]

By induction, we have \( \Delta \cup \Delta' \Gamma' \vdash e' : s \). Since \( e \not\in \Delta \) (i.e., \( e' \sigma \not\in \Delta' \)), we have \( \sigma \not\in \Delta' \). Then \( \sigma \not\in \Delta \cup \Delta' \). Therefore, by applying (inst) we get \( \Delta \cup \Delta' \Gamma' \vdash e' \sigma : s \).

(int): consider the following derivation:

\[
\frac{\forall j \in J. \Delta \Gamma \vdash e'[\sigma_j] : t_j}{\Delta \Gamma \vdash e'[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_j} \quad \text{(int)}
\]

By induction, we have \( \Delta \cup \Delta' \Gamma' \vdash e'[\sigma_j] : t_j \) for all \( j \in J \). Then the rule (int) gives us \( \Delta \cup \Delta' \Gamma' \vdash e'[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_j \).
(subsum): there exists a type \( s \) such that

\[
\frac{\Delta \Gamma \vdash e': s \quad s \leq t}{\Delta \Gamma' \vdash e': t} \quad \text{(subsum)}
\]

By induction, we have \( \Delta \cup \Delta' \Gamma' \vdash e' : s \). Then by applying the rule (subsum) we get \( \Delta \cup \Delta' \Gamma' \vdash e' : t \).

\[ \square \]

The next two lemmas are used to simplify sets of type-substitutions applied to expressions when they are redundant or they work on variables that are not in the expressions.

**Lemma B.9 (Useless Substitutions).** Let \( e \) be an expression and \( [\sigma_k]_{k \in K}, [\sigma'_k]_{k \in K} \) two sets of substitutions such that \( \text{dom}(\sigma'_k) \cap \text{dom}(\sigma_k) = \emptyset \) and \( \text{dom}(\sigma'_k) \cap \text{tv}(c) = \emptyset \) for all \( k \in K \). Then

\[
\Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \iff \Delta \Gamma' \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : t
\]

**Proof.** \( \Rightarrow \): The proof proceeds by induction and case analysis on the structure of \( e \). For each case we use an auxiliary internal induction on the typing derivation. We label \( \text{E} \) the main (external) induction and \( \text{I} \) the internal induction in what follows:

- \( e = c ; e \circ[\sigma_k]_{k \in K} = e \circ[\sigma_k \cup \sigma'_k]_{k \in K} = e \). The proof proceeds by induction on the typing derivation.
- Otherwise, the typing derivation ends with \( \text{I} \).
- Clearly, the typing derivation \( \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \) should end with either (\( \text{const} \)) or (\( \text{subsum} \)). Assume that the typing derivation ends with (\( \text{const} \)). Trivially, we have \( \Delta \Gamma' \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : b_k \).
- Otherwise, the typing derivation ends with an instance of (\( \text{subsum} \)).

Then by I-induction, we have \( \Delta \Gamma \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : s \). Since \( s \leq t \), the rule (\( \text{subsum} \)) gives us \( \Delta \Gamma' \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : t \).

- \( e = x \): \( e\circ[\sigma_k]_{k \in K} = e\circ[\sigma_k \cup \sigma'_k]_{k \in K} = x \). Clearly, the typing derivation \( \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \) should end with either (\( \text{var} \)) or (\( \text{subsum} \)). Assume that the typing derivation ends with (\( \text{var} \)). Trivially, we have \( \Delta \Gamma' \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : \Gamma(x) \).
- Otherwise, the typing derivation ends with an instance of (\( \text{subsum} \)), similarly to the case of \( e = c \), the result follows by I-induction.

- \( e = (e_1, e_2) ; e\circ[\sigma_k]_{k \in K} = (e_1 \circ[\sigma_k]_{k \in K}, e_2 \circ[\sigma_k]_{k \in K}) \) and \( e\circ[\sigma_k \cup \sigma'_k]_{k \in K} = (e_1 \circ[\sigma_k \cup \sigma'_k]_{k \in K}, e_2 \circ[\sigma_k \cup \sigma'_k]_{k \in K}) \). Clearly, the typing derivation \( \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \) should end with either (\( \text{pair} \)) or (\( \text{subsum} \)). Assume that the derivation ends with an instance of (\( \text{pair} \)):

\[
\Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : s_1 \quad \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : s_2
\]

By \( \text{E} \)-induction, we have \( \Delta \Gamma \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : s_1 \times s_2 \). Then the rule (\( \text{pair} \)) gives us

\[
\Delta \Gamma' \vdash (e_1 \circ[\sigma_k \cup \sigma'_k]_{k \in K}, e_2 \circ[\sigma_k \cup \sigma'_k]_{k \in K}) : (s_1 \times s_2)
\]

that is, \( \Delta \Gamma' \vdash (e_1, e_2)\circ[\sigma_k \cup \sigma'_k]_{k \in K} : (s_1 \times s_2) \).

- Otherwise, the typing derivation ends with an instance of (\( \text{subsum} \)), similarly to the case of \( e = c \), the result follows by I-induction.

- \( e = \pi_1(e_0) ; e\circ[\sigma_k]_{k \in K} = \pi_1(e_0\circ[\sigma_k]_{k \in K}) \) and \( e\circ[\sigma_k \cup \sigma'_k]_{k \in K} = \pi_1(e_0\circ[\sigma_k \cup \sigma'_k]_{k \in K}) \). Clearly, the typing derivation \( \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \) should end with either (\( \text{proj} \)) or (\( \text{subsum} \)). Assume that the derivation ends with an instance of (\( \text{proj} \)):

\[
\Delta \Gamma \vdash e_0\circ[\sigma_k]_{k \in K} : t_1 \times t_2
\]

By \( \text{E} \)-induction, we have \( \Delta \Gamma \vdash \pi_1(e_0\circ[\sigma_k \cup \sigma'_k]_{k \in K}) : t \). Then the rule (\( \text{proj} \)) gives us \( \Delta \Gamma' \vdash \pi_1(e_0\circ[\sigma_k \cup \sigma'_k]_{k \in K}) : t \), that is \( \Delta \Gamma' \vdash \pi_1(e_0)\circ[\sigma_k \cup \sigma'_k]_{k \in K} : t \).

- Otherwise, the typing derivation ends with an instance of (\( \text{subsum} \)), similarly to the case of \( e = c \), the result follows by I-induction.

- \( e = e_1 e_2 ; e\circ[\sigma_k]_{k \in K} = (e_1 \circ[\sigma_k]_{k \in K})(e_2 \circ[\sigma_k]_{k \in K}) \) and \( e\circ[\sigma_k \cup \sigma'_k]_{k \in K} = (e_1 \circ[\sigma_k \cup \sigma'_k]_{k \in K})(e_2 \circ[\sigma_k \cup \sigma'_k]_{k \in K}) \). Clearly, the typing derivation \( \Delta \Gamma \vdash e\circ[\sigma_k]_{k \in K} : t \) should end with either (\( \text{app} \)) or (\( \text{subsum} \)). Assume that the derivation ends with an instance of (\( \text{app} \)).

\[
\Delta \Gamma \vdash (e_1 \circ[\sigma_k]_{k \in K}, e_2 \circ[\sigma_k]_{k \in K}) : t
\]

By \( \text{E} \)-induction, we have \( \Delta \Gamma \vdash ((e_1 \circ[\sigma_k \cup \sigma'_k]_{k \in K}, e_2 \circ[\sigma_k \cup \sigma'_k]_{k \in K})) : t \). Then the rule (\( \text{app} \)) gives us \( \Delta \Gamma' \vdash e\circ[\sigma_k \cup \sigma'_k]_{k \in K} : t \), that is \( \Delta \Gamma' \vdash (e_1 e_2)\circ[\sigma_k \cup \sigma'_k]_{k \in K} : t \).

- Otherwise, the typing derivation ends with an instance of (\( \text{subsum} \)), similarly to the case of \( e = c \), the result follows by I-induction.
ends with an instance of (appl):

\[
\Delta \vdash e_1 [\sigma_1]_{k \in K} : t \\
\Delta \vdash e_2 [\sigma_2]_{k \in K} : s
\]

(appl)

By applying the E-induction hypothesis twice, we have \(\Delta \vdash e_1 [\sigma_1 \cup \sigma'_1]_{k \in K} : t \rightarrow s\) and \(\Delta \vdash e_2 [\sigma_2 \cup \sigma'_2]_{k \in K} : t\). Then the rule (appl) gives us \(\Delta \vdash (e_1 \circ e_2) [\sigma_1 \cup \sigma'_1]_{k \in K} : s\). Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = e\), the result follows by I-induction.

\[
e = \lambda^{i_1 \rightarrow s_2}_{j_1 \rightarrow s_3} \cdot x.e_0; e_0[\sigma_k]_{k \in K} = \lambda^{i_1 \rightarrow s_2}_{j_1 \rightarrow s_3} \cdot x.e_0 \text{ and } e_0[\sigma_k \cup \sigma'_k]_{k \in K} = \lambda^{i_1 \rightarrow s_2}_{j_1 \rightarrow s_3} \cdot x.e_0 \text{. Clearly, the typing derivation } \Delta \vdash e \text{ should end with either (abstr) or (subsum). Assume that the derivation ends with an instance of (abstr):}
\]

\[
\forall i \in I, j \in J, k \in K. \Delta' \vdash (x : (t_i \sigma_j) \sigma_k) \vdash (e_0 \sigma_j) [\sigma_k] : (s_i \sigma_j) \sigma_k
\]

By E-induction, we have

\[
\Delta' \vdash (x : (t_i \sigma_j) \sigma_k) \vdash (e_0[\sigma_j] \cup \sigma'_k) : (s_i \sigma_j) \sigma_k
\]

Since \(\text{dom}(\sigma'_k) \cap \text{tv}(e) = \emptyset\), we have \(\text{var}(t_i \sigma_j) \cap \text{dom}(\sigma'_k) = \emptyset\) and \(\text{var}(s_i \sigma_j) \cap \text{dom}(\sigma'_k) = \emptyset\).

And then \(\text{var}(t_i \sigma_j) \sigma_k = (t_i \sigma_j)(\sigma_k \cup \sigma'_k)\) and

\[
\text{var}(s_i \sigma_j) \sigma_k = (s_i \sigma_j)(\sigma_k \cup \sigma'_k)
\]

Let \(\Delta'' = \Delta \cup \text{var}(\bigwedge_{i \in I, j \in J, k \in K} t_i(\sigma_j \circ (\sigma_k \cup \sigma'_k)) \rightarrow s_i(\sigma_j \circ (\sigma_k \cup \sigma'_k)))\). Then \(\Delta'' = \Delta'\). Therefore, we have

\[
\Delta'' \vdash (x : (t_i \sigma_j)(\sigma_k \cup \sigma'_k)) \vdash (e_0 \sigma_j) [\sigma_k \cup \sigma'_k] : (s_i \sigma_j)(\sigma_k \cup \sigma'_k)
\]

that is,

\[
\Delta'' \vdash (x : (t_i \sigma_j)(\sigma_k \cup \sigma'_k)) \vdash (e_0 \sigma_j) [\sigma_k \cup \sigma'_k] : (s_i \sigma_j)(\sigma_k \cup \sigma'_k)
\]

Then by (abstr), we have

\[
\Delta \vdash \lambda^{i_1 \rightarrow s_2}_{j_1 \rightarrow s_3} \cdot x.e_0 \cdot \bigwedge_{i \in I, j \in J, k \in K} t_i(\sigma_j \circ (\sigma_k \cup \sigma'_k)) \rightarrow s_i(\sigma_j \circ (\sigma_k \cup \sigma'_k))
\]

Moreover, we also have

\[
\bigwedge_{i \in I, j \in J, k \in K} t_i(\sigma_j \circ (\sigma_k \cup \sigma'_k)) \rightarrow s_i(\sigma_j \circ (\sigma_k \cup \sigma'_k))
\]

Then by (subsum), the result follows.

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = e\), the result follows by I-induction.

\[
e = (c_0 \in t \parallel e_1 : e_2) \vdash (e_0 \sigma_j) [\sigma_k]_{k \in K} \in e_0[\sigma_k]_{k \in K} \in t \parallel e_1 \parallel e_2 \parallel [\sigma_k]_{k \in K}
\]

and

\[
e_0[\sigma_k \cup \sigma'_k]_{k \in K} \in (c_0 \in t \parallel e_1 : e_2) \parallel e_0 \parallel e_1[\sigma_k \cup \sigma'_k]_{k \in K} \parallel e_2[\sigma_k \cup \sigma'_k]_{k \in K}
\]

Clearly, the typing derivation \(\Delta \vdash e \in t\) should end with either (case) or (subsum). Assume that the derivation ends with an instance of (case):

\[
\begin{align*}
\Delta \vdash e_1[\sigma_k]_{k \in K} : s \\
\Delta \vdash e_2[\sigma_k]_{k \in K} : s
\end{align*}
\]

(case)
By E-induction, we have $\Delta \Gamma \vdash e_0[\sigma_k \cup \sigma_k'] : t'$ and $\Delta \Gamma \vdash e_1[\sigma_k \cup \sigma_k'] : s$ (for $i$ such that $\Delta \Gamma \vdash e_i[\sigma_k \cup \sigma_k'] : s$ has been type-checked in the original derivation). Then the rule (case) gives us

$$
\Delta \Gamma \vdash (e_0 \circ \sigma_k \cup \sigma_k')_{k \in K} \in t \wedge e_1 \circ \sigma_k \cup \sigma_k'_{k \in K} : s \quad (\text{for } i \text{ such that } \Delta \Gamma \vdash e_i[\sigma_k \cup \sigma_k'] : s)
$$

that is $\Delta \Gamma \vdash (e_0 \in t \wedge e_1 \circ \sigma_k \cup \sigma_k'_{k \in K} : s)$. Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of $e = c$, the result follows by I-induction.

$$
e = c(\sigma_j)_{j \in J} \in x \circ \sigma_k \cup \sigma_k'_{k \in K} = c \in x \circ \sigma_k \cup \sigma_k'_{k \in K} = c. \text{ Similar to the case } e = c.
$$

Then, $e = (e_1, e_2)(\sigma_j)_{j \in J}$ similar to the case $e = (e_1, e_2)$ where we by E-induction on $e_1[\sigma_j]_{j \in J}$ (since (for simplicity, we only consider $\lambda e_1 \in t \in t$)

$$
e = \pi_1(e_0) \circ \sigma_k : t \quad \text{similar to the case } e = \pi_1(e_0) \text{ where we by E-induction on } e_0[\sigma_j]_{j \in J}.
$$

Then, $e = (e_1, e_2)(\sigma_j)_{j \in J}$ similar to the case $e = e_1e_2$ where we by E-induction on $e_1[\sigma_j]_{j \in J}$. 

$$
e = (\lambda \sigma_j \in t \rightarrow s_i \circ \sigma_k \cup \sigma_k')_{j \in J} \in x \circ \sigma_k \cup \sigma_k'_{k \in K} = \lambda \sigma_j \in t \rightarrow s_i \circ \sigma_k \cup \sigma_k'_{j \in J} \circ \sigma_k \cup \sigma_k'_{k \in K} \in x \circ e_0. \text{ Similar to the case } e = \lambda \sigma_j \in t \rightarrow s_i \circ \sigma_k \cup \sigma_k'_{j \in J} \in x \circ e_0.
$$

Thus the typing derivation ends with an instance of (abstr), while the other cases are similar to the corresponding ones in the proof of $\Rightarrow$. The situation is as follows:

$$
\Delta' = \Delta \cup \text{var}(\sigma_j \in t \circ \sigma_k \cup \sigma_k')_{j \in J, k \in K} : t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k')
$$

$$\forall i \in I, j \in J, k \in K:
$$

$$
\Delta \Gamma, (x : t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k')) \vdash e_0[\sigma_j \circ \sigma_k \cup \sigma_k'] : s_i \circ (\sigma_j \circ \sigma_k \cup \sigma_k')
$$

From the premise, we have

$$
\Delta' \Gamma, (x : t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k')) : (e_0[\sigma_j \circ \sigma_k \cup \sigma_k'] : s_i) \circ (\sigma_j \circ \sigma_k \cup \sigma_k')
$$

By E-induction, we get

$$
\Delta' \Gamma, (x : t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k') : (e_0[\sigma_j \circ \sigma_k \cup \sigma_k'] : s_i) \circ (\sigma_j \circ \sigma_k \cup \sigma_k')
$$

Moreover, as $\text{dom}(\sigma_j) \cap \text{tv}(e) = \emptyset$, we have $\text{var}(t_i \circ \sigma_j \circ \text{dom}(\sigma_k') = \emptyset$ and $\text{var}(s_i \circ \sigma_j \circ \text{dom}(\sigma_k') = \emptyset$. And then $t_i \circ \sigma_j \circ \sigma_k = (t_i \circ \sigma_j \circ \sigma_k') \circ (s_i \circ \sigma_j \circ \sigma_k') = (s_i \circ \sigma_j \circ \sigma_k')$ and

$$
\text{var}(\sigma_j \in t \circ \sigma_k \cup \sigma_k')
$$

Then, $\Delta'' = \Delta \cup \text{var}(\sigma_j \in t \circ \sigma_k \cup \sigma_k') : t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k')$. Then $\Delta'' = \Delta$. Therefore, we have

$$
\Delta'' \Gamma, (x : t_i \circ \sigma_j \circ \sigma_k') : (e_0[\sigma_j \circ \sigma_k] : s_i \circ \sigma_j \circ \sigma_k)
$$

that is

$$
\Delta'' \Gamma, (x : t_i \circ \sigma_j \circ \sigma_k) : e_0[\sigma_j \circ \sigma_k] : s_i \circ \sigma_j \circ \sigma_k)
$$

Then, by (abstr), we have

$$
\Delta \Gamma \vdash \lambda \sigma_j \in t \rightarrow s_i \circ \sigma_k \cup \sigma_k'_{j \in J} \circ x \circ e_0 : \bigwedge_{i \in I, j \in J, k \in K} t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k')
$$

Finally, since

$$
\bigwedge_{i \in I, j \in J, k \in K} t_i \circ \sigma_j \circ \sigma_k \cup \sigma_k') \Rightarrow s_i \circ \sigma_j \circ \sigma_k)
$$

29

2012/7/11
by subsumption, the result follows.

□

Let “∪” denotes the union of multi-set, for instance, \(\{1, 2\} \cup \{1, 3\} = \{1, 2, 1, 3\}\).

**Lemma B.10 (Redundant Substitutions).** Let \([\sigma_j]_{j \in J}\) and \([\sigma_j']_{j \in J'}\) be two sets of substitutions such that \(J' \subseteq J\). Then

\[
\Delta_3 \Gamma \vdash e \circ [\sigma_j]_{j \in J \cup J'} : t \iff \Delta_3 \Gamma \vdash e \circ [\sigma_j]_{j \in J} : t
\]

**Proof.** \(\Rightarrow\): The proof proceeds by induction and case analysis on the structure of \(e\). For each case we use an auxiliary internal induction on the typing derivation. We label E the main (external) induction and I the internal induction in what follows:

\[
e = c_1; e_1[\sigma_j]_{j \in J \cup J'} = e_1[\sigma_j]_{j \in J} = c.
\]

Clearly, the typing derivation \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t\) should end with either (const) or (subsum). Assume that the typing derivation ends with (const). Trivially, we have \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J} : b_c\).

Otherwise, the typing derivation ends with an instance of (subsum):

\[
\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : s \quad s \leq t
\]

Then by I-induction, we have \(\Delta_3 \Gamma \vdash [\sigma_j]_{j \in J} : s\). Since \(s \leq t\), the rule (subsum) gives us \(\Delta_3 \Gamma \vdash [\sigma_j]_{j \in J} : t\).

\[
e = x; e_1[\sigma_j]_{j \in J \cup J'} = e_1[\sigma_j]_{j \in J} = x.
\]

Clearly, the typing derivation \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t\) should end with either (var) or (subsum). Assume that the typing derivation ends with (var). Trivially, we have \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J} : b_c\).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = c\), the result follows by I-induction.

\[
e = (e_1,e_2); e_1[\sigma_j]_{j \in J \cup J'} = (e_1, [\sigma_j]_{j \in J \cup J'})\text{ and } e_2[\sigma_j]_{j \in J \cup J'} = (e_2, [\sigma_j]_{j \in J \cup J'})\text{.}
\]

Clearly, the typing derivation \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t\) should end with either (pair) or (subsum). Assume that the derivation ends with an instance of (pair):

\[
\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : s_1 \quad \Delta_3 \Gamma \vdash e_2[\sigma_j]_{j \in J \cup J'} : s_2
\]

By applying the E-induction hypothesis twice, we have \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J} : s_1\). Then the rule (pair) gives us \(\Delta_3 \Gamma \vdash (e_1, e_2)[\sigma_j]_{j \in J} : (s_1 \times s_2)\), that is \(\Delta_3 \Gamma \vdash (e_1, e_2)[\sigma_j]_{j \in J} : (s_1 \times s_2)\).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = c\), the result follows by I-induction.

\[
e = \pi(x'; e_1'); e_1[\sigma_j]_{j \in J \cup J'} = \pi(x'; [\sigma_j]_{j \in J \cup J'})\text{ and } e_1[\sigma_j]_{j \in J} = \pi(x'; [\sigma_j]_{j \in J})\text{.}
\]

Clearly, the typing derivation \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t\) should end with either (proj) or (subsum). Assume that the derivation ends with an instance of (proj):

\[
\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t_1 \times t_2
\]

By E-induction, we have \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J} : t_1 \times t_2\). Then by (proj) we get \(\Delta_3 \Gamma \vdash \pi(x'; e_1')[\sigma_j]_{j \in J} : t_1\). That is \(\Delta_3 \Gamma \vdash \pi(x'; e_1')[\sigma_j]_{j \in J} : t_1\).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = c\), the result follows by I-induction.

\[
e = e_1; e_2[\sigma_j]_{j \in J \cup J'} = (e_1, [\sigma_j]_{j \in J \cup J'})\text{ and } e_2[\sigma_j]_{j \in J \cup J'} = (e_2, [\sigma_j]_{j \in J \cup J'})\text{.}
\]

Clearly, the typing derivation \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t\) should end with either (appl) or (subsum). Assume that the derivation ends with an instance of (appl):

\[
\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J \cup J'} : t \rightarrow s
\]

By applying the E-induction hypothesis twice, we have \(\Delta_3 \Gamma \vdash e_1[\sigma_j]_{j \in J} : t \rightarrow s\) and \(\Delta_3 \Gamma \vdash e_2[\sigma_j]_{j \in J} : t\). Then the rule (appl) gives us \(\Delta_3 \Gamma \vdash (e_1, e_2)[\sigma_j]_{j \in J} : s\), that is \(\Delta_3 \Gamma \vdash (e_1, e_2)[\sigma_j]_{j \in J} : s\).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \(e = c\), the result follows by I-induction.
end with either (abstr) or (subsum). Assume that the derivation ends with an instance of (abstr):

\[
\forall i \in I, k \in K, j \in J \cup J', \Delta' \vdash \Gamma, (x : t_i(\sigma_k \circ \sigma_j)) \vdash e' \alpha \sigma_k \circ \sigma_j : s_i(\sigma_k \circ \sigma_j)
\]

\[
\Delta' = \Delta \cup \text{var}(\land_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j))
\]

\[
\Delta \vdash \lambda_{[\sigma_k \in K][\sigma_j \in J]} x.e : \land_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

Since \( J' \subseteq J \), it is clear that

\[
\land_{i \in I, k \in K, j \in J, j' \in J'} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

and

\[
\text{var}(\land_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j))
\]

Let \( \Delta'' = \Delta \cup \text{var}(\land_{i \in I, k \in K, j \in J'} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)) \). Then \( \Delta'' = \Delta' \). Consider the premises where \( i \in I, k \in K \) and \( j \in J \). By applying (abstr), we have

\[
\Delta \vdash \lambda_{[\sigma_k \in K][\sigma_j \in J]} x.e : \land_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

that is,

\[
\Delta \vdash e \alpha \sigma_k \circ \sigma_j : \land_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

Then by (subsum), the result follows.

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \( e = c \), the result follows by I-induction.

\[
e = (e_0 \text{ t } e_1 : e_2)
\]

\[
e \alpha \sigma_j : \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

and

\[
e \alpha \sigma_j = (e_0 \alpha \sigma_j) \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j).
\]

Clearly, the typing derivation \( \Delta \vdash e \alpha \sigma_j : t \) should end with either (case) or (subsum). Assume that the derivation ends with an instance of (case):

\[
\Delta \vdash e \alpha \sigma_j : \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

By E-induction, we have \( \Delta \vdash e \alpha \sigma_j : t' \) and \( \Delta \vdash e \alpha \sigma_j : s \) (for \( s \) such that \( \Delta \vdash e \alpha \sigma_j : s \) has been type-checked in the original derivation). Then the rule (case) gives us

\[
\Delta \vdash e \alpha \sigma_j : \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

that is \( \Delta \vdash (e_0 \text{ t } e_1 : e_2) \alpha \sigma_j : s \).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \( e = c \), the result follows by I-induction.

\[
e = e' \alpha \sigma_j : e \alpha \sigma_j = e' \alpha \sigma_j \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)
\]

\[
s = e' \alpha \sigma_j : t = e' \alpha \sigma_j \land_{i \in I} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j).
\]

\[
\Delta \vdash e' \alpha \sigma_j : t
\]

\[
\Delta \vdash e \alpha \sigma_j : s
\]

\[
e \alpha \sigma_j : t
\]

\[
e \alpha \sigma_j : s
\]

\[
\Delta \vdash e \alpha \sigma_j : t
\]

\[
\Delta \vdash e \alpha \sigma_j : s
\]

\[
\Delta \vdash e \alpha \sigma_j : t
\]

\[
\Delta \vdash e \alpha \sigma_j : s
\]

\[
\Delta \vdash e \alpha \sigma_j : t
\]

\[
\Delta \vdash e \alpha \sigma_j : s
\]

\[
\Delta \vdash e \alpha \sigma_j : t
\]

\[
\Delta \vdash e \alpha \sigma_j : s
\]
Let \( \Delta'' = \Delta \cup \var\{\lambda_{i \in I, k \in K, j \in J'} t_i (\sigma_k \circ \sigma_j) \rightarrow s_i (\sigma_k \circ \sigma_j)\} \). Then \( \Delta'' = \Delta'. \) Let us construct (by duplication) the following typing judgments from the premises of the derivation:

\[
\forall i \in I, k \in K, j \in J', \Delta'' \vdash \Gamma (x : t_i (\sigma_k \circ \sigma_j)) \vdash e' \circ [\sigma_k \circ \sigma_j] : s_i (\sigma_k \circ \sigma_j)
\]

Then by applying (\textit{abstr}), we have

\[
\Delta'' \vdash x : \bigwedge_{i \in I, k \in K, j \in J'} t_i (\sigma_k \circ \sigma_j) \rightarrow s_i (\sigma_k \circ \sigma_j)
\]

that is,

\[
\Delta'' \vdash e' \circ [\sigma_j]_{j \in J'} : \bigwedge_{i \in I, k \in K, j \in J'} t_i (\sigma_k \circ \sigma_j) \rightarrow s_i (\sigma_k \circ \sigma_j)
\]

Then by (\textit{subsum}), the result follows.

\[\square\]

Lemma B.9 states that if a type variable \( \alpha \) in the domain of a type substitution \( \sigma \) does not occur in the applied expression \( e \), namely, \( \alpha \in \text{dom}(\sigma) \setminus \text{tv}(e) \), that part of the substitution is useless and can be safely eliminated. Lemma B.10 states that although our \( [\sigma_j]_{j \in J} \) are formally multisets of type-substitutions, in practice they behave as sets, since repeated entries of type substitutions can be safely removed. Therefore, to simplify an expression without altering its type (and semantics), we first eliminate the useless type variables, yielding concise type substitutions, and then remove the redundant type substitutions. This is useful especially for the translation from our calculus to the core C. Due as we need less type-case branches to encode abstractions (see Section F for more detail). It also explains why we do not apply relabeling when the domains of the type substitutions do not contain type variables in expressions in Definition A.12.

Moreover, Lemma B.10 also indicates that it is safe to keep only the type substitutions which are different from each other when we merge two sets of substitutions (e.g. Lemmas B.13 and B.14). In what follows, without explicit mention, we assume that there are no useless type variables in the domain of any type substitution and no redundant type substitutions in any set of type substitutions.

**Lemma B.11** (Relabeling). Let \( e \) be an expression, \([\sigma_j]_{j \in J}\) a set of type substitutions and \( \Delta \) a set of type variables such that \( \sigma_j \notin \Delta \) for all \( j \in J \). If \( \Delta'' \vdash e : t \), then

\[
\Delta'' \vdash e \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j
\]

**Proof.** The proof proceeds by induction and case analysis on the structure of \( e \). For each case we use an auxiliary internal induction on the typing derivation. We label \( E \) the main (external) induction and \( I \) the internal induction in what follows.

\( e = c; \) the typing derivation \( \Delta'' \vdash e : t \) should end with either (\textit{const}) or (\textit{subsum}). Assume that the typing derivation ends with (\textit{const}). Trivially, we have \( \Delta'' \vdash e : b_c \). Since \( c \circ [\sigma_j]_{j \in J} = c \) and \( b_c \simeq \bigwedge_{j \in J} b_c \sigma_j \), by subsumption, we have \( \Delta'' \vdash e \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} b_c \sigma_j \).

Otherwise, the typing derivation ends with an instance of (\textit{subsum}):

\[
\Delta'' \vdash e : s \leq t \quad \Delta'' \vdash e : t \quad (\text{subsum})
\]

Then by \( I \)-induction, we have \( \Delta'' \vdash e \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} s \sigma_j \). Since \( s \leq t \), we get \( \bigwedge_{j \in J} s \sigma_j \leq \bigwedge_{j \in J} t \sigma_j \). Then by applying the rule (\textit{subsum}), we have \( \Delta'' \vdash e \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \).

\( e = \lambda \); the typing derivation \( \Delta'' \vdash e : t \) should end with either (\textit{var}) or (\textit{subsum}). Assume that the typing derivation ends with (\textit{var}). Trivially, by (\textit{var}), we get \( \Delta'' \vdash e : \Gamma (x) \). Moreover, we have \( x \circ [\sigma_j]_{j \in J} = x \), and \( \Gamma (x) = \bigwedge_{j \in J} \Gamma (x) \sigma_j \) (as \( \var (\Gamma) \subseteq \Delta \)). Therefore, we deduce that \( \Delta'' \vdash e \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} \Gamma (x) \sigma_j \).

Otherwise, the typing derivation ends with an instance of (\textit{subsum}), similarly to the case of \( e = c \), the result follows by \( I \)-induction.

\( e = (e_1, e_2); \) the typing derivation \( \Delta'' \vdash e : t \) should end with either (\textit{pair}) or (\textit{subsum}). Assume that the typing derivation ends with (\textit{pair}):

\[
\Delta'' \vdash e_1 : t_1 \quad \Delta'' \vdash e_2 : t_2 \quad (\text{pair})
\]

By \( E \)-induction, we have \( \Delta'' \vdash (e_1 \circ [\sigma_j]_{j \in J}, e_2 \circ [\sigma_j]_{j \in J}) : \bigwedge_{j \in J} t_1 \sigma_j \). Then by (\textit{pair}), we get \( \Delta'' \vdash (e_1 \circ [\sigma_j]_{j \in J}, e_2 \circ [\sigma_j]_{j \in J}) : (\bigwedge_{j \in J} t_1 \sigma_j \times \bigwedge_{j \in J} t_2 \sigma_j) \), that is, \( \Delta'' \vdash (e_1, e_2) \circ [\sigma_j]_{j \in J} : \bigwedge_{j \in J} (t_1 \times t_2) \sigma_j \).

Otherwise, the typing derivation ends with an instance of (\textit{subsum}), similarly to the case of \( e = c \), the result follows by \( I \)-induction.
\[ e = \pi_i (e') : \] the typing derivation \( \Delta \Gamma \vdash e : t \) should end with either (proj) or (subsum). Assume that the typing derivation ends with (proj):

\[ \Delta \Gamma \vdash e' : t_1 \times t_2 \]

\[ \Delta \Gamma \vdash \pi_i (e') : t_i \]  \hspace{1em} (proj)

By E-induction, we have \( \Delta \Gamma \vdash e' \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} (t_1 \times t_2) \sigma_j \), that is, \( \Delta \Gamma \vdash e' \mathbin{\bullet} [\sigma_j]_{j \in J} : (\bigwedge_{j \in J} t_1 \sigma_j \times \bigwedge_{j \in J} t_2 \sigma_j) \). Then the rule (proj) gives us that \( \Delta \Gamma \vdash \pi_i (e' \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_i \sigma_j \), that is, \( \Delta \Gamma \vdash e \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_i \sigma_j \).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \( e = e \), the result follows by I-induction.

\[ e = e_1 e_2 : \] the typing derivation \( \Delta \Gamma \vdash e : t \) should end with either (appl) or (subsum). Assume that the typing derivation ends with (appl):

\[ \Delta \Gamma \vdash e_1 : t \rightarrow s \]

\[ \Delta \Gamma \vdash e_2 : s \]  \hspace{1em} (pair)

By E-induction, we have \( \Delta \Gamma \vdash e_1 \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} (t \rightarrow s) \sigma_j \) and \( \Delta \Gamma \vdash e_2 \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \). Since \( \bigwedge_{j \in J} (t \rightarrow s) \sigma_j \subseteq (\bigwedge_{j \in J} t \sigma_j) \rightarrow (\bigwedge_{j \in J} s \sigma_j) \), by (subsum), we have \( \Delta \Gamma \vdash e \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} (t \sigma_j) \rightarrow (\bigwedge_{j \in J} s \sigma_j) \). Then by (appl), we get

\[ \Delta \Gamma \vdash (e_1 \mathbin{\bullet} [\sigma_j]_{j \in J}) (e_2 \mathbin{\bullet} [\sigma_j]_{j \in J}) : \bigwedge_{j \in J} s \sigma_j \]

that is, \( \Delta \Gamma \vdash (e_1 e_2) \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} s \sigma_j \).

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \( e = e \), the result follows by I-induction.

\[ e = \lambda^{i \in I : t_i \rightarrow s_i}_{\sigma_k} x.e' : \] the typing derivation \( \Delta \Gamma \vdash e : t \) should end with either (abstr) or (subsum).

Assume that the typing derivation ends with (abstr):

\[ \forall i \in I, k \in K, \Delta' = \Delta \cup \text{var}(\bigcup_{i\in I, k\in K} t_i \sigma_k \rightarrow s_i \sigma_k) \]

\[ \Delta \Gamma \vdash \lambda^{i \in I : t_i \rightarrow s_i}_{\sigma_k} x.e' : \bigwedge_{i \in I, k \in K} t_i \sigma_k \rightarrow s_i \sigma_k \]  \hspace{1em} (abstr)

Using \( \alpha \)-conversion, we can assume that \( \sigma_j \notin (\text{var}(\bigcup_{i \in I, k \in K} t_i \sigma_k \rightarrow s_i \sigma_k) \setminus \Delta) \) for \( j \in J \). Hence \( \sigma_j \notin \Delta' \). By E-induction, we have

\[ \Delta' \Gamma, (x : t_i \sigma_k) \vdash (e' \mathbin{\bullet} [\sigma_k] \mathbin{\bullet} [\sigma_j]) : (s_i \sigma_k) \sigma_j \]

for all \( i \in I, k \in K \) and \( j \in J \).

By Lemma B.4 \( (e' \mathbin{\bullet} [\sigma_k] \mathbin{\bullet} [\sigma_j]) = e' \mathbin{\bullet} (\sigma_k \circ [\sigma_j]) \). So

\[ \Delta' \Gamma, (x : t_i \sigma_k) \vdash e' \mathbin{\bullet} (\sigma_k \circ [\sigma_j]) : (s_i \sigma_k) \sigma_j \]

Finally, by (abstr), we get

\[ \Delta \Gamma \vdash \lambda^{i \in I : t_i \rightarrow s_i}_{\sigma_k} x.e' : \bigwedge_{i \in I, k \in K, j \in J} t_i \sigma_k \circ s_i \sigma_k \rightarrow s_i \sigma_k \circ \sigma_j \]

that is,

\[ \Delta \Gamma \vdash \lambda^{i \in I : t_i \rightarrow s_i}_{\sigma_k} x.e' \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} (\bigwedge_{i \in I, k \in K} t_i \sigma_k \rightarrow s_i \sigma_k) \sigma_j \]

Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of \( e = e \), the result follows by I-induction.

\[ e = e' e \vdash e_1 : e_2 : \] the typing derivation \( \Delta \Gamma \vdash e : t \) should end with either (case) or (subsum). Assume that the typing derivation ends with (case):

\[ \Delta \Gamma \vdash e' : t' \]

\[ \{ t' \not\leq t \Rightarrow \Delta \Gamma \vdash e_1 : s \}

\[ t' \not\leq t \Rightarrow \Delta \Gamma \vdash e_2 : s \]  \hspace{1em} (case)

By E-induction, we have \( \Delta \Gamma \vdash e' \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} t' \sigma_j \). Suppose \( \bigwedge_{j \in J} t' \sigma_j \not\leq t \); then we must have \( t' \not\leq t \), and the branch for \( e_1 \) has been type-checked. By the E-induction hypothesis, we have \( \Delta \Gamma \vdash e_1 \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} s_1 \sigma_j \). Similarly, if \( \bigwedge_{j \in J} t' \sigma_j \leq t \), then the second branch \( e_2 \) has been type-checked, and we have \( \Delta \Gamma \vdash e_2 \mathbin{\bullet} [\sigma_j]_{j \in J} : \bigwedge_{j \in J} s_2 \sigma_j \) by the E-induction hypothesis. By (case),
Lemma B.13. If \( e = e'[σ] : t \) and \( \Delta_3 Γ' \vdash e'[σ'][J] : t' \), then \( \Delta_3 Γ' \vdash e'[σ][J,J'] : t \land t' \).

Proof. Immediate consequence of Lemma B.11.

Corollary B.12. If \( \Delta_3 Γ \vdash e[σ][J] : t \), then \( \Delta_3 Γ \vdash e[σ][J] : t \).

Proof. Immediate consequence of Lemma B.11.
Proof. The proof proceeds by induction and case analysis on the structure of $e$. For each case we use an auxiliary internal induction on both typing derivations. We label $E$ the main (external) induction and $I$ the internal induction in what follows.

$e = c; e'\sigma_j, e_j \in J = e'\sigma_j, e_j \in J' = c$. Clearly, both typing derivations should end with either (const) or (subsum). Assume that both derivations end with (const):

$$\Delta \Gamma \vdash e : b_c \quad \Delta' \Gamma' \vdash e : b_c \quad \text{(const)}$$

Trivially, by (const) we have $\Delta \cup \Delta' \Delta' \Gamma \vdash e : b_c$, that is $\Delta \cup \Delta' \Delta' \Gamma' \vdash e \sigma_j, e_j \in J, J' : b_c$. As $b_c \simeq b_c$, the result follows.

Otherwise, there exists at least one typing derivation which ends with an instance of (subsum), for instance,

$$\Delta \Gamma \vdash e \sigma_j, e_j \in J : s \quad \Delta' \Gamma' \vdash e \sigma_j, e_j \in J : t \quad \text{(subsum)}$$

Then by $I$-induction on $\Delta \Gamma \vdash e \sigma_j, e_j \in J : s$ and $\Delta' \Gamma' \vdash e \sigma_j, e_j \in J' : t'$, we have $\Delta \cup \Delta' \Delta' \Gamma' \vdash e \sigma_j, e_j \in J, J' : s \land t'$.

Since $s \leq t$, we have $s \land t' \leq t \land t'$. By (subsum), the result follows as well.

$e = s; e'\sigma_j, e_j \in J = e'\sigma_j, e_j \in J'$ and $e'\sigma_j, e_j \in J' = e\sigma_j, e_j \in J$. Clearly, both typing derivations should end with either (var) or (subsum). Assume that both derivations end with an instance of (var):

$$\Delta \Gamma \vdash e \sigma_j, e_j \in J : x(\Gamma(x)) \quad \Delta' \Gamma' \vdash e \sigma_j, e_j \in J : x(\Gamma'(x)) \quad \text{(var)}$$

Since $x \in \text{dom}(\Gamma)$ and $x \in \text{dom}(\Gamma')$, $x \in \text{dom}(\Delta' \Delta' \Gamma' \land \Gamma)$. By (var), we have $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e \sigma_j, e_j \in J, J' : x \land \Delta' \Delta' \Gamma' \land \Gamma' \vdash e \sigma_j, e_j \in J, J' : x$. Otherwise, there exists at least one typing derivation which ends with an instance of (subsum), similarly to the case of $e = c$, the result follows by $I$-induction.

$e = (e_1, e_2); e_1\sigma_j, e_j \in J = e_2\sigma_j, e_j \in J'$ and $e_1\sigma_j, e_j \in J' = e_2\sigma_j, e_j \in J$. Clearly, both typing derivations should end with either (pair) or (subsum). Assume that both derivations end with an instance of (pair):

$$\Delta \Gamma \vdash e_1\sigma_j, e_j \in J : s_1 \quad \Delta \Gamma \vdash e_2\sigma_j, e_j \in J : s_2 \quad \text{(pair)}$$

By $E$-induction, we have $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e_1\sigma_j, e_j \in J, J', e_2\sigma_j, e_j \in J, J' : s_1 \land s_2$, that is $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e_1\sigma_j, e_j \in J, J', e_2\sigma_j, e_j \in J, J' : s_1 \land s_2$. Moreover, because the intersection distributes over product, we have $(s_1 \land s_2) \times (s_1 \land s_2) \simeq (s_1 \times s_1) \land (s_2 \times s_2)$. Finally, by (pair), we have $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e_1\sigma_j, e_j \in J, J', e_2\sigma_j, e_j \in J, J' : s_1 \land s_2$. Otherwise, there exists at least one typing derivation which ends with an instance of (subsum), similarly to the case of $e = c$, the result follows by $I$-induction.

$e = \pi_i(e' \sigma_j, e_j \in J, J') \pi_i(e'^i \sigma_j, e_j \in J, J') \pi_i(e'^i \sigma_j, e_j \in J, J')$, where $i = 1, 2$. Clearly, both typing derivations should end with either (proj) or (subsum). Assume that both derivations end with an instance of (proj):

$$\Delta \Gamma \vdash e'^i \sigma_j, e_j \in J : s_1 \times s_2 \quad \text{(proj)}$$

By $E$-induction, we have $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e'^i \sigma_j, e_j \in J, J' : s_1 \times s_2$, that is $\Delta \cup \Delta' \Delta' \Gamma' \land \Gamma' \vdash e'^i \sigma_j, e_j \in J, J' : s_1 \times s_2$. Since $(s_1 \times s_2) \land (s_1' \times s_2') \simeq (s_1 \times s_1') \times (s_2 \times s_2')$, the result follows.

$e = (e_1, e_2); e_1\sigma_j, e_j \in J = (e_1, e_2)\sigma_j, e_j \in J'$, $e_2\sigma_j, e_j \in J = (e_1, e_2)\sigma_j, e_j \in J'$, and $e_2\sigma_j, e_j \in J' = (e_1, e_2)\sigma_j, e_j \in J'$. Clearly, both typing derivations should end with either (appl) or (subsum). Assume that both derivations end with an instance of (appl):

$$\Delta \Gamma \vdash e_1\sigma_j, e_j \in J : s_1 \rightarrow s_2 \quad \Delta \Gamma \vdash e_2\sigma_j, e_j \in J : s_2 \quad \text{(appl)}$$
By E-induction, we have \( \Delta \cup D \triangledown \Gamma \vdash (e \otimes [\sigma_j]_{j \in J^T}) (e_2 \otimes [\sigma_j]_{j \in J^T}) : s' \) and \( \Delta \cup D \triangledown \Gamma \vdash (e \otimes [\sigma_j]_{j \in J^T}) : s \). Therefore, by the rule (subsum), we get

\[
\Delta \cup D \triangledown \Gamma \vdash (e \otimes [\sigma_j]_{j \in J^T}) (e_2 \otimes [\sigma_j]_{j \in J^T}) : s_2 \quad \text{and} \quad (e \otimes [\sigma_j]_{j \in J^T}) : s_1.
\]

Because intersection distributes over arrows, we have \( (s_1 \to s_2) \land (s_2 \to s' \Delta) \leq (s_1 \to s') \). Then by the rule (subsum), we get

\[
\Delta \cup D \triangledown \Gamma \vdash (e_1 \otimes [\sigma_j]_{j \in J^T}) : s_1 \land s_2 \quad \text{and} \quad (s_2 \to s').
\]

Finally by (app), we have \( \Delta \cup D \triangledown \Gamma \vdash (e \otimes [\sigma_j]_{j \in J^T}) : s_2 \rightarrow s' \). Therefore, there exists at least one typing derivation which ends with an instance of (subsum), similarly to the case of \( e \equiv e_2 \), the result follows by I-induction.

\[ e = \lambda_{[\sigma_k]_{k \in K}} x.e' : \sigma \quad \text{and} \quad e \otimes [\sigma_j]_{j \in J^T} = \lambda_{[\sigma_k]_{k \in K}} x.e'. \]

Clearly, both typing derivations should end with either (abstr) or (subsum). Assume that both derivations end with an instance of (abstr):

\[ \forall i \in I, k \in K, j \in J. \Delta \triangledown \Gamma, (x : t_i(\sigma_k \circ \sigma_j)) \vdash e' \otimes [\sigma_k] \circ [\sigma_j] : s_i(\sigma_k \circ \sigma_j) \]

\[ \Delta_1 = \Delta \cup \forall \lambda \in I \in K \in J \in J^T \vdash t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j) \]

\[ \Delta_2 = \Delta \cup \forall \lambda \in K \in K \in J \in J^T \vdash t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j) \]

Consider any expression \( e' \otimes [\sigma_k] \circ [\sigma_j] \) and any \( e_0[\sigma_{j_0}]_{j_0 \in J^T} \) in \( e' \otimes [\sigma_k] \circ [\sigma_j] \), where \( k \in K, j \in J \). Then \( e_0[\sigma_{j_0}]_{j_0 \in J^T} \) must be \( e' \). All type variables in \( \bigcup_{j \in J} \text{dom}(\sigma_{j_0}) \) must be of \( \sigma_k \circ \sigma_j \), otherwise, \( e' \otimes [\sigma_k] \circ [\sigma_j] \) is not well-typed under \( \Delta_1 \) or \( \Delta_2 \). Using \( \alpha \)-conversion, we can assume that these polymorphic type variables are different from \( \Delta_1 \cup \Delta_2 \), that is \( (\bigcup_{j \in J} \text{dom}(\sigma_{j_0})) \cap (\Delta_1 \cup \Delta_2) = \emptyset \). So we have \( e' \otimes [\sigma_k] \circ [\sigma_j] \notin \Delta_1 \cup \Delta_2 \). According to Lemma 8, we have

\[ \Delta_1 \cup \Delta_2 \triangledown \Gamma \land \Gamma', (x : t_i(\sigma_k \circ \sigma_j)) \vdash e' \otimes [\sigma_k] \circ [\sigma_j] : s_i(\sigma_k \circ \sigma_j) \]

for all \( i \in I, k \in K \) and \( j \in J \). It is clear that

\[ \Delta_1 \cup \Delta_2 = \Delta \cup \Delta' \cup \forall \lambda \in I \in K \in J \in J^T \vdash t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j) \]

By (abstr), we have

\[ \Delta \cup \Delta' \triangledown \Gamma \vdash \lambda_{[\sigma_k]_{k \in K}} x.e' : \bigwedge_{i \in I, k \in K \in J \in J^T} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j) \]

that is, \( \Delta \cup \Delta' \triangledown \Gamma \vdash \forall \lambda \in I \in K \in J \in J^T \vdash t(\sigma_k \circ \sigma_j) \rightarrow t'(\sigma_k \circ \sigma_j) \land t' = \bigwedge_{i \in I, k \in K \in J \in J^T} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j) \). Therefore, there exists at least one typing derivation which ends with an instance of (subsum), similarly to the case of \( e \equiv e_2 \), the result follows by I-induction.

\[ e = (e_0 \in \xi e_1 : e_2) \quad \text{and} \quad e_0[\sigma_{j_0}]_{j_0 \in J^T} = (e_0 \in \xi e_1 : e_2) \quad \text{and} \quad e_0[\sigma_{j_0}]_{j_0 \in J^T} = (e_0 \in \xi e_1 : e_2) \]

Clearly, both typing derivations should end with either (case) or (subsum). Assume that both derivations end with an instance of (case):

\[ \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : t_0 \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_1[\sigma_j]_{j \in J^T} : s \]

\[ \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : t_0 \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_1[\sigma_j]_{j \in J^T} : s \]

\[ \Delta \triangledown \Gamma \vdash (e_0 \in e_1 \in \xi e_2) \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : s \]

\[ \Delta \triangledown \Gamma \vdash (e_0 \in e_1 \in \xi e_2) \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : s \]

\[ \Delta \triangledown \Gamma \vdash (e_0 \in e_1 \in \xi e_2) \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : s \]

\[ \Delta \triangledown \Gamma \vdash (e_0 \in e_1 \in \xi e_2) \quad \text{and} \quad \Delta \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : s \]

By E-induction, we have \( \Delta \cup \Delta' \triangledown \Gamma \vdash e_0[\sigma_j]_{j \in J^T} : t_0 \land t_0 \). Suppose \( t_0 \land t_0 \leq \neg t \), then we must have \( t_0 \leq \neg t \) and \( t_0 \leq \neg t \), and the first branch has been checked in both derivations. Therefore
we have $\Delta \cup \Delta' \vdash e; \Gamma \vdash t : s$ by the induction hypothesis. Similarly, if $t_0 \land t'_0 \not\subseteq t$, we have $\Delta \cup \Delta' \vdash \Gamma \vdash t_0; \Gamma \vdash t'_0 : s \land s'$ by the rule (case), we have
\[
\Delta \cup \Delta' \vdash \Gamma \vdash e_0 \circ [\sigma_j]_{j \in J} \land \Delta' \vdash e_2 \circ [\sigma_j]_{j \in J} : s \land s'
\]
that is, $\Delta \cup \Delta' \vdash \Gamma \vdash (e_0 \circ [\sigma_j]_{j \in J} \land \Delta' \vdash e_2 \circ [\sigma_j]_{j \in J}) : s \land s'$.

Otherwise, there exists at least one typing derivation which ends with an instance of (subsum), similarly to the case of $e = c$, the result follows by I-induction.

$e \equiv (e' [\sigma_i]_{i \in I}) \subseteq e \equiv (e_0 \circ [\sigma_i]_{i \in I}) \land \Delta' \vdash e_0 \circ [\sigma_j]_{j \in J} : t \land t'$.

**Corollary B.14.** If $\Delta \vdash e_1 : t_1$ and $\Delta \vdash e_2 : t_2$, then $\Delta \vdash (e_1 \circ e_2) : t_1 \land t_2$.

**Proof.** Immediate consequence of Lemmas B.13 and B.8.

### B.2 Type Soundness

**Theorem B.15 (Subject Reduction).** Let $e$ be an expression and $t$ a type. If $\Delta \vdash e : t$ and $e \rightarrow e'$, then $\Delta \vdash e' : t$.

**Proof.** By induction on the derivation of $\Delta \vdash e : t$. We proceed by a case analysis on the last rule used in the derivation of $\Delta \vdash e : t$.

1. **(const):** the expression $e$ is a constant. It cannot be reduced. Thus the result follows.
2. **(var):** similar to the (const) case.
3. **(pair):** $e = (e_1, e_2)$, $t = t_1 \times t_2$. We have $\Delta \vdash e_i : t_i$ for $i = 1, 2$. There are two ways to reduce $e$, that is,
   - $e \rightarrow (e_1, e_2)$ by induction, we have $\Delta \vdash e_i : t_i$. Then the rule (pair) gives us $\Delta \vdash e : t_1 \times t_2$.
   - The case $(e_1, e_2) \rightarrow (e_1', e_2')$ is treated similarly.
4. **(proj):** $e = \pi_j(e_0)$, $t = t_i$, $\Delta \vdash e_0 : t_i \times t_2$.
   - $e_0 \rightarrow e_0'$. By induction, we have $\Delta \vdash e_0' : t_1 \times t_2$. Then the rule (proj) gives us $\Delta \vdash e' : t_i$.
   - $e_0 = (v_1, v_2)$, $e' = v_i$. By Lemma B.3, we get $\Delta \vdash e : t_1$.
5. **(app):** $e = e_1 \circ e_2$, $\Delta \vdash e : t \rightarrow s$ and $\Delta \vdash e_1 \circ e_2 : t$.
   - $e_1 \rightarrow e_1'$. By Lemma B.3, we get $\Delta \vdash e_1 : t_1$.
   - $e_2 \rightarrow e_2'$. By Lemma B.3, we get $\Delta \vdash e_2 : t_2$.
   - $e \rightarrow (e_1 \circ e_2)$ by induction, we have $\Delta \vdash e_i : t_i$. Then the rule (app) gives us $\Delta \vdash e : t_1 \circ t_2$.

A consequence, we get $\Lambda_{(i,j) \in P} s_i \neq s_j$. Moreover, since $e_1$ is well-typed under $\Delta$ and $\Gamma$, there exists an instance of the rule (abstr) which infers a type $\Lambda_{(i,j) \in P} t_i \rightarrow s_i \sigma_j$ for $e_1$ under $\Delta$ and $\Gamma$ and whose premise is $\Delta' \vdash e_0 : \Gamma \vdash s_i \sigma_j$ for all $i \in I$ and $j \in J$, where $\Delta' = \Delta \cup \{ (\sigma_j) \mid (i,j) \in P \} \cup \{ (\sigma_j) \mid (\sigma_j) \in \Gamma \}$. Since $\Gamma, \Gamma' \vdash e_0 : \Gamma \vdash s_i \sigma_j$, we have $\Delta \vdash e_0 : \Gamma \vdash s_i \sigma_j$.

**Lemma B.8.** We have $\Delta \vdash e_0 : \Gamma \vdash s_i \sigma_j$.

**Proof.** By induction on $e_0$.

1. $e_0 \rightarrow e_1$. By Lemma B.3, we get $\Delta \vdash e_1 : e_0$. The reduction can only occur in $e_0$; this case is similar to (pair) one.
   - $e = e_0 \circ e_2$. $e_0$ is a constant.
   - $e = e_0 \circ e_2$. $e_0$ is a constant.

2. $e_0 \rightarrow e_1$. By Lemma B.3, we get $\Delta \vdash e_1 : e_0$. The reduction can only occur in $e_0$; this case is similar to (pair) one.
   - $e = e_0 \circ e_2$. $e_0$ is a constant.
   - $e = e_0 \circ e_2$. $e_0$ is a constant.

3. $e = e_0 \circ e_2$. $e_0$ is a constant.

Finally, by (subsum), we obtain $\Delta \vdash e' : s$ as expected.
Theorem B.16 (Progress). Let \( e \) be a well-typed closed expression, that is, \( \vdash e : t \) for some \( t \). If \( e \) is not a value, then there exists an expression \( e' \) such that \( e \leadsto e' \).

\[
\begin{align*}
\text{(const)}: & \text{ immediate since a constant is a value.} \\
\text{(var)}: & \text{ impossible since a variable cannot be well-typed in an empty environment.} \\
\text{(pair)}: & \text{ \( e = (e_1, e_2) \), \( t = t_1 \times t_2 \), and \( \vdash e_i : t_i \) for \( i = 1...2 \). If one of the \( e_i \) can be reduced, then \( e \) can also be reduced. Otherwise, by induction, both \( e_1 \) and \( e_2 \) are values, and so is \( e \).} \\
\text{(proj)}: & \text{ \( e = \pi_i(e_0) \), \( t = t_0 \), and \( \vdash e_0 : t_1 \times t_2 \). If \( e_0 \) can be reduced to \( e_0' \), then \( e \leadsto \pi_i(e_0') \). Otherwise, \( e_0 \) is a value. By Lemma B.3, we get \( e_0 = (e_1, e_2) \), and thus \( e \leadsto v_i \).} \\
\text{(apply)}: & \text{ \( e = e_1, e_2, \vdash e_1 : \tau \rightarrow \sigma \) and \( \vdash e_2 : t \). If one of the \( e_i \) can be reduced, then \( e \) can also be reduced. Otherwise, by induction, both \( e_1 \) and \( e_2 \) are values. By Lemma B.3, we get \( e_1 = \lambda_{[\sigma]}^{t_1 \rightarrow s_1} x_0 \) such that \( t \leq \bigvee_{i \in I} t_i \sigma_i \rightarrow s_i \sigma_i \preceq t \rightarrow s \). By the definition of subtyping for arrow types, we have \( t \leq \bigvee_{i \in I} t_i \sigma_i \). Moreover, as \( \vdash e_2 : t \), the set \( P = \{ j \in J \mid \exists \sigma_i \in I : \vdash e_2 : t_i \sigma_i \} \) is non-empty. Then \( e \leadsto e_0 \equiv (e_0 \equiv (e_0 \equiv e_0) \equiv e_0) \).} \\
\text{(abstr)}: & \text{ the expression \( e \) is an abstraction which is well-typed under the empty environment. It is thus a value.} \\
\text{(case)}: & \text{ \( e = e_0 \sigma \) ? \( e_1 : e_2 \). If \( e_0 \) can be reduced, then \( e \) can also be reduced. Otherwise, by induction, \( e_0 \) is a value. If \( \vdash v : s \), then we have \( e \leadsto e_1 \). Otherwise, \( e \leadsto e_2 \).} \\
\text{(inst)}: & \text{ \( e = e_1[\sigma] \), \( t = \sigma \) and \( \vdash e_1 : t \). Then \( e \leadsto e_1[\sigma] \).} \\
\text{(inter)}: & \text{ \( e = e_1[\sigma] \), \( t = \bigwedge_{i \in I} t_i \) and \( \vdash e_1[\sigma] : t_j \) for all \( j \in J \). It is clear that \( e \leadsto e_1[\sigma] \).} \\
\text{(subsum)}: & \text{ straightforward application of the induction hypothesis.}
\end{align*}
\]

B.3 Expressing Intersection Types

We now prove that the calculus with explicit substitutions is able to derive the same typings as the Barendregt, Coppo, Dezani (BCD) intersection type system [1] without the universal type \( \omega \). We remind the BCD typing rules (without \( \omega \)) and subtyping relation in Figure 7, where we let \( m \) range over pure lambda expressions. To make the correspondence between the systems easier, we adopt a n-ary version of the intersection typing rule. Henceforth, we let \( D \) range over BCD typing derivations. We first remark that the BCD subtyping relation is included in the one of this work.

Lemma B.17. If \( t_1 \leq_{\text{BCD}} t_2 \) then \( t_1 \leq t_2 \).

Proof. All the BCD subtyping rules are admissible in [2] and, a fortiori, in our system.
Typing rules:

\[
\begin{align*}
\Gamma \vdash_{BCD} x : \Gamma(x) & \quad (BCD \text{ var}) \\
\Gamma, x : t_1 \vdash_{BCD} m : t_2 & \quad (BCD \text{ abstr}) \\
\Gamma \vdash_{BCD} \lambda x. m : t_1 \rightarrow t_2 & \quad (BCD \text{ sub}) \\
\end{align*}
\]

Subtyping relation:

\[
\begin{array}{c c c c c}
\hline
\text{ } & \quad \text{ } & \quad \text{ } & \quad \text{ } & \quad \text{ } \\
\Gamma \vdash_{BCD} m : t & \quad \Gamma \vdash_{BCD} m : t \land t & \quad t_1 \land t_2 \leq_{BCD} t_1 & \quad (t_1 \rightarrow t_2) \land (t_1 \rightarrow t_3) \leq_{BCD} t_1 \rightarrow (t_2 \land t_3) & \quad t_1 \leq_{BCD} t_2 \\
\hline
\text{ } & \quad \text{ } & \quad \text{ } & \quad \text{ } & \quad \text{ } \\
\Gamma \vdash_{BCD} m : t & \quad \Gamma \vdash_{BCD} m : t_2 \land t_3 & \quad t_1 \land t_2 \leq_{BCD} t_3 & \quad t_2 \leq_{BCD} t_4 & \quad t_3 \leq_{BCD} t_1 & \quad t_2 \leq_{BCD} t_4 & \quad t_1 \rightarrow t_2 \leq_{BCD} t_3 \rightarrow t_4 \\
\hline
\end{array}
\]

Figure 7. The BCD type system

We prove that any BCD typing judgement can be proved with a derivation in intersection-abstraction normal form.

**Lemma B.18.** If \( \Gamma \vdash_{BCD} m : t \), then there exists a derivation in intersection-abstraction normal form proving this judgement.

**Proof.** Let \( D \) be the derivation proving \( \Gamma \vdash_{BCD} m : t \). We proceed by induction on the size of \( D \), defined as the number of rules used in \( D \). If \( D \) is of size 1, then the rule \((BCD \text{ var})\) has been used, and \( D \) is in intersection-abstraction normal form.

Let \( D \) be of size strictly greater than one. We proceed by case analysis on the last rule used in \( D \). If \( D \) ends with \((BCD \text{ subsum})\), then

\[
D = \frac{D'}{\Gamma \vdash_{BCD} m : t}
\]

By the induction hypothesis, there exists a derivation \( D'' \) in intersection-abstraction normal form which proves the same judgement as \( D' \). Then

\[
\frac{D''}{\Gamma \vdash_{BCD} m : t}
\]

is in intersection-abstraction normal form and proves the same judgement as \( D \). The cases \((BCD \text{ abstr})\) and \((BCD \text{ app})\) are proved similarly.

If \( D \) ends with \((BCD \text{ inter})\), then

\[
D = \frac{D_i}{\Gamma \vdash_{BCD} m : t \in I}
\]

where each \( D_i \) proves a judgement \( \Gamma \vdash_{BCD} m : t_i \) and \( t = \bigwedge_{i \in I} t_i \). We distinguish several cases.

**If one of the derivations ends with \((BCD \text{ sub})\),** there exists \( i_0 \in I \) such that

\[
D_{i_0} = \frac{D'_{i_0}}{\Gamma \vdash_{BCD} m : t'_{i_0}}
\]

The derivation

\[
\frac{D' = \frac{D_i}{\Gamma \vdash_{BCD} m : \bigwedge_{i \in I \setminus \{i_0\}} t_i \land t'_{i_0}}}{i \in I \setminus \{i_0\}}
\]
is smaller than \( D \), so by the induction hypothesis, there exists \( D'' \) in intersection-abstraction normal form which proves the same judgement as \( D' \). Then the derivation

\[
D'' \quad \bigwedge_{i \in I \setminus \{i_0\}} t_i \land t_i' \leq_{BCD} \bigwedge_{i \in I} t_i
\]

\[
\frac{}{\Gamma \vdash_{BCD} m : t}
\]

is in intersection-abstraction normal form, and proves the same judgement as \( D \).

**If one of the derivations ends with \((BCD\text{ inter})\),** there exists \( i_0 \in I \) such that

\[
D_{i_0} = \frac{D_{j, i_0}}{\Gamma \vdash_{BCD} m : \bigwedge_{j \in J} t_{j, i_0}} \quad j \in J
\]

with \( t_{i_0} = \bigwedge_{j \in J} t_{j, i_0} \). The derivation

\[
D' = \frac{D_i \quad D_{j, i_0}}{\Gamma \vdash_{BCD} m : t} \quad i \in I \setminus \{i_0\} \quad j \in J
\]

is smaller than \( D \), so by the induction hypothesis, there exists \( D'' \) in intersection-abstraction normal form which proves the same derivation as \( D' \), which is the same as the judgement of \( D \).

**If all the derivations are uses of \((BCD\text{ var})\),** then for all \( i \in I \), we have

\[
D_i = \frac{\Gamma \vdash_{BCD} x : \Gamma(x)}{\Gamma \vdash_{BCD} x : \Gamma(x)}
\]

which implies \( t = \bigwedge_{i \in I} \Gamma(x) \) and \( m = x \). Then the derivation

\[
\frac{\Gamma \vdash_{BCD} x : \Gamma(x) \quad \Gamma(x) \leq_{BCD} t}{\Gamma \vdash_{BCD} x : t}
\]

is in intersection-abstraction normal form and proves the same judgement as \( D \).

**If all the derivations end with \((BCD\text{ app})\),** then for all \( i \in I \), we have

\[
D_i = \frac{D_i^1 \quad D_i^2}{\Gamma \vdash_{BCD} m_1 : t_i \quad \Gamma \vdash_{BCD} m_2 : t_i}
\]

where \( m = m_1 \cdot m_2 \), \( D_i^1 \) proves \( \Gamma \vdash_{BCD} m_1 : s_i \to t_i \), and \( D_i^2 \) proves \( \Gamma \vdash_{BCD} m_2 : s_i \) for some \( s_i \). Let

\[
D_1 = \frac{D_i^1}{\Gamma \vdash_{BCD} m_1 : \bigwedge_{i \in I} s_i \to t_i} \quad D_2 = \frac{D_i^2}{\Gamma \vdash_{BCD} m_2 : \bigwedge_{i \in I} s_i}
\]

Both \( D_1 \) and \( D_2 \) are smaller than \( D \), so by the induction hypothesis, there exist \( D'_1, D'_2 \) in intersection-abstraction normal form which prove the same judgements as \( D_1 \) and \( D_2 \) respectively. Then the derivation

\[
D'_1 \quad \bigwedge_{i \in I} s_i \to t_i \leq_{BCD} \bigwedge_{i \in I} s_i \to \bigwedge_{i \in I} t_i
\]

\[
\frac{}{\Gamma \vdash_{BCD} m_1 : \bigwedge_{i \in I} s_i \to \bigwedge_{i \in I} t_i} \quad \Gamma \vdash_{BCD} m_1 : t_i
\]

\[
\frac{\Gamma \vdash_{BCD} m_1 : t_i}{}\]

\[
D' = \frac{\Gamma \vdash_{BCD} m_1 \cdot m_2 : t_i}{\Gamma \vdash_{BCD} m_1 \cdot m_2 : t_i}
\]

is in intersection-abstraction normal form and proves the same derivation as \( D \).

**If all the derivations end with \((BCD\text{ abstr})\),** then for all \( i \in I \), we have

\[
D_i = \frac{D'_i}{\Gamma \vdash_{BCD} \lambda x. m' : t_i \quad (BCD\text{ abstr})}
\]

with \( m = \lambda x. m' \). For all \( i \in I \), \( D'_i \) is smaller than \( D \), so by induction there exists \( D''_i \) in intersection-abstraction normal form which proves the same judgement as \( D'_i \). Then the derivation

\[
D''_i \quad \bigwedge_{i \in I} t_i
\]

\[
\frac{}{\Gamma \vdash_{BCD} \lambda x. m' : \bigwedge_{i \in I} t_i}
\]

is in intersection-abstraction normal form, and proves the same judgement as \( D \).
We now sketch the construction of the λ term from given M.

If \( D \) proves a judgement \( \Gamma \vdash_{BCD} \lambda x.m : t \to s \), then we may put \( t \to s \) in the interface of the corresponding expression \( \lambda t.e \).

For a judgement \( \Gamma \vdash_{BCD} \lambda x.m : \Delta \to \delta \), we build an expression \( \lambda^{\alpha \to \beta}_{\sigma_1 \to \sigma_2}f.\lambda x.\phi \) where each \( \sigma_i \) corresponds to the derivation which types \( \lambda x.m \) with \( t_i \to s_i \). For example, let \( m = \lambda f.\lambda x.f \) and \( \sigma = \phi \). We first annotate each abstraction with types \( \alpha \), \( \beta \), where \( \alpha \) and \( \beta \) are fresh, distinct variables, giving us \( \epsilon = \lambda^{\alpha_1 \to \beta_1}f.\lambda x.\phi \). Comparing \( \lambda^{\alpha_1 \to \beta_1}f.\lambda x.\phi \) to \( \lambda x.\phi \) to the judgement \( \Gamma \vdash t_i \to s_i \), we compute \( \sigma_1 = \{ t_i \to s_i \} \). We then use the inductive hypothesis to compute \( \sigma_2 = \{ s_i \to s_i \} \) and \( \sigma = \{ s_i \to s_i \} \).

The problem becomes more complex when we have nested uses of the intersection typing rule. For example, let \( m = \lambda f.\lambda x.f (\lambda y.x.y) \) and \( \sigma = \phi \). To detect \( f \) we have to compute four substitutions \( \sigma_1, \sigma_2, \sigma_3, \sigma_4 \) to obtain a decorated expression \( \lambda^{\alpha_1 \to \beta_1}f.\lambda x.\phi \). Because the intersection typing rule is used twice (once to \( \phi \) and once to \( \phi \)), we want to compute four substitutions \( \sigma_1, \sigma_2, \sigma_3, \sigma_4 \) to obtain a decorated expression \( \lambda^{\alpha_1 \to \beta_1}f.\lambda x.\phi \). The difficult part is in computing \( \sigma_3 \) and \( \sigma_4 \); in one case (corresponding to the \( \Gamma \vdash_{BCD} \lambda x.m : t \to s \)) we want \( \sigma_3 = \sigma_4 = \{ t_i \to s_i \} \) to obtain \( \Gamma \vdash \lambda x.m : t \to s \). In the other case (corresponding to the \( \Gamma \vdash_{BCD} \lambda x.m : t \to s \)) we want \( \sigma_3 = \{ s_i \to s_i \} \) to obtain \( \Gamma \vdash \lambda x.m : t \to s \). To resolve this issue, we use intermediate fresh variables \( \alpha_6, \beta_6, \alpha_7, \beta_7 \) in the definition of \( \sigma_3 \) and \( \sigma_4 \). We define

\[
\sigma_1 = \{ t_i \to s_i \}, \sigma_2 = \{ t_i \to s_i \}, \sigma_3 = \{ s_i \to s_i \}, \sigma_4 = \{ s_i \to s_i \}
\]

Because the substitutions compose themselves, we obtain \( \Gamma \vdash_{BCD} \lambda x.m : t \to s \) as wished.

In the next lemma, given \( n \) derivations \( D_1 \ldots D_n \) in intersection-abstraction normal form for a same expression \( m \), we construct an expression \( \epsilon \) containing fresh interfaces and decorations with fresh variables if needed (as explained in the example above) and \( n \) substitutions \( \sigma_1 \ldots \sigma_n \) corresponding to \( D_1 \ldots D_n \).

**Lemma B.19.** Let \( m \) be a pure lambda expression, \( \Delta \) be a set of type variables, and \( D_1 \ldots D_n \) be derivations in intersection-abstraction normal form such that \( D_i \) proves \( \Gamma_i \vdash_{BCD} \lambda x.m : t_i \to s_i \). Let \( \Delta' \) be a set containing all the type variables of all the types occurring in \( D_1 \ldots D_n \). There exist \( \epsilon, \sigma_1 \ldots \sigma_n \) such that \( [\epsilon] = \text{m} \), \( \text{dom} (\sigma_i) = \text{m} \), \( \text{dom}(\sigma_i) \subseteq \text{tv}(e) \), \( \text{tv}(e) \cap (\Delta \cup \Delta') = \emptyset \), and \( \Delta' \vdash \text{e} \epsilon \sigma_1 \ldots \sigma_n \) for all \( i \).

**Proof.** We proceed by induction on the size of \( D_1 \ldots D_n \), where the size of a derivation is defined as in the proof of Lemma B.18. If this sum is equal to \( n \), then each \( D_i \) is a use of the \( \text{(BCD var)} \) rule, and we have \( m = x \) for some \( x \). Let \( e = x \) and \( \sigma_i \) be the identity; we can then easily check that the result holds.

If this sum is strictly greater than \( n \), we proceed by case analysis on \( D_1 \ldots D_n \).

**If one of the derivations ends with (BCD sub),** there exists \( i_0 \) such that

\[
D_{i_0} = D_{i_0} \leq_{BCD} t_{i_0} \vdash_{BCD} \lambda x.m : t_{i_0}
\]
The sequence of derivations $D_1 \ldots D_{i_0} \ldots D_n$ is smaller than $D_1 \ldots D_n$, so by the induction hypothesis, there exist $e, \sigma_1, \ldots, \sigma_n$ such that $[e] = m$, $\Delta \vdash \Gamma_i, e \sigma_1[i] : t_i$ for all $i \neq 0$, and $\Delta \vdash \Gamma_{i_0}, e \sigma_1[i_0] : t_{i_0}$. We have $t_{i_0} \leq_{BCD} t_i$, so by Lemma B.17, we have $t_{i_0} \leq t_i$. Therefore $\Delta \vdash \Gamma_i, e \sigma_1[i] : t_i$ holds, and for all $i$, we have $\Delta \vdash \Gamma_i, t_i$ as wished.

If all the derivations end with $(BCD \text{ app})$, then we have $m = m_1, m_2$, and for all $i$:

$$D_i = \frac{D_i^1}{\Gamma_i \vdash_{BCD} m_1} \frac{D_i^2}{\Gamma_i \vdash_{BCD} m_2} : t_i$$

where $D_i^1$ proves $\Gamma_i \vdash_{BCD} m_1 : s_i \rightarrow t_i$ and $D_i^2$ proves $\Gamma_i \vdash_{BCD} m_2 : s_i$. By the induction hypothesis, there exist $e_1, \sigma_1, \ldots, \sigma_n$ such that $[e_1] = m_1, dom(\sigma_1) = \ldots = dom(\sigma_n) = tv(e_1)$, $tv(e_1) \in (\Delta \cup \Delta') = 0$, and $\Delta \vdash \Gamma_i, e \sigma_1[i] : s_i \rightarrow t_i$ for all $i$. There exist also $e_2, \sigma_1, \ldots, \sigma_n, [e_2] = m_2, dom(\sigma_2) = \ldots = dom(\sigma_n) = tv(e_2)$, $tv(e_2) \in (\Delta \cup \Delta') = 0$, and $\Delta \vdash \Gamma_i, e \sigma_2[i] : s_i$ for all $i$. From $tv(e_2) \in (\Delta \cup \Delta') = 0$, we deduce $tv(e_1) \cap tv(e_2) = 0$. Let $i \in 1 \ldots n$. Because $dom(\sigma_1) \subseteq tv(e_1)$ and $dom(\sigma_2) \subseteq tv(e_2)$, we have $dom(\sigma_1) \cap dom(\sigma_2) = 0$, $dom(\sigma_1) \cap tv(e_2) = 0$, and $dom(\sigma_2) \cap tv(e_1) = 0$. Consequently, by Lemma B.9, we have $\Delta \vdash \Gamma_i, e \sigma_1[i] \cup \sigma_2[i] : s_i \rightarrow t_i$ and $\Delta \vdash \Gamma_i, e \sigma_2[i] \cup \sigma_1[i] : s_i$. Therefore, we have $\Delta \vdash \Gamma_i, (e_1) \cup \sigma_1[i] \cup \sigma_2[i] : t_i$, so we have the required result with $e = e_1$ and $\sigma_i = \sigma_1[i] \cup \sigma_2[i]$.

If all the derivations end with $(BCD \text{ abstr})$, then $m = \lambda x.m_1$, and for all $i$,

$$D_i = \frac{D_i'}{\Gamma_i \vdash_{BCD} m : t_i}$$

where $D_i'$ proves $\Gamma_i, x : t_i \vdash_{BCD} m_1 : t_i'$ and $t_i = t_i' \rightarrow t_i''$. By the induction hypothesis, there exist $e_1, \sigma_1, \ldots, \sigma_n$ such that $[e_1] = m_1, dom(\sigma_1) = \ldots = dom(\sigma_n) = tv(e_1)$, $tv(e_1) \cap (\Delta \cup \Delta') = 0$, and $\Delta \vdash \Gamma_i, x : t_i \vdash e \sigma_1[i] : t_i''$ for all $i$. Let $\alpha, \beta$ such that $(\alpha, \beta) \cap \Delta \cup \Delta' = 0$ and $(\alpha, \beta) \cap tv(e_1) = 0$. Let $i \in 1 \ldots n$. Let $\sigma_j = \{t_i^j, \alpha, \beta\} \cup \sigma_i'$, and $e = \lambda x.\rightarrow \alpha.\beta \sigma_i[t_1^j, \alpha, \beta]$. We have $dom(\sigma) \subseteq dom(\sigma_i') \cup \sigma_i'[\alpha, \beta]$, and $\Delta \vdash tv(e_1) \cap (\Delta \cup \Delta') = 0$. Because $tv(e_1) \cap (\Delta \cup \Delta') = 0$, we have $dom(\sigma_i') \cap (\alpha, \beta) = 0$, and $\Delta \vdash \Gamma_i, x : t_i \vdash e \sigma_1[i] : t_i''$ by Lemma B.9 which is equivalent to $\Delta \vdash \Gamma_i, x : \alpha \sigma_1[i] \cup \alpha \sigma_1[i] : t_i''$. Therefore, by the abstraction rule, we have $\Delta \vdash \Gamma_i, \lambda x.\rightarrow \alpha.\beta \sigma_i[t_1^j, \alpha, \beta] : t_i''$. Therefore, we have the required result.

If one of the derivations ends with $(BCD \text{ inter})$, then $m = \lambda x.m_1$. The derivations end with either $(BCD \text{ inter})$ or $(BCD \text{ abstr})$ (we omit the already treated case of $(BCD \text{ sub})$). For simplicity, we suppose they all end with $(BCD \text{ inter})$, the reasoning is the same if some of them end with $(BCD \text{ abstr})$. For all $i$, we have

$$D_i = \frac{D_i'}{\Gamma_i \vdash_{BCD} m : \sum_{j \in J_i} s_i^j \rightarrow t_i^j}$$

where $D_i'$ proves $\Gamma_i, x : s_i^j \vdash_{BCD} m_1 : t_i^j$ for all $j \in J_i$, and $t_i = \bigwedge_{j \in J_i} s_i^j \rightarrow t_i^j$ for all $i$. By the induction hypothesis, there exist $e_1, (\sigma_1^j)_{j \in J_i}, \ldots, (\sigma_n^j)_{j \in J_i}$ such that $[e_1] = m_1, tv(e_1) \cap (\Delta \cup \Delta') = 0$, $dom(\sigma_j^j) = dom(\sigma_j')$ for all $i, j', j''$, and $dom(\sigma_j') \subseteq tv(e_1)$. Let $\sigma_j = \bigcup_{j \in J_i} \{t_i^j, \alpha, \beta, j\} \cup \bigcup_{j \in J_i} \{t_i^j, \alpha, \beta, j\} \cup \bigcup_{j \in J_i} \{t_i^j, \alpha, \beta, j\}$

For all $i, j, k$, we have by construction $(\sigma_j(\alpha_1)) \sigma_i = \alpha_k \sigma_i$, $(\sigma_j(\alpha)) \sigma_i = s_i^j$, and $(\sigma_j(\beta)) \sigma_i = t_i^j$. Because $\Delta \vdash \Gamma_i, x : s_i^j \rightarrow e \sigma_1[i] : t_i^j$, which is equivalent to $\Delta \vdash \Gamma_i, x : e \sigma_1[i] \cup \alpha \sigma_1[i] \cup \beta \sigma_1[i] : t_i^j$ for all $j$, $k$, we have $\Delta \vdash \Gamma_i, e \sigma_1[i] : t_i^j$. Therefore, we have $\Delta \vdash \Gamma_i, e \sigma_1[i] : t_i^j \leq \bigwedge_{j \in J_i} s_i^j \rightarrow t_i^j = t_i$, by the abstraction typing.
This is equivalent to $\Delta^t \Gamma_i \vdash \lambda_{\sigma_j \in t_1 \ldots p} x.e_i \oplus [\sigma_j] : t_i$, hence $\Delta^t \Gamma_i \vdash e \oplus [\sigma_j] : t_i$ holds for all $i$, as wished.

We are now ready to prove the main result of this subsection.

**Theorem B.20.** If $\Gamma \vdash_{BCD} m : t$, then there exist $e$, $\Delta$ such that $\Delta \vdash e : t$ and $[e] = m$.

**Proof.** By Lemma B.18 there exists $D$ in intersection-abstraction normal form such that $D$ proves $\Gamma \vdash_{BCD} m : t$. Let $\Delta$ be the set of all the type variables of all the types occurring in $D$. We prove by induction on $D$ that there exists $e$ such that $\Delta \vdash e : t$ and $[e] = m$.

**Case (BCD var).** The expression $m$ is a variable and the result holds with $e = m$.

**Case (BCD sub).** We have

$$D = \frac{D'}{\Gamma \vdash_{BCD} m : t}$$

where $D'$ proves $\Gamma \vdash_{BCD} m : t'$. The derivation $D'$ is in intersection-abstraction normal form, so by the induction hypothesis, there exists $e$ such that $[e] = m$ and $\Delta \vdash e : t'$. By Lemma B.17 we have $t' \preceq t$, therefore we have $\Delta \vdash e : t$, as wished.

**Case (BCD app).** We have

$$D = \frac{D_1 D_2}{\Gamma \vdash_{BCD} m : t}$$

where $D_1$ proves $\Gamma \vdash_{BCD} m_1 : s \to t$, $D_2$ proves $\Gamma \vdash_{BCD} m_2 : s$, and $m = m_1 m_2$. Both $D_1$ and $D_2$ are in intersection-abstraction normal form, so by the induction hypothesis, there exist $e_1$ and $e_2$ such that $[e_1] = m_1$, $[e_2] = m_2$, $\Delta \vdash e_1 : s \to t$, and $\Delta \vdash e_2 : s$. Consequently we have $\Delta \vdash e_1 e_2 : t$, with $[e_1 e_2] = m_1 m_2$, as wished.

**Case (BCD abstr).** Because $D$ is in intersection-abstraction normal form, we have

$$D = \frac{\Gamma \vdash_{BCD} \lambda x.m' : s_i \to t_i}{\Gamma \vdash_{BCD} m : t}$$

where each $D_i$ is in intersection-abstraction normal form and proves $\Delta \vdash \Gamma, x : s_i \vdash_{BCD} m_i' : t_i$, with $t = \bigwedge_{i \in I} t_i \to t$ and $m = \lambda x.m'$. By Lemma B.19 there exist $e', \sigma_1 \ldots \sigma_n$ such that $[e'] = m'$, $\text{dom}(\sigma_1) = \ldots = \text{dom}(\sigma_n) \subseteq \text{tv}(e')$, and $\Delta \vdash \Gamma, x : s_i \vdash e' \oplus [\sigma_i] : t_i$ for all $i \in I$. Let $i \in I$; we define $\sigma_i = \sigma_i \cup \{s_i \mapsto t_i\}$. Let $\alpha, \beta$ such that $\{\alpha, \beta\} \cap \text{tv}(e') = \emptyset$. Because $\text{dom}(\sigma_i) \cap \{\alpha, \beta\} = \emptyset$, we have $\Delta \vdash \Gamma, x : s_i \vdash e' \oplus [\sigma_i] : t_i$. For $\Delta \vdash \Gamma, x : \sigma_i \vdash e' \oplus [\sigma_i] : \beta \sigma_i$, note that $\Delta \cup \text{Var}(\bigwedge_{i \in I}(\alpha \to \beta) \sigma_i) = \Delta \cup \text{Var}(\bigwedge_{i \in I} s_i \to t_i) = \Delta$ by definition of $\Delta$, so by rule (abstr), we have $\Delta \vdash \Gamma, x : \lambda_{[\sigma_i]} \in \bigwedge_{i \in I} s_i \to t_i$. Hence we have the required result with $e = \lambda_{[\sigma_i]} \in \bigwedge_{i \in I} x.e_i$.

\[\square\]

### C. Algorithmic Type Checking

The typing rules provided in Section A.3 are not syntax-directed because of the presence of the subsumption rule. In this section we present an equivalent type system with syntax-directed rules. In order to define it we consider the rules of Section A.3. First, we merge the rules (inst) and (inter) into one rule (intersection is interesting only to merge different instances of a same expression), and then we consider where subsumption is used and whether it can be postponed by moving it down the derivation tree.

#### C.1 Merging Intersection and Instantiation

Intersection is used to merge different types derived for the same term. In this calculus, we can derive different types for a term because of either subsumption or instantiation. However, the intersection of different supertypes can be obtained by subsumption itself (if $t \leq t_1$ and $t \leq t_2$, then $t \leq t_1 \land t_2$), so intersection is really useful only to merge different instances of a same expression, as we can see with rule (inter) in Figure B.5! Note that all the subjects in the premise of (inter) share the same structure $e[\sigma]$, and the typing derivations of these terms must end with either (inst) or (subsum). We show that we can in fact postpone the uses of (subsum) after (inst), and we can therefore merge the rules (inst) and (inter) into one rule (instinter) as follows:

$$\frac{\Delta \vdash e : t \quad \forall j \in J, \sigma_j \notin \Delta}{\Delta \vdash e[\sigma_j] : \bigwedge_{j \in J} t \sigma_j} \text{ (instinter)}$$
Let $\Delta \vdash e : t$ denote the typing judgments derivable in the type system with the typing rule (instinter) but not (inst) and (inter). The following theorem proves that the type system $\vdash_m$ is equivalent to the original one $\vdash$.

**Theorem C.1.** Let $e$ be an expression. Then $\Delta \vdash e : t$ $\iff$ $\Delta \vdash e : t$.

**Proof.** $\Rightarrow$: It is clear that (inst) is a special case of (instinter) where $|J| = 1$. We simulate each instance of (instinter) where $|J| > 1$ by using several instances of (inst) followed by one instance of (inter). In detail, consider the following derivation

$$
\frac{\Delta \vdash e' : t' \quad \sigma_j \# \Delta'}{\Delta \vdash e' \; \sigma_j \in J : \bigwedge_{j \in J} t' \sigma_j} \quad \text{ (instinter)}
$$

We can rewrite this derivation as follows:

$$
\frac{\Delta \vdash e' : t' \quad \sigma_1 \# \Delta'}{\Delta \vdash e' \; \sigma_1 : t' \sigma_1} \quad \frac{\Delta \vdash e' : t' \quad \sigma_{|J|} \# \Delta'}{\Delta \vdash e' \; \sigma_{|J|} : t' \sigma_{|J|}} \quad \text{ (inst)}
$$

$$
\frac{\Delta \vdash e' \; \sigma_j \in J : \bigwedge_{j \in J} t' \sigma_j}{\Delta \vdash e' : t} \quad \text{ (inter)}
$$

We simulate each instance

$$
\frac{\Delta \vdash e' : t'}{\Delta \vdash e' : \sigma_j \in J : \bigwedge_{j \in J} t' \sigma_j} \quad \text{ (instinter)}
$$

$\Rightarrow$: The proof proceeds by induction and case analysis on the structure of $e$. For each case we use an auxiliary internal induction on the typing derivation. We label $E$ the main (external) induction and $I$ the internal induction in what follows.

- $e = e'$: the typing derivation $\Delta \vdash e : t$ should end with either (const) or (subsum). If the typing derivation ends with (const), the result follows straightforward.

- Otherwise, the typing derivation ends with an instance of (subsum):

$$
\frac{\Delta \vdash e : s \quad s \leq t}{\Delta \vdash e : t} \quad \text{ (subsum)}
$$

Then by $I$-induction, we have $\Delta \vdash m \vdash e : s$. Since $s \leq t$, by subsumption, we get $\Delta \vdash \vdash_m e : t$.

- $e = e^2$: similarly to the case of $e = e'$.

- $e = (e_1, e_2)$: the typing derivation $\Delta \vdash e : t$ should end with either (pair) or (subsum). Assume that the typing derivation ends with (pair):

$$
\frac{\Delta \vdash e_1 : t_1 \quad \Delta \vdash e_2 : t_2}{\Delta \vdash (e_1, e_2) : t_1 \times t_2} \quad \text{ (pair)}
$$

By $E$-induction, we have $\Delta \vdash \vdash_m e_1 : t_1$. Then the rule (pair) gives us $\Delta \vdash \vdash_m (e_1, e_2) : t_1 \times t_2$.

- Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of $e = e'$, the result follows by $I$-induction.

- $e = \pi_i(e')$: the typing derivation $\Delta \vdash e : t$ should end with either (proj) or (subsum). Assume that the typing derivation ends with (proj):

$$
\frac{\Delta \vdash e' : (t_1 \times t_2)}{\Delta \vdash \pi_i(e') : t_i} \quad \text{ (proj)}
$$

By $E$-induction, we have $\Delta \vdash \vdash_m e' : (t_1 \times t_2)$. Then the rule (proj) gives us $\Delta \vdash \vdash_m \pi_i(e') : t_i$.

- Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of $e = e'$, the result follows by $I$-induction.

- $e = e_1 e_2$: the typing derivation $\Delta \vdash e : t$ should end with either (appl) or (subsum). Assume that the typing derivation ends with (appl):

$$
\frac{\Delta \vdash e_1 : t_1 \rightarrow t_2 \quad \Delta \vdash e_2 : t_2}{\Delta \vdash e_1 e_2 : t_2} \quad \text{ (appl)}
$$

By $E$-induction, we have $\Delta \vdash \vdash_m e_1 : t_1 \rightarrow t_2$ and $\Delta \vdash \vdash_m e_2 : t_2$. Then the rule (appl) gives us $\Delta \vdash \vdash_m e_1 e_2 : t_2$.

- Otherwise, the typing derivation ends with an instance of (subsum), similarly to the case of $e = e'$, the result follows by $I$-induction.
The typing derivation $\Delta \vdash e : t$ should end with either \emph{(abstr)} or \emph{(subsum)}. Assume that the typing derivation ends with \emph{(abstr)}:

\[
\begin{align*}
\forall i \in I, j \in J, \Delta \vdash e_i : t_i \rightarrow s_i x.e' & \quad (\text{abstr}) \\
\Delta' = \Delta \cup \text{var}(\bigcup_{i \in I, j \in J} (t_i, s_i)) \\
\Delta \vdash e_{\pi^i} : e' : \bigcup_{i \in I, j \in J} (t_i, s_i) & \\
\end{align*}
\]

By $E$-induction, for all $i \in I$ and $j \in J$, we have $\Delta \vdash (x : t_i) \vdash e'_i [\sigma_j] : s_i \sigma_j$. Then the rule \emph{(abstr)} gives us $\Delta \vdash \Delta' \vdash e' : \bigcup_{i \in I, j \in J} (t_i, s_i)$. Otherwise, the typing derivation ends with an instance of \emph{(subsum)}, similarly to the case of $e = e$, the result follows by $I$-induction.

$e = e' \in \ell \ ? e_1 : e_2$; the typing derivation $\Delta \vdash e : t$ should end with either \emph{(case)} or \emph{(subsum)}. Assume that the typing derivation ends with \emph{(case)}:

\[
\begin{align*}
\Delta \vdash e' & \quad (\text{case}) \\
\end{align*}
\]

By $E$-induction, we have $\Delta \vdash \Delta \vdash e' : t'$. Since $\sigma \notin \Delta$, applying \emph{(inst)} where $|J| = 1$, we get $\Delta \vdash \Delta \vdash e'[\sigma] : t$. Otherwise, the typing derivation ends with an instance of \emph{(subsum)}, similarly to the case of $e = e$, the result follows by $I$-induction.

$e = e'[\sigma]$; the typing derivation $\Delta \vdash e : t$ should end with either \emph{(inst)} or \emph{(subsum)}. Assume that the typing derivation ends with \emph{(inst)}:

\[
\begin{align*}
\Delta \vdash e' : t & \quad (\text{inst}) \\
\Delta \vdash \sigma \in \Delta & \\
\end{align*}
\]

By $E$-induction, we have $\Delta \vdash \Delta \vdash e'[\sigma] : t$. Since $\sigma \notin \Delta$, applying \emph{(inst)} where $|J| = 1$, we get $\Delta \vdash \Delta \vdash e'[\sigma] : t$. Otherwise, the typing derivation ends with an instance of \emph{(subsum)}, similarly to the case of $e = e$, the result follows by $I$-induction.

$e = e'[\sigma]_j \in J$; the typing derivation $\Delta \vdash e : t$ should end with either \emph{(inter)} or \emph{(subsum)}. Assume that the typing derivation ends with \emph{(inter)}:

\[
\begin{align*}
\forall j \in J, \Delta \vdash e'[\sigma]_j : t_j & \quad (\text{inter}) \\
\end{align*}
\]

As an intermediary result, we first prove that the derivation can be rewritten as

\[
\begin{align*}
\forall j \in J, \Delta \vdash e'[\sigma]_j : s_j & \quad (\text{inst}) \\
\Delta \vdash e'[\sigma]_j : s_j \in \Delta & \quad (\text{inter}) \\
\Delta \vdash e'[\sigma]_j : s_j \in \bigcup_{j \in J} t_j & \quad (\text{subsum}) \\
\end{align*}
\]

We proceed by induction on the original derivation. It is clear that each sub-derivation $\Delta \vdash e'[\sigma]_j : t_j$ ends with either \emph{(inst)} or \emph{(subsum)}. If all the sub-derivations end with an instance of \emph{(inst)}, then for all $j \in J$, we have

\[
\begin{align*}
\Delta \vdash e'[\sigma]_j : s_j & \quad (\text{inst}) \\
\end{align*}
\]

By Lemma [B.2], we have $\Delta \vdash e'[\sigma]_j : \bigcup_{j \in J} s_j$. Let $s = \bigcup_{j \in J} s_j$. Then by \emph{(inst)}, we get $\Delta \vdash e'[\sigma]_j : s \sigma_j$. Finally, by \emph{(inter)} and \emph{(subsum)}, the intermediary result holds. Otherwise, there is at least one of the sub-derivations ends with an instance of \emph{(subsum)}, the intermediary result also hold by induction.

Now that the intermediary result is proved, we go back to the proof of the lemma. By $E$-induction on $e'$ (i.e., $\Delta \vdash e' : s$), we have $\Delta \vdash e' : s$. Since $\sigma_j \notin \Delta$, applying \emph{(inst)}), we get $\Delta \vdash e'[\sigma]_j \in \bigcup_{j \in J} s_j$. Finally, by subsumption, we get $\Delta \vdash e'[\sigma]_j : \bigcup_{j \in J} t_j$. Otherwise, the typing derivation ends with an instance of \emph{(subsum)}, similarly to the case of $e = e$, the result follows by $I$-induction.
From now on we will use \( \vdash \) to denote \( \vdash_m \), that is the system with the merged rule.

C.2 Algorithmic Typing Rules

In this section, we analyse the typing derivations produced by the rules of Section A.3 to see where subsumption is needed and where it can be pushed down the derivation tree. We need first some preliminary definitions and decomposition results about pair and function types to deal with the projection and application rules.

C.2.1 Pair types

A type \( s \) is a pair type if \( s \leq \emptyset \times \emptyset \). If an expression \( e \) is typable with a pair type \( s \), we want to compute from \( s \) a valid type for \( \pi_i(e) \). In C.Duce, a pair type \( s \) is a finite union of product types, which can be decomposed into a finite set of pairs of types, denoted as \( \pi(s) \). For example, we decompose \( s = (t_1 \times t_2) \vee (s_1 \times s_2) \) as \( \pi(s) = \{(t_1, t_2), (s_1, s_2)\} \). We can then compute easily a type \( \pi_e(s) \) for \( \pi_i(e) \) as \( \pi_e(s) = t_i \vee s_i \) (we used boldface symbols to distinguish these type operators from the projections used in expressions). In the calculus considered here, the situation becomes more complex because of type variables, especially top level ones. Let \( s \) be a pair type that contains a top-level variable \( \alpha \). Since \( \alpha \not\leq \emptyset \times \emptyset \) and \( s \leq \emptyset \times \emptyset \), then it is not possible that \( s \simeq s' \vee \alpha \). In other terms the top-level variable cannot appear alone in a union: it must occur intersected with some product type so that it does not “overtake” the \( \emptyset \times \emptyset \) bound. Consequently, we have \( s \simeq s' \wedge \alpha \) for some \( s' \leq \emptyset \times \emptyset \). However, in a typing derivation starting from \( \Delta \vdash e : s \) and ending with \( \Delta \vdash \pi_i(e) : t \), there exists an intermediary step where \( e \) is assigned a type of the form \( (t_1 \times t_2) \) (verifying \( s \leq (t_1 \times t_2) \)) before applying the projection rule. So it is necessary to get rid of the top-level variables of \( s \) (using subsumption) before computing the projection. The example above shows that \( \alpha \) does not play any role since it is the \( s' \) component that will be used to subsume \( s \) to a product type. To say it otherwise, since \( e \) has type \( s \) for all possible assignment of \( \alpha \), then the typing derivation must hold also for \( \alpha = \emptyset \). In whatever way we look at it, the top-level type variables are useless and can be safely discarded when decomposing \( s \).

Given a type \( t \), we write \( \text{dnf}(t) \) for a disjunctive normal form of \( t \), which is defined in [3]. Formally, we define the decomposition of a pair type as follows:

**Definition C.2.** Let \( \tau \) be a disjunctive normal form such that \( \tau \leq \emptyset \times \emptyset \). We define the decomposition of \( \tau \) as follows:

\[
\begin{align*}
\pi(\bigvee_{i \in I} \tau_i) &= \bigcup_{i \in I} \pi(\tau_i) \\
\pi(\bigwedge_{p \in P} (t_1^p \times t_2^p) \wedge \bigwedge_{n \in N} \neg(t_1^p \times t_2^p) \wedge \bigwedge_{n \in N} \neg\alpha \wedge \bigwedge_{n \in N} \neg\alpha' \quad (|P| > 0)) &= \pi(\bigvee_{N \subseteq N \cap P} (\bigwedge_{j \in P} t_1^j \wedge \bigwedge_{k \in N \setminus N, N} \neg t_1^j)) \\
\pi((t_1 \times t_2)) &= \begin{cases} \\
\{ (t_1, t_2) \} & t_1 \neq 0 \text{ and } t_2 \neq 0 \\ 
\emptyset & \text{otherwise} \\
\end{cases}
\end{align*}
\]

and the \( i \)-th projection as \( \pi_i(\tau) = \bigvee_{(s_1, s_2) \subseteq (\text{dnf}(\tau))} s_i \).

For all type \( t \) such that \( t \leq \emptyset \times \emptyset \), the decomposition of \( t \) is defined as

\[
\pi(t) = \pi(\text{dnf}(\emptyset \times \emptyset \wedge t))
\]

and the \( i \)-th projection as \( \pi_i(t) = \bigvee_{(s_1, s_2) \subseteq (\text{dnf}(t \times t))} s_i \).

The decomposition of a union of pair types is the union of each decomposition. When computing the decomposition of an intersection of product types and top-level type variables, we compute all the possible distributions of the intersections over the products, and we discard the top-level variables, as discussed above. Finally, the decomposition of a product is the pair of two components, provided that both components are not empty.

We now prove that the top-level type variables can be safely eliminated in a well-founded (convex) model with infinite support (see [3] for the definitions of model, convexity and infinite support).

**Lemma C.3.** Let \( \leq \) be the subtyping relation induced by a well-founded (convex) model with infinite support. Then

\[
\begin{align*}
\bigwedge_{p \in P} (t_p \times s_p) \wedge \alpha \leq \bigvee_{n \in N} (t_n \times s_n) & \iff \bigwedge_{p \in P} (t_p \times s_p) \leq \bigvee_{n \in N} (t_n \times s_n) \\
\end{align*}
\]

**Proof.** The result trivially holds if \( \bigwedge_{p \in P} (t_p \times s_p) = \emptyset \) or \(|P| = 0 \) (ie, \( \bigwedge_{p \in P} (t_p \times s_p) = \emptyset \)). Let us examine the remaining cases:

\( \Leftarrow \): straightforward.

\( \Rightarrow \): Assume that \( \bigwedge_{p \in P} (t_p \times s_p) \not\leq \bigvee_{n \in N} (t_n \times s_n) \). Let \( \tau \) be the type \( \bigwedge_{p \in P} (t_p \times s_p) \wedge \bigwedge_{n \in N} \neg(t_n \times s_n) \).

Then there exists an assignment \( \eta \) such that \( [\tau] \eta \neq 0 \) (see the subtyping relation defined in [3]). Using the procedure \text{explore}\_\text{pos} defined in the proof of Lemma 3.23 in [3], we can generate an element \( d \)
belonging to \([\tau \eta]\). The procedure explore_pos also generates an assignment \(\eta_0\) for the type variables in \(\text{var}(\tau)\). We define \(\eta'\) such that \(\eta'(\alpha) = \eta_0(\alpha) \cup \{d\}\), \(\eta'(-\alpha) = \eta_0(-\alpha) \setminus \{d\}\), and \(\eta' = \eta_0\) otherwise. Then we have \([\tau \land \alpha] \eta' \neq \emptyset\), which implies \(\bigwedge_{p \in P} (t_p \times s_p) \land \alpha \not\in \bigvee_{n \in N} (t_n \times s_n)\). The result follows by the contrapositive.

The decomposition of pair types has the following properties:

**Lemma C.4.** Let \(\leq\) be the subtyping relation induced by a well-founded (convex) model with infinite support and \(t\) a type such that \(t \leq 1 \times 1\). Then

1. For all \((t_1, t_2) \in \pi(t)\), we have \(t_1 \neq 0\) and \(t_2 \neq 0\).
2. For all \(s_1, s_2\), we have \(t \leq (s_1 \times s_2) \iff \bigvee_{(t_1, t_2) \in \pi(t)} (t_1 \times t_2) \leq (s_1 \times s_2)\).

**Proof.**

1. Straightforward.
2. Since \(t \leq 1 \times 1\), we have

\[
t \cong \bigvee_{(P,N) \in \text{dom}(t)} ((1 \times 1) \land \bigwedge_{j \in P \setminus V} (t_1^j \times t_2^j) \land \bigwedge_{k \in N \setminus V} \neg(t_1^k \times t_2^k) \land \bigwedge_{\alpha \in P \setminus V} \alpha \land \bigwedge_{\alpha' \in N \setminus V} \neg\alpha')
\]

If \(t \cong 0\), then \(\pi(t) = \emptyset\), and the result holds. Assume that \(t \neq 0\), \(|P| > 0\) and each summand of \(\text{dnf}(t)\) is not equivalent to \(0\) as well. Let \(\tau^{-1}\) denote the type \(\bigvee_{(P,N) \in \text{dom}(t)} (\bigwedge_{j \in P \setminus V} (t_1^j \times t_2^j) \land \bigwedge_{k \in N \setminus V} \neg(t_1^k \times t_2^k))\).

Using the set-theoretic interpretation of types we have that \(\tau^{-1}\) is equivalent to

\[
\bigvee_{(P,N) \in \text{dom}(t)} (\bigvee_{N' \subseteq N \setminus V} ((\bigwedge_{j \in P \setminus V} t_1^j \land \bigwedge_{k \in N' \setminus V} \neg t_2^k) \times (\bigwedge_{j \in P \setminus V} t_2^j \land \bigwedge_{k \in (N \setminus V) \setminus N'} \neg t_1^k)))
\]

This means that, \(\tau^{-1}\) is a equivalent to a union of product types. Let us rewrite this union more explicitly, that is, \(\tau^{-1}\) is equivalent to \(\bigvee_{i \in I} (t_1^i \times t_2^i)\) obtained as follows

\[
\bigvee_{(P,N) \in \text{dom}(t)} (\bigvee_{N' \subseteq N \setminus V} ((\bigwedge_{j \in P \setminus V} t_1^j \land \bigwedge_{k \in N' \setminus V} \neg t_2^k) \times (\bigwedge_{j \in P \setminus V} t_2^j \land \bigwedge_{k \in (N \setminus V) \setminus N'} \neg t_1^k)))
\]

We have

\(\pi(t) = \{(t_1^i, t_2^i) \mid i \in I \text{ and } t_1^i \neq 0 \text{ and } t_2^i \neq 0\}\).

Finally, for all pair of types \(s_1\) and \(s_2\), we have

\[
t \leq (s_1 \times s_2) \iff \bigvee_{(P,N) \in \text{dom}(t)} (\bigwedge_{j \in P \setminus V} (t_1^j \times t_2^j) \land \bigwedge_{k \in N \setminus V} \neg(t_1^k \times t_2^k) \land \bigwedge_{\alpha \in P \setminus V} \alpha \land \bigwedge_{\alpha' \in N \setminus V} \neg\alpha') \leq (s_1 \times s_2)
\]

\[
\bigvee_{(P,N) \in \text{dom}(t)} (\bigwedge_{j \in P \setminus V} (t_1^j \times t_2^j) \land \bigwedge_{k \in N \setminus V} \neg(t_1^k \times t_2^k) \land \bigwedge_{\alpha \in P \setminus V} \alpha \land \bigwedge_{\alpha' \in N \setminus V} \neg\alpha' \land \neg(s_1 \times s_2)) \leq 0 \quad \text{(Lemma C.3)}
\]

\[
\bigvee_{(P,N) \in \text{dom}(t)} (\bigwedge_{j \in P \setminus V} (t_1^j \times t_2^j) \land \bigwedge_{k \in N \setminus V} \neg(t_1^k \times t_2^k) \land \neg((s_1 \times s_2)) \leq 0 \quad \text{(Lemma C.3)}
\]

\[
\bigvee_{(t_1, t_2) \in \pi(t)} (t_1 \times t_2) \leq (s_1 \times s_2)
\]

\[
\bigvee_{(t_1, t_2) \in \pi(t)} (t_1 \times t_2) \leq (s_1 \times s_2)
\]

**Lemma C.5.** Let \(s\) be a type such that \(s \leq (t_1 \times t_2)\). Then

1. \(s \leq (\pi_1(s) \times \pi_2(s))\)
2. \(\pi_1(s) \leq t_1\)

**Proof.**

1. According to the proof of Lemma C.4, \(\bigvee_{(s_1, s_2) \in \pi(s)} (s_1 \times s_2)\) is equivalent to the type obtained from \(s\) by ignoring all the top-level type variables. Then it is trivial that \(s \leq \bigvee_{(s_1, s_2) \in \pi(s)} (s_1 \times s_2)\) and then \(s \leq (\pi_1(s) \times \pi_2(s))\).

2. Since \(s \leq (t_1 \times t_2)\), according to Lemma C.4, we have \(\bigvee_{(s_1, s_2) \in \pi(s)} (s_1 \times s_2) \leq (t_1 \times t_2)\). So for all \((s_1, s_2) \in \pi(s)\), we have \((s_1 \times s_2) \leq (t_1 \times t_2)\). Moreover, as \(s_1\) is not empty, we have \(s_1 \leq t_1\). Therefore, \(\pi_1(s) \leq t_1\).

\(\square\)

\(^7\) Strictly speaking, the procedure explore_pos of Lemma 3.24 in [2] supposes \(\tau\) contains only finite product types, but it can be extended to infinite product types by Lemma 3.24 in [2].
Lemma C.6. Let \( t \) and \( s \) be two types such that \( t \leq \emptyset \times \emptyset \) and \( s \leq \emptyset \times \emptyset \). Then \( \pi_i(t \land s) \leq \pi_i(t) \land \pi_i(s) \).

Proof. Let \( t = \bigvee_{j \in J_1} \tau_{j1} \) and \( s = \bigvee_{j \in J_2} \tau_{j2} \) such that
\[
\tau_j = (t_j^1 \times t_j^2) \land \bigwedge_{\alpha \in P_j} \alpha \land \bigwedge_{\alpha' \in N_j} \neg \alpha'
\]
and \( \tau_j \not\simeq 0 \) for all \( j \in J_1 \cup J_2 \). Then we have \( t \land s = \bigvee_{j \in J_1 \cup J_2} \tau_{j1} \land \tau_{j2} \). Let \( j \in J_1 \) and \( j \in J_2 \).

If \( \tau_{j1} \land \tau_{j2} \simeq 0 \), we have \( \pi_i(\tau_{j1} \land \tau_{j2}) = 0 \). Otherwise, \( \pi_i(\tau_{j1} \land \tau_{j2}) = t_j^1 \land t_j^2 = \pi_i(\tau_{j1}) \land \pi_i(\tau_{j2}) \).

For both cases, we have \( \pi_i(\tau_{j1} \land \tau_{j2}) \leq \pi_i(\tau_{j1}) \land \pi_i(\tau_{j2}). \)

Therefore
\[
\pi_i(t \land s) \\
\simeq \bigvee_{j \in J_1 \cup J_2} \pi_i(\tau_{j1} \land \tau_{j2}) \\
\leq \bigvee_{j \in J_1 \cup J_2} \pi_i(\tau_{j1}) \land \pi_i(\tau_{j2}) \\
\leq (\bigvee_{j \in J_1} \pi_i(\tau_{j1})) \land (\bigvee_{j \in J_2} \pi_i(\tau_{j2})) \\
= \pi_i(t) \land \pi_i(s)
\]

Lemma C.7. Let \( t \) be a type and \( \sigma \) be a type substitution such that \( t \leq \emptyset \times \emptyset \). Then \( \pi_i(t \sigma) \leq \pi_i(t) \sigma \)

Proof. We put \( t \) into its disjunctive normal form \( \bigvee_{j \in J} \tau_j \) such that
\[
\tau_j = (t_j^1 \times t_j^2) \land \bigwedge_{\alpha \in P_j} \alpha \land \bigwedge_{\alpha' \in N_j} \neg \alpha'
\]
and \( \tau_j \not\simeq 0 \) for all \( j \in J \). Then we have \( t \sigma = \bigvee_{j \in J} \tau_j \sigma \). So \( \pi_i(t \sigma) = \bigvee_{j \in J} \pi_i(\tau_j \sigma). \)

If \( \tau_j \sigma \simeq 0 \), then \( \pi_i(\tau_j \sigma) = 0 \) and trivially \( \pi_i(t \sigma) \leq \pi_i(t \sigma) \).

Otherwise, we have \( \tau_j \sigma = (t_j^1 \sigma \times t_j^2 \sigma) \land (\bigwedge_{\alpha \in P_j} \alpha \land \bigwedge_{\alpha' \in N_j} \neg \alpha') \).

By Lemma C.6, we get \( \pi_i(\tau_j \sigma) \leq t_j^1 \sigma \land \pi_i(t_j^2 \sigma) \).

Therefore, \( \bigvee_{j \in J} \pi_i(\tau_j \sigma) \leq \bigvee_{j \in J} \pi_i(\tau_j) \sigma, \) that is, \( \pi_i(t \sigma) \leq \pi_i(t \sigma) \).

Lemma C.8. Let \( t \) be a type such that \( t \leq \emptyset \times \emptyset \) and \( [\sigma_k]_{k \in K} \) be a set of type substitutions. Then \( \pi_i([\sigma_k]_{k \in K} \pi_i(t) \sigma_k) \leq \pi_i([\sigma_k]_{k \in K} \pi_i(t) \sigma_k) \)

Proof. Consequence of Lemmas C.6 and C.7

C.2.2 Function types

A type \( t \) is a function type if \( t \leq 0 \to \emptyset \). In order to type the application of a function having a function type \( t \), we need to determine the domain of \( t \), that is, the set of values the function can be safely applied to.

This problem has been solved for ground function types in [9]. Again, the problem becomes more complex if \( t \) contains top-level type variables. Another issue is to determine what is the result type of an application of a function type \( t \) to an argument of type \( s \) (where \( s \) belongs to the domain of \( t \)), knowing that both \( t \) and \( s \) may contain type variables.

Following the same reasoning as with pair types, if a function type \( t \) contains a top-level variable \( \alpha \), then \( t \cong t' \land \alpha \) for some function type \( t' \).

In a typing derivation for a judgement \( \Delta \vdash \pi \Gamma \vdash t : t \) which contains \( \Delta \vdash \Gamma \vdash e_1, e_2 : t \) with \( \Delta \vdash e_1 \to e_2 \) before using the application rule. It is therefore necessary to eliminate the top-level variables from the function type \( t \) before we can type an application. Once more, the top-level variables are useless when computing the domain of \( t \) and can be safely discarded.

Formally, we define the domain of a function type as follows:

**Definition C.9 (Domain).** Let \( \tau \) be a disjunctive normal form such that \( \tau \leq 0 \to \emptyset \). We define \( \text{dom}(\tau) \), the domain of \( \tau \), as:
\[
\text{dom}(\bigvee_{i \in I} \tau_i) = \bigwedge_{i \in I} \text{dom}(\tau_i)
\]
\[
\text{dom}(\bigwedge_{j \in J} (t_j^1 \to t_j^2) \land \bigwedge_{k \in K} \neg (t_k^1 \to t_k^2) \land \bigwedge_{\alpha' \in P_V} \alpha' \land \bigwedge_{\alpha' \in N_V} \neg \alpha')
\]
\[
= \begin{cases} 
\emptyset & \text{if } \bigwedge_{j \in J} (t_j^1 \to t_j^2) \land \bigwedge_{k \in K} \neg (t_k^1 \to t_k^2) \land \bigwedge_{\alpha' \in P_V} \alpha' \land \bigwedge_{\alpha' \in N_V} \neg \alpha' \simeq 0 \\
\bigvee_{j \in J} t_j^1 & \text{otherwise}
\end{cases}
\]

For any type \( t \) such that \( t \leq 0 \to \emptyset \), the domain of \( t \) is defined as
\[
\text{dom}(t) = \text{dom}(\text{dnf}(0 \to \emptyset) \land t)
\]

We also define a decomposition operator \( \Phi \) that —akin to the decomposition operator \( \pi \) for product types— decomposes a function type into a finite set of pairs:
Definition C.10. Let \( \tau \) be a disjunctive normal form such that \( \tau \leq 0 \rightarrow \bot \). We define the decomposition of \( \tau \) as:

\[
\phi(\bigvee_{i \in I} \tau_i) = \bigcup_{i \in I} \phi(\tau_i)
\]

\[
\phi\left(\bigwedge_{j \in P}(t_1 \rightarrow t_2^j) \land \bigwedge_{k \in N}(\neg t_1 \rightarrow t_2^k) \land \bigwedge_{\alpha \in P \cup N} \alpha \land \bigwedge_{\alpha' \in N \setminus V} -\alpha'\right) = \begin{cases} 
\emptyset & \text{if } \bigwedge_{j \in P}(t_1 \rightarrow t_2^j) \land \bigwedge_{k \in N}(\neg t_1 \rightarrow t_2^k) \land \bigwedge_{\alpha \in P \cup N} \alpha \land \bigwedge_{\alpha' \in N \setminus V} -\alpha' \preceq 0 \\
\{(\bigvee_{j \in P^r} t_1^j, \bigwedge_{j \in P \setminus P^r} t_2^j) \mid P^r \subseteq P\} & \text{otherwise}
\end{cases}
\]

For any type \( t \) such that \( t \leq 0 \rightarrow \bot \), the decomposition of \( t \) is defined as:

\[
\phi(t) = \phi(\text{dnf}(0 \rightarrow \bot) \land t).
\]

The set \( \phi(t) \) satisfies the following fundamental property: for every arrow type \( s \rightarrow s' \), the constraint \( t \leq s \rightarrow s' \) holds if and only if \( t \preceq \text{dom}(t) \) holds and for all \((t_1, t_2) \in \phi(t)\), either \( t_1 \leq t_1 \) or \( t_2 \leq t_2 \) hold (see Lemma C.12). As a result, the minimum type \( t \cdot s = \min\{s' \mid t \leq s \rightarrow s'\} \) exists, and it is defined as the union of all \( t_2 \) such that \( s \not\leq t_1 \) and \((t_1, t_2) \in \phi(t)\) (see Lemma C.13). The type \( t \cdot s \) is used to type the application of an expression of type \( t \) to an expression of type \( s \).

As with pair types, in a well-founded (convex) model with infinite support, we can safely eliminate the top-level type variables.

Lemma C.11. Let \( \leq \) be the subtyping relation induced by a well-founded (convex) model with infinite support. Then

\[
\bigwedge_{p \in P} (t_p \rightarrow s_p) \land \alpha \leq \bigvee_{n \in N} (t_n \rightarrow s_n) \iff \bigwedge_{p \in P} (t_p \rightarrow s_p) \leq \bigvee_{n \in N} (t_n \rightarrow s_n)
\]

Proof. Similar to the proof of Lemma C.3.

Lemma C.12. Let \( \leq \) be the subtyping relation induced by a well-founded (convex) model with infinite support and \( t \) a type such that \( t \leq 0 \rightarrow \bot \). Then

\[
\forall s_1, s_2 . (t \leq s_1 \rightarrow s_2) \iff \begin{cases} 
\text{dom}(t) \ni \forall (t_1, t_2) \in \phi(t). (s_1 \preceq t_1) \text{ or } (t_2 \preceq s_2)
\end{cases}
\]

Proof. Since \( t \leq 0 \rightarrow \bot \), we have

\[
t \simeq \bigvee_{(P, N) \in \text{dom}(t)} ((0 \rightarrow \bot) \land \bigwedge_{j \in P \cup N} (t_1^j \rightarrow t_2^j) \land \bigwedge_{k \in N \setminus V} (\neg t_1^j \rightarrow t_2^k) \land \bigwedge_{\alpha \in P \cup N} \alpha \land \bigwedge_{\alpha' \in N \setminus V} -\alpha'\)
\]

If \( t = 0 \), then \( \text{dom}(t) = 0 \), \( \phi(t) = \emptyset \), and the result holds. If \( t \simeq t_1 \lor t_2 \), then \( t_1, t_2 \leq 0 \rightarrow \bot \), \( \text{dom}(t) = \text{dom}(t_1) \cup \text{dom}(t_2) \) and \( \phi(t) = \phi(t_1) \cup \phi(t_2) \). So the result follows if it also holds for \( t_1 \) and \( t_2 \). Thus, without loss of generality, we can assume that \( t \) has the following form:

\[
t \simeq \bigwedge_{j \in P} (t_1^j \rightarrow t_2^j) \land \bigwedge_{k \in N \setminus V} (\neg t_1^j \rightarrow t_2^k) \land \bigwedge_{\alpha \in P \cup N} \alpha \land \bigwedge_{\alpha' \in N \setminus V} -\alpha'
\]

where \( P \neq \emptyset \) and \( t \neq 0 \). Then \( \text{dom}(t) = \bigvee_{j \in P} t_1^j \land \phi(t) = \{(\bigvee_{j \in P^r} t_1^j, \bigwedge_{j \in P \setminus P^r} t_2^j) \mid P^r \subseteq P\} \). For every pair of types \( s_1 \) and \( s_2 \), we have

\[
\begin{align*}
\text{dom}(t) & \leq (s_1 \rightarrow s_2) \\
\iff & \bigwedge_{j \in P} (t_1^j \rightarrow t_2^j) \land \bigwedge_{k \in N \setminus V} (\neg t_1^j \rightarrow t_2^k) \land \bigwedge_{\alpha \in P \cup N} \alpha \land \bigwedge_{\alpha' \in N \setminus V} -\alpha' \leq (s_1 \rightarrow s_2) \quad (\text{Lemma C.11}) \\
\iff & \bigwedge_{j \in P} (t_1^j \rightarrow t_2^j) \land \bigwedge_{k \in N \setminus V} (\neg t_1^j \rightarrow t_2^k) \leq (s_1 \rightarrow s_2) \quad (\text{Lemma C.11}) \\
\iff & \forall P' \subseteq P. \left( s_1 \leq \bigvee_{j \in P'} t_1^j \right) \lor \left( P \neq P' \land \bigwedge_{j \in P \setminus P'} t_2^j \leq s_2 \right) \\
\end{align*}
\]

where \( t' = \bigwedge_{j \in P} (t_1^j \rightarrow t_2^j) \land \bigwedge_{k \in N \setminus V} (\neg t_1^j \rightarrow t_2^k) \).

Lemma C.13. Let \( t \) and \( s \) be two types. If \( t \leq s \rightarrow \bot \), then \( t \leq s \rightarrow s' \) has a smallest solution \( s' \), which is denoted as \( t \cdot s \).
Lemma C.16. Let $t$ be a type such that $t \leq 0 \rightarrow \bot$. Then
\[
\begin{align*}
\text{(1)} & \quad \text{if } s' \leq s \leq \text{dom}(t), \text{ then } t \cdot s' \leq t \cdot s. \\
\text{(2)} & \quad \text{if } t' \leq t, s \leq \text{dom}(t') \text{ and } s' \leq \text{dom}(t), \text{ then } t' \cdot s \leq t \cdot s.
\end{align*}
\]
Proof. (1) Because $s' \leq s$, we have $s \rightarrow t \cdot s \leq s' \rightarrow t \cdot s$. By definition of $t \cdot s$, we have $t \leq t \cdot s$, therefore $t \cdot s' = t \cdot s$. Consequently, we have $t \cdot s' \leq t \cdot s$ by definition of $t \cdot s'$. \\
(2) By definition, we have $t \leq t \cdot s$, which implies $t' \leq s \rightarrow t \cdot s$. Therefore, $t \cdot s$ is a solution to $t' \leq s \rightarrow s'$, hence we have $t' \cdot s \leq t \cdot s$.

Lemma C.17. Let $t$ and $s$ be two types such that $t \leq 0 \rightarrow \bot$ and $s \leq 0 \rightarrow \bot$. Then $\text{dom}(t) \lor \text{dom}(s) \leq \text{dom}(t \land s)$.

Proof. Let $t = \bigvee_{i \in I} t_i$ and $s = \bigvee_{i \in I} s_i$ such that $t_i \neq 0$ for all $i \in I_1 \cup I_2$. Then we have $t \land s = \bigvee_{i \in I \land I_2} t_i \land s_i$. Let $t_i \in I_1$ and $s_i \in I_2$. If $t_i \land s_i = 0$, then $\text{dom}(t_i \land s_i) = \bot$. Otherwise, $\text{dom}(t_i \land s_i) = \text{dom}(t_i) \lor \text{dom}(s_i)$ in both cases, we have $\text{dom}(t_1 \land s_1) \geq \text{dom}(t_1) \lor \text{dom}(s_1)$. Therefore
\[
\begin{align*}
\text{dom}(t \land s) & = \bigvee_{i \in I} \bigvee_{i \in I_1 \land I_2} \text{dom}(t_i) \lor \text{dom}(s_i) \\
& = \bigvee_{i \in I} \bigvee_{i \in I_1} \text{dom}(t_i) \lor \bigvee_{i \in I_2} \text{dom}(s_i) \\
& = \bigvee_{i \in I} \bigvee_{i \in I_1} \text{dom}(t_i) \lor \bigvee_{i \in I_2} \text{dom}(s_i) \\
& = \bigvee_{i \in I} \bigvee_{i \in I_1} \text{dom}(t_i) \lor \bigvee_{i \in I_2} \text{dom}(s_i) \\
& = \bigvee_{i \in I} \text{dom}(t_i) \lor \text{dom}(s_i) \\
& \leq \text{dom}(t) \lor \text{dom}(s).
\end{align*}
\]
Similarly, $\text{dom}(t \land s) \geq \text{dom}(s)$. Therefore $\text{dom}(t) \lor \text{dom}(s) \leq \text{dom}(t \land s)$.

Lemma C.18. Let $t$ be a type and $\sigma$ be a type substitution such that $t \leq 0 \rightarrow \bot$. Then $\text{dom}(t) \sigma \leq \text{dom}(t \sigma)$.

Proof. We put $t$ into its disjunctive normal form $\bigvee_{i \in I} t_i$ such that $t_i \neq 0$ for all $i \in I$. Then we have $t \sigma = \bigvee_{i \in I} t_i \sigma$. So $\text{dom}(t \sigma) = \bigvee_{i \in I} \text{dom}(t_i \sigma)$. Let $i \in I$. If $t_i \sigma \simeq 0$, then $\text{dom}(t_i \sigma) = \bot$. Otherwise, let $t_i = \bigvee_{j \in P} t^1_i \land \bigvee_{k \in N} \neg t^2_i \land \bigvee_{n \in P} \alpha \land \bigvee_{n' \in N} \neg \alpha'$. Then $\text{dom}(t_i) = \bigvee_{j \in P} t^1_i$ and $\text{dom}(t_i \sigma) = \bigvee_{j \in P} t^1_i \sigma \lor \text{dom}(\bigvee_{n \in P} \alpha \land \bigvee_{n' \in N} \neg \alpha' \sigma \land 0 \rightarrow \bot)$. In both cases, we have $\text{dom}(t_i \sigma) \leq \text{dom}(t_i \sigma)$.

Lemma C.19. Let $t_1, s_1, t_2, s_2$ be types such that $t_1 \land s_1$ and $t_2 \land s_2$ exist. Then $(t_1 \land t_2) \cdot (s_1 \land s_2)$ exists and $(t_1 \land t_2) \cdot (s_1 \land s_2) \leq (t_1 \cdot s_1) \land (t_2 \cdot s_2)$.
Proof. According to Lemma C.13, we have $s_1 \leq \text{dom}(t_1)$ and $t_1 \leq s_1 \rightarrow (t_1 \cdot s_1)$. Then by Lemma C.15, we get $s_1 \land s_2 \leq \text{dom}(t_1) \land \text{dom}(t_2) \leq \text{dom}(t_1 \land t_2)$. Moreover, $t_1 \land t_2 \leq (s_1 \rightarrow (t_1 \cdot s_1) \land (s_2 \rightarrow (t_2 \cdot s_2))) \leq (s_1 \land s_2) \rightarrow ((t_1 \cdot s_1) \land (t_2 \cdot s_2))$. Therefore, $(t_1 \land t_2) \cdot (s_1 \land s_2)$ exists and $(t_1 \land t_2) \cdot (s_1 \land s_2) \leq (t_1 \cdot s_1) \land (t_2 \cdot s_2)$. \boxend

Lemma C.19. Let $t$ and $s$ be two types such that $t \cdot s$ exists and $\sigma$ be a type substitution. Then $(t\sigma) \cdot (s\sigma)$ exists and $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$.

Proof. Because $t \cdot s$ exists, we have $s \leq \text{dom}(t)$ and $t \leq s \rightarrow (t \cdot s)$. Then $s\sigma \leq \text{dom}(t)\sigma$ and $t\sigma \leq s\sigma \rightarrow (t \cdot s)\sigma$. By Lemma C.16, we get $\text{dom}(t)\sigma \leq \text{dom}(t)\sigma$. So $s\sigma \leq \text{dom}(t)\sigma$. Therefore, $(t\sigma) \cdot (s\sigma)$ exists. Moreover, since $(t \cdot s)\sigma$ is a solution to $t\sigma \leq s\sigma \rightarrow s'\sigma$, by definition, we have $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$. \boxend

Lemma C.20. Let $t$ and $s$ be two types and $[\sigma_k]_{k \in K}$ be a set of type substitutions such that $t \cdot s$ exists. Then $(\bigwedge_{k \in K} t\sigma_k) \cdot (\bigwedge_{k \in K} s\sigma_k)$ exists and $(\bigwedge_{k \in K} t\sigma_k) \cdot (\bigwedge_{k \in K} s\sigma_k) \leq (\bigwedge_{k \in K} (t\cdot s)\sigma_k)$.

Proof. According to Lemmas C.19 and C.18, $(\bigwedge_{k \in K} t\sigma_k) \cdot (\bigwedge_{k \in K} s\sigma_k)$ exists. Moreover, 
\[
\begin{align*}
\bigwedge_{k \in K} (t \cdot s)\sigma_k &\geq (\bigwedge_{k \in K} (t\sigma_k \cdot s\sigma_k)) \quad \text{(Lemma C.19)} \\
&\geq (\bigwedge_{k \in K} t\sigma_k) \cdot (\bigwedge_{k \in K} s\sigma_k) \quad \text{(Lemma C.18)}
\end{align*}
\]
\]

C.2.3 Syntax-Directed Rules

Because of subsumption, the typing rules provided in Section A.3 are not syntax-directed and so they cannot not yield a type-checking algorithm directly. Generally, subsumption is used to bridge gaps between the types expected by functions and the actual types of their arguments in applications (in simply type $\lambda$-calculus) \[\text{[11]}\]. In our calculus, we identify four situations where the subsumption is needed, namely, the rules for projections, abstractions, applications, and type cases. To see why, we consider a typing derivation ending with each typing rule whose immediate sub-derivation ends with (subsum). For each case, we explain how the use of subsumption can be pushed through the typing rule under consideration, or how the rule should be modified to take subtyping into account.

First we consider the case where a typing derivation ends with (subsum) whose immediate sub-derivation also ends with (subsum). The two consecutive uses of (subsum) can be merged into one, because the subtyping relation is transitive.

Lemma C.21. If $\Delta \vdash \Gamma \vdash e : t$, then there exists a derivation for $\Delta \vdash \Gamma \vdash e : t$ where there are no consecutive instances of (subsum).

Proof. Assume that there exist two consecutive instances of (subsum) occurring in a derivation of $\Delta \vdash \Gamma \vdash e : t$, that is,
\[
\begin{array}{l}
\Delta' \vdash \Gamma' \vdash e' : s_2' \\
\Delta' \vdash \Gamma' \vdash e' : s_1' \quad \text{(subsum)} \\
\Delta' \vdash \Gamma' \vdash e' : t' \\
\end{array}
\]
\[
\begin{array}{l}
\vdots \\
\Delta \vdash \Gamma \vdash e : t \\
\end{array}
\]

Since $s_2' \leq s_1' \leq t'$, we have $s_2' \leq t'$. So we can rewrite this derivation as follows:
\[
\begin{array}{l}
\Delta' \vdash \Gamma' \vdash e' : s_2' \\
\Delta' \vdash \Gamma' \vdash e' : t' \quad \text{(subsum)} \\
\vdots \\
\Delta \vdash \Gamma \vdash e : t \\
\end{array}
\]

Therefore, the result follows. \boxend

Next, consider an instance of (pair) such that one of its sub-derivation ends with an instance of (subsum), for example, the left sub-derivation:
\[
\begin{array}{l}
\Delta \vdash \Gamma \vdash e_1 : s_1 \\
\Delta \vdash \Gamma \vdash e_1 : t_1 \quad \text{(subsum)} \\
\Delta \vdash \Gamma \vdash e_2 : t_2 \\
\end{array}
\]
\[
\Delta \vdash \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \quad \text{(pair)}
\]

51 2012/7/11
As \( s_1 \leq t_1 \), we have \( s_1 \times t_2 \leq t_1 \times t_2 \). Then we can move subsumption down through the rule \((pair)\), giving the following derivation:

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : s_1 & \quad \Delta \Gamma \vdash e_2 : t_2 \\
\Delta \Gamma \vdash (e_1, e_2) : (s_1 \times t_2) & \quad (pair) \\
\Delta \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) & \quad (subsum)
\end{align*}
\]

The rule \((proj)\) is a little trickier than \((pair)\). Consider the following derivation:

\[
\begin{align*}
\Delta \Gamma \vdash e : s \quad s \leq t_1 \times t_2 & \quad (subsum) \\
\Delta \Gamma \vdash e : (t_1 \times t_2) & \quad (proj)
\end{align*}
\]

As \( s \leq t_1 \times t_2 \), \( s \) is a pair type. According to the decomposition of \( s \) and Lemma \([C.5]\), we can rewrite the previous derivation into the following one:

\[
\begin{align*}
\Delta \Gamma \vdash e : s & \quad s \leq \mathbb{1} \times \mathbb{1} \\
\Delta \Gamma \vdash e : \pi_i(e) : \pi_i(s) & \quad \pi_i(s) \leq t_i \\
\Delta \Gamma \vdash e : \pi_i(e) : t_i & \quad (proj)
\end{align*}
\]

Note that the subtyping check \( s \leq \mathbb{1} \times \mathbb{1} \) ensures that \( s \) is a pair type.

Next consider an instance of \((abstr)\) (where \( \Delta' = \Delta \cup \text{var}(\bigwedge_{i \in I, j \in J} (t_i \sigma_j \rightarrow s_i \sigma_j)) \)). All the sub-derivations may end with \((subsum)\):

\[
\begin{align*}
\forall i \in I, j \in J. \quad \Delta' \Gamma, (x : t_i \sigma_j) \vdash e[\sigma_j] : s'_{ij} & \quad s'_{ij} \leq s_i \sigma_j \\
\Delta' \Gamma, (x : t_i \sigma_j) \vdash e[\sigma_j] : s_i \sigma_j & \quad \text{(abstr)}
\end{align*}
\]

Without subsumption, we would assign the type \( \bigwedge_{i \in I, j \in J} (t_i \sigma_j \rightarrow s'_{ij}) \) to the abstraction, while we want to assign the type \( \bigwedge_{i \in I, j \in J} (t_i \sigma_j \rightarrow s_i \sigma_j) \) to it because of the type annotations. Consequently, we have to keep the subtyping checks \( s'_{ij} \leq s_i \sigma_j \) as side-conditions of an algorithmic typing rule for abstractions.

\[
\begin{align*}
\forall i \in I, j \in J. \quad \Delta' \Gamma, (x : t_i \sigma_j) \vdash e[\sigma_j] : s'_{ij} & \quad s'_{ij} \leq s_i \sigma_j \\
\Delta' \Gamma, (x : t_i \sigma_j) \vdash e[\sigma_j] : s_i \sigma_j & \quad \text{(abstr)}
\end{align*}
\]

In \((appl)\) case, suppose that both sub-derivations end with \((subsum)\):

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t \quad t \leq t' & \quad (appl) \\
\Delta \Gamma \vdash e_1 : t' \rightarrow s' & \quad \Delta \Gamma \vdash e_2 : t' \\
\Delta \Gamma \vdash e_1 e_2 : s' & \quad (appl)
\end{align*}
\]

Since \( s \leq t' \), then by the contravariance of arrow types we have \( t' \rightarrow s' \leq s \rightarrow s' \). Hence, such a derivation can be rewritten as

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t \quad t \leq t' & \quad (appl) \\
\Delta \Gamma \vdash e_1 : t' \rightarrow s' & \quad \Delta \Gamma \vdash e_2 : t' \\
\Delta \Gamma \vdash e_1 e_2 : s' & \quad (appl)
\end{align*}
\]

Applying Lemma \([C.21]\), we can merge the two adjacent instances of \((subsum)\) into one:

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t \quad t \leq t' & \quad (appl) \\
\Delta \Gamma \vdash e_1 : s' & \quad \Delta \Gamma \vdash e_2 : s' \\
\Delta \Gamma \vdash e_1 e_2 : s' & \quad (appl)
\end{align*}
\]

A syntax-directed typing rule for applications can then be written as follows

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t \quad \Delta \Gamma \vdash e_2 : s \quad t \leq s & \quad (appl) \\
\Delta \Gamma \vdash e_1 e_2 : s' & \quad \Delta \Gamma \vdash e_1 : s \quad \Delta \Gamma \vdash e_2 : s' \\
\end{align*}
\]

where subsumption is used as a side condition to bridge the gap between the function type and the argument type.

This typing rule is not algorithmic yet, because the result type \( s' \) can be any type verifying the side condition. Using Lemma \([C.12]\), we can equivalently rewrite the side condition as \( t \leq 0 \rightarrow \mathbb{1} \) and \( s \leq \text{dom}(t) \) without involving the result type \( s' \). The first condition ensures that \( t \) is a function type.
and the second one that the argument type \( s \) can be safely applied by \( t \). Moreover, we assign the type \( t \cdot s \) to the application, which is by definition the smallest possible type for it. We obtain then the following algorithmic typing rule.

\[
\frac{\Delta \Gamma \vdash e_1 : t \quad \Delta \Gamma \vdash e_2 : s \quad t \leq 0 \Rightarrow \emptyset \quad s \leq \text{dom}(t)}{\Delta \Gamma \vdash e_1 e_2 : t \cdot s}
\]

Next, let us discuss the rule (case):

\[
\frac{\Delta \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq -t & \Rightarrow \Delta \Gamma \vdash e_1 : s \\ t' \not< t & \Rightarrow \Delta \Gamma \vdash e_2 : s \end{cases}}{\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s}
\]

The rule covers four different situations, depending on which branches of the type-cases are checked: (i) no branch is type-checked, (ii) the first branch \( e_1 \) is type-checked, (iii) the second branch \( e_2 \) is type-checked, and (iv) both branches are type-checked. Each case produces a corresponding algorithmic rule.

In case (i), we have simultaneously \( t' \leq t \) and \( t' \leq -t \), which means that \( t' = 0 \). Consequently, \( e \) does not reduce to a value (otherwise, subject reduction would be violated), and neither does the whole type case. Consequently, we can assign type \( 0 \) to the whole type.

\[
\frac{\ldots}{\Delta \Gamma \vdash e : 0}
\]

Suppose we are in case (ii) and the sub-derivation for the first branch \( e_1 \) ends with (subsum):

\[
\frac{\Delta \Gamma \vdash e : t' \quad t' \leq t}{\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s}
\]

Such a derivation can be rearranged as:

\[
\frac{\Delta \Gamma \vdash e : t' \quad t' \leq t \quad \Delta \Gamma \vdash e_1 : s_1}{\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1}
\]

Moreover, (subsum) might also be used at the end of the sub-derivation for \( e \):

\[
\frac{\Delta \Gamma \vdash e : t'' \quad t'' \leq t''}{\Delta \Gamma \vdash e : t' \quad t' \leq t \quad \Delta \Gamma \vdash e_1 : s_1}
\]

From \( t'' \leq t' \) and \( t' \leq t \), we deduce \( t'' \leq t \) by transitivity. Therefore this use of subtyping can be merged with the subtyping check of the type case rule. We then obtain the following algorithmic rule.

\[
\frac{\ldots \Delta \Gamma \vdash e : t'' \quad t'' \leq t \quad \Delta \Gamma \vdash e_1 : s}{\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s}
\]

We obtain a similar rule for case (iii), except that \( e_2 \) is type-checked instead of \( e_1 \), and \( t'' \) is tested against \(-t\).

Finally, consider case (iv). We have to type-check both branches and each typing derivation may end with (subsum):

\[
\begin{cases}
 t' \not\leq -t & \text{and} & \Delta \Gamma \vdash e_1 : s_1 \\
 t' \not< t & \text{and} & \Delta \Gamma \vdash e_2 : s_2 \end{cases}
\]

Subsumption is used there just to unify \( s_1 \) and \( s_2 \) into a common type \( s \), which is used to type the whole type case. Such a common type can also be obtained by taking the least upper-bound of \( s_1 \) and \( s_2 \), i.e.,
\(s_1 \lor s_2\). Because \(s_1 \leq s\) and \(s_2 \leq s\), we have \(s_1 \lor s_2 \leq s\), and we can rewrite the derivation as follows:

\[
\Delta \Gamma \vdash e : t' \quad \begin{cases} 
  t' \not\subseteq \neg t' \quad \text{and} \quad \Delta \Gamma \vdash e_1 : s_1 \\
  t' \not\subseteq t \\
\end{cases}
\]

\[
\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1 \lor s_2 \quad \text{(case)}
\]

Suppose now that the sub-derivation for \(e\) ends with \((\text{subsum})\):

\[
\Delta \Gamma \vdash e : t'' \quad t'' \leq t'
\]

\[
\Delta \Gamma \vdash e : t' \quad \begin{cases} 
  t' \not\subseteq \neg t' \quad \text{and} \quad \Delta \Gamma \vdash e_1 : s_1 \\
  t' \not\subseteq t \\
\end{cases}
\]

\[
\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1 \lor s_2 \quad \text{(case)}
\]

The relations \(t'' \leq t', t' \not\subseteq \neg t, t'' \leq t, t' \not\subseteq t\) do not necessarily imply \(t'' \not\subseteq \neg t\), and \(t'' \leq t, t' \not\subseteq t\) do not necessarily imply \(t'' \not\subseteq \neg t\). Therefore, by using the type \(t''\) instead of \(t\) for \(e\), we may type-check less branches. If so, then we would be in one of the cases \((i) - (iii)\), and the result type \((i.e., \ \text{a} \ \text{type} \ \text{among} \ \emptyset, s_1 \ \text{or} \ \text{s}_2\) for the whole type case would be smaller than \(s_1 \lor s_2\). It would then be possible to type the case with \(s_1 \lor s_2\) by subsumption. Otherwise, we type-check as much branches with \(t''\) as with \(t\), and we can modify the rule into

\[
\Delta \Gamma \vdash e : t'' \quad \begin{cases} 
  t'' \not\subseteq \neg t' \quad \text{and} \quad \Delta \Gamma \vdash e_1 : s_1 \\
  t'' \not\subseteq t \\
\end{cases}
\]

\[
\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1 \lor s_2
\]

Finally, consider the case where the last rule in a derivation is \((\text{instinter})\) and all its sub-derivations end with \((\text{subsum})\):

\[
\Delta \Gamma \vdash e : s \quad s \leq t \quad \text{(subsum)}
\]

\[
\Delta \Gamma \vdash e : t 
\]

\[
\Delta \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \quad \text{(instinter)}
\]

Since \(s \leq t\), we have \(\bigwedge_{j \in J} \sigma_j \leq \bigwedge_{j \in J} t \sigma_j\). So such a derivation can be rewritten into

\[
\Delta \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \quad \text{(instinter)}
\]

\[
\Delta \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \quad \text{(subsum)}
\]

In conclusion, by applying the aforementioned transformations repeatedly, we can rewrite an arbitrary typing derivation into a special form where subsumption are used at the end of sub-derivations of projections, abstractions or applications, in the conditions of type cases and at the very end of the whole derivation. Thus, this transformations yields a set of syntax-directed typing rules, given in Figure 8 Let \(\Delta \Gamma \vdash e : t\) denote the typing judgments derivable by the set of syntax-directed typing rules.

**Theorem C.22 (Soundness).** Let \(e\) be an expression. If \(\Gamma \vdash e : t\), then \(\Gamma \vdash e : t\).

**Proof.** By induction on the typing derivation of \(\Delta \Gamma \vdash e : t\). We proceed by a case analysis on the last rule used in the derivation.

- (ALG-CONST): straightforward.
- (ALG-VAR): straightforward.
- (ALG-PAIR): consider the derivation

\[
\Delta \Gamma \vdash e_1 : t_1 \quad \Delta \Gamma \vdash e_2 : t_2
\]

\[
\Delta \Gamma \vdash \pi_i(e) : \pi_i(t)
\]

By induction, we have \(\Delta \Gamma \vdash e : t\). According to Lemma C.5 we have \(t \leq (\pi_1(t) \times \pi_2(t))\). Then by (subsum), we get \(\Delta \Gamma \vdash e : (\pi_1(t) \times \pi_2(t))\). Finally, the rule (proj) gives us \(\Delta \Gamma \vdash \pi_i(e) : \pi_i(t)\)
Figure 8. Syntax-Directed Typing Rules
By induction, we have $\Delta \Gamma \vdash e : 0$. No branch is type-checked by the rule (case), so any type can be assigned to the type case expression, and in particular we have $\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : 0$

(Alg-CASE-FST): consider the derivation

\[\Delta \Gamma \vdash_A e : t' \quad t' \leq t \quad \Delta \Gamma \vdash_A e_1 : s_1 \]

By induction, we have $\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1$. As $t' \leq t$, then we only need to type-check the first branch. Therefore, by the rule (case), we have $\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1$.

(Alg-CASE-SND): similar the case of (Alg-CASE-FST).

(Alg-CASE-BOTH): consider the derivation

\[\Delta \Gamma \vdash_A e : t' \begin{cases} t' \leq t \quad \text{and} \quad \Delta \Gamma \vdash_A e_1 : s_1 \cr t' \leq t \quad \text{and} \quad \Delta \Gamma \vdash_A e_2 : s_2 \end{cases} \]

By induction, we have $\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1 \lor s_2$. Moreover, as $t' \leq t$ we have to type-check both branches. Finally, by the rule (case), we get $\Delta \Gamma \vdash (e \in t ? e_1 : e_2) : s_1 \lor s_2$.

(Alg-INST): consider the derivation

\[\Delta \Gamma \vdash_A e : t \quad \forall j \in J. \sigma_j \not\in \Delta \]

\[\Delta \Gamma \vdash_A e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j \]

By induction, we have $\Delta \Gamma \vdash e : t$. As $\forall j \in J. \sigma_j \not\in \Delta$, we get $\Delta \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t \sigma_j$.

\[\square\]

Theorem C.23 (Completeness). Let $\leq$ be a subtyping relation induced by a well-founded (convex) model with infinite support and $e$ an expression. If $\Delta \Gamma \vdash e : t$, then there exists a type $s$ such that $\Delta \Gamma \vdash e : s$ and $s \leq t$.

Proof. By induction on the typing derivation of $\Delta \Gamma \vdash e : t$. We proceed by case analysis on the last rule used in the derivation.

(const): straightforward (take $s$ as $b_e$).

(var): straightforward (take $s$ as $\Gamma(x)$).

(pair): consider the derivation

\[\Delta \Gamma \vdash e_1 : t_1 \quad \Delta \Gamma \vdash e_2 : t_2 \]

\[\Delta \Gamma \vdash (e_1, e_2) : t_1 \times t_2\] (pair)

Applying the induction hypothesis twice, we have $\Delta \Gamma \vdash_A (e_1, e_2) : s_1 \times s_2$. Since $s_1 \leq t_1$, we deduce $(s_1 \times s_2) \leq (t_1 \times t_2)$.

(proj): consider the derivation

\[\Delta \Gamma \vdash e : (t_1 \times t_2) \]

\[\Delta \Gamma \vdash \pi_i(e) : t_i\] (proj)

By induction, there exists $s$ such that $\Delta \Gamma \vdash_A e : s$ and $s \leq (t_1 \times t_2)$. Clearly we have $s \leq \subseteq \times \subseteq$. Applying (Alg-PROJ), we have $\Delta \Gamma \vdash_A \pi_i(e) : \pi_i(s)$. Moreover, as $s \leq (t_1 \times t_2)$, according to Lemma C.5 we have $\pi_i(s) \leq t_i$. Therefore, the result follows.

(app): consider the derivation

\[\Delta \Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Delta \Gamma \vdash e_2 : t_1 \]

\[\Delta \Gamma \vdash_A e_1 e_2 : t \cdot s\] (app)

Applying the induction hypothesis twice, we have $\Delta \Gamma \vdash_A e_1 : t$ and $\Delta \Gamma \vdash_A e_2 : s$ where $t \leq t_1 \rightarrow t_2$ and $s \leq t_1$. Clearly we have $t \leq 0 \rightarrow 1$ and $t \leq s \rightarrow t_2$ (by contravariance of arrows). From Lemma C.12 we get $s \leq \text{dom}(t)$. So, by applying the rule (Alg-APPL), we have $\Delta \Gamma \vdash_A e_1 e_2 : t \cdot s$. Moreover, it is clear that $t_2$ is a solution for $t \leq s \rightarrow s'$. Consequently, it is a super type of $t \cdot s$, that is $t \cdot s \leq t_2$. 

56

2012/7/11
Proof. By induction on the structure of $e$.

\[(abstr): \text{consider the derivation}
\]

\[
\begin{array}{c}
\forall i \in I, j \in J. \Delta^{\#} \Gamma, (x : t[s_j]) \vdash e_0[s_j] : s_i[s_j] \\
\Delta^{\#} \Gamma \vdash \lambda^{e_0}_{i \in I, j \in J} x.e : \bigwedge_{i \in I, j \in J} (t_i[s_j] \rightarrow s_i[s_j])
\end{array}
\]

where $\Delta' = \Delta \cup \text{var}(\lambda^{e_0}_{i \in I, j \in J} t_i[s_j] \rightarrow s_i[s_j])$. By induction, for all $i \in I$ and $j \in J$, there exists $s'_{ij}$ such that $\Delta^{\#} \Gamma, (x : t[s_j]) \vdash e_0[s_j] : s'_{ij}$ and $s'_{ij} \leq s_i[s_j]$. Then the rule (ALG-ABSTR) gives us $\Delta^{\#} \Gamma \vdash \lambda^{e_0}_{i \in I, j \in J} x.e : \bigwedge_{i \in I, j \in J} (t_i[s_j] \rightarrow s_i[s_j])$.

\[(case): \text{consider the derivation}
\]

\[
\begin{array}{c}
\Delta^{\#} \Gamma \vdash e : t' \\
\{ t' \not= t \Rightarrow \Delta^{\#} \Gamma \vdash e : s \\
 \quad \quad \quad \quad \vdots \}
\end{array}
\]

By induction hypothesis on $\Delta^{\#} \Gamma \vdash e : t'$, there exists a type $t''$ such that $\Delta^{\#} \Gamma \vdash e : t''$ and $t'' \leq t$. If $t'' \not= 0$, by (ALG-CASE-NONE), we have $\Delta^{\#} \Gamma \vdash e : t'' : 0$. The result follows straightforwardly. In what follows, we assume that $t'' = 0$.

Assume that $t'' \leq t$. Because $t'' \leq t'$, we have $t' \not= -t$ (otherwise, $t'' = 0$). Therefore the first branch is type-checked, and by induction, there exists a type $s_1$ such that $\Delta^{\#} \Gamma \vdash e : s_1$ and $s_1 \leq t$. Then the rule (ALG-CASE-FST) gives us $\Delta^{\#} \Gamma \vdash e : (t_1 \not= e_1 : e_2) : s_1$.

Otherwise, $t' \not= t$. In this case, we have $t' \not= t$ (otherwise, $t'' \not= 0$). The second branch is type-checked. By induction, there exists a type $s_2$ such that $\Delta^{\#} \Gamma \vdash e : s_2$ and $s_2 \leq s$. If $t'' \leq t$, then by the rule (ALG-CASE-SND), we have $\Delta^{\#} \Gamma \vdash (e \in t \not= e_1 : e_2) : s_2$. The result follows. Otherwise, we also have $t' \not= -t$. Then we also have $t' \not= -t$ (otherwise, $t'' \not= t$). So the first branch should be type-checked as well. By induction, we have $\Delta^{\#} \Gamma \vdash e : s_1$ where $s_1 \leq s$. By applying (ALG-CASE-BOTH), we get $\Delta^{\#} \Gamma \vdash (e \in t \not= e_1 : e_2) : s_1 \land s_2$. Since $s_1 \leq s$ and $s_2 \leq s$, we deduce that $s_1 \lor s_2 \leq s$. The result follows as well.

\[(instinter): \text{consider the derivation}
\]

\[
\begin{array}{c}
\Delta^{\#} \Gamma \vdash e : t \\
\forall j \in J. \sigma_j \not= \Delta \\
\Delta^{\#} \Gamma \vdash e[\sigma_j] \in J : \bigwedge_{j \in J} \tau_{\sigma_j}
\end{array}
\]

By induction, there exists a type $s$ such that $\Delta^{\#} \Gamma \vdash e : s$ and $s \leq t$. Then the rule (ALG-INST) gives us that $\Delta^{\#} \Gamma \vdash e[\sigma_j] \in J : \bigwedge_{j \in J} s_{\sigma_j}$. Since $s \leq t$, we have $\bigwedge_{j \in J} s_{\sigma_j} \leq \bigwedge_{j \in J} \tau_{\sigma_j}$. Therefore, the result follows.

\[\square\]

Corollary C.24 (Minimum typing). Let $e$ be an expression. If $\Delta^{\#} \Gamma \vdash e : t$, then $t = \min\{s \mid \Delta^{\#} \Gamma \vdash e : s\}$.


To prove the termination of the type-checking algorithm, we define the size of an expression $e$ as follows.

Definition C.25. Let $e$ be an expression. We define the size of $e$ as:

\[
\begin{align*}
\text{size}(e) &= 1 \\
\text{size}(x) &= 1 \\
\text{size}(e_1, e_2) &= \text{size}(e_1) + \text{size}(e_2) + 1 \\
\text{size}(\pi_i(e)) &= \text{size}(e) + 1 \\
\text{size}(\lambda^{e_0}_{i \in I, j \in J} x.e) &= \text{size}(e) + 1 \\
\text{size}(e \in t \not= e_1 : e_2) &= \text{size}(e) + \text{size}(e_1) + \text{size}(e_2) + 1 \\
\text{size}(e[\sigma_j] \in J) &= \text{size}(e) + 1
\end{align*}
\]

The relabeling does not enlarge the size of the expression.

Lemma C.26. Let $e$ be an expression and $[\sigma_j]_{j \in J}$ a set of type substitutions. Then $\text{size}(e \circ [\sigma_j]_{j \in J}) \leq \text{size}(e)$.

Proof. By induction on the structure of $e$. \[\square\]

Theorem C.27 (Termination). Let $e$ be an expression. Then the type-checking of $e$ terminates.
We want sets of type-substitutions to be inferred by the system, not written by the programmer. To this end, we define a calculus without type substitutions (called implicitly-typed, in contrast with the calculus of Section A.2, which we henceforth call explicitly-typed), for which we define a type-substitutions inference system. As explained in Section A.2, we do not try to infer decorations in \(\lambda\)-abstractions, and we therefore look for completeness of the type-substitutions inference system with respect to the expressions written according to the following grammar.

\[
e : ::= c \mid x \mid (e_1, e_2) \mid \pi_1(e) \mid e \mid e \mid \lambda^{\alpha \in I_{t_1 \rightarrow s_1}} x.e \mid e \alpha \in t \? e : c \mid e[\sigma_j]_{j \in J}
\]

We write \(\mathcal{E}_0\) for the set of such expressions. The implicitly-typed calculus defined in this section corresponds to the type-substitution erasures of the terms in terms of \(\mathcal{E}_0\). These are the terms generated by the grammar above without using the last production, that is, without the application of sets of type-substitutions. We then define the type-substitutions inference system by determining where the rule (ALG-INST) have to be used in the typing derivations of explicitly-typed expressions. Finally, we propose an incomplete but more tractable restriction of the type-substitutions inference system, which, we believe, is powerful enough to be used in practice.

### D. Implicitly-Typed Calculus

**Definition D.1.** An implicitly-typed expression \(a\) is an expression without any type substitutions. It is inductively generated by the following grammar:

\[
a : ::= c \mid x \mid (a, a) \mid \pi_1(a) \mid a \mid \lambda^{\alpha \in I_{t_1 \rightarrow s_1}} x.a \mid a \alpha \in t \? a : a
\]

where \(t, s_1\) range over types and \(t \in T_0\) is a ground type. We write \(\mathcal{E}_A\) to denote the set of all implicitly-typed expressions.

Clearly, \(\mathcal{E}_A\) is a proper subset of \(\mathcal{E}_0\).

**Definition D.2.** A \(\text{erasure}\) is the mapping from \(\mathcal{E}_0\) to \(\mathcal{E}_A\) defined as

\[
estimate(c) = c
\]

\[
estimate(x) = x
\]

\[
estimate((e_1, e_2)) = (\estimate\(e_1\), \estimate\(e_2\))
\]

\[
estimate(\pi_1(e)) = \pi_1(\estimate\(e\))
\]

\[
estimate(\lambda^{\alpha \in I_{t_1 \rightarrow s_1}} x.e) = \lambda^{\alpha \in I_{t_1 \rightarrow s_1}} x.\estimate\(e\)
\]

\[
estimate(e_1 \alpha_2) = \estimate(e_1) \estimate(e_2)
\]

\[
estimate(e \alpha_1 : e_2) = \estimate(e) \alpha_1 : \estimate(e_2)
\]

\[
estimate(e[\sigma_j]_{j \in J}) = \estimate(e)
\]

Prior to introducing the type inference rules, we define a preorder on types, which is similar to the type variable instantiation in ML but with respect to \(a\) set of type substitutions.

**Definition D.3.** Let \(s\) and \(t\) be two types and \(\Delta\) a set of type variables. We define the following relations:

\[
[s]_{I \in \mathcal{I}} \vdash \iff \bigwedge_{i \in I} s_{\sigma_i} \leq t \text{ and } \forall i \in I, \sigma_i \not\subseteq \Delta
\]

\[
s \subseteq \Delta t \iff \exists [\sigma]_{I \in \mathcal{I}} \text{ such that } [\sigma]_{I \in \mathcal{I}} \vdash \iff s \subseteq \Delta t
\]

We write \(s \not\subseteq \Delta t\) if it does not exist a set of type substitutions \([\sigma]_{I \in \mathcal{I}}\) such that \([\sigma]_{I \in \mathcal{I}} \vdash \iff s \subseteq \Delta t\).

**Lemma D.4.** Let \(t_1\) and \(t_2\) be two types and \(\Delta\) a set of type variables. If \(t_1 \subseteq \Delta s_1\) and \(t_2 \subseteq \Delta s_2\), then \((t_1 \land t_2) \subseteq \Delta (s_1 \land s_2)\) and \((t_1 \times t_2) \subseteq \Delta (s_1 \times s_2)\).
Proof. Let \([\sigma_1] t_i \in T_1 \vdash t_1 \in \Delta \) and \([\sigma_2] t_j \in T_2 \vdash t_2 \in \Delta \). Then

\[
\begin{align*}
\frac{\emptyset \vdash t_1 \in \Delta \quad \emptyset \vdash t_2 \in \Delta}{\vdash t_1 \land t_2 \in \Delta}
\end{align*}
\]

and

\[
\begin{align*}
\frac{\emptyset \vdash t_1 \in \Delta \quad \emptyset \vdash t_2 \in \Delta}{\vdash (t_1 \times t_2) \sigma}
\end{align*}
\]

Lemma D.5. Let \(t_1\) and \(t_2\) be two types and \(\Delta\) a set of type variables such that \((\var{t_1} \setminus \Delta) \cap (\var{t_2} \setminus \Delta) = \emptyset\). If \(t_1 \in \Delta s_1\) and \(t_2 \in \Delta s_2\), then \(t_1 \land t_2 \in \Delta s_1 \lor s_2\).

Proof. Let \([\sigma_{i1}] t_i \in T_1 \vdash t_1 \in \Delta s_1\) and \([\sigma_{i2}] t_j \in T_2 \vdash t_2 \in \Delta s_2\). Then we construct another set of type substitutions \([\sigma_{i1, i2}] t_i \in T_1, t_j \in T_2 \vdash \sigma\) such that

\[
[\sigma_{i1, i2}]\ (\alpha) = \begin{cases} 
\sigma_{i1}(\alpha) & \alpha \in (\var{t_1} \setminus \Delta) \\
\sigma_{i2}(\alpha) & \alpha \in (\var{t_2} \setminus \Delta) \\
\alpha & \text{otherwise}
\end{cases}
\]

So we have

\[
\begin{align*}
\frac{\emptyset \vdash t_1 \in \Delta s_1 \quad \emptyset \vdash t_2 \in \Delta s_2}{\vdash (t_1 \lor t_2) \sigma_{i1, i2}}
\end{align*}
\]

In order to guess where to insert sets of type-substitutions in a implicitly-typed expression, we consider each typing rule of the explicitly-typed calculus used in conjunction with the instantiation rule (ALG-INST). If instantiation can be moved through a given typing rule without affecting typability or changing the result type, then it is not necessary to infer type substitutions at the level of this rule. First of all, notice that two successive instantiations can be safely merged into one.

Lemma D.6. Let \(e\) be an explicitly-typed expression and \([\sigma_{i}] t_i \in T_i\), \([\sigma_{j}] t_j \in T_j\) two sets of type substitutions. Then

\[
\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{j}] t_j : t \iff \Delta \vdash e[\sigma_{i} \circ \sigma_{j}] t_i \vdash e[\sigma_{i} \circ \sigma_{j}] t_j : t
\]

Proof. \(\implies\): consider the following derivation:

\[
\begin{align*}
\frac{\Delta \vdash e \vdash s \quad \sigma_i \notin \Delta}{\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_i : s(\sigma_{i} \circ \sigma_{j})}
\end{align*}
\]

As \(\sigma_i \notin \Delta\), \(\sigma_j \notin \Delta\) and \(\dom(\sigma_i \circ \sigma_j) = \dom(\sigma_i) \cup \dom(\sigma_j)\), we have \(\sigma_i \circ \sigma_j \notin \Delta\). Then by (ALG-INST), we have \(\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_i : s(\sigma_i \circ \sigma_j)\), that is \(\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_i : s(\sigma_i \circ \sigma_j)\), \(\sigma_i \circ \sigma_j \notin \Delta\).

\(\impliedby\): consider the following derivation:

\[
\begin{align*}
\frac{\Delta \vdash e \vdash s \quad \sigma_i \circ \sigma_j \notin \Delta}{\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_j : s(\sigma_i \circ \sigma_j)}
\end{align*}
\]

As \(\sigma_i \circ \sigma_j \notin \Delta\) and \(\dom(\sigma_i \circ \sigma_j) = \dom(\sigma_i) \cup \dom(\sigma_j)\), we have \(\sigma_i \notin \Delta\) and \(\sigma_j \notin \Delta\). Then applying the rule (ALG-INST) twice, we have \(\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_j : s(\sigma_i \circ \sigma_j)\), that is \(\Delta \vdash e[\sigma_{i}] t_i \vdash e[\sigma_{i}] t_j : s(\sigma_i \circ \sigma_j)\).

\(\blacksquare\)
Henceforth, we assume that there are no successive instantiations in a given derivation tree. Consider a typing derivation ending with (ALG-PAIR) where both of its sub-derivations end with (ALG-INST)\[^8\].

\[
\begin{align*}
\Delta \vdash_\Lambda e_1 : t_1 & \quad \forall j \in J_1, \sigma_j \notin \Delta \\
\Delta \vdash_\Lambda e_2 : t_2 & \quad \forall j \in J_2, \sigma_j \notin \Delta \\
\Delta \vdash_\Lambda (e_1, e_2) : \{ \forall j \in J_1, t_1 \sigma_j \} \times \{ \forall j \in J_2, t_2 \sigma_j \}
\end{align*}
\]

We rewrite such a derivation as follows:

\[
\begin{align*}
\Delta \vdash_\Lambda e_1 : t_1 & \quad \forall j \in J_1, \sigma_j \notin \Delta \\
\Delta \vdash_\Lambda e_2 : t_2 & \quad \forall j \in J_2, \sigma_j \notin \Delta \\
\Delta \vdash_\Lambda (e_1, e_2) : t_1 \times t_2 & \quad \forall j \in J_1 \cup J_2, \sigma_j \notin \Delta \\
\end{align*}
\]

Clearly, \( \bigwedge_{j \in J_1 \cup J_2} (t_1 \times t_2) \sigma_j \leq \left( \bigwedge_{j \in J_1} t_1 \sigma_j \right) \times \left( \bigwedge_{j \in J_2} t_2 \sigma_j \right) \). Then we can deduce that \( (e_1, e_2) \sigma_j \) also has the type \( \left( \bigwedge_{j \in J_1} t_1 \sigma_j \right) \times \left( \bigwedge_{j \in J_2} t_2 \sigma_j \right) \) by subsumption. Therefore, we can disregard the sets type substitutions that are applied inside a pair since inferring them outside is equivalent. Hence, we can use the following inference rule for pairs.

\[
\begin{align*}
\Delta \vdash_\Lambda a_1 : t_1 & \quad \Delta \vdash_\Lambda a_2 : t_2 \\
\Delta \vdash_\Lambda (a_1, a_2) : t_1 \times t_2
\end{align*}
\]

Next, consider a derivation ending of (ALG-PROJ) whose premise is derived by (ALG-INST):

\[
\begin{align*}
\Delta \vdash_\Lambda e : t & \quad \forall j \in J, \sigma_j \notin \Delta \\
\Delta \vdash_\Lambda e \sigma_j : \{ \forall j \in J, t \sigma_j \} & \quad (\bigwedge_{j \in J} t \sigma_j) \leq i \times i \\
\Delta \vdash_\Lambda \pi_i (e \sigma_j) : \pi_i (\bigwedge_{j \in J} t \sigma_j)
\end{align*}
\]

According to Lemma C.8 we have \( \pi_i (\bigwedge_{j \in J} t \sigma_j) \leq \bigwedge_{j \in J} \pi_i (t) \sigma_j \), but the converse does not necessarily hold. For example, \( \pi_i (\{ t_1 \times t_2 \} \cdot \{ s_1 \times s_2 \}) = t_1 \{ s_2 / s_1 \} \) while \( \pi_i (\{ t_2 \times t_1 \} \cdot \{ s_2 / s_1 \}) = t_2 \{ s_1 / s_2 \} \). So we cannot exchange the instantiation and projection rules without losing completeness. However, as \( (\bigwedge_{j \in J} t \sigma_j) \leq i \times i \) and \( \forall j \in J, \sigma_j \notin \Delta \), we have \( t \leq i \times i \). This indicates that for an implicitly-typed expression \( \pi_i (a) \), if the inferred type for \( a \) is \( t \) and there exists \( \sigma_j \) such that \( \sigma_j \) \( \vdash_\Lambda t \sigma_j \) \( \leq i \times i \), then we infer the type \( \pi_i (\bigwedge_{j \in J} t \sigma_j) \) for \( \pi_i (a) \). Let \( \Pi_{\Delta} (t) \) denote the set of such result types, that is,

\[
\Pi_{\Delta} (t) = \{ u \mid \sigma_j \vdash_\Lambda t \leq i \times i, u = \pi_i (\bigwedge_{j \in J} t \sigma_j) \}
\]

This set is, in the sense expressed by Lemma D.7, closed under intersection. Formally, we have the following inference rule for projections

\[
\Delta \vdash_\Lambda a : t \quad u \in \Pi_{\Delta} (t) \\
\Delta \vdash_\Lambda \pi_i (a) : u
\]

The following lemma tells us that \( \Pi_{\Delta} (t) \) is “morally” closed by intersection, in the sense that if we take two solutions in \( \Pi_{\Delta} (t) \), then we can take also their intersection as a solution, since there always exists in \( \Pi_{\Delta} (t) \) a solution at least as precise as their intersection.

**Lemma D.7.** Let \( t \) be a type and \( \Delta \) a set of type variables. If \( u_1 \in \Pi_{\Delta} (t) \) and \( u_2 \in \Pi_{\Delta} (t) \), then \( \exists u_0 \in \Pi_{\Delta} (t) \) such that \( u_0 \leq u_1 \land u_2 \).

*Proof.* Let \( \sigma_j \vdash_\Lambda t \leq i \times i \) and \( u_k = \pi_i (\bigwedge_{j \in J_k} t \sigma_j) \) for \( k = 1, 2 \). Then \( \sigma_j \vdash_\Lambda t \leq i \times i \). So \( \pi_i (\bigwedge_{j \in J_1 \cup J_2} t \sigma_j) \leq \pi_i (\bigwedge_{j \in J_1} t \sigma_j) \land \pi_i (\bigwedge_{j \in J_2} t \sigma_j) \). Moreover, by Lemma C.6 we have

\[
\pi_i (\bigwedge_{j \in J_1 \cup J_2} t \sigma_j) = u_1 \land u_2
\]

Since we only consider \( \lambda \)-abstractions with empty decorations, we can consider the following simplified version of (ALG-ABSTR) that does not use relabeling

\[
\begin{align*}
\forall i \in I, \Delta \cup \text{var}(\bigwedge_{t_i \to s_i} x) & \rightarrow \Pi \vdash_\Lambda e : s_i' \text{ and } s_i' \leq s_i \\
& \Delta \vdash_\Lambda A^\Lambda (\bigwedge_{t_i \to s_i} x) e : \bigwedge_{t_i \to s_i} x, s_i \in \text{var}
\end{align*}
\]

\(^8\) If one of the sub-derivations does not end with (ALG-INST), we can apply a trivial instance of (ALG-INST) with an identity substitution \( \sigma_{ed} \).
Suppose the last rule used in the sub-derivations is (ALG-INST).

\[
\forall i \in I. \begin{cases}
\Delta \vdash \Gamma, x : t_i \vdash A : s_i' & \forall j \in J, \sigma_j \vdash \Delta' \\
\Delta \vdash \Gamma, x : t_i \vdash A[\sigma_j] : \bigwedge_{j \in J} s_i'[\sigma_j] \\
\bigwedge_{j \in J} s_i'[\sigma_j] \leq s_i
\end{cases}
\]

\[
\Delta' = \Delta \cup \text{var}(\bigwedge_{i \in I} (t_i \rightarrow s_i))
\]

From the side conditions, we deduce that $s_i' \subseteq \Delta' \leq s_i$ for all $i \in I$. Instantiation may be necessary to bridge the gap between the (domain of the) function type and its argument.

\[
\text{Instantiation may be needed to bridge the gap between the computed type } s_i' \text{ for } e \text{ and the type } s_i \text{ required by the interface, so inferring type substitutions at this stage is mandatory. Therefore, we propose the following inference rule for abstracts.}
\]

\[
\forall i \in I. \begin{cases}
\Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \vdash \Gamma, (x : t_i) \vdash A : s_i' \\
s_i' \subseteq \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \leq s_i
\end{cases}
\]

In the application case, suppose both sub-derivations end with (ALG-INST):

\[
\Delta \vdash A : e_1 : t \quad \forall j_1 \in J_1, \sigma_{j_1} \vdash \Delta \\
\Delta \vdash A : e_2 : s \quad \forall j_2 \in J_2, \sigma_{j_2} \vdash \Delta \\
\bigwedge_{j_1 \in J_1} \sigma_{j_1} \leq u \quad \forall j_2 \in J_2, \sigma_{j_2} \vdash \Delta \\
\text{Instantiation may be needed to bridge the gap between the (domain of the) function type and its argument (e.g., to apply } \lambda x. x \text{ to } 42). \text{ The side conditions imply that } \sigma_{j_1}[j_1 \in J_1] \vdash t \subseteq \Delta \quad 0 \rightarrow \uparrow \text{ and } \sigma_{j_2}[j_2 \in J_2] \vdash s \subseteq \Delta \text{ dom}(\bigwedge_{j_1 \in J_1} \sigma_{j_1}). \text{ Therefore, given an implicitly-typed application } a_1 a_2 \text{ where } a_1 \text{ and } a_2 \text{ are typed with } t \text{ and } s \text{ respectively, we have to find two sets of substitutions } [\sigma_{j_1}[j_1 \in J_1] \text{ and } [\sigma_{j_2}[j_2 \in J_2] \text{ verifying the above preorder relations to be able to type the application. If such sets of substitutions exist, then we can type the application with } (\bigwedge_{j_1 \in J_1} \sigma_{j_1}) \cdot (\bigwedge_{j_2 \in J_2} \sigma_{j_2}). \text{ Let } t \cdot s \text{ denote the set of such result types, that is,}
\]

\[
t \cdot s = \bigcup \left\{ u \mid \begin{array}{l}
[\sigma_{j_1}[j_1 \in J_1] \vdash t \subseteq \Delta \rightarrow \uparrow \\
[\sigma_{j_2}[j_2 \in J_2] \vdash s \subseteq \Delta \text{ dom}(\bigwedge_{j_1 \in J_1} \sigma_{j_1})
\end{array} \right. \\
\text{This set is closed under intersection (see Lemma D.8). Formally, we get the following inference rule for applications}
\]

\[
\Delta \vdash A_1 : t \quad \Delta \vdash A_2 : s \\
\Delta \vdash A_1 A_2 : u \\
\Delta \vdash A_1 A_2 : u
\]

**Lemma D.8.** Let $t$, $s$ be two types and $\Delta$ a set of type variables. If $u_1 \in t \cdot s$ and $u_2 \in t \cdot s$, then $\exists u_0 \in t \cdot s. u_0 \leq u_1 \wedge u_2$.

**Proof.** Let $u_k = \bigwedge_{i \in I} t_{\sigma_{i_k}} \cdot (\bigwedge_{j \in J} s_{\sigma_{j_k}})$ for $k = 1, 2$. According to Lemma C.18 we have

\[
(\bigwedge_{i \in I_1 \cup I_2} t_{\sigma_{i}}) \cdot (\bigwedge_{j \in J_1 \cup J_2} s_{\sigma_{j}}) \leq \bigwedge_{k=1,2} (\bigwedge_{i \in I_k} t_{\sigma_{i_k}}) \cdot (\bigwedge_{j \in J_k} s_{\sigma_{j_k}}) = u_1 \wedge u_2.
\]

For type cases, we distinguish the four possible behaviours: (i) no branch is selected, (ii) the first branch is selected, (iii) the second branch is selected, and (iv) both branches are selected. In all cases, we assume that the premises end with (ALG-INST). In case (i), we have the following derivation:

\[
\Delta \vdash A : t' \\
\Delta \vdash [e_{j}] : [\sigma_{j}] [j \in J] \\
\text{Clearly, the side conditions implies } t' \subseteq \Delta. \text{ The type inference rule for implicitly-typed expressions corresponding to this case is then}
\]

\[
\Delta \vdash a : t' \quad t' \subseteq \Delta \\
\Delta \vdash a : t' \\
\]
For case (ii), consider the following derivation:

\[
\frac{\Delta \Gamma \vdash A \ e : t' \quad \sigma_j \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_j] : \bigwedge_{j \in J} t' \sigma_j} \quad \frac{\Delta \Gamma \vdash A \ e[\sigma_j] : \bigwedge_{j \in J} t' \sigma_j \leq t}{\Delta \Gamma \vdash A \ e[\sigma_j] \{ e_1 \} : \bigwedge_{j \in J_1} \beta_1 \sigma_j} \quad \frac{\Delta \Gamma \vdash A \ e_1 : s_1 \quad \sigma_j \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e_2 : s_2}
\]

First, such a derivation can be rewritten as

\[
\frac{\Delta \Gamma \vdash A \ e : t' \quad \sigma_j \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_j] : \bigwedge_{j \in J} t' \sigma_j} \quad \frac{\Delta \Gamma \vdash A \ e[\sigma_j] \{ e_1 \} : \bigwedge_{j \in J_1} \beta_1 \sigma_j}{\Delta \Gamma \vdash A \ e_1 : s_1 \quad \sigma_j \not\subseteq \Delta}
\]

This indicates that it is equivalent to apply the substitutions \([\sigma_{j_1}]_{j_1 \in J_1}\) to \(e_1\) or to the whole type case expression. Looking at the derivation for \(e\), for the first branch to be selected we must have \(t' \not\subseteq \Delta t\). Note that if \(t' \not\subseteq \Delta t\), we would have \(t' \not\subseteq \Delta 0\) by Lemma [D.4] and no branch would be selected. Consequently, the type inference rule for a type case where the first branch is selected is as follows.

\[
\frac{\Delta \Gamma \vdash X \ a : t' \quad t' \not\subseteq \Delta t \quad t' \not\subseteq \Delta \not\equiv t}{\Delta \Gamma \vdash X \ a_1 : s}
\]

Case (iii) is similar to case (ii) where \(t\) is replaced by \(\neg t\).

At last, consider a derivation of Case (iv):

\[
\frac{\Delta \Gamma \vdash A \ e : t' \quad \forall j \in J, \sigma_j \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_j] : \bigwedge_{j \in J} t' \sigma_j}
\]

\[
\frac{\bigwedge_{j \in J} t' \sigma_j \not\subseteq \not t \quad \Delta \Gamma \vdash A \ e_1 : s_1 \quad \forall j_1 \in J_1, \sigma_{j_1} \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_{j_1}] \{ e_1 \} \{ e_2 \} : \bigwedge_{j_1 \in J_1} \beta_1 \sigma_{j_1} \quad \bigwedge_{j_2 \in J_2} s_2 \sigma_{j_2}}
\]

Using \(\alpha\)-conversion if necessary, we can assume that the polymorphic type variables of \(e_1\) and \(e_2\) are distinct, and therefore we have \((\text{var}(s_1) \setminus \Delta) \cap (\text{var}(s_2) \setminus \Delta) = \emptyset\). According to Lemma [D.5], we get \(s_1 \lor s_2 \not\subseteq \Delta \bigwedge_{j_1 \in J_1} s_1 \sigma_{j_1} \lor \bigwedge_{j_2 \in J_2} s_2 \sigma_{j_2}\). Let \(\sigma_{j_1} \lor \sigma_{j_2} \not\subseteq \Delta \bigwedge_{j_1 \in J_1} s_1 \sigma_{j_1} \lor \bigwedge_{j_2 \in J_2} s_2 \sigma_{j_2}\).

We can rewrite this derivation as

\[
\frac{\Delta \Gamma \vdash A \ e : t' \quad \forall j \in J, \sigma_j \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_j] : \bigwedge_{j \in J} t' \sigma_j}
\]

\[
\frac{\bigwedge_{j \in J} t' \sigma_j \not\subseteq \not t \quad \Delta \Gamma \vdash A \ e_1 : s_1 \quad \forall j_1 \in J_1, \sigma_{j_1} \not\subseteq \Delta}{\Delta \Gamma \vdash A \ e[\sigma_{j_1}] \{ e_1 \} \{ e_2 \} : \bigwedge_{j_1 \in J_1} \beta_1 \sigma_{j_1} \quad \bigwedge_{j_2 \in J_2} s_2 \sigma_{j_2}}
\]

As \(\bigwedge_{j_1 \in J_1} s_1 \sigma_{j_1} \lor \bigwedge_{j_2 \in J_2} s_2 \sigma_{j_2}\), by subsumption, we can deduce that \(\{e[\sigma_j] \mid j \in J\} \not\subseteq \Delta t\) and \(e[\sigma_j] \not\subseteq \Delta t\), but it does not mean that we have \(t' \not\subseteq \Delta t\) and \(t' \not\subseteq \Delta \not\equiv t\) in general. If \(t' \not\subseteq \Delta t\) and/or \(t' \not\subseteq \Delta \not\equiv t\) hold, then we are in one of the previous cases (i) – (iii) (i.e., we type-check at most one branch), and the inferred result type for the whole type case belongs to \(s_1\) or \(s_2\). We can then use subsumption to the whole type-case expression with \(s_1 \lor s_2\). Otherwise, both branches are type-checked, and we deduce the corresponding inference rule as follows.

\[
\Delta \Gamma \vdash X \ a : t' \quad \{ t' \not\subseteq \Delta t\ \text{and} \ \Delta \Gamma \vdash A \ e_1 : s_1 \} \quad \{ t' \not\subseteq \Delta \not\equiv t\ \text{and} \ \Delta \Gamma \vdash A \ e_2 : s_2 \}
\]
From this study, we deduce the type-checking rules for implicitly-typed expressions given in Figure 9. We now prove that these rules are sound and complete w.r.t. the type system of the explicitly-typed calculus.

**Definition D.9.** Let $e, e' \in \mathcal{E}_0$ be two expressions. Their intersection $e \cap e'$ is defined by induction as:

- $c \cap c = c$
- $x \cap x = x$
- $(e_1, e_2) \cap (e'_1, e'_2) = ((e_1 \cap e'_1), (e_2 \cap e'_2))$
- $\pi_i(e) \cap \pi_i(e') = \pi_i(e \cap e')$
- $e_1 \cap e_2 = (e_1 \cap e_2)$
- $(\lambda^{\alpha \in t_i \rightarrow s_i} \cdot e) \cap (\lambda^{\alpha \in t_i \rightarrow s_i} \cdot e') = \lambda^{\alpha \in t_i \rightarrow s_i} \cdot (e \cap e')$
- $(e_0 \in t \cap e_2) \cap (e'_0 \in t \cap e'_2) = (e_0 \cap e'_0 \in t \cap e_2 \cap e'_2)$
- $(e[s_{\sigma}]_j) \cap (e'[s_{\sigma}]_j) = (e \cap e')[s_{\sigma}]_j$
- $e \cap (e'[s_{\sigma}]_j) = (e \cap e')[s_{\sigma}]_j$
- $(e[s_{\sigma}]_j) \cap e' = (e[s_{\sigma}]_j \cap e')$

(where $\sigma_{\alpha}$ is the identity type substitution) and is undefined otherwise.

### Type-Substitution Inference Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premises</th>
<th>Conclusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF-CONST</td>
<td>$\Delta \vdash c : b_c$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td>INF-VAR</td>
<td>$\Delta \vdash a : t_1$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td>INF-PAIR</td>
<td>$\Delta \vdash a : t_2$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td>INF-APPL</td>
<td>$\Delta \vdash a : t_1 \times t_2$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td>INF-ABST</td>
<td>$\Delta \vdash a : t'$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
<tr>
<td>INF-CONTEST</td>
<td>$\Delta \vdash a : t$</td>
<td>$\Delta \vdash x : \Gamma(x)$</td>
</tr>
</tbody>
</table>

**Figure 9.** Type-Substitution Inference Rules
Lemma D.10. Let $e, e' \in \mathcal{E}_0$ be two expressions. If $\text{erase}(e) = \text{erase}(e')$, then $e \sqcap e'$ exists and $\text{erase}(e \sqcap e') = \text{erase}(e) = \text{erase}(e')$.

Proof. By induction on the structures of $e$ and $e'$. Because $\text{erase}(e) = \text{erase}(e')$, the two expressions have the same structure up to their sets of type substitutions.

$\pi_i : \pi_i(e) \sqcap \pi_i(e')$ straightforward.

$\pi_i(e_1, e_2) \sqcap (e_1', e_2')$: we have $\text{erase}(e_i) = \text{erase}(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $\text{erase}(e_i \sqcap e_i') = \text{erase}(e_i')$. Therefore $(e_1, e_2) \sqcap (e_1', e_2')$ exists and

$$
\text{erase}((e_1, e_2) \sqcap (e_1', e_2')) = \text{erase}((e_1 \sqcap e_1') \sqcap (e_2 \sqcap e_2')) = (\text{erase}(e_1 \sqcap e_1'), \text{erase}(e_2 \sqcap e_2')) = (\text{erase}(e_1), \text{erase}(e_2)) = \text{erase}((e_1, e_2)).
$$

Similarly, we also have $\text{erase}((e_1, e_2) \sqcap (e_1', e_2')) = \text{erase}((e_1', e_2'))$.

$\pi_i(e), \pi_i(e')$: we have $\text{erase}(e) = \text{erase}(e')$. By induction, $e \sqcap e'$ exists and $\text{erase}(e \sqcap e') = \text{erase}(e')$. Therefore $\pi_i(e) \sqcap \pi_i(e')$ exists and

$$
\text{erase}(\pi_i(e) \sqcap \pi_i(e')) = \text{erase}(\pi_i(e)) = \text{erase}(\pi_i(e')) = \text{erase}((e_1, e_2)) = \text{erase}((e_1', e_2')).
$$

Similarly, we also have $\text{erase}(\pi_i(e) \sqcap \pi_i(e')) = \text{erase}(\pi_i(e'))$.

$\pi_i(e_1, e_2) \sqcap (e_1', e_2')$: we have $\text{erase}(e_i) = \text{erase}(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $\text{erase}(e_i \sqcap e_i') = \text{erase}(e_i) = \text{erase}(e_i')$. Therefore $(e_1, e_2) \sqcap (e_1', e_2')$ exists and

$$
\text{erase}((e_1, e_2) \sqcap (e_1', e_2')) = \text{erase}((e_1 \sqcap e_1') \sqcap (e_2 \sqcap e_2')) = (\text{erase}(e_1 \sqcap e_1'), \text{erase}(e_2 \sqcap e_2')) = (\text{erase}(e_1), \text{erase}(e_2)) = \text{erase}((e_1, e_2)).
$$

Similarly, we also have $\text{erase}((e_1, e_2) \sqcap (e_1', e_2')) = \text{erase}((e_1', e_2'))$.

$\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot t_i, \lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e':$ we have $\text{erase}(e) = \text{erase}(e')$. By induction, $e \sqcap e'$ exists and $\text{erase}(e \sqcap e') = \text{erase}(e)$.

Therefore $(\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e) \sqcap (\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e') \sqcap (\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e) = \text{erase}(e')$. Therefore $\text{erase}((\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e) \sqcap (\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e')) = \text{erase}(\lambda^i_{t_1 \rightarrow s_1} \cdot x \cdot e')$.

$e_0 \in t \land e_1 : e_2, e_0' \in t \land e_1' : e_2'$: we have $\text{erase}(e_i) = \text{erase}(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $\text{erase}(e_i \sqcap e_i') = \text{erase}(e_i) = \text{erase}(e_i')$. Therefore $(e_0 \in t \land e_1 : e_2) \sqcap (e_0' \in t \land e_1' : e_2')$ exists and

$$
\text{erase}((e_0 \in t \land e_1 : e_2) \sqcap (e_0' \in t \land e_1' : e_2')) = \text{erase}((e_0 \sqcap e_0') \in t \land (e_1 \sqcap e_1') : (e_2 \sqcap e_2')) = \text{erase}(e_0 \sqcap e_0') \sqcap (e_1 \sqcap e_1') \sqcap (e_2 \sqcap e_2') = \text{erase}(e_0) \sqcap e_1 \sqcap e_2 = \text{erase}(e_0 \sqcap e_1 \sqcap e_2).
$$

Similarly, we also have

$\text{erase}((e_0 \in t \land e_1 : e_2) \sqcap (e_0' \in t \land e_1' : e_2')) = \text{erase}(e_0' \in t \land e_1' : e_2')$.

$e_1 \in \sigma_j \sqcap e_2 \in \sigma_j$: we have $\text{erase}(e) = \text{erase}(e')$. By induction, $e \sqcap e'$ exists and $\text{erase}(e \sqcap e') = \text{erase}(e')$. Therefore $(e_1 \in \sigma_j \sqcap e_2 \in \sigma_j) \sqcap (e_1' \in \sigma_j \sqcap e_2') \sqcap (e_1 \in \sigma_j \sqcap e_2) \sqcap (e_1' \in \sigma_j \sqcap e_2')$ exists and

$$
\text{erase}((e_1 \in \sigma_j \sqcap e_2 \in \sigma_j) \sqcap (e_1' \in \sigma_j \sqcap e_2')) = \text{erase}((e_1 \sqcap e_1') \in \sigma_j \sqcap (e_2 \sqcap e_2')) = \text{erase}(e_1 \sqcap e_1') \sqcap (e_2 \sqcap e_2') = \text{erase}(e_1) \sqcap e_2 \sqcap e_2' \sqcap e_1 = \text{erase}(e_1),
$$

Similarly, we also have

$\text{erase}((e_1 \in \sigma_j \sqcap e_2 \in \sigma_j) \sqcap (e_1' \in \sigma_j \sqcap e_2')) = \text{erase}(e_2 \sqcap e_2')$.
Similarly, we also have \( erase(e|\sigma|) \cap (e'|\sigma|) = erase(e'|\sigma|) \).
\( e, e'|\sigma| \) : a special case of \( e|\sigma| \) and \( e'|\sigma| \) where \( |\sigma| = |\sigma| \).
\( e|\sigma|, e' : \) a special case of \( e|\sigma| \) and \( e'|\sigma| \) where \( |\sigma| = |\sigma| \).

\[ \therefore \]

**Lemma D.11.** Let \( e, e' \in \mathcal{E} \) be two expressions. If \( erase(e) = erase(e'), \Delta \Gamma \vdash e : t', e \not\Delta' \) and \( e \not\Delta' \), then \( \Delta \Gamma \vdash e \cap e' : t \) and \( \Delta' \Gamma \vdash e \cap e' : t' \).

**Proof.** According to Lemma D.10 \( e \cap e' \) exists and \( erase(e \cap e') = erase(e) \) or \( erase(e') \). We only prove \( \Delta \Gamma \vdash e \cap e' : t \) as the other case is similar. For simplicity, we just consider one set of type substitutions. For several sets of type substitutions, we can either compose them or apply (instructor) several times. The proof proceeds by induction on \( \Delta \Gamma \vdash e : t \).

(\( const \)): \( \Delta \Gamma \vdash e : b_c \). As \( erase(e') = c \), \( e' \) is either \( e \) or \( c|\sigma| \). If \( e' = c \), then \( e \cap e' = c \), and the result follows straightforwardly. Otherwise, we have \( e \cap e' = c|\sigma|, e|\sigma| \). Since \( e' \not\Delta \Delta, \) we have \( \sigma \not\Delta \Delta \). By (instructor), we have \( \Delta \Gamma \vdash c|\sigma|, e|\sigma| : b_c \). Then \( \Delta \Gamma \vdash e \cap e' : t \) follows straightforwardly. Otherwise, we have \( e \cap e' = c|\sigma|, e|\sigma| \). Since \( e' \not\Delta \Delta, \) we have \( \sigma \not\Delta \Delta \). By (instructor), we have \( \Delta \Gamma \vdash e \cap e' : t \).

(\( var \)): \( \Gamma \vdash x : \Gamma(x) \). As \( erase(e') = x \), \( e' \) is either \( x \) or \( \Gamma(x) \). If \( e' = x \), then \( e \cap e' = x \), and the result follows straightforwardly. Otherwise, we have \( e \cap e' = x|\sigma|, e|\sigma| \). Since \( e' \not\Delta \Delta, \) we have \( \sigma \not\Delta \Delta \). By (instructor), we have \( \Delta \Gamma \vdash e \cap e' : t \).

(\( pair \)): consider the following derivation:

\[
\begin{array}{ll}
\Delta \Gamma \vdash e_1 : t_1 & \Delta \Gamma \vdash e_2 : t_2 \\
\Delta \Gamma \vdash e_1 : t_1 \times t_2 & (\text{pair})
\end{array}
\]

As \( erase(e') = \pi_1(erase(e)) \), \( e' \) is either \( \pi_1(e_0) \) or \( \pi_1(e_0)\sigma_{\sigma j} \) such that \( erase(e_0) = erase(e_0) \). By induction, we have \( \Delta \Gamma \vdash e_0 : t_1 \times t_2 \). Then by (pair), we have \( \Delta \Gamma \vdash (e_1 \cap e_1': e_2 \cap e_2') : t_1 \times t_2 \). So the result follows.

(\( proj \)): consider the following derivation:

\[
\begin{array}{ll}
\Delta \Gamma \vdash e_1 : t \rightarrow s & \Delta \Gamma \vdash e_2 : t \\
\Delta \Gamma \vdash e_1' : s & (\text{pair})
\end{array}
\]

As \( erase(e') = erase(e) \), \( e' \) is either \( e_1' \) or \( e_2' \), such that \( erase(e') = erase(e) \). By induction, we have \( \Delta \Gamma \vdash e_0 : t_1 \times t_2 \). Then by (pair), we have \( \Delta \Gamma \vdash (e_1 \cap e_1': e_2 \cap e_2') : t_1 \times t_2 \). So the result follows.

(\( app \)): consider the following derivation:

\[
\begin{array}{ll}
\Delta \Gamma \vdash e_1 : t \rightarrow s & \Delta \Gamma \vdash e_2 : t \\
\Delta \Gamma \vdash e_1 : e_2 : s & (\text{pair})
\end{array}
\]

As \( erase(e') = erase(e) \), \( e' \) is either \( e_1' \) or \( e_2' \), such that \( erase(e') = erase(e) \). By induction, we have \( \Delta \Gamma \vdash e_0 : t_1 \times t_2 \). Then by (pair), we have \( \Delta \Gamma \vdash e_1 \cap e_1' : t \rightarrow s \) and \( \Delta \Gamma \vdash e_2 \cap e_2' : t \). So the result follows.

(\( abstr \)): consider the following derivation:

\[
\begin{array}{ll}
\forall i \in I. \Delta' \Gamma_i (x : t_i) \vdash e_0 : s_i \\
\Delta'' = \Delta' \cup \text{var}(\bigcup_{i \in I} t_i \rightarrow s_i) & (\text{abstr})
\end{array}
\]

As \( erase(e') = \lambda^{t_1 \rightarrow s_1} e_0 \), \( e' \) is either \( \lambda^{t_1 \rightarrow s_1} e_0 \), \( \lambda^{t_1 \rightarrow s_1} e_0 \) such that \( erase(e_0) = erase(e_0) \). As \( \Lambda^{t_1 \rightarrow s_1} e_0 \) is well-typed under \( \Delta' \) and \( \Gamma' \), \( e_0 \not\Delta' \cup \text{var}(\bigcup_{i \in I} t_i \rightarrow s_i) \). By induction, we have \( \Lambda \Gamma (x : t_i) \vdash e_0 \cap e_0 : s_i \). Then by (abstr), we have \( \Delta \Gamma \vdash \lambda^{t_1 \rightarrow s_1} e_0 : \bigcup_{i \in I} t_i \rightarrow s_i \). So the result follows.

Otherwise, \( e \cap e' = \lambda^{t_1 \rightarrow s_1} e_0 \cap e_0 : \bigcup_{i \in I} t_i \rightarrow s_i \). Since \( e' \not\Delta \Delta, \) we have \( \sigma \not\Delta \Delta \). By (instructor),
we have \( \Delta \Gamma \vdash (\lambda^{i:t_i \to x:s_i}; x_0 \in e_0)[\sigma_{id}, \sigma_j]_{j \in J} : (\bigwedge_{i \in I} t_i \to s_i) \wedge \bigwedge_{j \in J} (\lambda_{i \in I} t_i \to s_i) \sigma_j \).

Finally, by (subsum), we get \( \Delta \Gamma \vdash (\lambda^{i:t_i \to x:s_i}; x_0 \in e_0)[\sigma_{id}, \sigma_j]_{j \in J} : \bigwedge_{i \in I} t_i \to s_i \).

**Proof**. Consider the following derivation:

\[
\begin{array}{c}
\Delta \Gamma \vdash e_0 : t' \\
\Delta \Gamma \vdash e_0 : t''
\end{array}
\]

If \( t' \not\simeq t'' \), then by (A), we have \( \Delta \Gamma \vdash e_0 : t' \) and \( \Delta \Gamma \vdash e_0 : t'' \).

As \( \text{erase}(e') = \text{erase}(e_0) \) and \( e' \) is either \( e_0 \) or \( e' \), we have \( \Delta \Gamma \vdash e_0 : t' \) or \( \Delta \Gamma \vdash e_0 : t'' \).

Finally, by (subsum), we get \( \Delta \Gamma \vdash e_0 : t' \) or \( \Delta \Gamma \vdash e_0 : t'' \). So the result follows.

**Corollary D.12**. Let \( e, e' \in \mathcal{E} \) be two expressions. If \( \text{erase}(e) = \text{erase}(e') \), \( \Delta \Gamma \vdash e : t \) and \( \Delta \prime \Gamma' \vdash e' : t' \), then \( \Delta \Gamma \vdash e \simeq e' \).

1. there exists \( s \) such that \( \Delta \Gamma \vdash e \in e' : s \) and \( s \leq t \).
2. there exists \( s' \) such that \( \Delta \Gamma \vdash e \in e' : s' \) and \( s' \leq t' \).

**Proof**. Immediate consequence of Lemma D.11 and Theorems C.22 and C.23.

These type-substitution inference rules are sound and complete with respect to the typing algorithm in Section C.2, modulo the restriction that all the decorations in the \( \lambda \)-abstractions are empty.

**Theorem D.13** (Soundness). If \( \Delta \Gamma \vdash a : t \), then there exists an explicitly-typed expression \( e \in \mathcal{E} \) such that \( \text{erase}(e) = a \) and \( \Delta \Gamma \vdash e : t \).

**Proof**. By induction on the derivation of \( \Delta \Gamma \vdash a : t \). We proceed by a case analysis of the last rule used in the derivation.

1. **INF-CONST**: straightforward (take \( e \) as \( c \)).
2. **INF-VAR**: straightforward (take \( e \) as \( x \)).
3. **INF-PAIR**: consider the derivation:

\[
\begin{array}{c}
\Delta \Gamma \vdash a_1 : t_1 \\
\Delta \Gamma \vdash a_2 : t_2
\end{array}
\]

Applying the induction hypothesis, there exists an expression \( e_i \) such that \( \text{erase}(e_i) = a_i \) and \( \Delta \Gamma \vdash e_i : t_i \). Then by (ALG-PAIR), we have \( \Delta \Gamma \vdash (e_1, e_2) : t_1 \times t_2 \). Moreover, according to Definition D.2, we have \( \text{erase}(e_1, e_2) = (\text{erase}(e_1), \text{erase}(e_2)) = (a_1, a_2) \).
(INF-PROJ): consider the derivation
\[
\begin{align*}
\Delta \Gamma \vdash a : t & \quad u \in \text{H}_\Delta(t) \\
\Delta \Gamma \vdash \pi(a) : u
\end{align*}
\]
By induction, there exists an expression \(e\) such that \(\text{erase}(e) = a\) and \(\Delta \Gamma \vdash e : t\). Let \(u = \pi_i(\bigwedge_{i \in I} t_i)\). As \(t_i \notin \Delta\), by (ALG-INST), we have \(\Delta \Gamma \vdash e[\sigma_i] : \bigwedge_{i \in I} t_i\). Moreover, since \(\bigwedge_{i \in I} t_i \leq t \times t\), by (ALG-PROJ), we get \(\Delta \Gamma \vdash e[\sigma_i] : \pi_i(\bigwedge_{i \in I} t_i)\). Finally, according to Definition D.2, we have
\[
\text{erase}(\pi_i(e[\sigma_i]_{i \in I})) = \pi_i(\text{erase}(e[\sigma_i]_{i \in I})) = \pi_i(\text{erase}(e)) = \pi_i(a)
\]
(INF-APPL): consider the derivation
\[
\begin{align*}
\Delta \Gamma \vdash a_1 : t & \quad \Delta \Gamma \vdash a_2 : s \quad u \in t \bullet s \\
\Delta \Gamma \vdash a_1 a_2 : u
\end{align*}
\]
By induction, we have (i) there exists an expression \(e_1\) such that \(\text{erase}(e_1) = a_1\) and \(\Delta \Gamma \vdash e_1 : t\) and (ii) there exists an expression \(e_2\) such that \(\text{erase}(e_2) = a_2\) and \(\Delta \Gamma \vdash e_2 : s\). Let \(u = \bigwedge_{i \in I} t_i \cdot \bigwedge_{j \in J} s_j\). As \(t_i \notin \Delta\) for \(h \in I \cup J\), applying (ALG-INST), we get \(\Delta \Gamma \vdash e_1[\sigma_i]_{i \in I} : \bigwedge_{i \in I} t_i\) and \(\Delta \Gamma \vdash e_2[\sigma_j]_{j \in J} : \bigwedge_{j \in J} s_j\). Then by (ALG-APPL), we have \(\Delta \Gamma \vdash (e_1[\sigma_i]_{i \in I})(e_2[\sigma_j]_{j \in J}) : \bigwedge_{i \in I} t_i \cdot \bigwedge_{j \in J} s_j\). Furthermore, according to Definition D.2, we have
\[
\text{erase}((e_1[\sigma_i]_{i \in I})(e_2[\sigma_j]_{j \in J})) = \text{erase}(e_1)\text{erase}(e_2) = a_1a_2
\]
(INF-ABSTR): consider the derivation
\[
\begin{align*}
\forall i \in I. \left\{ \begin{array}{l}
\Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \llcorner \Gamma, (x : t_i) \vdash a : s_i' \\
 s_i' \subseteq \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \\
\end{array} \right. \\
\Delta \vdash \lambda^{i \in I} t_i \rightarrow s_i.x.a : \bigwedge_{i \in I} t_i \rightarrow s_i
\end{align*}
\]
Let \(\Delta' = \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i)\) and \(\bigwedge_{j \in J} s_j \subseteq \Delta'\). By induction, there exists an expression \(e_i\) such that \(\text{erase}(e_i) = a\) and \(\Delta' \vdash e_i : s_i'\) for all \(i \in I\). Since \(\sigma_j \notin \Delta'\), by (ALG-INST), we have \(\Delta' \vdash (x : t_i) \vdash e_i[\sigma_j]_{j \in J} : \bigwedge_{j \in J} s_j\). Clearly, \(e_i[\sigma_j]_{j \in J} \notin \Delta'\) and \(\text{erase}(e_i[\sigma_j]_{j \in J}) = \text{erase}(e_i) = a\). Then by Lemma D.10, the intersection \(\bigwedge_{i \in I} (e_i[\sigma_j]_{j \in J})\) exists and we have \(\text{erase}(\bigwedge_{i \in I} (e_i[\sigma_j]_{j \in J})) = a\) for any non-empty \(I' \subseteq I\). Let \(e = \bigwedge_{i \in I} (e_i[\sigma_j]_{j \in J})\). According to Corollary D.12, there exists a type \(t_i'\) such that \(\Delta \vdash (x : t_i) \vdash e : t_i'\) and \(t_i' \leq \bigwedge_{j \in J} s_j\) for all \(i \in I\). Moreover, since \(t_i' \leq \bigwedge_{j \in J} s_j \leq s_i\), by (ALG-ABSTR), we have \(\Delta \vdash \lambda x^{i \in I} t_i \rightarrow s_i.x.e : \bigwedge_{i \in I} t_i \rightarrow s_i\). Finally, according to Definition D.2, we have
\[
\text{erase}(\lambda^{i \in I} t_i \rightarrow s_i.x.e) = \lambda^{i \in I} t_i \rightarrow s_i.x.\text{erase}(e) = \lambda^{i \in I} t_i \rightarrow s_i.x.a
\]
(INF-CASE-NONE): consider the derivation
\[
\begin{align*}
\Delta \Gamma \vdash a : t' \quad t' \subseteq 0 \\
\Delta \Gamma \vdash (a \in t ? a_1 : a_2) : 0
\end{align*}
\]
By induction, there exists an expression \(e\) such that \(\text{erase}(e) = a\) and \(\Delta \Gamma \vdash e : t\). Let \(\sigma_i \notin \Delta\). Since \(e_1 \notin \Delta\), by (ALG-INST), we have \(\Delta \Gamma \vdash e_1[\sigma_i] \in I \vdash e_i : t_i\). Let \(e_1\) and \(e_2\) be two expressions such that \(\text{erase}(e_1) = a_1\) and \(\text{erase}(e_2) = a_2\). Then we have
\[
\text{erase}((e[\sigma_i]_{i \in I})_{i \in t ? e_1 : e_2}) = (a \in t ? a_1 : a_2)
\]
Moreover, since \(\bigwedge_{i \in I} t_i \sigma_i \leq 0\), by (ALG-CASE-NONE), we have
\[
\Delta \Gamma \vdash (e[\sigma_i]_{i \in I})_{i \in t ? e_1 : e_2} : 0
\]
(INF-CASE-FST): consider the derivation
\[
\begin{align*}
\Delta \Gamma \vdash a : t' \quad t' \subseteq 0 & \quad t' \not\subseteq 0 \\
\Delta \Gamma \vdash a \in t \vdash a_1 : a_2 : s \\
\Delta \Gamma \vdash (a \in t ? a_1 : a_2) : s
\end{align*}
\]
By induction, there exists \(e, e_1\) such that \(\text{erase}(e) = a, \text{erase}(e_1) = a_1, \Delta \Gamma \vdash e : t',\) and \(\Delta \Gamma \vdash e_1 : s\). Let \(\sigma_i \in I \vdash t' \subseteq 0\). Since \(\sigma_i \notin \Delta\), by (ALG-INST), we get \(\Delta \Gamma \vdash e_1[\sigma_i] \in I \vdash e_i \vdash t_i \sigma_i\). Let \(e_2\) be an expression such that \(\text{erase}(e_2) = a_2\). Then we have
\[
\text{erase}((e[\sigma_i]_{i \in I})_{i \in t ? e_1 : e_2}) = (a \in t ? a_1 : a_2)
\]
Finally, since \(\bigwedge_{i \in I} t_i \sigma_i \leq t\), by (ALG-CASE-FST), we have
\[
\Delta \Gamma \vdash (e[\sigma_i]_{i \in I})_{i \in t ? e_1 : e_2} : s
\]
(INF-CASE-SND): similar to the case of (INF-CASE-FST).

(INF-CASE-BOTH): consider the derivation

\[
\begin{align*}
\Delta \vdash a : t' \\
\text{and} \\
\Delta \Gamma \vdash a_1 : s_1 \\
\Delta \Gamma \vdash a_2 : s_2
\end{align*}
\]

By induction, there exist e, e_i such that \(\text{erase}(e) = a\), \(\text{erase}(e_i) = a_i\), \(\Delta \Gamma \vdash e : t'\), and \(\Delta \Gamma \vdash e_i : s_i\). According to \textbf{Definition 3.2}, we have

\[\text{erase}(e) = \{e \in t? e \in a\}\]

Clearly \(t' \neq 0\). We claim that \(t' \neq t\). Let \(\sigma\) be any identity type substitution. If \(t' \leq t\), then \(t' \subseteq t\), i.e., \(t' \subseteq t\), which is in contradiction with \(t' \neq t\). Similarly, we have \(t' \neq t\).

Therefore, by \textbf{(ALG-CASE-SND)}, we have \(\Delta \Gamma \vdash (e \in t? e : e : s_1) \cup s_2\).

\(\square\)

The proof of the soundness property constructs along the derivation for a some expression e that satisfies the statement of the theorem. We denote by \(\text{erase}^{-1}(e)\) the set of expressions e that satisfy the statement.

\textbf{Theorem D.14} (Completeness). Let \(e \in \mathcal{D}\) be an explicitly-typed expression. If \(\Delta \Gamma \vdash e : t\), then there exists a type \(t'\) such that \(\Delta \Gamma \vdash t'\). Let e : t.

\textbf{Proof}. By induction on the typing derivation of \(\Delta \Gamma \vdash e : t\). We proceed by the cases of the last rule used in the derivation.

\textbf{(ALG-CONST)}: take \(t'\) as \(b_c\).

\textbf{(ALG-VAR)}: take \(t'\) as \(\Gamma(x)\).

\textbf{(ALG-PAIR)}: consider the derivation

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t_1 \\
\Delta \Gamma \vdash e_2 : t_2
\end{align*}
\]

\[
\Delta \Gamma \vdash (e_1, e_2) : t_1 \times t_2
\]

Applying the induction hypothesis twice, we have

\[
\begin{align*}
\exists t'_1, \Delta \Gamma \vdash \text{erase}(e_1) : t'_1 &\text{ and } t'_1 \subseteq t_1 \\
\exists t'_2, \Delta \Gamma \vdash \text{erase}(e_2) : t'_2 \text{ and } t'_2 \subseteq t_2
\end{align*}
\]

Then by \textbf{(INF-PAR)}, we have \(\Delta \Gamma \vdash \text{erase}(e_1) \times \text{erase}(e_2) : t'_1 \times t'_2\), that is, \(\Delta \Gamma \vdash \text{erase}((e_1, e_2)) : t'_1 \times t'_2\). Finally, Applying \textbf{Lemma D.4}, we have \(\exists t'_1 \times t'_2 \subseteq (t_1 \times t_2)\).

\textbf{(ALG-PROJ)}: consider the derivation

\[
\begin{align*}
\Delta \Gamma \vdash e : t \leq a \times b \\
\Delta \Gamma \vdash \pi_i(e) : \pi_i(t)
\end{align*}
\]

By induction, we have

\[
\exists t', \exists (\sigma_k)_{k \in K}. \Delta \Gamma \vdash \text{erase}(e) : t' \text{ and } \exists (\sigma_k)_{k \in K} \vdash t' \subseteq t
\]

It is clear that \(\bigwedge_{k \in K} \sigma_k \leq a \times b\). So \(\pi_i(\bigwedge_{k \in K} \sigma_k) \subseteq \Pi_{\Delta}(t')\). Then by \textbf{(INF-APL)}, we have \(\Delta \Gamma \vdash \pi_i(\text{erase}(e)) : \pi_i(\bigwedge_{k \in K} \sigma_k)\), that is, \(\Delta \Gamma \vdash \text{erase}(\pi_i(e)) : \pi_i(\bigwedge_{k \in K} \sigma_k)\). According to \textbf{Lemma C.5}, \(t \leq (\pi_1(t), \pi_2(t))\). Then \(\bigwedge_{k \in K} \sigma_k \leq (\pi_1(t), \pi_2(t))\). Finally, applying \textbf{Lemma C.5} again, we get \(\pi_i(\bigwedge_{k \in K} \sigma_k) \subseteq \pi_i(t)\) and a fortiori \(\pi_i(\bigwedge_{k \in K} \sigma_k) \subseteq \Delta \pi_i(t)\).

\textbf{(ALG-APL)}: consider the derivation

\[
\begin{align*}
\Delta \Gamma \vdash e_1 : t \\
\Delta \Gamma \vdash e_2 : s
\end{align*}
\]

\[
\Delta \Gamma \vdash e_1 \cdot e_2 : t \cdot s
\]

Applying the induction hypothesis twice, we have

\[
\begin{align*}
\exists t'_1, \exists (\sigma^1_k)_{k \in K_1}. \Delta \Gamma \vdash \text{erase}(e_1) : t'_1 \text{ and } \exists (\sigma^1_k)_{k \in K_1} \vdash t'_1 \subseteq t \\
\exists t'_2, \exists (\sigma^2_k)_{k \in K_2}. \Delta \Gamma \vdash \text{erase}(e_2) : t'_2 \text{ and } \exists (\sigma^2_k)_{k \in K_2} \vdash t'_2 \subseteq s
\end{align*}
\]

It is clear that \(\bigwedge_{k \in K_1} \sigma^1_k \leq a \rightarrow b\), that is, \(\bigwedge_{k \in K_1} \sigma^1_k \text{ is a function type. So we get } \text{dom}(t) \leq \text{dom}(\bigwedge_{k \in K_1} \sigma^1_k)\). Then we have

\[
\bigwedge_{k \in K_2} t'_2 \sigma^2_k \leq s \leq \text{dom}(t) \leq \text{dom}(\bigwedge_{k \in K_1} \sigma^1_k)
\]
Therefore, \( (\bigwedge_{k \in K_1} t'_1 \sigma_k^1 \cdot (\bigwedge_{k \in K_2} t'_2 \sigma_k^2)) \in t'_2 \cdot_\Delta t'_1 \). Then applying (INF-APPL), we have \( \Delta_3 \Gamma \vdash_X \) erase\( (e_1) \) erase\( (e_2) : (\bigwedge_{k \in K_1} t'_1 \sigma_k^1) \cdot (\bigwedge_{k \in K_2} t'_2 \sigma_k^2) \), that is, \( \Delta_3 \Gamma \vdash_X \) erase\( (e_1 : e_2) : (\bigwedge_{k \in K_1} t'_1 \sigma_k^1) \cdot (\bigwedge_{k \in K_2} t'_2 \sigma_k^2) \). Moreover, as \( \bigwedge_{k \in K_2} t'_2 \sigma_k^2 \subset \text{dom}(t) \cdot (\bigwedge_{k \in K_2} t'_2) \) exists. According to Lemma C.14, we have
\[
(\bigwedge_{k \in K_1} t'_1 \sigma_k^1) \cdot (\bigwedge_{k \in K_2} t'_2 \sigma_k^2) \leq t \cdot (\bigwedge_{k \in K_2} t'_2) \leq t \cdot s.
\]
Thus, \( (\bigwedge_{k \in K_1} t'_1 \sigma_k^1) \cdot (\bigwedge_{k \in K_2} t'_2 \sigma_k^2) \subseteq t \cdot s \).

(ALG-ABST): consider the derivation

\[
\forall i \in I. \Delta \cup \text{var}(\bigwedge_{t_i \rightarrow s_i}) \Gamma, (x : t_i) \vdash_A e : s'_i \quad \text{and} \quad s'_i \leq s_i
\]

Let \( \Delta' = \Delta \cup \text{var}(\bigwedge_{t_i \rightarrow s_i}) \). By induction, for each \( i \in I \), we have
\[
\exists t'_i, \Delta_3 \Gamma, (x : t_i) \vdash_X \text{erase}(e) : t'_i \quad \text{and} \quad t'_i \subseteq \Delta', s'_i
\]
Clearly, we have \( t'_i \subseteq \Delta', s_i \). By (INF-ABST), we have
\[
\Delta_3 \Gamma \vdash_X \lambda^{\bigwedge_{t_i \rightarrow s_i}} x. \text{erase}(e) : \bigwedge_{i \in I} t_i \rightarrow s_i
\]
that is, \( \Delta_3 \Gamma \vdash_X \text{erase}(\lambda^{\bigwedge_{t_i \rightarrow s_i}} x. e) : \bigwedge_{i \in I} t_i \rightarrow s_i \).

(ALG-CASE-NONE): consider the derivation

\[
\Delta_3 \Gamma \vdash_A e : 0 \\
\Delta_3 \Gamma \vdash_A (t \in e ? 1 : 2) : 0
\]
By induction, we have
\[
\exists t'_0, \Delta_3 \Gamma \vdash_X \text{erase}(e) : t'_0 \quad \text{and} \quad t'_0 \subseteq \Delta 0
\]
By (INF-CASE-NONE), we have \( \Delta_3 \Gamma \vdash_X (\text{erase}(e) \in t ? \text{erase}(e_1) \cdot \text{erase}(e_2)) : 0 \), that is, \( \Delta_3 \Gamma \vdash_X \text{erase}(e \in t ? e_1 : e_2) : 0 \).

(ALG-CASE-FST): consider the derivation

\[
\Delta_3 \Gamma \vdash_A e : t' \quad t' \leq t \\
\Delta_3 \Gamma \vdash_A e_1 : s_1
\]
Applying the induction hypothesis twice, we have
\[
\exists t'_0, \Delta_3 \Gamma \vdash_X \text{erase}(e) : t'_0 \quad \text{and} \quad t'_0 \subseteq \Delta t' \]
\[
\exists t'_1, \Delta_3 \Gamma \vdash_X \text{erase}(e_1) : t'_1 \quad \text{and} \quad t'_1 \subseteq \Delta s_1
\]
Clearly, we have \( t'_0 \subseteq \Delta t \). If \( t'_0 \subseteq \Delta \not\vdash t \), then by Lemma D.4, we have \( t'_0 \leq \Delta 0 \). By (INF-CASE-NONE), we get
\[
\Delta_3 \Gamma \vdash_X (\text{erase}(e) \in t ? \text{erase}(e_1) : \text{erase}(e_2)) : 0
\]
that is, \( \Delta_3 \Gamma \vdash_X \text{erase}(e \in t ? e_1 : e_2) : 0 \). Clearly, we have \( 0 \subseteq \Delta s_1 \).
Otherwise, by (INF-CASE-FST), we have
\[
\Delta_3 \Gamma \vdash_X (\text{erase}(e) \in t ? \text{erase}(e_1) : \text{erase}(e_2)) : t'_1
\]
that is, \( \Delta_3 \Gamma \vdash_X \text{erase}(e \in t ? e_1 : e_2) : t'_1 \). The result follows as well.

(ALG-CASE-SND): similar to the case of (ALG-CASE-FST).

(ALG-CASE-BOTH): consider the derivation

\[
\Delta_3 \Gamma \vdash_A e : t' \quad \begin{cases} 
\Delta_3 \Gamma \vdash_A e_1 : s_1 \\
\Delta_3 \Gamma \vdash_A e_2 : s_2 
\end{cases}
\]
By induction, we have
\[
\exists t'_0, \Delta_3 \Gamma \vdash_X \text{erase}(e) : t'_0 \quad \text{and} \quad t'_0 \subseteq \Delta t' \\
\Delta_3 \Gamma \vdash_X \text{erase}(e_1) : t'_1 \quad \text{and} \quad t'_1 \subseteq \Delta s_1 \\
\Delta_3 \Gamma \vdash_X \text{erase}(e_2) : t'_2 \quad \text{and} \quad t'_2 \subseteq \Delta s_2
\]
If \( t'_0 \subseteq \Delta 0 \), then by (INF-CASE-NONE), we get
\[
\Delta_3 \Gamma \vdash_X (\text{erase}(e) \in t ? \text{erase}(e_1) : \text{erase}(e_2)) : 0
\]
that is, \( \Delta \vdash_\Pi e \vdash e_1 \? e_2 : 0 \). Clearly, we have \( 0 \subseteq_\Delta s_1 \lor s_2 \).

If \( t_0' \subseteq_\Delta t \), then by (INF-CASE-FST), we get

\[
\Delta \vdash_\Pi (\text{erase}(e) \? e_1 \? \text{erase}(e_2)) : t'_1
\]

that is, \( \Delta \vdash_\Pi e \? e_1 \? e_2 : t'_1 \). Moreover, it is clear that \( t'_1 \subseteq_\Delta s_1 \lor s_2 \), the result follows as well. Similarly for \( t_0' \not\subseteq_\Delta t \).

Otherwise, by (INF-CASE-BOTH), we have

\[
\Delta \vdash_\Pi (\text{erase}(e) \? e_1 \? \text{erase}(e_2)) : t'_1 \lor t'_2
\]

that is, \( \Delta \vdash_\Pi e \? e_1 \? e_2 : t'_1 \lor t'_2 \).

Using \( \alpha \)-conversion, we can assume that the polymorphic type variables of \( t'_1 \) and \( t'_2 \) (and of \( e_1 \) and \( e_2 \)) are distinct, i.e., \( (\text{var}(t'_1) \setminus \Delta) \cap (\text{var}(t'_2) \setminus \Delta) = 0 \).

Then applying Lemma D.5, we have \( t'_1 \lor t'_2 \subseteq_\Delta t_1 \lor t_2 \).

\section{A More Tractable Type Inference System}

\subsection{ALG-INST: consider the derivation}

\[
\begin{align*}
\Delta &\vdash_\Pi e : t \\
\Delta &\vdash_\Pi e_1 : t_1 \\
\Delta &\vdash_\Pi e_2 : t_2 \\
\Delta &\vdash_\Pi e \? e_1 \? e_2 : t_1 \lor t_2
\end{align*}
\]

By induction, we have

\[
\exists \sigma_k' \in K. \Delta \vdash_\Pi \text{erase}(e) : e' \text{ and } [\sigma_k']_{k \in K} \vdash e' \subseteq_\Delta e
\]

Since \( \text{erase}(e_1)_{j \in J} = \text{erase}(e_1) \), we have \( \Delta \vdash_\Pi \text{erase}(e_1)_{j \in J} : e'_1 \). As \( \Delta \vdash_\Pi \text{erase}(e_1)_{j \in J} = e'_1 \), we have \( \bigwedge_{j \in J} [\sigma_k'_{j \in J} \vdash e'_1 \subseteq_\Delta \sigma_k'_{j \in J} \vdash e \], that is \( \bigwedge_{j \in J} [\sigma_k'_{j \in J} \vdash e'_1 \subseteq_\Delta \sigma_k'_{j \in J} \vdash e \]. Moreover, it is clear that \( \sigma_k' \vdash e \). Therefore, we get \( e' \subseteq_\Delta \bigwedge_{j \in J} \sigma_k'_{j \in J} \).

\subsection{D.3 A More Tractable Type Inference System}

With the rules of Figure 9 when type-checking an implicitly-typed expression, we have to compute sets of type substitutions for projections, applications, abstractions and type cases. Because type substitutions inference is a costly operation, we would like to perform it as less as possible. To this end, we give in this section a restricted version of the inference system, which is non complete but still sound and powerful enough to be used in practice.

First, we want to simplify the type inference rule for projections:

\[
\begin{align*}
\Delta &\vdash_\Pi a : t \\
\Delta &\vdash_\Pi \pi_1(a) : \pi_1(t)
\end{align*}
\]

where \( \Pi_1(t) = \{ u | \sigma_{j \in J} \vdash e \subseteq_\Delta u = \pi_1(\bigwedge_{j \in J} \sigma_k) \} \). Instead of picking any type in \( \Pi_1(t) \), we would like to simply project \( t \), i.e., assign the type \( \pi_1(t) \) to \( \pi_1(a) \). By doing so, we lose completeness on pair types that contain top-level variables. For example, if \( t = (\text{Int} \times \text{Int}) \land \alpha \), then \( \text{Int} \land \text{Bool} \in \Pi_1(t) \) (because \( \alpha \) can be instantiated with \( \text{Bool} \land \text{Bool} \)), but \( \pi_1(t) = \text{Int} \). We also lose typability if \( t \) is not a pair type, but can be instantiated in a pair type. For example, the type of \( (\text{Int} \lor (\beta \rightarrow \alpha) \lor (\beta \rightarrow \beta) \) is \( (\text{Int} \times \text{Int}) \lor ((\beta \rightarrow \beta) \land \text{Int} \rightarrow \text{Int}) \) which is not a pair type, but can be instantiated in \( (\text{Int} \times \text{Int}) \) by taking \( \beta = \text{Int} \). We believe these kinds of types will not be written by programmers, and it is safe to use the following projection rule in practice.

\[
\begin{align*}
\Delta &\vdash_\Pi a : t \\
\Delta &\vdash_\Pi \pi_1(a) : \pi_1(t)
\end{align*}
\]
We now look at the type inference rules for the type case $a \in t \ ? a_1 : a_2$. The four different rules consider the different possible instantiations that make the type $t'$ inferred for a fit $t$ or not. For the sake of simplicity, we decide not to infer type substitutions for polymorphic arguments of type cases. Indeed, in the term $(\lambda^{\alpha \rightarrow \alpha}.x.x) \in \text{Int} \rightarrow \text{Int} ? \\text{true} : \text{false}$, we assume the programmer wants to do a type case on the polymorphic identity, and not on one of its instance (otherwise, he would have written the instantiated interface directly), so we do not try to instantiate it. And in any case there is no real reason for which the inference system should choose to instantiate the indentity by $\text{Int} \rightarrow \text{Int}$ (and thus make the test succeed) rather than $\text{Bool} \rightarrow \text{Bool}$ (and thus make the test fail). If we decide not to infer types for polymorphic arguments of type-case expression, then since $\alpha \rightarrow \alpha$ is not a subtype of $\text{Int} \rightarrow \text{Int}$ (we have $\alpha \rightarrow \alpha \not\subseteq \text{Int} \rightarrow \text{Int}$ but $\alpha \rightarrow \alpha \not\subseteq \text{Int} \rightarrow \text{Int}$) the term evaluates to $\text{false}$. With this choice, we can merge the different inference rules into the following one.

\[
\begin{align*}
\Delta \vdash \Gamma \vdash a : t' \quad t_1 = t' \land t & \quad t_2 = t' \land \lnot t \\
\Delta \vdash \Gamma \vdash a_i : s_i & \\
\vdash \Gamma \vdash (a \in t ? a_1 : a_2) : \bigvee_{i \neq 0} s_i
\end{align*}
\]

(INF-CASE')

Finally, consider the inference rule for abstractions:

\[
\begin{align*}
\forall i \in I. & \quad \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \vdash \Gamma, (x : t_i) \vdash \lambda_i : t_i \rightarrow s_i a : \bigwedge_{i \in I} t_i \rightarrow s_i \\
\Delta \vdash \Gamma \vdash \lambda x : t_i \rightarrow s_i a : \bigwedge_{i \in I} t_i \rightarrow s_i
\end{align*}
\]

(INF-ABSTR')

We verify that the abstraction can be typed with each arrow type $t_i \rightarrow s_i$ in the interface. Meanwhile, we also infer a set of type substitutions to tally the type $s_i'$ we infer for the body expression with $s_i$. In practice, similarly, we expect that the abstraction is well-typed only if the type $s_i'$ we infer for the body expression is a subtype of $s_i$. For example, the expression $\lambda^{\text{Bool} \rightarrow (\text{Int} \rightarrow \text{Int})}.x.x \in \text{true} ? (\lambda^{\alpha \rightarrow \alpha}.y.(\lambda^{\alpha \rightarrow \alpha}.z.y))$ is well-typed while $\lambda^{\text{Bool} \rightarrow (\alpha \rightarrow \alpha)}.x \in \text{true} ? (\lambda^{\alpha \rightarrow \alpha}.y.(\lambda^{\alpha \rightarrow \alpha}.z.y))$ is well-typed. So we use the following restricted rule for abstractions instead:

\[
\begin{align*}
\forall i \in I. & \quad \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \vdash \Gamma, (x : t_i) \vdash a : s_i \\
\Delta \vdash \Gamma \vdash \lambda x : t_i \rightarrow s_i a : \bigwedge_{i \in I} t_i \rightarrow s_i
\end{align*}
\]

(INF-ABSTR')

In conclusion, we restrict the inference of type substitutions to applications. We give in Figure 10 the inference rules of the system which respects the above restrictions. With these new rules, the system remains sound, but is not complete.

\[
\begin{align*}
\Delta \vdash \Gamma \vdash c : b & \quad \text{(INF-CONST)} \\
\Delta \vdash \Gamma \vdash a_1 : t_1 & \quad \Delta \vdash \Gamma \vdash a_2 : t_2 & \quad \text{(INF-PAIR)} \\
\Delta \vdash \Gamma \vdash (a_1, a_2) : t_1 \times t_2 & \\
\Delta \vdash \Gamma \vdash x : \Gamma(x) & \quad \text{(INF-VAR)} \\
\Delta \vdash \Gamma \vdash a : t & \quad t \leq \pi \in \Delta & \quad \text{(INF-PROJ')} \\
\Delta \vdash \Gamma \vdash \pi(a) : \pi(t) & \quad \text{(INF-APPL)} \\
\forall i \in I. & \quad \Delta \cup \text{var}(\bigwedge_{i \in I} t_i \rightarrow s_i) \vdash \Gamma, (x : t_i) \vdash a : s_i \\
\Delta \vdash \Gamma \vdash \lambda x : t_i \rightarrow s_i a : \bigwedge_{i \in I} t_i \rightarrow s_i & \quad \text{(INF-ABSTR')} \\
\Delta \vdash \Gamma \vdash a : t' \quad t_1 = t' \land t & \quad t_2 = t' \land \lnot t \\
\Delta \vdash \Gamma \vdash a_i : s_i & \\
\vdash \Gamma \vdash (a \in t ? a_1 : a_2) : \bigvee_{i \neq 0} s_i & \quad \text{(INF-CASE')} \\
\end{align*}
\]

Figure 10. Restricted Type-Substitution Inference System

71

2012/11
E. Type Tallying

Given two types $t$ and $s$, the goal of this section is to find pairs of sets of type-substitutions $[\sigma_j]_{j \in I}, s[\sigma_i]_{i \in I}$ such that $t[\sigma_j]_{j \in J} \leq s[\sigma_i]_{i \in I}$. Assuming that the cardinalities of $I$ and $J$ are known, this problem can be reduced to a type tallying problem, that we define and solve first. We then explain how we can reduce the original problem to the type tallying problem, and provide a semi-algorithm for the original problem. Finally, we give some heuristic to establish the upper bounds (which depend on $t$ and $s$) for the cardinalities of $I$ and $J$.

E.1 Type Tallying Problem

Given a finite set $C$ of pairs of types and a finite set $\Delta$ of type variables, the tallying problem for $C$ and $\Delta$ consists in verifying whether there exists a substitution $\sigma$ such that $\sigma \notin \Delta$ and for all $(s, t) \in C$, $s\sigma \leq t\sigma$ holds. In this section we denote constraints as triples. The notation is different from the one used in Section 4 in that it also specifies the symbol of the relation. So a pair of types $(s, t) \in C$ corresponds to the constraint $(s \leq t)$:

**Definition E.1 (Constraints).** A constraint $(t, c, s)$ is a triple belonging to $T \times \{\leq, \geq\} \times T$. Let $C$ denote the set of all constraints.

Given a constraint-set $C$, the set of type variable occurring in $C$ is defined as

$$\text{var}(C) = \bigcup_{(t, c, s) \in C} \text{var}(t) \cup \text{var}(s)$$

**Definition E.2 (Normalized Constraint).** A constraint $(t, c, s)$ is said to be normalized if $t$ is a type variable. A constraint-set $C \subseteq C$ is said to be normalized if every constraint $(t, c, s) \in C$ is normalized. Given a normalized constraint-set $C$, its domain is defined as $\text{dom}(C) = \{\alpha | \exists (\alpha, c, s) \in C\}$.

**Definition E.3 (Constraint Solution).** Let $C \subseteq C$ be a constraint-set. A solution to $C$ is a substitution $\sigma$ such that

$$\forall(t, \leq, s) \in C. t\sigma \leq s\sigma \text{ holds and } \forall(t, \geq, s) \in C. s\sigma \leq t\sigma \text{ holds.}$$

If $\sigma$ is a solution to $C$, we write $\sigma \models C$.

**Definition E.4.** Given two sets of constraint-sets $S_1, S_2 \subseteq \mathcal{P}(C)$, we define their union as $S_1 \cup S_2 = S_1 \cup S_2$ and their intersection as $S_1 \cap S_2 = \{C_1 \cup C_2 \mid C_1 \in S_1, C_2 \in S_2\}$.

Given a constraint-set $C$, the constraint solving algorithm produces the set of all the solutions of $C$ by following the algorithm given in Section 4.3.1. Let us examine each step of the algorithm on some examples.

Step 1: normalize each constraint.

Because normalized constraints are easier to solve than regular ones, we first turn any constraint-set into an equivalent set of normalized constraint-sets according to the decomposition rules in [3]. For example, the following constraint-set which contains only one constraint

$$\{(\text{Int } \to \text{Int}) \land (\text{Bool } \to \text{Bool}), \leq, \alpha \to \alpha\}$$

is equivalent to the following set of normalized constraint-sets

$$S = \{((\alpha, \leq, 0), (\alpha, \geq, \text{Int } \land \text{Bool})) \}
\cap \{((\alpha, \leq, \text{Int}), (\alpha, \geq, \text{Bool})) \}
\cap \{((\alpha, \leq, \text{Bool}), (\alpha, \geq, \text{Int})) \}
\cap \{((\alpha, \leq, \text{Int } \lor \text{Bool})) \}$$

The set $S$ contains in fact 8 constraint-sets (see Definition E.4) In what follows, we explain how we build a solution to [19] from the constraint-set

$$\{((\alpha, \geq, \text{Int}), (\alpha, \geq, \text{Bool}), (\alpha, \leq, \text{Int } \lor \text{Bool}))\}.$$  \hspace{1cm} (20)

which belongs to $S$.

Step 2: saturate each constraint-set.
Each constraint-set contained in $S$ may contain several constraints of the form $(\alpha, \geq, t)$ (resp. $(\alpha, \leq, t)$), which give different lower bounds (resp. upper bounds) for $\alpha$. We merge all these constraints into one using union (resp. intersection). For example, from $\alpha \geq \text{Int}$ and $\alpha \geq \text{Bool}$, we deduce $\alpha \geq \text{Int} \lor \text{Bool}$. As a result, (20) is equivalent to

$$\{(\alpha, \geq, \text{Int} \lor \text{Bool}), (\alpha, \leq, \text{Int} \lor \text{Bool})\}.$$  (21)

If a type variable has both a lower bound $s$ and an upper bound $t$, then the solutions we are looking for should satisfy the constraint $(s, \leq, t)$ as well. Therefore, we have to saturate $C'$ with $(s, \leq, t)$, which has to be normalized, merged, and saturated itself first. Because $\text{Int} \lor \text{Bool} \leq \text{Int} \lor \text{Bool}$ holds trivially, the constraint-set (21) is already saturated.

Finally, Step 3 of the algorithm can be logically split in two simpler steps.

Step 3.1: transform each constraint-set into an equation system.

To transform constraints into equations, we use the property that some set of constraints is satisfied for all assignments of $\alpha$ included between $s$ and $t$ if and only if the this same set in which we replace $\alpha$ by $(s \lor \alpha') \land \mu$ is satisfied for all possible assignments of $\alpha'$ (with $\alpha'$ fresh). Of course such a transformation works only if $s \leq t$, but remember that we checked that this hold at the moment of the saturation. By performing this replacement for each variable we obtain a system of equations. For example, the constraint set (21) is equivalent to the following equation

$$\alpha = ((\text{Int} \lor \text{Bool}) \lor \alpha') \land (\text{Int} \lor \text{Bool})$$  (22)

where $\alpha'$ is a fresh type variable.

Step 3.2: solve each equation system.

Finally, we solve each equation system, which gives us a substitution which is a solution of the original constraint-set $C$. For example, Equation (22) is equivalent to $\alpha = \text{Int} \lor \text{Bool}$. Therefore, the type substitution $\{\text{Int} \lor \text{Bool}\}_{\alpha}$, which is a solution to the constraint-set (19).

In the following subsections we study in details each step of the algorithm.

E.1.1 Constraints Normalization

The type tallying problem is quite similar to the subtyping problem presented in [3]. We therefore reuse most of the technology developed in [3], such as, for example, the transformation of the subtyping problem into an emptiness decision problem, the elimination of top-level constructors, and so on. One of the main differences is that we do not want to eliminate top-level type variables from constraints, but, rather, we want to isolate them to build sets of normalized constraints (from which we then construct sets of substitutions).

In general, normalizing a constraint generates a set of constraints. For example, $(\alpha \lor \beta, \geq, 0)$ holds if and only if $(\alpha, \geq, 0)$ or $(\beta, \geq, 0)$ holds; therefore the constraint $(\alpha \lor \beta, \geq, 0)$ is equivalent to the normalized constraint-set $\{(\alpha, \geq, 0), (\beta, \geq, 0)\}$. Consequently, the normalization of a constraint-set $C$ yields a set $S$ of normalized constraint-sets.

Several normalized sets may be suitable replacements for a given constraint; for example, $\{(\alpha, \leq, \text{Int} \lor \text{Bool}), (\alpha, \geq, \text{Int} \lor \text{Bool})\}$ and $\{(\alpha, \leq, \text{Int} \lor \text{Bool}), (\alpha, \geq, \text{Int} \lor \text{Bool})\}$ are clearly equivalent normalized sets. However, the equation systems generated by the algorithm for these two sets are completely different equation systems and yield different substitutions. Concretely, $\{(\alpha, \leq, \text{Int} \lor \text{Bool}), (\beta, \leq, \alpha \lor t_1), (\beta, \leq, \alpha \lor t_2)\}$ generates the equation system $\alpha = \alpha' \land (\beta \lor t_1), \beta = \beta' \land (\alpha \lor t_2)$, which in turn gives the substitution $\sigma_1$ such that

$$\sigma_1(\alpha) = \mu x. ((\alpha' \land \beta' \land x) \lor (\alpha' \land \beta' \land t_2) \lor (\alpha' \land t_1))$$
$$\sigma_1(\beta) = \mu x. ((\beta' \land \alpha' \land x) \lor (\beta' \land \alpha' \land t_1) \lor (\beta' \land t_2))$$

where $\alpha'$ and $\beta'$ are fresh type variables (see Section E.1.4 for more details) and we used the $\mu$ notation to denote regular recursive types. These recursive types are not valid in our calculus, because $x$ does occur under a type constructor (this means that the unfolding of the type does not satisfy the property that every infinite branch contains infinitely many occurrences of type constructors. In contrast, the equation system built from $\{(\alpha, \leq, \text{Int} \lor \text{Bool}), (\alpha, \geq, \text{Int} \lor \text{Bool})\}$ is $\alpha = ((\beta' \lor t_2) \land \alpha') \land (\beta \lor t_1)$, and the corresponding substitution is $\sigma_2 = \mu x. (((\beta' \lor t_2) \lor \alpha') \land (\beta \lor t_1))$, which is valid since it maps the type variable $\alpha$ into a well-formed type. Ill-formed recursive types are generated when there exists a chain $\alpha_0 = a_0 \ B_0 \ a_1 \ B_1 \ a_2 \ B_2 \ ... \ a_n \ B_n = \alpha_n = a_{n+1} \ B_{n+1} \ a_{n+2} \ B_{n+2} \ ...$ (where $B_i \in \{\land, \lor\}$ for all $i$, and $n \geq 0$) in the equation system built from the normalized constraint-set. This chain implies the equation $\alpha_0 = a_0 \ B \ t'$ for some $B \in \{\land, \lor\}$ and $t'$, whose the solution is an ill-formed recursive type for $\alpha_0$. To avoid this issue, we give an arbitrary order on type variables occurring in the constraint-set $C$ such that different type variable have different ordinals. Then we always select the normalized constraint $(\alpha, c, t)$ such that the ordinal of $\alpha$ is smaller than all the ordinals of the top-level type variables in $t$. As a result, the transformed equation system does not contain any problematic chain like the one above.

\*Or by $s \lor (\alpha' \land t)$. 73

2012/7/11
Definition E.5 (Order). Let $V$ be a set of type variables. An order $O$ on $V$ is an injective map from $V$ to $\mathbb{N}$.

We formalize normalization as a judgement $\Sigma \vdash^N C \rightsquigarrow \mathcal{S}$, which states that under the environment $\Sigma$ (which, informally, contains the types that have already been processed at this point), $C$ is normalized to $\mathcal{S}$. The judgement is derived according the rules of Figure 11. These rules describe the same algorithm as the function $\text{norm}$ given in Figure 4 (i.e., $\Sigma \vdash^N \{(t, \leq 0)\} \rightsquigarrow \text{norm}(t, \Sigma)$ is provable in the system of Figure 11) but extended to handle also product types. We just switched to a deduction systems since it ease the formal treatment.

\[
\begin{align*}
\frac{\Sigma \vdash^N \emptyset}{\Sigma \vdash^N \emptyset \rightarrow \emptyset} \quad & \text{(NEMPTY)} \\
\frac{\Sigma \vdash^N \{(t_i, c_i t_i')\} \rightarrow S_i \quad \bigwedge \quad \Sigma \vdash^N \{(t_i, c_i t_i')\} \rightarrow \bigoplus_{i \in I} S_i}{\Sigma \vdash^N \{(t_i \land t_i' \leq 0)\} \rightarrow S \quad \text{if } t_i' \neq 0} \quad & \text{(JOIN)} \\
\frac{\Sigma \vdash^N \{(t_i \land t_i' \leq 0)\} \rightarrow S \quad \text{if } t_i' \neq 0}{\Sigma \vdash^N \{(t_i \leq t_i')\} \rightarrow S} \quad & \text{(ZERO)} \\
\frac{\Sigma \vdash^N \{(t \leq t')\} \rightarrow S \quad \Sigma \vdash^N \{(t' \geq t)\} \rightarrow S}{\Sigma \vdash^N \{(t \leq t')\} \rightarrow S} \quad & \text{(NSYM)} \\
\frac{\Sigma \vdash^N \{(t \leq 0)\} \rightarrow S \quad \Sigma \vdash^N \{(t \leq 0)\} \rightarrow S}{\Sigma \vdash^N \{(t \leq 0)\} \rightarrow S} \quad & \text{(NDNF)} \\
\frac{\Sigma \vdash^N \{(\tau \in \mathcal{N})\} \rightarrow \emptyset}{\text{thv}(\tau_0) = \emptyset} \quad & \text{(NTP)} \\
\frac{\Sigma \vdash^N \{(\tau \in \mathcal{N})\} \rightarrow \emptyset}{\Sigma \vdash^N \{(\tau_0 \leq 0)\} \rightarrow \emptyset} \quad & \text{(NASSUM)} \\
\frac{\forall \alpha \in \mathcal{N}, \forall b_i \land \neg b_j \leq 0}{\Sigma \vdash^N \{(b_i \land \neg b_j) \leq 0\} \rightarrow \emptyset} \quad & \text{(NBASIC-T)} \\
\frac{\forall \alpha \in \mathcal{N}, \forall b_i \land \neg b_j \leq 0}{\Sigma \vdash^N \{(b_i \land \neg b_j) \leq 0\} \rightarrow \emptyset} \quad & \text{(NBASIC-F)} \\
\frac{\forall \alpha \in \mathcal{N}, \forall b_i : S_{t_i'}^0 \rightarrow \bigoplus_{i \in \mathcal{N}} S_{t_i'}^0 \rightarrow S_{t_i'}^0 \quad \text{if } \neg t_i' \leq 0}{\Sigma \vdash^N \{(t \land t' \leq 0)\} \rightarrow \bigoplus_{i \in \mathcal{N}} (S_{t_i'}^0 \cup S_{t_i'}^0)} \quad & \text{(NPROD)} \\
\frac{\exists \alpha \in \mathcal{N}, \forall b_i : S_{t_i'}^0 \rightarrow \bigoplus_{i \in \mathcal{N}} S_{t_i'}^0 \rightarrow S_{t_i'}^0 \quad \text{if } \neg t_i' \leq 0}{\Sigma \vdash^N \{(t \land t' \leq 0)\} \rightarrow \bigoplus_{i \in \mathcal{N}} (S_{t_i'}^0 \cup S_{t_i'}^0)} \quad & \text{(NARROW)} \\
\end{align*}
\]

where $t$ and $t'$ with scripts are types, $c_i$ belongs to $\{\leq, \geq\}$, $\tau_0$ and $\tau_i$ are single normal forms, $\alpha \vdash^O P \cup \mathcal{N}$ denotes $\alpha$ has the smallest ordinal in $P \cup \mathcal{N}$ under $O$, and $t_{\alpha'}$ is the type obtained from $\bigwedge_{\alpha' \in \mathcal{N}} \alpha \land \bigwedge_{\alpha' \in \mathcal{N}} \neg \alpha \land \neg \tau_0$ by eliminating $\alpha'$.

Figure 11. Normalization rules.

If the constraint-set is empty, then clearly any substitution is a solution, and, the result of the normalization is simply the singleton containing the empty set (rule (NEMPTY)). Otherwise, each constraint is normalized separately, and the normalization of the constraint-set is the intersection of the normalizations of each constraint (rule (NJOIN)). By using rules (NSYM), (ZERO), and (NDNF) repeatedly, we transform any constraint into the constraint of the form $(\tau \in \mathcal{N})$ where $\tau$ is disjunctive normal form: the first rule reverses $(t' \geq t)$ into $(t \leq t')$, the second rule moves the type $t'$ from the right of $\leq$ to the left, yielding $(t \land \neg t' \leq 0)$, and finally the last rule puts $t \land \neg t'$ in disjunctive normal form. Such a type $\tau$ is called the type to be normalized. If $\tau$ is a union of single normal forms, the rule (UNION) splits the union of single normal forms into constraints featuring each of the single normal form. Then the results of each constraint normalization would be joined by the rule (NJOIN).

The following rules handle constraints of the form $(\tau \leq 0)$, where $\tau$ is a single normal form. If there are some top-level type variables, the rule (N Lomb) generates a normalized constraint for the top-level type variable whose ordinal is the smallest. Otherwise, there are no top-level type variables. If $\tau$ has already
been normalized (i.e., which belongs to $\Sigma$) is not processed again (rule (NHYP)). Otherwise, we memoize it and then process it using the predicate for single normal forms $\Sigma \vdash_C^* C \rightsquigarrow \mathcal{S}$ (rule (NASSUM)). Note that switching from $\Sigma \vdash_C^* C \rightsquigarrow \mathcal{S}$ to $\Sigma \vdash_C^* C \rightsquigarrow \mathcal{S}$ prevents the incorrect use of (NHYP) just after (NASSUM), which would wrongly say that any type is normalized without doing any computation.

Finally, the last four rules state how to normalize constraints of the form $(\tau, \leq, \emptyset)$ where $\tau$ is a single normal form and contains no top-level type variables. Thereby $\tau$ should be an intersection of atoms with the same constructor. If $\tau$ is an intersection of basic types, normalizing is equivalent to checking whether $\tau$ is empty or not: if it is (rule (NBASIC-T)), we return the singleton containing the empty set (any substitution is a solution), otherwise there is no solution and we return the empty set (rule (NBASIC-F)). When $\tau$ is an intersection of products, the rule (NPROD) decomposes $\tau$ into several candidate types (following Lemma 3.11 in [3]), which are to be further normalized. The case when $\tau$ is an intersection of arrows (rule (NARROW)) is treated similarly. Note that, in the last two rules, we switch from $\tau \rightsquigarrow \mathcal{S}$ in the premises to ensure termination.

If $\emptyset \vdash_C^* C \rightsquigarrow \mathcal{S}$, then $\mathcal{S}$ is the result of the normalization of $C$. We now prove soundness, completeness, and termination of the constraint normalization algorithm.

To prove soundness, we use a family of subtyping relations $\leq_n$ that layer $\leq$ (i.e., such that $\bigcup_{n \in \mathbb{N}} \leq_n = \leq$), defined as follows.

**Definition E.6.** Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support ($\llbracket \cdot \rrbracket, \mathcal{D}$). We define the family $(\leq_n)_{n \in \mathbb{N}}$ of subtyping relations as

$$t \leq_n s \iff \forall \eta. [t]_n[\eta] \subseteq [s]_n[\eta]$$

where $[\cdot]_n$ is the rank $n$ interpretation of a type, defined as

$$[t]_n[\eta] = \{ d \in [t]_\eta | \text{height}(d) \leq n \}$$

and $\text{height}(d)$ is the height of an element $d$ in $\mathcal{D}$, defined as

$$\begin{align*}
\text{height}(c) &= 1 \\
\text{height}((d, d')) &= \max(\text{height}(d), \text{height}(d')) + 1 \\
\text{height}(\{ (d_1, d'_1), \ldots, (d_n, d'_n) \}) &= \begin{cases} 1 & n = 0 \\ \max(\text{height}(d_i), \text{height}(d'_i)) + 1 & n > 0 \end{cases}
\end{align*}$$

**Lemma E.7.** Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then

(1) $t \leq_0 s$ for all $t, s \in \mathcal{T}$.
(2) $t \leq s \iff \forall n. t \leq_n s$.
(3) $\bigwedge_{i \in I} (t_i \times s_i) \leq_{n+1} \bigvee_{j \in J} (t_j \times s_j) \iff \forall J', \subseteq J. \begin{cases} \bigwedge_{i \in I} t_i \leq_n \bigvee_{j \in J \setminus J'} t_j \\ \bigwedge_{i \in I} s_i \leq_n \bigvee_{j \in J \setminus J'} s_j \end{cases}$
(4) $\bigwedge_{i \in I} (t_i \rightarrow s_i) \leq_{n+1} \bigvee_{j \in J} (t_j \rightarrow s_j) \iff \exists j_0 \in J. \forall I', \subseteq I. \begin{cases} \bigwedge_{i \in I} t_i \leq_n \bigvee_{i \in I'} t_i \\ \bigwedge_{i \in I} s_i \leq_n \bigvee_{i \in I \setminus I'} s_i \end{cases}$

Proof: (1) straightforward.
(2) straightforward.
(3) it follows by Lemma 3.11 in [3] and Definition E.6.
(4) it follows by Lemma 3.12 in [3] and Definition E.6.

Given a constraint-set $C$ and a type substitution $\sigma$, we define the rank $n$ satisfaction predicate $\models_n$ as

$$\sigma \models_n C \iff \forall (t, c, s) \in C. \ t \models_n s$$

**Lemma E.8.** Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then

(1) $\sigma \models_0 C$ for all $\sigma$ and $C$.
(2) $\sigma \models C \iff \forall n. \sigma \models_n C$.

\(^{10}\) See [3] for the definitions of the notions of models, interpretations, and assignments.

Given a set of types \( \Sigma \), we write \( C(\Sigma) \) for the constraint-set \( \{ (t, \leq, 0) \mid t \in \Sigma \} \).

**Lemma E.9** (Soundness). Let \( C \) be a constraint-set. If \( \emptyset \vdash_{\sim} C \leadsto S \), then for all normalized constraint-set \( C' \in S \) and all substitution \( \sigma \), we have \( \sigma \vdash_{\sim} C' \Rightarrow \sigma \vdash_{\sim} C \).

**Proof.** We prove the following stronger statements.

1. Assume \( \Sigma \vdash_{\sim} C \leadsto S \). For all \( C' \in S \), \( \sigma \) and \( n \), if \( \sigma \vdash_{\sim} C(\Sigma) \) and \( \sigma \vdash_{\sim} C' \), then \( \sigma \vdash_{\sim} C \).
2. Assume \( \Sigma \vdash_{\sim} C \leadsto S \). For all \( C' \in S \), \( \sigma \) and \( n \), if \( \sigma \vdash_{\sim} C(\Sigma) \) and \( \sigma \vdash_{\sim} C' \), then \( \sigma \vdash_{\sim} C' \).

Before proving these statements, we explain how the first property implies the lemma. Suppose \( \emptyset \vdash_{\sim} C \leadsto S \). Given a set of types \( \sigma \in S \), then there exists \( \emptyset \vdash_{\sim} C \leadsto S \). By Property (1), we have \( \sigma \vdash_{\sim} C \) for all \( \sigma \), and we have then the required result by Lemma E.8.

We prove the two properties simultaneously by induction on the derivations of \( \Sigma \vdash_{\sim} C \leadsto S \) and \( \Sigma \vdash_{\sim} C \leadsto S \).

(NEMPTY): trivially.

(NJOIN): according to Definition E.4, if there exists \( C_1 \in S \), such that \( C_1 = \emptyset \), then \( \emptyset \vdash_{\sim} C_1 = \emptyset \), and the result follows immediately. Otherwise, we have \( C' = \bigcup_{i \in I} C_i \), where \( C_i \in S_i \). As \( \sigma \vdash_{\sim} C' \), then clearly \( \sigma \vdash_{\sim} C_i \). By induction, we have \( \sigma \vdash_{\sim} C \{ (t_i, c_i, t_i') \} \). Therefore, we get \( \sigma \vdash_{\sim} C \{ (t_i, c_i, t_i') \mid i \in I \} \).

(NSYM): by induction, we have \( \sigma \vdash_{\sim} C \{ (t' \spadesuit t \leq t' \} \). Then clearly \( \sigma \vdash_{\sim} C \{ (t' \spadesuit t \leq t' \}.

(NZERO): by induction, we have \( \sigma \vdash_{\sim} C \{ (t \spadesuit t \leq t) \} \). According to set-theory, we have \( \sigma \vdash_{\sim} C \{ (t \spadesuit t) \} \).

(NDNEF): similar to the case of (NZERO).

(NUNION): similar to the case of (NZERO).

(NNTL): assume \( \alpha' \) has the smallest ordinal in \( P \cup N \). If \( \alpha' \in P \), then we have \( C' = (\alpha', \leq, - \alpha') \). From \( \sigma \vdash_{\sim} C' \), we deduce \( \sigma(\alpha') \leq \alpha' \). According to set-theory, we have \( \sigma(\alpha') \leq \alpha' \). Hence, we can prove \( \emptyset \vdash_{\sim} C(\Sigma) \) by induction on \( \sigma \vdash_{\sim} C \).

(NHASUM): if \( n = 0 \), then \( \sigma \vdash_{\sim} C \{ (\tau_0 \leq 0) \} \). Suppose \( n > 0 \). From \( \sigma \vdash_{\sim} C(\Sigma) \) and \( \sigma \vdash_{\sim} C \), it is easy to prove that \( \sigma \vdash_{\sim} C(\Sigma) \). By induction hypothesis, we have \( \sigma \vdash_{\sim} C \{ (\tau_0 \leq 0) \} \).

(NHYP): by induction, we have \( \sigma \vdash_{\sim} C \{ (\tau_0 \leq 0) \} \).

(NPROD): if \( \bigcap_{N \in C(\Sigma)} (S^N \cup S^N) = \emptyset \), then the result follows trivially. Otherwise, we have \( C' = \bigcup_{N \in C(\Sigma)} (S^N \cup S^N) \). As \( \sigma \vdash_{\sim} C' \), we have \( \sigma \vdash_{\sim} C(N) \) for all \( N \). Moreover, following Definition E.4, either \( C(N) \in S^N \cup S^N \). By induction, we have either \( \sigma \vdash_{\sim} C(\Sigma) \{ \tau_0 \leq 0 \} \) or \( \sigma \vdash_{\sim} C \{ \tau_0 \leq 0 \} \).

Applying Lemma E.7, we have

\[
\bigwedge_{i \in P} t_i^1 \sigma \land \bigwedge_{j \in N'} -t_j^1 \sigma \leq 0 \quad \text{or} \quad \bigwedge_{i \in P} t_i^2 \sigma \land \bigwedge_{j \in N' \cap N'} -t_j^2 \sigma \leq 0
\]

(NARROW): similar to the case of (NPROD).

**Lemma E.10** (Completeness). Let \( C \) be a constraint-set such that \( \emptyset \vdash_{\sim} C \leadsto S \). For all substitution \( \sigma \), if \( \sigma \vdash_{\sim} C \), then there exists \( C' \in S \) such that \( \sigma \vdash_{\sim} C' \).

**Proof.** We prove the following stronger statements.

1. Assume \( \Sigma \vdash_{\sim} C \leadsto S \). For all \( \sigma \), if \( \sigma \vdash_{\sim} C(\Sigma) \) and \( \sigma \vdash_{\sim} C \), then there exists \( C' \in S \) such that \( \sigma \vdash_{\sim} C' \).
2. Assume \( \Sigma \vdash_{\sim} C \leadsto S \). For all \( \sigma \), if \( \sigma \vdash_{\sim} C(\Sigma) \) and \( \sigma \vdash_{\sim} C \), then there exists \( C' \in S \) such that \( \sigma \vdash_{\sim} C' \).

The result is then a direct consequence of the first item (indeed, we have \( \sigma \vdash_{\sim} C(\emptyset) \) for all \( \sigma \)).
(\textbf{NEMPTY}): trivially.
(\textbf{NOJOIN}): as $\sigma \models \{(t_i, c_i, t')_i \mid i \in I\}$, we have in particular $\sigma \models \{(t_i, c_i, t')_i \}$ for all $i$. By induction, there exists $C_i \in S_i$ such that $\sigma \models C_i$. So $\sigma \models \bigcup_{i \in I} C_i$. Moreover, according to Definition \textbf{E.4}, $\bigcup_{i \in I} C_i$ must be in $\prod_{i \in I} S_i$. Therefore, the result follows.
(\textbf{NSYM}): if $\sigma \models \{(t_i' \geq t_i)\}$, then clearly $\sigma \models \{(t_i \leq t_i')\}$. By induction, the result follows.
(\textbf{NZERO}): since $\sigma \models \{(t_i \leq t_i')\}$, according to set-theory, $\sigma \models \{(t_i \wedge \neg t_i' \leq 0)\}$. By induction, the result follows.
(\textbf{NDN}: similar to the case of (\textbf{NZERO}).
(\textbf{UNION}): similar to the case of (\textbf{NZERO}).
(\textbf{NTLY}): assume $\alpha'$ has the smallest ordinal in $P \cup N$. If $\alpha' \in P$, then according to set-theory, we have $\alpha' \sigma \leq \{\bigcup\alpha \in |P|, \alpha'\} \cup \{\alpha \wedge \neg \tau_0\}$, that is $\sigma \models \{\langle \alpha' \leq \neg \tau_0 \rangle\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.
(\textbf{NYHP}): it is clear that $\sigma \models \emptyset$.
(\textbf{ASSUM}): as $\sigma \models C(\Sigma)$ and $\sigma \models \{(\tau_0 \leq 0)\}$, we have $\sigma \models C(\Sigma \cup \{\tau_0\})$. By induction, the result follows.
(\textbf{NBASIC}): trivially.
(\textbf{NPROD}): as $$\sigma \models \{(\bigwedge_{i \in P} t_i^1 \times t_i^2 \wedge \bigwedge_{j \in N} \neg (t_j^1 \times t_j^2) \leq 0)\}$$
we have $$\bigwedge_{i \in P} t_i^1 \sigma \wedge \bigwedge_{j \in N} \neg t_j^1 \sigma \leq 0$$
Applying Lemma 3.11 in \cite{3}, for all subset $N' \subseteq N$, we have $$\bigwedge_{i \in P} t_i^1 \sigma \wedge \bigwedge_{j \in N'} \neg t_j^1 \sigma \leq 0$$
that is, $$\sigma \models \{(\bigwedge_{i \in P} t_i^1 \wedge \bigwedge_{j \in N'} \neg t_j^1 \leq 0)\}$$
By induction, either there exists $C_{N'} \in S_{N'}$ such that $\sigma \models C_{N'}$, or there exists $C_N \in S_N$ such that $\sigma \models C_N$. According to Definition \textbf{E.4}, we have $C_{N'} \subseteq C_N \cup S_N$. Thus there exists $C_N \in S_N$. Therefore $\sigma \models C_N$. Moreover, according to Definition \textbf{E.4} again, $\bigcup_{N' \subseteq N} C_{N'} \subseteq \bigcup_{N' \subseteq N} (S_{N'} \cup S_N)$. Hence, the result follows.
(\textbf{NARROW}): similar to the case (\textbf{NPROD}) except we use Lemma 3.12 in \cite{3}.}

We now prove termination of the algorithm.

\textbf{Definition E.11 (Plinth).} A plinth $\Sigma \subseteq T$ is a set of types with the following properties:
- $\Sigma$ is finite
- $\Sigma$ contains $\emptyset$, $0$, and is closed under Boolean connectives ($\wedge$, $\vee$, $\neg$)
- for all types $(t_1 \times t_2)$ or $(t_1 \rightarrow t_2)$ in $\Sigma$, we have $t_1 \in \Sigma$ and $t_2 \in \Sigma$

As stated in \cite{3}, every finite set of types is included in a plinth. Indeed, we already know that for a regular type $t$ the set of its subtrees $S$ is finite. The definition of the plinth ensures that the closure of $S$ under Boolean connective is also finite. Moreover, if $t$ belongs to a plinth $\Sigma$, then the set of its subtrees is contained in $\Sigma$. This is used to show the termination of algorithms working on types.

\textbf{Lemma E.12 (Decidability).} Let $C$ be a finite constraint-set. Then the normalization of $C$ terminates.

\textit{Proof.} Let $T$ be the set of types occurring in $C$. As $\Sigma$ is finite, $T$ is finite as well. Let $\Sigma$ be a plinth such that $\Sigma \subseteq T$. Then when we normalize a constraint $\langle t \leq 0 \rangle$ during the process of $\emptyset \vdash \Sigma, C$, $t$ would belong to $\Sigma$. We prove the lemma by induction on $|\Sigma \setminus \Sigma, U, |C|\}$, where $\Sigma$ is the set of types we have normalized, $U$ is the number of unions $\vee$ occurring in the constraint-set $C$ plus the number of constraints $(t, \geq, s)$ and the number of constraint $(t, \leq, s)$ where $s \neq 0$ or $t$ is not in disjunctive normal form, and $C$ is the constraint-set to be normalized.

(\textbf{NEMPTY}): it terminates immediately.
(\textbf{NOJOIN}): $|C|$ decreases.
(\textbf{NSYM}): $U$ decreases.
(\textbf{NZERO}): $U$ decreases.
Lemma E.13 (Finiteness). Let \( C \) be a constraint-set and \( \emptyset \vdash_{N} C \rightarrow S \). Then \( S \) is finite.

Proof. It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints.

Definition E.14. Let \( C \) be a normalized constraint-set and \( O \) an order on \( \text{var}(C) \). We say \( C \) is well-ordered if for all normalized constraint \( (\alpha, c, t_{\alpha}) \in C \) and for all \( \beta \in \text{tv}(t_{\alpha}) \), \( O(\alpha) < O(\beta) \) holds.

Lemma E.15. Let \( C \) be a constraint-set and \( \emptyset \vdash_{N} C \rightarrow S \). Then for all normalized constraint-set \( C' \in S, \ C' \rightarrow S \).

Proof. The only way to generate normalized constraints is Rule (NTLV), where we have selected the normalized constraint for the type variable \( \alpha \) whose ordinal is minimum as the representative one, that is, \( \forall \beta \in \text{tv}(t_{\alpha}) \cdot O(\alpha) < O(\beta) \). Therefore, the result follows.

Lemma E.16. Let \( t, s \) be two types and \( [\rho_{i}]_{i \in I}, [\rho_{j}]_{j \in J} \) two sets of general renamings. Then if \( \emptyset \vdash_{N} (s \land t, \leq, 0) \rightarrow \emptyset \rightarrow \emptyset \vdash_{N} \{ (s \land t, \leq, 0) \rightarrow \emptyset \} \rightarrow \emptyset \)

where \( |K| \geq 2 \) and \( \rho_{i} \)'s are general renamings. For simplicity, we only consider \( |K| = 2 \), as it is easy to extend to the case of \( |K| > 2 \).

Case 1: \( \tau = \tau_{b} \wedge \tau_{b} \) and \( \tau \neq 0 \), where \( \tau_{b} \) (\( \tau_{b} \) resp.) is an intersection of basic types from \( s \) (\( t \) resp.). Then the expansion of \( \tau \) is

\[
(\bigland_{j \in J} \tau_{b} \rho_{j}) \land (\bigland_{i \in I} \tau_{b} \rho_{i}) \simeq \tau_{b} \land \tau_{b} \neq 0
\]

So \( \emptyset \vdash_{N} \{ (\bigland_{j \in J} \tau_{b} \rho_{j}) \land (\bigland_{i \in I} \tau_{b} \rho_{i}), \leq, 0 \} \rightarrow \emptyset \).

Case 2: \( \tau = \bigland_{p_{i} \in P} (w_{p_{i}} \times v_{p_{i}}) \land (\bigland_{n_{i} \in N_{s}} w_{n_{i}} \wedge (\bigland_{n_{i} \in N_{s}} v_{n_{i}}) \land (\bigland_{n_{i} \in N_{s}} - (w_{n_{i}} \wedge v_{n_{i}})).

Since \( \emptyset \vdash_{N} \{ \tau, \leq, 0 \} \rightarrow \emptyset \), by the rule (NPROD), there exists two sets \( N'_{s} \subseteq N_{s} \) and \( N'_{t} \subseteq N_{t} \) such that

\[
\emptyset \vdash_{N} \{ \bigland_{p_{i} \in P} w_{p_{i}} \land \bigland_{n_{i} \in N'_{s} \setminus N'_{t}} - w_{n_{i}} \land \bigland_{p_{i} \in P} v_{p_{i}} \land \bigland_{n_{i} \in N'_{t} \setminus N'_{s}} - v_{n_{i}}, \leq, 0 \} \rightarrow \emptyset
\]

By induction, we have

\[
\emptyset \vdash_{N} \{ (\bigland_{j \in J} (w_{p_{i}} \times v_{p_{i}}) \land (\bigland_{n_{i} \in N'_{s} \setminus N'_{t}} - w_{n_{i}}) \rho_{j} \land (\bigland_{i \in I} (w_{p_{i}} \times v_{p_{i}}) \land (\bigland_{n_{i} \in N'_{t} \setminus N'_{s}} - w_{n_{i}}) \rho_{i}, \leq, 0) \} \rightarrow \emptyset
\]
Then by the rule (NPROD) again, we get
\[ \emptyset \vdash N \{ \bigwedge_{j \in J} (\tau_j) \rho_j \land \bigwedge_{i \in I} (\tau_i) \rho_i, \leq 0 \} \rightsquigarrow \emptyset \]
where \( \tau_s = \bigwedge_{p_i \in P_s} (w_{p_i} \times v_{p_i}) \land \bigwedge_{n_z \in N_s} \neg (w_{n_z} \times v_{n_z}) \) and \( \tau_t = \bigwedge_{p_i \in P_t} (w_{p_i} \times v_{p_i}) \land \bigwedge_{n_z \in N_t} \neg (w_{n_z} \rightarrow v_{n_z}) \).

**Case 3:** \( \tau = \bigwedge_{p_i \in P_s} (w_{p_i} \rightarrow v_{p_i}) \land \bigwedge_{n_z \in N_s} \neg (w_{n_z} \rightarrow v_{n_z}) \land \bigwedge_{p_i \in P_t} (w_{p_i} \rightarrow v_{p_i}) \land \bigwedge_{n_z \in N_t} \neg (w_{n_z} \rightarrow v_{n_z}) \). Since \( \emptyset \vdash N \{ (\tau, \leq 0) \} \rightsquigarrow \emptyset \), by the rule (NARROW), for all \( w \rightarrow v \in N_s \cup N_t \), there exists a set \( P'_s \subseteq P_s \) and a set \( P'_t \subseteq P_t \) such that
\[
\begin{align*}
\emptyset \vdash N \{ \bigwedge_{p_i \in P'_s} \neg w_{p_i} \land \bigwedge_{p_i \in P'_t} \neg w_{p_i} \land w, \leq 0 \} \rightsquigarrow \emptyset \\
P'_s = P_s \land P'_t = P_t \text{ or } \emptyset \vdash N \{ \bigwedge_{p_i \in P'_s} v_{p_i} \land \bigwedge_{p_i \in P'_t} v_{p_i} \land v, \leq 0 \} \rightsquigarrow \emptyset
\end{align*}
\]

By induction, for all \( \rho \in [\rho_i]_{i \in I} \cup [\rho_j]_{j \in J} \), we have
\[
\begin{align*}
\emptyset \vdash N \{ \bigwedge_{j \in J} (\bigwedge_{p_i \in P'_s} \neg w_{p_i}) \rho_j \land \bigwedge_{i \in I} (\bigwedge_{p_i \in P'_t} \neg w_{p_i}) \rho_i, \leq 0 \} \rightsquigarrow \emptyset \\
P'_s = P_s \land P'_t = P_t
\end{align*}
\]

Then by the rule (NARROW) again, we get
\[ \emptyset \vdash N \{ \bigwedge_{j \in J} (\tau_j) \rho_j \land \bigwedge_{i \in I} (\tau_i) \rho_i, \leq 0 \} \rightsquigarrow \emptyset \]
where \( \tau_s = \bigwedge_{p_i \in P_s} (w_{p_i} \rightarrow v_{p_i}) \land \bigwedge_{n_z \in N_s} \neg (w_{n_z} \rightarrow v_{n_z}) \) and \( \tau_t = \bigwedge_{p_i \in P_t} (w_{p_i} \rightarrow v_{p_i}) \land \bigwedge_{n_z \in N_t} \neg (w_{n_z} \rightarrow v_{n_z}) \).

**Case 4:** \( \tau = \bigvee_{k_j \in K_s} (\tau_{k_j}) \land \bigvee_{k_j \in K_t} (\tau_{k_j}) \), where \( \tau_{k_j} \) and \( \tau_{k_i} \) are single normal forms. As \( \emptyset \vdash N \{ (\tau, \leq 0) \} \rightsquigarrow \emptyset \), there must exists at least one \( k_s \in K_s \) and at least one \( k_t \in K_t \) such that \( \emptyset \vdash N \{ (\tau_{k_s} \land \tau_{k_t}, \leq 0) \} \rightsquigarrow \emptyset \). By Cases (1) – (3), the result follows.

The type tallying problem is parametrized with a set \( \Delta \) of type variables that cannot be instantiated, but so far, we have not considered these monomorphic variables in the normalization procedure. Taking \( \Delta \) into account affects only the (NTLV) rule, where a normalized constraint is built by singling out a variable \( \alpha \). Since the type substitution \( \sigma \) we want to construct should not touch the type variables in \( \Delta \) (i.e., \( \sigma \not\in \Delta \)), we cannot choose a variable \( \alpha \) in \( \Delta \). To avoid this, we order the variables in \( C \) so that those belonging to \( \Delta \) are always greater than those not in \( \Delta \). If, by choosing the minimum top-level variable \( \alpha \), we obtain \( \alpha \in \Delta \), it means that all the top-level type variables are contained in \( \Delta \). According to Lemmas C.3 and C.11, we can then safely eliminate these type variables. So taking \( \Delta \) into account, we amend the (NTLV) rule as follows.

\[
\text{tlv}(\tau_0) = 0 \quad \alpha' \notin \Delta \quad P \cup N \quad S = \{ \{(\alpha', \leq \neg t_{\alpha'})\} \} \quad \alpha' \in P \setminus \Delta \\
\{ \{(\alpha', \geq t_{\alpha'})\} \} \quad \alpha' \in N \setminus \Delta \\
\Sigma \vdash N \{ (\tau_0, \leq 0) \} \rightsquigarrow S \quad \text{(NTLV)}
\]

Furthermore, it is easy to prove the soundness, completeness, and termination of the algorithm extended with \( \Delta \).

**E.1.2 Constraints Saturation**

A normalized constraint-set may contain several constraints for the same type variable, which can eventually be merged together. For instance, the constraints \( \alpha \geq t_1 \) and \( \alpha \geq t_2 \) can be replaced by \( \alpha \geq t_1 \lor t_2 \), and the constraints \( \alpha \leq t_1 \) and \( \alpha \leq t_2 \) can be replaced by \( \alpha \leq t_1 \land t_2 \). That is to say, we can merge all the lower bounds (resp. upper bounds) of a type variable into only one by unions (resp. intersections). The merging rules are given in Figure 12, which describe the same algorithm as Step M1 of the function merge given in Subsection 4.3.1.

After repeated uses of the merging rules, a set \( C \) contains at most one lower bound constraint and at most one upper bound constraint for each type variable. If both lower and upper bounds exists for a given \( \alpha \), that is, \( \alpha \geq t_1 \) and \( \alpha \leq t_2 \) belong to \( C \), then the substitution we want to construct from \( C \) should satisfy the constraint \( (t_1, \leq, t_2) \) as well. Therefore, we need to saturate \( C \) with any normalized constraint-set \( C' \in S \),
If \( C \) does not contain any couple \( \alpha \geq t_3 \) and \( \alpha \leq t_2 \) belongs to the constraint-set \( C \) that is being saturated, and \( t_2 \leq t_3 \) has already been processed (i.e., \( (t_1, t_2) \in \Sigma_p \)), then the rule (SHYP) simply extends \( C_S \) (the result of the saturation so far) with \( \{\alpha \geq t_1, \alpha \leq t_2\} \). Otherwise, the rule (SASSUM) first normalizes the fresh constraint \( \{t_1 \leq t_2\} \), yielding a set of normalized constraint-sets \( S \). It then saturates (joins) \( C \) and \( C_S \) with each constraint-set \( C_S \in S \), the union of which gives a new set \( S' \) of normalized constraint-sets. Each \( C' \in S' \) may contain several constraints for the same type variable, so they have to be merged and saturated themselves. Finally, if \( C \) does not contain any couple \( \alpha \geq t_1 \) and \( \alpha \leq t_2 \) for a given \( \alpha \), the process is over and the rule (SDONE) simply returns \( C \cup C_S \).

If \( \emptyset, \emptyset \vdash_C C \sim S \), the result of the normalization of \( \{(t_1, \leq, t_2)\} \). Formally, we describe the saturation process as a judgement \( \Sigma_p, C_S \vdash_S C \sim S \), where \( \Sigma_p \) is a set of type pairs (if \( (t_1, t_2) \in \Sigma_p \), then the constraint \( t_1 \leq t_2 \) has already been treated at this point), \( C_S \) is a normalized constraint-set (which is the result of the saturation of the constraint \( t_1 \leq t_2 \), for \( (t_1, t_2) \in \Sigma_p \), \( C \) is the normalized constraint-set we want to saturate, and \( S \) is a set of sets of normalized constraints (the result of the saturation of \( C \) joined with \( C_S \)). The saturation rules are given in Figure 13, which describe the same algorithm as Step M2 of the function merge given in Subsection 4.3.1.

**Figure 12. Merging rules**

\[
\forall i \in I . (\alpha \geq t_i) \in C \quad |I| \geq 2
\]
\[
\vdash_{M} C \sim (C \setminus \{(\alpha \geq t_i) \mid i \in I\}) \cup \{(\alpha \geq \bigvee_{i \in I} t_i)\} \tag{MLB}
\]

\[
\forall i \in I . (\alpha \leq t_i) \in C \quad |I| \geq 2
\]
\[
\vdash_{M} C \sim (C \setminus \{(\alpha \leq t_i) \mid i \in I\}) \cup \{(\alpha \leq \bigwedge_{i \in I} t_i)\} \tag{MUB}
\]

---

**Figure 13. Saturation Rules**

\[
\Sigma_p, C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_S C \sim S \quad (t_1, t_2) \in \Sigma_p
\]
\[
\Sigma_p, C_S \vdash_S \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \sim S \tag{SHYP}
\]

\[
(t_1, t_2) \notin \Sigma_p \quad \emptyset \vdash_{M} \{(t_1 \leq t_2)\} \sim S
\]
\[
S' = \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_S \cap S
\]
\[
\forall C' \in S', \Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{MS} C' \sim S_{C'} \tag{SASSUM}
\]
\[
\forall, t_1, t_2 \nexists \{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C
\]
\[
\Sigma_p, C_S \vdash_S C \sim (C \cup C_S) \tag{SDONE}
\]

where \( \Sigma_p, C_S \vdash_{MS} C \sim S \) means that there exists \( C' \) such that \( \vdash_{M} C \sim C' \) and \( \Sigma_p, C_S \vdash_S C' \sim S \).

---

**Lemma E.17 (Soundness).** Let \( C \) be a normalized constraint-set. If \( \emptyset, \emptyset \vdash_{MS} C \sim S \), then for all normalized constraint-set \( C' \in S \) and all substitution \( \sigma \), we have \( \sigma \vdash C' \Rightarrow \sigma \vdash C \).

**Proof.** We prove the following statements.

- Assume \( \vdash_{M} C \sim C' \). For all \( \sigma \), if \( \sigma \vdash C' \), then \( \sigma \vdash C \).
- Assume \( \Sigma_p, C_S \vdash_{S} C \sim S \). For all \( \sigma \) and \( C_0 \in S \), if \( \sigma \vdash C_0 \), then \( \sigma \vdash C_S \cup C \).

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of \( \Sigma_p, C_S \vdash_{S} C \sim S \).

(\text{SHYP}) by induction, we have \( \sigma \vdash (C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\}) \cup C \), that is \( \sigma \vdash C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \).
(SASSUM): according to Definition E.4 $C_0 \in S_{C_1}$ for some $C' \in S'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{MS}} C' \rightsquigarrow S_{C'}$, we have $\sigma \models C'$. Moreover, the equation $S' = \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_S \cap \bar{S}$ gives us $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_S \subseteq C'$. Therefore, we have $\sigma \models C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C$.

(SDONE): straightforward.

Lemma E.18 (Completeness). Let $C$ be a normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow S$. Then for all substitution $\sigma$, if $\sigma \models C$, then there exists $C' \in S$ such that $\sigma \models C'$.

Proof. We prove the following statements.

- Assume $\vdash_M C \rightsquigarrow C'$. For all $\sigma$, if $\sigma \models C$, then $\sigma \models C'$.
- Assume $\Sigma_p, C_S \vdash S \rightsquigarrow S$. For all $\sigma$, if $\sigma \models C_S \cup C$, then there exists $C_0 \in S$ such that $\sigma \models C_0$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_S \vdash S \rightsquigarrow S$.

(SHYP): by induction, the result follows.

(SASSUM): as $\sigma \models C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C$), we have $\sigma \models \{(t_1 \leq t_2)\}$. As $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\}$, applying Lemma E.10 there exists $C_0 \in S$ such that $\sigma \models C_0$. Let $C' = C_S \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C_S'$. Clearly we have $\sigma \models C' \cap C' \in S'$. By induction on the premise $C_0 \in \bigcup_{C \in S} S_C$, there exists $C_0 \in S_{C'}$ such that $\sigma \models C_0$. Moreover, it is clear that $C_0 \subseteq C_{C'}$. Therefore, the result follows.

(SDONE): straightforward.

Lemma E.19 (Decidability). Let $C$ be a finite normalized constraint-set. Then $\emptyset, \emptyset \vdash_{\mathcal{MS}} C$ terminates.

Proof. Let $T$ be the set of type occurring in $C$. As $C$ is finite, $T$ is finite as well. Let $\mathfrak{B}$ be a plinth such that $T \subseteq \mathfrak{B}$. Then when we saturate a fresh constraint $(t_1, t_2)$ during the process of $\emptyset, \emptyset \vdash_{\mathcal{MS}} C$, $(t_1, t_2)$ would belong to $\mathfrak{B} \times \mathfrak{B}$. According to Lemma E.12 we know that $\emptyset \vdash_{\mathcal{N}} \{(t_1, t_2)\}$ terminates. Moreover, the termination of the merging of the lower bounds or the upper bounds of a same type variable is trivial. Finally, we have to prove termination of the saturation process. The proof proceeds by induction on $|\bigcup_{C \in S} S_C| - |\Sigma_p|, |C|$: $|C|$ decreases. $\Sigma_p$: as $(t_1, t_2) \notin \Sigma_p$ and $t_1, t_2 \in \mathfrak{B}$, $|\bigcup_{C \in S} S_C| - |\Sigma_p|$ decreases.

(SDone): it terminates immediately.

Definition E.20 (Sub-Constraint). Let $C_1, C_2 \in 2^{\mathfrak{B}}$ be two normalized constraint-sets. We say $C_1$ is a sub-constraint of $C_2$, denoted as $C_1 \ll C_2$, if for all $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ such that $t' c t$, where $c \in \{\leq, \geq, =\}$.

Lemma E.21. Let $C_1, C_2 \in 2^{\mathfrak{B}}$ be two normalized constraint-sets and $C_1 \ll C_2$. Then for all substitution $\sigma$, if $\sigma \models C_2$, then $\sigma \models C_1$.

Proof. Considering any constraint $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ and $t' c t$, where $c \in \{\leq, \geq, =\}$. Since $\sigma \models C_2$, then $\sigma(\alpha) c t' \sigma$. Moreover, as $t' c t$, we have $t' c t \sigma$. Thus $\sigma(\alpha) c t \sigma$.

Definition E.22. Let $C \in 2^{\mathfrak{B}}$ be a normalized constraint-set. We say $C$ is saturated if for each type variable $\alpha$,

1. there exists at most one form $(\alpha \geq t_1) \in C$.
2. there exists at most one form $(\alpha \leq t_2) \in C$.
3. if $(\alpha \geq t_1), (\alpha \leq t_2) \in C$, then $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow S$ and there exists $C' \in S$ such that $C' \ll C$.

Lemma E.23. Let $C$ be a finite normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow S$. Then for all normalized constraint set $C' \in S$, $C'$ is saturated.

Proof. We prove a stronger statement:

- assume $\Sigma_p, C_S \vdash_{\mathcal{MS}} C \rightsquigarrow S$. If (i) for all $(t_1, t_2) \in \Sigma_p$ there exists $C' \in (\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\})$ such that $C' \ll C_S \cup C$ and (ii) for all $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C_S$ the pair $(t_1, t_2)$ is in $\Sigma_p$, then $C_0$ is saturated for all $C_0 \in S$. 


The conditions (1) and (2) for a saturated constraint-set is straightforward for all $C \in S$. The proof of the condition (3) proceeds by induction on the derivation $\Sigma_p$, $C\in S \rightarrow \sigma$. 

**(SHYP):** as $(t_1, t_2) \in \Sigma_p$, the conditions (i) and (ii) hold for the premise. By induction, the result follows. 

**(SASSUMEP):** take any premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash C'' \rightarrow S_{C''}$, where $C' \in S'$ and $\vdash_{MS} C' \rightarrow C''$. For any $(s_1, s_2) \in \Sigma_p$, the condition (i) gives us that there exists $C_0 \in \{\emptyset \vdash_{MS} \{(s_1 \leq s_2)\}\}$ such that $C_0 \leq C_2 \cup \{\{\alpha \geq t_1\}, \{\alpha \leq t_2\}\} \cup C$). Since $S' = C_2 \cup \{\{\alpha \geq t_1\}, \{\alpha \leq t_2\}\} \cup C \cap S$, we have $C_0 \leq C''$. Moreover, consider $(t_1, t_2)$. As $\emptyset \vdash \{(t_1 \leq t_2)\} \rightarrow S$, there exists $C_0 \in S$ such that $C_0 \leq C''$. Thus the condition (i) holds for the premise. Moreover, the condition (ii) holds trivially for premise. By induction, the result follows. 

**(SDONE):** the result follows by the conditions (i) and (ii).

---

**Lemma E.24** (Finiteness). Let $C$ be a constraint-set and $\emptyset, \emptyset \vdash_{MS} C \rightarrow S$. Then $S$ is finite. 

**Proof.** It follows by Lemma E.13.

**Lemma E.25.** Let $C$ be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{MS} C \rightarrow S$. Then for all normalized constraint-set $C' \in S$, $C'$ is well-ordered.

**Proof.** The merging of the lower bounds (or the upper bounds) of a same type variable preserves the orders. The result of saturation is well-ordered by Lemma E.15.

Normalization and saturation may produce redundant constraint-sets. For example, consider the constraint-set $\{\{\alpha \times \beta\} \leq (\text{Int} \times \text{Bool})\}$. Applying the rule (NPROD), the normalization of this set is $\{\{\alpha \leq 0\}, \{\beta \leq 0\}, \{\alpha \leq 0, \beta \leq 0\}, \{\alpha \leq \text{Int}, \beta \leq \text{Bool}\}\}$. Clearly it is a saturated one. Note that $\{\{\alpha \leq 0\}, \{\beta , 0\}\}$ is redundant, since any solution to this constraint-set is a solution to $\{\{\alpha \leq 0\}\}$. Therefore, it is safe to eliminate it. Generally, for any two different normalized constraint sets $C_1, C_2 \in S$, if $C_1 \subseteq C_2$, then according to Lemma E.21, any solution to $C_2$ is a solution to $C_1$. Therefore, $C_2$ can be eliminated from $S$.

**Definition E.26.** Let $S$ be a set of normalized constraint-sets. We say $S$ is minimal if for any two different normalized constraint-sets $C_1, C_2 \in S$, neither $C_1 \subseteq C_2$ nor $C_2 \subseteq C_1$. Moreover, we say $S \simeq S'$ if for all substitution $\sigma$ such that $\exists C \in S$, $\sigma \models C \iff \exists C' \in S', \sigma \models C'$. 

**Lemma E.27.** Let $C$ be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{MS} C \rightarrow S$. Then there exists a minimal set $S_0$ such that $S_0 \simeq S$. 

**Proof.** By eliminating the redundant constraint-sets in $S$. 

---

**E.1.3 From Constraints to Equations**

In this section, we transform a saturated constraint-set into an equivalent equation system. This shows that the type tallying problem is essentially a unification problem.

**Definition E.28** (Equations System). An equation system $E$ is a set of equations of the form $\alpha = t$ such that there exists at most one equation in $E$ for every type variable $\alpha$. We define the domain of $E$, written $\text{dom}(E)$, as the set $\{\alpha \mid \exists t \in E, \alpha = t\}$. 

**Definition E.29** (Equations System Solution). Let $E$ be an equation system. A solution to $E$ is a substitution $\sigma$ such that $\forall \alpha = t \in E, \sigma(\alpha) \simeq t \sigma$ holds. 

If $\sigma$ is a solution to $E$, we write $\sigma \models E$.

From a normalized constraint-set $C$, we obtain some explicit conditions for the substitution $\sigma$ we want to construct from $C$. For instance, from the constraint $\alpha \leq t$ (resp. $\alpha \geq t$), we know that the type substituted for $\alpha$ must be a subtype of $t$ (resp. a super type of $t$).

We assume that each type variable $\alpha \in \text{dom}(C)$ has a lower bound $t_1$ and an upper bound $t_2$ using, if necessary, the fact that $0 \leq \alpha \leq 1$. Formally, we rewrite $C$ as follows.

\[
\begin{cases}
    t_1 \leq \alpha \leq 1 & \text{if only } \alpha \geq t_1 \in C \\
    0 \leq \alpha \leq t_2 & \text{if only } \alpha \leq t_2 \in C \\
    t_1 \leq \alpha \leq t_2 & \text{if } \alpha \geq t_1, \alpha \leq t_2 \in C
\end{cases}
\]

We then transform each constraint $t_1 \leq \alpha \leq t_2$ in $C$ into an equation $\alpha = (t_1 \lor \alpha') \land t_2$ where $\alpha'$ is a new type variable. The type $(t_1 \lor \alpha') \land t_2$ ranges from $t_1$ to $t_2$, so the equation $\alpha = (t_1 \lor \alpha') \land t_2$ expresses the fact that $t_1 \leq \alpha \leq t_2$, as wished. We prove the soundness and completeness of this transformation.

---

82 2012/7/11
Lemma E.30 (Soundness). Let $C \subseteq C$ be a saturated normalized constraint-set and $E$ its transformed equation system. Then for all substitution $\sigma$, if $\sigma \vdash E$ then $\sigma \vdash C$.

Proof. Without loss of generality, we assume that each type variable $\alpha \in \text{dom}(C)$ has a lower bound and an upper bound, that is $t_1 \leq \alpha \leq t_2$. We prove a stronger statement: for all $\sigma$ and $C_\Sigma \subseteq C$, if $\sigma \vdash E, \sigma \vdash C_\Sigma$ and $C^{' \Sigma} \cup C_\Sigma = C$, then $\sigma \vdash C'$. Here $C_\Sigma$ denotes the set of constraints that have been checked. The proof proceeds by induction on $(|C \setminus C_\Sigma|, |C'|)$.

$C' = \emptyset$: trivially.

$C' \subseteq \{(t_1 \leq \alpha \leq t_2)\} \cup C^{' \Sigma}$: if $(t_1 \leq \alpha \leq t_2) \in C_\Sigma$, that is $(t_1 \leq \alpha \leq t_2)$ has been checked before, then clearly $\sigma \vdash \{(t_1 \leq \alpha \leq t_2)\}$. By induction, we also have $\sigma \vdash C^{' \Sigma}$. So $\sigma \vdash \{(t_1 \leq \alpha \leq t_2)\} \cup C^{' \Sigma}$.

Otherwise, there exists a corresponding equation $\alpha = (t_1 \lor \alpha') \land t_2 \in E$. As $\sigma \vdash E$, we have $\sigma(\alpha) \simeq ((t_1 \lor \alpha') \land t_2)\sigma$. Then,

$$\sigma(\alpha) \land \neg t_2\sigma \simeq ((t_1 \lor \alpha') \land t_2)\sigma \land \neg t_2\sigma \simeq 0$$

Therefore, $\sigma(\alpha) \leq t_2\sigma$.

Consider the constraint $(t_1 \leq \alpha)$. We have

$$t_1\sigma \land \neg \sigma(\alpha) \simeq t_1\sigma \land \neg ((t_1 \lor \alpha') \land t_2)\sigma \simeq t_1\sigma \land \neg t_2\sigma$$

Now we assume that the constraints for $\alpha$ have been checked and we do not want to check them again, so we put them into the set $C_\Sigma$, that is $C_\Sigma = C_\Sigma' \cup \{(t_1 \leq \alpha \leq t_2)\}$. The remaining is to check the subtyping relation $t_1\sigma \land \neg t_2\sigma \leq 0$, that is, to check the judgement $\sigma \vdash \{(t_1 \leq t_2)\}$.

Since the whole constraint-set $C_\Sigma \cup C_\Sigma'$ is saturated, according to Definition E.22 we have $\emptyset \vdash \{(t_1 \leq t_2)\} \equiv \Sigma$ and there exists $C_\delta \in \Sigma$ such that $C_\delta \subseteq C_\Sigma \cup C_\Sigma'$. By assumption, we have $\sigma \vdash C_\delta$. By induction, we also have $\sigma \vdash C_\Sigma'$. So we get $\sigma \vdash C_\Sigma \cup C_\Sigma'$. Applying Lemma E.21 we have $\sigma \vdash C_\delta$. Finally, by Lemma E.9 we get $\sigma \vdash \{(t_1 \leq t_2)\}$. Therefore, the result follows.

Lemma E.31 (Completeness). Let $C \subseteq C$ be a saturated normalized constraint-set and $E$ its transformed equations system. Then for all substitution $\sigma$, if $\sigma \vdash C$ then there exists $\sigma'$ such that $\sigma' \not\leq \sigma$ and $\sigma \vdash \sigma' \vdash E$.

Proof. Let $\sigma' = \{\sigma(\alpha)/\alpha' | \alpha \in \text{dom}(C)\}$. Consider each equation $\alpha = (t_1 \lor \alpha') \land t_2 \in E$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. As $\sigma \vdash C$, then $t_1\sigma \leq \sigma(\alpha)$ and $\sigma(\alpha) \leq t_2\sigma$. Thus

$$\begin{align*}
(t_1 \lor \alpha') \land t_2)(\sigma \cup \sigma') &= (t_1(\sigma \cup \sigma') \lor \alpha'(\sigma \cup \sigma')) \land t_2(\sigma \cup \sigma') \\
&= (t_1\sigma \lor \sigma(\alpha)) \land t_2\sigma \\
&\simeq \sigma(\alpha) \land t_2\sigma \quad (t_1\sigma \leq \sigma(\alpha)) \\
&\simeq \sigma(\alpha) \land (\sigma(\alpha) \leq t_2\sigma) \\
&= (\sigma \cup \sigma')(\alpha)
\end{align*}$$

Definition E.32. Let $E$ be an equations system and $O$ an order on $\text{dom}(E)$. We say $E$ is well-ordered if for all equation $\alpha = t_\alpha \in E$, $O(\alpha) < O(\beta)$ for all $\beta \in \text{tv}(t_\alpha) \cap \text{dom}(E)$.

Lemma E.33. Let $C$ be a well-ordered saturated normalized constraint-set and $E$ its transformed equations system. Then $E$ is well-ordered.

Proof. Clearly, $\text{dom}(E) = \text{dom}(C)$. Consider an equation $\alpha = (t_1 \lor \alpha') \land t_2$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. By Definition E.14, for all $\beta \in (\text{tv}(t_1) \cup \text{tv}(t_2)) \cap \text{dom}(C)$, $O(\alpha) < O(\beta)$. Moreover, $\alpha'$ is a fresh type variable in $C$, that is $\alpha' \not\in \text{dom}(C)$. And then $\alpha' \not\in \text{dom}(E)$. Therefore, $\text{tv}(t_1 \lor \alpha') \land t_2 \land \text{dom}(E) = (\text{tv}(t_1) \cup \text{tv}(t_2)) \cap \text{dom}(C)$. Thus the result follows.

E.1.4 Equations Systems Solving

We now extract a solution (i.e., a substitution) from the equations system we build from $C$. In an equation $\alpha = t_\alpha$, $\alpha$ may also appear in the type $t_\alpha$; such an equality reminds the definition of a recursive type. As a first step, we introduce a recursion operator in all the equations of the system, transforming $\alpha = t_\alpha$ into $\alpha = \mu \alpha. t_\alpha \{x_\alpha|\alpha\}$. This ensures that type variables do not appear in the right-hand side of the equalities, making the whole solving process easier. If some recursion operators are in fact not needed in the solution we obtain (that is, we have $\alpha = \mu \alpha. t_\alpha$ with $x_\alpha \not\in \text{fv}(t_\alpha)$), then we can simply eliminate them.

Let $(\alpha = \mu \alpha. t_\alpha) \cup E$ be an equations system such that $E$ contains only equations closed with the recursion operator $\mu$, as explained above. The next step is to substitute the content expression $\mu x_\alpha. t_\alpha$ for all
the occurrences of $\alpha$ in equations in $E$, yielding a new equations system $E'$. Let $\beta = \mu x. t_\beta \in E$; because $t_\alpha$ may contain $\beta$, we in fact replace the equation $\beta = \mu x. t_\beta$ with $\beta = \mu x. t_\beta \{\mu x. t_\alpha \{x_\alpha / \alpha\} \{x_\beta / \beta\}\}$. Finally, assume that the equations system $E'$ has a solution $\sigma'$. Then the substitution $\{t_\alpha \sigma / \alpha\} \oplus \sigma'$ is a solution to the equations system $\{\alpha = \mu x_. t_\alpha\} \cup E$. The solving algorithm Unify() is defined in Figure 14.

**Define E.34 (General Solution).** Let $E$ be an equations system. A general solution to $E$ is a substitution $\sigma$ from $\text{dom}(E)$ to $T$ such that

$$\forall \alpha \in \text{dom}(\sigma) . \text{var}(\sigma(\alpha)) \cap \text{dom}(\sigma) = \emptyset$$

and

$$\forall \alpha = t \in E . \sigma(\alpha) \simeq t \sigma$$

**Lemma E.35.** Let $E$ be an equations system. If $\sigma = \text{Unify}(E)$, then $\forall \alpha \in \text{dom}(\sigma), \text{var}(\sigma(\alpha)) \cap \text{dom}(\sigma) = \emptyset$ and $\text{dom}(\sigma) = \text{dom}(E)$.

**Proof.** The algorithm Unify() consists of two steps: (i) to transform types into recursive types and (ii) to extract the substitution. After the first step, for each equation $(\alpha = t_\alpha) \in E$, we have $\alpha \notin \text{var}(t_\alpha)$. Consider the second step. Let $\text{var}(E) = \bigcup_{(\alpha = t_\alpha) \in E} \text{var}(t_\alpha)$ and $S = V \setminus S$ where $S$ is a set of type variables. We prove a stronger statement: $\forall \alpha \in \text{dom}(\sigma), \text{var}(\sigma(\alpha)) \cap (\text{dom}(\sigma) \cup \overline{\text{var}(E)}) = \emptyset$ and $\text{dom}(\sigma) = \text{dom}(E)$. The proof proceeds by induction on $E$:

- **Base Case:** $E = \emptyset$: trivially.

- **Inductive Step:** $E = \{ (\alpha = t_\alpha) \} \cup E'$: let $E'' = \{ (\beta = t_\beta \{t_\alpha / \alpha\}\{x_\beta / \beta\}) \mid (\beta = t_\beta) \in E' \}$. Then there exists a substitution $\sigma''$ such that $\sigma'' = \text{Unify}(E'')$ and $\sigma = \{t_\alpha \sigma'' / \alpha\} \oplus \sigma''$. By induction, we have $\forall \beta \in \text{dom}(\sigma''), \text{var}(\sigma''(\beta)) \cap (\text{dom}(\sigma'') \cup \text{var}(E'')) = \emptyset$ and $\text{dom}(\sigma'') = \text{dom}(E'')$. As $\alpha \notin \text{dom}(E'')$, we have $\alpha \notin \text{dom}(\sigma'')$, and then $\text{dom}(\sigma) = \text{dom}(\sigma'') \cup \{\alpha\} = \text{dom}(E)$.

Moreover, $\alpha \notin \text{var}(E'')$, then $\text{dom}(\sigma) \subset \text{dom}(\sigma'') \cup \text{var}(E'')$. Thus, for all $\beta \in \text{dom}(\sigma'')$, we have $\text{var}(\sigma''(\beta)) \cap \text{dom}(\sigma) = \emptyset$. Consider $t_\alpha \sigma''$. It is clear that $\text{var}(t_\alpha \sigma'') \cap \text{dom}(\sigma) = \emptyset$. Besides, the algorithm does not introduce any fresh variable, then for all $\beta \in \text{dom}(\sigma)$, we have $\text{var}(t_\beta) \cap \text{var}(E) = \emptyset$. Therefore, the result follows.

**Lemma E.36 (Soundness).** Let $E$ be an equations system. If $\sigma = \text{Unify}(E)$, then $\sigma \vdash E$.

**Proof.** By induction on $E$.

- **Base Case:** $E = \emptyset$: trivially.

- **Inductive Step:** $E = \{ (\alpha = t_\alpha) \} \cup E'$: let $E'' = \{ (\beta = t_\beta \{t_\alpha / \alpha\}\{x_\beta / \beta\}) \mid (\beta = t_\beta) \in E' \}$. Then there exists a substitution $\sigma''$ such that $\sigma'' = \text{Unify}(E'')$ and $\sigma = \{t_\alpha \sigma'' / \alpha\} \oplus \sigma''$. By induction, we have $\sigma'' \vdash E''$. According to Lemma E.35 we have $\text{dom}(\sigma'') = \text{dom}(E'')$. So $\text{dom}(\sigma) = \text{dom}(\sigma'') \cup \{\alpha\}$.
Considering any equation \((\beta = t_\beta) \in E\) where \(\beta \in \text{dom}(\sigma'')\). Then

\[
\sigma(\beta) = \sigma''(\beta) \quad (\text{apply } \sigma)
\]
\[
\simeq t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \quad (\text{as } \sigma'' \vdash E'')
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma''
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \oplus \sigma'\quad (\text{expand } x_\beta)
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \oplus \sigma''
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \oplus \sigma''
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma''
\]

Finally, consider the equation \((\alpha = t_\alpha)\). As

\[
\sigma(\alpha) = t_\alpha \sigma'' \quad (\text{apply } \sigma)
\]
\[
= t_\alpha \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \quad (\text{expand } \sigma'')
\]
\[
= t_\alpha \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \quad (\text{as } \beta \sigma = \beta''\sigma'')
\]
\[
= t_\alpha \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \quad (\text{as } \alpha \notin \text{var}(t_\alpha))
\]
\[
= t_\alpha \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma'' \quad (\text{as } \text{dom}(\sigma) = \text{dom}(\sigma'') \cup \{\alpha\})
\]
\[
= t_\alpha \sigma
\]

Thus, the result follows.

\[
\square
\]

Lemma E.37. Let \(E\) be an equations system. If \(\sigma = \text{Unify}(E)\), then \(\sigma\) is a general solution to \(E\).

\[
\square
\]

\[
\text{Proof. } \text{Immediate consequence of Lemmas } E.35 \text{ and } E.36
\]

Clearly, given an equations system \(E\), the algorithm \(\text{Unify}(E)\) terminates with a substitution \(\sigma\).

Lemma E.38. Given an equations system \(E\), the algorithm \(\text{Unify}(E)\) terminates with a substitution \(\sigma\).

\[
\square
\]

\[
\text{Proof. } \text{By induction on the number of equations in } E.
\]

Definition E.39. Let \(\sigma, \sigma'\) be two substitutions. We say \(\sigma \simeq \sigma'\) if and only if \(\forall \alpha. \sigma(\alpha) \simeq \sigma'(\alpha)\).

Lemma E.40 (Completeness). Let \(E\) be an equations system. For all substitution \(\sigma\), if \(\sigma \vdash E\), then there exist \(\sigma_0\) and \(\sigma'\) such that \(\sigma_0 = \text{Unify}(E)\) and \(\sigma \simeq \sigma_0 \sigma'\).

\[
\square
\]

\[
\text{Proof. } \text{According to Lemma } E.38 \text{ there exists } \sigma_0 \text{ such that } \sigma_0 = \text{Unify}(E). \text{ For any } \alpha \notin \text{dom}(\sigma_0), \text{ clearly we have } \alpha \sigma_0 \sigma = \alpha \sigma \text{ and then } \alpha \sigma_0 \sigma \simeq \sigma. \text{ What remains to prove is that if } \sigma \vdash E \text{ and } \sigma_0 = \text{Unify}(E) \text{ then } \forall \alpha \in \text{dom}(\sigma_0), \alpha \sigma_0 \sigma \simeq \alpha \sigma. \text{ The proof proceeds by induction on } E:
\]

\[
E = \emptyset; \text{ trivially.}
\]
\[
E = \{(\alpha = t_\alpha)\} \cup E': \text{ let } E'' = \{(\beta = t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta}) \mid (\beta = t_\beta) \in E'\}. \text{ Then there exists a substitution } \sigma'' \text{ such that } \sigma'' = \text{Unify}(E'') \text{ and } \sigma_0 = \{t_\alpha \sigma''/\alpha\} \oplus \sigma''. \text{ Considering each equation } (\beta = t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta}) \in E'', \text{ we have}
\]

\[
\sigma(\beta) = t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma \oplus \sigma'\quad (\text{expand } x_\beta)
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma \oplus \sigma''
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma \oplus \beta \sigma
\]
\[
= t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma\quad (\text{as } \sigma \vdash E)
\]
\[
\simeq t_\beta \{t_\alpha/\alpha\}_{x_\beta/\beta} \sigma
\]
\[
\simeq \beta \sigma
\]

Therefore, \(\sigma \vdash E''\). By induction on \(E''\), we have \(\forall \beta \in \text{dom}(\sigma'')\). \(\beta \sigma'' \sigma \simeq \beta \sigma\). According to Lemma E.35 \(\text{dom}(\sigma'') = \text{dom}(E'')\). As \(\alpha \notin \text{dom}(E'')\), then \(\text{dom}(\sigma_0) = \text{dom}(\sigma'') \cup \{\alpha\}\). Therefore for any
$$\beta \in \text{dom}(\sigma^\prime) \cap \text{dom}(\sigma_0), \beta \sigma_0 \sigma \approx \beta \sigma^\prime \sigma \approx \beta \sigma. \text{ Finally, considering } \alpha, \text{ we have}$$

\[
\begin{align*}
\alpha \sigma_0 \sigma &= t_\alpha \sigma^\prime \sigma / \beta \mid \beta \in \text{dom}(\sigma^\prime) \{ \sigma \} & (\text{apply } \sigma_0) \\
&= t_\alpha \{ \beta \sigma^\prime / \beta \mid \beta \in \text{dom}(\sigma^\prime') \} \oplus \sigma & (\text{expand } \sigma^\prime') \\
&= t_\alpha \{ \beta \sigma / \beta \mid \beta \in \text{dom}(\sigma) \} \oplus \sigma & (\text{as } \forall \beta \in \sigma^\prime, \beta \sigma \approx \beta \sigma^\prime \sigma) \\
&= t_\alpha \sigma & (\text{as } \sigma \vdash E)
\end{align*}
\]

Therefore, the result follows.

In our calculus, a type is well-formed if and only if the recursion traverses a constructor. In other words, the recursive variable should not appear at the top level of the recursive content. For example, the type $\mu x. x \lor t$ is not well-formed. To make the substitutions usable, we should avoid these substitutions with ill-formed types. Fortunately, this can been done by giving an order on the domain of an equations system to make sure that the equations system is well-ordered.

**Lemma E.41.** Let $E$ be a well-ordered equations system. If $\sigma = \text{Unify}(E)$, then for all $\alpha \in \text{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.

**Proof.** Assume that there exists an ill-formed $\sigma(\alpha)$. That is, $\sigma(\alpha) = \mu x. t$ where $x$ occurs at the top level of $t$. According to the algorithm $\text{Unify}()$, there exists a sequence of equations $(\alpha =)\alpha_0 = t_{\alpha_0}, \alpha_1 = t_{\alpha_1}, \alpha_n = t_{\alpha_n}$ such that $\alpha_i \in \text{tv}(t_{\alpha_{i-1}})$ and $\alpha_0 \in \text{tv}(t_{\alpha_n})$ where $i \in \{1, \ldots, n\}$ and $n \geq 0$. According to Definition E.32, $O(\alpha_{i-1}) < O(\alpha_i)$ and $O(\alpha_n) < O(\alpha_0)$. Therefore, we have $O(\alpha_0) < O(\alpha_1) < \ldots < O(\alpha_n) < O(\alpha_0)$, which is impossible. Thus the result follows.

As mentioned above, there may be some useless recursion constructor $\mu$. They can be eliminated by checking whether the recursive variable appears in the content expression or not. Moreover, if a recursive type is empty (which can be checked with the subtyping algorithm), then it can be replaced by $0$.

To conclude, we now describe the solving procedure $\text{Sol}_{\Delta}(C)$ for the type tallying problem $C$. We first normalize $C$ into a finite set $S$ of normalized constraint-sets. If $S$ is empty, then there are no solutions to $C$. Otherwise, each normalized constraint-set $C_i \in S$ is merged and saturated into a finite set $S_{C_i}$ of saturated constraint-sets. Then all these sets are collected into another set $S'$ (i.e., $S' = \bigcup C_i \in S S_{C_i}$). If $S'$ is empty, then there are no solutions to $C$. Otherwise, for each constraint-set $C'_i \in S'$, we transform $C'_i$ into an equation system $E_i$ and then construct a general solution $\sigma_i$ from $E_i$. Finally, we collect all the solution $\sigma_i$, yielding a set $\Theta$ of solutions to $C$. We write $\text{Sol}_{\Delta}(C) \rightsquigarrow \Theta$ if $\text{Sol}_{\Delta}(C)$ terminates with $\Theta$, and we call $\Theta$ the solution of the type tallying problem $C$.

**Theorem E.42 (Soundness).** Let $C$ be a constraint-set. If $\text{Sol}_{\Delta}(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \vdash C$.

**Proof.** Consequence of Lemmas E.9, E.17, E.23, E.30 and E.36.

**Theorem E.43 (Completeness).** Let $C$ be a constraint-set and $\text{Sol}_{\Delta}(C) \rightsquigarrow \Theta$. Then for all substitution $\sigma$, if $\sigma \vdash C$, then there exists $\sigma' \in \Theta$ and $\sigma''$ such that $\sigma \approx \sigma' \circ \sigma''$.

**Proof.** Consequence of Lemmas E.10, E.18, E.31 and E.40.

**Theorem E.44 (Decidability).** Let $C$ be a constraint-set. Then $\text{Sol}_{\Delta}(C)$ terminates.

**Proof.** Consequence of Lemmas E.12, E.19 and E.38.

**Lemma E.45.** Let $C$ be a constraint-set and $\text{Sol}_{\Delta}(C) \rightsquigarrow \Theta$. Then

1. $\Theta$ is finite.
2. for all $\sigma \in \Theta$ and for all $\alpha \in \text{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.

**Proof.**


According to Lemmas C.12 and C.13, we can reduce these two inequalities to the constraint set
\[ \bigwedge_{i \in I} t \sigma_i \leq 0 \rightarrow \exists \gamma \]
where \( \gamma \).

Proof. Let \( \sigma \) be a substitution such that \( t \sigma \) is a type variable such that \( \gamma \). As \( t \sigma \) is a type variable such that \( \gamma \), then there exists \( \exists s \sigma \) for all \( s \sigma \) such that \( t \sigma \). Then by Lemma C.13, we can reduce these two inequalities to the constraint set \( C = \{(t \sigma, \exists s \sigma, \gamma)\} \)

where \( \gamma \) is a fresh type variable. We have reduced the original application problem \( t \bullet \sigma \) to the solving of \( C \), which can be done following Section E.1. We write \( \text{AppFix}_{\Delta}(t, s) \) for the set of substitutions obtained by solving the constraint-set corresponding to the application problem (with fixed cardinalities) \( t \bullet \sigma \).

\[ \begin{align*}
\text{Lemma E.46.} & \quad \text{Let } t, s \text{ be two types and } \gamma \text{ a type variable such that } \gamma \notin \text{var}(t) \cup \text{var}(s). \text{ Then for all substitution } \sigma, \text{ if } \sigma \leq \sigma \rightarrow \gamma \sigma, \text{ then } \sigma \leq \text{dom}(t \sigma) \text{ and } \gamma \sigma \geq t \sigma \cdot \sigma. \\
\text{Proof.} & \quad \text{Consider any substitution } \sigma. \text{ As } t \sigma \leq \sigma \rightarrow \gamma \sigma \text{ by Lemma C.12, we have } \sigma \leq \text{dom}(t \sigma). \text{ Then by Lemma C.13, we get } \sigma(\gamma) \geq t \sigma \cdot \sigma. \qed
\end{align*} \]

\[ \begin{align*}
\text{Lemma E.47.} & \quad \text{Let } t, s \text{ be two types and } \gamma \text{ a type variable such that } \gamma \notin \text{var}(t) \cup \text{var}(s). \text{ Then for all substitution } \sigma, \text{ if } \sigma \leq \text{dom}(t \sigma) \text{ and } \gamma \notin \text{dom}(\sigma), \text{ then there exists } \sigma' \text{ such that } \sigma' \neq \sigma \text{ and } t(\sigma \cup \sigma') \leq (s \rightarrow \gamma)(\sigma \cup \sigma'). \\
\text{Proof.} & \quad \text{Consider any substitution } \sigma. \text{ As } s \sigma \leq \text{dom}(t \sigma), \text{ by Lemma C.13, the type } (t \sigma) \cdot (s \sigma) \text{ exists and } t \sigma \leq \sigma \rightarrow ((t \sigma) \cdot (s \sigma)). \text{ Let } \sigma' = \{(t \sigma) \cdot (s \sigma)\}. \text{ Then}
\end{align*} \]

\[ \begin{align*}
& t(\sigma \cup \sigma') \\
& = t \sigma \\
& \leq \sigma \rightarrow ((t \sigma) \cdot (s \sigma)) \\
& = \sigma \rightarrow \gamma \sigma' \\
& = (s \rightarrow \gamma)(\sigma \cup \sigma')
\end{align*} \]

\[ \text{Indeed, the first constraint } (t', \leq 0 \rightarrow \exists) \text{ can be eliminated since it is implied by the second one.} \]
Note that the solution of the $\gamma$ introduced in the constraint $(t, s \leq \gamma)$ represents a result type for the application of $t$ to $s$. In particular, completeness for the tallying problem ensures that each solution will assign to $\gamma$ (which occurs in a covariant position) the minimum type for that solution. So the minimum solutions for $\gamma$ are in $t \ast s$ (see the substitution $\sigma'(:gamma) = (t\sigma) \cdot (s\sigma)$) in the proof of Lemma E.47.

**Theorem E.48** (Soundness). Let $t$ and $s$ be two types. Then for all $\sigma \in \text{AppFix}_\Delta(t,s)$, we have $t\sigma \leq s \rightarrow$ and $s\sigma \leq \text{dom}(t\sigma)$.

**Proof.** Consequence of Lemmas E.46 and E.42.

**Theorem E.49** (Completeness). Let $t$ and $s$ be two types. For all substitution $\sigma$, if $t\sigma \leq 0 \rightarrow$ and $s\sigma \leq \text{dom}(t\sigma)$, then there exists $\sigma' \in \text{AppFix}_\Delta(t,s)$ such that $\sigma \simeq \sigma' \circ \sigma''$.

**Proof.** Consequence of Lemmas E.47 and E.43.

### E.2.2 General application problem

Now we take the cardinalities of $I$ and $J$ into account to solve the general application problem. As stated before, we start with $I$ and $J$ both of cardinality 1 and explore all the possible combinations of the cardinalities of $I$ and $J$ by, say, a dove-tail order until we get a solution. More precisely, the algorithm consists of two steps:

**Step A:** we generate a constraint set as explained in Subsection E.2.1 and apply the algorithm described in Subsection E.1, yielding either a solution or a failure.

**Step B:** if all attempts to solve the constraint sets have failed at Step I of the algorithm of Subsection E.1, then fail (the expression is not typable). If they all failed but at least one did not fail in Step I, then increment the cardinalities $I$ and $J$ to their successor in the dove-tail order and start from Step A again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if the solving of the constraint-set fails at Step I of the algorithm for the tallying problem. The reason is that there is no type variable in $I$ and $J$ so that the procedure may not terminate, which makes it only a semi-algorithm. The following lemma justifies why we do not try to expand if normalization (i.e., Step 1 of the tallying algorithm) fails.

**Lemma E.50.** Let $t$, $s$ be two types, $\gamma$ a fresh type variable and $\{\rho_i\}_{i \in I}$, $\{\rho_j\}_{j \in J}$ two sets of general renamings. Then if $\emptyset \vdash_\mathcal{N} \{ (t, \leq 0 \rightarrow s), (t, \leq s \rightarrow \gamma) \} \rightsquigarrow \emptyset$, then $\emptyset \vdash_\mathcal{N} \{ (\bigwedge_{i \in I} \rho_i, \leq 0 \rightarrow \emptyset), (\bigwedge_{i \in I} \rho_i, \leq \emptyset, (\bigwedge_{j \in J} \rho_j) \rightarrow \gamma) \} \rightsquigarrow \emptyset$.

**Proof.** As $\emptyset \vdash_\mathcal{N} \{ (t, \leq 0 \rightarrow \emptyset), (t, \leq s \rightarrow \gamma) \} \rightsquigarrow \emptyset$, then either $\emptyset \vdash_\mathcal{N} \{ (t, \leq 0 \rightarrow \emptyset) \} \rightsquigarrow \emptyset$ or $\emptyset \vdash_\mathcal{N} \{ (t, \leq s \rightarrow \gamma) \} \rightsquigarrow \emptyset$. If the first one holds, then according to Lemma E.16 we have $\emptyset \vdash_\mathcal{N} \{ (\bigwedge_{i \in I} \rho_i, \leq 0 \rightarrow \emptyset) \} \rightsquigarrow \emptyset$, and a fortiori $\emptyset \vdash_\mathcal{N} \{ (\bigwedge_{i \in I} \rho_i, \leq 0 \rightarrow \emptyset), (\bigwedge_{i \in I} \rho_i, \leq \emptyset, (\bigwedge_{j \in J} \rho_j) \rightarrow \gamma) \} \rightsquigarrow \emptyset$.

Assume that $\emptyset \vdash_\mathcal{N} \{ (t, \leq s \rightarrow \gamma) \} \rightsquigarrow \emptyset$. Without loss of generality, we consider the disjunctive normal form $\tau$ of $\tau$:

$$\tau = \bigvee_{k_b \in K_b} \tau_{b_k} \lor \bigvee_{k_p \in K_p} \tau_{b_p} \lor \bigvee_{k_a \in K_a} \tau_{b_a}$$

where $\tau_{b_k}$ ($\tau_{b_p}$ and $\tau_{b_a}$ resp.) is an intersection of basic types (products and arrows resp.) and type variables. Then there must exist $k \in K_b \cup K_p \cup K_a$ such that $\emptyset \vdash_\mathcal{N} \{ (\tau_k, \leq 0 \rightarrow \emptyset) \} \rightsquigarrow \emptyset$. If $k \in K_b \cup K_p$, then the constraint $(\tau_k, \leq 0 \rightarrow \emptyset)$ is equivalent to $(\tau_k, \leq 0)$. By Lemma E.16 we get
\[ \emptyset \vdash_N (\forall_{i \in I} \tau_k \rho_i, \leq, 0) \Rightarrow \emptyset, \text{ that is, } \emptyset \vdash_N (\forall_{i \in I} \tau_k \rho_i, \leq, (\bigwedge_{j \in J} s_{\rho_j}) \rightarrow \gamma) \Rightarrow \emptyset. \] So the result follows.

Otherwise, it must be that \( k \in K_a \) and \( \tau_k = \bigwedge_{p \in P} (w_p \rightarrow v_p) \land \bigwedge_{n \in N} (w_n \rightarrow v_n). \) Clearly, we have \( \emptyset \vdash_N (\tau_k, \leq, 0) \Rightarrow \emptyset \) (otherwise, \( \emptyset \vdash_N (\tau_k, \leq, s \rightarrow \gamma) \Rightarrow \emptyset \) does not hold). Applying Lemma E.16 again, we get \( \emptyset \vdash_N (\forall_{i \in I} \tau_k \rho_i, \leq, 0) \Rightarrow \emptyset. \) Moreover, following the rule \((\text{Narrow})\), there exists a set \( P' \subseteq P \) such that

\[
\begin{align*}
\emptyset \vdash_N \left\{ \bigwedge_{p \in P'} \neg w_p \land s, \leq, 0 \right\} \Rightarrow \emptyset \\
\left\{ \emptyset \vdash_N \left\{ \bigwedge_{p \in P \setminus P'} v_p \land \neg \gamma, \leq, 0 \right\} \Rightarrow \emptyset \right. \\
P' = P \text{ or } \emptyset \vdash_N \left\{ \bigwedge_{p \in P \setminus P'} v_p \land \neg \gamma, \leq, 0 \right\} \Rightarrow \emptyset
\end{align*}
\]

Applying E.16 we get

\[
\begin{align*}
\emptyset \vdash_N \left\{ \bigwedge_{p \in P'} (\bigwedge_{i \in I} (w_p \rightarrow v_p)) \rho_i, \leq, (\bigwedge_{j \in J} s_{\rho_j}) \rightarrow \gamma \right\} \Rightarrow \emptyset \\
P' = P \text{ or } \emptyset \vdash_N \left\{ \bigwedge_{p \in P \setminus P'} v_p \land \neg \gamma, \leq, 0 \right\} \Rightarrow \emptyset
\end{align*}
\]

By the rule \((\text{Narrow})\), we have

\[
\emptyset \vdash_N (\forall_{i \in I} \tau_k \rho_i, \leq, (\bigwedge_{j \in J} s_{\rho_j}) \rightarrow \gamma) \Rightarrow \emptyset.
\]

Therefore, we have \( \emptyset \vdash_N (\forall_{i \in I} \tau_k \rho_i, \leq, (\bigwedge_{j \in J} s_{\rho_j}) \rightarrow \gamma) \Rightarrow \emptyset. \) So the result follows.

\( \square \)

Let \( \text{App}_\Delta(t, s) \) denote the semi-algorithm for the general application problem.

Theorem E.51. Let \( t, s \) be two types and \( \gamma \) the special fresh type variable introduced in \( (\forall_{i \in I} t_{\sigma_i}, \leq, (\bigwedge_{j \in J} s_{\sigma_j}) \rightarrow \gamma) \). If \( \text{App}_\Delta(t, s) \) terminates with \( \Theta \), then

1. if \( \Theta \neq \emptyset \), then for each \( \sigma \in \Theta \), \( \sigma(\gamma) \in t \, \bullet_\Delta \, s. \)
2. if \( \Theta = \emptyset \), then \( t \, \bullet_\Delta \, s = \emptyset. \)

Proof. (1): consequence of Theorem E.48 and Lemma E.46
(2): consequence of Lemma E.50

Consider the application map even, whose types are

\[
\text{map} :: \ (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\
\text{even} :: \ (\text{Int} \rightarrow \text{Bool}) \land ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}))
\]

Then we start with the constraint-set

\[
C_1 = \{(\alpha_1 \rightarrow \beta_1) \rightarrow [\alpha_1] \rightarrow [\beta_1] \leq ((\text{Int} \rightarrow \text{Bool}) \land ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}))) \rightarrow \gamma\}
\]

where \( \gamma \) is a fresh type variable (and where we \( \alpha \)-converted the type of \text{map}). Then the algorithm \text{Sol}_\Delta(C_1) \ generates a set of eight constraint-sets at Step 2:

\[
\begin{align*}
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0 \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \alpha \setminus \text{Int} \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \lor (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \land (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \lor (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \land (\alpha \setminus \text{Int}) \} \\
\{ \gamma \geq [\alpha_1] \rightarrow [\beta_1], \alpha_1 \leq 0, \beta_1 \geq \text{Bool} \lor (\alpha \setminus \text{Int}) \}
\end{align*}
\]

Clearly, the solutions to the 2nd-5th constraint-sets are included in those to the first constraint-set. For the other four constraint-sets, by minimum instantiation, we can get four solutions for \( \gamma \) (i.e., the result types of \text{map}: \( [\beta] \rightarrow [\gamma] \), or \( \text{Int} \rightarrow \text{Bool} \), or \( [\alpha] \rightarrow [\beta] \), or \( \text{Int} \lor [\alpha] \rightarrow [\beta] \)). Of these solutions only the last two are minimal (the first instance of the third one and the second is an instance of the fourth one) and since both are valid we can take their intersection, yielding the (minimum) solution

\[
(\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}) \land (\text{Int} \lor [\alpha]) \rightarrow [\text{Bool} \lor (\alpha \setminus \text{Int})])
\]

(25)

Moreover, we can dually follow the algorithm, perform an iteration, expand the type of the function, yielding the constraint-set

\[
((\alpha_1 \rightarrow \beta_1) \rightarrow [\alpha_1] \rightarrow [\beta_1]) \land ((\alpha_2 \rightarrow \beta_2) \rightarrow [\alpha_2] \rightarrow [\beta_2]) \\
\leq ((\text{Int} \rightarrow \text{Bool}) \land ((\alpha \setminus \text{Int}) \rightarrow (\alpha \setminus \text{Int}))) \rightarrow \gamma
\]

89
form which we will get the type (25) directly.

As stated in Subsection E.1, we chose an arbitrary order on type variables, which affects the generated substitutions and then the result types. Assume that σ₁ and σ₂ are two type substitutions generated by different ordinals. Thanks to the completeness of the tallying problem, there exist σ′₁ and σ′₂ such that σ₂ ≃ σ₁ o σ′₁ and σ₁ ≃ σ₂ o σ′₂. Therefore, the result types corresponding to σ₁ and σ₂ are equivalent under \( \sqsubseteq_\Delta \), that is \( σ₂(γ) \sqsubseteq_\Delta σ₁(γ) \) and \( σ₁(γ) \sqsubseteq_\Delta σ₂(γ) \). However, this does not imply that \( σ₁(γ) ≃ σ₂(γ) \).

For example, \( α \sqsubseteq_\Delta 0 \) and \( 0 \sqsubseteq_\Delta α \), but \( α \not≃ 0 \). Moreover, some result types are easier to understand or more precise than some others. Which one is better is a language design and implementation problem\(^{13}\). For example, consider the \texttt{map even} again. The type (25) is obtained under the order \( o(α₁) < o(β₁) < o(α) \). While under the order \( o(α) < o(α₁) < o(β₁) \), we will instead get

\[ (\[ β \setminus \text{Int} \rightarrow [β]) \land ([\text{Int} \lor \text{Bool} \lor β] \rightarrow [\text{Bool} \lor β]) \]  

(26)

It is clear that (25) \( \sqsubseteq_0 (26) \) and (26) \( \sqsubseteq_0 (25) \). However, compared with (25), (26) is less precise and less comprehensible, if we look at the type \([\text{Int} \lor \text{Bool} \lor β] \rightarrow [\text{Bool} \lor β] \) : (1) there is a \texttt{Bool} in the domain which is useless here and (2) we know that \texttt{Int} cannot appear in the returned list, but this is not expressed in the type.

A final word on completeness, which states that for every solution of the application problem, our algorithm finds a solution that is more general. However this solution is not necessary the first one found by the algorithm: even if we find a solution, continuing with a further expansion may yield a more general solution. We have just seen that, in the case of \texttt{map even}, the good solution is the second one, although this solution could already have been deduced by intersecting the first minimal solutions we found. Another simple example is the case of the application of a function of type \((α \times β) \rightarrow (β \times α)\) to an argument of type \((\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int})\). For this applications our algorithm (extended for product types) returns after one iteration the type \([\text{Int} \lor \text{Bool}] \times [\text{Int} \lor \text{Bool}]\) (since it unifies \( α \) with \( β \)) while one further iteration allows the system to deduce the more precise type \([(\text{Int} \lor \text{Bool}) \lor (\text{Bool} \times \text{Int})\]. Of course this raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist? This is a problem we did not tackle in this work, but if the answer to the previous question were negative then it would be easy to prove the existence of a principal type: since at each iteration there are only finitely many solutions, then the principal type would be the intersection of the minimal solutions of the last iteration (how to decide that an iteration is the last one is yet another problem).

**E.2.3 Heuristics to stop type-substitution inference**

We only have a semi-algorithm for \( t \bullet s \) because, as long as we do not find a solution, we may increase the cardinalities of \( I \) and \( J \) (where \( I \) and \( J \) are defined as in the previous sections) indefinitely. In this section, we propose two define two heuristic numbers \( p \) and \( q \) for the cardinalities of \( I \) and \( J \) that are established according to the form of \( t \) and \( s \). These heuristic numbers set the upper limit for the procedure: if no solution is found when the cardinalities of \( I \) and \( J \) have reached these heuristic numbers, then the procedure stops returning failure. This yields a terminating algorithm for \( t \bullet s \) which is clearly sound, but in our case, not complete. Whether it is possible to define these boundaries so that they ensure termination and completeness is still an open issue.

Through some examples, we first analyse the reasons one needs to expand the function type \( t \) and/or the argument type \( s \): the intuition is that type connectives are what makes the expansions necessary. Then based on these analyses, we give some heuristic numbers for the expansions. These heuristics are following some simple (but we believe reasonable) guidelines. First, when the substitutions found for a given \( p \) and \( q \) yield a useless type (e.g. “0 → 0” that is, the type of a function that cannot be applied to any value), it seems sensible to expand the types (that is increase \( p \) or \( q \)), in order to find a more informative substitutions. Second, if iterating the process does not give a more precise type, (in the sense of \( \sqsubseteq_\Delta \)) then it seems sensible to stop. Last, when the process always yields a more precise type, we choose to stop when the type is “good enough” for the programmer. In particular we choose to not introduce too many new fresh variables that make the type arbitrarily more precise but at the same time less “programmer friendly”. We illustrate these behaviours for three strategies: increasing \( p \) (that is, expanding the domain of the function), increasing \( q \) (that is, expanding the type of the argument) or lastly increasing both \( p \) and \( q \) at the same time.

**Expansion of \( t \).** A simple reason to expand \( t \) is the presence of (top-level) unions in \( s \). Generally, it is better to have as many copies of \( t \) as there are disjunctions in \( s \). Consider the example,

\[
\begin{align*}
t &= (α \rightarrow α) \rightarrow (α \rightarrow α) \\
s &= (\text{Int} \rightarrow \text{Int}) \lor (\text{Bool} \rightarrow \text{Bool})
\end{align*}
\]

(27)

If we do not expand \( t \) (i.e., if \( p = 1 \)), then the result type computed for the application of \( t \) to \( s \) is \( 0 → 0 \). However, this result type cannot be applied hereafter, since its domain is \( 0 \), and is therefore useless. When \( p = 2 \), we get an extra result type, \((\text{Int} \rightarrow \text{Int}) \lor (\text{Bool} \rightarrow \text{Bool})\), which is obtained by instantiating \( t \) twice, by \texttt{Int} and \texttt{Bool} respectively. Carrying on expanding \( t \) does not give more result types, as we always

\(^{13}\)In the current implementation we assume that the type variables in the function type always have smaller ordinals than those in the argument type.
select only two copies of $t$ to match the two summands in $s$, according to the decomposition rule for arrows \ref{eq:decomposition_rule}.

Another example is

$$
t = (\alpha \times \beta) \to (\beta \times \alpha)
$$

$$
s = (\text{Int} \to \text{Boo}) \lor (\text{Bool} \times \text{Int})
$$

Without expansion, the result type is $(([\text{Int} \lor \text{Bool}] \times (\text{Bool} \lor \text{Int})))$ (\alpha unifies \text{Int} and \text{Bool}). If we expand $t$, there exists a more precise result type $(\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int})$, each summand of which corresponds to a different summand in $s$. Besides, due to the decomposition rule for product types \ref{eq:product_decomposition}, there also exist some other result types which involve type variables, like $((\text{Int} \lor \text{Bool}) \times \alpha) \lor ((\text{Int} \lor \text{Bool}) \times (\text{Int} \lor \text{Bool}) \lor \alpha)$. Expanding further $t$ makes more product decompositions possible, which may in turn generate new result types. However, we believe the type $(\text{Int} \times \text{Bool}) \lor (\text{Bool} \times \text{Int})$ is informative enough, and so we pick the heuristic number as 2, i.e., the number of summands in $s$.

We may have to expand $t$ also because of intersection. First, suppose $s$ is an intersection of basic types; it can be viewed as a single basic type. Consider the example

$$
t = \alpha \to (\alpha \times \alpha) \text{ and } s = \text{Int}
$$

Without expansion, the result type is $\gamma_1 = (\text{Int} \times \text{Int})$. With two expansions, besides $\gamma_1$, we get an other result type $\gamma_2 = (\beta \times \beta) \lor (\text{Int} \lor \beta \times \text{Int} \lor \beta)$. Generally, with $k$ expansions, we get $k$ result types of the form

$$
\gamma_k = (\beta_1 \times \beta_1) \lor \ldots \lor (\beta_{k-1} \times \beta_{k-1}) \lor ((\text{Int} \lor (\beta_1 \times \beta_1) \lor \ldots \lor (\beta_{k-1} \times \beta_{k-1})) \lor ((\text{Int} \lor (\beta_1 \times \beta_1) \lor \ldots \lor (\beta_{k-1} \times \beta_{k-1})) \lor \ldots \lor (\text{Int} \lor (\beta_1 \times \beta_1) \lor \ldots \lor (\beta_{k-1} \times \beta_{k-1})))
$$

It is clear that $\gamma_{k+1} \subseteq \gamma_k$. More over, it is easy to find two substitutions $[\sigma_1, \sigma_2]$ such that $[\sigma_1, \sigma_2] \vdash \gamma_k \subseteq \gamma_{k+1} (k \geq 2)$. Therefore, $\gamma_k$ is the minimum (with respect to $\subseteq$) of $\{\gamma_k, k \geq 1\}$, so expanding $t$ more than once is useless (we do not get more a precise type than $\gamma_2$). However, we think the programmer expects $(\text{Int} \times \text{Int})$ as a result type instead of $\gamma_2$. So we take the heuristic number here as 1.

An intersection of product types is equivalent to $\bigvee_{i \in I} (s_i \times s_2)$, so we consider just a single product type (and then use union for the general case). For instance,

$$
t = ((\text{Int} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd}) \lor (\text{Odd} \rightarrow \text{Odd}))
$$

For the application to succeed, we have a constraint generated for each component of the product type, namely $(\text{Int} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd})$ and $(\text{Odd} \rightarrow \text{Odd})$. As with Example \ref{eq:example_map}, it is better to expand $\alpha \rightarrow \alpha$ once for the first constraint, while there is no need to expand $\beta \rightarrow \beta$ for the second one. As a result, we expand the whole type $t$ once, and get the result type $((\text{Int} \rightarrow \text{Bool}) \lor ((\text{Int} \rightarrow \text{Even}) \lor (\text{Odd} \rightarrow \text{Odd})) \lor (\text{Odd} \rightarrow \text{Odd}))$ as expected. Generally, if the heuristic numbers of the components of a product type are respectively $p_1$ and $p_2$, we take $p_1 \times p_2$ as the heuristic number for the whole product.

Finally, suppose $s$ is an intersection of arrows, like for example map even.

$$
t = (\alpha \rightarrow \beta) \lor [\alpha] \rightarrow [\beta]
$$

$$
s = (\text{Int} \rightarrow \text{Boo}) \lor ((\gamma \setminus \text{Int}) \rightarrow (\gamma \setminus \text{Int}))
$$

When $p = 1$, the constraint to solve is $(\alpha \rightarrow \beta \geq s)$. As stated in Subsection \ref{eq:subsection}, we get four possible result types: $[\alpha] \rightarrow [\beta]$, $[\text{Int}] \rightarrow [\text{Boo}]$, $[\text{Boo}] \rightarrow [\text{Int}]$, or $[\text{Int} \lor \alpha] \rightarrow [\text{Boo} \lor (\alpha \setminus \text{Int})]$, and we can build the minimum one by taking the intersection of them. If we continue expanding $t$, any result type we obtain is an intersection of some of the result types we have deduced for $p = 1$. Indeed, assume we expand $t$ so that we get $p$ copies of $t$. Then we would have to solve either $(\bigvee_{i=1..p} \alpha_i \rightarrow \beta_i \geq s)$ or $(\bigwedge_{i=1..p} \alpha_i \rightarrow \beta_i \geq s)$. For the first constraint to hold, by the decomposition of arrows \ref{eq:decomposition_rule}, there exists $i_0$ such that $s \leq \alpha_{i_0} \rightarrow \beta_{i_0}$, which is the same constraint as for $p = 1$. The second constraint implies $s \leq \alpha_i \rightarrow \beta_i$ for all $i$; we recognize again the same constraint as for $p = 1$ (except that we intersect $p$ copies of it). Consequently, expanding does not give us more information, and it is enough to take $p = 1$ as the heuristic number in this case.

Following the discussion above, we propose in Table 1 a heuristic number $H_p(s)$ for how many times we should expand $t$, depending on the shape of $s$. We assume that $s$ is in normal form. Besides, it can be easily extended to recursive types by memoization.

The next example shows that performing $H_p(s)$ expansions of $t$ may not be enough to get a result type, confirming that this number is only a heuristic. Let

$$
t = ((\text{true} \times (\text{Int} \rightarrow \alpha)) \rightarrow t_1) \land ((\text{false} \times (\alpha \rightarrow \text{Boo})) \rightarrow t_2)
$$

$$
s = (\text{Bool} \rightarrow (\text{Int} \rightarrow \text{Boo}))
$$

Here $\text{dom}(t)$ is $(\text{true} \times (\text{Int} \rightarrow \alpha)) \lor (\text{false} \times (\alpha \rightarrow \text{Boo}))$. The type $s$ cannot be completely contained in either summand of $\text{dom}(t)$, but it can be contained in $\text{dom}(t)$. Indeed, the first summand requires the substitution of $\alpha$ to be a supertype of $\text{Bool}$ while the second one a subtype of $\text{Int}$. As $\text{Boo}$ is not a subtype of $\text{Int}$, to make the application possible, we have to expand the function type at least once. However, according to Table 1 the heuristic number in this case is 1.
Table 1. Heuristic number $H_p(s)$ for the expansions of $t$

<table>
<thead>
<tr>
<th>Shape of $s$</th>
<th>Number $H_p(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bigwedge_{i \in I} s_i$</td>
<td>$\Sigma_{i \in I} H_p(s_i)$</td>
</tr>
<tr>
<td>$\bigwedge_{i \in P} b_i \land \bigwedge_{i \in N} \neg b_i \land \bigwedge_{i \in P} \alpha_i \land \bigwedge_{i \in N} \neg \alpha_i$</td>
<td>1</td>
</tr>
<tr>
<td>$\bigwedge_{i \in P} (s_i \times s_i') \land \bigwedge_{i \in N} \neg (s_i \times s_i') \quad (s_1 \times s_2)$</td>
<td>$\Sigma_{N \subseteq N'} H_p(s_{N'} \times s_{N''})$</td>
</tr>
<tr>
<td>$\bigwedge_{i \in P} (s_i \to s_i') \land \bigwedge_{i \in N} \neg (s_i \to s_i')$</td>
<td>$H_p(s_1) \ast H_p(s_2)$</td>
</tr>
</tbody>
</table>

where $(s_1 \times s_2') = (\bigwedge_{i \in P} s_i \times \bigwedge_{i \in N} \neg s_i \times \bigwedge_{i \in P} s_i' \land \bigwedge_{i \in N \setminus N'} \neg s_i')$.

Expansion of $s$. For simplicity, we assume that $\text{dom}(\bigwedge_{i \in I} t_i) = \bigvee_{i \in I} \text{dom}(t_i) \sigma_i$, so that the tallying problem for the application becomes $\bigwedge_{i \in J} s \sigma_j'^2 \leq \bigvee_{i \in I} \text{dom}(t_i) \sigma_i$. We now give some heuristic numbers for $|J|$ depending on $\text{dom}(t)$.

First, consider the following example where $\text{dom}(t)$ is a union:

$$
\text{dom}(t) = (\text{Int} \to ((\text{Bool} \to \text{Bool}) \land (\text{Int} \to \text{Int}))) \\
\vee (\text{Bool} \to ((\text{Bool} \to \text{Bool}) \land (\text{Real} \to \text{Real})))
$$

(33)

For the application to succeed, we need to expand $\text{Int} \to ((\alpha \to \alpha) \land (\beta \to \beta))$ once (so that we can make two distinct instantiations $\alpha = \text{Bool}$ and $\alpha = \text{Int}$) and $\text{Bool} \to (\beta \to \beta)$ twice (for three instantiations $\beta = \text{Bool}$, $\beta = \text{Int}$, and $\beta = \text{Real}$), corresponding to the first and the second summand in $\text{dom}(t)$ respectively. Since the expansion distributes the union over the intersections, we need to get six copies of $s$. In detail, we need the following six substitutions: $\{\alpha = \text{Bool}, \beta = \text{Bool}\}$, $\{\alpha = \text{Bool}, \beta = \text{Int}\}$, $\{\alpha = \text{Bool}, \beta = \text{Real}\}$, $\{\alpha = \text{Int}, \beta = \text{Bool}\}$, $\{\alpha = \text{Int}, \beta = \text{Int}\}$, and $\{\alpha = \text{Int}, \beta = \text{Real}\}$, which are the Cartesian products of the substitutions for $\alpha$ and $\beta$.

If $\text{dom}(t)$ is an intersection of basic types, we take the heuristic number as 1. If it is an intersection of product types, we can rewrite it as a union of products and we only need to consider the case of just a single product type. For instance,

$$
\text{dom}(t) = ((\text{Int} \to \text{Int}) \times (\text{Bool} \to \text{Bool})) \\
\sigma = ((\alpha \to \alpha) \times (\alpha \to \alpha))
$$

(34)

It is easy to infer that the substitution required by the left component requires $\alpha$ to be $\text{Int}$, while the one required by the right component requires $\alpha$ to be $\text{Bool}$. Thus, we need to expand $s$ at least once. Assume that $s = (s_1 \times s_2)$ and we need $q_i$ copies of $s_i$ times with the type substitutions: $\sigma_1^i, \ldots, \sigma_{q_i}^i$. Generally, we can expand the whole product type so that we get $s_1 \times s_2$ copies as follows:

$$
\bigwedge_{j=1,...,q_1} (s_1 \times s_2) \sigma_j^1 \land \bigwedge_{j=1,...,q_2} (s_1 \times s_2) \sigma_j^2
$$

$\bigwedge_{j=1,...,q_1} \bigwedge_{j=1,...,q_2} s_1 \sigma_j^1 \times \bigwedge_{j=1,...,q_1} \bigwedge_{j=1,...,q_2} s_2 \sigma_j^2
$$

Clearly, this expansion is a subtype of $(\bigwedge_{j=1,...,q_1} s_1) \times \bigwedge_{j=1,...,q_2} s_2) \sigma_j^2$) and so the type tallying succeeds.

Next, consider the case where $\text{dom}(t)$ is an intersection of arrows:

$$
\text{dom}(t) = (\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool}) \\
\sigma = \alpha \to \alpha
$$

(35)

Without expansion, we need $(\alpha \to \alpha) \leq (\text{Int} \to \text{Int})$ and $(\alpha \to \alpha) \leq (\text{Bool} \to \text{Bool})$, which reduce to $\alpha = \text{Int}$ and $\alpha = \text{Bool}$; this is impossible. Thus, we have to expand $s$ once, for the two conjunctions in $\text{dom}(t)$.

Note that we may also have to expand $s$ because of unions or intersections occurring under arrows. For example,

$$
\text{dom}(t) = t' \to ((\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})) \\
\sigma = t' \to (\alpha \to \alpha)
$$

(36)

As in Example 35, expanding once the type $\alpha \to \alpha$ (which is under an arrow in $s$) makes type tallying succeed. Because $(t' \to s_1) \land (t' \to s_2) \preceq (t' \to s_1 \land s_2)$, we can in fact perform the expansion on $s$ and then use subsumption to obtain the desired result. Likewise, we may have to expand $s$ if $\text{dom}(t)$ is an arrow type and contains an union in its domain. Therefore, we have to look into $\text{dom}(t)$ and $s$ deeply if they contain both arrow types.

Following these intuitions, we define in Table2 a heuristic number $H_q(\text{dom}(t))$ for the expansions of $s$, which depends on $\text{dom}(t)$.

Together. Up to now, we have considered the expansions of $t$ and $s$ separately. However, it might be the case that the expansions of $t$ and $s$ are interdependent, namely, the expansion of $t$ causes the expansion of $s$ and vice versa. Here we informally discuss the relationship between the two, and hint as why decidability is difficult to prove.
Let \( \text{dom}(t) = t_1 \lor t_2 \), \( s = s_1 \lor s_2 \), and suppose the type tallying between \( \text{dom}(t) \) and \( s \) requires that \( s_1 \sigma_1 \geq s_2 \), where \( \sigma_1 \) and \( \sigma_2 \) are two conflicting type substitutions. Then we can simply expand \( \text{dom}(t) \) with \( \sigma_1 \) and \( \sigma_2 \), yielding \( t_1 \sigma_1 \lor t_2 \sigma_1 \lor t_1 \sigma_2 \lor t_2 \sigma_2 \). Clearly, this expansion type is a supertype of \( t_1 \sigma_1 \lor t_2 \sigma_2 \) and thus a supertype of \( s \). Note that as \( t \) is on the bigger side of \( \leq \), what the extra type (i.e., \( t_1 \sigma_1 \lor t_2 \sigma_2 \)) the expansion brings does not matter. That is to say, the expansion of \( t \) would not cause the expansion of \( s \).

However, the expansion of \( s \) could cause the expansion of \( t \), and even a further expansion of \( s \) itself. Assume that \( s = s_1 \lor s_2 \) and \( s_1 \) requires a different substitution \( \sigma_1 \) (i.e., \( s_1 \sigma_1 \leq \text{dom}(t) \)). If we expand \( s \) with \( \sigma_1 \) and \( \sigma_2 \), then we have

\[
(s_1 \lor s_2)\sigma_1 \land (s_1 \lor s_2)\sigma_2 = (s_1 \sigma_1 \land s_2 \sigma_2) \lor (s_2 \sigma_1 \land s_1 \sigma_2) \lor (s_2 \sigma_1 \land s_2 \sigma_2)
\]

It is clear that \( s_1 \sigma_1 \land s_2 \sigma_2 \), \( s_1 \sigma_1 \land s_2 \sigma_2 \), and \( s_2 \sigma_1 \land s_2 \sigma_2 \) are subtypes of \( \text{dom}(t) \). Consider the extra type \( s_1 \sigma_2 \land s_2 \sigma_1 \). If this extra type is empty (e.g., because \( s_1 \) and \( s_2 \) have different top-level constructors), or if it is a subtype of \( \text{dom}(t) \), then the type tallying succeeds. Otherwise, in some sense, we need to solve another type tallying between \( s \land (s_2 \sigma_1 \land s_1 \sigma_2) \) and \( \text{dom}(t) \), which would cause the expansion of \( t \) or \( s \). This is mainly the reason why we fail to prove the decidability of the application problem (that is, deciding \( \bullet_\triangle \)) so far.

To illustrate this phenomenon, consider the following example:

\[
\begin{align*}
\text{dom}(t) &= ((\text{Bool} \to \text{Bool}) \to (\text{Int} \to \text{Int})) \\
&\lor ((\text{Bool} \to \text{Bool}) \lor (\text{Int} \to \text{Int})) \lor ((\beta \to \beta) \lor (\text{Bool} \to \text{Bool})) \\
&\lor (\beta \lor \beta) \\
\text{s} &= (\alpha \to (\text{Int} \to \text{Int})) \lor ((\text{Bool} \to \text{Bool}) \lor \alpha) \lor (\text{Boo} \times \text{Bool})
\end{align*}
\]

Let us consider each summand in \( s \) respectively. For the first one, a solution is \( \alpha \geq \text{Bool} \to \text{Bool} \), which corresponds to the first summand in \( \text{dom}(t) \). The second one requires \( \alpha \leq \text{Int} \to \text{Int} \) and the third one \( \beta \geq \text{Boo} \). Since \( \text{Boo} \to \text{Boo} \) is not subtype of \( \text{Int} \to \text{Int} \), we need to expand \( s \) once, that is,

\[
\begin{align*}
\text{s'} &= s\{\text{Bool} \to \text{Bool}/\beta\} \lor s\{\text{Int} \to \text{Int}/\alpha\} \\
&= ((\text{Boo} \to \text{Boo}) \lor (\text{Int} \to \text{Int})) \lor ((\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})) \\
&\lor ((\text{Boo} \to \text{Boo}) \lor (\text{Boo} \to \text{Boo}) \lor (\text{Boo} \to \text{Boo}) \lor (\text{Boo} \to \text{Boo})) \lor ((\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})) \\
&\lor ((\text{Int} \to \text{Int}) \lor (\text{Int} \to \text{Int}) \lor (\text{Int} \to \text{Int})) \\
&\lor (\text{Boo} \times \text{Boo})
\end{align*}
\]

Almost all the summands of \( s' \) are contained in \( \text{dom}(t) \) except the extra type

\[(\text{Boo} \to \text{Boo}) \lor (\text{Boo} \to \text{Boo}) \lor (\text{Int} \to \text{Int}) \]

Therefore, we need to consider another type tallying involving this extra type and \( \text{dom}(t) \). By doing so, we obtain \( \beta = \text{Int} \); however we have inferred before that \( \beta \) should be a supertype of \( \text{Boo} \). Consequently, we need to expand \( \text{dom}(t) \); the expansion of \( \text{dom}(t) \) with \( \{\text{Boo}/\beta\} \) and \( \{\text{Int}/\beta\} \) makes the type tallying succeed.

In day-to-day examples, the extra type brought by the expansion of \( s \) is always a subtype of \( \text{dom}(t) \), and we do not have to expand \( \text{dom}(t) \) or \( s \) again. The heuristic numbers we gave seems to be enough in practice.

## F. Compilation into CoreCduce

In this section, we want to compile the polymorphic calculus into a variant of the (monomorphic) CoreCduce. The aim of this translation is to provide an execution model for our polymorphic calculus that does not depend on dynamic propagation of type substitution. We first introduce the core calculus of Cduce (Subsection F.1). Then we present the translation from our polymorphic calculus to Cduce and prove the soundness of the translation (Subsection F.2). Lastly we discuss some limitations of this approach and hint at some possible improvements.
F.1 CoreCDuce

In this section, we simply introduce the core calculus of CDuce, dubbed CoreCDuce. Then we introduce a “binding” type-case, which is needed by the translation.

**Definition F.1.** An expression $e$ of CoreCDuce is inductively defined by:

\[ e ::= c \mid x \mid (e, e) \mid \pi_i(e) \mid ee \mid \lambda^{i_1:e_1\rightarrow t_1}.x.e \mid e\epsilon t?e : e \]

where $t_1$, $s_1$, $t$ range over types. The set of all expressions is denoted as $\mathcal{E}_C$.

Type variables are considered as special atom types in CoreCDuce and the subtyping relation is the one defined in [3]. The typing rules are present in Figure F.1, which are similar to those in [5].

\[ \Gamma \vdash_C c : b_0 \quad \Gamma \vdash_C x : \Gamma(x) \quad \Gamma \vdash_C e_1 : t_1 \quad \Gamma \vdash_C e_2 : t_2 \]

\[ \Gamma \vdash_C e : t_1 \times t_2 \]

\[ \Gamma \vdash_C \pi_i(e) : t_i \quad \Gamma \vdash_C e_1 : t_1 \rightarrow t_2 \]

\[ \Gamma \vdash_C e_1 e_2 : t_2 \quad \Gamma \vdash_C \lambda^{i_1:e_1\rightarrow t_1}.x.e : \bigwedge_{i \in I} t_1 \rightarrow s_i \]

\[ \Gamma \vdash_C e : t_0 \quad \begin{cases} t_0 \not\leq t \Rightarrow \Gamma \vdash_C e_1 : s \\ t_0 \leq t \Rightarrow \Gamma \vdash_C e_2 : s \end{cases} \]

\[ \Gamma \vdash_C (e\epsilon t?e_1 : e_2) : s \quad \Gamma \vdash_C e : s \leq t \]

**Figure 15.** Typing Rules of CoreCDuce

**Definition F.2.** An expression $e$ is a value if it is closed, well-typed ($\vdash_C e : t$ for some type $t$), and produced by the following grammar:

\[ v ::= c \mid (v, v) \mid \lambda^{i_1:e_1\rightarrow t_1}.x.e \]

We write $\mathcal{V}_C$ to denote the set of all values.

For simplicity, we restrict the contexts to the evaluation contexts (Definition A.15). The reduction rules are shown in Figure F.16.

\[ (CRproj) \quad \pi_i(v_1, v_2) \rightsquigarrow_C v_i \]

\[ (CRappl) \quad (\lambda^{i_1:e_1\rightarrow t_1}.x.e')v \rightsquigarrow_C e'[v/x] \]

\[ (CRCase) \quad (v\epsilon t?e_1 : e_2) \rightsquigarrow_C \begin{cases} e_1 \text{ if } \vdash_C v : t \\ e_2 \text{ otherwise} \end{cases} \]

\[ (CRCtx) \quad E[e] \rightsquigarrow_C E'[e'] \text{ if } e \rightsquigarrow_C e' \]

**Figure 16.** Reduction rules of CoreCDuce

The translation we wish to define is type-driven, and therefore expects an annotated, well-typed expression of the polymorphic calculus. We therefore assume that we have access to the (already computed) type of any expression. The translation relies on an extension of the (monomorphic) type-case expression that features binding, which we write $(x=e)\epsilon t?e_1 : e_2$, and that can be encoded as:

\[ (\lambda^{i_1:e_1\rightarrow t_1}.x.e)\epsilon t?e_1 : e_2 \]

where $s$ is the type of $e$, $t_1$ the type of $e_1$, and $t_2$ the type of $e_2$. An extremely useful and frequent case (also in practice) is when the expression $e$ in $(x=e)\epsilon t?e_1 : e_2$ is syntactically equal to the binding variable $x$, that is $(x=x)\epsilon t?e_1 : e_2$. For this particular case it is worth introducing specific syntactic sugar (distinguished by a boldface “belongs to” symbol): $x \epsilon t?e_1 : e_2$. The reader may wonder what is the interest of binding a variable to itself. Actually, the two occurrences of $x$ in $(x=x)\epsilon t$ denote two distinct variables: the one on the right is recorded in the environment with some type $s$; this variable does not occur either in $e_1$ or
e_2$ because it is hidden by the $x$ on the left; this binds the occurrences of $x$ in $e_1$ and $e_2$ but with different types, $s$ and $t$ in $e_1$ and $s \land \neg t$ in $e_2$. The advantage of such a notation is to allow the system to use different type assumptions for $x$ in each branch, as stated by the typing rule directly derived from the encoding:

$$
\begin{align*}
\Delta \Gamma \vdash t_1 & \implies \Delta \Gamma \vdash t_2 \\
\Gamma(x) \land t_1 & \implies \Delta \Gamma \vdash e_1 : s_i \\
\Gamma(x) \land \neg t_1 & \implies \Delta \Gamma \vdash e_2 : \bigvee_{i \neq 0} s_i
\end{align*}
$$

(Case-var)

Note that $x$ is defined in $\Gamma$ but its type $\Gamma(x)$ is overridden in the premises to type the branches. With this construction, map can be defined as:

$$
\lambda (\alpha \rightarrow \beta) \cdot \lambda (\alpha \rightarrow \beta) \cdot \ell \cdot e \cdot \text{match}_{\text{with}} \ e \ (\ell \cdot \text{nil} \ ? \ (f(\pi_1 \ell), mf(\pi_2 \ell)))
$$

In practice any real programming language would implement either $e$ (and not $\in$) or an even more generic construct (such as match_with or case_of pattern matching where each pattern may re-bind the $x$ variable to a different type).

F.2 Translation to CoreDuce

We first illustrate how to translate our polymorphic calculus to CoreDuce and then prove that the translation is sound.

The translation we propose creates CoreDuce expressions whose evaluation simulates the run-time type substitutions that may occur during the evaluation of an expression of the polymorphic calculus.

As a starting point, let us recall what happens during the evaluation of an expression of the polymorphic calculus (Figure 6 of Section A.4). First, an explicit substitution is propagated using the relabeling operation (Rule (Relsub) in Figure 6). This propagation stops at the lambda abstractions, which becomes annotated with the propagated set of substitutions. Second, when a lambda abstraction is applied to some argument the annotations of that lambda abstraction are propagated to the body (Rule (Rappl)). However the type of the argument determine which substitutions are propagated to the body. The translation from the polymorphic calculus to CoreDuce reproduces these two steps: (1) it first pushes the explicit substitutions to decorations of the underlying abstractions, and (2) it encodes the run-time choice of which substitution to propagate by a set of nested type-case expressions.

Consider the following expression

$$
\lambda (\alpha \rightarrow \beta) \cdot f. \lambda (\alpha \rightarrow \beta) \cdot x.fx\{\text{Int}_{\alpha}, \text{Bool}_{\alpha}\}
$$

Intuitively, the type substitutions take effect only at polymorphic abstractions. So we first push the explicit type substitutions to decorations of the underlying abstractions. To do so, we perform the relabeling on the expression, namely the application of $\theta$, yielding

$$
\lambda (\alpha \rightarrow \beta) \cdot f. \lambda (\alpha \rightarrow \beta) \cdot x.fx
$$

Secondly we show how we encode the dynamic relabeling that occurs at application time. In our example, the type for the whole abstraction is

$$
((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}) \land ((\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool})
$$

but we cannot simply propagate all the substitutions to the body expression since this leads to to an ill-typed expression. The idea is therefore to use the “bind” type case, to simulate different relabeling on the body expression with different type cases. That is, we check which type substitutions are used by the type of the parameter and then propagate them to the body expression. So an encoded expression of (38) is as follows:

$$
\begin{align*}
\lambda (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} & \rightarrow \text{Int}) \land ((\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \rightarrow \text{Bool}) \ \ell \cdot e \cdot (\text{Int} \rightarrow \text{Int}) \\
& \implies \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
& \implies \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\end{align*}
$$

Where $C[e]$ denotes the encoding $e$. The first branch simulates the case where both type substitutions are selected and propagated to the body, that is, the parameter $f$ belongs to the intersection of different instances of $\alpha \rightarrow \alpha$ with these two type substitutions. The second and third branches simulate the case where exactly one of the substitution is used. Finally, the last branch denotes the case where no type substitution is selected. Note that this last case can never happen, since by typing, we know that the application is well-typed and therefore that the argument is of type $\text{Bool} \rightarrow \text{Bool}$ or $\text{Int} \rightarrow \text{Int}$ (or of course, their intersection). This last case is only here to keep the expression syntactically correct (it is the “else” part of the last type-case) and can be replaced with a dummy expression.

Here, we see that the “binding” type-case is essential since it allows to “refine” the type of the parameter of a lambda abstraction (replacing it by one of its instances). Note also that although the binding type case is encoded using a lambda abstraction, is a lambda abstraction of CoreDuce which are not decorated.
The trickiest aspect of our encoding is that the order of the branches is essential: the more precise the type (in the sense of the subtyping relation) the sooner it must occur in the chain of type-cases. Indeed, consider the previous example but suppose that one tests for \((\text{Int} \to \text{Int})\) before \((\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})\). For an argument of type \((\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})\), the branch \(C[\lambda x.(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})]x.x\) would be evaluated instead of the more precise \(C[\lambda x.(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})]_s x.x\). Worst, if one now tests for \(\text{Bool} \to \text{Bool}\) before \((\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})\) and if the argument is again of type \(\text{Int} \to \text{Int}\), then branch \(C[\lambda x.(\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool})]_s x.x\) is taken (since the intersection between \(\text{Int} \to \text{Int}\) and \(\text{Bool} \to \text{Bool}\) is not empty). This clearly yields an unsound expression since now \(f\) is bound to a value of type \(\text{Int} \to \text{Int}\) but used in a branch where the substitution \(\{\text{Bool}/\alpha\}\) is applied.

**Definition F.3.** Let \(S\) be a set. We say that a sequence \(L = S_0, \ldots, S_n\) is a sequence of ordered subsets of \(S\) if and only if

- \(\mathcal{P}(S) = \{S_0, \ldots, S_n\}\)
- \(\forall i, j \in \{0, \ldots, n\}, S_i \subset S_j \implies i > j\)

Given a set \(S\), there exists several sequences of ordered subsets of \(S\). We consider the sequences equivalent modulo permutation of incomparable subsets and denote by \(\text{OrdSet}(S)\) the representative of this equivalence class. In layman’s terms, \(\text{OrdSet}(S)\) is a sequence of all subsets of \(S\) such that any element of \(S\) appears in the sequence before its proper subsets. We also introduce the notation:

\[
\{x \in t_i \mid e_i\}_{i=1..n}
\]

which is short for

\[
\left(\begin{array}{c}
x \in t_1 ? e_1 :
\vdots
x \in t_{n-1} ? e_{n-1} :
\mid x \in t_n ? e_n
\end{array}\right)
\]

We can now formally define our translation to CoreCduce:

**Definition F.4.** Let \(C[\_\_]\) be a function from \(\&\) to \(\&\), defined as

\[
\begin{align*}
C[c] &= c \\
C[x] &= x \\
C[(c_1, c_2)] &= (C[c_1], C[c_2]) \\
C[π(c)] &= π(C[c]) \\
C[e_1 e_2] &= C[e_1] C[e_2] \\
C[e? e_1 : e_2] &= C[e] e_1 ? C[e_1] : C[e_2] \\
C[e[σ]_{i,j}] &= C[e[σ]_{i,j}] \\
C[λ\alpha^{i \to j} x. e] &= λ\alpha^{i \to j} x. e. \\
\end{align*}
\]

where \(t_P = \bigwedge_{(i,j) \in P} t_{σ_j} \) and \(j \propto P \) means that \(j \in \{k \mid \exists i \cdot (i, k) \in P\}\).

We must show that this translation is faithful, that is that given an expression and its translation, they reduce to the same value and that both have the same type. We proceed in several steps, using auxiliary lemmas. First we show the translation preserves types. Then that values and their translations are the same and have the same types. Lastly we show (at the end of the section) that the translation preserves the reduction of well-typed expressions.

We prove that the translation is **type-preserving** (Lemma F.6). We first show an auxiliary lemma that states that the translation of relabeled expression preserves its type:

**Lemma F.5.** Let \(e\) be an expression. If \(\Delta \Gamma \vdash e[σ]_{i,j} : t\), then \(\Gamma \vdash C[e[σ]_{i,j}] : t\).

**Proof.** The proof proceeds by induction and case analysis on the structure of the typing derivation \(e\).

**Case (subsum):** the typing derivation has the form:

\[
\Delta \Gamma \vdash e[σ]_{i,j} : s \quad s \leq t
\]

By applying the induction hypothesis on the premise, we have \(\Gamma \vdash C[e[σ]_{i,j}] : s\). Since \(s \leq t\), by subsumption, we have \(\Gamma \vdash C[e[σ]_{i,j}] : t\).

**Case (const) \(e \equiv c\):** here, \(C[e[σ]_{i,j}] = e[σ]_{i,j} = c\). Trivially, we have \(\Gamma \vdash C[e[σ]_{i,j}] : b_c\).

**Case (var) \(e \equiv x\):** similar to the previous case.
(pair) $e \equiv (e_1, e_2)$: $e_0@[[\sigma]]_j \mapsto (e_1@[[\sigma]]_j, e_2@[[\sigma]]_j)$ and 
$C[e_0@[[\sigma]]_j]_j = (C[e_1@[[\sigma]]_j], C[e_2@[[\sigma]]_j]).$ The typing derivation ends with:

$$\frac{\Gamma \vdash e_1@[[\sigma]]_j : s_1 \quad \Gamma \vdash e_2@[[\sigma]]_j : s_2}{\Delta \vdash (e_1@[[\sigma]]_j, e_2@[[\sigma]]_j) : (s_1 \times s_2)}$$

Applying the induction hypothesis on each premise, we obtain $\Gamma \vdash C[e_1@[[\sigma]]_j)_j : s$, we conclude by applying rule (Cpair), which gives us 
$$\Gamma \vdash C[e_1@[[\sigma]]_j), C[e_2@[[\sigma]]_j)_j : (s_1 \times s_2)$$

that is $\Gamma \vdash C[(e_1, e_2)@[[\sigma]]_j)_j : (s_1 \times s_2)$.

(proj) $e \equiv \pi_1(e')$: here, $e_0@[[\sigma]]_j = \pi_1(e'@[[\sigma]]_j)$ and $C[e_0@[[\sigma]]_j)_j = \pi_1(C[e'@[[\sigma]]_j)_j).$ The typing derivations ends with:

$$\frac{\Gamma \vdash e_0@[[\sigma]]_j : s \quad \Gamma \vdash e_0@[[\sigma]]_j : s}{\Delta \vdash \pi_1(e'@[[\sigma]]_j)_j : t}$$

By induction, we have $\Gamma \vdash C[e'@[[\sigma]]_j)_j : t \times s$. We conclude this case by applying rule (Cproj), which gives us $\Gamma \vdash \pi_1(C[e'@[[\sigma]]_j)_j) : t_1 \times s_1$, that is $\Gamma \vdash C[\pi_1(e')@[[\sigma]]_j)_j : t_1$.

(app) $e \equiv e_1 e_2$: here, $e_0@[[\sigma]]_j = (e_1@[[\sigma]]_j)(e_2@[[\sigma]]_j)$ and 
$C[e_0@[[\sigma]]_j)_j = C[e_1@[[\sigma]]_j), C[e_2@[[\sigma]]_j)_j.$ The typing derivation ends with:

$$\frac{\Gamma \vdash e_1@[[\sigma]]_j : t \to s \quad \Gamma \vdash e_2@[[\sigma]]_j : t}{\Delta \vdash e_1 e_2@[[\sigma]]_j) : t}$$

We apply the induction hypothesis on the premises and obtain $\Gamma \vdash C[e_1@[[\sigma]]_j)_j : t \to s$ and $\Gamma \vdash C[e_2@[[\sigma]]_j)_j : t$. Then the rule (Capp) gives us the result: $\Gamma \vdash C[e_1@[[\sigma]]_j)_j C[e_2@[[\sigma]]_j)_j : s$, that is $\Gamma \vdash C[(e_1 e_2)@[[\sigma]]_j)_j : s$.

(abstr) $e \equiv \lambda x. x':$ unsurprisingly, this case is the trickiest. We have 
$$e_0@[[\sigma]]_j = \lambda x. x'@[[\sigma]]_j x',$$
and the typing derivation ends with:

$$\forall i \in I, k \in K, j \in J, \Delta \vdash \Gamma, (x : t_i(\sigma_k \circ \sigma_j)) \vdash e_0@[[\sigma]]_j : s_i(\sigma_k \circ \sigma_j)$$

$$\Delta' = \Delta \cup \text{var}((\bigcup_{i \in I, k \in K, j \in J} t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)))$$

$$\Gamma \vdash \lambda x. x'@[[\sigma]]_j x' : \lambda x \in I, k \in K, j \in J, t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)$$

Let us consider Definition [index] in which we replace $\sigma_j$ by $\sigma_k \circ \sigma_j$. We obtain:

$$C[\lambda x. x'@[[\sigma]]_j x'] = \lambda x \in I, k \in K, j \in J, t_i(\sigma_k \circ \sigma_j) \rightarrow s_i(\sigma_k \circ \sigma_j)$$

where $t_P = \bigcup_{(k, j) \in P} t_i(\sigma_k \circ \sigma_j)$ and $(k, j) \in P$ means that $(k, j) \in \{(k, j) | \exists x, (i, k, j) \in P\}$. Let us consider the premises of the (abstr) rule. For each arrow type $t_{i_0}(\sigma_{k_0} \circ \sigma_{j_0}) \rightarrow s_{i_0}(\sigma_{k_0} \circ \sigma_{j_0})$, we need to prove

$$\Gamma, (x : t_{i_0}(\sigma_{k_0} \circ \sigma_{j_0})) \vdash C \{x \in t_P ? C[e'@[[\sigma]]_j]_{(k, j) \times P} \}_{P \in \text{OrdSet}(I \times K \times J) : s_{i_0}(\sigma_{k_0} \circ \sigma_{j_0})}$$

or, said differently, that each branch of the type case is either not type-checked or if it is, that is has type $s_{i_0}(\sigma_{k_0} \circ \sigma_{j_0})$. Let us consider any branch whose condition type is $t_P$. First let us remark than if $(i_0, k_0, j_0)$ (the triple that defines the type of $x$) is not in $P$, then the branch is not type-checked. Indeed, since the branches are ordered according to $\text{OrdSet}(I \times K \times J)$, if $(i_0, k_0, j_0) \notin P$, then there exists $P' = P \cup \{(i_0, k_0, j_0)\}$. Since $P \subset P'$ the branch whose condition type is $t_P$ is placed before the one with type $t_P$ in the type case. We can therefore deduce that for the case $t_P, x$ has type $t_{i_0}(\sigma_{k_0} \circ \sigma_{j_0}) \land t_P \land \neg d_{i_0} \land \cdots \land 0$ (the negations coming from the types of the previous branches—including $t_P$—that are removed, see rule (case)). Since $x$ has type 0 the branch is not type-checked.

Therefore we can assume that $(i_0, k_0, j_0) \in P$ and that the branch is taken. In this branch, the type of $x$ is restricted to:

$$t_P = \bigcup_{(i, k, j) \in P} t_i(\sigma_k \circ \sigma_j) \land \bigcup_{(i, k, j) \in (I \times K \times J) \setminus P} \neg t_i(\sigma_k \circ \sigma_j)$$

We now only have to prove that:

$$\Gamma, (x : t_P) \vdash C \{e'@[[\sigma]]_j \}_{(k, j) \times P} : s_{i_0}(\sigma_{k_0} \circ \sigma_{j_0})$$
for all \( t' \) such that \((i_0, k_0, j_0) \in P \) and \( t' \neq 0 \). Based on the premises of the rules \((\text{abstr})\), by Lemma [B.13] we get

\[
\Delta \triangleright s' (\bigwedge_{(i, k, j) \in P} \Delta, (x : t_P) \vdash e' \circ [\sigma_k \circ \sigma_j]_{(i, k, j)} : \bigvee_{(i, k, j) \in P} s_i(\sigma_k \circ \sigma_j))
\]

Then following Lemma [B.8] we have

\[
\Delta \triangleright r \Gamma, (x : t_P) \vdash e' \circ [\sigma_k \circ \sigma_j]_{(i, k, j)} : \bigwedge_{(i, k, j) \in P} s_i(\sigma_k \circ \sigma_j)
\]
on to which we can apply the induction hypothesis to obtain:

\[
\Gamma, (x : t_P) \vdash_C C[e' \circ [\sigma_k \circ \sigma_j]_{(i, k, j)} : \bigvee_{(i, k, j) \in P} s_i(\sigma_k \circ \sigma_j)]
\]

Since \((i_0, k_0, j_0) \in P\), we have \( \bigwedge_{(i, k, j) \in P} s_i(\sigma_k \circ \sigma_j) \leq s_{i_0}(\sigma_{k_0} \circ \sigma_{j_0}) \). The result follows by sub-

\[
\text{(case)} \quad e \equiv (e_0 : t ? e_1 : e_2) ; \text{ here, } e_0 \circ [\sigma_j]_{j \in J} = e_0 \circ [\sigma_j]_{j \in J} \in t ? e_1 \circ [\sigma_j]_{j \in J} : e_2 \circ [\sigma_j]_{j \in J} \text{ and } C[e_0 \circ [\sigma_j]_{j \in J}] = C[e_1 \circ [\sigma_j]_{j \in J}] = C[e_2 \circ [\sigma_j]_{j \in J}] . \text{ The typing derivation ends with:}
\]

\[
\Delta \triangleright s \Gamma \vdash e_0 \circ [\sigma_j]_{j \in J} : t \quad \Delta \triangleright s \Gamma \vdash e_1 \circ [\sigma_j]_{j \in J} : s
\]

By induction hypothesis, we have \( \Gamma \vdash_C C[e_0 \circ [\sigma_j]_{j \in J}] : t' \) and \( \Gamma \vdash_C C[e_1 \circ [\sigma_j]_{j \in J}] : s \). We can apply the typing rule \((C \text{case})\), which proves this case.

\[
\text{(instinter)} \quad e \equiv e' \circ [\sigma_j]_{j \in J} ; \text{ here, } e' \circ [\sigma_j]_{j \in J} = e' \circ [\sigma_j]_{j \in J} \in (t \times J) \text{ and } C[e' \circ [\sigma_j]_{j \in J}] = C[e' \circ [\sigma_j]_{j \in J}] . \text{ By induction on } e', \text{ we have } \Gamma \vdash_C C[e' \circ [\sigma_j]_{j \in J}] : t, \text{ that is, } \Gamma \vdash_C C[e' \circ [\sigma_j]_{j \in J}] : t .
\]

\[
\square
\]

**Lemma F.6.** Let \( e \) be an expression. If \( \Delta \triangleright s \Gamma \vdash e : t \), then \( \Gamma \vdash_C C[e] : t \).

**Proof.** We proceed by case on the last typing rule used to derive the judgment \( \Delta \triangleright s \Gamma \vdash e : t \) and build a corresponding derivation for the judgment \( \Gamma \vdash_C C[e] : t \).

\[
\text{(const): } \quad \Delta \triangleright s \Gamma \vdash c : b_c \text{ and } C[e] = c . \text{ It is clear that } \Gamma \vdash_C c : b_c .
\]

\[
\text{(var): } \quad \Delta \triangleright s \Gamma \vdash x : \Gamma(x) \text{ and } C[x] = x . \text{ It is clear that } \Gamma \vdash_C x : \Gamma(x) .
\]

\[
\text{(pair): consider the derivation:}
\]

\[
\Delta \triangleright s \Gamma \vdash e_1 : t_1 \quad \Delta \triangleright s \Gamma \vdash e_2 : t_2 \quad \text{(pair)}
\]

Applying the induction hypothesis on each premise, we get \( \Gamma \vdash_C C[e_1] : t_i \) for \( i = 1, 2 \). Then by applying \((C \text{pair})\), we get \( \Gamma \vdash_C C[e_1], C[e_2] : (t_1 \times t_2) \), that is, \( \Gamma \vdash_C C[(e_1, e_2)] : (t_1 \times t_2) \).

\[
\text{(proj): consider the derivation:}
\]

\[
\Delta \triangleright s \Gamma \vdash e : t_1 \times t_2 \quad \text{(proj)}
\]

By induction, we have \( \Gamma \vdash_C C[e] : t_1 \times t_2 \). Then by \((C \text{proj})\), we get \( \Gamma \vdash_C \pi_i(C[e]) : t_i \), and consequently \( \Gamma \vdash_C C[\pi_i(e)] : t_i \).

\[
\text{(appl): consider the derivation:}
\]

\[
\Delta \triangleright s \Gamma \vdash e_1 : t \to s \quad \Delta \triangleright s \Gamma \vdash e_2 : t \quad \text{(pair)}
\]

By induction, we have \( \Gamma \vdash_C C[e_1] : t \to s \) and \( \Gamma \vdash_C C[e_2] : t \). Then by \((C \text{appl})\), we get \( \Gamma \vdash_C C[e_1]C[e_2] : s \), that is, \( \Gamma \vdash_C C[e_1e_2] : s \).
\[(\textit{abstr}):\text{ consider the derivation:}\]
\[
\forall i, j \in J, \Delta, \Gamma \vdash \Delta_{i} \vdash e : \sigma_{i} \\vdash \frac{\Delta_{i} \Gamma \vdash e : \sigma_{i} \\vdash \Delta_{i} \vdash \lambda_{i \in I, j \in J} t_{i} \sigma_{j} \rightarrow s_{i} \sigma_{j}}{\Delta_{i} \vdash \lambda_{i \in I, j \in J} t_{i} \sigma_{j} \rightarrow s_{i} \sigma_{j}} \quad (\textit{abstr})
\]

According to Definition 2.4, we have
\[
\forall \Delta, \Gamma \vdash C_{\lambda_{i \in I, j \in J} t_{i} \sigma_{j} \rightarrow s_{i} \sigma_{j}} = \lambda_{i \in I, j \in J} t_{i} \sigma_{j} \rightarrow s_{i} \sigma_{j}
\]
where \( t_{p} = \bigcup_{(i, j) \in P} t_{i} \sigma_{j}, j \propto P \) means that \( j \in \{ k \mid \exists i, (i, k) \in P \} \).

Then following Lemma B.8, we have
\[
\frac{\Gamma, x : t_{i} \sigma_{j} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}{\Delta_{i} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}
\]
for each arrow type \( t_{i} \sigma_{j} \rightarrow s_{i} \sigma_{j} \), we need to prove that
\[
\Gamma, x : t_{i} \sigma_{j} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}
\]
for all \( t_{p} \) such that \((i, j) \in P\) and \( t_{p} \neq 0 \). Using Lemma B.13, we get
\[
\Delta_{i} \vdash \Gamma, (x : t_{i} \sigma_{j}) \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}
\]
Then following Lemma B.8, we have
\[
\frac{\Gamma, x : t_{i} \sigma_{j} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}{\Delta_{i} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}
\]
By Lemma F.5, we get
\[
\frac{\Gamma, x : t_{i} \sigma_{j} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}{\Delta_{i} \vdash \{ x \in \mathcal{P} ? C [ e \in \sigma_{j} ] \propto \mathcal{P} \}}
\]
Since \((i, j) \in P\), we have \( \bigcup_{(i, j) \in P} s_{i} \sigma_{j} \leq s_{i} \sigma_{j} \) and the result follows by subsumption.

\[(\textit{case})\text{: consider the following derivation}\]
\[
\frac{\Delta_{i} \vdash e : t}{\Delta_{i} \vdash e : t \land \Delta_{i} \vdash c : s}
\]
By induction, we have \( \Gamma \vdash C [ e ] : t \) and \( \Gamma \vdash C [ e ] : s \). Then by (case), we get \( \Gamma \vdash C [ e ] : s \), which gives \( \Gamma \vdash C [ e ] : s \).

\[(\textit{instinter})\text{: consider the following derivation}\]
\[
\frac{\Delta_{i} \vdash e : t \land \Delta_{i} \vdash c : s}{\Delta_{i} \vdash e[\sigma_{j}]_{j \in J} : \bigwedge_{j \in J} t_{j} \sigma_{j}}
\]
According to Corollary B.12, we get \( \Delta_{i} \vdash e[\sigma_{j}]_{j \in J} : \bigwedge_{j \in J} t_{j} \sigma_{j} \). Then by Lemma F.5, we have \( \Gamma \vdash C [ e[\sigma_{j}]_{j \in J} ] : \bigwedge_{j \in J} t_{j} \sigma_{j} \), that is, \( \Gamma \vdash C [ e[\sigma_{j}]_{j \in J} ] : \bigwedge_{j \in J} t_{j} \sigma_{j} \).

\[(\textit{subsum})\text{: there exists a type } s \text{ such that}\]
\[
\frac{\Delta_{i} \vdash e : s \land \Delta_{i} \vdash e : t}{\Delta_{i} \vdash e : s \leq t}
\]
By induction, we have \( \Gamma \vdash C [ e ] : s \). Then by subsumption, we get \( \Gamma \vdash C [ e ] : t \).

Although desirable, preserving the type of an expression is not enough. The translation also preserves values, as stated by the following lemma:

**Lemma F.7.** Let \( v \in \mathcal{Y} \) be a value. Then \( C [ v ] \in \mathcal{Y} \).

**Proof.** By induction on \( v \).
Proof.\] It is clear that $\mathcal{C}[e] \in \mathcal{Y}_C$.

\(\lambda^{\sigma_1\in I_1, \ldots, \sigma_j\in I_j, \sigma_j\mapsto \sigma_j} x.e : C[e] \in \mathcal{Y}_C.\)

\((v_1,v_2) : C[v] = (C[v_1], C[v_2]).\) By induction, we have $C[v_1] \in \mathcal{Y}_C$. And so does $(C[v_1], C[v_2]) \in \mathcal{Y}_C.$

Relating arbitrary expressions of the core calculus and their translation in CoreCDuce is tricky. Indeed, since the translation forces the propagation of substitution, expressions of the polymorphic calculus that are not well-typed may have a well-typed translation. For instance, the expression $(\lambda^{x\in \mathbb{I}_1, y\in \mathbb{I}_2} x.3)\{\text{Int}^\text{t}_I\}$ is not well-typed (since the inner $\lambda$-abstraction is not) but for its translation we can deduce: $\vdash_C C[(\lambda^{x\in \mathbb{I}_1, y\in \mathbb{I}_2} x.3)\{\text{Int}^\text{t}_I\}] : \text{Int} \rightarrow \text{Int}$. To circumvent this problem, we restrict ourselves to values for which we can show that a value and its translation have the same minimal type:

**Lemma F.8.** Let $v \in \mathcal{Y}$ be a value. There exists a type $t$ such that

1. $\vdash v : t$ and for all $s$ if $\vdash v : s$ then $t \leq s$;
2. $\vdash_C C[v] : t$ and for all $s$ if $\vdash_C C[v] : s$ then $t \leq s$.

**Proof.** By induction on $v$.

\[\begin{align*}
\vdash v : t & \quad \Rightarrow \quad C[v] = v, \\
\vdash_C C[v] : t & \quad \Rightarrow \quad C[v] = v.
\end{align*}\]

We now want to show that the translation preserves the reduction, that is if an expression $e$ reduces to $e'$ in the polymorphic calculus, then $C[e]$ reduces to $C[e']$ in CoreCDuce. Prior to that we show a technical (but straightforward) substitution lemma.

**Lemma F.9.** Let $e$ be an expression, $x$ an expression variable and $v$ a value. Then $C[e\{v/x\}] = C[e]C[v/x]$.

**Proof.** By induction on $e$.

\[\begin{align*}
C[e\{v/x\}] & = C[e]C[v/x], \\
C[v/x] & = v, \\
C[v] & = C[v]C[x/v], \\
x & = C[x]C[v/x].
\end{align*}\]
Lemma F.10. If \( \Gamma \vdash e : t \) and \( e \leadsto e' \), then \( C[e] \leadsto^\pi C[e'] \). More specifically

1. if \( e \leadsto_{\text{Rinst}} e' \), then \( C[e] \leadsto C[e'] \);
2. if \( e \leadsto_{\text{R}} e' \) and \( (R) \neq \text{(Rinst)} \), then \( C[e] \leadsto C[e'] \).

Proof. By induction and case analysis on \( e \).

\( e, x \) or \( \lambda^\phi_{\mathcal{T}_x \rightarrow^\pi t} z. e_0 \): irreducible.

\( e_1, e_2 \): there are two ways to reduce \( e' \):

1. \( e_1 \leadsto e' \). By induction, \( C[e_1] \leadsto C[e'] \). Then we have \( (C[e_1], C[e_2]) \leadsto C[e'], (C[e_1], C[e_2]) \), that is, \( C[e_1, e_2] \leadsto C[e_1, e_2] \).

2. \( e_1 = v_1 \) and \( e_2 \leadsto e_2' \). By induction, \( C[e_2] \leadsto C[e'_2] \). Moreover, according to Lemma F.7, \( C[v_1] \in \mathcal{V}_C \). So we have \( (C[v_1], C[e_2]) \leadsto C[v_1], C[e_2] \), that is, \( C[[v_1, e_2]] \leadsto C[[v_1, e_2]] \).

We can now show that our translation preserves the reductions relation of the polymorphic calculus.

\[ \pi_t(e') : \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
\[ = C[t(\pi_t(e') \{ v/x \})] \]
\[ = C[\pi_t(e') \{ v/x \}] \]
Definition F.12. If $e_0 \leadsto e'$, By induction, $C[e_0] \leadsto_C C[e']$. Then we have $\pi_i(C[e_0]) \leadsto_C \pi_i(C[e'])$, that is, $C[\pi_i(e_0)] \leadsto_C C[\pi_i(e')]$.

(2) $e_0 = (v_1, v_2)$ and $e \leadsto v_i$. According to Lemma F.7 $C[(v_1, v_2)] \in \mathcal{Y}_C$. Moreover, $C[(v_1, v_2)] = (C[v_1], C[v_2])$. Therefore, $\pi_i(C[v_1], C[v_2]) \leadsto_C C[v_i]$, that is, $C[\pi_i(v_1, v_2)] \leadsto_C C[v_i]$.

e_1 \epsilon \mathcal{E}_2$: there are three ways to reduce $e$:

(1) $e_1 \leadsto e'_1$. By induction, $C[e_1] \leadsto_C C[e'_1]$. Then we have $C[e_1] \rightarrow C[e'_1]$, that is, $C[e_1] \leadsto C[e'_1]$.

(2) $e_1 = v_1$ and $e_2 \leadsto e'_2$. By induction, $C[e_2] \leadsto_C C[e'_2]$. Moreover, according to Lemma F.7 $C[v_1] \in \mathcal{Y}_C$. So we have $C[v_1] \rightarrow C[e'_1]$, that is, $C[v_1, v_2] \leadsto_C C[v_1, v_2]$.

(3) $e_1 = \lambda_{(j_1,j_2)}(x_0, e_1, e_2 = v_2 \in \mathcal{E}_2 \rightarrow (e_0 \in [\pi_j]_{j \in P}) \{v_2\}_{i \in J}$, where $P = \{ j \in J | \exists i \in I, \vdash v_2 : t_j \sigma_j \}$. Without loss of generality, we assume that $(e_0 \in [\pi_j]_{j \in P}) \notin \{v_2\}_{i \in J}$.

Then we have $C[e_1] = \lambda_{(j_1,j_2)}(x_0, \pi_j,t) \{ e_0 \in [\pi_j]_{j \in P} \} P \in \text{OrdSet}(I \times J)$.

(1) $e_0 \epsilon \mathcal{E}_t? e_1 : e_2$: By induction, $C[e_0] \leadsto_C C[e'_0]$. Then we have $C[e_0] \epsilon \mathcal{E}_t? C[e_1] : C[e_2] \leadsto_C C[e'_0] \epsilon \mathcal{E}_t? C[e_1] : C[e_2]$.

that is, $C[e_0 \epsilon \mathcal{E}_t? e_1 : e_2] \leadsto_C C[e'_0 \epsilon \mathcal{E}_t? e_1 : e_2]$.

(2) $e_0 = v_0, \vdash t : e \leadsto e_1$. According to Lemmas F.7 and F.6 $C[v_0] \in \mathcal{Y}_C$ and $\vdash_C C[v_0] : t$. So we have $C[v_0] \epsilon \mathcal{E}_t? C[e_1] : C[e_2] \leadsto_C C[e'_1]$. 

(3) $e_0 = v_0, \vdash v_0 : t \leadsto e_2$. By Lemma F.7 $C[v_0] \in \mathcal{Y}_C$. According to Lemma F.8 there exists a minimum type $t_0$ such that $\vdash v_0 : t_0$ and $\vdash_C C[v_0] : t_0$. It is clear that $t_0 \neq t$ (otherwise $\vdash v_0 : t$). So we also have $\vdash_C C[v_0] : t$. Therefore, $C[v_0] \epsilon \mathcal{E}_t? C[e_1] : C[e_2] \leadsto_C C[e'_2]$.

We can finally state the soundness of our translation:

**Theorem F.11.** If $\vdash t : C[e]$ and $C[e] \leadsto_C v_C$, then

1. $\exists v \in \mathcal{E} \vdash e \leadsto v \wedge C[v] = v_C$.
2. $\vdash_C v_C : t$.

**Proof.** Since $\vdash t : C[e]$, according to Theorem B.16 and B.15 there exists $v$ such that $e \leadsto v$ and $\vdash v : t$. By Lemmas F.10 and F.7 we have $C[e] \leadsto_C v_C$ and $C[v] \in \mathcal{Y}_C$. From the reduction rules in Figure 16 the reduction is deterministic. Therefore, $C[v] = v_C$. Finally, following Lemma F.6 we get $\vdash_C v_C : t$. 

In addition, according to Lemma F.10 the translations of an instantiation expression $e(\pi_j)_{j \in J}$ and its corresponding relabeling expression $e(\pi'_j)_{j \in J}$ are the same: this is because the relabeling only propagates type substitutions without “changing” expressions. Assume that all the relabelings ($\text{Rinst}$) in expressions are performed implicitly. Then we are in a calculus, called a normalized calculus, where expressions contain no instantiation sub-expressions.

**Definition F.12.** A normalized expression $n$ is an expression without any instantiation sub-expressions, which is defined:

$\forall n \in \mathcal{E}$

$n ::= e \mid x \mid (n, n) \mid \pi(n) \mid \lambda x. n \mid \lambda x_{i \in I} t_i. x.n \mid n \epsilon \mathcal{E} n$.

The set of all normalized expressions is denoted as $\mathcal{E}_N$.

Clearly, $\mathcal{E}_N$ is a proper subset of $\mathcal{E}$. Every expression can be translated into a normalized expression: just perform all the relabelings ($\text{Rinst}$). Moreover, $\mathcal{E}_N$ is closed under the reduction rules, and we can safely disregard ($\text{Rinst}$) since it cannot be applied. Then the normalized calculus also possess soundness property.

---

14 Without the restriction of context values, we can prove that the reduction satisfies congruence.
Therefore, restricting to the normalized calculus, the translation still possesses all the properties presented above. Note that we can use $C[e] \rightsquigarrow C'[e']$ instead of $C[e] \rightsquigarrow C'[e']$ in Lemma [F.10].

F.3 Current limits and improvements

The translation we present allows one to encode the relabeling operation using only basic monomorphic constructs and the polymorphic subtyping relation of Duce. While of theoretical interest, this "blunt" approach has, in our opinion, two important drawbacks, that we discuss now before presenting some possible optimizations and workarounds. First, it is clear that the number of type-cases generated for a function has, in our opinion, two important drawbacks, that we discuss now before presenting some possible constructs and the polymorphic subtyping relation of [3]. While of theoretical interest, this "blunt" approach

F.4 Improvements

We have shown how to compile our calculus to CoreDuce where abstractions are translated into a set of type case expressions. In this section, we hint at some possible improvements that alleviate the issues highlighted in the previous section. The key idea is of course to reduce the number of type cases for the translation of abstractions.

Single Substitution. When only one substitution is present in the label of an abstraction, it can straightforwardly be propagated to the body expression without any encoding:

$$C[λ^α_{(\tau \to \sigma)} x . e] = λ^{\alpha'_{(\tau' \to \sigma')}} x . C[e[\sigma_0[\alpha]]]$$

Global Elimination. As stated in Section [B.1] the type variables in the domain of any type substitution that do not occur in the expression can be safely eliminated (Lemma [B.9]) and so can redundant substitutions (Lemma [B.10]), since their elimination does not alter the type of the expression. This elimination simplifies the expression itself and consequently also simplifies its translation since fewer type cases are needed to encode the abstraction. For instance, consider the expression

$$\lambda^{\alpha \to \tau}_{([\tau_1 \to \sigma_1] | \{\tau_2 \to \sigma_2\})} x . x$$

(40)

Since $\beta$ is useless it can be eliminated, yielding $\lambda^{\alpha \to \tau}_{([\tau_1 \to \sigma_1] | \{\tau_2 \to \sigma_2\})} x . x$. Now since the two type substitutions are the same (the second one being redundant), we can safely remove one and obtain the simpler expression $\lambda^{\alpha \to \tau}_{([\tau_1 \to \sigma_1] | \{\tau_2 \to \sigma_2\})} x . x$. Finally since only one substitution remain, the expression can be further simplified to $\lambda^{\alpha \to \tau}_{([\tau_1 \to \sigma_1] | \{\tau_2 \to \sigma_2\})} x . x$.

Local Elimination. One can also work on the condition type in the generated type cases and eliminate those that are empty. Consider the translation of (40):

$$C[λ^{\alpha \to \tau}_{(\tau_1 \to \sigma_1) \cup \{\tau_2 \to \sigma_2\}} x . x] \equiv \lambda^{\alpha' \to \tau_1 \cap \sigma_1} x . x$$

Clearly, the second and third branches can never be used since they are excluded by the first branch and the last branch is trivially useless. Thus the translated expression is equivalent to $\lambda^{\alpha \to \tau}_{(\tau_1 \to \sigma_1) \cup \{\tau_2 \to \sigma_2\}} x . x$.

More generally, consider a branch $x \in t_P ? e_P$ in the translation of $C[λ^{\alpha \to \tau}_{(\tau_1 \to \sigma_1) \cup \{\tau_2 \to \sigma_2\}} x . e]$. If the type $t'_P = t_P \cap \bigwedge_{(i,j) \in I} \neg t_i \sigma_j$ is 0, then the branch never be used. So we can eliminate it. Therefore, the translation of an abstraction can be simplified as

$$\lambda^{\alpha \to \tau}_{(\tau_1 \to \sigma_1) \cup \{\tau_2 \to \sigma_2\}} x . \{x \in t_P ? C[e[\sigma_0[\alpha]]] \cap \mathbb{O}_\sigma \neq 0\}$$

No Abstractions. The reason for encoding an abstraction by a set of type cases is to simulate the relabeling. In each branch, a modified version of $e$ (the body of the original function) is created where nested lambda abstractions are relabeled accordingly. Obviously, if $e$ does not contain any lambda abstraction whose type variable need to be instantiated, then the type substitutions can be straightforwardly propagated to $e$:

$$C[λ^{\alpha \to \tau}_{(\tau_1 \to \sigma_1) \cup \{\tau_2 \to \sigma_2\}} x . e] = λ^{\alpha' \to \tau_1 \cap \sigma_1} x . C[e[\sigma_0[\alpha]]]$$

Union Rule. Consider the following union typing rule

$$\Delta \vdash \Gamma, (x : s_1) \vdash e : t$$

$$\Delta \vdash \Gamma, (x : s_2) \vdash e : t$$

$$\Delta \vdash \Gamma, (x : s_1 \cup s_2) \vdash e : t$$

(union)
(which is sound). This rules allows us to simplify the translation of abstractions by combining the branches with the same relabeling

\[
C[\lambda i \in I t_i \rightarrow s_i] = \lambda i \in I J t_i \sigma_j \rightarrow s_i \sigma_j \{ x \in t_P \ ? \ C[e @ (\sigma_j)_{j \in P}] \}_{P \in \text{OrdSet}(J)}
\]

where \( t_P = \bigwedge_{j \in P} (\bigvee_{i \in I} t_i) \sigma_j \). Notice how the \( t_i \)s are combined inside each \( t_P \). The size of the encoding now only exponential in \(|J|\).