Typing Communicating Component Assemblages

Michaël Lienhardt\(^3\), Vivien Quémé\(^2\), Alan Schmitt\(^1\), and Jean-Bernard Stefani\(^1\)

\(^1\) Institut National de Recherche en Informatique et en Automatique (INRIA)
\(^2\) Centre National de la Recherche Scientifique (CNRS)
\(^3\) Université Joseph Fourier (Grenoble I)

[michael.lienhardt, vivien.quema, alan.schmitt, jean-bernard.stefani]@inrialpes.fr

Abstract. Building complex component-based software architectures can lead to assemblage errors that are not captured by classical type systems of host programming languages such as Java, C++, or ML. These assemblage errors can occur in particular when using component-based frameworks for building communication subsystems and middleware, for instance when using incompatible protocol stacks in a client and a server. In this paper, we illustrate how a domain specific type system can be used to avoid assemblage errors in a component-based communication framework, called DREAM. Our approach relies on the definition of a small process calculus that captures the operational essence of the DREAM framework, and on the definition of a novel type system that combines rows à la Rémy with a form of process types à la Yoshida. Type inference is undecidable for this type system, but we present an algorithm for semi-inference which makes our approach usable in practice.

[Erratum]: In the paper, we gave a wrong definition of a minimal type. See section 5 for the valid definition.

1 Introduction

Building software systems from components has many benefits compared to less modular approaches [36]: easier design and development, easier adaptation, maintenance, and evolution. However, constructing a system from components can give rise to non-trivial assemblage errors, as has been noted e.g. in [18]. This is the case, in particular, with communication systems built with dedicated component-based frameworks such as Appia [22], Click [24], Coyote [4], DREAM [15], or Ensemble [37].

Such systems are generally distributed, comprise many components (sometimes called micro-protocols), and their assembly must obey non-trivial, mostly implicit, consistency rules, giving rise to subtle errors, such as e.g. the failure of a component in a protocol stack to handle an incoming message because of incorrectly structured data.

Dealing with assemblage errors in communication systems and middleware has already been approached in two ways. First, one can formally specify the
expected behavior of individual components and proving correctness properties on the assemblage using a theorem prover, e.g. as in Ensemble [18]. Or one can specify assemblage constraints (typically, component dependencies) in some architecture description language (such as e.g. Armani [23] or DCDL [34]), and automatically verifying the assemblage consistency as is proposed e.g. in Aster [12], Knit [32], or Plastik [13]. The former approach requires theorem-proving expertise, which is unlikely to be available for systems programmers, while the latter typically supports a limited set of architectural constraints that fail to address the behavioral errors we consider in this paper.

In this paper we advocate an approach that extends and complements the ADL-based approach to deal with non-trivial behavioral errors that may occur in ill-formed assemblages. Our approach involves: the definition of a specific process calculus that captures the operational essence of the target component assemblages; the definition of a domain-specific type system that ensures typable assemblages do not exhibit the targeted class of errors; and the definition of an inference algorithm that automates, or assists users in, typing assemblages. We illustrate this approach with the handling of certain assemblage errors that can occur when using the DREAM framework [15], however it is not limited to this framework only. For instance, the same calculus and type system could be readily applied to Click [24] and Coyote [4].

Technically, the paper makes the following contributions: (i) we define a novel type system combining extensible records with row polymorphism and process types; (ii) we prove that the type inference problem for our type system is undecidable; and (iii) we present a semi-inference algorithm that infers types for component assemblages based on user-provided type annotations.

The paper is organized as follows. Section 2 motivates and defines the process calculus used to specify the high-level behavior of DREAM components. Section 3 introduces the type system, proves its correction and subject reduction. Section 4 defines how we can encode the Post Correspondence Problem [28] into a type inference decision, thus proving that type inference is undecidable. Section 5 proposes an inference semi-algorithm, which may never stop for some programs. Section 6 presents our semi-inference algorithm. Section 7 discusses related work. Section 8 concludes the paper and outlines future lines of research.

2 A Component Calculus for DREAM Assemblages

We present in this section a process calculus whose purpose is to encompass the operational semantics of component assemblages that can be defined with the DREAM framework. This calculus combines features of the asynchronous \( \pi \)-calculus [5] (asynchronous communication and replication operator) and value-passing CCS [21]. Processes are structured as locality trees as in in the kell-calculus [35], to encode the structure of a component assemblage. Values exchanged on channels are extensible records [33] which can be routed depending on their inner structure.
Dream is built using the Fractal component model [6], which allows structuring systems as component hierarchies. Dream components communicate through interfaces, which are represented by channels in our calculus. Messages exchanged between components in Dream consist in extensible sequences of chunks (named pieces of data), and are modeled by records in our calculus. Finally, the Dream framework also comprises a library of components encapsulating functions and behaviors commonly found in communication systems and middleware. Examples include message queues, stampers, multiplexers, and routers, whose behaviors can all be encoded in the calculus. In particular, the presence of router components motivates the introduction in the calculus of the special routing process. Note that this calculus abstracts away features of the implementation language (Java) that are not relevant to message handling, such as e.g. the thread execution model and exception handling.

### 2.1 Abstract Syntax

The syntax of our calculus is presented Figure 1.

A component $b[D]$ is a locality with name $b$ and content $D$. $D_1 | D_2$ denotes the parallel composition of processes $D_1$ and $D_2$. Basic processes $B$ comprise the

![Fig. 1. Abstract Syntax](image-url)
null process 0, message actions $R$, receivers, and replications of basic processes. A receiver $e(x).(B_1 | \ldots | B_n)$ awaits messages on channel $e$. As in the $\pi$-calculus, $!B$ stands for the replication of process $B$. In the following, $e$ and $s$ range over channel names. A message action $R$ is either the sending of a message $M$ on a channel $e$, written $e(M)$, or a routing process $\text{IPre}(a,M,s_1,s_2)$. The latter tests whether the message $M$ has a field with label 'a'; if so it sends it on channel $s_1$; otherwise, it sends it on $s_2$. Messages comprise variables $x$, records $\{a_1 = M_1; \ldots; a_n = M_n\}$, constants $c$, and applications $(M M)$. Constants allow us to parameterize the calculus over a set of basic values and data type operations. We assume that the set of constants $C$ contains basic operations on records (i.e. accessing, adding, and removing a field), integers, and arithmetic operations. Values $v$ are fully computed messages. Finally, to simplify the presentation of the operational semantic and properties of the type system, we let $L$ stand for any construct $D$ or $M$ of our calculus.

2.2 Operational Semantics

The operational semantics of our calculus is defined by a reduction relation, denoted $\triangleright$, defined on closed terms. It is defined modulo a structural equivalence relation on process terms, evaluation contexts and constants semantic.

Structural equivalence. Formally, structural equivalence, written $\equiv$, is the smallest congruence satisfying the rules of Figure 2.

![Fig. 2. Structural Equivalence](image)

Evaluation Contexts. The reduction rules presented below use evaluation contexts, which are terms with a hole. Evaluation contexts $E_v$ are given by the following grammar:
\[
E_v ::= \begin{array}{ll}
\text{Evaluation context} & \text{Hole} \\
\{ M E_v \} & \text{Application} \\
\{ a_1 = E_v, a_2 = M_2, \ldots, a_n = M_n \} & \text{Record} \\
\pi(E_v) & \text{Message sending} \\
\text{IfPre}(a, E_v, s_1, s_2) & \text{Routing} \\
E_v \mid D & \text{Parallel composition} \\
b[E_v] & \text{Components}
\end{array}
\]

**Constants Semantics.** The set of constants, written \(C\), is defined as the disjoint union of \(C^p\), the set of primitive values, and \(C^f\), the set of basic functions. The relation \(\text{match}\) between elements of \(C^f\) and values specifies the valid arguments of each basic function. Finally, \(\text{eval}\) is a function from \(\text{match}\) into messages, giving the result of applying a basic function to a valid argument.

In the following, we write \(\text{match}(c, v)\) iff \((c, v) \in \text{match}\). The definition of \(\text{match}\) and \(\text{eval}\) for messages operators \(a + (a = v)\) and \(-a\) (respectively accessing, adding, and removing a chunk) are

\[
\text{match} \triangleq \left( \begin{array}{l}
\{ (a, \{ a = v, a_1 = v_1; \ldots; a_n = v_n \}) \mid 0 \leq n \} \\
\cup \{ (-a, \{ a = v, a_1 = v_1; \ldots; a_n = v_n \}) \mid 0 \leq n \} \\
\cup \{ (+a = v), \{ a_1 = v_1; \ldots; a_n = v_n \} \mid \forall 0 < i \leq n, a_i \neq a \}\end{array} \right)
\]

\[
\text{eval}(a, \{ a = v, a_1 = v_1; \ldots; a_n = v_n \}) \triangleq v \\
\text{eval}(-a, \{ a = v, a_1 = v_1; \ldots; a_n = v_n \}) \triangleq \{ a_1 = v_1; \ldots; a_n = v_n \} \\
\text{eval}(+a = v), \{ a_1 = v_1; \ldots; a_n = M_n \}) \triangleq \{ a = v; a_1 = v_1; \ldots; a_n = v_n \}
\]

**Reduction Rules.** The reduction relation is defined as the smallest relation that verifies the rules of Figure 3.

### 2.3 Examples

We present now some simplified examples of DREAM components.

**Multiplexer.** A multiplexer has three ports: two inputs \(e_1\) and \(e_2\) and one output \(s\). The two receivers executing in parallel within the component send every message they receive on output \(s\), thus multiplexing them on one output channel.

**Router.** A router also has three ports: one input \(e\) and two outputs \((s_1\) and \(s_2\). The routing is implemented using the 'IfPre' operator which upon the reception of a message \(M\), sent it on \(s_1\) or \(s_2\) depending on the presence of the field \('a'\) in \(M\).
Connector. A connector, or binding, is a process that links two or more ports. Assuming component $b_1$ can send messages on port $s$, and component $b_2$ can receive messages on port $e$, the connector $s(x).e(x)$ allows $b_1$ and $b_2$ to communicate by forwarding messages it receives on port $s$ to port $e$.

An assemblage. We present here a little assemblage that allows two different communication protocols to coexist.

```
Producer[!s(x).e(x)] | !s(x) | !o1(x).ip(x) | !o2(x).fsr(x)
| !s(x).ip(x) | !o2(x).ip(x) | IP_Protocol[ip(x).ip(x)]
| !s(x).fsr(x) | !o2(x).fsr(x) | FSR_Protocol[fsr(x).fsr(x)]

In this example, we have the IP protocol, encoded via the component IP_Protocol, and the protocol FSR [9] encoded via the component FSR_Protocol. We focus our encoding on the fact that IP needs an IP address, and FSR uses a time stamp TS for its diffusion algorithm.

Output messages are then sent on the channel $o$, and then routed depending on the protocol they use: a message having the IP field will be sent using the IP protocol, and one which doesn’t have it is then supposed to be sent using FSR. In our example, we have a producer component, named Producer which send messages for both protocols.
Such an assemblage can thus be very useful, yet it is unlikely to be well
typed using an ML-like type system. Informally, messages sent on $o$ have the
fields $\text{Val}$ and either $\text{IP}$ or $\text{TS}$. Thus, using row polymorphism and sub-typing,
we can characterize these messages at most with a type of the form \{ $\text{IP} : \bot; \text{TS} : \bot; \text{Val} : \text{Pre}(...); \text{Abs}^{(\text{IP, TS, Val})}$ \}. But such an input type is not enough for the
$\text{IP}_0\text{PROTOCOL}$ component, which need messages with the field $\text{IP}$ defined.

The following type system resolve this problem by using \textit{process types}.

3 Type System

We introduce in this section a type system that guarantees that configuration
errors, as presented next, do not occur for well-typed programs.

\textbf{Definition 1.} A program $D$ has an error iff either:

- There exist $E_v$, $v$, and $v'$ such that $D = E_v[(v \ v')]$ and $(v \ v') \notin \text{match}$.
- There exist $E_v$, $v$, $a$, $s_1$, and $s_2$ such that $D = E_v[\text{IfPre}(a, v, s_1, s_2)]$ with $v$
  not being a record.

Informally, this definition means a program error can be a message manipulation
error or a routing error. It is motivated, from a practical point of view, by
assemblage errors that occur most when using the \textsc{Dream} framework. \textsc{Dream}
components manipulate messages by adding a chunk to it, or by removing or
accessing one of its chunks. Such operations may fail when applied to messages
with the wrong structure, \textit{e.g.} not having the field the component has to access. A
\textsc{Dream} assemblage can typecheck correctly as a Java program, but still exhibit
run-time errors because of this message manipulation operations.

Our type system combines Rémyn’s \textit{rows} [33] (for typing messages) and
process types inspired in part by process types in the \textit{$\lambda\pi_v$}-calculus [40] and
sequence types for the \textit{\pi}-calculus [19] (for typing component assemblages). Fol-
lowing [40] and [19], our process types map channels to finite sets of \textit{message
types}. One may think that a first possible approach to define process types can
be to extend the type system for \textsc{Pict} [27] with rows. However, process types in
\textsc{Pict} maps channels to one \textit{message type} (in our case rows), which is too restric-
tive in presence of routing procedures. Indeed, upon reception of a message, a
routing procedure sends it on different channels depending on the message struc-
ture. As our types reveal this structure, a routing procedure should accept \textit{several
input types}, differing at least on the presence or absence of the field being routed
on. Using polymorphism or sub-typing may address this problem if the rest of
the type is uniform, mapping a channel to $\forall \kappa.\{a : \kappa; W^{(a)}\}$ or $\{a : \bot; W^{(a)}\}$. This
is however too strong a restriction in our case as we want to allow part of
a type to depend on the present or absence of another field. For instance, in the
assemblage $A$, a message can either have the field $\text{IP}$ or $\text{TS}$, but not both.

Our process types thus follow the design of used for the \textit{$\lambda\pi_v$}-calculus [40] and
the $\pi$-calculus [19]: each channel carries a finite, potentially empty, set of base
types. As our channels only carry messages, our base types are simply record
types. Process types are sets containing channel names and pairs of a channel name and a base type. For instance, the type \( e \cup s : (T_1) \cup s : (T_2) \) specifies that the channel \( e \) is defined but carries nothing, while \( s \) may carry messages of types \( T_1 \) and \( T_2 \).

Type Syntax The syntax of our base and process types is presented Figure 4.

Fig. 4. Type syntax

The set \( V^m \) consists of base type variables, ranged over by \( \eta \) and its variants. The sets \( V^l \), where \( l \) is a finite set of labels, consist of row variables whose instantiation cannot contain any label in the set \( l \). They are ranged over by \( \rho_l \) and their variants. The set \( V^k \) is used for the field presence syntactic definition. Its variables are ranged over by \( \kappa \) and its variants. Finally, the set \( V^s \) consists of variables for process types, ranged over by \( \zeta \) and its variants. When no precision is needed, we write \( \alpha \) or \( \beta \) a type variable of any of these sets, and \( V \) the set of all such variables.

Base Types. They comprise rows, functions and basic types constructors \( t(E_1, \ldots, E_n) \), which can include integers (the int type), strings (the string type) and other data types. A row is a family of labeled presence information, defining the structure of the typed record. For instance, the record \( \{a = 1; c = “record”\} \) is typed \( \{a : \text{Pre}(\text{int}); c : \text{Pre}(\text{string}); \text{Abs}^{(a,c)}\} \) : fields \( a \) and \( c \) are present and respectively contain an integer and a string. A row of unknown structure can
be associated to a row variable $\rho^l$, allowing us to type generic record operators using row polymorphism. Presence information is explicit to type operations on records. For instance, adding a field ‘a’ in a record is possible only if the record doesn’t already have this field present. Such operator can then be typed as $\forall \eta, \rho^{(a)} \cdot \eta \rightarrow \{ a : \text{Abs}; \rho^{(a)} \} \rightarrow \{ a : \text{Pre}(\eta); \rho^{(a)} \}$, the first argument being the message which will be put in the field ‘a’ added to the second argument. By convention, we often omit the label information on the empty row Abs$^l$ when the context is clear. For instance, we may write $\{ a : \text{Pre}(E); \text{Abs} \}$ for $\{ a : \text{Pre}(E); \text{Abs}^{(a)} \}$. Finally, we have type scheme $\forall \alpha. T$ which handle type and row polymorphism. To ease the definition of type scheme, we will write either $\forall \pi . E$ or $\forall \alpha_1, \ldots, \alpha_n . E$ for $\forall \alpha_1 \ldots \forall \alpha_n . E$.

**Process Types.** Process types declare the channels used by a process, and map them to finite, possibly empty, sets of base types. The type $\emptyset$ states that no channel is used by the typed process. The type $e$ declares channel $e$, and maps it to an empty base type set, whereas $e : (T)$ declare $e$, and maps it to the singleton $\{T\}$. A process type such as $e$ is typically used to define that the typed program uses the channel $e$, even if no message is sent on it. The union construct $S \cup S'$ declares all the channels in $S$ and $S'$, and maps them to the union of the mapping induced by $S$ and $S'$. For instance, the type $s \cup e : (T_1) \cup e : (T_2)$ declares the channels $s$ and $e$, maps $s$ to the empty set, and $e$ to $\{T_1, T_2\}$.

**Structural Equivalence.** Structural equivalence on types is defined as the smallest equivalence relation that satisfies the rules of Figure 5. This relation is similar the one defined for rows [33], extended to process types. It identifies process types defining the same channels and mapping them to the same sets.

Using the equivalence on process type, we can see that every type $S$ is equivalent to a type of the following form, where the $e_i$ may be mentioned several times:

$$\left( \bigcup_{i \in I} e_i : (T_i) \right) \cup \left( \bigcup_{j \in J} e_j \right) \cup \bigcup_{k \in K} \zeta_k$$

We then define:

- $dc(S) \triangleq \{ e_k \mid k \in I \cup J \}$: $dc(S)$ is the set of all the channels $S$ defines.
- $S(e) \triangleq \{ T_i \mid i \in I \land e_i = e \}$: $S(e)$ is the set $S$ maps to $e$.
- $S' \subset S \triangleq dc(S') \subset dc(S) \land \forall e \in dc(S'), S'(e) \subset S(e)$. This defines an inclusion predicate on channel and component types.

**Well-defined Types.** Finally, we constrain the base type we can construct from this syntax to be only well-defined types. Informally, such restriction allow only functional constants which generate a result which can either be predefined, or depending only on the message applied to it. Allowing only well-defined types is quite natural with the definition of the constant semantic, which states that they are simple functions on messages. Moreover, such restriction is needed for the inference algorithm, to ensure that all rows in a routing procedure have all their presence informations defined.
For instance, the type \( E \triangleq \{ a : \kappa ; \text{Abs} \} \) is not well defined, as \( \text{fv}^+(E) = \{ \kappa \} \).
3.1 Typing rules

Our type system is based on several constructions we introduce before presenting the typing rules.

Substitutions. Substitutions are commonly used to deal with polymorphism and to instantiate type schemes into monomorphic ones. In this work, they are a simple extension to multi-sorted variables, of the one defined in common type system [26].

**Definition 3.** A substitution \( \sigma \) is a function from type variables \( \mathcal{V} \) to types \( \tau \) such that:

- The set \( \{ \alpha \mid \sigma(\alpha) \neq \alpha \} \) is finite.
- \( \sigma(\eta) \) (resp. \( \sigma(\rho') \), \( \sigma(\kappa) \) and \( \sigma(\zeta) \)) is an element of \( E \) (resp. \( W \), \( K \) and \( S \)).

We extend naturally the definition of substitution on any type and type schemes.

In the following, we will note \( \text{dom}(\sigma) \) the set \( \text{dom}(\sigma) \), which is called the domain of \( \sigma \). We will also note \( \exists(\sigma) \) the set \( \bigcup_{\alpha \in \text{dom}(\sigma)} \text{fv}(\sigma(\alpha)) \), which correspond to the type variables in the image of \( \sigma \).

Derivation Relation. In the typing rules, we also use a derivation predicate, noted \( \Gamma \vdash T_2 \Leftarrow T_1 \) which states that \( T_1 \) can be derived, using instantiation and generalization, into \( T_2 \). Informally, such a statement means that \( T_1 \) is more general than \( T_2 \).

**Definition 4.** Given a typing environment \( \Gamma \) and two type schemes \( T \) and \( T' \), \( T' \) is derived from \( T \) for \( \Gamma \) (written \( \Gamma : T' \Leftarrow T \)) iff:

- there exists a set of type variables \( \pi \) with \( \pi \cap \text{fv}(\Gamma) = \emptyset \) and a type \( T_1 \) such that \( T = \forall \pi . T_1 \);
- there exists a set of type variables \( \pi' \) with \( \pi' \cap \text{fv}(\Gamma) = \emptyset \) and a type \( T_2 \) such that \( T' = \forall \pi'. T_2 \); and
- there exists a substitution \( \sigma \) with \( \text{dom}(\sigma) \subset \pi \) and \( \sigma(T_1) = T_2 \).

This definition for instance validates the statement \( \Gamma \vdash \forall \eta_2 . \{ a : \text{Pre}(\eta_2 \rightarrow \eta_2) ; \text{Abs} \} \Leftarrow \forall \eta_1 , \rho . \{ a : \text{Pre}(\eta_1) ; \rho \} \): we first instantiate \( \eta_1 \) into \( \eta_2 \rightarrow \eta_2 \) and \( \rho \) into \( \text{Abs} \), and then generalize \( \eta_2 \). Let remark that \( \eta_2 \) could not have been generalized it it was bound in \( \Gamma \), like in the \( T:\text{Gen} \) typing rule.

This predicate is used for instance in the \( T:\text{IfPre1} \) typing rule to ensure that the input message has its field ‘a’ defined.

Types for constants. The function \( \partial \) maps constants to closed types. For correctness to hold, it must follow some constraints that ensure that types are compatible with the behavior of the constants:

1. Types \( \partial(c) \) allow only valid arguments as arguments of a basic function.
2. The types resulting of the application of a basic function \( c \) with a good element \( v \) must be the type of the result of the application \( \text{eval}(c,v) \).
Formally, for all \( c \in C \) and \( v \), if \( \emptyset \vdash (c \ v) : T \), then \( c \in C' \), \( v \in \text{match}(c) \) and \( \emptyset \vdash \text{eval}(c,v) : T \) (let remark that \( \partial(c) \) is indeed used in this formula, to type \((c \ v)\)).

The following are valid types for message operators:

\[
\begin{align*}
\partial(\cdot) & \triangleq \forall \eta, \rho^{\cdot} \cdot \{ a : \text{Pre}(\eta) ; \rho^{\cdot} \} \rightarrow \eta \\
\partial(-\cdot) & \triangleq \forall \eta, \rho^{\cdot} \cdot \{ a : \text{Pre}(\eta) ; \rho^{\cdot} \} \rightarrow \{ a : \text{Abs} ; \rho^{\cdot} \} \\
\partial(+(\cdot)=0) & \triangleq \forall \eta, \rho^{\cdot} \cdot \{ a : \text{Abs} ; \rho^{\cdot} \} \rightarrow \eta \rightarrow \{ a : \text{Pre}(\eta) ; \rho^{\cdot} \}
\end{align*}
\]

The typing rules are presented Figures 7 and 8, which respectively handle typing messages and typing processes. Type judgments have the form \( \Gamma \vdash L : \tau \), where \( \Gamma \) is a typing environment, \( L \) is the typed construct, i.e. either \( M \) or \( D \), and \( \tau \) its type. Typing environments are defined as usual: they are mappings from calculus variables to base types:

\[
\Gamma ::= \emptyset \mid \Gamma ; x : T
\]

Typing messages. The typing rules for messages are inspired from [31]. Rule T:MESSAGE is direct, except from some technical side conditions on bound variables, which are needed to ensure that no free variable is bound by this rule.

Typing processes. The typing rules for processes can be seen as a set of constraints imposed on process types. For instance, the rule T:CHANNEL states that a type \( S \) for \( \tau(M) \) must contain \( \epsilon : (T) \) (where \( T \) is the type of \( M \)), but does not specify \( S \) in full. Another example is the rule T:ZERO: as 0 defines no channel, there is no constraint on its type. Such typing rules allow flexibility in the definition of the type of a process without requiring a sub-typing rule for processes [40].

The rule T:IFPRE1 is applied when the input message contains a field ‘\( a \)’: the message is then sent on \( s_1 \), which is represented in the rule by the side condition \( T \in S(s_1) \). The second rule is applied when the message doesn’t contain the field ‘\( a \)’: it requests the process type to map \( s_2 \) to a set containing \( T \).

The rule T:RECEIVER ensures, for each type associated with the input channel, that the type of the receiver process matches that of its body. Finally, note that components have no impact on the process types: they are merely a convenience to help structure processes.

### 3.2 Typing examples

In the following, we try to find a process type \( S \) for some processes (the typing environment is supposed empty).
Fig. 7. Typing rules for messages

Multiplexer. This multiplexer \texttt{Mult} introduced in Section 2.3 can admit several types, depending on the messages it has in input. For instance, $e_1 \cup e_2$ and $e_1 : (T) \cup e_2 : (T_1) \cup s : (T) \cup s : (T_1) \cup s : (T_2)$ are valid types for this program. One can for instance verify the second type using the typing rules \texttt{T:Parallel} and \texttt{T:Receiver} on the functions $e_1(x).s(x)$ with $x$ typed $T$, and $e_2(x).s(x)$ with $x$ first typed $T_1$ and then typed $T_2$.

Router. It may seem that the typing rules are too restrictive to capture the expressiveness of the ‘IfPre’ construct. However, consider what happens when typing a router program such as $e(x).\text{IfPre}(a,x,s_1,s_2)$. Because of rule \texttt{T:Receiver}, we have to consider all the message types mapped on channel $e$. Combining the rules \texttt{T:IfPre1}, \texttt{T:IfPre2} and \texttt{T:Receiver}, we can thus recover expected types for the routing process. For instance, one can verify that the following process type validate the router program above, using the rules \texttt{T:IfPre1}, \texttt{T:IfPre2} and \texttt{T/Receiver}:

$$S \triangleq e : (\{a : \text{Pre}(\text{int}); \text{Abs}^{(a)}\}) \cup e : (\{b : \text{Pre}(\text{int}); \text{Abs}^{(b)}\}) \cup e : (\{\text{Abs}^0\})$$

$$\cup s_1 : (\{a : \text{Pre}(\text{int}); \text{Abs}^{(a)}\}) \cup s_2 : (\{b : \text{Pre}(\text{int}); \text{Abs}^{(a)}\}) \cup s_2 : (\{\text{Abs}^0\})$$
Fig. 8. Typing rules for processes

Small assemblage. The assemblage $\text{Bip}$ introduced in Section 2.3 can be typed with the following process type:


It is interesting to note that assemblage $\text{Bip}$ cannot be well typed using an ML-like type system. Informally, messages sent on $o$ have the fields $\text{Val}$ and either $\text{IP}$ or $\text{TS}$. Thus, using row polymorphism and sub-typing, we can characterize these messages at most with a type of the form \{ $\text{IP}$ : $\bot$; $\text{TS}$ : $\bot$; $\text{Val}$ : Pre(...); $\text{Abs}$ \}. But such an input type is not enough for the $\text{IP}$ protocol component, which requires messages with the field $\text{IP}$ defined. For the same reason, such an assemblage cannot be well-typed in Pict.
3.3 Basic Properties of the Type System

This type system is sound with the current definition of errors, as stated by the correction and subject reduction theorems. The proof of the theorems in this paper can be found in the companion technical report [17].

**Theorem 1 (Correction).** Given a valid typing statement $\emptyset \vdash L : \tau$, $L$ has no error.

**Theorem 2 (Subject reduction).** Given a valid typing statement $\Gamma \vdash L : \tau$ and a valid reduction $L \triangleright L'$, there exists a type derivation of $\Gamma \vdash L' : \tau$.

3.4 Principal Types

We prove in this section that our type system doesn’t enjoy the principal type property, i.e. there exist programs which doesn’t have a principal type in the sense of Hindley [11]. Such property also implies that principal typing in the sense of Wells [39] doesn’t hold either for this type system.

**Preliminary definitions.** In our type system, we have two main constructions: message types, which correspond to classic ML types, and process types. In the following definitions, we adapt the definition of principal types to message types, and extend it to encompass process types.

**Definition 5.** A principal type for a term $M$ is a type $T$ such that:

1. There exists a type environment $\Gamma$ such that $\Gamma \vdash M : T$.
2. If $\Gamma' \vdash M : T'$ holds, then there exist a substitution $\sigma$ such that $T' = \sigma(T)$.

Unfortunately, such a definition cannot be extended as such on process types. Indeed, a set variable can be instantiated into any process type: as a result, a valid type for a given program can be instantiated into a non-valid type for the program. To address this problem, we forbid the possibility of set variable in a principal type, but propose a generalized substitution to still allow some form of process type manipulation.

**Definition 6.** A set substitution $\Sigma$ is a non-empty finite set of substitution. We define the application of set substitutions $\Sigma = \{ \sigma_i \mid i \in I \}$ on set types as:

$$\Sigma(S) = \bigcup_{i \in I} \sigma_i(S)$$

Let suppose given two set types $S_1$ and $S_2$. $S_1$ is more general than $S_2$ (written $S_1 \preceq S_2$) iff there exists a set substitution $\Sigma$ such that $S_2 \subset \Sigma(S_1)$.

**Definition 7.** Let suppose given a program $D$. A process type $S$ is a principal type for $D$ iff

- $fv(S) \cap \mathcal{V}^s = \emptyset$. 

There exists $\Gamma$ such that $\Gamma \vdash D : S$ holds;
for all $S'$ with $dc(S') \subset dc(D)$ and $\Gamma'$ such that $\Gamma' \vdash D : S'$ holds, we have $S \preceq S'$.

Our type system has the principal type property iff every typable program has a principal type.

We now exhibit a program which doesn’t have a principal type:

$$b[l e(x).\text{IfPre}(a, x, i, s) \mid i(x).\pi((x.a))]$$

This component returns the record under an arbitrary depth of $a$ label. For instance, if one sends $\{a = \{\}\}$ (resp. $\{a = \{b = 2\}; c = 3\}$) on the channel $e$, the component will return on the channel $s$ the message $\{\}$ (resp. $\{b = 2\}$).

Such a component admit valid types, like its minimal type $e \cup i \cup s$. However, it admit no principal type: let’s prove it by contradiction. Suppose there exist a typing environment $\Gamma$ and a process type $S$ such that $\Gamma \vdash D : S$ holds. We thus have $e \in dc(S)$. Let consider the two cases:

(i) $S(e) = \emptyset$: we clearly have that $S$ is not principal. Indeed, Let’s take $S' = e : (\{\text{Abs}\}) \cup i \cup s : (\{\text{Abs}\})$. It is easy to see that $\Gamma \vdash D : S'$ holds, and there is no substitution $\sigma$ such that $\{\text{Abs}\} \in \sigma(\emptyset)$.

(ii) Let now suppose $S(e) \neq \emptyset$. We define:

- A family of context: $E^{(0)}_a := [\_]$ \quad $E^{(n+1)}_a := \{a : \text{Pre}(E^{(n)}_a); W^{(a)}\}$.
- A function $d_a$ on record type schemes such that $d_a(T) = n$ iff there exists a set of variables $\pi$, a context $E^{(n)}_a$ and a type $E$ which has not the form 
  \quad $\{a : \text{Pre}(E'); W^{(a)}\}$ such that $T = \forall \pi. E^{(n)}_a[|E|]$.

Because the channel $e$ is the input channel of a routing procedure, it is evident that all $T \in S(e)$ has the form $\forall \pi. \{W\}$.

Let define $n = \max \{T \in S(e) | d_a(T)\}$ and take $T \in S(e)$ (we write $T = \forall \pi. E$) such that $d_a(T) = n$. We define $T' = \forall \pi. \{a : \text{Pre}(E); \text{Abs}\}$ and $S' = S \cup e : (T') \cup i : (T')$. Per construction, we have $\vdash D : S'$. As $S$ is a principal type, there exist $\sigma$ and $T_1 \in S(e)$ such that $T' = \sigma(T_1)$. As we have $d_a(T_1) < d_a(T')$ (per construction), $T_1 = \forall \pi. E_1$, where $E_1$ can either be of the form (with $n = d_a(T_1)$):

- $E^{(n)}_a[\{a_1 : K_1; \ldots; a_m : K_m; \rho\}]$ with $l = \{a_i \mid 1 \leq i \leq m\}, a \notin l$ and \quad $\sigma(\rho) = a : \text{Pre}(E'); W^{l\cup(a)}$.
- $E^{(n-1)}_a[\{a : \text{Pre}(a); W^{(a)}\}]$ with $\sigma(a) = \{a : \text{Pre}(E'); W^{(a)}\}$.

We can remark that if $\forall \pi. \{a : \text{Pre}(E); W^{(a)}\} \in S(e)$, then $\forall \pi. E \in S(e)$ (per definition of the type system). Thus, inductively, we can then see that $\forall \pi. E^{(n)}_a[|E|] \in S(e) \Rightarrow \forall \pi. E \in S(e)$. So, we have $\{a_1 : K_1; \ldots; a_m : K_m; \rho\}$ or $\alpha$ in $S(e)$, which is impossible: the routing procedure only accepts in input types where the field $a$ is defined (present or absent). Thus $S$ cannot be a principal type for $D$.

This implies that this program doesn’t have a principal type and thus, our type system doesn’t have the principal type property.
4 Inference’s undecidability

We propose in this section a construction showing that type inference in our setting is undecidable. Formally, we use some specific programs to encode instances of the post correspondence problem (PCP) [28], reducing this undecidable problem into a typing problem.

In this section, we first present the post correspondence problem. We then introduce the encoding of the problem by some operator, showing how we encode basic structures, like lists and words in our calculus. Finally, after describing how we encode an instance of PCP in our calculus, we show that such program is typable if and only if the instance has a solution.

As it is undecidable to tell if a PCP instance has a solution, it would then mean that it is undecidable to know whether or not a program is typable: the type inference would then be undecidable.

Definition of the PCP problem. The input of the problem consists of a finite lists \((u_1, v_1), \ldots, (u_n, v_n)\) of words pairs over some alphabet \(A\) having at least two symbols. A solution to this problem is a non-empty sequence of indexes \(i_1, \ldots, i_m\) with \(1 \leq i_k \leq n\) for all \(1 \leq k \leq m\), such that

\[
u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}\]

The decision problem then is to decide whether such a solution exists or not.

To simplify the encoding of PCP, we will in the rest of the section suppose that \(A = \{a, b\}\).

First encoding. Let first make the reasonable supposition that the set

\[A \uplus \{\text{first, second, third, tail, head, tmp}\}\]

is included into the set of field names. From these fields, we define some structures, like words, pairs, lists. We write \(\|k\|_D\) the dream program which encode the structure \(k\):

- \(\|\varepsilon\|_D \triangleq \{\}\) and \(\|d.u\|_D \triangleq \{d = \|u\|_D\}\), with \(\varepsilon\) being the empty word and \(d \in A\).
- \(\|(u, v)\|_D \triangleq \{\text{first} = \|u\|_D; \text{second} = \|v\|_D\}\) and \(\|(u, v, w)\|_D \triangleq \{\text{first} = \|u\|_D; \text{second} = \|v\|_D; \text{third} = \|w\|_D\}\).
- \(\|[]\|_D \triangleq \{\}\) and \(\|l :: t\|_D \triangleq \{\text{head} = \|l\|_D; \text{tail} = \|t\|_D\}\), with \([]\) and :: being the usual list constructors.

We also define the encoding operator into types \(\|\|_T\) :

- \(\|\varepsilon\|_T \triangleq \{\text{Abs}\}^\emptyset\) and \(\|d.u\|_T \triangleq \{d : \text{Pre}(\|u\|_T); \text{Abs}^{\{d\}}\}\), with \(\varepsilon\) being the empty word and \(d \in A\).
- \(\|(u, v)\|_T \triangleq \{\text{first} : \text{Pre}(\|u\|_T); \text{second} : \text{Pre}(\|v\|_T); \text{Abs}^{\text{first, second}}\}\). We also define the encoding for a triplet as \(\|(u, v, w)\|_T \triangleq \{\text{first} : \text{Pre}(\|u\|_T); \text{second} : \text{Pre}(\|v\|_T); \text{third} : \text{Pre}(\|w\|_T); \text{Abs}^{\text{first, second, third}}\}\).
\[ |[]|_T \triangleq \{ \text{Abs}^\emptyset \} \text{ and } ||l::t||_T \triangleq \{ \text{head} : \text{Pre}(|l||l||T) ; \text{tail} : \text{Pre}(|l||t||T) ; \text{Abs}^{\text{head}, \text{tail}} \}, \]

with \([]\) and :: being the usual list constructors.

Finally, if \(L = (k_i)_{1 \leq i \leq n}\) is a list, we will write \(\text{lts}(L)\) the set \(\{k_i \mid 1 \leq i \leq n\}\).

To ease the presentation of the program, we will identify the list \(L\) to the set \(\text{lts}(L)\), when the order of the elements in the list doesn’t matter.

4.1 Reduction of the problem

Let suppose given in this subsection a finite list of word’s pairs \((u_i, v_i)_{1 \leq i \leq n}\), raising an instance of the PCP problem. We suppose, without lose off generality, that for each \(1 \leq i \neq j \leq n\), \((u_i, v_i) \neq (u_j, v_j)\).

The program we construct from such a PCP instance is a semi-algorithm which search for a solution of the problem: it stops when there is a solution to the problem, or continues indefinitely to search for a solution when it doesn’t exists.

Technically, this program must be totally sequential in its execution, \(i.e\). all function must only produce \(R\) constructs. Indeed, when a program is constructed like this, each type carried by a channel corresponds to an actual message sent on it. We need such property to ensure that the type of our encoding really corresponds to the computation done by the program.

Our program is constructed in two parts:

1. The first part computes iteratively all the possible word pair \((u_{i_1} \ldots u_{i_n}, v_{i_1} \ldots v_{i_n})\), where \(i_k\) are valid indices.

2. The second part then test if the pair is a solution for the given instance of PCP. If it is, then the program stops, and if the word is not a solution, the testing part asks for another word to test.

The construction part

Let first introduce the definition of a useful set:

**Definition 8.** Let \(m \in \mathbb{N}\). We define the set of word pair \(L(m)\) as:

- \(L(0) = \{ (\varepsilon, \varepsilon) \} \) : the list with just one element, the pair of two empty words.
- \(L(m) = \{ (u_{i_1}u_{i_2}v_{i_2}, v_{i_1}v_{i_2}) \mid 1 \leq i_1 \leq i_2 \leq n \land (u, v) \in L(m - 1) \}\)

We previously stated that this part of the program computes all possible word pairs resulting of the concatenation of the pairs defining the instance of the problem. More precisely, it computes, given a input list \(L(m)\) the list \(L(m + 1)\).

Hence, it gives inductively, if it takes in parameter \(L(0)\), all the \(L(m)\) \((1 \leq m)\), and thus, all the pairs we wanted.

This program part, named \(D_n\), uses \(n\) auxiliary programs. Each of this program, named \(D_n(u_i, v_i)\) \((1 \leq i \leq n)\), computes the list \(\{(u_{i_1}u_iu_{i_2}v_i, v_{i_1}v_i) \mid (u, v) \in L(m)\}\) where \(L\) is the parameter of the program, \(i.e.\) \(L(m)\). Hence, merging all these lists result in the creation of \(L(m + 1)\) from the list \(L(m)\). As all these programs must be executed sequentially, the input of the program must carry some auxiliary informations:
A copy of the input list \( L \). Indeed, during the computation of the resulting list, the input one is destroyed. We then need a copy for the following computation.

A temporary list storing the result of the overall computation.

Hence, the input of a program \( D_a(u_i, v_i) \) is a triplet where the first element is the list used for the computation, and the other two are the auxiliary informations.

As these programs have the same structure, we will present a general definition \( D_a(u, v) \), where the words \( u \) and \( v \) can be instantiated to any words.

The program \( D_a(u, v) \). This program listen on the channel input \( u, v \). Upon receipt of a triplet structured as we defined, it takes inductively all the word pair \((u', v')\) in the first list, and add the pair \((u.u', v.v')\) to the resulting list. This will result by the merging of the initial temporary list with the list \( \{(u.u', v.v') \mid (u', v') \in L\} \).

Once the list is computed, the triplet is sent on the channel output \( u, v \).

As in the program, we compute a word of the form \( u.u' \) where \( u \) is defined, we need to define somehow a concatenation operator. As \( u \) is fixed, such thing is possible to encode in our calculus. We will use the notation \( \bullet \) for this operator, which can be defined inductively as:

\[
\{ \} \bullet M \triangleq M \quad \{d = M_1\} \bullet M_2 \triangleq \{d = (M_1 \bullet M_2)\} \quad \text{where } d \in A
\]

The program \( D_n \). This program waits for a list \( L(m) \) to be sent on its input channel. Upon receipt of such list, it executes every \( D_a(u_i, v_i) \) programs in sequence, which will computes the different parts of the list \( L(m+1) \) and automatically merge them. The resulting list is then sent on the channel output.

\[
D_n \triangleq \\
!\text{input}_{u,v}(x).\text{HPre}(\text{head, } x.\text{first } + (\text{tmp } = x), \text{loop}_{u,v}, \text{finish}_{u,v}) \\
| !\text{loop}_{u,v}(x).\text{tmp}_{u,v}(x.\text{tmp}) \\
| !\text{finish}_{u,v}(x).\text{output}_{u,v}(x.\text{tmp}) \\
| !\text{tmp}_{u,v}(x).\text{input}_{u,v}(\{\text{first } = x.\text{first}.\text{tail}; \\
\text{second } = x.\text{second}; \\
\text{third } = (\text{head } = \{\text{first } = ||u||_D \bullet x.\text{first}.\text{head}.\text{first}; \\
\text{second } = ||v||_D \bullet x.\text{first}.\text{head}.\text{second}; \\
\text{tail } = x.\text{third}\})\}
\]

The program \( D_n \). This program waits for a list \( L(m) \) to be sent on its input channel. Upon receipt of such list, it executes every \( D_a(u_i, v_i) \) programs in sequence, which will computes the different parts of the list \( L(m+1) \) and automatically merge them. The resulting list is then sent on the channel output.

\[
D_n \triangleq \\
!\text{input}(x).\text{input}_{u_1,v_1}(\{\text{first } = x; \text{second } = x; \text{third } = ||[]||_D\}) \\
| D_a(u_1, v_1) \\
| !\text{output}_{u_1,v_1}(x).\text{input}_{u_2,v_2}(\{\text{first } = x.\text{second}; \text{second } = x.\text{second}; \text{third } = x.\text{third}\}) \\
| \ldots \\
| D_a(u_n, v_n) \\
| !\text{output}_{u_n,v_n}(x).\exists(x.\text{third})
\]

The program \( D_a(u, v) \). This program listen on the channel input \( u, v \). Upon receipt of a triplet structured as we defined, it takes inductively all the word pair \((u', v')\) in the first list, and add the pair \((u.u', v.v')\) to the resulting list. This will result by the merging of the initial temporary list with the list \( \{(u.u', v.v') \mid (u', v') \in L\} \).

Once the list is computed, the triplet is sent on the channel output \( u, v \).

As in the program, we compute a word of the form \( u.u' \) where \( u \) is defined, we need to define somehow a concatenation operator. As \( u \) is fixed, such thing is possible to encode in our calculus. We will use the notation \( \bullet \) for this operator, which can be defined inductively as:

\[
\{ \} \bullet M \triangleq M \quad \{d = M_1\} \bullet M_2 \triangleq \{d = (M_1 \bullet M_2)\} \quad \text{where } d \in A
\]

The program \( D_n \). This program waits for a list \( L(m) \) to be sent on its input channel. Upon receipt of such list, it executes every \( D_a(u_i, v_i) \) programs in sequence, which will computes the different parts of the list \( L(m+1) \) and automatically merge them. The resulting list is then sent on the channel output.

\[
D_n \triangleq \\
!\text{input}(x).\text{input}_{u_1,v_1}(\{\text{first } = x; \text{second } = x; \text{third } = ||[]||_D\}) \\
| D_a(u_1, v_1) \\
| !\text{output}_{u_1,v_1}(x).\text{input}_{u_2,v_2}(\{\text{first } = x.\text{second}; \text{second } = x.\text{second}; \text{third } = x.\text{third}\}) \\
| \ldots \\
| D_a(u_n, v_n) \\
| !\text{output}_{u_n,v_n}(x).\exists(x.\text{third})
\]
The testing part. Now that we have a list of word pairs, we must test if it contains a solution for the instance of the problem. Our algorithm simply tests each pairs in the list, and succeed when it find a pair containing two identical words. If no such pair is found, it send the list on the channel eq\text{fail}, where it can be sent to the previous program, to construct another list to test.

Technically, we structure this program in two: one which define if a pair is a solution, and the global one, which apply the first one on each element of the list.

The program $D_{eq}$. This program listen on the channel input\text{eqp}, waiting for a word pair. It then test each word recursively, and send an acknowledgement on test\text{ok} if the two words are equal. Otherwise, the acknowledgement is sent on test\text{fail}.

Technically, the manipulated message must contain some auxiliary data, as the current $L(m)$ list, used in the rest of the program. This data is stored in the third field of the input message. The acknowledgement message is then constituted from this data.

$$D_{eq} \triangleq$$
$$!\text{input}_{eqp}(x).\text{IfPre}(a, x.\text{first} + (\text{tmp} = x), \text{test}_{ua}, \text{test}_{na})$$
$$|!\text{test}_{ua}(x).\text{IfPre}(a, x.\text{tmp}.\text{second} + (\text{tmp} = x.\text{tmp}), \text{test}_a, \text{eq}_{\text{err}})$$
$$|!\text{test}_{na}(x).\text{IfPre}(a, x.\text{tmp}.\text{second} + (\text{tmp} = x.\text{tmp}), \text{eq}_{\text{err}}, \text{test}_{tb})$$
$$|!\text{test}_{tb}(x).\text{IfPre}(b, x.\text{tmp}.\text{first} + (\text{tmp} = x.\text{tmp}), \text{test}_{ab}, \text{test}_{u: \varepsilon})$$
$$|!\text{test}_{u: \varepsilon}(x).\text{IfPre}(b, x.\text{tmp}.\text{second} + (\text{tmp} = x.\text{tmp}), \text{eq}_{\text{err}}, \text{test}_{eb})$$
$$|!\text{test}_{eb}(x).\text{IfPre}(b, x.\text{tmp}.\text{second} + (\text{tmp} = x.\text{tmp}), \text{eq}_{\text{err}}, \text{test}_{e: \varepsilon})$$
$$|!\text{test}_{e: \varepsilon}(x).\text{eq}_{\text{err}}(x.\text{tmp}.\text{third})$$

The program $D_{eqp}$. This program listen on the channel eq for a list of word pair (ideally a list $L(m)$). It then uses the program $D_{eq}$ to test the equality between each pair in the list. The program sends an acknowledgement on the channel eq\text{a}, if there exists a pair $(u, u)$ in the list. Otherwise, it sends the input list on the channel eq\text{fail}, to continue the computation.

The auxiliary data sent to the program $D_{eqp}$ with the word pair is constituted of the input list, and the current state of the processed list.
The total program The only work left is to assemble the two parts in a coherent program. As previously proposed, we send \(L(0)\) on the channel \(e\), which is the input channel of \(D_n\). We then take the resulting list of this part (made available on the channel \(s\)), put it in a pair, and send the result on eq. This pair is then processed by \(D_{eqp}\), which send the list on eq, if there is a solution, or \(eq_{fail}\) when we didn’t find one so far. We finally take the list present on \(eq_{fail}\), and send it on \(e\), for another round (computing \(L(m + 1)\), testing it, ......).

\[
D_{eqp} \triangleq \\
| !\text{eq}(x).\overline{\text{test}}(\{\text{first} = x; \text{second} = x\}) \\
| !\text{test}_\text{fail}(x).\overline{\text{test}}(\{\text{first} = x.\text{first}; \text{second} = x.\text{second}\}) \\
| !\text{test}_\text{ok}(x).\overline{\text{eq}}(\{\}) \\
| !\text{list}_{\text{r}}(x).\overline{\text{eq}_{\text{fail}}}(\{x.\text{tmp}.\text{second}\})
\]

This finalise the definition of our semi-algorithm which search for a solution of the PCP instance.

4.2 Properties of this program

Theorem 3. Let suppose the given instance of PCP has a solution. Then \(D_{pcp}\) is typable.

Theorem 4. Let suppose the given instance of PCP has no solution. Then \(D_{pcp}\) is not typable.

Finally, as it is undecidable to decide if an instance has a solution, it is undecidable to decide if the program constructed from this instance is typable. Hence, as an type inference algorithm must state whether or not a program is typable, such algorithm doesn’t exist.

5 Inference Semi-Algorithm

We propose in this section an inference semi-algorithm, \textit{i.e.} an algorithm which states for most programs if it has a type and computes it, but in some cases, doesn’t stop.
Technically, this semi-algorithm is based on a total type inference algorithm for $R$ constructs. Using this algorithm, we are able to compute the type of each message in the program, and from them, the types of the messages they will evolve into. The resulting typing is not principal, but minimal:

**Definition 9.** Let suppose given a typing environment $\Gamma$ and two set types $S_1$, $S_2$. We say that $S_2$ is derived from $S_1$ for $\Gamma$, written $\Gamma : S_2 \leftarrow S_1$ iff

- $dc(S_1) \subseteq dc(S_2)$.
- For all $e \in dc(S_1)$ and all $T \in S_1(e)$, there exists $T' \in S_2(e)$ with $\Gamma : T' \leftarrow T$.

**Definition 10.** Given a typing environment $\Gamma$ and a program $D$, we say that $D$ admits a minimal type $S$ for $\Gamma$ iff:

- $\Gamma \vdash D : S$ holds.
- For all $S'$ such that $\Gamma \vdash D : S'$, we have $\Gamma : S' \leftarrow S$.
- $fv(S) \setminus fv(\Gamma) = \emptyset$.

We finally note $\Gamma \vdash_m D : S$ when $S$ is a minimal type for $D$, $\Gamma$.

Let note that a minimal type is minimal in term of process type inclusion, while it is the most general in term of message types. Let also note that the last condition ensure that the minimal type is entirely generalized.

We first present the type inference algorithm for $R$ constructs, which computes a minimal type of the input program. We then present the inference semi-algorithm, called propagation algorithm, per analogy with the messages evolving during the program execution, and propagating through the whole program.

### 5.1 The inference for $R$ constructs

We propose here a total inference algorithm for $R$ constructs. Such an algorithm require no initial annotation from the programmer. It computes, when it exists, a valid type for the given calculus construction, and fails if the given construct is not typable. The algorithm we present here is constraint-based [25, 29, 1], and thus, works in two steps.

**Constraints generation.** First, our algorithm explore the structure of the program, in order to extract some constraints the types must verify. Basic functions have generic types which are instantiated, for instance before the typing rule $T:App$, to see if the parameter is valid, and to define the type of the result of the application. This instantiation is encoded via a equality constraint: let suppose given the type annotations $c : \forall \alpha.E \rightarrow E'$ and $M : E''$. The application $(c M)$ raises the constraint $E = E''$, meaning that the substitution we compute must unify $E$ with $E''$. This is the principle of the constraint generation.

A second kind of constraint is needed by the routing procedure. Indeed, as this step of the inference doesn’t compute types – only constraints – we cannot know
the structure of a message $M$. Thus, it is impossible for the inference algorithm, to know on which channel the program $\text{IfPre}(a, M, s_1, s_2)$ will send the message $M$. The only way the constraint generation algorithm has to represent the type of such program is a set type variable. Moreover, the instantiation of this set variable is defined by the presence or not of the field $a$ in the message $M$. The constraint we use to encode such a conditional instantiation are conditional constraints \[1, 29\]. Such constraints make use of $K$ variables to represent the unknown state of the field the routing procedure is based on.

Constraint resolution. The purpose of this algorithm is to compute one substitution satisfying the constraints resulting of the constraint generation algorithm. Then, using this computed substitution, we would apply it on the set type the constraint generation algorithm affected to the program. The result of this application would be a valid type for the program. Indeed, the constraint encode all the substitutions needed for the different applications and routing procedure in the program. And as the computed substitution validate the whole constraint, it stand for all these needed substitutions.

Specifically, a constraint can be viewed as a finite set of equality and conditional constraints. We can easily define a substitution validating a equality constraint. By combining these different substitutions, we then can have enough knowledge of the types of the different messages to compute the substitutions standing for the routing procedures. We can thus compute the substitution corresponding to all the simple constraints in the set, and combine them to obtain the wanted substitution.

Constraints We present in this section the constraints, central constructs of the $R$ inference algorithm, and their semantic.

Constraint definition. We saw that we have two kinds of constraint: equality, and conditional constraints. The equality constraint is the usual equality predicate $=\text{ being applied on two message types: } E = E'$. In order to present the constraint resolution algorithm, we also define equality constraints on $K$ constructs. We have two constructs for the conditional constraints, one for each case we presented in introduction: when the field $a$ is present, and when it is absent. Each case imposes an equality constraint on set types, in order to define the set type of the routing procedure.

$$C_s ::= \begin{array}{ll}
\text{Simple constraint} & E = E' \mid K = K' \text{ Equality constraint} \\
\mid & K = \text{Pre?}\ S = S' \text{ Present conditional constraint} \\
\mid & K = \text{Abs?}\ S = S' \text{ Absent conditional constraint}
\end{array}$$

A conditional constraint of the form $K = \text{Pre?}\ S = S'$ means that if $K$ is present (there exists $E$ such that $K = \text{Pre}(E)$), then the set types $S$ and $S'$ must be equal. A similar meaning stand for the other conditional constraint.
We then present the constraint as a finite set whose elements are *simple* constraints:

\[
C ::= \text{true} \quad \text{Empty set} \\
| \quad C_s \quad \text{Simple constraint} \\
| \quad C \land C \quad \text{Many constraints}
\]

the **true** construct stand for the empty set, meaning that any substitution verify the constraint in the set. The \( \land \) operator stand for the union of sets, meaning that a substitution validating the whole set must verify the two subsets.

**Constraint semantic.** Once the constraint computed from an input program, our algorithm computes a substitution *satisfying* this constraint. We propose in this paragraph the definition of a relation \( \models \) between constraints and substitutions satisfying them.

**Definition 11.** A substitution \( \sigma \) satisfy the simple constraint \( C_s \) (written \( \sigma \models C_s \)) iff either:

- \( C_s = (E = E') \), and \( \sigma(E) = \sigma(E') \).
- \( C_s = (K = K') \), and \( \sigma(K) = \sigma(K') \).
- \( C_s = K = \text{Pre?} \ S = S' \), and \( \sigma(K) = \text{Pre}(E) \) for some \( E \) implies that \( \sigma(S) = \sigma(S') \).
- \( C_s = K = \text{Abs?} \ S = S' \), and \( \sigma(K) = \text{Abs} \) implies that \( \sigma(S) = \sigma(S') \).

This previous definition is quite intuitive. For the equality constraint, the syntactic predicate = is replaced by a actual syntactic equality between the two terms \( \sigma(E) \) and \( \sigma(E') \). As the two other constraints are conditional, their meaning is an implication, meaning that the syntactic equality of types is required only if the condition is fulfill. We then extend the \( \models \) relation on a constraint:

**Definition 12.** A substitution \( \sigma \) satisfy the constraint set \( C \) (written \( \sigma \models C \)) iff either:

- \( C = \text{true} \).
- \( C = C_s \), and we have \( \sigma \models C \).
- \( C = (C_1 \land C_2) \) and \( \sigma \) satisfy both \( C_1 \) and \( C_2 \).

A constraint \( C \) is satisfiable (written \( \models C \)) iff there exists a substitution \( \sigma \) such that \( \sigma \models C \).

**Constraint generation** The constraint generation algorithm computes two informations:

1. The *basic* type of the program in parameter As we make no computation on types at this stage of the algorithm, this type only describes the basic structure of the actual type of the program.
2. The constraint associated to the computed type. These constraints represents the properties of the computed type. From this constraint, the constraint resolution algorithm gives a substitution, which, applied to the basic type, will return a valid type of the program.

Technically, the constraint generation algorithm is constructed using rules of the form \( F, \Gamma \vdash L : [F'] \tau \mid C \), where:

- \( F \) and \( F' \) are finite sets of type variables. They represent as in [30] the sets of variables already used by our algorithm. \( F \) is the set used before the application of the rule, and \( F' \) after: it is \( F \) plus some variables used specifically by the rule. Such sets give us a formal definition of *fresh* variables.
- \( \Gamma \) is a typing environment. It define the known types of the message and store references used in the inferred construct.
- \( \tau \) is the *basic* type of the inferred construct. It is either a message type \( T \) or a set type \( S \), depending on the inferred construct.
- \( C \) is the generated constraint.

We now present the rules of our algorithm, which are structured in two parts: the one used for the \( M \) constructs, and the one generating constraint for the \( R \) constructs.

\[
\text{Fig. 9. The constraint generation for } M \text{ constructs}
\]
Constraint resolution  The algorithm we present in this section computes a substitution validating its input constraint. This computed substitution has several important properties, which are needed in some proofs. In the three following definitions, we present different functions on substitutions, the mgu definition and the total property.

Definition 13. Let suppose given a substitution $\sigma$. We write:

- $\text{dom}(\sigma)$ the set $\{\alpha \mid \alpha \in \mathcal{V} \land \sigma(\alpha) \neq \alpha\}$.
- $\mathcal{I}(\sigma)$ the set $\bigcup_{\alpha \in \text{dom}(\sigma)} \text{fv}(\sigma(\alpha))$.

Definition 14. Let supposes given a constraint $C$ such that $\vdash C$. We write $\llbracket C \rrbracket$ the set $\{\sigma \mid \sigma \vdash C\}$. We say that $\sigma \in \llbracket C \rrbracket$ is an mgu for $C$ (denoted by $\sigma = \text{mgu}(C)$) iff for all $\sigma_1 \in \llbracket C \rrbracket$, there exists $\sigma_1'$ such that $\sigma_1 = \sigma_1' \circ \sigma$.

Definition 15. Let suppose given a substitution $\sigma$ and a constraint $C$. We write $\sigma = \text{mgu}^i(C)$ iff

- $\sigma = \text{mgu}(C)$.
- $\mathcal{I}(\sigma) \cup \text{dom}(\sigma) \subset \text{fv}(C)$.
- $\sigma^2 = \sigma$.

Technically, the constraint resolution algorithm is presented using rules of the form $C \Rightarrow \sigma$ where $C$ is the input constraint and $\sigma$ is the computed substitution. The constraint we use in the algorithm are We structure the rules in two parts, the constraint resolution for equality constraints, and for conditional constraints.

From the shape of its rules, this algorithm indeed computes a substitution. The next theorem states that the computed substitution indeed have the properties we want.

Theorem 5 (Constraint resolution). Let suppose given a valid constraint generation statement $F, \Gamma \vdash L : \llbracket F' \rrbracket \tau \mid C$, where $\text{fs}(\Gamma) \subset F$. Then, the two following properties are equivalent:

i) $C$ is satisfiable.
ii) there exists $\sigma$ such that $C \Rightarrow \sigma$ and $\sigma = \text{mgu}^i(C)$.
Properties of the inference algorithm The two stages of the R construct inference are now defined. The constraint generation algorithm computes a basic type for the input program, and a constraint defining the inner structure of the computed type. The constraint resolution algorithm constructs a substitution from the constraint, which, applied to the basic type, gives an instantiated type for the input program.

This computed type is still not a valid type for the input program. Indeed, for the sake of the constraint generation algorithm, we replaced the type schemes with monomorphic types. In order to re-bind the variables in the computed type, we have a generalization operator, defined as follow.

Definition 16. Let suppose given a typing environment $\Gamma$. The generalization of a message type $T$ for $\Gamma$ (noted $\text{Gen}(\Gamma, T)$) is the unique type (modulo $\alpha$-conversion) $\forall (\text{fv}(T) \setminus \text{fv}(\Gamma)).T$. The generalization of a process type $S$ for $\Gamma$ (noted $\text{Gen}(\Gamma, S)$) is defined inductively by the following rules:
We now present two theorems stating the good properties of our inference algorithm.

**Theorem 6 (Correction).** Let suppose given:

- A valid constraint generation statement \( F, \Gamma \vdash L : [F'] \tau \mid C \), with \( \text{fv} (\Gamma) \subset F \), and \( \models C \).
- A valid constraint resolution \( C \Rightarrow \sigma \).

Then, there exists a type derivation of \( \sigma (\Gamma) \vdash L : \text{Gen} (\sigma (\Gamma), \sigma (\tau)) \).

**Theorem 7 (Completeness).** Let suppose given a valid typing statement \( \Gamma \vdash L : \tau \), and a set of type variable \( F \) such that \( \text{fv} (\Gamma) \subset F \). Then, there exists:

- A valid constraint generation statement \( F, \Gamma \vdash L : [F'] \tau' \mid C \), with \( \models C \).
- A valid constraint resolution \( C \Rightarrow \sigma \).

Moreover, there exist a permutation \( \sigma' \) such that:

- \( \sigma' \circ \sigma (\Gamma) = \Gamma \).
- \( \Gamma \vdash \tau \Leftarrow \text{Gen} (\Gamma, \sigma' \circ \sigma (\tau')) \).

**Corollary 1.** Let suppose given \( R, S \) and a typing environment \( \Gamma \) such that the typing statement \( \Gamma \vdash R : S \) is valid. Then, there exists a set type \( S' \) such that \( \Gamma \vdash_m R : S' \).

### 5.2 Propagation Algorithm

The principle of our type inference semi-algorithm (called propagation algorithm) is based on a property of a program’s minimal type. Indeed, a minimal type for a program \( D \) maps every channel \( e \) used by \( D \) to the set of the types of all messages which may be sent on \( e \). The purpose of this algorithm is then to compute the types of all messages which can be created during the program execution. This computation is done inductively, by adding to a partial type annotation a channel declaration corresponding to a message which may be sent in the program.

For instance, let consider the program \( D \equiv e \{ b = 'hi' \} e(x).s \langle x + (a = 1) \rangle \) and the partial annotation \( \emptyset \). This annotation can then be augmented into \( \emptyset \cup e : \{ b : \text{Pre(string)}; \text{Abs} \} \). To this type, we can also add \( s : \{ a : \text{Pre(int)}; b : \text{Pre(string)}; \text{Abs} \} \), which will give us the minimal type of the program.

This example presented a possible run of our propagation algorithm. The initial annotation is the empty set, to which we add the type of the different messages present in the program, or which might be created during its execution.
Preliminary definitions. The two next definitions present some notations used in the process type extension. The first one introduces a unified notation to reference sub-programs, and in particular, messages which may be sent on channels.

**Definition 17.** Let suppose given two programs \( D, D' \) and a finite family of pair \( (e_i, x_i)_{1 \leq i \leq n} \). We say \((e_1 : x_1; \ldots ; e_n : x_n/D') \subset D\), iff either:

- There exists \( E_v \) such that \( D = E_v[D'] \) and \( n = 0 \). In such case, we will write \((\emptyset/D') \subset D\),
- There exists \( E_v \) and \( D'' \) such that \( D = E_v[e_1(x_1). (D'')] \) and \((e_2 : x_2; \ldots ; e_n : x_n/D'') \subset D''\).

**Definition 18.** The input set of a \( D \) construct, written \( I(D) \), consists of the channel \( D \) is listening on.

\[
\begin{align*}
I(0) &= \emptyset  & I(\pi(M)) &= \emptyset  & I(\text{IfPre}(a, M, s_1, s_2)) &= \emptyset  \\
I(e(x).(B_1 \mid \ldots \mid B_n)) &= \bigcup_{1 \leq i \leq n} I(B_i) \cup \{e\}  & I(!B) &= I(B)  \\
I(D_1 \mid D_2) &= I(D_1) \cup I(D_2)  & I(b[D]) &= I(D) 
\end{align*}
\]

**Propagation algorithm.** Our propagation algorithm manipulates judgements of the form \( \Gamma \vdash D : S \), where \( \Gamma \) is a typing environment, \( D \) is the program whose type is being inferred and \( S \) is the partial annotation computed so far. This algorithm is defined as the transitive closure of the two propagation rules presented Figure 13. The first rule adds channel declaration corresponding to message which may be send in the program: it is the main rule of the algorithm. The second one is mainly technical and adds the input channel of receivers, as requested in the \( T:\text{RECEIVER} \) typing rule.

**Propagation Properties** The propagation algorithm being only a type inference semi-algorithm, we cannot present a correction and completeness theorem for it. Hence, we propose here just one theorem, stating how it behaves for typable programs.

This theorem is based on the definition of a propagation error, which states when the propagation algorithm fails, and of a terminal statement, presenting a successful computation.

**Definition 19.** Let suppose given a typing statement \( \Gamma \vdash D : S \) (let note that we don’t suppose that this statement is valid). We say that a propagation error occurs on this statement iff there exist:
Fig. 13. The propagation algorithm

- A sub-program \( (e_1 : x_1; \ldots; e_n : x_n/R) \subset D \).
- A tuple \( (T_1, \ldots, T_n) \in S(e_1) \times \cdots \times S(e_n) \).
- A valid statement \( \text{fe}(\Gamma) \cup \text{fe}(S), \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash R : [F^\prime] S' \mid C \quad C \Rightarrow \sigma \)

\[
\text{Gen}(\sigma(\Gamma), \sigma(S^\prime)) \not\subset \sigma(S) \vee (\exists \alpha_1 \neq \alpha_2 \in \text{fe}(\Gamma) \cup \text{fe}(S), \sigma(\alpha_1) = \sigma(\alpha_2) \vee \sigma(\alpha_1) \not\in V) \]

\[
\Gamma \vdash D : S \rightarrow \sigma(\Gamma) \vdash D : (\sigma(S) \cup \text{Gen}(\sigma(\Gamma), \sigma(S^\prime))) \]

\[
P : \text{CHANNEL} \quad \exists \varepsilon \in \text{I}(D) \setminus \text{dc}(S) \]

\[
\Gamma \vdash D : S \rightarrow \Gamma \vdash D : S \cup \varepsilon
\]

Definition 20. Let suppose given a statement \( \Gamma' \vdash D : S' \). We say that this statement is terminal iff no propagation error occurs on it, and no propagation rule can be applied.

We can then present the main property of our propagation algorithm:

Theorem 8. Let suppose given a valid typing statement \( \Gamma \vdash D : S_k \). Then there exist a terminal statement \( \Gamma' \vdash D : S' \) such that \( (\Gamma, \emptyset) \rightarrow^* \Gamma' \vdash D : S' \). Moreover, the statement \( \Gamma' \vdash D : S' \) holds, and there exist a permutation \( \sigma \) such that \( \Gamma = \sigma(\Gamma') \) and \( \Gamma \vdash S_k \leftarrow \sigma(S') \).

From this result, we can then state that:

Corollary 2. Let suppose given a valid statement \( \Gamma \vdash D : S \). Then there exist \( S' \) such that \( \Gamma \vdash m D : S' \) holds.

Case of non-Termination Per construction, the propagation algorithm will not finish if there is an infinite number of message type to compute, i.e. when an infinite number of messages with a different structure are sent during the program execution. The following program has such a property:

\[
!e(x).\pi(\{a = x\}) \mid \pi(1)
\]

Indeed, at first, the message present on \( e \) is ‘1’ typed int, but after one execution of the receiver, the message will be \( \{a = 1\} \) typed \( \{a : \text{Pre(int)}; \text{Abs}\} \). Thus, the propagation algorithm will compute inductively types of the form \( \{a : \text{Pre}(\ldots \{a : \text{Pre(int)}; \text{Abs}\} \ldots)\}; \text{Abs} \} \) and never finish, because no message
operation error will occur, and each application of first propagation rule will compute a different type.

Let finally remark that this program has no valid type: informally, to handle the generality of the messages sent on \( e \), one should use a type of the form \( e : (\alpha) \), which is not well-defined.

## 6 Semi-Inference Algorithm

We present in this section an algorithm which computes, with the aid of some initial type annotation, the minimal type of its input program. The principle of this algorithm is quite simple: it computes iteratively, for every channel in the program, the set of message types it should be mapped to. But such principle is not that easy to implement. To illustrate how this algorithm works, let us suppose given a typing environment \( \Gamma \), a typable program \( D \), and some annotation \( S' \), and let \( S \) be the minimal type of \( D \) given the typing environment \( \Gamma \).

\( S' \) is essential to this algorithm, as it gives the informations on \( S \) our algorithm cannot compute (type inference being undecidable). Specifically, we request that \( S' \) maps at least one channel \( e \) for each loop in \( D \) to \( S(e) \), plus the input channels of the whole program. Using this initial annotation, we can compute iteratively the different sets to which \( S \) maps all channels in \( D \): the resulting process type for \( D \) is then \( S \).

Actually, our semi-inference algorithm is a little more flexible than that: asking the annotation given by the developer to be exactly a part of \( S \) is not practical. So we allow \( S' \) to map bigger sets to channels than \( S \) does. In such case, the semi-inference algorithm computes the \( S' \)-relative minimal type for \( D \), i.e. the minimal type of \( D \) which contains \( S' \). But to simplify the presentation of the algorithm, we will suppose until Section 6.3 that \( S' \) is a part of \( S \) and defines the specified channels.

The computation of the set mapped by \( S \) to a specific channel \( e \) is based on what process types mean: \( S(e) \) defines all message types which may be sent on \( e \). Now, consider a message \( M \) on \( e \): it can either be currently sent on \( e \), or it will be created during the program execution by a receiver \( B \) listening on a some other channel \( e' \). Thus \( S(e) \) depends on \( S(e') \) because of \( B \). In other words, the computation of \( S(e) \) will need to know \( S(e') \) and \( B \) to define the types of the output messages which will be sent on \( e \).

In this section, we first present how one can compute the dependency relation between two channels before introducing how our algorithm uses it to compute \( S \).

### 6.1 Dependency Graph

A channel \( e \) directly depends on channel \( e' \) when there exists a receiver \( B \) listening on \( e' \) which has an output on \( e \). We give in this section a formal definition of the dependency graph of a program, which extends the direct dependency relation. A dependency graph defines when there is a dependency between two
channels, and gives some auxiliary information to help the computation of the set mapped to a channel. From this definition, one will see that such a graph can easily be computed from the input program.

**Preliminary Definition.** We first introduce the definition of the dependency graph with two definitions. The first one presents the output channels of a function: such channels indeed depends on the input channel of the function.

**Definition 21.** The output set of a B construct, written O(B), is constituted of all the channels B can send a message on. This set is defined as follow:

\[
\begin{align*}
O(0) &= \emptyset \\
O(\pi(M)) &= \{s\} \\
O(\text{IfPre}(a, M, s_1, s_2)) &= \{s_1, s_2\} \\
O(e(x).R_1 | \ldots | R_n)) &= O(R_1) \cup \cdots \cup O(R_n)
\end{align*}
\]

The second definition presents the channel used in a program.

**Definition 22.** We write \(dc(D')\) the set of all channels used in the program \(D'\). Such a set can be defined inductively as:

\[
dc(B) = O(B) \cup I(B) \\
dc(D_1 \mid D_2) = dc(D_1) \cup dc(D_2) \\
dc(b[D'']) = dc(D'')
\]

Let’s finally recall the notation \((e_1 : x_1; \ldots; e_n : x_n/D') \subset D\), which means that \(D'\) is a sub-program of \(D\), within functions waiting on the channels \(e_1, \ldots, e_n\).

**Dependency Definition.** We consider given a program \(D\) and now define how to construct a dependency graph from \(D\).

**Definition 23.** Let \(e, e' \in dc(D)\) be two channel. We say that \(e\) depends on \(e'\) iff there exist \((e_1 : x_1; \ldots; e_n : x_n/R) \subset D\) such that \(e \in O(R)\) and \(e' = e_i\) for some \(1 \leq i \leq n\). We note \(V(D)\) the set of all pairs \((e, e')\) with \(e\) depending on \(e'\).

The vertices of the dependency graph are \(dc(D)\) and \(V(D)\) is the set of its edges. To simplify the computation of the type, we annotate every vertex \(e\) with the set of the sub-programs that may produce a message on \(e\).

**Definition 24.** We write \(\varphi\) the function having \(dc(D)\) as domain and returning a set of sub-programs, such that \((e_1 : x_1; \ldots; e_n : x_n/R) \in \varphi(e)\) iff:
\( (e_1 : x_1; \ldots; e_n : x_n/R) \subset D. \)

\(-\ e \in 0(R).\)

We finally present the dependency graph of \( D \):

**Definition 25.** The dependency graph of \( D \) is the tuple \((dc(D), V, \varphi)\). Such graph is noted \( D(D) \).

### 6.2 Semi-Inference Algorithm

Let \( \Gamma \) be a typing environment, \( D \) a typable program, \( S \) its minimal type, and \( S' \) an annotation. Given the dependency graph \( D(D) \), the computation of \( S \) goes as follows: we find a channel \( s \) for which we can compute \( S(s) \), compute it, add it to the type annotation, and iterate until we have \( S \). Such a channel \( s \) is characterized by two properties: it must not be defined in \( S' \) \((s \notin dc(S'))\), and \( S(s) \) must be computable, i.e. for all channel \( e \) on which \( s \) depends, \( e \) must be defined in \( S' \). When such conditions are met, we note \( D, S' \rightarrow s \):

**Definition 26.** Let suppose given a program \( D \), a set type \( S \), and a channel \( s \). Let’s write \((N, V, \varphi)\) the graph \( D(D) \). We write \( \vdash D : S \rightarrow s \) iff:

\(- s \in dc(D) \setminus dc(S). \)

\(- \text{For all } (e, s) \in V, we have } e \in dc(S). \)

**Channel Computation** As in the propagation algorithm, the actual computation of \( S(s) \) uses the \( R \)-inference algorithm (see Section 5.1). But in the contrary of the propagation algorithm, we compute here the type carried by one channel at a type. And because of the routing procedure, one cannot know before any computation on which channel a message might be sent.

For instance, let consider the program \( D \equiv \pi\{a = 2\} | !e(x).\text{IfPre}(a, x, s_1, s_2) \). Because the routing procedure might send a message on \( s_2 \), we have \( \varphi(s_2) = \{e : x/\text{IfPre}(a, x, s_1, s_2)\} \) \((dc(D), E, \varphi)\) being the graph \( D(D) \)). Thus, if one want to compute the set mapped to \( s_2 \), one must use the type \{\( a : \text{Pre}(\text{int}); \text{Abs}\)\} on the routing procedure, and then see that \( s_2 \) actually carries nothing: \( s_1 \) does.

We use the same principle in the computation of the set mapped to \( s \): we compute the output type of all constructs which might send on \( s \), and then extract the types only relevant to \( s \), using the following extraction procedure.

**Definition 27.** Let suppose given a process type \( S \) and a channel \( s \). We define inductively the process type \((S)[s]\) as:

\[(e)[s] \triangleq \emptyset \text{ when } e \neq s \quad (s)[s] \triangleq s \quad (e : (T))[s] \triangleq \emptyset \text{ when } e \neq s \]

\[(s : (T))[s] \triangleq s : (T) \quad (S_1 \cup S_2)[s] \triangleq (S_1)[s] \cup (S_2)[s] \quad (\alpha)[s] \triangleq \alpha\]
The computation of \( S(s) \) consists of giving a type to all the messages which may be sent on \( s \). These messages are represented by the sub-programs in \( \varphi(s) \), which are of the form \((e_1 : x_1; \ldots ; e_n : x_n/R)\), meaning that \( R \) is in a receiver listening on all the \( e_i \). Note that sub-programs of the form \((\emptyset/R)\) represent messages which are currently sent on \( s \).

The channel computation, presented Figure 14, is structured in two rules. The first one computes from the given annotation and the previous computation the type of a sent message. This rule uses statements of the form \( \Gamma \vdash R : (S, \sigma) \), which is a short-hand for:

- \( \text{ft}(\Gamma), \Gamma \vdash R : [F'] \mid C \).
- \( C \Rightarrow \sigma \).
- \( S \triangleq \text{Gen}(\sigma(\Gamma), \sigma(S')) \).

The resulting statement is of the form \( S, \Gamma \vdash (e_1 : x_1; \ldots ; e_n : x_n/R) : (S', \sigma) \), which means that the inferred type is \( S' \), and the inferred substitution is \( \sigma \).

The second rule then applies this first one on all sub-program in \( \varphi(s) \), thus computing the set mapped to this channel. The resulting statement has the form \( S, \Gamma, D \vdash s : (S', \sigma) \), meaning that the channel \( s \) can be mapped to \( S' \) in the program \( D \), given the annotation \( S \) and the typing environment \( \Gamma \). \( \sigma \) is the substitution resulting of the computation made during the \( R \)-inference.

\[
\begin{align*}
\{(T_1, \ldots , T_n) \mid 1 \leq i \leq m\} & \triangleq S(e_1) \times \cdots \times S(e_n) \quad \forall 1 \leq i \leq m, \sigma_i \stackrel{\circ}{\circ} \cdots \circ \sigma_1 \circ \text{id}(\Gamma; x_1 : T_1; \ldots ; x_n : T_n) \vdash R : (S, \sigma_i) \\
\{ (e_1 : x_1; \ldots ; e_n : x_n/R) \mid 1 \leq i \leq n\} & \triangleq \varphi(s) \\
\forall 1 \leq i \leq n, \sigma_i \circ \cdots \circ \sigma_1 \circ \text{id}(S, \Gamma) \vdash (e_1 : x_1; \ldots ; e_n : x_n/R) : (S, \sigma_i) \\
S, \Gamma, D \vdash s : (\bigcup_{1 \leq i \leq m-1} \sigma_m \circ \cdots \circ \sigma_{i+1}(S_i) \cup S_m, \sigma_m \circ \cdots \circ \sigma_1) \\
\end{align*}
\]

**Fig. 14.** Channel Computation

**Semi-inference** The final step into constructing the semi-inference algorithm is the iteration of the extension of \( S' \). This is done by taking the transitive closure of the relation \( \leadsto \) (annotation extension) presented Figure 15. As in the propagation algorithm, the relation \( \leadsto \) manipulates typing statements of the form \( \Gamma \vdash D : S \), where \( D \) is the program whose type is being inferred.\( \Gamma \) is the typing environment and \( S \) is the currently computed annotation.
Fig. 15. The semi-inference rule

We can remark that this algorithm always finishes: the annotation extension adds a channel \( e \) to \( S' \), which is not defined in \( S' \) and defined in \( D \). The set \( \text{dc}(D) \) being finite, the extension must stop at least when every channel in \( D \) has been added to \( S' \).

6.3 Semi-Inference Properties

Until now, we supposed that the program \( D \) was typable, and the annotation \( S' \) was a part of the minimal type \( S \). But the semi-inference algorithm can be applied in a more general setting: the only assumption we keep is that \( S' \) defines at least one channel per loop in \( D \) and its input channels, to initiate the annotation extension process:

**Definition 28.** Let suppose given a program \( D \), a set type \( S \), and let’s write \((N,V,\varphi)\) the graph \( \mathbb{D}(D) \). We say that \( S \) is a good annotation for \( D \), and write \( \mathcal{G}(D,S) \) iff:

- For all \( e \in \text{dc}(D) \) such that \( \{ (s,e) \mid s \in \text{dc}(D) \} \cap V = \emptyset, e \in \text{dc}(S) \).
- For all cycle \( e_1 \rightarrow \cdots \rightarrow e_n \rightarrow e_1 \) of the graph \( \mathbb{D}(D) \), there exists \( 1 \leq i \leq n \) such that \( e_i \in \text{dc}(S) \).

Usually, an inference algorithm terminating without error implies the validity of the computed type. It is not the case here, as the annotation extension always finishes, even when the given program is not typable, we provide a predicate which computes if the inferred type is indeed valid.

**Definition 29.** Let suppose given a semi-inference derivation \( \Gamma \vdash D : S' \rightsquigarrow^{*} \Gamma' \vdash D : S'' \). We say that this semi-inference statement is valid iff:

- \( \text{dc}(D) \subset \text{dc}(S'') \).
- For all \( e \in \text{dc}(S') \), all \( (e_1 : x_1; \ldots; e_n : x_n/R) \in \varphi(e) \) and all \( (T_1,\ldots,T_n) \in S''(e_1) \times \cdots \times S''(e_n) \), \( \Gamma'; x_1 : T_1; \ldots; x_n : T_n \vdash R : S'' \) holds.

Informally, this predicate tests the consistency of the loops of the program, one part being the annotation and the other the computed type.

Finally, one can remark that our semi-inference algorithm might compute a process type defining too much channel, i.e. channel not used by the program. This happens typically with a router using only one routing possibility, or when a
function is never used. Indeed, the unused output channel is nonetheless defined in the inferred type to state that the algorithm indeed computed its mapped set (which is empty). To nonetheless have a minimality property on the computed type, we propose a predicate which removes the unnecessary channel declaration

Definition 30. Let suppose given a program \( D \) and a process type \( S \). We define inductively the process type \((S)[D]\) as:

\[
\begin{align*}
(\emptyset)[D] & \triangleq \emptyset \\
(\zeta)[D] & \triangleq \zeta \\
(s)[D] & \triangleq \begin{cases} 
  s & \text{when } s \in I(D) \\
  \emptyset & \text{else}
\end{cases} \\
(e : (T))[D] & \triangleq e : (T) \\
(S_1 \cup S_2)[D] & \triangleq ((S_1)[D]) \cup ((S_2)[S])
\end{align*}
\]

We can now present the correction and completeness theorems for our semi-inference algorithm. The correction theorem states that a terminated annotation extension, which is valid, computes a valid type for the given program.

**Theorem 9 (Correction).** Let \( \Gamma \vdash D : S' \rightsquigarrow \ast \Gamma \vdash D : S'' \) be a valid semi-inference derivation. Then, there exists a type derivation of \( \Gamma \vdash D : (S'')[D] \).

The completeness theorem states that if there exists a valid type for the input program, based on the given annotation, the semi-inference algorithm will then succeed.

**Theorem 10 (Completeness).** Assume given a typing environment \( \Gamma \), a program \( D \) and two process types \( S, S' \) such that \( S \) is a valid annotation for \( D \), and \( dc(S) \cap dc(S') = \emptyset \) and \( \Gamma \vdash D : S \cup S' \) holds. Then there exists \( S'' \) such that \( \Gamma \vdash D : S \rightsquigarrow \ast \sigma(\Gamma) \vdash D : S'' \) is a valid semi-inference derivation. Moreover, we have \( \sigma(\Gamma) \vdash \sigma(S \cup S') \Rightarrow (S'')[D] \).

In this theorem, \( S \) represents the annotation given by the user, and supposed to be a part of the minimal type \( S_m \) of \( D \). We said that \( S \) should be a valid annotation to specify that it should define at least the channels our algorithm needs to work properly. To express that \( D \) is typable, we introduce an auxiliary process type \( S' \), which completes \( S \) into a valid type for \( D \). Then, we extend \( S \) with our semi-inference algorithm into \((S'')[D]\). \((S'')[D]\) has then two properties: it is a valid type for \( D \); and it can be derived into \( S \cup S' \): \((S'')[D]\) is thus the \( S \)-relative minimal type for \( D \), which implies when \( S \) is part of \( S_m \) that \((S'')[D] = S_m \).

7 Related work

Type systems checking architectural constraints or component assemblages have been the subject of various works in the past decade. For instance, the work done
on the Wright language [3] supports the verification of behavioral compatibility constraints in a software architecture. Work on ArchJava [2] uses ownership types to enforce communication integrity between components. Recent work develops behavioral contracts for component assembly [7], which is close to the notion of session types as developed e.g. in [41]. None of these type systems allow to capture the errors we deal with in this paper, due to incorrect message manipulation operations. The type system we propose in this paper is more related to the ones defined for PICT [27], the \( \pi \)-calculus [19] or the \( \lambda\pi \)-calculus [40], although with provision for extensible record types that these systems do not have. Type inference for distributed calculi has been studied for the Join-calculus [8], Mobile Ambients-like calculi [20], \( D\pi \) [16], which have an inference algorithm, and PICT, which has not. While the reasons for type inference undecidability in PICT are typically higher-order polymorphism and sub-typing, we believe that in our case it is more related to polymorphic recursion [10]. Indeed, undecidability in our case is caused by channels being mapped to a finite set whose cardinality is not constrained, thus allowing a form of polymorphic recursion in loops. Finally, one can consider the routing process present in the calculus as a weak form of type analysis [38] on rows.

8 Conclusion

This paper describes a novel approach to check component assemblages and introduces a type system along with its semi-inference algorithm ensuring that no message manipulation errors can occur in a valid configuration. We plan to extend our work in two different ways. First, we plan to add operations to the calculus that allow manipulating the component structure of an assemblage as in e.g. the Kell calculus [35], and consider additional errors due to reconfiguration operations. For instance, Yoshida [40] used dependent types and sub-typing to handle processes being carried on channel like other values. However, we expect difficulties for the inference problem in this extended calculus and type system (e.g. one can encode directly extensible records in the Kell calculus). Second, we also plan to consider not only messages errors, but also concurrency ones, such as deadlocks, as was investigated in specific calculi by Kobayashi [14].

References


A Type system

A.1 Preliminary results

In the following results, we use context of the following form, to handle replacement anywhere in a program.

\[
E_a ::= \begin{array}{ll}
\text{Evaluation Context} & \text{Hole} \\
\text{Application} & (E_a M) \ | \ (M E_a) \\
\text{Message} & \{a_1 = E_a; a_2 = M_2; \ldots; a_n = M_n\} \\
\text{Message sending} & s \langle E_a \rangle \\
\text{Routing procedure} & \text{IfPre}(a, E_a, s_1, s_2) \\
\text{Receiver} & e(x).(E_a | B_2 | \ldots | B_n) \\
\text{Replication} & !E_a \\
\text{Parallel composition} & E_a \ | \ D \\
\text{Components} & b[E_a]
\end{array}
\]

Indeed, a program changes during its execution, replacing parts with new ones. Using such context and the following lemma, we then just have to prove that the new part can be typed with the same type as the previous one to have subject-reduction.

**Lemma 1 (Replacement).** Let suppose given a typing environment \( \Gamma \), an evaluation context \( E_a \), two constructs \( L_1, L_2 \), two types \( \tau_1, \tau_2 \) and three typing derivation \( J_1, J_2 \) and \( J_3 \) with:

- \( J_1 \) is a sub-derivation of \( J_3 \).
- \( \Gamma \vdash L_1 : \tau_1 \) concludes the typing derivation \( J_1 \).
- \( \Gamma \vdash L_2 : \tau_1 \) concludes the typing derivation \( J_2 \).
- \( \Gamma \vdash E_a[L_1] : \tau_2 \) concludes the typing derivation \( J_3 \).

Then, there exists a typing derivation of \( \Gamma \vdash E_a[L_2] : \tau_2 \).

**Proof.** By induction on \( J_3 \):

- Case \( J_3 = J_1 \). We then have \( \tau_1 = \tau_2 \), which gives us the result.
- Case \( J_3 \) has the following form, with \( J_1 \) being a sub-derivation of \( J_k \): there exists \( E'_a \) such that \( E_a = (E'_a M_2) \): we have \( M_1 = E'_a[L_1] \).

\[
\frac{T:\text{App} \quad J^1_k \\ \Gamma \vdash M_1 : E \rightarrow E' \quad \Gamma \vdash M_2 : E} {\Gamma \vdash L : \tau}
\]

Let write \( M'_1 \) the message \( E'_a[L_2] \). Using the induction hypothesis, \( \Gamma \vdash M'_1 : E \rightarrow E' \) holds. We can thus apply the typing rule \( T:\text{App} \) to get the result.
– Case $\mathcal{J}_3$ has the following form, with $\mathcal{J}_1$ being a sub-derivation of $\mathcal{J}_2$: there exists $E'_a$ such that $E_a = (M_1'E'_a)$: we have $M_2 = E'_a[L_1]$.

\[
T:\text{App} \quad \frac{\mathcal{J}_1 \quad \mathcal{J}_2}{\Gamma \vdash M_1 : E' \rightarrow E' \quad \Gamma \vdash M_2 : E} \quad \Gamma \vdash L : \tau
\]

Let write $M'_2$ the message $E'_a[L_2]$. Using the induction hypothesis, $\Gamma \vdash M'_2 : E$ holds. We can thus apply the typing rule $T:\text{App}$ to get the result.

– Case $\mathcal{J}_3$ has the following form. We can suppose, with the structural equivalence, that $\mathcal{J}_1$ is a sub-derivation of $\mathcal{J}_2$. As in the previous cases, there exists $E'_a$ such that $E_a = \{a_1 = E'_a; a_2 = M_2; \ldots; a_n = M_n\}$; we have $M_1 = E_a[L_1]$.

\[
T:\text{Message} \quad \frac{\forall 0 < i \leq n, \frac{\mathcal{J}_1}{\Gamma \vdash M_i : \forall \alpha.T_i, E_i} \quad \forall i \neq j, a_i \neq a_j \land (\text{fv}(E_i) \cup \alpha_i) \cap \alpha_j = \emptyset}{\Gamma \vdash \{a_1 = M_1; \ldots; a_n = M_n\} : \tau_2}
\]

Let write $M'_1$ the message $E'_a[L_2]$. Using the induction hypothesis, we have $\Gamma \vdash M'_1 : \forall \alpha.T_i$. We can then apply the typing rule $T:\text{Message}$ to get the result.

– Case $\mathcal{J}_3$ has the following form: we have $M = E_a[L_1]$.

\[
T:\text{Inst} \quad \frac{\mathcal{J}}{\Gamma \vdash M : \forall \alpha.T} \quad \tau_2 \triangleq T\{E/\alpha\} \quad \Gamma \vdash M : \tau_2
\]

Let write $M'$ the message $E_a[L_2]$. Using the induction hypothesis, we have $\Gamma \vdash M' : \forall \alpha.T$. We can then apply the typing rule $T:\text{Inst}$ to get the result.

– Case $\mathcal{J}_3$ has the following form: we have $M = E_a[L_1]$.

\[
T:\text{Gen} \quad \frac{\mathcal{J}}{\Gamma \vdash M : T} \quad \alpha \notin \text{fv}(\Gamma) \quad \tau_2 \triangleq \forall \alpha.T \quad \Gamma \vdash M : \tau_2
\]

Let write $M'$ the message $E_a[L_2]$. Using the induction hypothesis, we have $\Gamma \vdash M' : T$. We can then apply the typing rule $T:\text{Gen}$ to get the result.

– Case $\mathcal{J}_3$ has the following form: there exists $E'_a$ such that $E_a = \pi(E'_a)$: we have $M = E'_a[L_1]$.

\[
T:\text{Channel} \quad \frac{\mathcal{J}}{\Gamma \vdash M : T} \quad T \in \tau_2(s) \quad \Gamma \vdash \pi(M) : \tau_2
\]

Let write $M'$ the message $E'_a[L_2]$. Using the induction hypothesis, we have $\Gamma \vdash M' : T$. We can then apply the typing rule $T:\text{Channel}$ to get the result.
– Case $J_3$ finishes either with the rule $T:\text{IFPRE}1$ or $T:\text{IFPRE}2$. Using the same approach as before, we get the result.

– Case $J_3$ has the following form, with $J_1$ being a sub-derivation of the $J_T^1$:

$T \in S(e)$: there exists $E'_a$ such that $E_a = e(x). (E'_a \mid B_2 \mid \ldots \mid B_n)$; we have $B_1 = E'_a|L_1$.

$$
\frac{
\forall T \in S(e), \quad \forall 1 \leq i \leq n, \quad \Gamma; x : T \vdash B_i : S
}{
\Gamma \vdash e(x). (B_1 \mid \ldots \mid B_n) : S
}
$$

Let write $L'$ the program $E'_a|L_2$. Using the induction hypothesis, we have for all $T \in S(e)$, $\Gamma; x : T \vdash L' : S$. We can then apply the typing rule $T:\text{RECEIVER}$ to get the result.

Let remark that using the structural equivalence of the parallel composition, this case covers every possibility of $B_i = E_a[L_1]$, $1 \leq i \leq n$.

– Case $J_3$ has the following form, with $J_1$ being a sub-derivation of $J_T^1$: there exists $E'_a$ such that $E_a = E'_a \mid D_2$; we have $D_1 = E'_a[L_1]$.

$$
\frac{
T:\text{PARALLEL}
}{
\frac{J_1^1 \quad \frac{J_1^2}{J_1}}{
\Gamma \vdash L : \tau_2
}
}\quad \frac{J_1}{
\Gamma \vdash D_1 : \tau_2
}\quad \frac{J_2}{
\Gamma \vdash D_2 : \tau_2
}
$$

Let write $D'_1$ the message $E'_a|L_2$. Using the induction hypothesis, $\Gamma \vdash D'_1 : \tau_2$ holds. We can thus apply the typing rule $T:\text{PARALLEL}$ to get the result.

Let remark that using the structural equivalence of the parallel composition, this case covers also the possibility of $D_2 = E'_a[L_1]$.

– Case $J_3$ has the following form: there exists $E'_a$ such that $E_a = !E'_a$; we have $B = E'_a[L_1]$.

$$
\frac{\vdash B : \tau_2}{
\Gamma \vdash !B : \tau_2
}
$$

Let write $B'$ the program $E'_a[L_2]$. Using the induction hypothesis, we have $\Gamma \vdash B' : \tau_2$. We can then apply the typing rule $T:\text{BANG}$ to get the result.

– Case $J$ finishes with the typing rule $T:\text{BOX}$. Using the same approach as before, we have the result.

**Lemma 2.** Let suppose given a typing environment $\Gamma$, a variable $x$, a message $M$, a type $T$, a program $L$ and a type $\tau$ such that:

- $\Gamma(x) = T$.
- $\Gamma \vdash M : T$.
- $M$ contains no occurrence of $x$.
- $\Gamma \vdash L : \tau$. 


Then $\Gamma \vdash L^{\{M/x\}} : \tau$ holds.

Proof. By induction on the number $n$ of occurrence of $x$ in $L$:

- $n = 0$. We have $L^{\{M/x\}} = L$, which gives us the result.
- $n = n' + 1$, with $0 \leq n'$. Per definition of occurrence, there exists an evaluation context $E_a$ such that $L = E_a[x]$. Using the lemma 1, the typing statement $\Gamma \vdash E_a[M] : \tau$ holds. Moreover, The number of occurrence of $x$ in $E_a[M]$ is $n'$. We can thus apply the induction hypothesis to get a typing derivation of $\Gamma \vdash (E_a[M])^{\{M/x\}} : \tau$. Let finally remark that $(E_a[M])^{\{M/x\}} = (E_a[x])^{\{M/x\}} = L^{\{M/x\}}$: we have the result.

Lemma 3. Let suppose given a typing environment $\Gamma$, a variable $y$, a type $T$, a program $L$ and a type $\tau$ such that $\Gamma; y : T \vdash L : \tau$ holds. Then, if $L$ has no occurrence of $y$, there exists a type derivation of $\Gamma \vdash L : \tau$.

Proof. By induction on the typing derivation of $\Gamma; x: T \vdash L : \tau$:

- Case T:VAR. Per hypothesis, we have $x \neq y$. We thus have $\Gamma(x) = (\Gamma; y: T)(x)$, which gives us the result.
- Case T:PRIMITIVE. Considering the definition of this typing rule, we have the result.
- Case T:ZERO. Considering the definition of this typing rule, we have the result.
- Case T:GEN. Using the induction hypothesis, we have $\Gamma \vdash M : T$. Moreover, as $\alpha \notin \text{fv}(\Gamma)$, we have $\alpha \notin \text{fv}(\Gamma)$. We can thus apply the typing rule T:GEN to have the result.
- Other cases: using the induction hypothesis, we have the result in all these cases.

A.2 Subject reduction

Theorem 11. If $\Gamma \vdash L : \tau$ is derivable, and $L \triangleright L'$, then $\Gamma \vdash L' : \tau$ holds.

Proof. We prove the result by case on the reduction rule applied.

- Case CONTEXT. This cases is evident with the lemma 1.
- Case APP: $L = (c M)$ for some $c$ and $M$: we have $L' = \text{eval}(c, M)$. By hypothesis we have on the types of a basic function, the type statement $\Gamma \vdash L' : \tau$. We have
  \[
  \frac{\Gamma \vdash M : T \quad T \in \tau(s_1)}{\Gamma \vdash L : \tau}
  \]
  We can then easily see with the typing rule T:CHANNEL, that $\Gamma \vdash \text{eval}(M) : \tau$ holds.
– Case IfAbs: $G = \text{IfPre}(a, M, s_1, S_2)$ with ‘$a$’ being an absent field of the message $M$. Hence, we have $L' = \overline{S_2}(M)$. Let note $J$ the tying derivation of $\Gamma \vdash L : \tau$. We have

$$
\begin{array}{c}
\frac{J'}{\Gamma \vdash M : T} \\
T \in \tau(s_2)
\end{array}
\Rightarrow
\frac{}{\Gamma \vdash L : \tau}
$$

We can then easily see with the typing rule $T:\text{CHANNEL}$, that $\Gamma \vdash \overline{S_2}(M) : \tau$ holds.

– Case Com1: $L = \overline{\pi}(v) \mid B_1 \mid \ldots \mid B_n) \mid D$. From the typing derivation of $\Gamma \vdash L : \tau$, we have the following properties:

• There exists a type derivation of $\Gamma \vdash \overline{\pi}(v) : S$. This implies the existence of $T$ such that $\Gamma \vdash v : T$ and $T \in S(e)$.

• There exists a type derivation of $\Gamma \vdash e(x).(B_1 \mid \ldots \mid B_n) : S$. This implies that for all $T \in S(e)$, for all $0 < i \leq n$ we have $\Gamma \cup \{x : T\} \vdash B_i : S$.

Thus, for all $0 < i \leq n$, the typing statement $\Gamma \cup \{x : T\} \vdash B_i : S$ holds. Using the lemma 2, $\Gamma \cup \{x : T\} \vdash B_i\{v/x\} : S$ holds for all $1 \leq i \leq n$. Using the lemma 3, $\Gamma \vdash B_i\{v/x\} : S$ holds for all $1 \leq i \leq n$. We can then apply $n$ times the typing rule $T:\text{PARALLEL}$ to get a typing derivation of $\Gamma \vdash (B_1 \mid \ldots \mid B_n)\{v/x\} : S$.

– Cases Com2 and Com3. Using the same approach as before, we get the result.

A.3 Correction

Theorem 12 (Correction). Let suppose given a typing environment $\Gamma$, a program $L$ and a type $\tau$ such that $\Gamma \vdash L : \tau$. Then $L$ hasn’t any error.

Proof. Suppose on the contrary that $L$ has an error:

– $D = E_r[(c v)]$. As $D$ is well-typed, so is $(c v)$. So, per definition of the constant’s type, $(c, v) \in \text{match}$, and $(c v)$ is not an error. Contradiction.

– $D = E_r[\text{IfPre}(v, a, s_1, s_2)]$. If this construct cannot be reduced, then $v$ is not a message. But as the program is well-typed, this construct must be typed with either $T:\text{IfPre1}$ or $T:\text{IfPre2}$, and both of this typing rules request that $v$ is a message. Contradiction.

A.4 Auxiliary Results

We present in this section other properties of this type system, which will be used to prove the properties of the different algorithm based on this type system.
**Well-defined Types.** The first result we propose here shows that a valid type derivation always concludes with a well-defined type. This result requires a little preliminary lemma:

**Lemma 4.** Let suppose given a type $\tau$ and a substitution $\sigma$. Then, we have $fv^+(\sigma(\tau)) = \bigcup_{\alpha \in fv^+(\tau)} fv(\sigma(\alpha))$.

**Proof.** By induction on $\sigma$:

- Case $\tau \triangleq \alpha$. The result is evident in this case.
- Case $\tau \triangleq \{W^0\}$. Per induction, we have $fv^+(\{W^0\}) = \bigcup_{\alpha \in fv^+(\{W^0\})} fv(\sigma(\alpha))$.
  As $\sigma(\{W^0\}) = \{\sigma(W^0)\}$, the result is evident.
- Case $\tau \triangleq E_1 \rightarrow E_2$. Per induction, we have $fv^+(\sigma(E_1)) = \bigcup_{\alpha \in fv^+(E_1)} fv(\sigma(\alpha))$ and $fv^+(\sigma(E_2)) = \bigcup_{\alpha \in fv^+(E_2)} fv(\sigma(\alpha))$. We thus have:
  $$fv^+(\sigma(E_1 \rightarrow E_2)) = fv^+(\sigma(E_1) \rightarrow \sigma(E_2))$$
  $$fv^+(\sigma(E_1)) \setminus fv^+(\sigma(E_2))$$
  $$\bigcup_{\alpha \in fv^+(E_1)} fv(\sigma(\alpha)) \setminus \bigcup_{\alpha \in fv^+(E_2)} fv(\sigma(\alpha))$$
  $$\bigcup_{\alpha \in fv^+(E_1)} fv(\sigma(\alpha))$$

- Case $\tau \triangleq t(E_1, \ldots, E_n)$. Per induction, for all $1 \leq i \leq n$ $fv^+(\sigma(E_i)) = \bigcup_{\alpha \in fv^+(E_i)} fv(\sigma(\alpha))$. We thus have:
  $$fv^+(\sigma(t(E_1, \ldots, E_n))) = fv^+(t(\sigma(E_1), \ldots, \sigma(E_n)))$$
  $$\bigcup_{1 \leq i \leq n} fv^+(\sigma(\alpha))$$
  $$\bigcup_{1 \leq i \leq n} \bigcup_{\alpha \in fv^+(E_i)} fv(\sigma(\alpha))$$
  $$\bigcup_{\alpha \in fv(t(E_1, \ldots, E_n))} fv(\sigma(\alpha))$$

- The other cases can then be easily derived from the ones presented.

**Lemma 5.** Let $\Gamma \vdash M : T$ be a valid typing statement (with $\Gamma$ having only meaningful types). Then $T$ is meaningful.

**Proof.** By induction on the typing derivation of $\Gamma \vdash M : T$:

- Case T:VAR. As $\Gamma(x)$ is supposed meaningful, we have the result.
- Case T:PRIMITIVE. As $\delta(c)$ is supposed meaningful, we have the result.
- Case T:APP. Using the induction hypothesis, the typing statements $\Gamma \vdash M_1 : E \rightarrow E', \Gamma \vdash M_2 : E''$ are meaningful. We thus have $fv^+(E') \subset fv^+(E) \subseteq \emptyset$. We can then conclude that $\Gamma \vdash (M_1 M_2) : E'$ is meaningful.
- Case T:MESSAGE. Using the induction hypothesis, for all $1 \leq i \leq n$, we have $fv^+(E_i) = \emptyset$. Using the definition of the function $fv^+$, we have $fv^+\{\{a_1 \colon Pre(E_1); \ldots; a_n \colon Pre(E_n); Abs^{a_1, \ldots, a_n}\} = \bigcup_{1 \leq i \leq n} fv^+(E_i) = \emptyset$. We thus have the result.
- Case T:INST. Using the induction hypothesis and writing $\forall \alpha. T = \forall \alpha. \forall \alpha'. E'$, we have $fv^+(E') = \emptyset$. Using the lemma 4, we have that $fv^+(E' \{E'_\alpha\}) = \emptyset$, which gives us the result.
- Case T:GEN. Using the induction hypothesis, $T$ is well-defined, which trivially implies that $\forall \alpha. T$ is; we have the result in this case.
Substitution stability (normal typing). We prove in this paragraph that typing is stable under substitution, i.e. if \( \Gamma \vdash L : \tau \) holds, i.e. if \( \Gamma \vdash L : \tau \) holds, then \( \sigma(\Gamma) \vdash L : \sigma(\tau) \) holds too. This result is proved in two steps: first we prove that \( \sigma(\Gamma) \vdash L : \sigma_{\alpha}(\Gamma)(\tau) \) holds, and then we extend it using the typing rules T:GEN and T:INST to get the result.

Lemma 6. Let suppose given a typing environment \( \Gamma \), a message \( M \) and a type \( T \) such that \( \Gamma \vdash M : T \) holds. Then, for all substitution \( \sigma \) for all permutation \( \sigma' \) with the following properties, the statement \( \sigma(\Gamma) \vdash M : \sigma \circ \sigma'(T) \) holds.

- \( \forall \alpha \in (fv(T) \setminus fv(\Gamma)), \sigma'(\alpha) \) is fresh.
- \( \sigma(\Gamma) \cap \text{dom}(\sigma') = \emptyset \).

Proof. By induction on the type derivation of \( \Gamma \vdash D : S \) (we note \( \sigma_k \triangleq \sigma \circ \sigma' \)):

- Case T:VAR. We have \( f_\sigma(\Gamma(x)) \setminus f_\sigma(\Gamma) = \emptyset \) if \( f_\sigma(\Gamma(x)) = \sigma(\Gamma(x)) \) then, Using the typing rule T:VAR, the statement \( \sigma(\Gamma) \vdash x : \sigma_k(\Gamma(x)) \) holds: we thus have the result.
- Case T:PRIMITIVE. As per hypothesis, we have \( f_\sigma(\emptyset) = \emptyset \), we have the result using the typing rule T:PRIMITIVE.
- Case T:MESSAGE. Using the induction hypothesis, for all \( 1 \leq i \leq n \) there exist a type derivation of \( \sigma(\Gamma) \vdash M_i : \sigma_k(\forall \pi_i.E_i) \). We suppose, using \( \alpha \)-conversion, that \( (\text{dom}(\sigma_k) \cup \text{dom}(\sigma_k)) \cap (\bigcup_{1 \leq i \leq n} \pi_i) = \emptyset \). We then have \( \sigma(\forall \pi_i.E_i) = \forall \pi_i.\sigma_k(E_i) \) for all \( 1 \leq i \leq n \). Using the typing rule T:MESSAGE, there exists a typing derivation of

\[
\sigma_k(\Gamma) \vdash \{a_1 = M_1; \ldots; a_n = M_n\} : \forall 1 \leq i \leq n \quad \pi_i \vdash \{a_1 : \text{Pre}(\sigma_k(E_1)); \ldots; a_n : \text{Pre}(\sigma_k(E_n)); \text{Abs}\}
\]

Finally, we can see that:

\[
\forall 1 \leq i \leq n \quad \pi_i \vdash \{a_1 : \text{Pre}(\sigma_k(E_1)); \ldots; a_n : \text{Pre}(\sigma_k(E_n)); \text{Abs}\}
= \forall 1 \leq i \leq n \quad \pi_i \vdash \sigma_k\{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}\}
= \sigma_k(\forall 1 \leq i \leq n \quad \pi_i \vdash \{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}\})
\]

- Case T:APP. Using the induction hypothesis, there exist a typing derivation of \( \sigma(\Gamma) \vdash M_1 : \sigma_k(E \rightarrow E') \) and \( \sigma_k(\Gamma) \vdash M_2 : \sigma(E) \). As \( \sigma_k(E \rightarrow E') = \sigma_k(E) \rightarrow \sigma_k(E') \), we can apply the typing rule T:APP to get a type derivation of \( \sigma(\Gamma) \vdash (M_1 M_2) : \sigma_k(E') \).
- Case INST. Using the induction hypothesis, there exists a type derivation of \( \sigma(\Gamma) \vdash \forall x.\tau \). Let write \( T = \forall x.\tau \) and define \( E_k = \forall x.\tau \). We can then use the typing rule T:INST with the type \( E_k \) to have the result.
- Case GEN. Using the induction hypothesis, there exists a type derivation of \( \sigma(\Gamma) \vdash M : \sigma_k(T) \). As \( \alpha \notin f_\sigma(\Gamma) \) there exists a fresh variable \( \beta \) such that \( \sigma'(\alpha) = \beta \). We can thus apply the typing rule T:GEN to have a type derivation of \( \sigma(\Gamma) \vdash M : \forall \beta.\sigma_k(T) \). Finally, we can see that:

\[
\forall \beta.\sigma_k(T) = \forall \beta.\sigma \circ \sigma'(T)
= \sigma(\forall \beta.\sigma'(T))
= \sigma \circ \sigma'(\forall \alpha.T)
\]
Lemma 7. Let suppose given a typing environment \( \Gamma \), a program \( L \) and a type \( \tau \) such that \( \Gamma \vdash L : \tau \) holds. Then, for all substitution \( \sigma \), the statement \( \sigma(\Gamma) \vdash L : \sigma(\tau) \) holds.

Proof. By induction on \( L \):

- Case \( T:Box \). Using the induction hypothesis, there exists a typing derivation of \( \sigma(\Gamma) \vdash D:\sigma_k(S) \). We can then apply the rule \( T:Box \) to get the result.
- Case \( T:Parallel \). Using the induction hypothesis, there exists a typing derivation of \( \sigma(\Gamma) \vdash D_1:\sigma_k(S) \) and \( \sigma(\Gamma) \vdash D_2:\sigma_k(S) \). We can then apply the rule \( T:Parallel \) to get the result.
- Case \( T:Zero \). This case is evident.
- Case \( T:Bang \). Using the induction hypothesis, there exists a typing derivation of \( \sigma(\Gamma) \vdash R:\sigma_k(S) \). We can then apply the rule \( T:Bang \) to get the result.
- Case \( T:Channel \). Using the induction hypothesis, there exists a typing derivation of \( \sigma(\Gamma) \vdash M:\sigma_k(T) \). As \( T \in S(e) \), we trivially have \( \sigma_k(T) \in \sigma_k(S) \). We can then apply the rule \( T:Channel \) to get the result.
- Case \( T:IfPre1 \) and \( T:IfPre2 \). Using the same approach as before, we have the result.
- Case \( T:Function \). The induction hypothesis gives us for all \( 1 \leq i \leq n \) and all \( T \in S(e) \) a typing derivation of \( \sigma(\Gamma):x:T \vdash B_i:\sigma_k(S) \). Moreover, we have \( T \in S(e) \iff \sigma_k(T) \in (\sigma(S))(e) \). Thus, for all \( 1 \leq i \leq n \) and all \( T \in (\sigma(S))(e) \) a typing derivation of \( \sigma(\Gamma):x:T \vdash B_i:\sigma(S) \). We can then apply the rule \( T:Function \) to get the result.
– Case T: FUNCTION. The induction hypothesis gives us for all \(1 \leq i \leq n\) and all \(T \in S(e)\) a typing derivation of \(\sigma(\Gamma; x : T) \vdash B_i : \sigma(S)\). Moreover, we have \(T \in S(e) \iff \sigma(T) \in (\sigma(S))(e)\). Thus, for all \(1 \leq i \leq n\) and all \(T \in (\sigma(S))(e)\) a typing derivation of \(\sigma(\Gamma); x : T \vdash B_i : \sigma(S)\). We can then apply the rule T: FUNCTION to get the result.

**Substitution stability (minimal typing).** The following lemma proves a difficult result: minimal typing is stable under substitution. Such a property is actually used a lot for both the propagation and semi-inference algorithms, where substitutions are applied all the time on the computed types.

**Lemma 8.** Let suppose given a typing environment \(\Gamma\), a program \(L\) of the syntactic definition \(M\) or \(R\), and a type \(\tau\) such that \(\Gamma \vdash^* \tau\). Then, for all substitution \(\sigma\), the statement \(\sigma(\Gamma) \vdash^* \sigma\) holds.

**Proof.** By induction on a typing derivation of \(\sigma(\Gamma) \vdash L : \tau\):

– Case T: VAR: we have \(\tau' = \sigma(\Gamma(x))\). Moreover, we can easily see that \(\tau = \Gamma(x)\). This implies that \(\sigma(\tau) = \tau'\), and as \(\Gamma \vdash T \iff T\) for all type \(T\), we have the result in this case.

– Case T: PRIMITIVE. We have \(\tau' = \emptyset(c)\). Moreover, we can easily see that \(\tau = \emptyset(c)\). As \(\Gamma \vdash T \iff T\) for all type \(T\), we have the result in this case.

– Case T: MESSAGE. As we have \(\Gamma \vdash^* \tau\), for all \(1 \leq i \leq n\), there exist \(E_i\) and \(\pi_i\) such that:

\[
\begin{align*}
\tau &= \forall \bigcup_{1 \leq i \leq n} \pi_i, \{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}^{a_1, \ldots, a_n}\} \\
\end{align*}
\]

Then we have \(\Gamma \vdash^* M_i : \forall \pi_i. E_i\) for all \(1 \leq i \leq n\). Indeed, let suppose given \(1 \leq k \leq n\) and \(T\) such that \(\Gamma \vdash E_k : T\). Let write \(\forall \beta. E\) the type \(T\). Using the structural equivalence on types, we can suppose that \(k = 1\). Using the \(\alpha\)-conversion, we can suppose that \((\text{fv}(E) \cup \beta) \cap \pi_i = \emptyset\) for all \(2 \leq i \leq n\). We can also suppose that \((\text{fv}(E_n) \cup \pi_i) \cap \beta = \emptyset\) for all \(2 \leq i \leq n\). Let note \(T'\) the type \(\forall \bigcup_{1 \leq i \leq n} \pi_i, \{a_1 : \text{Pre}(E); a_2 : \text{Pre}(E_2); \ldots; a_n : \text{Pre}(E_n); \text{Abs}^{a_1, \ldots, a_n}\}\).

Thus the typing statement \(\Gamma \vdash L : \forall \beta. \tau\) holds. Thus per hypothesis, there exist \(\tau'\) such that:

\[
\begin{align*}
\cdot \quad \text{dom}(\sigma(\tau')) &\subset \bigcup_{1 \leq i \leq n} \pi_i; \\
\cdot \quad \forall \beta. T' &= \forall \bigcup_{1 \leq i \leq n} \pi_i, \sigma'(\{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}^{a_1, \ldots, a_n}\}) \\
\end{align*}
\]

We can then see that \(\Gamma \vdash T' \iff \forall \text{fv}(E_i), \text{Abs}^{\text{Pre}(E_i)\ldots\text{Pre}(E_n)}\). We can then apply the induction hypothesis: For all \(1 \leq i \leq n\), we have \(\sigma(\Gamma) \vdash^* \sigma(\forall \pi_i E_i)\). Using the \(\alpha\)-conversion, we can suppose that \((\text{dom}(\sigma(\tau')) \cup \exists(\sigma)) \cap \bigcup_{1 \leq i \leq n} \pi_i = \emptyset\). Let write \(\forall \bigcup_{1 \leq i \leq n} \pi_i, \{a_1 : \text{Pre}(E_1); \ldots; a_n : \text{Pre}(E_n); \text{Abs}^{a_1, \ldots, a_n}\}\) the type \(\tau'\). We can suppose, using \(\alpha\)-conversion, that \(\bigcap_{1 \leq i \leq n} \pi_i \cap \text{fv}(\sigma(\Gamma)) = \emptyset\). For all \(1 \leq i \leq n\), there exists a substitution \(\sigma_i\) with:

\[
\begin{align*}
\cdot \quad \text{dom}(\sigma_i) &\subset \pi_i; \\
\cdot \quad E'_i &= \sigma_i \circ \sigma(E_i) \\
\end{align*}
\]

We can then define the substitution \(\sigma\) such that:
- Case $\mathbf{T:App}$. Let consider the types $\forall \alpha.E_1$ and $\forall \beta.E_2$ such that:
  - $\Gamma \vdash^* M_1 : \forall \alpha.E_1$ holds.
  - $\Gamma \vdash^* M_2 : \forall \beta.E_2$ holds.
  - $\alpha \cap \beta = \alpha \cap \text{fv}(\Gamma) \cap \beta \cap \text{fv}(\Gamma) = \emptyset$.
  - $\alpha \cap (\text{dom}(\sigma) \cup \exists(\sigma)) = \beta \cap (\text{dom}(\sigma) \cup \exists(\sigma)) = \emptyset$.

  Let also consider the constraint $C \triangleq E_1 = E_2 \rightarrow \gamma$ with $\gamma$ fresh. Then, for all substitution $\sigma'$ with $\sigma' \vdash C$ and $\text{dom}(\sigma) \subseteq \alpha \cup \beta \cup \{\gamma\}$, the statement $\Gamma \vdash L : \sigma'(\gamma)$ holds. Indeed, we have $\Gamma \vdash \sigma'[\alpha}(E_1) \Leftarrow \forall \alpha.E_1$ and $\Gamma \vdash \sigma'[\beta}(E_2) \Leftarrow \forall \beta.E_2$. As $\text{fv}(E_1) \subseteq \text{fv}(\Gamma) \cup \alpha$ and $\text{fv}(E_2) \subseteq \text{fv}(\Gamma) \cup \beta$, we have $\sigma'[\alpha}(E_1) = \sigma'[\beta}(E_2) = \sigma'(E_2)$. Using the typing rule $T:\text{Inst-Gen}$, we thus have a typing derivation of both
  - $\Gamma \vdash M_1 : \sigma'(E_1)$.
  - $\Gamma \vdash M_2 : \sigma'(E_2)$.

As $\sigma' \vdash C$, we have $\sigma'(E_1) = \sigma'(E_2) \rightarrow \sigma'(\gamma)$. We can thus apply the typing rule $T:\text{App}$ to get the result.

Moreover, if $\Gamma \vdash (M_1, M_2) : \forall \alpha.E$ holds, then there exist a substitution $\sigma'$ such that:
  - $\text{dom}(\sigma') \subseteq \alpha \cup \beta \cup \{\gamma\}$.
  - $\sigma' \vdash C$.
  - $E = \sigma'\gamma$.

Indeed, as $\Gamma \vdash (M_1, M_2) : \forall \alpha.E$ holds, there exists three substitutions $\sigma_1, \sigma_2, \sigma_3$ such that:
  - $\text{dom}(\sigma_1) \cup \text{dom}(\sigma_2) \subseteq \alpha$.
  - $\sigma_1(E_1) = E_k \rightarrow E'_k$.
  - $\sigma_2(E_2) = E'$.
  - $\text{dom}(\sigma_3) \subseteq \text{fv}(E'_k) \setminus \text{fv}(\Gamma)$ and $\sigma_3(E'_k) = E$.

We can then define $\sigma'$ as:
  - $\text{dom}(\sigma') \subseteq \text{dom}(\sigma_1) \cup \text{dom}(\sigma_2) \cup (\text{dom}(\sigma_3) \cap (\alpha \cup \beta)) \cup \{\gamma\}$.
  - $\sigma'(\alpha) = \sigma_3 \circ \sigma_1(\alpha)$ for all $\alpha \in \text{dom}(\sigma_1)$.
  - $\sigma'(\alpha) = \sigma_3 \circ \sigma_1(\alpha)$ for all $\alpha \in \text{dom}(\sigma_1)$.
  - $\sigma'(\alpha) = \sigma_3(\alpha)$ for all $\alpha \in (\text{dom}(\sigma_3) \cap (\alpha \cup \beta)) \setminus (\text{dom}(\sigma_1) \cup \text{dom}(\sigma_2))$.
  - $\sigma'(\gamma) = E$.

We then have

\[
\begin{align*}
\sigma'(E_1) &= \sigma_3 \circ \sigma_1(E_1) \\
&= \sigma_3(E_k \rightarrow E'_k) \\
&= \sigma_3(\sigma_2(E_2)) \rightarrow \sigma_3(E'_k) \\
&= \sigma'(E_2) \rightarrow E \\
&= \sigma'(E_2 \rightarrow \gamma)
\end{align*}
\]

We can then deduce that there exists $\sigma' = \text{mgu}(C)$, with $\text{dom}(\sigma') \subseteq \alpha \cup \beta \cup \{\gamma\}$ such that $\tau = \text{Gen}(\Gamma, \sigma'(\gamma))$.

Let now use the induction hypothesis: we have
Lemma 9 (Environment Stability). Let suppose given a typing environment $\Gamma$, a finite set of pair $\{(x_j : T_j) \mid 1 \leq j \leq m\}$, a program $L$ and a type $\tau$ such that:

$$\Gamma ; x_1 : T_1 ; \ldots ; x_m : T_m \vdash L : \tau$$

and a set of type variables $\overline{\alpha}$ such that:

- $\sigma(\Gamma) \vdash_{m} M_1 : \sigma(\forall \overline{\alpha}.E_1)$ holds.
- $\sigma(\Gamma) \vdash_{m} M_2 : \sigma(\forall \overline{\alpha}.E_2)$ holds.

Per hypothesis, we have $\sigma(\forall \overline{\alpha}.E_1) = \forall \overline{\alpha}.\sigma(E_1)$ and $\sigma(\forall \overline{\alpha}.E_2) = \forall \overline{\alpha}.\sigma(E_2)$.

Using the same approach as before, there exists a substitution $\sigma''$ and a set of type variables $\overline{\gamma}$ such that:

- $\text{dom}(\sigma'') \subset \overline{\alpha} \cup \overline{\beta} \cup \{\gamma\}$.
- $\overline{\alpha} \cap \text{fv}(\sigma(\Gamma)) = \emptyset$.
- $\sigma'' \neq \sigma(C)$.
- $\tau' = \forall \overline{\alpha}.\sigma''(\gamma)$.

As $\sigma'' \vdash \sigma(C)$, we have $\sigma'' \circ \sigma \models C$, which means that there exists a substitution $\sigma_k$ with $\sigma'' \circ \sigma = \sigma_k \circ \sigma'$. As $\sigma'$ is an mgu, we can suppose that $\text{dom}(\sigma_k) \subset (\text{dom}(\sigma) \cup \text{dom}(\sigma'')) \setminus \text{dom}(\sigma')$. Let define the substitution $\sigma'_k \triangleq \sigma''|_{\text{dom}(\sigma')}$:

- If $\alpha \in \text{dom}(\sigma)$, we have $\alpha \notin \overline{\alpha} \cup \overline{\beta} \cup \{\gamma\}$, which gives that $\sigma_k(\alpha) = \sigma(\alpha).$

  We thus have $\sigma_k(\alpha) = \sigma(\alpha) = \sigma'_k(\alpha)$.

- If $\alpha \in \text{dom}(\sigma'') \setminus \text{dom}(\sigma')$, we have $\alpha \in \overline{\alpha} \cup \overline{\beta} \cup \{\gamma\}$, which implies that $\sigma(\alpha) = \alpha$.

  Moreover, as $\sigma'$ is an mgu, we have $\alpha \notin \exists(\sigma')$. We thus have $\sigma'_k(\alpha) = \alpha = \sigma_k(\alpha)$.

- If $\alpha \in \text{dom}(\sigma'') \setminus \text{dom}(\sigma')$, we have $\alpha \in \overline{\alpha} \cup \overline{\beta} \cup \{\gamma\}$, which implies that $\sigma(\alpha) = \alpha$. Thus we can conclude $\sigma''(\alpha) = \sigma'' \circ \sigma(\alpha) = \sigma_k \circ \sigma'(\alpha) = \sigma_k(\alpha)$.

We can finally remark that

$$\sigma'_{k|\text{fv}(\sigma'' \circ \sigma(\gamma))} \cap \text{fv}(\sigma(\Gamma)) \subset \text{dom}(\sigma'') \cap \text{fv}(\sigma(\Gamma))$$

$$\subset (\overline{\alpha} \cup \overline{\beta} \cup \{\gamma\} \cap (\exists(\sigma) \cup \text{fv}(\Gamma)))$$

$$= \emptyset$$

This implies that we indeed have $\Gamma \vdash \tau' \leftarrow \sigma(\tau)$.

- Case T:Inst. Using the induction hypothesis, we have $\sigma(\Gamma) \vdash T \leftarrow \tau$ and $\text{fv}(\tau) \setminus \text{fv}(\Gamma) = \emptyset$. Thus, the derivation computed so far gives us the result.

- Case T:Channel. Per definition of $\Gamma \vdash S \leftarrow \tau$, we have $dc(\tau) = \{s\}$ and $\tau(s) = T$ with $\Gamma \vdash_{m} M : T$. Using the induction hypothesis $\sigma(\Gamma) \vdash_{m} M : \sigma(T)$, which trivially implies that $\sigma(\Gamma) \vdash_{m} \pi(M) : \sigma(\tau)$ holds.

- Case T:IfPre1 and T:IfPre2. Using the same approach as before, we have the result.

Environmental stability. The following lemma is also defined for the sake of both the propagation and semi-inference algorithms. In this lemma, we handle the fact that the typing environment in both algorithm are constantly modified.

Lemma 9 (Environment Stability). Let suppose given a typing environment $\Gamma'$, a finite set of pair $\{(x_j : T_j) \mid 1 \leq j \leq m\}$, a program $L$ and a type $\tau$ such that:

$$\Gamma ; x_1 : T_1 ; \ldots ; x_m : T_m \vdash L : \tau$$

holds.
Then, for all n-uplet \((T'_1, \ldots, T'_m)\) such that \(\Gamma \vdash T_j \iff T'_j\) and \(fv(T'_j) \subset fv(T_j)\) for all \(1 \leq j \leq m\), the typing statement
\[
\Gamma; x_1 : T'_1; \ldots ; x_m : T'_m \vdash L : \tau \quad \text{holds.}
\]

Proof. In the following, we will note \(\Gamma_1\) (resp. \(\Gamma_2\)) the typing environment \(\Gamma; x_1 \vdash T_1; \ldots ; x_m \vdash T_m\) (resp. \(\Gamma; x_1 : T'_1; \ldots ; x_m : T'_m\)).

By induction on the typing derivation of \(\Gamma; x_1 : T_1; \ldots ; x_n : T_m \vdash L : \tau\):

- Case T:VAR. We have two cases:
  1. Case \(x \in \Gamma\). This case is trivial, with the typing rule T:VAR.
  2. Case there exists \(1 \leq j \leq m\) such that \(x_j = x\). Per definition, we have \(\tau = T_j\). As \(\Gamma \vdash T_j \iff T'_j\), there exist \(\overline{\alpha}, \overline{\beta}, T,\) and \(\sigma\) such that:
     - \((\overline{\alpha} \cup \overline{\beta}) \cap fv(\Gamma) = \emptyset\).
     - \(\text{dom}(\sigma) \subset \overline{\alpha}\).
     - \(T'_j = \forall \overline{\alpha}.T\).
     - \(T_j = \forall \overline{\beta}.\sigma(T)\).

     Using the \(\alpha\)-conversion, we can suppose modulo permutation, that \(\overline{\beta} \cap fv(\Gamma) = \emptyset\). Then, let note \(\{\alpha_i \mid 1 \leq i \leq n\} \equiv \overline{\alpha}\) and \(E_i \equiv \sigma(\alpha_i)\). We can then use the typing rule T:VAR to get a derivation of \(\Gamma_2 \vdash x : T'_j\).

     Applying \(n\) times the rule T:INST, we then get a type derivation of \(\Gamma_2 \vdash x : \sigma(T)\). Then, if we note \(\{\beta_i \mid 1 \leq i \leq k\}\) the set \(\overline{\beta}\), we can apply the rule T:GEN \(k\) times to get the result.

- Case T:PRIMITIVE. This case is evident, as \(\overline{\sigma}(c)\) does not depend on the typing environment.

- Case T:MESSAGE. Using the induction hypothesis, for all \(1 \leq i \leq n\), there exist a type derivation of \(\Gamma_2 \vdash M_i : T''_i\). We can then apply the typing rule T:MESSAGE to get the result.

- Case T:APP. Using the induction hypothesis, there exist a type derivation of \(\Gamma_2 \vdash M : E \rightarrow E'\) and \(\Gamma_2 \vdash M' : E\). We can thus apply the typing rule T:APP to get the result.

- Case T:INST. Per induction, we have a type derivation of \(\Gamma_2 \vdash M : \forall \alpha.T\).

     Thus, using the typing rule T:INST, we have the result.

- Case T:GEN. Using the induction hypothesis, there exists a type derivation of \(\Gamma_2 \vdash M : T\). Moreover, as \(fv(T'_j) \subset fv(T_j)\) for all \(1 \leq j \leq m\), we have \(fv(\Gamma_2) \subset fv(\Gamma_1)\). Thus, we have \(\overline{\alpha} \notin fv(\Gamma_2)\), which implies that we can use the typing rule T:GEN to get the result.

- Case T:PARALLEL. Using the induction hypothesis, there exist a type derivation of \(\Gamma_2 \vdash D_1 : S\) and \(\Gamma \vdash D_2 : S\). We can then apply the typing rule T:PARALLEL to get the result.

- Case T:ZERO. As any set type types the 0 construct with no regard to the typing environment, we have the result.

- Case T:BANG. Using the induction hypothesis, there exists a type derivation of \(\Gamma_2 \vdash B : S\). We can then apply the typing rule T:BANG to get the result.
Case T:CHANNEL. Using the induction hypothesis, there exists a type derivation of $I_2 \vdash M : T$. As per hypothesis, we have $T \in S(s_1)$, using the typing rule T:CHANNEL, we have the result.

Case T:IFPRE1. Using the induction hypothesis, there exists a type derivation of $I_2 \vdash M : T$. Per hypothesis, we have $T \in S(s_1)$ and $T = \forall \sigma.\{a : \text{Pre}(E) ; W^{(a)}\}$ for some $\sigma$, E and $W^{(a)}$. We can then apply the typing rule T:IFPRE1 to get the result.

Case T:IFPRE2. Using the induction hypothesis, there exists a type derivation of $I_2 \vdash M : T$. Per hypothesis, we have $T \in S(s_2)$ and $T = \forall \sigma.\{a : \text{Abs} ; W^{(a)}\}$ for some $\sigma$ and $W^{(a)}$. We can then apply the typing rule T:IFPRE2 to get the result.

Case T:FUNCTION. As for all type $T$ and all typing environment $\Gamma'$, we have $\text{fn}(T) \subset \text{fn}(T')$ and $\Gamma' \vdash T \iff T$, the typing environment $\Gamma_1; x : T$ and $\Gamma_2; x : T$ validate the hypothesis of our lemma. We can then apply the induction hypothesis, to get for all $T \in S(e)$ and all $1 \leq i \leq n$ a type derivation of $\Gamma_2; x : T : B_i : S$. As per hypothesis, we have $e \in \text{dc}(S)$, we only have to apply the typing rule T:FUNCTION to get the result.

**Typing characterization.** The following lemma presents the main characterization of a program’s type: it correspond to all message which may be sent on the program channels. Indeed, typing a program is equivalent than taking all possible message in it, and typing them:

**Lemma 10.** Let suppose given a typing environment $\Gamma$, a program $D$ and a set type $S$. The two following propositions are equivalent:

1. The typing statement $\Gamma \vdash D : S$ holds.
2. $\Gamma(D) \subset \text{dc}(S)$ and for all $(e_1 : x_n; \ldots ; e_n : x_n/R) \subset D$, all $(T_1, \ldots , T_n) \in S(e_1) \times \cdots \times S(e_n)$, we have $\Gamma, x_1 : T_1; \ldots ; x_n : T_n \vdash R : S$.

**Proof.** The implication i) $\Rightarrow$ ii) is evident from the definition of the type system. Let prove the other implication by induction on the structure of $D$:

- Case 0. As 0 is always typable, this case is trivial.
- Case R. Per hypothesis, we have $\Gamma \vdash R : S$, which gives us the result.
- Case $e(x). (B_i | B_n)$. Let note $\{T_j \mid 1 \leq j \leq l\}$ the set $S(e)$. Per hypothesis, for all $1 \leq j \leq m$ and all $1 \leq k \leq l$ we have:

$$\forall (e_1 : x_n; \ldots ; e_n : x_n/R) \subset B_i, \quad \forall (T_1, \ldots , T_n) \in S(e_1) \times \cdots \times S(e_n), \quad \Gamma, x : T_k; x_1 : T_1; \ldots ; x_n : T_n \vdash R : S$$

Moreover, we have $((e_1 : x_n; \ldots ; e_n : x_n/R) \subset B_i) \Leftrightarrow ((e : x; e_1 : x_n; \ldots ; e_n : x_n/R) \subset D)$. We can then deduct that for all $1 \leq j \leq m$ and all $1 \leq k \leq l$, there exists a type derivation of $\Gamma; x : T_k \vdash B_i : S$. Finally, as $e \in \text{dc}(S)$, we can then use the typing rule T:RECEIVER to conclude.

- Cases !B, D_1 | D_2 and b[D]. Using the induction hypothesis and the proper typing rule, we trivially have the result in these cases.
Process types manipulation. We propose in this last paragraph a result manipulating process types.

**Lemma 11.** Let suppose given a typing environment $\Gamma$, a program $D$ and a set type $S$ such that $\Gamma \vdash D : S$ holds. Let also suppose given a channel $s \in dc(S) \setminus dc(D)$. Then, there exist a type derivation of $\Gamma \vdash D : S \setminus s$.

**Proof.** Let consider a sub-program $(e_1 : x_1; \ldots ; e_n : x_n/R) \subset D$. As $s \not\in dc(D)$, for all $1 \leq i \leq n$, we have $e_i \neq s$. This implies that $(S \setminus s)(e_1) \times \cdots \times (S \setminus s)(e_n) = S(e_1) \times \cdots \times S(e_n)$. Let then take a tuple $(T_1, \ldots , T_n) \in (S \setminus s)(e_1) \times \cdots \times (S \setminus s)(e_n)$. Using the lemma 10, there exists a type derivation of $\Gamma; x_1 : T_1; \ldots ; x_n : T_n \vdash R : S$. Let consider the two cases: either $R = \tau(M)$ or $R = \text{IfPre}(a,M,s_1,s_2)$. In both cases, there exists $T$ such that $\Gamma; x_1 : T_1; \ldots ; x_n : T_n \vdash M : T$, and $e' \in dc(R)$ such that $T \in S(e')$. As $e' \in dc(R)$, we have $T \in (S \setminus s)(e')$, which implies that $\Gamma; x_1 : T_1; \ldots ; x_n : T_n \vdash R : S \setminus s$ holds.

Finally, using the lemma 10, $\Gamma \vdash D : S \setminus s$ holds.
B R-Inference

B.1 Constraint resolution

Definition 31. In this definition, we will note \( Y_e \) a type construct of the form \( E, K \) or \( W^l \), or a constraint \( C \). We define on such constructs a function \( d \) defined inductively as:

\[
\begin{align*}
  d(\alpha = 1) & \quad d(\text{Abs}) = 1 & \quad d(E) = n & \quad d(K) = n & \quad d(W^l = n') = n + n' + 1 \\
  \text{Pre}(E) = n + 1 & \quad d(a : K; W^l) = n + n' + 1 \\
  d(W^\emptyset) = n & \quad d(E_i) = n_i & \quad d(s(E_1, \ldots, E_m)) = (n_1 + \cdots + n_m) + 1 \\
  d(Y) = n & \quad d(Y') = n' & \quad d(K = \text{Pre}? S = S') = n \\
  d(K = \text{Abs}? S = S') = n & \quad d(C) = n & \quad d(C') = n' \\
  d(K = \text{Abs}? S = S') = n & \quad d(C \land C') = n + n'
\end{align*}
\]

We finally define a strict ordering on \( Y_e \) constructs such that \( Y_e < Y'_e \) iff either:

- \#fv(\( Y_e \)) < \#fv(\( Y'_e \)).
- \#fv(\( Y_e \)) = \#fv(\( Y'_e \)) and \( d(Y_e) < d(Y'_e) \).

Theorem 13 (Constraint resolution correction). Let also suppose given a constraint \( C \), a substitution \( \sigma \) such that \( C \Rightarrow \sigma \) holds. Then \( \sigma \vdash C \).

Proof. By induction on the derivation of \( C \Rightarrow \sigma \):

- Cases true, \( \alpha = \alpha, \alpha = Y \) with \( \alpha \not\in \text{fv}(Y) \), Abs = Abs. These cases are evident.
- Cases Abs = Abs? \( \alpha = S \) with \( \alpha \not\in \text{fv}(S) \), Pre(\( E \)) = Abs? \( \alpha = S \) with \( \alpha \not\in \text{fv}(S) \). Still evident.
- Cases Abs = Pre? \( \alpha = S \) with \( \alpha \not\in \text{fv}(S) \), Pre(\( E \)) = Pre? \( \alpha = S \) with \( \alpha \not\in \text{fv}(S) \). Still evident.
- Case Pre(\( E \)) = Pre(\( E' \)). Using the induction hypothesis, we have \( \sigma \vdash E = E' \). Hence, we have \( \sigma \vdash C \).
- Case \( \{ W^\emptyset_1 \} = \{ W^\emptyset_2 \} \). Using the same approach as before, we have the result.
- Case \( a : K_1; W^l_1 = a : K_2; W^l_2 \). Using the induction hypothesis, we have \( \sigma \vdash C'' \Leftrightarrow K_1 = K_2 \land W^l_1 = W^l_2 \). We can then easily see that \( \sigma(a : K_1; W^l_1) = \sigma(a : K_2; W^l_2) \) which gives us the result.
– Case $s(E_1, \ldots, E_n) = s(E'_1, \ldots, E'_n)$. Using the same approach as before, we have the result.
– Case $C_1 \land C_2$. Using the induction hypothesis, we have $\sigma_1 \models C_1$ and $\sigma_2 \models C_2$. Let write $\sigma$ the substitution $\sigma_2 \circ \sigma_1$. As $\sigma_1 \models C_1$, we have $\sigma \models C_1$. As $\sigma_2 \models C_1(C_2)$, we have $\sigma \models C_2$. We thus have that $\sigma \models C$.

**Theorem 14 (Constraint resolution completeness).** Let suppose given a valid constraint generation statement $F, \Gamma \vdash L : [F'] \tau \mid C$, where $\text{fs}(\Gamma) \subset F$ and $\not\vdash C$. Then there exists a substitution $\sigma$ such that $C \Rightarrow \sigma$ holds. Moreover, we have $\sigma = \text{mgu}^i(C)$.

**Proof.** Let suppose given $C$. Our induction hypothesis is that for all satisfiable constraint $C'$ with $C' < C$, we have the result. Let now prove our result by case on $C$:

– Cases **true** and $\alpha = \alpha$. This cases are evident, as any substitutions valid such constraints.
– Case $\alpha = Y$, with $Y \neq \alpha$. As this constraint is satisfiable, we have $\alpha \not\in \text{fv}(Y)$. We can then apply the proper substitution computation rule, which will gives us the result. It indeed clearly computes an mgu$^i$ for such constraint.
– Case $\text{Abs} = \text{Abs}$. Any substitution validates such constraint, and id is trivially an mgu$^i$ for this constraint.
– Case $\text{Pre}(E) = \text{Pre}(E')$. Let consider the constraint $C' \hat{=} E = E'$: we clearly have that $C'$ is satisfiable and $C' < C$. Thus, there exists a derivation $C \Rightarrow' \sigma$ with $\sigma = \text{mgu}^i(C')$. We can then see that $\sigma$ gives us the result.
– Case $a : K_1; W_1^i = a : K_2; W_2^i$. Let consider the constraint $C' \hat{=} K_1 \land W_1^i = W_2^i$: we clearly have that $C'$ is satisfiable and $C' < C$. Thus, there exists a derivation $C \Rightarrow' \sigma$ with $\sigma = \text{mgu}^i(C')$. We can then see that $\sigma$ gives us the result.
– Case $\text{Pre}(E) = \text{Pre}(E')$ and $s(E_1, \ldots, E_n) = s(E'_1, \ldots, E'_n)$. Using the same approach as in the previous case, we have the result.
– Case $C_1 \land C_2$. Let first suppose that $C$ is satisfiable: by definition, so is $C_1$. As $\text{fs}(C_1) \subset \text{fs}(C)$, we have $C_1 < C$: using the induction hypothesis, there exists a derivation of $C_1 \Rightarrow \sigma_1$ with $\sigma_1 = \text{mgu}^i(C_1)$. We then can see that $\sigma_1(C_2)$ is satisfiable: if $\sigma' \models C$, we have per definition $\sigma' \models C_1$. This implies, per definition of an mgu$^i$, that there exists $\sigma''$ such that $\sigma'' = \sigma'' \circ \sigma_1$. Thus, as we have $\sigma' \models C_2$, we have $\sigma'' \circ \sigma_1 \models C_2$, which implies that $\sigma'' \models \sigma_1(C_2)$.

Finally, we have that $\sigma(C_2) < C$: we have two cases:
1. Case $\sigma_1(C_2) = C_2$. The inequality statement is trivial from the definition of $<.
2. Case $\sigma_1(C_2) \neq C_2$. First, let remark that $\sigma_1 = \text{mgu}^i(C_1)$, and thus, $\text{dom}(\sigma_1) \cap \text{fv}(C_2)$: we then have $\alpha \not\in \text{fv}(C_2)$. We then have $\text{fs}(\sigma_1(C_2)) < \text{fs}(C)$.

We can then deduce that there is a valid derivation of $\sigma_1(C_2) \Rightarrow \sigma_2$, with $\sigma_2 = \text{mgu}^i(\sigma_1(C_2))$. Let write $\sigma$ the substitution $\sigma_2 \circ \sigma_1$. We easily have
that \( \text{dom}(\sigma) \cup \exists(\sigma) \subset \text{fv}(C) \): we then have that \( \sigma \circ \sigma = \sigma \). Now let take \( \sigma' \) such that \( \sigma' \models C \). As previously, there exists \( \sigma'' \) such that \( \sigma' = \sigma'' \circ \sigma_1 \), with \( \sigma'' \models \sigma_1(C_2) \). Thus, per construction of \( \sigma_2 \), there exists \( \sigma''' \) with \( \sigma'' = \sigma''' \circ \sigma_2 \): we then have \( \sigma' = \sigma''' \circ \sigma \). We can finally conclude that \( \sigma = \text{mgu}(C) \).

- Case \( C_1 \land E = \{ a : \alpha_1; \alpha_2 \} \land \alpha_1 = \text{Pre}_? \alpha_3 = S_1 \land \alpha_1 = \text{Abs}_? \alpha_3 = S_2 \), where \( \alpha_1, \alpha_2 \) and \( \alpha_3 \) are fresh. As \( C_1 < C \), there exists a valid substitution computation \( C_1 \land E = \{ a : \alpha_1; \alpha_2 \} \Rightarrow \sigma_1 \) with \( \sigma_1 = \text{mgu}(C_1 \land E = \{ a : \alpha_1; \alpha_2 \}) \). Thus, using the theorem 15 and the lemma 5, \( \sigma_1(E) \) is meaningful. Hence, \( \sigma_1(\alpha_1) \) is either \( \text{Abs} \) or \( \text{Pre}(E') \) for some type \( E' \). We then have two cases:

1. Case \( \sigma_1(\alpha_1) = \text{Abs} \). Let define \( \sigma_2 \) such that \( \text{dom}(\sigma_2) = \{ \alpha_3 \} \) and \( \sigma_2(\alpha_3) = \sigma_1(S_1) \). The computed substitution for \( C \) will then be \( \sigma = \sigma_2 \circ \sigma_1 \). We trivially have that \( \sigma \models C \) and \( \text{dom}(\sigma) \cup \exists(\sigma) \subset \text{fv}(C) \): we have \( \sigma \circ \sigma = \sigma \). Now, let take \( \sigma' \) such that \( \sigma' \models C \); we have \( \sigma' \models C_1 \land E = \{ a : \alpha_1; \alpha_2 \} \). There exists \( \sigma'' \) such that \( \sigma' = \sigma'' \circ \sigma_1 \). As previously, we have \( \sigma'' \models \sigma_1(\alpha_1 = \text{Pre} ? \alpha_3 = S_1 \land \alpha_1 = \text{Abs} ? \alpha_3 = S_2) \), i.e.

\[
\sigma'' \models \text{Abs} \text{Pre} ? \alpha_3 = S_1 \land \text{Abs} ? \alpha_3 = S_2.
\]

We then have \( \sigma''(\alpha_3) = \sigma''(S_1) \). Thus, we have \( \sigma'' = \sigma'' \circ \sigma_2 \): we can then conclude that \( \sigma = \text{mgu}(C) \).

2. Case \( \sigma_1(\alpha_1) = \text{Pre}(E') \). This case is similar to the previous one, with \( \sigma_2(\alpha_3) = \sigma_1(S_2) \).

### B.2 Correction

**Theorem 15.** Let suppose given a constraint generation \( F, \Gamma \vdash L : \{ F' \} \tau | C \), with \( \text{fv}(\Gamma) \subset F \), and \( \models C \). Let also suppose given a substitution \( \sigma \) such that \( \sigma \models C \). Then there exists a typing derivation of \( \sigma(\Gamma) \vdash L : \text{Gen}(\sigma(\Gamma), \sigma(\tau)) \).

**Proof.** By induction on \( L \). In each case, we will assume the same notation as in the inference rules.

- Case \( x \), with \( \Gamma(x) = \forall (\alpha_i)_{1 \leq i \leq n}.E \). First, one can remark that using \( \alpha \)-conversion, we have \( \forall (\gamma_i)_{1 \leq i \leq n}.E(\gamma_i/\alpha_i) = \Gamma(x) \). Thus, using the typing rule \( \text{T:VAR} \), we have a type derivation of \( \sigma(\Gamma) \vdash x : \sigma(\forall (\gamma_i)_{1 \leq i \leq n}.E(\gamma_i/\alpha_i)) \).

Then, let note for all \( 1 \leq i \leq n \), \( \tau_i \vdash \sigma(\gamma_i) \). Using \( n \) times the typing rule \( \text{T:INST} \), we get a type derivation of \( \sigma(\Gamma) \vdash x : \sigma(E(\gamma_i/\alpha_i)) \). Finally, using the typing rule \( \text{T:GEN} \), \( \sigma(\Gamma) \vdash x : \text{Gen}(\sigma(\Gamma), \sigma(E(\gamma_i/\alpha_i))) \) holds: we have the result in this case.

- Case \( \mathcal{C} \), with \( \exists(c) = \forall (\alpha_i)_{1 \leq i \leq n}.E \). Using the same approach as before, we have the result.

- Case \( \{ a_1 = M_1; \ldots ; a_n = M_n \} \). Let take \( \sigma \models C \); for all \( 1 \leq i \leq n \), we have \( \sigma \models C_i \). For all \( 1 \leq i \leq n \), the typing statement \( \sigma(\Gamma) \vdash M_i : \text{Gen}(\sigma(\Gamma), \sigma(E_i)) \) holds. Let then write \( \varpi_i, E'_i \) the type \( \text{Gen}(\sigma(\Gamma), \sigma(E_i)) \); per construction, we have \( \varpi_i \cap \varpi_j = \emptyset \) for all \( 1 \leq i \neq j \leq n \). Using the typing rule \( \text{T:MESSAGE} \), there exists a typing derivation of

\[
\sigma(\Gamma) \vdash \{ a_1 = M_1; \ldots ; a_n = M_n \} : \forall \bigcup_{1 \leq i \leq n} \varpi_i, \{ a_1 : \text{Pre}(E'_1); \ldots ; a_n : \text{Pre}(E'_n); \text{Abs} \}
\]
We can then remark that this typing derivation is our result.

- Case \((M M')\). Let take \(\sigma \vdash C\); we have \(\sigma \vdash C_1\) and \(\sigma \vdash C_2\). Thus, using the induction hypothesis, the typing statements \(\sigma(\Gamma) \vdash M : \text{Gen}(\sigma(\Gamma), \sigma(E_1))\) and \(\sigma(\Gamma) \vdash M : \text{Gen}(\sigma(\Gamma), \sigma(E_1))\) hold. Then, using the instantiation typing rules, we have a typing derivation of \(\sigma(\Gamma) \vdash M : \sigma(E_1)\) and \(\sigma(\Gamma) \vdash M : \sigma(E_1)\). Moreover, as \(\sigma \vdash E_1 \to E_2 \to \alpha\), we have \(\sigma(E_1) = \sigma(E_2) \to \sigma(\alpha)\).

We can then apply the typing rule \(\text{T:App}\) to get a typing derivation of \(\sigma(\Gamma) \vdash (M M') : \sigma(\alpha)\). We can finally use the generalisation rule to get a typing derivation of \(\sigma(\Gamma) \vdash (M M') : \text{Gen}(\sigma(\Gamma), \sigma(\alpha))\).

- Case \(\pi(M)\). Let take \(\sigma \vdash C\); using the induction hypothesis, there exists a type derivation of \(\sigma(\Gamma) \vdash M : \text{Gen}(\sigma(\Gamma), \sigma(E))\). We can then apply the typing rule \(\text{T:Channel}\) to get a typing derivation of \(\sigma(\Gamma) \vdash \pi(M) : s : (\text{Gen}(\sigma(\Gamma), \sigma(E)))\).

- Case \(\text{IfPre}(a, M, s_1, s_2)\). Let take \(\sigma \vdash C\); using the induction hypothesis, there exists a typing derivation of \(\sigma(\Gamma) \vdash M : \text{Gen}(\sigma(\Gamma), \sigma(E))\). As \(\sigma \vdash E = \{a : \alpha_1; \alpha_2\}\) and using the lemma 5, \(\sigma(E)\) is a record with its field \(a\) either present or absent. We thus have two cases:
  1. Case \(\sigma(\alpha_1) = \text{Pre}(E')\) for some \(E'\). As \(\sigma \vdash C\), we have \(\sigma(\alpha_3) = s_1 : (\sigma(E)) \cup s_2\). Finally, using the typing rule \(\text{T:IfPre1}\), we have a typing derivation of \(\sigma(\Gamma) \vdash \text{IfPre}(a, M, s_1, s_2) : \text{Gen}(\sigma(\Gamma), \sigma(\alpha_3))\).
  2. Case \(\sigma(\alpha_1) = \text{Abs}As\). As \(\sigma \vdash C\), we have \(\sigma(\alpha_3) = s_1 \cup s_2 : (\sigma(E))\). Finally, using the typing rule \(\text{T:IfPre2}\), we have a typing derivation of \(\sigma(\Gamma) \vdash \text{IfPre}(a, M, s_1, s_2) : \text{Gen}(\sigma(\Gamma), \sigma(\alpha_3))\).

**Corollary 3 (Correction).** Let suppose given:

- A valid constraint generation \(F, \Gamma \vdash L : [F'] | \tau | C\), with \(\text{fs}(\Gamma) \subset F\), and \(\vdash C\).
- A valid constraint resolution \(C \Rightarrow \sigma\).

Then, there exists a type derivation of \(\sigma(\Gamma) \vdash L : \text{Gen}(\sigma(\Gamma), \sigma(\tau))\).

**Proof.** Using the theorem 14, we have \(\sigma \vdash C\). Thus, with the theorem 15, we have the result.

**B.3 Completeness**

**Lemma 12.** Let suppose given a valid constraint generation statement \(F, \Gamma \vdash L : [F'] | \tau | C\). Then we have \(F \subset F'\) and \(\text{fs}(\tau) \cup \text{fs}(C) \subset (F' \setminus F) \cup \text{fs}(\Gamma)\).

**Proof.** By induction on \(L\):

- Case \(x\) with \(\Gamma(x) = \forall(a_i)_{1 \leq i \leq n}.E\): we have \(\text{fs}(E) \subset \text{fs}(\Gamma) \cup \text{fs}(a_i)_{1 \leq i \leq n}\). Hence, we have \(\text{fs}(E) \setminus \text{fv}(\Gamma) \subset \text{fs}(\Gamma) \cup (F' \setminus F)\). Moreover, as \(C = \text{true}\), the second inclusion holds. Finally, \(F \subset F'\) is evident by construction.
- Case \(c\) with \(\delta(c) = \forall(a_i)_{1 \leq i \leq n}.E\). Using the same approach as in the previous case, we have the result.
Theorem 16. Let suppose given a valid constraint generation statement \( F, \Gamma \vdash L : [F'] \tau' | C \). Let also suppose given a substitution \( \sigma \) with \( \mathcal{S}(\sigma) \cap (F' \setminus F) = \emptyset \) and a type \( \tau \) such that \( \sigma(\Gamma) \vdash L : \tau \) holds. Then, there exist a substitution \( \sigma' \) and a set of type variables \( \Pi \) such that:

- \( \Pi \cap \text{fs}(\sigma(\Gamma)) = \emptyset \).
- \( \sigma' \vdash C \).
- \( \text{dom}(\sigma') \subset (F' \setminus F) \cup \text{fs}(\Gamma) \).
- \( \sigma'(\Gamma) = \sigma(\Gamma) \).
- \( \tau = \forall \Pi. \sigma'(\tau') \) in the case of \( L \) is a \( M \) construct.
- \( \tau \supset \forall \Pi. \sigma'(\tau') \) in the case of \( L \) is a \( R \) construct.

Proof. By induction on type derivation of \( \Gamma \vdash L : \tau \). To simplify the proof, and improve its readability, we will use the notation of the typing rules, along with the ones used in the inference rules.

- Case T:VAR with \( \Gamma(x) = \forall (\alpha_i)_{1 \leq i \leq n}.E \): we have \( \tau = \forall (\alpha_i)_{1 \leq i \leq n}.E \). Let define \( \sigma' \triangleq \sigma_{|\text{fs}(\Gamma)} \) and \( \Pi \triangleq (\gamma_i)_{1 \leq i \leq n} \). Per hypothesis, we have \( \gamma_i \notin \text{fs}(\Gamma) \), and \( \gamma_i \notin \mathcal{S}(\sigma) \). This implies that \( \Pi \cap \text{fs}(\sigma(\Gamma)) = \emptyset \).

As any substitution validate the true constraint, we have \( \sigma' \vdash C \). We also clearly have that \( \text{dom}(\sigma') = \emptyset \subset (F' \setminus F) \cup \text{fs}(\Gamma) \). Finally, using the \( \alpha \)-conversion, we have \( \tau = \sigma(\forall \Pi. \sigma'(\tau')) \).

- Case T:PRIMITIVE, with \( \exists(c) = \forall (\alpha_i)_{1 \leq i \leq n}.E \). Using the same approach as before, we have the result.

- Case T:MESSAGE. Using the induction hypothesis, for all \( 1 \leq i \leq n \), there exists a substitution \( \sigma_i \) and a set \( \Pi_i \) such that:
\[ \gamma_i \cap \text{fs}(\sigma(\Gamma)) = \emptyset. \]
\[ \sigma_i \vdash C_i \]
\[ \text{dom}(\sigma_i) \subset F_i \setminus F_{i-1}. \]
\[ \forall \gamma_i, E_i = \forall \gamma_i, \sigma_i(E_i'). \]

Let note for all \( 1 \leq i \leq n \) \( \{ \gamma_i^j \mid 1 \leq j \leq n_i \} \) the set \( \gamma_i \). Let take for all \( 1 \leq i \leq n \) and all \( 1 \leq j \leq n_i \) a fresh variable \( \beta_i^j \), and note \( \beta_i \) the set \( \{ \beta_i^j \mid 1 \leq j \leq n_i \} \). We can now define for all \( 1 \leq i \leq n \) the permutation \( \sigma'_i \) replacing the \( \gamma_i^j \) with the \( \beta_i^j \). Per construction, we have:

- For all \( 1 \leq i \leq n \), \( \forall \gamma_i, \sigma'_i \circ \sigma_i(E_i') = \forall \gamma_i, \sigma_i(E_i') \)
- For all \( 1 \leq i \neq j \leq n \), \( (\beta_i \cup \text{fs}(\sigma'_i \circ \sigma_i(E_i'))) \cap \beta_j = \emptyset \).

Let now define \( \sigma' \) as:

- \( \text{dom}(\sigma') \subset (F_n \setminus F_0) \cup \text{fs}(\Gamma) \).
- For all \( \alpha \in \text{fs}(\Gamma) \), \( \sigma'(\alpha) = \sigma(\alpha) \).
- For all \( 1 \leq i \leq n \) and all \( \alpha \in F_i \setminus F_{i-1} \), \( \sigma'(\alpha) = \sigma_i^j \circ \sigma_i(\alpha) \).

Then, for all \( 1 \leq i \leq n \), we have \( \sigma'(E_i') = \sigma'_i \circ \sigma_i(E_i') \). Moreover, as \( \sigma_i \vdash C_i \), we have \( \sigma'_i \circ \sigma_i \vdash C_i \). Let remark that \( \text{fs}(C_i) \subset (F_i \setminus F_{i-1}) \cup \text{fs}(\Gamma) \). As \( \sigma'(\alpha) = \sigma_i^j \circ \sigma_i(\alpha) \) for all \( \alpha \in \text{fs}(C_i) \), we have \( \sigma' \vdash C \), which implies that \( \sigma' \vdash C \).

Per construction, we have \( (\text{dom}(\sigma) \cup \exists(\sigma)) \cap \bigcup_{1 \leq i \leq n} \beta_i = \emptyset \). We then have
\[ \sigma(\forall \beta_i, \sigma_i(E_i')) = \forall \beta_i, \sigma \circ \sigma'(E_i') \]. This implies the equality \( \forall \beta_i, E_i = \forall \beta_i, \sigma \circ \sigma'(E_i') \).

Finally, we have:
\[ \tau = \forall \bigcup_{1 \leq i \leq n} \beta_i \{ a_1 : \text{Pre}(\sigma'(E'_1)); \ldots ; a_n : \text{Pre}(\sigma'(E'_{n_i})); \text{Abs}^{a_1, \ldots , a_n} \} \]
\[ = \forall \bigcup_{1 \leq i \leq n} \beta_i, \sigma'(\{ a_1 : \text{Pre}(E'_1); \ldots ; a_n : \text{Pre}(E'_{n_i}); \text{Abs}^{a_1, \ldots , a_n} \}) \]
\[ = \sigma(\forall \bigcup_{1 \leq i \leq n} \beta_i, \sigma'(\tau')) \]

- Case T:App. Using the induction hypothesis, there exist two substitution \( \sigma_1 \) and \( \sigma_2 \) such that:
  - \( \sigma_1 \vdash C_1 \) and \( \sigma_2 \vdash C_2 \).
  - \( \text{dom}(\sigma_1) \subset (F_1 \setminus F) \cup \text{fs}(\Gamma) \) and \( \text{dom}(\sigma_2) \subset (F_2 \setminus F_1) \cup \text{fs}(\Gamma) \).
  - \( \sigma_1(\Gamma) = \sigma_2(\Gamma) = \sigma(\Gamma) \)
  - \( E \rightarrow E' \rightarrow \sigma_1(E_1) \) and \( E = \sigma_2(E_2) \).
Let define \( \sigma' \) as:
  - \( \text{dom}(\sigma') \subset (F_2 \setminus F) \cup \text{fs}(\Gamma) \cup \{ \alpha \} \).
  - For all \( \alpha \in \text{fs}(\Gamma) \), \( \sigma'(\alpha) = \sigma(\alpha) \).
  - For all \( \alpha \in F_1 \setminus F \), \( \sigma'(\alpha) = \sigma_1(\alpha) \).
  - For all \( \alpha \in F_2 \setminus F_1 \), \( \sigma'(\alpha) = \sigma_2(\alpha) \).
  - \( \sigma'(\alpha) = E' \).

Using the same approach as before, we have \( \sigma' \vdash C_1 \wedge C_2 \). Moreover, we have \( \sigma'(E_2 \rightarrow \alpha) = E \rightarrow E' \); we have \( \sigma' \vdash C \).

Hence, we have the result in this case.

- Case T:Inst—Gen. Using the induction hypothesis, there exists a substitution \( \sigma_1 \) and a set of type variables \( \beta_1 \) such that:
  - \( \beta_1 \cap \text{fs}(\sigma(\Gamma)) = \emptyset \).
  - \( \sigma_1 \vdash C \).
As \( \sigma(\Gamma) \vdash T' \iff T \), there exists \( \sigma_2, T'', \overline{\pi}_2 \) and \( \overline{\pi}_3 \) such that:

- \( \text{dom}(\sigma_2) \subseteq \overline{\pi}_2 \).
- \( (\overline{\pi}_3 \cup \overline{\pi}_2) \cap \text{fv}(\sigma(\Gamma)) = \emptyset. \)
- \( T = \forall \overline{\pi}_2. T'' \).
- \( T' = \forall \overline{\pi}_3. \sigma_2(T''). \)

Using \( \alpha \)-conversion, we can suppose that \( (\text{dom}(\sigma_2) \cup \exists(\sigma_2)) \cap \overline{\pi}_3 = \emptyset. \)

Let define \( \sigma' \triangleq (\sigma_2 \circ \sigma_1)_{|\text{fv}(C)\cap\text{dom}(\sigma_1)} \) and \( \overline{\pi} = (\overline{\pi}_1 \setminus \overline{\pi}_2) \cup \overline{\pi}_3. \) We easily have that \( T' = \forall \overline{\pi}. \sigma_2 \circ \sigma_1(\tau') = \forall \overline{\pi}. \sigma'(\tau'). \)

As for all \( \alpha \in \text{fv}(C) \), we have \( \sigma'(\alpha) = \sigma_2 \circ \sigma_1(\alpha) \), we have \( \sigma' \models C. \) Moreover, as \( \text{fv}(C) \subseteq (F' \setminus F) \cup \text{fv}(\Gamma) \), we have the result. Finally, as \( \overline{\pi}_2 \cap \text{fv}(\sigma(\Gamma)) = \emptyset, \) we have \( \sigma'(\Gamma) = \sigma_1(\Gamma) = \sigma(\Gamma). \)

- Case T:CHANNEL. Using the induction hypothesis, there exist a substitution \( \sigma' \) and a set of type variables \( \overline{\pi} \) such that:
  - \( \overline{\pi} \cap \text{fv}(\sigma(\Gamma)) = \emptyset. \)
  - \( \sigma' \models C. \)
  - \( \text{dom}(\sigma') \subseteq (F' \setminus F) \cup \text{fv}(\Gamma). \)
  - \( \sigma'(\Gamma) = \sigma(\Gamma). \)
  - \( T = \forall \overline{\pi}. \sigma'(\tau'). \)

As \( T' \in \tau(s) \), we clearly have that \( s : (T) \subseteq \tau \), which gives us the result.

- Case T:IFPRE1. Using the induction hypothesis, there exist a substitution \( \sigma_1 \) and a set of type variables \( \overline{\pi} \) such that:
  - \( \overline{\pi} \cap \text{fv}(\sigma(\Gamma)) = \emptyset. \)
  - \( \sigma_1 \models C. \)
  - \( \text{dom}(\sigma_1) \subseteq (F_1 \setminus F). \)
  - \( T = \forall \overline{\pi}. \sigma_1(\tau'). \)

By definition of the typing rule, there exist \( E \) and \( W^{(a)} \) such that \( T = \forall \overline{\pi}. \{ a : \text{Pre}(E); W^{(a)} \} \). Let define \( \sigma' \) as

- \( \text{dom}(\sigma') \subseteq (F_1 \setminus F_2) \cup \text{fv}(\Gamma) \cup \{ \alpha_1, \alpha_2, \alpha_3 \}. \)
- For all \( \alpha \in \text{dom}(\sigma_1) \), \( \sigma'(\alpha) = \sigma_1(\alpha) \).
- \( \sigma'(\alpha_1) = \text{Pre}(E). \)
- \( \sigma'(\alpha_2) = W^{(a)}. \)
- \( \sigma'(\alpha_3) = s_1 : (\sigma_1(\tau')). \)

With this substitution and \( \overline{\pi} \), the result is evident.

- Case T:IFPRE2. Using the induction hypothesis, there exist a substitution \( \sigma_1 \) and a set of type variables \( \overline{\pi} \) such that:
  - \( \overline{\pi} \cap \text{fv}(\sigma(\Gamma)) = \emptyset. \)
  - \( \sigma_1 \models C. \)
  - \( \text{dom}(\sigma_1) \subseteq (F_1 \setminus F). \)
  - \( T = \forall \overline{\pi}. \sigma_1(\tau'). \)

By definition of the typing rule, there exist \( W^{(a)} \) such that \( T = \forall \overline{\pi}. \{ a : \text{Abs}; W^{(a)} \} \).

Let define \( \sigma' \) as

- \( \text{dom}(\sigma') \subseteq (F_1 \setminus F_2) \cup \text{fv}(\Gamma) \cup \{ \alpha_1, \alpha_2, \alpha_3 \}. \)
For all $\alpha \in \text{dom}(\sigma_1)$, $\sigma'(\alpha) = \sigma_1(\alpha)$.

$\bullet$ $\sigma'(\alpha_1) = \text{Abs}$.

$\bullet$ $\sigma'(\alpha_2) = W'(\alpha)$.

$\bullet$ $\sigma'(\sigma_3) = s_1 : (\sigma_3(\tau'))$.

With this substitution and $\overline{\sigma}$, the result is evident.

**Corollary 4 (Completeness).** Let suppose given a valid typing statement $\Gamma \vdash L : \tau$, and a set of type variable $F$ such that $\text{fs}(\Gamma) \subseteq F$. Then, there exists:

- A valid constraint generation statement $F, \Gamma \vdash L : [F'] \tau' | C$, with $\vdash C$.

- A valid constraint resolution $C \Rightarrow \sigma$.

Moreover, there exist a permutation $\sigma'$ such that:

- $\sigma' \circ \sigma(\Gamma) = \Gamma$.
- $\Gamma \vdash \tau \Leftarrow \text{Gen}(\Gamma, \sigma' \circ \sigma(\tau'))$.

**Proof.** Using the theorem 16, there exist $\sigma_1$ and a set of type variables $\overline{\sigma}$ such that:

- $\overline{\sigma} \cap \text{fs}(\Gamma) = \emptyset$.
- $\sigma_1 \vdash C$.
- $\text{dom}(\sigma_1) \subset (F' \setminus F) \cup \text{fs}(\Gamma)$.
- $\sigma_1(\Gamma) = \Gamma$.
- $\tau = \forall \overline{\alpha}. \sigma_1(\tau')$ in the case of $L$ is a $M$ construct.
- $\tau \supset \forall \overline{\alpha}. \sigma_1(\tau')$ in the case of $L$ is a $R$ construct.

As $\sigma = \text{mgu}^i(C)$, there exists $\sigma_1'$ such that $\sigma_1 = \sigma_1' \circ \sigma$. As $\sigma_1(\Gamma) = \Gamma$, if we define $\{\alpha_i \mid 1 \leq i \leq n\} \triangleq \text{fs}(\Gamma)$, we have $\sigma(\alpha_i) = \beta_i$ for some $\beta_i$, and $\sigma_1'(\beta_i) = \alpha_i$. We thus define $\sigma'$ as being the permutation switching the $\alpha_i$'s into the $\beta_i$'s: we have $\sigma' \circ \sigma(\Gamma) = \Gamma$. As $\sigma'$ is a permutation, it is invertible, and is its own opposite: we thus have $\sigma_1 = \sigma_1' \circ \sigma' \circ \sigma'. \sigma$. We thus can conclude: $\Gamma \vdash \tau \Leftarrow \text{Gen}(\Gamma, \sigma' \circ \sigma(\tau'))$ (the involved substitution being $\sigma_1' \circ \sigma'$).

**Corollary 5 (Principal type).** Let suppose given a valid constraint generation statement $F, \Gamma \vdash L : [F'] \tau' | C$, with $\vdash C$. Let note $\sigma$ the substitution such that $C \Rightarrow \sigma$. Then, for all substitution $\sigma'$, we have $\sigma' \circ \sigma(\Gamma) \vdash_m L : \sigma'(\text{Gen}(\sigma(\Gamma), \sigma(\tau)))$.

**Proof.** Using the lemma 14, we have $\sigma = \text{mgu}^i(C)$, which implies that $\sigma \vdash C$. We can then apply the theorem 15 to get a type derivation of $\sigma(\Gamma) \vdash L : \text{Gen}(\sigma(\Gamma), \sigma(\tau))$.

Let then take a type $\tau'$ such that $\sigma(\Gamma) \vdash L : \tau'$ holds. Using the lemma 16, there exists $\sigma_k$ and $\overline{\sigma}$ such that

- $\overline{\sigma} \cap \text{fs}(\sigma(\Gamma)) = \emptyset$.
- $\sigma_k \vdash C$.
- $\text{dom}(\sigma_k) \subset (F' \setminus F) \cup \text{fs}(\Gamma)$.
- $\sigma_k(\Gamma) = \sigma(\Gamma)$.
- $\tau' = \forall \overline{\alpha}. \sigma_k(\tau)$.
As $\sigma = \text{mgu}^i(C)$, there exist $\sigma'_k$ with $\text{dom}(\sigma'_k) \subset \text{dom}(\sigma_k)$ such that $\sigma_k = \sigma'_k \circ \sigma$. Moreover, as $\sigma_k(\Gamma) = \sigma(\Gamma)$, we have $\text{dom}(\sigma'_k) \cap \text{fv}(\sigma(\Gamma)) = \emptyset$. We thus have $\sigma(\Gamma) \vdash \tau' \leftarrow \text{Gen}(\sigma(\Gamma), \sigma(\tau))$.

Finally, as $\text{fv}(\text{Gen}(\sigma(\Gamma), \sigma(\tau))) \subset \text{fv}(\sigma(\Gamma))$, the typing statement $\sigma(\Gamma) \vdash_m L : \text{Gen}(\sigma(\Gamma), \sigma(\tau))$ holds. Hence, we can apply the lemma 8 to get the result.

**Corollary 6.** Let suppose given $R$, $S$ and a typing environment $\Gamma$ such that the typing statement $\Gamma \vdash R : S$ is valid. Then, there exists a set type $S'$ such that $\Gamma \vdash_m R : S'$.

**Proof.** Using the theorem 16, there exists $F_1$, a set type $S_1$, a constraint $C$, a set of variables $\pi$ and a substitution $\sigma_1$ such that:

- $\text{fv}(\Gamma), \Gamma \vdash R : [F_1] S_1 | C$ holds.
- $\sigma_1 \models C$.
- $\pi \cap \text{fv}(\Gamma) = \emptyset$.
- $\text{dom}(\sigma_1) \subset F_1 \setminus \text{fv}(\Gamma)$.
- $S \supset \forall x. \sigma_1(S_1)$.

Let then take the substitution $\sigma_2$ such that $C \Rightarrow \sigma_2$: there exists $\sigma_3$ with $\sigma_1 = \sigma_3 \circ \sigma_2$. Let then define $\text{fv}(\Gamma) = \{ \beta_i \mid i \in I \}$. As we have $\text{dom}(\sigma_1) \cap \text{fv}(\Gamma) = \emptyset$, for all $i \in I$, there exists $\beta'_i$ such that $\sigma_2(\beta_i) = \beta'_i$. Moreover, we have $i \neq j \in I \Rightarrow \beta'_i \neq \beta'_j$. Let consider the permutation $\sigma_g$ such that:

- $\text{dom}(\sigma_g) = \text{fv}(\Gamma) \uplus \{ \beta'_i \mid i \in I \}$ (we have $\text{dom}(\sigma_2) \cap \exists(\sigma_2) = \emptyset$).
- $\sigma_g(\beta_i) = \beta'_i$ for all $i \in I$.
- $\sigma_g(\beta'_i) = \beta_i$ for all $i \in I$.

Let define the substitution $\sigma \triangleq \sigma_g \circ \sigma_2$. We can then easily see that $\sigma = \text{mgu}(C)$ and $\text{dom}(\sigma) \cap \text{fv}(\Gamma) = \emptyset$. We can then consider the type $S' \triangleq \text{Gen}(\Gamma, \sigma(S_1))$.

Using the theorem 15, there exists a type derivation of $\Gamma \vdash R : S'$. Let define $\sigma' \triangleq \sigma_3 \circ \sigma_g$: we have $\sigma_1 = \sigma' \circ \sigma$ per construction. As $\text{dom}(\sigma_1) \cap \text{fv}(\Gamma) = \emptyset$ and $\text{dom}(\sigma') = \text{fv}(\Gamma) = \emptyset$, we have $\text{dom}(\sigma') \cap \text{fv}(\Gamma) = \emptyset$. We can then conclude that $\sigma_1(S_1) \leftarrow S'$, which implies that $\Gamma \vdash S \leftarrow S'$. 


We prove in this section that our inference semi-algorithm always succeed when given in input a typable program. Moreover, it computes its minimal type, modulo some permutation.

**Definition 32.** Let suppose given a typing environment $\Gamma$, a program $D$ and a set type $S$. We write $\text{next}(\Gamma \vdash D : S)$ the set

$$\{(\Gamma', S') \mid \Gamma \vdash D : S \rightarrow \Gamma' \vdash D : S'\}$$

**C.1 Preliminary Results**

For the next lemma only, we will allow the construction of type schemes of the form $\forall \alpha.K$ or $\forall \alpha.W^l$. This is for the sake of a simple presentation of the proof, which is done by induction on the structure of the type.

**Lemma 13.** Let suppose given a typing environment $\Gamma$ and a type $\tau$. The set $\{\tau' \mid \Gamma : \tau \Rightarrow \tau'\}$ is finite.

**Proof.** Let note $U(\tau)$ the set $\{\tau' \mid \Gamma : \tau \Leftarrow \tau'\}$. We prove the result by induction on $\tau$:

- Case $\alpha$. We trivially have that $U(\tau) \subset \{\alpha, \forall \beta.\beta\}$, which gives us the result (we don’t have the equality of the sets in the case of $\alpha$ is a set type variable, or $\alpha \notin \text{fv}(\Gamma)$).
- Case Abs. We trivially have that $U(\tau) = \{\text{Abs}, \alpha\}$, which gives us the result.
- Case $\text{Pre}(E)$. Let write $\{\forall \pi_i, E_i \mid 1 \leq i \leq n\}$ the finite (per the induction hypothesis) set $U(E)$. We can then see that $U(\text{Pre}(E))$ is included into the set $\{\forall \pi_i.\text{Pre}(E_i) \mid 1 \leq i \leq n\} \cup \{\forall \alpha.\alpha\}$. As this last set is finite, we have the result.
- Case $a : K; W^l$. Let write $U(K) \triangleq \forall \pi_i, K_i \mid i \in I$ and $U(W^l) \triangleq \forall \pi_j, W^l_j \mid j \in J$. Using the induction hypothesis, we have that $I$ and $J$ are finite. We suppose, with the $\alpha$-conversion, that for all $i \in I$ and all $j \in J$, we have that $\pi_i \cap \pi_j = \emptyset$. We can then easily see that $U(a : K; W^l)$ is included into

$$\forall \pi_i \cup \pi_j, a : K_i; W^l_j \mid i \in I, j \in J \cup \{\forall \alpha.\alpha\}$$

As this set is finite, we have the result.
- Case $s(E_1, \ldots, E_n)$. Let write for all $1 \leq i \leq n \{\forall \pi_j, E_j \mid j \in J_i\}$ the set $U(E_i)$. Using the induction hypothesis, we have that $J_i$ is finite for all $1 \leq i \leq n$. We can suppose, with the $\alpha$-conversion, that for all $1 \leq i \neq i' \leq n$ and all $j \in J_i$, $j' \in J_i'$, we have $\pi_j \cap \pi_{j'} = \emptyset$. We can then easily see that $U(s(E_1, \ldots, E_n))$ is included into

$$\forall \bigcup_{1 \leq i \leq n} \pi_j, s(E_j_1, \ldots, E_{j_n}) \mid j_1 \in J_1, \ldots, j_n \in J_n \cup \{\forall \alpha.\alpha\}$$

As this set is finite, we have the result.
By induction on the type derivation of $\Gamma : \sigma \leq \tau'$, the induction hypothesis gives us directly the result.

Case $e : (T)$. Let write $\{T_i \mid 1 \leq i \leq n\}$ the finite (per the induction hypothesis) set $U(T)$. We can then easily see that $U(e : (T_i)) \subset \{ e : (T_i) \mid 1 \leq i \leq n \} \cup \{ e, \emptyset \}$, which gives us the result.

Case $S \cup S'$. Let write $U(S) \doteq \{ S_i \mid i \in I \}$ and $U(S_j) \doteq \{ S_j \mid j \in J \}$. Using the induction hypothesis, we have that $I$ and $J$ are finite. We can then easily see that $U(S \cup S')$ is included into $\{ S_i \cup S_j \mid i \in I, j \in J \}$. As this set is finite, we have the result.

Lemma 14. Let suppose given a typing environment $\Gamma$, a program $L$ and a type $\tau$ such that $\Gamma \vdash L : \tau$ holds. Let write $\Gamma \doteq \emptyset ; x_1 : T_1; \ldots ; x_n : T_n$, and define $\Gamma' \doteq \emptyset ; x_1 : T'_1; \ldots ; x_n : T'_n$ such that $\Gamma : T_i \leq T'_i$. Then, the typing statement $\Gamma' \vdash L : \tau$ holds.

Proof. By induction on the type derivation of $\Gamma \vdash L : \tau$:

- Case $T : \text{VAR}$, with $1 \leq i \leq n$ such that $x = x_i$. As we have $\Gamma : T_i \leq T'_i$, there exists two sets of type variables $\alpha, \beta$ a type $T$ and a substitution $\sigma$ such that
  \begin{itemize}
  \item $fv(T'_i) \subset fv(\Gamma)$.
  \item $\alpha \cap fv(\Gamma) = \emptyset$.
  \item $\text{dom}(\sigma) \subset \beta$.
  \item $T'_i = \forall \beta. T$.
  \item $T_i = \forall \alpha. \sigma(T)$.
  \end{itemize}

We have a type derivation of $\Gamma' \vdash x_i : T'_i$. Using the typing rule $T:\text{INST}$, we have a derivation of the statement $\Gamma' \vdash x_i : \sigma(T)$. Then, as for all $1 \leq j \leq n$ we have $fv(T'_j) \subset fv(\Gamma)$, we have $fv(\Gamma') \subset fv(\Gamma)$. We can then apply the typing rule $T:\text{GEN}$ to get a type derivation of $\Gamma' \vdash x_i : T_i$.

- Case $T : \text{PRIMITIVE}$. As the typing environment doesn’t act in this typing rule, we have trivially the result.

- Cases $T : \text{MESSAGE}$, $T : \text{APP}$, $T : \text{INST}$. This cases result from a simple application of the induction hypothesis.

- Case $T : \text{GEN}$. As $fv(\Gamma') \subset fv(\Gamma)$, this case is evident.

- Cases $T: \text{CHANNEL}$, $T: \text{IFPRE1}$, $T: \text{IFPRE2}$, $T: \text{FUNCTION}$, $T: \text{BANG}$. Using the induction hypothesis, the result is evident in these cases.

- Case $T : \text{ZERO}$. As the typing environment doesn’t act in this typing rule, we have trivially the result.

- Cases $T : \text{BOX}$, $T : \text{PARALLEL}$. Using the induction hypothesis, the result is evident in these cases.

Lemma 15. Let suppose given a typing environment $\Gamma$ and two types $\tau$ and $\tau'$ such that:

- Neither $\tau$ nor $\tau'$ contain a set type variable : $(fv(\tau) \cup fv(\tau')) \cap \forall \alpha = \emptyset$.
- $\Gamma \vdash \tau' \leq \tau$. 

Then, for all substitution $\sigma$, we have $\sigma(\Gamma) \vdash \sigma(\tau') \Leftarrow \sigma(\tau)$.

**Proof.** Let first consider the case $\tau' = T'$ and $\tau = T$. As $\Gamma \vdash T' \Leftarrow T$, there exists $\overline{\pi}, \overline{\beta}, \overline{\gamma}, E$ and $\sigma'$ such that:

- $T = \forall \overline{\pi}.\forall \overline{\beta}.E$,
- $T' = \forall \overline{\gamma}.\sigma'(\forall \overline{\beta}.E)$,
- $\text{dom}(\sigma') \subset \overline{\pi}$,
- $(\overline{\pi} \cup \overline{\gamma}) \cap \text{fv}(\Gamma') = \emptyset$.

Using $\alpha$-conversion and a type variables permutation, we can suppose that:

- $(\text{dom}(\sigma) \cup \exists(\sigma)) \cap (\overline{\pi} \cup \overline{\gamma}) = \emptyset$,
- $\overline{\pi} \cap \overline{\beta} = \overline{\gamma} \cap \overline{\beta} = \emptyset$.

Let define $\sigma'' \triangleq (\sigma \circ \sigma')(\text{dom}(\sigma'))$. Then, the equality $\sigma'' \circ \sigma(\forall \overline{\beta}.E) = \sigma \circ \sigma'(\forall \overline{\beta}.E)$ holds:

- Let take $\alpha \in \text{dom}(\sigma')$. As $\text{dom}(\sigma') \cap \text{dom}(\sigma) \subset \overline{\pi} \cap \text{dom}(\sigma) = \emptyset$, we have:

  \[
  \sigma'' \circ \sigma(\alpha) = \sigma''(\sigma'(\alpha)) = \sigma \circ \sigma'(\alpha)
  \]

- Let now take $\alpha \in \text{dom}(\sigma)$. As $\text{dom}(\sigma') \cap (\text{dom}(\sigma) \cup \exists(\sigma)) \subset \overline{\pi} \cap (\text{dom}(\sigma) \cup \exists(\sigma)) = \emptyset$, we have

  \[
  \sigma'' \circ \sigma(\alpha) = \sigma''(\sigma'(\alpha)) = \sigma'(\alpha).\]

- We can finally see that $\text{dom}(\sigma'') \cup \text{dom}(\sigma \circ \sigma') \subset \text{dom}(\sigma) \cup \text{dom}(\sigma')$.

We can then conclude with:

- $\sigma(\Gamma) = \forall \overline{\pi}.\sigma(\forall \overline{\beta}.E)$,
- $\sigma(\Gamma') = \forall \overline{\gamma}.\sigma''(\forall \overline{\beta}.E))$.
- $\text{dom}(\sigma'') = \text{dom}(\sigma') \subset \overline{\pi}$.
- $\overline{\gamma} \cap \text{fv}(\sigma(\Gamma)) \subset (\overline{\gamma} \cap \text{fv}(\Gamma)) \cup (\overline{\gamma} \cap \exists(\sigma)) = \emptyset$.

Let now consider the case $\tau' = S'$ and $\tau = S$: we have

- $dc(S) \subset dc(S')$.
- For all $e \in dc(S)$, for all $T' \in S'(e)$ there exists $T \in S(e)$ with $\Gamma : T' \Leftarrow T$.

We can conclude:

- $dc(\sigma(S)) = dc(S) \subset dc(S') = dc(\sigma(S'))$. 
Lemma 16. Let suppose given a typing environment $\Gamma$, a program $R$ and a set type $S$ such that $\Gamma \vdash R : S$ holds. Then, for all set type $S'$ the statement $\Gamma \vdash R : S \cup S'$ holds.

Proof. By case on the last typing rule of the derivation of $\Gamma \vdash R : S$:

- Case $T:\text{Channel}$. We have $T \in S(e)$. By construction, this implies that $T \in (S \cup S')(e)$. We can then apply the typing rule $T:\text{Channel}$ to get the result.
- Case $T:\text{IfPre}1$ and $T:\text{IfPre}2$. Using the same approach as before, we have the result.

Lemma 17. Let suppose given a typing environment $\Gamma$, a program $R$ and two set types $S$, $S'$ such that $\Gamma \vdash R : S$ and $\Gamma \vdash R : S'$ hold. Then, we have $\text{dc}(S) \cap \text{dc}(S') \neq \emptyset$.

Proof. By case on the last typing rule of the derivation of $\Gamma \vdash R : S$:

- Case $T:\text{Channel}$. As we have a type derivation of $\Gamma \vdash R : S$ and $\Gamma \vdash R : S'$, we have a type derivation of:
  - $\Gamma \vdash M : T$ with $T \in S(s)$.
  - $\Gamma \vdash M : T'$ with $T' \in S'(s)$.
  
  As $T \in S(s)$ and $T' \in S'(s)$, we have $e \in \text{dc}(S) \cap \text{dc}(S')$.
- Case $T:\text{IfPre}1$: there exists $T$ with $\Gamma \vdash M : T$ and $T \in S(s_1)$. Using the lemma 6, there exists $T''$ such that $\Gamma \vdash M : T''$. Using the lemma 5, there exist $\pi$, $E$ and $W^{(a)}$ such that $T'' = \forall \pi.\{a : \text{Pre}(E); W^{(a)}\}$. Let then consider $J$ the typing derivation of $\Gamma \vdash R : S'$, $J$ without its last typing rule gives a type $T'$ such that $\Gamma \vdash M : T'$. As $\Gamma \vdash T' \Leftrightarrow T''$, there exist $\beta$, $E'$ and $W^{(a)}_1$ such that $T' = \forall \beta.\{a : \text{Pre}(E'); W^{(a)}_1\}$. Thus, the last typing rule of $J$ is $T:\text{IfPre}1$, which implies that $T' \in S'(s_1)$. We then have $s_1 \in \text{dc}(S) \cap \text{dc}(S')$.
- Case $T:\text{IfPre}2$. Using the same approach as before, we get the result.

C.2 General Properties

Lemma 18. Let suppose given a valid typing statement $\Gamma \vdash D : S_k$, a permutation $\sigma$ and a process type $S$ such that $\Gamma \vdash S_k \Leftarrow \sigma(S)$ and $\text{fv}(S) \cap V^s = \emptyset$. Then, for all $(\Gamma', S') \in \text{next}(\sigma^{-1}(\Gamma) \vdash D : S)$, there exist a permutation $\sigma'$ and a process type $S''$ such that:

- $\sigma^{-1}(\Gamma) = \sigma'(\Gamma')$.
- $S \cup S'' = \sigma'(S')$.
Moreover, we have \( \Gamma \vdash S_k \iff \sigma \circ \sigma'(S') \).

**Proof.** From the definition of \((\Gamma', S') \in \text{next}(\Gamma \vdash D : S)\), there exists:

- A sub-program \((e_1 : x_1; \ldots; e_n : x_n/R) \in D\)
- A message type \(T_i \in S(e_i)\) for all \(1 \leq i \leq n\)
- A valid constraint generation \(fv(\sigma^{-1}(\Gamma)) \cup \text{fvt}(S), \sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash R : [F']_S \mid C\)
- A substitution \(\sigma_1\) such that \(C \Rightarrow \sigma_1\)

Moreover, if we define \(S_2 \triangleq \text{Gen}(\sigma_1 \circ \sigma^{-1}(\Gamma), \sigma_1(S_1))\), we have \(S_2 \not\subseteq \sigma_1(S)\), \(\Gamma' = \sigma_1 \circ \sigma^{-1}(\Gamma)\) and \(S' = \sigma_1(S) \cup S_2\).

As we have \(\Gamma \vdash S_k \iff \sigma(S)\), for all \(1 \leq i \leq n\), there exist \(T^i_k \in S_k(e_i)\) such that \(\Gamma \vdash T^i_k \Leftarrow \sigma(T_i)\). Then, using the lemma 10, the typing statement \(\Gamma; x_1 : T^1_k; \ldots; x_n : T^n_k \vdash R : S_k\) holds. We also use the lemma 14 to get a derivation of \(\Gamma; x_1 : \sigma(T_1); \ldots; x_n : \sigma(T_n) \vdash R : S_k\). Let then use the lemma 7: the typing statement \(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash \sigma^{-1}(S_k)\) holds. We can finally use the corollary 4 to get a permutation \(\sigma'\) such that:

\[
- \sigma'((\Gamma')) = \sigma^{-1}(\Gamma).
- \sigma'(\sigma_1(T_i)) = T_i\text{ for all }1 \leq i \leq n.
- \text{And }\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash \sigma^{-1}(S_k) \Leftarrow \text{Gen}(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)).
\]

Thus, per definition of the propagation algorithm, if we define \(S'' \triangleq \sigma'(S_2)\), we have \(S \cup S'' = \sigma'(S')\). Moreover, we have:

\[
\begin{align*}
\text{Gen}(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)) \\
= \sigma'(\text{Gen}(\sigma^{-1} \circ \sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma_1(S_1))) \\
= \sigma'(\text{Gen}(\sigma^{-1} \circ \sigma^{-1}(\Gamma); x_1 : x_1 : \sigma^{-1}(T_1); \ldots; x_n : \sigma^{-1}(T_n), \sigma_1(S_1))) \\
= \sigma'(\text{Gen}(\sigma_1 \circ \sigma^{-1}(\Gamma); x_1 : \sigma_1(T_1); \ldots; x_n : \sigma_1(T_n), \sigma_1(S_1))) \\
\end{align*}
\]

We can then see that there exist some set of type variables \(\pi\) such that

\[
\forall \pi. \text{Gen}(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)) = \sigma'(S_2)
\]

This implies that

\[
\begin{align*}
\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash \sigma^{-1}(S_k) \Leftarrow \text{Gen}(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)) \\
\Rightarrow \sigma^{-1}(\Gamma) \vdash \sigma^{-1}(S_k) \Leftarrow \text{Gen}(\sigma^{-1}(\Gamma); x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)) \\
\Rightarrow \sigma^{-1}(\Gamma) \vdash \sigma^{-1}(S_k) \Leftarrow \forall \pi. \text{Gen}(\Gamma; x_1 : T_1; \ldots; x_n : T_n, \sigma' \circ \sigma_1(S_1)) \\
\Rightarrow \sigma^{-1}(\Gamma) \vdash \sigma^{-1}(S_k) \Leftarrow \sigma'(S_2)
\end{align*}
\]

We can then use the lemma 15 to prove the statement \(\Gamma \vdash S_k \Leftarrow \sigma \circ \sigma'(S_2)\).

Finally, as we have \(\Gamma \vdash S_k \Leftarrow \sigma \circ \sigma'(S_2)\), we have \(\Gamma \vdash S_k \Leftarrow \sigma(S \cup \sigma'(S_2))\), i.e. \(\Gamma \vdash S_k \Leftarrow \sigma \circ \sigma'(S')\).
Propagation Termination. In this paragraph, we prove that, given a typable program, the propagation algorithm always stops. To prove such a property, we first define the completion and the cardinality of a process type.

**Definition 33.** Let suppose given a typing environment $\Gamma$ and a process type $S$. The completion of $S$ given $\Gamma$ is the smallest process type $S'$ such that $S \subseteq S'$ and for all $e \in dc(S')$, all $T \in S'(e)$ and all $T'$ with $\Gamma \vdash T \Leftarrow T'$, we have $T' \in S'(e)$. We note such a process type $\text{Cmp}(\Gamma, S)$.

Let remark that with the lemma 13, all process type have a completion type.

**Definition 34.** We define the cardinality of a process type $S$ (and note it $\#(S)$) inductively as follow:

$$
\#(\emptyset) \triangleq 0 \quad \#(\zeta) \triangleq 0 \quad \#(S \cup e) \triangleq \begin{cases} 
\#(S) + 1 & \text{If } e \notin dc(S) \\
\#(S) & \text{Else}
\end{cases}
$$

$$
\#(S \cup e : (T)) \triangleq \begin{cases} 
\#(S) + 1 & \text{If } e : (T) \subset S \\
\#(S) & \text{Else}
\end{cases}
$$

We now propose two preliminary lemma to introduce the proof of the property. The first lemma states that a process type computed by the propagation algorithm is always contained in a specific process type:

**Lemma 19.** Let suppose given a valid typing statement $\Gamma \vdash D : S$. Then, for all $\Gamma \vdash D : \emptyset \rightarrow^* \Gamma' \vdash D : S'$, there exist a permutation $\sigma$ such that:

- $\sigma(\Gamma') = \Gamma$.
- $\Gamma' \vdash S \Leftarrow \sigma(S')$.
- $\sigma(S') \subset \text{Cmp}(\Gamma, S)$.

Moreover, we have $fv(S') \cap \mathcal{V}^s = \emptyset$.

**Proof.** By induction on the propagation algorithm execution:

- Case $\Gamma' = \Gamma$ and $S' = \emptyset$. We have the result with the identity substitution.
- Case $\Gamma \vdash D : \emptyset \rightarrow^* \Gamma_k \vdash D : S_k \rightarrow \Gamma' \vdash D : S'$. Using the induction hypothesis, we have $fv(S_k) \cap \mathcal{V}^s = \emptyset$, and there exist $\sigma_k$ such that
  - $\sigma_k(\Gamma_k) = \Gamma$.
  - $\Gamma \vdash S \Leftarrow \sigma_k(S_k)$.
  - $\sigma_k(S_k) \subset \text{Cmp}(\Gamma, S)$.

Per construction, we can then easily see that $fv(S') \cap \mathcal{V}^s = \emptyset$. We can then apply the lemma 18 to get a substitution $\sigma'$ such that:

- $\Gamma_k = \sigma'(\Gamma')$. This trivially implies that $\Gamma = \sigma_k \circ \sigma'(\Gamma')$.
- $\Gamma \vdash S \Leftarrow \sigma_k \circ \sigma'(S')$. 


Let define $\sigma \triangleq \sigma_k \circ \sigma'$. If we take $e \in dc(S')$ and $T \in \sigma(S')(e)$, per
definition, there exist $T' \in S(e)$ such that $\Gamma \vdash T' \Leftarrow T$: hence, we have
$T \in (\bar{C}\Gamma S)(e)$. We can then conclude that $\sigma(\Gamma') = \Gamma$, $\Gamma \vdash S \Leftarrow \sigma(S')$
and $\sigma(S') \subset C\text{mp}(\Gamma, S)$.

The second lemma states that the type computed by the propagation algorithm
always grow bigger:

**Lemma 20.** Let suppose given a valid propagation statement $\Gamma \vdash D : S \rightarrow
\Gamma' \vdash D : S'$ with $\text{fs}(S) \cap \inV = \emptyset$. Let also suppose that there exist a
permutation $\sigma$ and a process type $S''$ such that $\sigma(\Gamma') = \Gamma$ and $S \cup S'' = \sigma(S')$. Then, we
have $\#(S') + 1 \geq (S)$.

**Proof.** Per construction, we have $\#(S'') = 1$, and as $\sigma$ is a permutation, we
have $\#(\sigma(S)) = \#(S)$. Per definition of the propagation algorithm, there exist a
substitution $\sigma'$ and a process type $S_k$ such that $\Gamma'' = \sigma'(\Gamma)$ and $S' = \sigma(S_k) \cup S_k$.
We thus have that $\sigma'(\alpha_1) \neq \sigma(\alpha_2)$ and $\sigma'(\alpha_1) \in \inV$ for all $\alpha_1 \neq \alpha_2 \in \text{fs}(\Gamma) \cup \text{fs}(S)$.
Hence, we have $S'' \not\subset \sigma(S)$: per definition of the propagation algorithm, $\#(S') = 
\#(S) + 1$ holds.

**Lemma 21.** Let suppose given a valid typing statement $\Gamma \vdash D : S$. Then, the
set $K$, inductively defined as $(\Gamma, \emptyset) \in K$ and $(\Gamma', S') \in K \Rightarrow \text{next}(\Gamma' \vdash D : S') \subset 
K$, is finite.

**Proof.** Let suppose on the contrary that $K$ is infinite. This implies that for all
$1 \leq i$, there exists $(\Gamma_i, S_i) \in K$ such that $(\Gamma_{i+1}, S_{i+1}) \in \text{next}(\Gamma_i \vdash D : S_i)$, with
$(\Gamma_1, S_1) = (\emptyset, \emptyset)$. We can use the lemma 18 to have for all $1 \leq i$ a permutation
$\sigma_i$ and a process type $S'_i$ such that $\sigma_i(\Gamma_{i+1}) = \Gamma_i$ and $\sigma_i(S_{i+1}) = S_i \cup S'_i$.
Let then consider the series $(\#(S_i))_{1 \leq i}$: with the lemma 20, we can see that
$\#(S_i) = \#(S'_i) \leq 1$ for all $1 \leq i$. But let consider the lemma 19: for all $1 \leq i$
there exist a permutation $\sigma'_i$ with $\sigma'_i(S_i) \subset \bar{C}\Gamma S$. We thus have for all $1 \leq i$
$\#(S_i) = \#(\sigma'_i(S_i)) \leq (\#(\bar{C}\Gamma S)$, which is impossible as $\#(\bar{C}\Gamma S)$ is finite.

Our hypothesis of $K$ being infinite is thus false, which gives us the result.

**Typing Property.** We prove here a main property of the propagation algorithm:
when it stops, it gives a valid type for the given program. This result is nonetheless
defined according to a definition of failure of the propagation algorithm.
Indeed, in the $R$-inference algorithm, a failure was identified as the impossibility
to define a substitution validating the computed constraint. Here, our definition
of failure is a little different:

**Definition 35.** Let suppose given a typing statement $\Gamma \vdash D : S$ (let note that
we don’t suppose that this statement is valid). We say that a propagation error
occurs on this statement if there exist:

- A sub-program $(e_1 : x_1; \ldots; e_n : x_n / R) \subset D$.
- A tuple $(T_1, \ldots, T_n) \in S(e_1) \times \cdots \times S(e_n)$.
- A valid statement $\text{fs}(\Gamma) \cup \text{fs}(S), \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash R : [F] S' \mid C$ such
  that there is no substitution $\sigma$ with $C \Rightarrow \sigma$. 

Lemma 22. Let suppose given a typing statement \( \Gamma \vdash D : S \) such that next\((\Gamma \vdash D : S) = \emptyset \) and no propagation error occurs on \( \Gamma \vdash D : S \). Then \( \Gamma \vdash D : S \) is valid.

Proof. Let consider a sub-program \((e_1 : x_1 ; \ldots ; e_n : x_n / R) \) \( \subset D \) and \((T_1, \ldots , T_n) \in S(e_1) \times \cdots \times S(e_n)\). Let note \( \Gamma' \triangleq \Gamma ; x_1 : T_1 ; \ldots ; x_n : T_n \). As no propagation error occurs on \( \Gamma \vdash D : S \), the statements \( f_o(\Gamma') \cup f_o(S) \), \( \Gamma' \vdash R : [F] S' \cup C \) and \( C \Rightarrow \sigma \) hold. Using the corollary 3, there exist a type derivation of \( \sigma(\Gamma') \vdash \Gamma \vdash D : S \) in \( \emptyset \). By definition of the propagation algorithm, \( \sigma(\Gamma') \vdash \sigma(S') \) is injective have its image in \( V \), and \( \Gamma' \vdash \sigma(S') \) holds. Let then define the substitution \( \sigma' \) such that \( \text{dom}(\sigma') = \sigma(\Gamma') \) and for all \( \alpha \in \text{fv}(\Gamma') \cup \text{fv}(S) \), we have \( \sigma' \circ \sigma(\alpha) = \alpha \). We thus have \( \sigma'(\sigma(\Gamma'), \sigma(S')) \) holds. Thus, using the lemma 16, \( \Gamma' \vdash R : S \) holds.

Finally, we can use the lemma 10 to have the result.

Error-free Propagation. In this paragraph, we prove that given a typable program, the propagation algorithm never has any error.

Lemma 23. Let suppose given a valid typing statement \( \Gamma \vdash D : S \). Let define inductively the set \( K \) as \((\Gamma, \emptyset) \in K \) and \((\Gamma', S') \in K \Rightarrow \text{next}(\Gamma' \vdash D : S') \subset K \). Then, for all \((\Gamma', S') \in K \), the statement \( \Gamma' \vdash D : S' \) has no propagation error.

Proof. As \((\Gamma', S') \in K \), we have \((\Gamma, \emptyset) \rightarrow^* (\Gamma', S') \). Using the lemma 19, there exist a permutation \( \sigma \) such that \( \sigma(\Gamma') = \Gamma \) and \( \Gamma \vdash S \subseteq \sigma(S') \). Let then take \((e_1 : x_1 ; \ldots ; e_n : x_n / R) \subset D \), \((T_1, \ldots , T_n) \in S'(e_1) \times \cdots \times S'(e_n)\). Let also note \( \Gamma'' \triangleq \Gamma' ; x_1 : T_1 ; \ldots ; x_n : T_n \) and take a valid statement \( f_o(\Gamma') \cup f_o(S') \), \( \Gamma'' \vdash R : [F_k] S_k \cup C_k \).

As we have \( \Gamma \vdash S \subseteq \sigma(S') \), for all \( 1 \leq i \leq n \), there exist \( T^k_i \in S(e_i) \) such that \( \Gamma \vdash T^k_i \subseteq \sigma(T_i) \). Then, using the lemma 10, the typing statement \( \Gamma ; x_1 : T^k_1 ; \ldots ; x_n : T^k_n \vdash R : S \) holds. We also use the lemma 14 to get a derivation of \( \Gamma ; x_1 : \sigma(T_1) ; \ldots ; x_n : \sigma(T_n) \vdash R : S \). Let then use the lemma 7: the typing statement \( \Gamma' ; x_1 : T_1 ; \ldots ; x_n : T_n \vdash R : \sigma^{-1}(S) \) holds. We can the conclude using the corollary 4: there exist a substitution \( \sigma_k \) such that \( C_k \Rightarrow \sigma_k \).

C.3 Main Properties

Definition 36. Let suppose given a statement \( \Gamma' \vdash D : S' \). We say that this statement is terminal iff no propagation error occurs on it, and no propagation rule can be applied.

Theorem 17. Let suppose given a valid typing statement \( \Gamma \vdash D : S_k \). Then there exist a terminal statement \( \Gamma' \vdash D : S' \) such that \((\Gamma, \emptyset) \rightarrow^* (\Gamma', S') \). Moreover, the statement \( \Gamma' \vdash D : S' \) holds, and there exist a permutation \( \sigma \) such that \( \Gamma = \sigma(\Gamma') \) and \( \Gamma \vdash S_k \subseteq \sigma(S') \).
Proof. Let define inductively the set $K$ as $(\Gamma, \emptyset) \in K$ and $(\Gamma', S') \in K \Rightarrow \text{next}(\Gamma' \vdash D : S') \subset K$. Using the lemma 21, $K$ is finite, which implies that there exist $\Gamma' \vdash D : S'$ such that:

- $(\Gamma, \emptyset) \rightarrow^* \Gamma' \vdash D : S'$.
- $\text{next}(\Gamma' \vdash D : S') = \emptyset$.

Using the lemma 23, $\Gamma' \vdash D : S'$ has no propagation error: it is then terminal. We can then apply the lemma 22 to get that $\Gamma' \vdash D : S'$ holds. Finally, we can use the lemma 19 to get a permutation $\sigma$ such that $\Gamma = \sigma(\Gamma')$ and $\Gamma \vdash S_k \Leftarrow \sigma(S')$.

**Corollary 7.** Let suppose given a valid statement $\Gamma \vdash D : S$. Then there exist $S'$ such that $\Gamma' \vdash m D : S'$ holds.

Proof. Let consider any process type $S_k$ such that $\Gamma \vdash D : S_k$. Using the theorem 17, there exist a terminal statement $\Gamma' \vdash D : S'$ and a permutation $\sigma$, which does not depend on $S_k$ (per definition of the propagation algorithm), such that:

- $(\Gamma, \emptyset) \rightarrow^* \Gamma' \vdash D : S'$.
- $\Gamma' \vdash D : S'$ holds.
- $\Gamma = \sigma(\Gamma')$ and $\Gamma \vdash S_k \Leftarrow \sigma(S')$.

Using the lemma 7, the statement $\Gamma \vdash D : \sigma(S')$ holds. As $\Gamma \vdash S_k \Leftarrow \sigma(S')$ for all valid process type $S_k$ and $\text{fv}(\sigma(S')) \subset \text{fv}(\Gamma)$ per construction, we have the result.
D Inference undecidability

D.1 Introduction

In this section, we use the propagation to compute some process types for the
programs in our encoding, with an empty typing environment. Indeed, the pro-
grams used in the encoding of PCP are all without any free variables, and thus,
having a non-empty typing environment is not necessary. This implies that the
computed types have no free variables per construction.

As a result, we simplify the presentation of the propagation algorithm in this
case:

\[
\text{P:Message} \quad \exists(e_1 : x_1; \ldots; e_n : x_n / R) \subset D \quad \forall 1 \leq i \leq n, \exists T_i \in S(e_i) \\
\emptyset, x_1 : T_1; \ldots; x_n : T_n \vdash_R [F'] S' | C \quad C \Rightarrow \sigma \quad \text{Gen}(\emptyset, \sigma(S')) \not\subset S \\
\emptyset \vdash D : S \rightarrow \emptyset \vdash D : S \cup \text{Gen}(\emptyset, \sigma(S'))
\]

\[
\text{P:Channel} \quad \exists e \in \text{I}(D) \setminus \text{dc}(S) \\
\Gamma \vdash D : S \rightarrow \emptyset \vdash D : S \cup e
\]

One can then remark that in the rule P:Message, we have \(x_1 : T_1; \ldots; x_n : T_n \vdash_m R : \text{Gen}(\emptyset, \sigma(S'))\).

We also simplify the set next(\(\Gamma \vdash D : S\)) into next_s(\(D : S\)):

**Definition 37.** Let suppose given a program \(D\) and a process type \(S\). We write
\(\text{next}_s(D : S)\) the set
\[
\{S' \mid \emptyset \vdash D : S \rightarrow \emptyset \vdash D : S'\}
\]

Finally, we use a special process type to ease the presentation of the different
proofs:

**Definition 38.** Let \(D\) be a program. We write \(\text{dct}(D)\) the process type such that:
- \(\text{dc} (\text{dct}(D)) = \text{dc}(D)\).
- For all \(e \in \text{dc}(D)\), we have \(\text{dct}(D)(e) = \emptyset\).

D.2 Preliminary results

**Lemma 24.** Let suppose given a program \(D\) such that for all \((e_1 : x_1; \ldots; e_n : x_n / D') \subset D \Rightarrow n \leq 1\), and two closed process types \(S_1, S_2\). The following equality holds:
\[
\text{next}_s(D : S_1 \cup S_2) = \left(\{S' \cup S_2 \mid S' \in \text{next}_s(D : S_1)\}\right) \cup \{S' \cup S_1 \mid S' \in \text{next}_s(D : S_2)\}
\]
We then have $S \in \text{next}_s(D : S_1 \cup S_2)$. We have three cases, depending on the propagation rule applied:

1. Case P:MESSAGE with $(\emptyset / R) \subseteq D$ and $\emptyset \vdash R : S'$ with $S' \not\subseteq S_1 \cup S_2$ and $S = S_1 \cup S_2 \cup S'$. We can see in this case that we can apply the same propagation rule on $\emptyset \vdash D : S_1$ (resp. $\emptyset \vdash D : S_2$). This gives the statement $\emptyset \vdash D : S_1 \cup S'$ (resp. $\emptyset \vdash D : S_2 \cup S'$).

2. Case P:MESSAGE with $(e : x/R) \subseteq D$ and $x : T \vdash R : S'$ with $T \in (S_1 \cup S_2)(e)$, $S' \not\subseteq S_1 \cup S_2$ and $S = S_1 \cup S_2 \cup S'$. As the problem is symmetric, we can suppose that $T \in S_1(e)$. We can thus apply the same propagation rule on $\emptyset \vdash D : S_1$, which gives the statement $\emptyset \vdash D : S_1 \cup S'$.

3. Case P:CHANNEL with $e \in I(D) \setminus \text{dc}(S_1 \cup S_2)$ and $S = S_1 \cup S_2 \cup e$. As $\text{dc}(S_1) \subseteq \text{dc}(S_1 \cup S_2)$, we then have $e \in I(D) \setminus \text{dc}(S_1)$. We can thus apply the same propagation rule on $\emptyset \vdash D : S_1$, which gives the statement $\emptyset \vdash D : S_1 \cup e$.

From these different cases, we can see that next$_s(D : S_1 \cup S_2) \subseteq \{S' \cup S_2 \mid S' \in$ next$_s(D : S_1)\} \cup \{S' \cup S_1 \mid S' \in$ next$_s(D : S_2)\} \setminus \{S' \mid S' \subseteq S_1 \cup S_2\}$.

Let take $S \in$ next$_s(D : S_1) \setminus \{S' \mid S' \subseteq S_1 \cup S_2\}$: we have $S \not\subseteq S_1 \cup S_2$. We have three cases, depending on the propagation rule applied:

1. Case P:MESSAGE with $(\emptyset / R) \subseteq D$: there exist $S'$ with $\emptyset \vdash R : S'$, $S' \not\subseteq S_1$ and $S = S_1 \cup S'$. As $S \not\subseteq S_1 \cup S_2$, we have $S' \not\subseteq S_1 \cup S_2$. We can see in this case that we can apply the same propagation rule on $\emptyset \vdash D : S_1 \cup S_2$. This gives the statement $\emptyset \vdash D : S_1 \cup S_2 \cup S'$, i.e. $\emptyset \vdash D : S_3 \cup S$.

2. Case P:MESSAGE with $(e : x/R) \subseteq D$: there exist $S'$ and $T \subseteq S_1(e)$ with $x : T \vdash R : S'$, $S' \not\subseteq S_1$ and $S = S_1 \cup S'$. As $S \not\subseteq S_1 \cup S_2$, we have $S' \not\subseteq S_1 \cup S_2$. We can thus apply the same propagation rule on $\emptyset \vdash D : S_1 \cup S_2$, which gives the statement $\emptyset \vdash D : S_1 \cup S_2 \cup S'$, i.e. $\emptyset \vdash D : S_3 \cup S$.

3. Case P:CHANNEL with $e \in I(D) \setminus \text{dc}(S_1 \cup S_2)$ and $S = S_1 \cup S_2 \cup e$. As $S \not\subseteq S_1 \cup S_2$, we have $e \in I(D) \setminus \text{dc}(S_1 \cup S_2)$. We can thus apply the same propagation rule on $\emptyset \vdash D : S_1$, which gives the statement $\emptyset \vdash D : S_1 \cup S_2 \cup e$, i.e. $\emptyset \vdash D : S_2 \cup S$.

We then have $\{S' \cup S_2 \mid S' \in$ next$_s(D : S_1)\} \setminus \{S' \mid S' \subseteq S_1 \cup S_2\} \subseteq$ next$_s(D : S_1 \cup S_2)$. We can use the same approach with $S \subseteq$ next$_s(D : S_1) \setminus \{S' \mid S' \subseteq S_1 \cup S_2\}$. This gives us $\{S' \cup S_2 \mid S' \in$ next$_s(D : S_1)\} \setminus \{S' \mid S' \subseteq S_1 \cup S_2\} \subseteq$ next$_s(D : S_1 \cup S_2)$.

We thus have the result.

**Lemma 25.** Let suppose given a program $D$ and three closed process types $S_1$, $S_2$, $S_3$ such that:

- for all for all $(e_1 : x_1; \ldots ; e_n : x_n / D') \subseteq D \Rightarrow n \leq 1$.
- For all $S' \in$ next$_s(D : S_1)$, we have $S' \subseteq S_1 \cup S_2$.
- $\emptyset \vdash D : S_2 \rightarrow^* \emptyset \vdash D : S_3$.

Then, there exists $S_4$ such that:
Proof. By induction on the propagation derivation:

\[ \emptyset \vdash D : S_1 \cup S_2 \rightarrow^* \emptyset \vdash D : S_4. \]

\[ S_4 \equiv S_1 \cup S_3. \]

Moreover, if no propagation rule can be applied on \( \emptyset \vdash D : S_3 \), no propagation rule can be applied on \( \emptyset \vdash D : S_4 \).

**Proof.** By induction on the propagation algorithm application:

- Case \( S_3 = S_2 \): no propagation rule have been applied on \( \emptyset \vdash D : S_2 \). Let then define \( S_4 \triangleq S_1 \cup S_2 \). We easily have \( \emptyset \vdash D : S_1 \cup S_2 \rightarrow^* \emptyset \vdash D : S_4 \).

Then, using the lemma 24, we have:

\[
\text{next}_s(D : s_4) = \left( \{S' \cup S_2 \mid S' \in \text{next}_s(D : s_1)\} \cup \{S' \cup S_3 \mid S' \in \text{next}_s(D : s_2)\} \right) \setminus \{S' \mid S' \subset S_1 \cup S_2\}.
\]

Using the second hypothesis we thus have that no propagation rule can be applied on \( \emptyset \vdash D : S_4 \) when \( \text{next}_s(D : s_3) = \emptyset \).

- Case \( \emptyset \vdash D : S_2 \rightarrow^* \emptyset \vdash D : S_2' \). By the induction hypothesis, there exists \( S_4 \) such that
  
  \[ \emptyset \vdash D : S_1 \cup S_2' \rightarrow^* \emptyset \vdash D : S_4. \]
  
  \[ S_4 \equiv S_1 \cup S_3. \]

With \( \emptyset \vdash D : S_3 \rightarrow^* \emptyset \vdash D : S_4 \), we have two cases:

1. If \( S_1 \cup S_2' \equiv S_1 \cup S_2 \), we trivially have \( \emptyset \vdash D : S_1 \cup S_2 \rightarrow^* \emptyset \vdash D : S_4 \).

2. In the other case, we can apply on \( S_1 \cup S_2 \) the same propagation rule as the one used in \( \emptyset \vdash D : S_2 \rightarrow \emptyset \vdash D : S_2' \). This would then give the statement \( \emptyset \vdash D : S_1 \cup S_2' \), which gives us the result.

**Lemma 26.** Let suppose given two programs \( D, D' \) with \((\emptyset / D') \subset D\) and two process types \( S_1, S_2 \) such that \( \emptyset \vdash D' : S_1 \rightarrow^* \emptyset \vdash D' : S_2 \) holds. Then the statement \( \emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2 \) holds.

**Proof.** By induction on the propagation derivation:

- If \( S_2 = S_1 \), the result is evident.

- Let suppose there exists \( S \) such that \( \emptyset \vdash D : S_1 \rightarrow \emptyset \vdash D' : S \) and \( \emptyset \vdash D : S \rightarrow^* \emptyset \vdash D' : S_2 \) hold. We have two cases, considering the propagation rule used:

  1. Case P:MESSAGE. As we have \((\emptyset / D') \subset D\), we have \((e_1 : x_1; \ldots ; e_n : x_n/R) \subset D\). Thus, as we have \( S' \not\subset S_1 \), we can apply the same propagation rule on \( \emptyset \vdash D : S_1 \), which would give \( \emptyset \vdash D : S \). Then, using the induction hypothesis, we have the result.

  2. Case P:CHANNEL. As we have \((\emptyset / D') \subset D\), we have \( I(D') \subset I(D) \). This implies that \( s \in I(D) \setminus dc(S_1) \): we can then apply the same propagation rule to get \( \emptyset \vdash D : S \). Using the induction hypothesis, we have then the result.

**Theorem 18.** Let suppose given:
Then the statement $\emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2 \cup S_4$ holds, and we have:

$$\text{next}_s(D : S_1 \cup S_2) = \left( \{ S' \cup S_2 \mid S' \in \text{next}_s(D : S_1) \} \cup \{ S' \mid S' \in \text{next}_s(D : S_2) \} \right) \setminus \{ S' \mid S' \subseteq S_1 \cup S_2 \}$$

**Proof.** Using the lemma 26, the statement $\emptyset \vdash D : S_3 \rightarrow^* \emptyset \vdash D : S_4$ holds. Using the lemma 24, the last equality holds too. Let’s prove the result by induction on the derivation of $\emptyset \vdash D : S_3 \rightarrow^* \emptyset D : S_4$:

- Case $S_4 = S_3$: as $S_2 \cup S_4 = S_2$ we have per hypothesis $\emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2 \cup S_4$.
- Case $\emptyset \vdash D : S_3 \rightarrow \emptyset \vdash D : S_3'$ and $\emptyset \vdash D : S_3' \rightarrow^* \emptyset \vdash D : S_4$. In the case that $S_3' \subseteq S_2$, we have the result with a simple application of the induction hypothesis. Let now suppose that $S_3' \not\subseteq S_2$ and define $S_3' \triangleq S_3 \cup S'$. As $S_3' \not\subseteq S_2$, we have $S' \not\subseteq S_2$. We can then apply the same propagation rule on $\emptyset \vdash D : S_2$, which gives us the statement $\emptyset \vdash D : S_2 \cup S'$. We then have a derivation of $\emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2 \cup S'$, with $S_3' \subseteq S_2 \cup S'$. We can then apply the induction hypothesis on $S_3'$ and $S_4$, which would give us a derivation of $\emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2 \cup S' \cup S_4$. Per definition of the propagation algorithm, we have $S' \subseteq S_4$: this implies that $S_2 \cup S' \cup S_4 = S_2 \cup S_4$.

### D.3 Elements of types for the different programs

The following results compute a valid closed type for each program in the encoding of PCP. As these big types are computed using the propagation algorithm, we will use the notation ‘...’ to denote the type we just computed.

**Program $D_a(u, v)$**

**Lemma 27.** Let suppose given a structure $k = (L_1, L_2, L_3)$ such that the three elements $L_i$ are lists of word pairs. Then there exists a closed process type $S$ such that:

- $\emptyset \vdash D_a(u, v) : \text{dct}(D_a(u, v)) \cup \text{input}_{u,v} : (\|k\|_T) \rightarrow^* \emptyset \vdash D_a(u, v) : S$.
- No propagation rule can be applied on $\emptyset \vdash D_a(u, v) : S$.
- $S(\text{output}_{u,v}) = \{ \|k'\|_T \} \text{ where } k' \in (\|L_2, L_3\|)$ and $\text{lts}(L_3) = \{ (u', v') \mid (u', v' \in \text{lts}(L_1)) \}$.

**Proof.** We prove the result by induction on $L_1$:

- Two programs $D, D'$ such that $(\emptyset/D') \subseteq D$ holds and for all $(e_1 : x_1; \ldots; e_n : x_n/D') \subseteq D \Rightarrow n \leq 1$.
- Two closed process types $S_1, S_2$ such that $\emptyset \vdash D : S_1 \rightarrow^* \emptyset \vdash D : S_2$ holds.
- Two closed process types $S_3, S_4$ with $S_3 \subseteq S_2$, such that $\emptyset \vdash D' : S_3 \rightarrow^* \emptyset \vdash D' : S_4$ holds.
− Case $L_1 = \emptyset$. We have

$$\emptyset \vdash D_a(u, v) : \text{dct}(D_a(u, v)) \cup \text{input}_{u,v} : (\|k\|_T)$$

$$\rightarrow \emptyset \vdash D_a(u, v) : \cdots \cup \text{finish}_{u,v} : \{\{\text{tmp} : \text{Pre}(\|k\|_T)\}; \text{Abs}\}$$

$$\rightarrow \emptyset \vdash D_a(u, v) : \cdots \cup \text{output}_{u,v} : (\|k\|_T)$$

We can easily see that the resulting type has the stated properties.

− Case $L_1 = (u_1, v_1) :: L'$. We have

$$\emptyset \vdash D_a(u, v) : \text{dct}(D_a(u, v)) \cup \text{input}_{u,v} : (\|k\|_T)$$

$$\rightarrow \emptyset \vdash D_a(u, v) : \cdots \cup \text{loop}_{u,v} : \{(\text{head} : \text{Pre}((\|u_1, v_1\|_T)); \text{tail} : \text{Pre}((\|L'\|_T)); \text{Abs}^{(\text{head}, \text{tail}, \text{tmp})})\}$$

$$\rightarrow \emptyset \vdash D_a(u, v) : \cdots \cup \text{tmp}_{u,v} : (\|k\|_T)$$

We note $S_1$ this resulting type.

$$\rightarrow \emptyset \vdash D_a(u, v) : \cdots \cup \text{input}_{u,v} : (\|L', L_2, (u, u_1, v, v_1) :: L_3\|_T)$$

Let’s write $S_2$ the set type $\text{input}_{u,v} : (\|L', L_2, (u, u_1, v, v_1) :: L_3\|_T)$. We can use the induction hypothesis, which gives us a set type $S$ such that

− $\emptyset \vdash D_a(u, v) : \text{dct}(D_a(u, v)) \cup S_2 \rightarrow \ast \emptyset \vdash D_a(u, v) : S$.

− No propagation rule can be applied on $\emptyset \vdash D_a(u, v) : S$.

− $S(\text{output}_{u,v}) = \{\|k'\|_T\}$ where $k' = (\|L_2, L_3\|)$ and $\text{lts}(L'_3) = \text{lts}((u, u_1, v, v_1) :: L_3) \cup \{(u, u', v, v') \mid (u', v' \in \text{lts}(L'))\}$.

Moreover, per construction we have $\text{next}_a(D_a(u, v) : S_1) \subseteq \{S_1 \cup S_2\}$. We can then conclude with the lemma 25:

− We have the following propagation derivation

$$\emptyset \vdash D_a(u, v) : \text{dct}(D_a(u, v)) \cup \text{input}_{u,v} : (\|k\|_T)$$

$$\rightarrow \emptyset \vdash D_a(u, v) : S_1 \cup S_2$$

$$\rightarrow \emptyset \vdash D_a(u, v) : S_1 \cup S$$

− No propagation rule can be applied on $\emptyset \vdash D_a(u, v) : S_1 \cup S$.

− The resulting list $L'_3$ is such that:

$$\text{lts}(L'_3)$$

$$= \text{lts}((u, u_1, v, v_1) :: L_3) \cup \{(u, u', v, v') \mid (u', v' \in \text{lts}(L'))\}$$

$$= \text{lts}(L_3) \cup \{(u, u_1, v, v_1) \cup \{(u, u', v, v') \mid (u', v' \in \text{lts}(L'))\}\}$$

$$= \text{lts}(L_3) \cup \{(u, u', v, v') \mid (u', v' \in \text{lts}(L'))\}$$

Thus, $\text{lts}(L') = \{\|L'\|_T\}$.

D.4 Program $D_n$

Lemma 28. Let suppose given a list $L$ such that $\text{lts}(L) = L(m)$ for some $m \in \mathbb{N}$. Then, there exists a closed process type $S$ such that:

− $\emptyset \vdash D_n : \text{dct}(D_n) \cup \text{input} : (\|\|L\|_T\|) \rightarrow \ast \emptyset \vdash D_n : S$.

− No propagation rule can be apply on $\emptyset \vdash D_n : S$.

− $S(\text{output}) = L'$ where $L'$ is such that $\text{lts}(L') = L(m + 1)$.
Proof. First, let’s prove that for all $1 \leq i \leq n$, there exists $S_i$ such that:

- $\emptyset \vdash D_n : \text{dct}(D_n) \cup \text{input} : (\|(L)\|_T)$
- next$_S(D_n : S_i)$ is the singleton $\{S_i \cup \text{input}_{u_i,v_i} : (\|(L, L_i)\|_T)\}$ where $\text{lts}(L_i) = \{(u_k,u,v_k,v) \mid 1 \leq k < i \wedge (u,v) \in L\}$.

We prove this first result by induction on $i$:

- Case $i = 1$: the set type $S_1 = \text{dct}(D_n) \cup \text{input} : (\|(L)\|_T)$ gives us the result in this case.
- Case $i = i' + 1$ with $1 \leq i' < n$. Let define $S_{i'} \triangleq \text{dct}(D_n) \cup \text{input} : (\|(L)\|_T)$

Using the induction hypothesis, there exists $S_{i'}$ such that:

- $\emptyset \vdash D_n : S' \longrightarrow^* \emptyset \vdash D_n : S_{i'}$.
- next$_S(D_n : S_{i'})$ is the singleton $\{S_{i'} \cup \text{input}_{u_i,v_i} : (\|(L, L_{i'})\|_T)\}$ where $\text{lts}(L_{i'}) = \{(u_k,u,v_k,v) \mid 1 \leq k < i' \wedge (u,v) \in L\}$.

We can then apply the lemma 27 with the type input$_{u_{i'},v_{i'}} : (\|(L, L, L_{i'})\|_T)$

there exists a set type $S$ such that:

- $\emptyset \vdash D_n(u_{i'}, v_{i'}) : \text{dct}(D_n(u_{i'}, v_{i'})) \cup \text{input}_{u_{i'}, v_{i'}} : (\|(L)\|_T)$
- No propagation rule can be applied on $\emptyset \vdash D_n(u_{i'}, v_{i'}) : S$.
- $S$ is the singleton $\{(k',L,L')\}$ where $k' = ([],L,L')$ and $\text{lts}(L') = \{(u_k,u,v_k,v) \mid 1 \leq k < i(u,v) \in L\}$.

We can also remark that $\text{dct}(D_n(u_{i'}, v_{i'})) \cup \text{input}_{u_{i'}, v_{i'}} : (\|(L, L, L_{i'})\|_T) \subset S_{i'}$.

We can then apply the theorem 18 to get a valid propagation derivation of $\emptyset \vdash D_n : S' \longrightarrow^* \emptyset \vdash D_n : S_{i'} \cup S$. As no propagation rule can be applied on $\emptyset \vdash D_n(u_{i'}, v_{i'}) : S$, we can see that next$_S(D_n : S) = \{S \cup \text{input}_{u_i,v_i} : (\|(L, L, L_i)\|_T)\}$

We thus have:

$$
\text{next}_S(D_n : S_{i'} \cup S) = \begin{cases}
\{S' \mid S' \in \text{next}_S(D_n : S_{i'})\} \\
\cup \{S' \mid S' \in \text{next}_S(D_n : S)\}
\end{cases}
\setminus \{S' \mid S' \subset S_{i'} \cup S\}
= \begin{cases}
\{S \cup S_{i'} \cup \text{input}_{u_i,v_i} : (\|(L, L, L_i)\|_T)\} \\
\cup \{S' \mid S' \in \text{next}_S(D_n : S)\}
\end{cases}
\setminus \{S' \mid S' \subset S_{i'} \cup S\}
= \{S_{i'} \cup S \cup \text{input}_{u_i,v_i} : (\|(L, L, L_i)\|_T)\}
\setminus \{S' \mid S' \subset S_{i'} \cup S\}
= \{S_{i'} \cup S \cup \text{input}_{u_i,v_i} : (\|(L, L, L_i)\|_T)\}

We can now conclude, by defining $S_i = S_{i'} \cup S$.

Let’s now take $S^1_n \triangleq \text{input}_{u_n,v_n} : (\|(L, L, L_n)\|_T)$: we can use the previous lemma 27, which gives us the existence of $S^2_n$ such that:

- $\emptyset \vdash D_a(u_n, v_n) : \text{dct}(D_n) \cup S^1_n \longrightarrow^* \emptyset \vdash D_a(u_n, v_n) : S^2_n$.
- No propagation rule can be applied on $\emptyset \vdash D_a(u_n, v_n) : S^2_n$.
- $S^2_n$ is the singleton $\{k' \mid k' = ([],L,L')\}$ and $\text{lts}(L') = L(m + 1)$.
As no propagation rule can be applied on $\emptyset \vdash D_n(u_n, v_n) : S^2_n$, we can see that $\text{next}_s(D_n : S^2_n) = \{S^2_n \cup \text{output} : (L')\}$. We can then use the theorem 18 to get a valid propagation derivation of

$\emptyset \vdash D_n : \text{dct}(D_n) \cup \text{input} : (\|L\|_T)$

$\vdash^* \emptyset \vdash D_n : S_n \cup S^2_n$

$\vdash^* \emptyset \vdash D_n : S_n \cup S^2_n \cup \text{output} : (L')$

We can then conclude by noting $S$ this resulting type.

**Program $D_{eq}$**

**Lemma 29.** Let suppose given a structure $k \triangleq (u, v, k')$ such that both $u$ and $v$ are words. Then, there exists a closed process type $S$ such that:

- $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eq} : (\|k\|_T)$
- $\vdash^* \emptyset \vdash D_{eq} : S$.
- No propagation rule can be applied on $\emptyset \vdash D_{eq} : S$.
- $S(\text{test}_{fail}) = \{\|k\|_T\}$ and $S(\text{test}_{ok}) = \emptyset$ iff $u \neq v$.
- $S(\text{test}_{fail}) = \emptyset$ and $S(\text{test}_{ok}) = \{\text{Abs}^\emptyset\}$ iff $u = v$.

**Proof.** We use an approach by induction on $u$ and $v$:

- Case $u = \varepsilon$ and $v = a.v'$. We have:
  \[
  \emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eq} : (\|k\|_T)
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{na} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{tb} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{we} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{walk} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{ok} : (\{\text{Abs}^{\emptyset}\})
  \]

The algorithm finishes here, giving us the result.

- Case $u = \varepsilon$ and $v = b.v'$. We have:
  \[
  \emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eq} : (\|k\|_T)
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{na} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{tb} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{we} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{walk} : (\{\text{tmp} : \text{Pre}(\|k\|_T) ; \text{Abs}^{\text{tmp}}\})
  \]
  \[
  \vdash^* \emptyset \vdash D_{eq} : \cdots \cup \text{test}_{ok} : (\{\text{Abs}^{\emptyset}\})
  \]

The algorithm finishes here, giving us the result.
- Case $u = a.u'$ and $v = \varepsilon$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eqp} : (||k||_T)
\]

\[
\begin{align*}
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{ua} : \{a : \text{Pre}((u'')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(a, \text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{eqp}_{err} : \{\text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(\text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{fail} : (||k''||_T)
\end{align*}
\]

The algorithm finishes here, giving us the result.

- Case $u = a.u'$ and $v = a.v'$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eqp} : (||k||_T)
\]

\[
\begin{align*}
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{ua} : \{a : \text{Pre}((u'')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(a, \text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{ua} : \{a : \text{Pre}((v')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(a, \text{tmp})})\}
\end{align*}
\]

Let’s write $S_1$ this computed type: we can easily see that $\text{next}_s(D_{eq} : S_1) = \{S_1 \cup \text{input}_{eqp} : (||u', v', k'||_T)\}$. Let define the structure $k_2 \triangleq (u', v', k')$. We can apply the induction hypothesis with $k_2$, which gives us a set type $S'$ such that:

- $\emptyset \vdash D_{eq} : S_1 \rightarrow^* \emptyset \vdash D_{eq} : S'$.
- No propagation rule can be applied on $\emptyset \vdash D_{eq} : S'$.
- $S'(\text{test}_{fail}) = (||k'||_T)$ and $S(\text{test}_{ok}) = \emptyset$ if $u' \neq v'$.
- $S'(\text{test}_{fail}) = \emptyset$ and $S(\text{test}_{ok}) = \{\text{Abs}^{(a)}\}$ if $u' = v'$.

We can then apply the lemma 25, which gives us:

- $\emptyset \vdash D_{eq} : S_1 \rightarrow^* \emptyset \vdash D_{eq} : S_1 \cup S'$.
- $\text{next}_s(D_{eq} : S_1 \cup S') = \emptyset$.

As the propagation algorithm is transitive, the derivation $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eqp} : (||k||_T) \rightarrow^* \emptyset \vdash D_{eq} : S_1 \cup S'$ holds. We can then conclude by defining $S \triangleq S_1 \cup S'$.

- Case $u = a.u'$ and $v = b.v'$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eqp} : (||k||_T)
\]

\[
\begin{align*}
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{na} : \{b : \text{Pre}((u'')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(b, \text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{ui} : \{\text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(\text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{eqp}_{err} : \{\text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(\text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{fail} : (||k''||_T)
\end{align*}
\]

The algorithm finishes here, giving us the result.

- Case $u = b.u'$ and $v = \varepsilon$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{eqp} : (||k||_T)
\]

\[
\begin{align*}
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{na} : \{b : \text{Pre}((u'')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(b, \text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{na} : \{b : \text{Pre}((u'')(||k||_T); \text{tmp} : \text{Pre}((||k||_T); \text{Abs}^{(b, \text{tmp})})\} \\
\rightarrow &\emptyset \vdash D_{eq} : \ldots \cup \text{test}_{fail} : (||k''||_T)
\end{align*}
\]

The algorithm finishes here, giving us the result.
We use an approach by induction on the list \( L \). Let's suppose given a list of word pair \( L \).

**Lemma 30.** Let's write \( S \) such that: \[
\{\langle \langle u',v' \rangle \rangle | (u',v') \in \text{lts}(L), u \neq v \}
\]

**Proof.** First, let prove that for all structure \( k' \), there exists a set type \( S \) such that:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq} : \langle \langle L \rangle \rangle \\
\rightarrow \emptyset \vdash D_{eq} : \ldots \cup \text{test}_{na} : \{b : \text{Pre}(\langle \langle u' \rangle \rangle) \land \text{tmp} : \text{Pre}(\langle \langle k \rangle \rangle); \text{Abs}^b_{\text{tmp}}\} \\
\rightarrow \emptyset \vdash D_{eq} : \ldots \cup \text{eq}_{\text{test}} : \{a : \text{Pre}(\langle \langle v' \rangle \rangle) \land \text{tmp} : \text{Pre}(\langle \langle k \rangle \rangle); \text{Abs}^a_{\text{tmp}}\} \\
\rightarrow \emptyset \vdash D_{eq} : \ldots \cup \text{test}_{\text{fail}} : \langle \langle k' \rangle \rangle
\]

The algorithm finishes here, giving us the result.

Let's write \( S \) this computed type: we can easily see that \( \text{next}_s(D_{eq} : S) = \{S \cup \text{input}_{\text{eq}} : \langle \langle u',v',k' \rangle \rangle \} \). Using the same approach as in the case where \( u = a.u' \) and \( v = a.v' \), we then have the result.

**Program** \( D_{eq} \)

Let suppose given a list of word pair \( L \). Then, there exists a closed process type \( S \) such that:

- \( \emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq} : \langle \langle L \rangle \rangle \rightarrow \emptyset \vdash D_{eq} : S. \)
- No propagation rule can be applied on \( \emptyset \vdash D_{eq} : S. \)
- \( S(\text{eq}_{\text{fail}}) = \{\langle \langle L \rangle \rangle \} \) and \( S(\text{eq}_{\text{na}}) = \emptyset \) iff for all \( (u,v) \in \text{lts}(L), u \neq v. \)
- \( S(\text{eq}_{\text{fail}}) = \emptyset \) and \( S(\text{eq}_{\text{na}}) = \text{Abs}^\emptyset \) iff there exists a word \( u \) such that \( (u,u) \in \text{lts}(L). \)

We use an approach by induction on the list \( L \):

- Case \( L = \emptyset \). We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : \langle \langle (L,k') \rangle \rangle \\
\rightarrow \emptyset \vdash D_{eq} : \ldots \cup \text{list}_r : \{\text{tmp} : \text{Pre}(\langle \langle L \rangle \rangle); \text{Abs}\} \\
\rightarrow \emptyset \vdash D_{eq} : \ldots \cup \text{eq}_{\text{fail}} : \langle \langle k' \rangle \rangle
\]

The algorithm finishes here, giving us the result.
Case $L = (u, u) :: L'$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L, k'\|_T)
\]
\[
\rightarrow \emptyset \vdash D_{eq} : \cdots \cup \text{send}_{\text{eq}} : \left\{ \begin{array}{l}
\text{head} : \text{Pre}(\|(u, u)\|_T); \text{tail} : \text{Pre}(\|L'\|_T); \\
\text{tmp} : \text{Pre}(\|(L, k')\|_T); \text{Abs}^{\text{head, tail, tmp}} \end{array} \right\}
\]

Let’s write $S_1$ this computed type: we can easily see that $\text{next}_s(D_{eq} : S_1) = \{S_1 \cup \text{input}_{\text{eq}} : (\|(u, u, (L, k'))\|_T)\}$. Let define the structure $k_2 \triangleq (u, u, (L, k'))$. We can use the lemma 29 with this structure, which gives us a set type $S'$ such that:

- $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{\text{eq}} : (\|k_2\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$.
- No propagation rule can be applied on $\emptyset \vdash D_{eq} : S'$.
- $S'(\text{test}_{\text{fail}}) = \emptyset$ and $S'(\text{test}_{\text{ok}}) = \{\text{Abs}^0\}$.

Using the lemma 26, we have a valid propagation derivation of $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{\text{eq}} : (\|k_2\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$. We also use the lemma 25 to get a propagation derivation of $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L, k'\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$. As no propagation rule can be applied on $\emptyset \vdash D_{eq} : S'$, we can see that $\text{next}_s(D_{eq} : S') = \{S' \cup \text{eq}_k : (\{\text{Abs}^0\})\}$. Using the lemma 24, we have that $\text{next}_s(D_{eq} : S_1 \cup S') = \{S_1 \cup S' \cup \text{eq}_k : (\{\text{Abs}^0\})\}$. We then have a valid propagation derivation of

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L, k'\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S_1 \cup S' \cup \text{eq}_k : (\{\text{Abs}^0\})
\]

We finally can see that this resulting type gives us the result.

- Case $L = (u, v) :: L'$ with $u \neq v$. We have:

\[
\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L, k'\|_T)
\]
\[
\rightarrow \emptyset \vdash D_{eq} : \cdots \cup \text{send}_{\text{eq}} : \left\{ \begin{array}{l}
\text{head} : \text{Pre}(\|(u, v)\|_T); \text{tail} : \text{Pre}(\|L'\|_T); \\
\text{tmp} : \text{Pre}(\|(L, k')\|_T); \text{Abs}^{\text{head, tail, tmp}} \end{array} \right\}
\]

Let’s write $S_2$ this computed type: we can easily see that $\text{next}_s(D_{eq} : S_2) = \{S_2 \cup \text{input}_{\text{eq}} : (\|(u, v, (L, k'))\|_T)\}$. Let define the structure $k_2 \triangleq (u, v, (L, k'))$. We can use the lemma 29 with this structure, which gives us a set type $S'$ such that:

- $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{\text{eq}} : (\|k_2\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$.
- No propagation rule can be applied on $\emptyset \vdash D_{eq} : S'$.
- $S'(\text{test}_{\text{fail}}) = \{\|L, k'\|_T\}$ and $S_2(\text{test}_{\text{ok}}) = \emptyset$.

Using the lemma 26, we have a valid propagation derivation of $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{input}_{\text{eq}} : (\|k_2\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$. We also use the lemma 25 to get a propagation derivation of $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L, k'\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S'$. As no propagation rule can be applied on $\emptyset \vdash D_{eq} : S'$, we can see that $\text{next}_s(D_{eq} : S') = \{S' \cup \text{eq}_{\text{test}} : (\|L, k'\|_T)\}$. Using the lemma 24, we have that $\text{next}_s(D_{eq} : S_1 \cup S') = \{S_1 \cup S' \cup \text{eq}_k : (\|L, k'\|_T)\}$. Let now consider the structure $k_3 \triangleq (L', k')$. Using the induction hypothesis, there exists a set type $S_3$ such that:

- $\emptyset \vdash D_{eq} : \text{dct}(D_{eq}) \cup \text{eq}_{\text{test}} : (\|L', k'\|_T) \rightarrow^* \emptyset \vdash D_{eq} : S_3$. 
Theorem 19. Let suppose the given instance of PCP has a solution. Then

- No propagation rule can be applied on $\emptyset \vdash D_{\text{eqp}} : S_3$.
- $S_3(\text{eqfail}) = \{ 1 \} \land S_3(\text{equ}) = \emptyset$ iff for all $(u, v) \in \text{its}(L')$, $u \neq v$.
- $S_3(\text{eqfail}) = \emptyset$ and $S_3(\text{equ}) = \{ \text{Abs}^0 \}$ iff there exists a word $u$ such that $(u, u) \in \text{its}(L')$.

We can then apply the lemma 25 to have a valid propagation derivation of

$$\emptyset \vdash D_{\text{eqp}} : S_1 \cup S' \cup S_{\text{test}} : (\|k\|_T \rightarrow \rightarrow^* \emptyset) \vdash D_{\text{eqp}} : S_1 \cup S' \cup S_3.$$  

Per transitivity of the propagation algorithm, we have $\emptyset \vdash D_{\text{eqp}} : \text{dct}(D_{\text{eqp}}) \cup S_{\text{test}} : (\|L, k'\|_T \rightarrow \rightarrow^* \emptyset) \vdash D_{\text{eqp}} : S_1 \cup S' \cup S_3$. Using the lemma 24, we can see that $\text{next}_s(D_{\text{eqp}} : S_1 \cup S' \cup S_3) = \emptyset$. This gives us the result.

Let suppose given a word pair list $L$, and let define the set type $S_1 \triangleq \text{dct}(D_{\text{eqp}}) \cup \text{eq} : (\|L\|_T)$. We can see that $\text{next}_s(D_{\text{eqp}} : S_1) = \{ S_1 \cup \text{eqtest} : (\|L, L\|_T) \}$. Then, using the lemma 25 and 24, we have the result.

### D.5 Properties of the global program

**Theorem 19.** Let suppose the given instance of PCP has a solution. Then $D_{pcp}$ is typable.

**Proof.** As our PCP instance has a solution, there exist $m \in \mathbb{N}^*$ and a word $u$ such that $(u, u) \in L(m)$. Let first prove that for all $0 \leq i < m$, there exists a set type $S_i$ and a list $(L_k)_{0 \leq k \leq i}$ such that:

- $\emptyset \vdash D_{\text{pcp}} : S_i \rightarrow \rightarrow^* \emptyset \vdash D_{\text{pcp}} : S_i$.
- For all $0 \leq k \leq i$, its$(L_k) = L(k)$.
- $\text{next}_s(D_{\text{pcp}} : S_i) = \{ S_{i+1} \}$.
- $S_i(e) = \{ L_k \mid 0 \leq k < i \}$.

By induction on $i$:

- **Case $i = 0$.** We have the result by defining $S_0 \triangleq \text{dct}(D_{\text{pcp}})$.
- **Case $i = i' + 1$.** Using the lemma 28, there exists a set type $S_{i'}$ such that:
  - $\emptyset \vdash D_{\text{pcp}} : \text{dct}(D_{\text{eqp}}) \cup \text{eq} : (\|L\|_T) \rightarrow \rightarrow^* \emptyset \vdash D_{\text{eqp}} : S_{i'}$.
  - No propagation rule can be applied on $\emptyset \vdash D_{\text{eqp}} : S_{i'}$.
  - $S(\text{out}) = L'$ such that its$(L') = L(i' + 1)$.

Using the theorem 18, we have a valid propagation derivation of

$$\emptyset \vdash D_{\text{pcp}} : S_i \rightarrow \rightarrow^* \emptyset \vdash D_{\text{pcp}} : S_{i'} \cup S_{i'}.$$  

As no propagation rule can be applied on $\emptyset \vdash D_{\text{eqp}} : S_{i'}$, we can see that $\text{next}_s(D_{\text{pcp}} : S_{i'} \cup S_{i'}) = \{ S_{i'} \cup \text{eq} : (L') \}$. We thus have $\text{next}_s(D_{\text{pcp}} : S_{i'} \cup S_{i'}) = \{ S_{i'} \cup \text{eq} : (L') \}$.

Using the lemma 28, there exists a set type $S_{i'}$ such that:

- $\emptyset \vdash D_{\text{eqp}} : \text{dct}(D_{\text{eqp}}) \cup \text{eq} : (\|L'\|_T) \rightarrow \rightarrow^* \emptyset \vdash D_{\text{eqp}} : S_{i'}$.
- No propagation rule can be applied on $\emptyset \vdash D_{\text{eqp}} : S_{i'}$.
- $S_{i'}(\text{eqfail}) = \{ \|L'\|_T \}$ and $S_{i'}(\text{equ}) = \emptyset$.

Using the theorem 18, we have a valid propagation derivation of

$$\emptyset \vdash D_{\text{pcp}} : S_i \rightarrow \rightarrow^* \emptyset \vdash D_{\text{eqp}} : S_{i'} \cup S_{i'} \cup S_{i'}.$$  

As no propagation rule can be applied on $\emptyset \vdash D_{\text{eqp}} : S_{i'}$, we can see that $\text{next}_s(D_{\text{pcp}} : S_{i'} \cup S_{i'} \cup S_{i'}) = \{ S_{i'} \cup \text{eq} : (L') \}$. We thus have $\text{next}_s(D_{\text{pcp}} : S_{i'} \cup S_{i'} \cup S_{i'}) = \{ S_{i'} \cup \text{eq} : (L') \}$. We can then define $S_i \triangleq S_{i'} \cup S_{i'} \cup S_{i'}$ to have the result.
Let now consider the set type $S_{m-1}$. Using the lemma 28, there exists a set type $S^1_{m-1}$ such that:

- $\emptyset \vdash D_n : \text{det}(D_n) \cup \text{input} : (\|L_{m-1}\|_T) \rightarrow^* \emptyset \vdash D_n : S^1_{m-1}$.
- No propagation rule can be apply on $\emptyset \vdash D_n : S^1_{m-1}$.
- $S(\text{output}) = L'$ where $L'$ is such that $\text{lts}(L') = L(m)$.

Using the theorem 18, we have a valid propagation derivation of $\emptyset \vdash D_{pcep} : \emptyset \rightarrow^* \emptyset \vdash D_{pcep} : S^1_{m-1} \cup S^2_{m-1}$. As no propagation rule can be apply on $\emptyset \vdash D_n : S^1_{m-1}$, we can see that $\text{next}_s(D_{pcep} : S^1_{m-1}) = \{ S^1_{m-1} \cup \text{eq} : (L') \}$. We thus have $\text{next}_s(D_{pcep} : S_{m-1} \cup S^1_{m-1} \cup S^2_{m-1}) = \{ S_{m-1} \cup S^1_{m-1} \cup S^2_{m-1} \}$.

Using the lemma 28, we have a set type $S^2_{m-1}$ such that:

- $\emptyset \vdash D_{eqp} : \text{det}(D_{eqp}) \cup \text{eq} : (\|L'\|_T) \rightarrow^* \emptyset \vdash D_{eqp} : S^2_{m-1}$.
- No propagation rule can be applied on $\emptyset \vdash D_{eqp} : S^2_{m-1}$.
- $S^2_{m-1}(\text{eqfail}) = \emptyset$ and $S(\text{eq}) = \{ \text{Abs}^\emptyset \}$.

Using the theorem 18, we have a valid propagation derivation of $\emptyset \vdash D_{pcep} : \emptyset \rightarrow^* \emptyset \vdash D_{pcep} : S^2_{m-1}$.

No propagation rule can be apply on $\emptyset \vdash D_{eqp} : S^2_{m-1}$, we can see that $\text{next}_s(D_{pcep} : S^2_{m-1}) = \emptyset$. We thus have $\text{next}_s(D_{pcep} : S_{m-1} \cup S^1_{m-1} \cup S^2_{m-1}) = \emptyset$. As the propagation doesn’t fail on $\emptyset \vdash D_{pcep} : S_{m-1} \cup S^1_{m-1} \cup S^2_{m-1}$, we can conclude that this is a valid typing statement.

**Theorem 20.** Let suppose the given instance of PCP has no solution. Then $D_{pcep}$ is not typable.

**Proof.** Let suppose on the contrary that there exists $\Gamma$ and $S_1$ such that $\Gamma \vdash D_{pcep} : S_1$ holds. Using the lemma 3, there exist a valid derivation of $\emptyset \vdash D_{pcep} : S_1$. Thus, we can apply the corollary 7, there exist $S$ such that $\emptyset \vdash_m D_{pcep} : S : S$ is closed. As $(\emptyset / \text{input} : (\|L'\|_T)) \subset D_{pcep}$, the set $K \triangleq \{ (L, m) | \text{lts}(L) = L(m) \wedge \|L'\|_T \in S(\text{eq}) \}$ is not empty. Let’s then take $(L, m) \in K$, with $m$ maximal. Using the lemma 28 and 7, there exists $L'$ such that $\text{lts}(L') = L(m + 1)$ and $\|L'\|_T \in S(\text{output})$. Then, using the propagation algorithm, we have $\|L'\|_T \in S(\text{eq})$. Using the lemma 30 and 7, we have $\|L'\|_T \in S(\text{eqfail})$. Finally, we can use the propagation algorithm and the lemma 7 to have that $\|L'\|_T \in S(\text{output})$ which is impossible. Hence, the program is not typable.
E Semi-inference proofs

E.1 Preliminary results

These preliminary lemma present the computation done during one step of the semi-inference algorithm. Informally, these lemma describe trivial properties, but because of the construction of the algorithm, which are technical to prove.

Lemma 31. Let suppose given a semi-inference derivation of $\Gamma \vdash D : S \rightarrow^* \Gamma' \vdash D : S'$. Then, there exists a substitution $\sigma$ and a set type $S''$ such that:

- $\Gamma' = \sigma(\Gamma)$.
- $S' = \sigma(S) \cup S''$.
- $dc(S'') = dc(S') \setminus dc(S)$

Proof. By induction on the semi-inference derivation:

- Case $\Gamma' = \Gamma$ and $S' = S$. The result is evident in this case, with $\sigma$ as the identity substitution, and $S''$ as the empty set type.
- Case $\Gamma \vdash D : S \rightarrow \Gamma_k \vdash D : S_k \rightarrow^* \Gamma' \vdash D : S'$. Per definition of the semi-inference rule, there exists a channel $s$ such that:
  - $D : S \rightarrow s$.
  - $S, \Gamma, D \vdash s : (S_k', \sigma_k)$.

Per construction, we have $dc(S_k') = \{s\}$ and $s \not\in dc(S)$. Moreover, we have $\Gamma_k = \sigma_k(\Gamma)$, and $S_k = S_k' \cup \sigma(S)$. Let now apply the induction hypothesis: there exists $\sigma'_k$ and $S''_k$ such that

- $\Gamma' = \sigma'_k(\Gamma_k)$
- $S' = \sigma'_k(S_k) \cup S''_k$.
- $dc(S''_k) = dc(S') \setminus dc(S_k)$.

We finally define $\sigma \triangleq \sigma'_k \circ \sigma_k$ and $S'' \triangleq \sigma(S_k') \cup S''_k$, which gives the result.

Lemma 32. Let suppose given a typing environment $\Gamma$, a program $D$, a set type $S$ and a channel $s$ such that $D : S \rightarrow s$. Let note $(N, V, \varphi)$ the graph $D(D)$. Then, if we take:

- A sub-program $(e_1 : x_1 : \ldots : e_n : x_n / R) \in \varphi(s)$.
- A valid statement $S, \Gamma, D \vdash s : (S', \sigma)$.
- A tuple of types $(T_1, \ldots, T_n) \in S(e_1) \times \cdots \times S(e_n)$.

There exists $S_p$ and $S_p$ such that the type statement $\sigma(\Gamma; x_1 : T_1 ; \ldots ; x_n : T_n) \vdash_m R : S_p$ holds and $\sigma(S) \cup S' = (S_p)[s] \cup S_p$.

Proof. Let first note $\{ (e_i^j : x_i^j : \ldots : e_{n_i}^j : x_{n_i}^j / R_i) \mid 1 \leq i \leq l \}$ the set $\varphi(s)$, such that $(e_1 : x_1 : \ldots : e_n : x_n / R) = (e_1^1 : x_1^1 : \ldots : e_{n_1}^1 : x_{n_1}^1 / R_1)$

Let also note $\{(T_1^j, \ldots, T_n^j) \mid 1 \leq j \leq m \}$ the set $S(e_1) \times \cdots \times S(e_n)$, such that $(T_1, \ldots, T_n) = (T_1^1, \ldots, T_n^1)$. Using the statement $S, \Gamma, D \vdash s : (S', \sigma)$, there exist $S_1, \sigma_1, C_1$ and $F_1$ such that

\begin{align*}
&D : S \rightarrow s.
&S, \Gamma, D \vdash s : (S', \sigma).
&S, \Gamma, D \vdash s : (S', \sigma).
&D : S \rightarrow s.
&S, \Gamma, D \vdash s : (S', \sigma).
\end{align*}
Using the corollary 5, we then have \( \sigma_1(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash_m R : \text{Gen}(\sigma_1(\Gamma), \sigma_1(S_1)) \).

Then, for all \( 2 \leq j \leq m \) there exists \( S'_j \) and \( \sigma'_j \) such that

\[
\sigma_{j-1} \circ \ldots \circ \sigma_1 \circ \text{id}(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash R : (S_j, \sigma_j)
\]

We then have

\[
S, \Gamma \vdash (e_1 : x_1; \ldots; e_n : x_n/R) : \left( \bigcup_{1 \leq j \leq m-1} \sigma_m \circ \ldots \circ \sigma_{j+1}(S_j) \cup S_m, \sigma_m \circ \ldots \circ \sigma_1 \right)
\]

Let define \( \sigma'_1 \triangleq \sigma_m \circ \ldots \circ \sigma_1 \) and \( S'_1 \triangleq \bigcup_{1 \leq j \leq m-1} \sigma_m \circ \ldots \circ \sigma_{j+1}(S_j) \cup S_m \). Then for all \( 2 \leq i \leq l \) there exist \( S'_i \) and \( \sigma'_i \) such that

\[
\sigma'_{i-1} \circ \ldots \circ \sigma'_1 \circ \text{id}(S, \Gamma) \vdash (e'_1 : x'_1; \ldots; e'_n : x'_n/R') : (S'_i, \sigma'_i)
\]

We then have

\[
S, \Gamma, D \vdash s : \left( \bigcup_{1 \leq i \leq l-1} \sigma'_i \circ \ldots \circ \sigma'_{i+1}((S_i)[s]) \cup (S'_i)[s] \cup s, \sigma'_i \circ \ldots \circ \sigma'_1 \right)
\]

Let first remark that we have \( \sigma \triangleq \sigma'_i \circ \ldots \circ \sigma'_1 \) and

\[
S' \triangleq \bigcup_{1 \leq i \leq l-1} \sigma'_i \circ \ldots \circ \sigma'_{i+1}((S_i)[s]) \cup (S'_i)[s] \cup s
\]

Let note \( S_o \triangleq \sigma(\text{Gen}(\sigma_1(\Gamma), \sigma_1(S_1))) \). Then, using the lemma 8, the statement \( \sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash_m R : S_o \) holds.

Let then note

\[
S_p \triangleq \left( \bigcup_{2 \leq i \leq l-1} \sigma'_i \circ \ldots \circ \sigma'_{i+1}((S_i)[s]) \cup (S'_i)[s] \cup s \right)
\]

Per construction, we have \( S' = (S_o)[s] \cup S_p \), which gives us the result.

One can remark that in this proof, we choose carefully the indices of which sub-program we took and which types we used to type it. We choose these indices to simplify the proof, but one can see that in general, the proof still holds (but being very intricate though).

**Lemma 33.** Let suppose given a valid semi-inference derivation \( \Gamma \vdash D : S \rightsquigarrow^* \Gamma' \vdash D : S' \), and a channel \( s \in dc(D) \setminus (S) \). Then, there exists \( \Gamma_k, \Gamma'_k, S_k \) and \( S'_k \) such that:

- \( \Gamma \vdash D : S \rightsquigarrow^* \Gamma_k \vdash D : S_k \rightsquigarrow^* \Gamma'_k \vdash D : S'_k \rightsquigarrow^* \Gamma' \vdash D : S' \)
- \( dc(S'_k) \setminus dc(S_k) = \{s\} \).
Proof. By induction on the semi-inference derivation:

- Case $\Gamma \vdash D : S \rightsquigarrow \Gamma' \vdash D : S'$. By definition of the semi-inference rule, there exist a set type $S''$ and a channel $s'$ such that $\text{dc}(S'') = \{s'\}$ and $S' = S \cup S''$. Hence, $s$ equals $s'$ and we have the result.

- Case $\Gamma \vdash D : S \rightsquigarrow \Gamma'_o \vdash D : S_o \rightsquigarrow^* \Gamma' \vdash D : S'$. By definition of the semi-inference rule, there exist a set type $S''$ and a channel $s'$ such that $\text{dc}(S'') = \{s'\}$ and $S_o = S \cup S''$. In the case that $s$ equals $s'$, we have the result. In the other case, we have $s \not\in \text{dc}(S_o)$; we can then apply the induction hypothesis, which gives us the result.

E.2 Correction

The first lemma in this section presents that the semi-inference algorithm is stable under the validity definition. The correction theorem then uses this property which, combined with the typing characterization (lemma 10), implies that the resulting type is valid.

**Lemma 34.** Let suppose given a valid semi-inference derivation $\Gamma \vdash D : S \rightsquigarrow \Gamma' \vdash D : S' \rightsquigarrow^* \Gamma'' \vdash D : S''$. Then the semi-inference derivation $\Gamma' \vdash D : S' \rightsquigarrow^* \Gamma'' \vdash D : S''$ is valid.

Proof. By definition of a valid semi-inference derivation, we have $\text{dc}(D) \subset \text{dc}(S'')$.

Let note $(N, V, \varphi)$ the graph $b(D)$. By construction of the semi-inference algorithm, there exists a channel $s$ such that $\{s\} = \text{dc}(S') \setminus \text{dc}(S)$: we have $D : S \rightsquigarrow s$. Let’s take $(e_1 : x_1 ; \ldots ; e_n : x_n / R) \in \varphi(s)$, and $(T_1, \ldots, T_n) \in S''(e_1) \times \ldots \times S''(e_n)$. As $D : S \rightsquigarrow s$, we have $e_i \in \text{dc}(S)$ for all $1 \leq i \leq n$.

Using the lemma 31, there exist two substitutions $\sigma_1$ and $\sigma_2$, two process types $S_1$ and $S_2$ such that

- $S' = \sigma_1(S) \cup S_1$, $\Gamma' = \sigma_1(\Gamma)$ and $\text{dc}(S_1) = \{s\}$,
- $S'' = \sigma_2(S') \cup S_2$, $\Gamma'' = \sigma_2(\Gamma')$ and $\text{dc}(S_2) = \text{dc}(S'') \setminus \text{dc}(S')$.

As $D : S \rightsquigarrow s$, we have $e_i \in \text{dc}(S)$ for all $1 \leq i \leq n$: for all $1 \leq i \leq n$ there exists $T'_i \in \text{S}(e_i)$ such that $s \in \sigma_i(T'_i) = T_i$. Thus, using the lemma 32, there exist $S_o$ and $S_p$ such that $\sigma_1(\Gamma; x_1 : T'_1; \ldots ; x_n : T'_n) \vdash R : m S_o$, and $S_1 = (S_o)[s] \cup S_p$.

As $S_o$ is minimal, per construction of the type system, there exists a channel $e$ such that $\text{dc}(S_o) = \{e\}$. We then have three cases:

1. Case $e = s$: we have $S_1 = S_o \cup S_p$, i.e. $S'' = \sigma_2(\sigma_1(S) \cup S_o \cup S_p) \cup S_2$. Using the lemma 8, the statement $\sigma_2 \circ \sigma_1(\Gamma; x_1 : T'_1; \ldots ; x_n : T'_n) \vdash R : \sigma(S_o)$ holds. Then, using the lemma 16, there exist a type derivation of $\Gamma''; x_1 : T_1; \ldots; x_n : T_n \vdash R : S''$.

2. Case $e \in \text{dc}(S)$. Per hypothesis, the typing statement $\Gamma''; x_1 : T_1; \ldots; x_n : T_n \vdash R : S''$ holds.

3. Case $e \in \text{dc}(S') \setminus \text{dc}(S'')$. As $e \in \text{dc}(R) \subset \text{dc}(D) \subset \text{dc}(S'')$ per hypothesis, we can apply the lemma 33. There exists $\Gamma_k$, $T'_k$, $S_k$ and $S'_k$ such that:

- $\Gamma' \vdash D : S' \rightsquigarrow^* \Gamma_k \vdash D : S_k \rightsquigarrow^* \Gamma'_k \vdash D : S'_k \rightsquigarrow^* \Gamma'' \vdash D : S''$.
\[ dc(S'_k) \setminus dc(S_k) = \{ e \}. \]

As \( e \in dc(R) \), we have \((e_1 : x_1; \ldots; e_n : x_n/R) \in \varphi(e)\). Using the lemma 31, there exists three substitutions \( \sigma'_1, \sigma'_2, \sigma'_3 \) and two process types \( S'_1, S'_2, S'_3 \) such that:

- \( S'_k = \sigma'_1(S_k) \cup S'_1, \Gamma'_k = \sigma'_1(\Gamma_k) \) and \( dc(S'_1) = \{ e \} \).
- \( S'' = \sigma'_2(S'_k) \cup S'_2, \Gamma'' = \sigma'_2(\Gamma'_k) \) and \( dc(S'_2) = dc(S'_3) \setminus dc(S'_k) \).
- \( S_k = \sigma'_3(S'_3) \cup S'_3, \Gamma_k = \sigma'_3(\Gamma'') \) and \( dc(S'_3) = dc(S_k) \setminus dc(S') \).

Hence, there exists \((T'_1, \ldots, T''_n) \in S_k(e_1) \times \cdots \times S_k(e_n)\) such that \( T''_i = \sigma''_i(T'_i) \) for all \( 1 \leq k \leq n \). Using the lemma 32, there exists \( S''_o \) and \( S''_p \) such that: \( \sigma'_1(\Gamma_k; x_1 : T''_1; \ldots; x_n : T''_n) \vdash_m R : S'_o \subseteq S''_o \) and \( S'_k = (S'_o)[e] \cup S''_p \). Using the lemma 8, we also have a valid derivation of \( \sigma'_1(\Gamma_k; x_1 : T''_1; \ldots; x_n : T''_n) \vdash_m R : T'' \sigma(S_o) \). Using the lemma 17, we have \( \{ e \} = dc(S'_o) \). We can finally apply the lemma 8 and 16 to get a type derivation of \( \Gamma''; x_1 : T''_1; \ldots; x_n : T''_n \vdash R_k : S'' \).

We can then conclude by taking any element in \( \varphi(s) \), and any tuple of types, that the semi-inference \( \Gamma' \vdash D : S' \rightarrow^* F'' \), \( \Gamma'' \vdash D : S'' \) is valid.

**Theorem 21 (Correction).** Let suppose given a valid semi-inference derivation \( \Gamma \vdash D : S \rightarrow^* \Gamma' \vdash D : S' \). Then, there exists a type derivation of \( \Gamma'' \vdash D : S' \).

**Proof.** By induction on the semi-inference derivation:

- Case \( S = S' \): we have \( dc(D) \subseteq dc(S) \). Let note \((N, V, \varphi)\) the dependence graph \( D(D) \). We have \( dc(D) \subseteq dc(S') \), which implies that \( T(D) \subseteq dc(S) \). It also implies that \( \bigcup_{e \in dc(S)} \varphi(e) = \{ (e_1 : x_1; \ldots; e_n : x_n/R) \mid (e_1 : x_1; \ldots; e_n : x_n/R) \subset D \} \). We can then apply the lemma 10 on the statement \( \Gamma' \vdash D : S' \) to have the result.
- Case \( \Gamma \vdash D : S \rightarrow I_k \vdash D : S_k \). Using the lemma 32, \( \Gamma_k \vdash D : S_k \rightarrow^* \Gamma' \vdash D : S' \) is valid. We can then apply the induction hypothesis, to get a type derivation of \( \Gamma'' \vdash D : S' \).

### E.3 Completeness

The completeness theorem is introduced by several lemma proposing some new properties of the semi-inference algorithm.

**Minimal computation.** The first lemma defines that the type computation gives the most general types one can find with the given type annotation. We recall that the notation \( \hat{G}(D, S) \) means that \( S \) is a valid annotation for \( D \).

**Lemma 35.** Let suppose given a valid typing statement \( \Gamma \vdash D : S \cup S_k \) with \( \hat{G}(D, S) \cap dc(S_k) = \emptyset \). Let also suppose given a valid semi-inference step \( \Gamma \vdash D : S \rightarrow \sigma(\Gamma) \vdash D : \sigma(S) \cup S' \). Then we have \( \sigma(\Gamma) \vdash \sigma(S_k) \equiv (S')[D] \).
Proof. Using the lemma 7, there exists a type derivation of $\sigma(\Gamma) \vdash D : \sigma(S \cup S_k)$. Let now note $(N, V, \varphi)$ the graph $G(D)$, and take $(e_1 : x_1; \ldots; e_n : x_n/R) \in \varphi(s)$ and $(T_1; \ldots; T_n) \in (\sigma(S \cup S_k))(e_1) \times \cdots \times (\sigma(S \cup S_k))(e_n)$. As $D : S \rightarrow s$, for all $1 \leq i \leq n$, we have $T_i \in (\sigma(S))(e_i)$, which implies that $T_i \in (\sigma(S) \cup S')(e_i)$. Using the lemma 32, there exist two set types $S_o$ and $S_p$ such that:

- $S' = (S_o)[s] \cup S_p$.
- $\sigma(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash_m R : S_o$.

Using the lemma 10, we have $\sigma(\Gamma); x_1 : T_1; \ldots; x_n : T_n \vdash R : \sigma(S \cup S_k)$, which implies that: $\sigma(\Gamma) \vdash \sigma(S \cup S_k) \Rightarrow \sigma(S) \cup S'$. And per definition of the derivation relation on process types, we have the result.

Possible computation. The next three lemma prove that if the program is typable, and dotted with a good annotation, the semi-inference can always continue.

**Lemma 36.** Let suppose given a program $D$ and a set type $S$ such that $G(D, S)$ and $\text{dc}(D) \notin \text{dc}(S)$. Then, there exists $s$ such that $D : S \rightarrow s$.

Proof. Let write $(N, V, \varphi)$ the graph $G(D)$. Let suppose on the contrary that there is no channel $s$ such that $D : S \rightarrow s$. We can the construct an infinite list of channel $(e_i)_{1 \leq i}$ such that $e_i \in \text{dc}(D) \setminus \text{dc}(S)$ and $(e_{i+1}, e_i) \in V$ for all $1 \leq i$. As $\text{dc}(D) \notin \text{dc}(S)$, we can take $e_1 \in \text{dc}(D) \setminus \text{dc}(S)$. Now, suppose given $e_i$ such that $e_i \in \text{dc}(D) \setminus \text{dc}(S)$. As $e_i$ is not such that $D : S \rightarrow e_i$, there exists $e_{i+1} \in \text{dc}(D) \setminus \text{dc}(S)$ with $(e_{i+1}, e_i) \in V$. As $\# \text{dc}(D)$ is finite, there exist $i \neq j$ such that $e_i = e_j$. This means that there is a loop in the dependence graph with no annotation, which is impossible, as $S$ is a good annotation for $D$.

Thus, there exist $s$ with $D : S \rightarrow s$.

**Lemma 37.** Let suppose given a valid statement $\Gamma \vdash D : S \cup S_k$, with $G(D, S)$ and $\text{dc}(S) \cap \text{dc}(S_k) = \emptyset$. Let also suppose given a channel $s$ such that $D : S \rightarrow s$ and let note $(V, E, \varphi)$ the graph $G(D)$. Then for all $(e_1 : x_1; \ldots; e_n : x_n/R) \in \varphi(s)$, all $(T_1; \ldots; T_n) \in S(e_1) \times \cdots \times S(e_n)$ and all substitution $\sigma$, there exists $S'$ and $\sigma'$ such that $\sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash R : (S', \sigma)$.

Proof. Using the lemma 7, there exist a type derivation of $\sigma(\Gamma) \vdash D : \sigma(S \cup S_k)$. This implies, using the lemma 10, that $\sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash R : \sigma(S \cup S_k)$ holds. Thus, using the corollary 4, there exist (we note $F$ the set $\psi(\sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n)))$:

- A valid constraint generation statement $F, \sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash R : [F^\top] S_j \mid C$, with $\vdash C$.
- A valid constraint resolution $C \Rightarrow \sigma'$.

Hence, if we define $S' \triangleq \text{Gen}(\sigma'(\Gamma), \sigma'(S_1))$, we have the result.

**Lemma 38.** Let suppose given a valid statement $\Gamma \vdash D : S \cup S_k$, with $G(D, S)$, $\text{dc}(D) \notin \text{dc}(S)$ and $\text{dc}(S) \cap \text{dc}(S_k) = \emptyset$. Then, there exists $\Gamma'$ and $S'$ such that $\Gamma \vdash D : S \rightarrow \Gamma' \vdash D : S'$.
Proof. Let suppose on the contrary that the relation \( \rightsquigarrow \) cannot be applied on \( \Gamma \vdash D : S \). As \( S \) is a good annotation for \( D \), with the lemma 36, there exists \( s \) such that \( D : S \rightsquigarrow s \). Let note \( (N,V,\varphi) \) the graph \( D(D) \) and \( \{ (e_1^i : x_1^i;\ldots;x_n^i,R_i) \mid 1 \leq i \leq n \} \) the set \( \varphi(s) \). As \( \Gamma \vdash D : S \not\rightsquigarrow \) there exist \( 1 \leq I \leq n \) such that:

1. \( \forall 1 \leq i < I, \) there exists \( \sigma_i \) and \( S_i \) such that
   \[
   \sigma_{i-1} \circ \cdots \circ \sigma_1 \circ \text{id}(S, \Gamma) \vdash (e_1^i : x_1^i; \ldots ; e_n^i : x_n^i, /R^i) : (S_i, \sigma_i) \quad \text{holds}
   \]

2. there exists no \( \sigma_I \) nor \( S_I \) with such property.

Let note \( \{ T_{1,j}^i, \ldots, T_{m,j}^i \mid 1 \leq j \leq m \} \) the set \( S(e_1^i) \times \cdots \times S(e_n^i) \). Because of 2), there exist \( 1 \leq J \leq m \) such that:

- \( \forall 1 \leq j < J, \) there exists \( s_j \) and \( S_j \) such that
  \[
  \sigma_{j-1} \circ \cdots \circ \sigma_1 \circ \text{id}(\sigma_{j-1} \circ \cdots \circ \sigma_1 \circ \text{id}(\Gamma; x_1 : T_{1,j}^i; \ldots ; x_n : T_{m,j}^i)) \vdash R : (S_j, s_j)
  \]
- there exists no \( s_j \) nor \( S_j \) with such property.

But this second property is impossible.

Indeed, let note \( \sigma \triangleq \sigma^{j-1}_1 \circ \cdots \circ \sigma_1 \circ \text{id} \). As the statement \( \Gamma \vdash D : S \cup S_k \) holds, we can use the lemma 37: there exists \( S' \) and \( \sigma' \) such that \( \sigma(\Gamma; x_1 : T_{1,j}^i; \ldots ; x_n : T_{m,j}^i) \vdash R : (S', \sigma') \).

This contradict our hypothesis that \( \rightsquigarrow \) cannot be applied on \( \Gamma \vdash D : S' \).

We thus have the result.

**Annotation Stability.** The following lemma proves some very important properties of the \( \rightsquigarrow \) relation. Indeed, it states that a valid annotation \( S \) is extended into another valid annotation \( S' \). Moreover, if the \( S \) could be a basis of a typing \( S \cup S_k \) of the program, then its extension also can, and raise a smaller type.

This lemma is based on a small definition of channel removal to construct the process type needed by \( S' \) to type the program.

**Definition 39.** Let suppose given a set type \( S \) and a channel \( s \). We define by induction the set type \( S \setminus s \) as following:

- \( \emptyset \setminus s \triangleq \emptyset \)
- \( e \setminus s \triangleq e \) when \( e \neq s \)
- \( s \setminus s \triangleq \emptyset \)
- \( e : (T) \setminus s \triangleq e : (T) \) when \( e \neq s \)
- \( s : (T) \setminus s \triangleq \emptyset \)
- \( (S \cup S') \setminus s \triangleq (S \setminus s) \cup (S' \setminus s) \)

**Lemma 39.** Let suppose given a valid statement \( \Gamma \vdash D : S \cup S_k \), with \( G(D,S) \), \( \text{dc}(D) \not\subseteq \text{dc}(S) \) and \( \text{dc}(S) \cap \text{dc}(S_k) = \emptyset \). Let also suppose given a valid semi-inference step \( \Gamma \vdash D : S \rightsquigarrow \sigma(\Gamma) \vdash D : S' \).

Then, there exists a set type \( S_k' \) with \( \text{dc}(S') \cap \text{dc}(S_k') = \emptyset \) such that \( \Gamma' \vdash D : S' \cup S_k' \) holds. Moreover, we have \( G(D,S') \) and \( \sigma(\Gamma) \vdash \sigma(S \cup S_k) \equiv S' \cup S_k' \).
Proof. By construction of the semi-inference algorithm, there exists a channel \( s \) such that \( D : S \rightsquigarrow s \). Using the lemma 31, there exists a process type \( S_1 \) such that:

- \( S' = \sigma(S) \cup S_1 \),
- \( dc(S_1) = \{ s \} \)

As \( S' = \sigma(S) \cup S_1 \), we have \( dc(S) \subset dc(S') \), which implies that \( G(D, S') \). Let define the set type \( S'_k \triangleq \sigma(S_k) \setminus s \). Per construction, we have \( \sigma(\Gamma) \vdash \sigma(S_k) \leftarrow \sigma(S'_k) \) (we have \( \sigma(S'_k) \subset \sigma(S_k) \)). Using the lemma 35, we have \( \sigma(\Gamma) \vdash \sigma(S_k) \leftarrow S_1 \). As \( \sigma(\Gamma) \vdash \sigma(S) \leftarrow \sigma(S_k) \), we can conclude that \( \sigma(\Gamma) \vdash \sigma(S \cup S_k) \leftarrow S' \cup S'_k \).

Finally, we have \( \sigma(\Gamma) \vdash D : S' \cup S'_k \). Indeed, let note \((N, V, \varphi)\) the graph \( B(D) \), and take: \((e_1 : x_1; \ldots; e_n : x_n/R) \subset D \) and \((T_1; \ldots; T_n) \in (S' \cup S'_k)(e_1) \times \cdots \times (S' \cup S'_k)(e_n) \). As \( S \) is a good annotation for \( D \), it is not possible that \( s \in dc(R) \cap \{ e_i | 1 \leq i \leq n \} \) (\( s \) would be the only channel of a loop, and thus a defined channel of \( S \)). We thus have three cases:

1. Case \( s \notin \{ e_i | 1 \leq i \leq n \} \cup dc(R) \). Using the lemma 10, \( \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash R : S \cup S_k \) holds. Let consider the two cases where \( R = s? (M) \) and \( R = \text{Sup}(a, M, s_1, s_2) \). Per definition of the type system, in both cases there exists \( s' \in dc(R) \) and \( T \) such that:
   - \( \Gamma; x_1 : T_1; \ldots; x_n : T_n \vdash M : T \) holds.
   - \( T \in (S \cup S_k)(s') \).

Using the lemma 7, there is a typing derivation of \( \sigma(\Gamma; x_1 : T_1; \ldots; x_n : T_n) \vdash M : \sigma(T) \). As \( s \neq s' \), we have \( \sigma(T) \in S' \cup S'_k \). We can then apply the proper typing rule to have the result.

2. Case \( s \in \{ e_i | 1 \leq i \leq n \} \). Let consider \((T_1, \ldots, T_n) \in (S' \cup S'_k)(e_1) \times \cdots \times (S' \cup S'_k)(e_l) \), and note \( l \) the index such that \( e_l = s \). For all \( 1 \leq i \neq l \leq n \), we have there exists \( T\_i' \in (S \cup S_k)(e_i) \) such that \( T_i = \sigma(T\_i') \).

Using the lemma 35, there exist \( T\_i' \in (S \cup S_k)(s) \) such that \( \Gamma' \vdash \sigma(T\_i') \leftarrow T_i \) and \( f_u(T_i) \in f_u(\sigma(T\_i')) \).

Per hypothesis, as \( \Gamma \vdash D : S \cup S_k \) holds, the typing statement \( \Gamma; x_1 : T\_1; \ldots; x_n : T\_n \vdash R : S \cup S_k \) holds. Using the lemma 7, we have a type derivation of \( \Gamma' ; x_1 : \sigma(T\_1) ; \ldots ; x_n : \sigma(T\_n) \vdash R : \sigma(S \cup S_k) \). We can then apply the lemma 9 to get a type derivation of \( \Gamma'' ; x_1 : T\_1; \ldots; x_n : T\_n \vdash R : \sigma(S \cup S_k) \). As \( s \notin dc(R) \), we can use the lemma 11 to get a type derivation of \( \Gamma'' ; x_1 : T\_1; \ldots; x_n : T\_n \vdash R : \sigma(S' \cup S'_k) \). Finally, we can use the lemma 16 to get a type derivation of \( \Gamma' ; x_1 : T\_1; \ldots; x_n : T\_n \vdash R : \sigma(S' \cup S'_k) \).

3. Case \( s \in dc(R) \). Using the lemma 10, \( \Gamma; x_1 : T\_1; \ldots; x_n : T\_n \vdash R : S \cup S_k \) holds. Let consider the two cases where \( R = s? (M) \) and \( R = \text{Sup}(a, M, s_1, s_2) \). Per definition of the type system, in both cases there exists \( s' \in dc(R) \) and \( T \) such that:
   - \( \Gamma; x_1 : T\_1; \ldots; x_n : T\_n \vdash M : T \) holds.
   - \( T \in (S \cup S_k)(s') \).

If \( s' = s \), we can apply the lemma 32 to get a type derivation of \( \sigma(\Gamma; x_1 : T\_1; \ldots; x_n : T\_n) \vdash R : S_1 \). We finally apply the lemma 16 to get a type derivation of \( \sigma(\Gamma; x_1 : T\_1; \ldots; x_n : T\_n) \vdash R : S' \cup S'_k \).
Let now suppose that $s' \neq s'$. Using the lemma 7, there is a typing derivation of $\sigma(\Gamma; x_1 : T_1; \ldots, x_n : T_n) \vdash M : \sigma(T)$. As $s \neq s'$, we have $\sigma(T) \in S' \cup S_2$.

We can then apply the proper typing rule to have the result.

We can then conclude that $\Gamma' \vdash D : S' \cup S'_k$ holds.

**Theorem 22 (Completeness).** Let suppose given a valid statement $\Gamma \vdash D : S \cup S_k$, with $G(D, S)$, $dc(D) \not\subseteq dc(S)$ and $dc(S) \cap dc(S_k) = \emptyset$. Then, there exists a valid semi-inference derivation $\Gamma \vdash D : S \rightsquigarrow^* F', \sigma(\Gamma) \vdash D : S'$. Moreover, we have $\sigma(\Gamma) \vdash \sigma(S \cup S_k) \triangleleft S'$

**Proof.** We then prove the result by induction on $\#(dc(S) \setminus dc(D))$:

- Case $\#(dc(S) \setminus dc(D)) = 0$. This implies that $dc(D) \subset dc(S)$. The semi-inference derivation $\Gamma \vdash D : S$ is thus valid. Moreover, we can see, using the lemma 11, that $\Gamma \vdash D : (S)[D]$ holds. Finally, as $(S)[D] \subset S \cup S_k$, we have $\Gamma \vdash S \cup S_k \triangleleft (S)[D]$.

- Case $\#(dc(S) \setminus dc(D)) = n + 1$ with $0 \leq n$. Then, using the lemma 38 and 39, there exists $\sigma''$, $S''$ with $G(D, S'')$ and $S'_k$ with $dc(S'') \cap dc(S_k) = \emptyset$ such that $\Gamma \vdash D : S \rightsquigarrow \sigma''(\Gamma) \vdash D : S''$ and $\Gamma \vdash D : S \rightsquigarrow \sigma''(\Gamma) \vdash D : S''$ hold. Moreover, we have $\sigma''(\Gamma) \sigma(S \cup S_k) \triangleleft S'' \cup S'_k$. We can then apply the induction hypothesis with $\sigma''(\Gamma)$ and $S''$ to get a valid derivation of $\sigma''(\Gamma) \vdash D : S'' \rightsquigarrow^* \Gamma' \vdash D : S'$.

Thus, we can easily see that $F, \Gamma \vdash D : S \rightsquigarrow^* F', \Gamma' \vdash D : S'$ is valid.