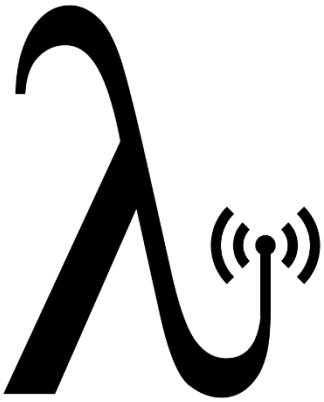

Typed Languages for Events and their Applications

Jorge Miguel Soares Ramos
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2021

Orientador
Sandra Alves, Professor Auxiliar, Faculdade de Ciência da Universidade do Porto

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

In this work, we develop typed languages, based on polymorphic record calculi, which are purposely designed to deal with events. We start by presenting EVL, a minimal higher-order functional programming language to deal with generic events. The notion of generic event extends the well-known notion of event traditionally used in a variety of areas such as database management, concurrency, reactive systems and cybersecurity. Generic events were introduced in the context of a metamodel to specify obligations in access control systems. EVL is based on an ML-style polymorphic record calculus originally developed by Ohori in the 90's, that supports polymorphic operations on records such as field selection and update, but lack support for operations such as field addition and removal. The second part if this work addresses this issue by presenting an ML-style polymorphic record calculus with extensible records based on Ohori's original calculus. Most ML-style polymorphic record calculi that support extensible records are based on row variables. We explore an alternative construction based on Ohori's original idea of using kind restrictions to express polymorphic operations on records such as field selection and modification, where other powerful operations on records such as field addition and removal can also be included. In the third and last part of this work, we show how the higher-order capabilities of EVL can be effectively used in the context of Complex Event Processing (CEP) by defining higher-order parameterised functions that deal with the usual CEP techniques, and we also show how the inclusion of other powerful operations on records such as field addition and removal can also be useful in the context of CEP.

Resumo

Neste trabalho, desenvolvemos linguagens tipadas, baseadas em cálculos de *records*, que são desenhadas com o propósito de lidar com evento. Começamos por apresentar a `EVL`, uma linguagem de programação de ordem superior mínima que lida com eventos genéricos. A noção de evento genérico estende a noção bem estabelecida de evento que é usada tradicionalmente numa variedade de áreas como gestão de bases de dados, concorrência, sistemas reactivos e cybersergurança. Eventos genéricos foram introduzidos no contexto de um metamodelo para especificar obrigações em sistemas de controlo de acesso. A `EVL` é baseada num cálculo polimórfico com *records* ao estilo ML desenvolvido originalmente por Ohori nos anos 90, que suporta operações polimórficas a *records* como a selecção e actualização de campos, mas que não suporta operações como a adição e remoção de campos. A segunda parte deste trabalho lida com este problema através da apresentação de um cálculo com *records* ao estilo ML com suporte de *records* extensíveis baseado no cálculo original do Ohori. A maioria dos cálculos polimórficos com *records* ao estilo ML que suportam *records* extensíveis são baseados em variáveis de linha. Nós exploramos uma construção alternativa baseada na ideia original do Ohori de usar restrições de espécie para expressar operações polimórficas sobre *records* como a selecção e actualização de campos, onde outras operações poderosas sobre *records* como a adição e remoção de campos também podem ser incluídas. Na terceira e última parte deste trabalho, mostramos como é que as capacidade de ordem superior da `EVL` podem ser usadas eficazmente no contexto do Processamento de Eventos Complexo (CEP) através da definição de funções de ordem superior parametrizadas que lidam com as técnicas típicas do CEP, e também mostramos como é que a inclusão de outras operações poderosas sobre *records* como a adição e remoção de campos também podem ser úteis no contexto do CEP.

Acknowledgements

There were a lot of ups and downs and backs and forwards during the execution of this work. Through it all I have had the support of my wonderful family. I would like to thank my parents Margarida and Jorge for the support they have always given me, but specially during these last couple of years. The fact that they have always trusted my choices in life is part of the reason why I was able to finish this work. The other part is due to my partner in life, Ana. We met when I was majoring in Biology and doing a minor in Informatics. She was the one that encouraged me to change majors and follow my newly found passion for Computer Science. At the time, it was very hard for me to even consider starting over and postponing the end of my studies. But she was right and I have never looked back since. If not for her, this work would have never existed. Finally, I would like to thank my supervisor Professor Sandra Alves. I will never forget our first conversation. It was my first year as an undergrad student after changing majors to Computer Science and approached her at the end of one of our Functional Programming classes and said something like "Haskell makes so much more sense to me than Java. Is this normal?". And we have been working together practically ever since. Only with time have I come to realize how lucky I am for having such an amazing tutor and (dare I say it) collaborator. If not for her, this work would not have been nearly as good as I believe it is.

To my parents Margarida and Jorge and my partner in life Ana ...

Contents

List of Figures

Chapter 1

Introduction

In [2], a general typed language to deal with the notion of event in the context of access control systems was defined, and the distinction between generic and specific events was made. Generic events represent the kind of action that can occur in a system. But specific events represent actual occurrences of those kinds of actions. As an example, the action of a doctor P_1 reading the medical record of a patient P_2 is described by the following generic event:

$$gen\_read(P_1^{\mathcal{D}}, P_2^{\mathcal{P}}) = \{act = read, doc = P_1^{\mathcal{D}}, obj = rec(P_2^{\mathcal{P}})\}.$$

And the following specific event describes the order to evacuate the neurology ward of a hospital issued by "Chief Jones":

$$\{action = evacuate, ward = neurology, principal = chief\_jones\}.$$

Typing rules for values, specification and events were given in [2], not only to ensure the well-formedness of the terms of the event language, but also to formally defined the notion of event instantiation. Associating specific events to generic events was achieved through the use an implicit notion of subtyping inspired by Ohori's type system in [46]. This allowed for type-checking of event-specification, but not for dealing with most general types for event-specifications.

Compound events in [2] were assumed to appear as a single event for simplicity reasons and a more detailed and realistic treatment of this type of events was left for future work. The notion of composition of events is a key feature of a relatively recent research area known as Complex Event Processing (CEP), where great emphasis is put on the ability to detect complex patterns in incoming streams of events and on the processing of those events.

The development of an event language that would facilitate the specification and processing of events is the main motivation behind this work.

Chapter 3 presents the `EVL` language. This language was inspired by the language in [2] and is both a restriction and an extension of Ohori's original ML-style polymorphic record calculus [46]. In [46], a record is a term of the form

$$\{l_1 = M_1, \ldots, l_n = M_n\}$$

that represents a structure with fields, each labelled with $l_1, \ldots, l_n$ and of value $M_1, \ldots, M_n$. The type system developed by Ohori in [46] supports polymorphic versions of the two most basic operations on records. Field selection $M.l$ allows us to obtain the value in a record M that has label l. Field update modify(M, l, N) allows us to update the value in a record M that has label l with the value N. As an example, consider the following term:

$$\lambda xy.\text{let } \mathit{getName} = \lambda z.(z.\mathit{name}) \text{ in } \mathit{getName}\ \{\mathit{name} = x, \mathit{address} = y\}.$$

Here, we apply the function *getName* to the concrete structure $\{\mathit{name} = x, \mathit{address} = y\}$. But it is easy to see that we should be able to apply *getName* to any other structure containing a field with label *name*. There are various ways of dealing with this type of record polymorphism. In both [46] and Chapter 3 of this work, this is achieved through the use of a system of kinds.

A kind represents a set of types and has two possible forms. The universal kind $\mathcal{U}$ represents the set of all types. A kind of the form $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$ represents the set of record types that have at least those same fields. Quantified types (or type schemes) are of the form $\forall \alpha :: \kappa.\sigma$, where α is a type variable, κ is a kind, and σ is some other type scheme. The fact that κ appears next to α means that α can only be instantiated (replaced) by types that belong to the set of types represented by κ.

In EVL, events are going to be represented as records. As an example, consider the following program written in EVL that corresponds to the example of a generic event given above:

$$\text{letEv } \mathit{GenRead}\ d\ p = \{\mathit{act} = \mathit{read}, \mathit{doc} = d, \mathit{obj} = \mathit{rec}\ p\} \text{ in } \mathit{GenRead}.$$

Because the notation used to represent records was also used in [2] to represent events, if we were to write the EVL program equivalent to the example of a specific event given above, there would be little difference. That being said, note that we did not include any type information in this last example. This is because we were able to adapt Ohori's original type inference algorithm in [46] to the type system of EVL, while keeping it sound and complete with respect to the typing rules of the language.

Regarding typing relations on events, such as generalization, specialization, and even membership, these are captured by the EVL type system in a very organic manner, through the notion of type instantiation, denoted by $\geq$. As an example, the previous EVL program describing a generic event has the type:

$$\forall \alpha_1 :: \mathcal{U}.\forall \alpha_2 :: \mathcal{U}.\alpha_1 \to \alpha_2 \to \{\mathit{act} : \tau_1, \mathit{doc} : \alpha_1, \mathit{obj} : \tau_2\},$$

where τ_1 is the type of *read* and τ_2 is the type of *rec p*, assuming *rec* has type $\alpha_2 \to \tau_2$. The following term is a specialization of that generic event:

$$\{\mathit{act} = \mathit{read}, \mathit{doc} = \mathit{john}, \mathit{obj} = \mathit{rec}\ \mathit{george}\},$$

since its type is

$$\{\mathit{act} : \tau_1, \mathit{doc} : \tau_3, \mathit{obj} : \tau_2\}$$

and

$$\forall \alpha_1 :: \mathcal{U}. \forall \alpha_2 :: \mathcal{U}. \alpha_1 \to \alpha_2 \to \{act : \tau_1, doc : \alpha_1, obj : \tau_2\} \geq \{act : \tau_1, doc : \tau_3, obj : \tau_2\}.$$

Regarding the processing of events, the higher-order capabilities of EVL as a functional programming language allows us to define parameterised functions to deal with usual CEP techniques [25]. More specifically, using EVL, we are able to process raw events produced by some event processing system and generate events that can then be passed on to an event consumer. As an example, consider the following EVL program that defines what is known as an event processing agent [26], that uses a higher-order function to filter events from a stream according to their locations:

$$\text{let } p = \lambda x.(x.\text{location}) == \texttt{"Porto"}^{String} \text{ in filter } p.$$

By following Ohori's approach in [46], we were able to define a language to deal with events. The fact that we choose Ohori's system as the basis for EVL was mainly due to the fact that the event language in [2] had already drawn some inspiration from it. But there are alternative polymorphic type systems that deal with records. And some of them can even support more powerful operations on records, such as field addition and removal. Out of these, the most common approaches are based on subtyping [14, 37] or row variables [32, 49, 52, 54], but there are others based on flags [48, 50], predicates [28, 33] and scope variables [42]. We believe that these operations are very useful in the context of event processing. This motivated us to add extensible records to Ohori's original ML-style polymorphic record calculus so we could incorporate these more powerful operations into EVL.

Chapter 4 presents an ML-style polymorphic record calculus with extensible records based on Ohori's work in [46].

In order to accommodate extensible records into Ohori's original type system, we had to refine the notions of kind and record type. In our type system, kinds still only have two possible forms. We have kept the notion of universal kind unchanged. But altered the notion of record kind. In our type system, a record kind is of the form

$$\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$$

that denotes the set of all record types that contain at least the fields to the left of $\parallel$ and do not contain the fields to the right of $\parallel$. The inclusion of negative information about the presence of fields in a record, in addition to the positive information already captured in the original definition of kind, is what allows us to add or remove fields from records and introduce two new operations on records. Terms of the form $M \setminus\setminus l$ represent the removal of a field with label l from a record M, if it exists. Terms of the form $\text{extend}(M, l, N)$ represent the addition of a field with value M and label l, if no field with that label exists in M.

We also introduce two new type constructors that we collectively call extensible types. Assuming that χ is the type of the record M and τ is the type of the term N, $\chi + \{l : \tau\}$ is the type of $M \setminus\setminus l$, if M contains the field $(l = N)$, and $\chi - \{l : \tau\}$ is the type of $\text{extend}(M, l, N)$, if M does not contain a field with label l.

By introducing these type constructors to the type system, we can no longer identify any pair of types that represent records with the same set of fields. As a first example, consider the two following records:

$$\text{extend}(\text{extend}(\{one = 1, two = 2\}, three, 3), four, 4),$$
$$\text{extend}(\text{extend}(\{one = 1, two = 2\}, four, 4), three, 3).$$

These records will have the following types:

$$\{one : Int, two : Int\} + \{three : Int\} + \{four : Int\},$$
$$\{one : Int, two : Int\} + \{four : Int\} + \{three : Int\}.$$

Even though they are syntactically different, they represent records with the same set of labels. Now consider the following example:

$$\text{extend}(\{one = 1, two = 2\}, three, 3) \setminus\setminus three.$$

Its type is the following:

$$\{one : Int, two : Int\} + \{three : Int\} - \{three : Int\}.$$

But note that it contains the same set of labels as:

$$\{one : Int, two : Int\},$$

even though their types are different.

In order to identify the type of every two records that have the same set of fields, we first introduced equality modulo ordering of what we are going to call field-alteration types. This allows us to identify the types of the first example. Then we introduced a type reduction system that will reduce types to a form where each label appears only once in every extensible type. Using this type reduction system, we are able to reduce $\{one : Int, two : Int\} + \{three : Int\} - \{three : Int\}$ to $\{one : Int, two : Int\}$ and identify these two types as well. Using the type reduction system, we are also able to develop a sound and complete type inference algorithm for this calculus with extensible records that has the additional property of always returning types in their reduced forms.

In Chapter 5 we illustrate how `EVL` can be used in the context of CEP and specification of obligation policies. And we finish the chapter by showing how extensible records can be used to represent events and how these new more powerful operations on records can be explored in the context of CEP.

1.1 List of Main Contributions

The following lists the main contribution of this work:

- The design of a minimal higher-order functional language with polymorphic record types tailored to the specific task of event processing, that we called `EVL`.
- A sound and complete type inference algorithm for `EVL`.
- A Call-by-Value operational semantics for `EVL` along with a proof of type soundness.
- The design definition of an ML-style polymorphic record calculus with extensible records with a type system based on the notion of extensible types.
- A sound and complete type inference algorithm for the ML-style polymorphic record calculus with extensible records.
- A comprehensive study of the effectiveness of `EVL` in the context of CEP and the specification of obligation policies.
- A demonstration of how `EVL` can be used in the context of CEP and specification of obligation policies.

Some of the work presented in this manuscript has already been published. Chapters 3 and 5 were based on [4] and Chapter 4 was based on [1].

Chapter 2

Background

In this chapter we present some of the well established concepts and results that are going to be necessary in order to better understand the work presented in the following chapters.

2.1 The λ-Calculus

In the 1930s [18], the λ-calculus was introduced by Alonzo Church as a foundation for logic and mathematics. Unfortunately, it was shown logically inconsistent in 1935 by Kleene and Rosser [40]. That being said, a consistent part of the theory was found quite successfully as a theory of computations [7]. Since then, the λ-calculus, along with its various typed versions, has found applications in many different areas in mathematics, philosophy, linguistics, category theory, and computer science. In the latter, the λ-calculus has played an important role in the development of the theory of programming languages and there are various functional programming languages that are based on it. For a detailed reference, we refer to [7].

Definition 2.1.1. Let x range over an infinite countable set of variables $\mathbb{V}$. The set of λ-terms, denoted by Λ, is given by the following grammar:

$$\begin{array}{rcll} M & ::= & x & \text{(variables)} \\ & | & (MM) & \text{(function application)} \\ & | & (\lambda x.M) & \text{(functional abstraction)} \end{array}$$

Whenever two terms $M, N \in \Lambda$ are ***syntactically equal***, we will write $M \equiv N$.

Example 2.1.1. Let $x, y \in \mathbb{V}$. The following are all well-formed terms from Λ:

$$(\lambda x.(xx)) \qquad (\lambda x.(\lambda y.(xy))) \qquad ((\lambda x.(xx))(\lambda y.(yy)))$$

We will follow the convention that function application is left-associative and functional

abstraction is right-associative:

$$(\dots(M_1M_2)\dots M_n) \equiv (M_1M_2\dots M_n)$$
$$(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))) \equiv (\lambda x_1x_2\dots x_n.M)$$

This allows us to remove some of the parenthesis from the terms in Example 2.1.1:

$$(\lambda x.xx) \qquad (\lambda xy.xy) \qquad (\lambda x.xx)(\lambda y.yy)$$

Notation 2.1.1. A subterm of a term M may have more that one ***occurrence*** in M. For example, the term

$$(\lambda y.xy)(\lambda z.xy)$$

has two occurrences of xy and two occurrences of x. A precise definition of "occurrence" can be written out, but a good intuitive understanding of the concept will be enough throughout this work. That being said, we will underline occurrences to distinguish them from subterms. For example, we may say

$$\text{Let } \underline{P} \text{ be any occurrence of } P \text{ in } M\text{".}$$

An occurrence of λx will be called an ***abstractor***, and the occurrence of x in it will be called a ***binding occurrence*** of x. All the occurrences of terms in M, other than binding occurrences of variables will be called ***components*** of M.

Definition 2.1.2. Let $\underline{\lambda x.P}$ be a component of a term M. Component $\underline{P}$ is called the ***body*** of $\underline{\lambda x.P}$ or the ***scope*** of the abstractor $\underline{\lambda x}$.

The ***covering abstractors*** of a component $\underline{R}$ of M are the abstractors in M whose scopes contain $\underline{R}$.

Definition 2.1.3. A non-binding variable-occurrence $\underline{x}$ in a term M is said to be ***free*** in M if, and only if, it is not in the scope of an occurrence of λx in M, otherwise it is said to be ***bound*** in M.

A variable x is said to be ***bound*** in M if, and only if, M contains an occurrence of λx; and x is said to be ***free*** in M if, and only if, M contains a free occurrence of x.

Example 2.1.2. In the term $x(\lambda y.xy)$ the variable x occurs free (twice) and y occurs bound. Note that a variable may occur both free and bound in a term: y occurs both free and bound in the term $y(\lambda y.y)$.

Definition 2.1.4. The set of ***free variables*** of $M \in \Lambda$, denoted by $FV(M)$, is defined inductively as follows:

$$FV(x) = \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$
$$FV(\lambda x.M) = FV(M) \setminus \{x\}$$

Definition 2.1.5. The set of ***bound variables*** of $M \in \Lambda$, denoted by $BV(M)$, is defined inductively as follows:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(MN) &= BV(M) \cup BV(N) \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \end{aligned}$$

Definition 2.1.6. We say that $M \in \Lambda$ is a ***closed term*** if and only if $FV(M) = \emptyset$.

In the λ-calculus, the concept of function application is captured by the β***-reduction*** rule and the definition of the latter depends on the ***substitution operator*** $M[N/x]$ which denotes the result of substituting N for x in M.

Definition 2.1.7. Let $M, N \in \Lambda$ and $x \in \mathbb{V}$. Then $M[N/x] \in \Lambda$ is obtained from M by replacing every free occurrence of x with N as follows:

$$y[N/x] \equiv \begin{cases} N & \text{if } y \equiv x \\ y & \text{otherwise} \end{cases}$$

$$(MP)[N/x] \equiv (M[N/x]P[N/x])$$

$$(\lambda y.M)[N/x] \equiv \begin{cases} (\lambda y.M) & \text{if } x \equiv y \\ (\lambda y.M[N/x]) & \text{otherwise} \end{cases}$$

Definition 2.1.8. We say that $N \in \Lambda$ is a ***subterm*** of $M \in \Lambda$, denoted by $N \sqsubseteq M$, if $N \in Sub(M)$, where $Sub(M)$, the collection of subterms of M, is defined inductively as follows:

$$\begin{aligned} Sub(x) &= \{x\} \\ Sub(\lambda x.M) &= Sub(M) \cup \{\lambda x.M\} \\ Sub(MN) &= Sub(M) \cup Sub(N) \cup \{MN\} \end{aligned}$$

Definition 2.1.9. Let $M, N \in \Lambda$.

- We say that M reduces to N in one β***-reduction*** step, denoted by $M \to_\beta^1 N$, if N can be obtained from M by replacing a subterm of M of the form $(\lambda x.P)Q$ (called a β***-redex***) with $P[Q/x]$ (its ***contractum***).
- We define $\to_\beta$ as the reflexive and transitive closure of $\to_\beta^1$.

Note that care is needed for substitutions. Consider $(\lambda x.(\lambda y.yx))y \in \Lambda$ and its one-step β-reduction

$$(\lambda x.(\lambda y.yx))y \to_\beta^1 (\lambda y.yx)[y/x] \equiv \lambda y.yy.$$

The free variable y becomes bound after substitution for x in term $\lambda y.yx$. This should not be allowed and can be avoided.

Definition 2.1.10. A ***change of bound variables*** in $M \in \Lambda$ is the replacement of a subterm $\lambda x.N \sqsubseteq M$ with $\lambda y.(N[y/x])$, where y does not occur (at all) in N.

Because we assume that y is ***fresh*** (i.e. that y does not occur (at all) in N), there is no danger of being accidentally bound in N after the substitution for x.

Definition 2.1.11. We say that $M \in \Lambda$ is α-congruent with $N \in \Lambda$, denoted by $M \equiv_\alpha N$, if N results from M by a series of changes of bound variables.

Example 2.1.3. Let $x, y, z \in \mathbb{V}$.

$$\lambda x.xy \equiv_\alpha \lambda z.zy \not\equiv_\alpha \lambda y.yy$$
$$\lambda x.x(\lambda x.x) \equiv_\alpha \lambda y.y(\lambda x.x) \equiv_\alpha \lambda y.y(\lambda z.z)$$

From now on, we will identify α-congruent terms, i.e, we will write $\lambda x.x \equiv \lambda y.y$ and so on. We will also adopt the ***Barendregt variable-name convention***: if $M_1, \ldots, M_n \in \Lambda$ occur in a certain mathematical context (e.g. a definition or a proof), we assume that all bound variables in these terms are chosen to be different from the free variables, i.e., $(BV(M_1) \cup \cdots \cup BV(M_n)) \cap (FV(M_1) \cup \cdots \cup FV(M_n)) = \emptyset$.

Consider the term $(\lambda y.yz)\ (\lambda x.x)$. This term can be β-reduced in one step to $(\lambda x.x)z$ and then further to z.

If we think of β-reduction as a computation mechanism, then each β-reduction step corresponds to a single computation step (if we ignore substitution steps) and z is the result of the computation, which we call the ***normal form*** of $(\lambda y.yz)\ (\lambda x.x)$.

Definition 2.1.12. Let $M \in \Lambda$.

- M is a β-normal form (or is *in* β-normal form) if M has no subterm of the form $(\lambda x.P)Q \sqsubseteq M$.
- We say that M has a β-normal form if there exists an $N \in \Lambda$ such that $N \equiv M$ and N is in β-normal form.

Example 2.1.4. Let $x, y, z \in \mathbb{V}$.

- $\lambda x.x$ is in β-normal form.
- z is the β-normal form of $(\lambda y.yz)\ (\lambda x.x)$.
- $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ does not have a β-normal form.

As seen in this example, not all terms have β-normal forms. On the other hand, if a term has a β-normal form, then it is unique.

Theorem 2.1.1 (Church-Rosser)**.** Let $M, N, P \in \Lambda$. If $M \to_\beta N$ and $M \to_\beta P$, then there is a $Q \in \Lambda$ such that $N \to_\beta Q$ and $N \to_\beta Q$.

Proof. See Theorem 1B5 in [36] or Theorem 3.2.8 in [7]. □

Example 2.1.5. Consider the term $(\lambda xy.y)\ \Omega$. To apply a β-reduction step to it, we can either choose to first reduce its leftmost β-redex,

$$(\lambda xy.y)\Omega \to^1_\beta \lambda y.y,$$

or its rightmost β-redex

$$(\lambda xy.y)\Omega \to^1_\beta (\lambda xy.y)\ \Omega.$$

In the former case, we reach its β-normal form. In the latter we do not, because Ω β-reduces to itself.

The order in which we apply a sequence of β-reduction to a term is meaningful and we must take care when choosing which β-redex to reduce first. In fact, there is a specific order in which we should apply β-reduction in order to always reach the β-normal form of a term that admits a normal form.

Definition 2.1.13. A sequence of β-reductions is said to be in ***normal order*** if, at each β-reduction step, we always choose to reduce its leftmost β-redex first.

Theorem 2.1.2 (Normalization)**.** Let $N \in \Lambda$ be the normal form of $M \in \Lambda$. Then there is a sequence of β-reduction steps in normal order that starts with M and ends at N.

Proof. See Theorem 1B9 in [36] or Theorem 13.2.2 in [7]. □

If we think of the λ-calculus as a programming language and think of β-reduction as its computation mechanism, we may realize the following correspondences:

- A term in β-normal form corresponds to the result of a (terminating) computation.
- Terms without a β-normal form correspond to non-terminating (infinite) computations.
- β-reducing a term in normal order corresponds to a call-by-name operational semantics.

2.2 The Simply Typed λ-Calculus

The simply typed λ-calculus is a typed interpretation of the type-free λ-calculus presented in the previous section. It was originally introduced by Alonzo Church in [19] to avoid paradoxical uses of the type-free λ-calculus and later by Haskell Curry and Robert Feys in [22] by adapting the type-assignment mechanism developed for combinatory logic in [21] to λ-terms.

Definition 2.2.1. Let α range over an infinite countable set of type variables $\mathbb{A}$. The set of types, denoted by $\mathbb{T}$, is given by the following grammar:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(type variables)} \\ & | & \tau \to \tau & \text{(function types)} \end{array}$$

We will write $\tau \equiv \tau'$ whenever two types $\tau, \tau' \in \mathbb{T}$ are ***syntactically equal***.

Example 2.2.1. Let $\alpha, \beta, \in \mathbb{A}$. The following are all well-formed types from $\mathbb{T}$:

$$(\alpha \to \alpha) \qquad (\alpha \to (\beta \to \beta)) \qquad ((\alpha \to \alpha) \to (\beta \to \beta)) \qquad ((\alpha \to \alpha) \to \beta)$$

Definition 2.2.2. The ***set of distinct type variables*** occurring in a type $\tau \in \mathbb{T}$, denoted by $\mathit{Vars}(\tau)$, is defined inductively as follows:

$$\begin{aligned} \mathit{Vars}(\alpha) &= \{\alpha\} \\ \mathit{Vars}(\tau \to \tau') &= \mathit{Vars}(\tau) \cup \mathit{Vars}(\tau') \end{aligned}$$

Definition 2.2.3. We say that $\tau \in \mathbb{T}$ is a ***subtype*** of $\tau' \in \mathbb{T}$, denoted by $\tau \sqsubseteq_{type} \tau'$, if $\tau \in \mathit{Subtype}(\tau')$, where $\mathit{Subtype}(\tau')$, the collection of subtypes of τ', is defined inductively as follows:

$$\begin{aligned} \mathit{Subtype}(\alpha) &= \{\alpha\} \\ \mathit{Subtype}(\tau \to \tau') &= \mathit{Subtype}(\tau) \cup \mathit{Subtype}(\tau') \cup \{\tau \to \tau'\} \end{aligned}$$

Let $\tau, \tau' \in \mathbb{T}$. According to [36], each type variable can be interpreted as a set, and each function type $\tau \to \tau'$ can be interpreted as a set of functions from τ to τ'. The precise nature of this set of functions (e.g. either all functions or only functions that can be defined in a given system) will depend on the particular interpretation we may have in mind.

We will follow the convention that function types are right-associative:

$$(\tau_1 \to (\tau_2 \to (\ldots \to (\tau_{n-1} \to \tau_n)))) \equiv \tau_1 \to \tau_2 \to \ldots \to \tau_{n-1} \to \tau_n$$

This will allow us to remove some of the parentheses from the types in Example 2.2.1:

$$\alpha \to \alpha \qquad \alpha \to \beta \to \beta \qquad (\alpha \to \alpha) \to \beta \to \beta \qquad (\alpha \to \alpha) \to \beta$$

Note that the associativity of function types is opposite to the one of function application: function application follows the convention in which an operator's input is written on the right; in function types, the type of the input appears on the left.

There are two main ways of introducing types in the λ-calculus, one is attributed to Alonzo Church [19] and the other to Haskell Curry [22]. In the former, the definition of λ-terms is

restricted by giving each term a unique type as part of its structure and only allowing an application MN to be defined when M has a function type of the form $\tau \to \tau'$ and N has the appropriate argument-type τ. Curry took a different approach. He proposed a language which would include all the type-free λ-terms, and a type-theory which would contain rules assigning types to some of these terms, but not others. That is, in Church style type theory, each term has a unique built-in type, while in Curry style type theory, term are assigned types according to typing rules. From a programming languages perspective, the Curry style type-theory allows us to write programs (λ-terms) without any type annotations and have a separate type system that can check if the programs we write are well-formed depending on whether they can be assigned a type or not. In this thesis, we will adopt this style.

Definition 2.2.4. Let $M \in \Lambda$ and $\tau \in \mathbb{T}$.

1. A ***type-assignment*** statement is of the form $M : \tau$, where M is the ***subject*** and τ the ***predicate*** of the statement.
2. A ***type-declaration*** is a statement with a variable as its subject.
3. A ***typing environment***, denoted by Γ, is any finite (and possibly empty) set of type-declarations $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ with distinct variables as subjects, in which case we say that the typing environment is ***consistent***.

Informally, $M : \tau$ should be read as "M can be assigned the type τ" or "M has type τ" or "M denotes a member of whatever set τ denotes".

From now on, the reader may assume that every typing environment is consistent unless it is stated otherwise.

Definition 2.2.5. Let $\Gamma = \{x : \tau_1, \ldots, x_n : \tau_n\}$ be a typing environment:

- The domain of Γ, denoted by $dom(\Gamma)$, is $\{x_1, \ldots, x_n\}$.
- $\Gamma(x_i) = \tau_i, 1 \leq i \leq n$.
- The result of removing the assignment whose subject is x (when it exists) from Γ, denoted by Γ_x, is $\Gamma \setminus \{x : \Gamma(x)\}$.
- The result of restricting the domain of Γ to a set V of variables, denoted $\Gamma|_V$, is $\{x : \Gamma(x) \mid x \in V\}$.

Notation 2.2.1. We write $\Gamma\{x : \tau\}$ for $\Gamma \cup \{x : \tau\}$ if $\Gamma \cup \{x : \tau\}$ is consistent.

Definition 2.2.6. We say that $M \in \Lambda$ can be assigned the type $\tau \in \mathbb{T}$ according to the type assignment Γ, if the statement $M : \tau$ can be derived from Γ using the typing rules in Figure 2.1.

Example 2.2.2. Consider the term $(\lambda xy.x)(\lambda x.x)$ and typing environment $\Gamma = \emptyset$. Then we can deduce $(\lambda xy.x)(\lambda x.x) : \beta \to \alpha \to \alpha$ from Γ using the typing rules as follows:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{Var})$$

$$\frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}\ (\text{App})$$

$$\frac{\Gamma\{x : \tau\} \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}\ (\text{Abs})$$

Figure 2.1: Typing rules for the Simply Typed λ-calculus.

$$\dfrac{\dfrac{\dfrac{\dfrac{(x : \alpha \to \alpha) \in \{x : \alpha \to \alpha, y : \beta\}}{\{x : \alpha \to \alpha, y : \beta\} \vdash x : \alpha \to \alpha}\ (\text{Var})}{\{x : \alpha \to \alpha\} \vdash \lambda y.x : \beta \to \alpha \to \alpha}\ (\text{Abs})}{\emptyset \vdash \lambda xy.x : (\alpha \to \alpha) \to \beta \to \alpha \to \alpha}\ (\text{Abs}) \qquad \dfrac{\dfrac{(x : \alpha) \in \{x : \alpha\}}{\{x : \alpha\} \vdash x : \alpha}\ (\text{Var})}{\emptyset \vdash \lambda x.x : \alpha \to \alpha}\ (\text{Abs})}{\emptyset \vdash (\lambda xy.x)(\lambda x.x) : \beta \to \alpha \to \alpha}\ (\text{App})$$

In other words, $(\lambda xy.x)(\lambda x.x)$ has type $\beta \to \alpha \to \alpha$.

We stated in Example 2.1.4 that Ω does not have a β-normal form. In the simply typed λ-calculus, it does not have a type as well. The reason is that in this calculus, every typeable term has a β-normal form.

Theorem 2.2.1 (Strong Normalization)**.** If M is a a typeable term in the simply typed λ-calculus, every β-reduction that starts at M is finite.

Proof. See Theorem 2D6 in [36]. □

Note that the converse does not hold in the simply typed λ-calculus. E.g., the term $\lambda x.xx$ is in β-normal form but it is not typable in this calculus because to type the subterm xx, x has to be both of type α of $\alpha \to \beta$. This self-application of x to itself is what gives rise to the paradoxes that Church wanted to avoid by introducing types to the type-free λ-calculus. But only the general concept of self-application is harmful, particular cases of self-application are still allowed.

Example 2.2.3. We can deduce $(\lambda x.x)\ (\lambda x.x) : \alpha \to \alpha$ from the empty typing environment Γ as follows:

$$\dfrac{\dfrac{\dfrac{(x : \alpha \to \alpha) \in \{x : \alpha \to \alpha\}}{\{x : \alpha \to \alpha\} \vdash x : \alpha \to \alpha}\ (\text{Var})}{\emptyset \vdash \lambda x.x : (\alpha \to \alpha) \to \alpha \to \alpha}\ (\text{Abs}) \qquad \dfrac{\dfrac{(x : \alpha) \in \{x : \alpha\}}{\{x : \alpha\} \vdash x : \alpha}\ (\text{Var})}{\emptyset \vdash \lambda x.x : \alpha \to \alpha}\ (\text{Abs})}{\emptyset \vdash (\lambda x.x)\ (\lambda x.x) : \alpha \to \alpha}\ (\text{App})$$

Besides avoiding logical paradoxes another main purpose of type-theories is to avoid the running programs that may result in runtime errors. If a term M has a type τ we can think of M as being in some sense "safe", i.e., if M represents a stage in some computation which

continues by β-reducing M, we would like to know that all later stages in the computation are just as safe as M.

Theorem 2.2.2 (Subject Reduction)**.** If $\Gamma \vdash M : \tau$ and $M \to_\beta N$, then $\Gamma \vdash N : \tau$.

Proof. See Theorem 2C1 in [36]. □

A typeable term has a potentially infinite set of types in the simply typed λ-calculus.

Example 2.2.4. Consider the term $\lambda x.x$ and typing environment $\Gamma = \emptyset$. Then we can deduce $\lambda x.x : \alpha \to \alpha$ from Γ using the typing rules as follows:

$$\dfrac{\dfrac{(x : \alpha) \in \{x : \alpha\}}{\{x : \alpha\} \vdash x : \alpha}\,(\text{Var})}{\emptyset \vdash \lambda x.x : \alpha \to \alpha}\,(\text{Abs})$$

But note that all other types in the infinite set of types that can be assigned to this term are ***instances*** of $\alpha \to \alpha$, which is a ***principal type*** for the term $\lambda x.x$.

In fact, every term of the simply typed λ-calculus has a principal type, which is the most general type it can receive. Most importantly, from a programming languages perspective, is the fact that there exists an algorithm for finding the most general type of a term.

Definition 2.2.7. A ***type-substitution***, denoted by S, is any expression of the form

$$[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n],$$

such that $\alpha_1, \ldots, \alpha_n \in \mathbb{A}$ are distinct type-variables and $\tau_1, \ldots, \tau_n \in \mathbb{T}$.

Let $S = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ be a type-substitution:

- For $\tau \in \mathbb{T}$, $S(\tau)$ is the type obtained by simultaneously substituting each α_i with τ_i throughout τ, for $1 \leq i \leq n$. We call $S(\tau)$ an ***instance*** of τ.
- The domain of S, denoted by $dom(S)$, is $\{\alpha_1, \ldots, \alpha_n\}$.

Type-substitutions can be extended to typing environments as follows.

Definition 2.2.8. Let $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ be a typing environment and S be a type-substitution:

$$S(\Gamma) = \{x_1 : S(\tau_1), \ldots, x_n : S(\tau_n)\}.$$

And to type derivations as follows.

Definition 2.2.9. Let $\Gamma \vdash M : \tau$ be the conclusion of some derivation of the statement $M : \tau$ from some typing environment Γ:

$$S(\Gamma \vdash M : \tau) = S(\Gamma) \vdash M : S(\tau).$$

Note that the consistency of Γ implies that of $S(\Gamma)$. Also, note that type-substitutions only apply to types or typing environments, not to terms. This means that terms are left unchanged whenever a type-substitution is applied to them.

We call $S(\Gamma)$ and $S(\Gamma \vdash M : \tau)$ ***instances*** of Γ and $\Gamma \vdash M : \tau$, respectively.

Definition 2.2.10. A ***variables-for-variables*** substitution is a substitution of the form $[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ where $\beta_1, \ldots, \beta_n$ are (not necessarily distinct) variables.

Let $S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$ be a variables-for-variables substitution. If $\beta_1, \ldots, \beta_n$ are distinct, S is called ***one-to-one***, and if also $\{\alpha_1, \ldots, \alpha_n\} = \mathit{Vars}(\tau)$, for a given $\tau \in \mathbb{T}$, S is called a ***renaming*** (of the variables) in τ.

Definition 2.2.11. We say that two type-substitutions S_1 and S_2 are ***extensionally equivalent***, denoted by $S_1 = S_2$, if and only if $S_1(\tau) \equiv S_2(\tau)$ for all $\tau \in \mathbb{T}$.

The composition of two type-substitutions S_1 and S_2 should be a simultaneous substitution that will have the same effect as applying S_2 and S_1 in succession.

Definition 2.2.12. The composition of two type-substitutions $S_1 = [\tau_1^1/\alpha_1^1, \ldots, \tau_n^1/\alpha_n^1]$ and $S_2 = [\tau_1^2/\alpha_1^2, \ldots, \tau_m^2/\alpha_m^2]$, denoted by $S_2 \circ S_1$, is defined as:

$$S_2 \circ S_1 = [\tau_{i_1}^2/\alpha_{i_1}^2, \ldots, \tau_{i_k}^2/\alpha_{i_k}^2, S_2(\tau_1^1)/\alpha_1^1, \ldots, S_2(\tau_n^1)/\alpha_n^1],$$

where $\{\alpha_{i_1}^2, \ldots, \alpha_{i_k}^2\} = \mathit{dom}(S_2) \setminus \mathit{dom}(S_1)$.

Also, we assume that this operation is right-associative so that

$$S_1 \circ (S_2 \circ \cdots \circ (S_{n-1} \circ S_n)) = S_1 \circ S_2 \circ \cdots S_{n-1} \circ S_n.$$

Now we can define precisely what it means for a type to be a principal type of a term.

Definition 2.2.13. In the simply typed λ-calculus, a ***principal type*** of a term M is a type $\tau \in \mathbb{T}$ such that:

- $\Gamma \vdash M : \tau$, for some typing environment Γ.
- If $\Gamma' \vdash M : \tau'$ for some typing environment Γ', then τ' is an instance of τ.

The principal type of a term can be thought of as completely characterizing the set of all types assignable to that term.

Definition 2.2.14. A ***principal pair*** for a term M is a pair (Γ, τ) such that $\Gamma \vdash M : \tau$ is deducible using the typing rules and every other deducible formula $\Gamma' \mid_{dom(\Gamma)} \vdash M : \tau'$ is an instance of $\Gamma \vdash M : \tau$.

Definition 2.2.15. A ***principal deduction*** for a term M is a deduction Δ of a formula $\Gamma \vdash M : \tau$ such that every other deduction whose conclusion's subject is M is an instance of Δ.

Theorem 2.2.3 (Principal Type)**.** Every typeable term has a principal-deduction and a principal-type in the simply typed λ-calculus. Further, there is an algorithm that will decide whether a given λ-term M is typeable in the simply typed λ-calculus and if "it is" will output a principal pair for M.

Proof. See Theorem 3A6 in [36]. □

The two main steps in the ***type inference algorithm*** will be the computation of the principal type of a term of the form $\lambda x.M$ from the principal type of the subterm M and the computation of the principal type of a term of the form MN from the principal type of M and N. The former is straightforward, while the latter requires a little work.

Example 2.2.5. Suppose we are trying to infer whether an application MN such that $FV(MN) = \emptyset$ (for simplicity reasons) is typeable in the simply typed λ-calculus, and we already know that the principal type of M is $\tau \to \tau'$ and the principal type of N is τ''. Then by Definition 2.2.13, we know that the set of types that can be assigned to M are instances of $\tau \to \tau'$ and that the set of types that can be assigned to N are instances of τ''. Therefore, there are type-substitutions S_1 and S_2 such that $S_1(\tau) \equiv S_2(\tau'')$ and we can apply the (App) rule to deduce a type for MN as follows:

$$\frac{\dfrac{\vdots}{\emptyset \vdash M : S_1(\tau) \to S_1(\tau')} \qquad \dfrac{\vdots}{\emptyset \vdash N : S_2(\tau'')}}{\emptyset \vdash MN : S_1(\tau')} \text{(App)}.$$

Thus the problem of deciding whether MN is typable reduces to that of finding type-substitutions S_1 and S_2 such that $S_1(\tau) \equiv S_2(\tau'')$ is the ***most general common instance*** of these types.

Definition 2.2.16. A ***unifier*** of $\tau \in \mathbb{T}$ and $\tau' \in \mathbb{T}$ is a type substitution S such that $S(\tau) \equiv S(\tau')$.

If two types have a unifier, then we say that they are ***unifiable*** and the instance that is common to both types is their ***common instance***.

Example 2.2.6. The following type-substitution

$$S = [\beta_1 \to \beta_1/\alpha_1, \alpha_2 \to \beta_1 \to \beta_1/\beta_2]$$

is a unifier of the types

$$\alpha_1 \to \alpha_2 \to \alpha_1 \qquad (\beta_1 \to \beta_1) \to \beta_2$$

and their common instance is

$$(\beta_1 \to \beta_1) \to \alpha_2 \to \beta_1 \to \beta_1.$$

Definition 2.2.17. The ***most general common instance*** of a pair of types (τ_1, τ_2) is a common instance τ_3 such that every other common instance is an instance of τ_3.

Note that unifiers are not necessarily unique.

Definition 2.2.18. A type substitution S_1 is the ***most general unifier*** of two types $\tau \in \mathbb{T}$ and $\tau' \in \mathbb{T}$ if for every other unifier S_2 of these two types, there is a type substitution S_3 such that $S_2 = S_3 \circ S_1$.

Theorem 2.2.4 (Unification)**.** There is an algorithm that decides whether a pair of types (τ, τ') has a unifier and if "they do" constructs its most general unifier. Also, if a pair of type (τ, τ') has a unifier, then it has a most general unifier.

Proof. See Theorem 3D4 in [36]. □

With the addition of the following lemma we can speak of *the* most general unifier as if most general unifiers were unique (see Notation 3D2.4 in [36]).

Lemma 2.2.5 (Renaming of Most General Unifiers)**.** If a type-substitution S_1 is a most general unifier of a pair of types (τ, τ') and S_2 is a renaming of the variables in $S_1(\tau)$, then $S_2 \circ S_1$ is a most general unifier of (τ, τ').

Proof. See Lemma 3D2.3 in [36]. □

Note that most general unifiers do not (in general) give rise to ***most general common instances***.

Example 2.2.7. The most general unifier of the pair of types

$$(\alpha_1 \to \alpha_2 \to \alpha_2, (\alpha_3 \to \alpha_3) \to \alpha_1)$$

is $[(\alpha_2 \to \alpha_2)/\alpha_1, \alpha_2/\alpha_3]$ and the common instance that is obtained from it is

$$(\alpha_2 \to \alpha_2) \to \alpha_2 \to \alpha_2.$$

In contrast, their most general common instance is

$$(\alpha_3 \to \alpha_3) \to \alpha_2 \to \alpha_2.$$

But most general unifiers and most general common instances can be reconciled through the following lemma.

$$\mathcal{U}(\alpha,\tau) = \begin{cases} [\tau/\alpha] & \text{if } \alpha \notin \mathit{Vars}(\tau) \\ \mathit{id} & \text{if } \tau = \tau \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\mathcal{U}(\tau \to \tau', \alpha) = \mathcal{U}(\alpha, \tau \to \tau')$$

$$\mathcal{U}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') = \begin{array}{l} \text{let } S = \mathcal{U}(\tau_1', \tau_2') \\ \text{in } \mathcal{U}(S(\tau_1), S(\tau_2)) \circ S \end{array}$$

We assume that the unification function fails if any of the recursive calls fails.

Figure 2.2: The unification algorithm for the Simply Typed λ-calculus.

Lemma 2.2.6 (Most General Unifier - Most General Common Instance)**.** If $\tau \in \mathbb{T}$ and $\tau' \in \mathbb{T}$ have no common variables ($\mathit{Vars}(\tau) \cap \mathit{Vars}(\tau') = \emptyset$), (τ, τ') has a most general unifier if and only if it has a most general common instance, and the two are identical.

Proof. See Lemma 3D3 in [36]. □

Now, the problem of finding the most general common instances is reduced to that of finding most general unifiers of pairs of types with no variables in common using Robinson's first-order unification algorithm [51].

Definition 2.2.19. Let $\tau, \tau' \in \mathbb{T}$. The ***unification function*** $\mathcal{U}(\tau, \tau')$ that returns the most general unifier of (τ, τ') is defined inductively in Figure 2.2. $\alpha \notin \mathit{Vars}(\tau)$ is usually called an "occur check".

Example 2.2.8.

$$\begin{array}{lrcl} 1) & \mathcal{U}(\alpha_1 \to \alpha_1, \beta_1 \to \beta_2) & = & [\beta_1/\beta_2, \beta_1/\alpha_1] \\ 1.1) & \mathcal{U}(\alpha_1, \beta_2) & = & [\beta_2/\alpha_1] \\ 1.2) & \mathcal{U}([\beta_2/\alpha_1](\alpha_1), [\beta_2/\alpha_1](\beta_1)) & = & [\beta_1/\beta_2] \end{array}$$

The most general unifier of the pair of types $(\alpha_1 \to \alpha_1, \beta_1 \to \beta_2)$ is $[\beta_1/\beta_2, \beta_1/\alpha_1]$ and their most general common instance is $\beta_1 \to \beta_1$.

As was stated in Theorem 2.2.4, there is an algorithm that will decide whether a given λ-term M is typeable in the simply typed λ-calculus and if it is will output a principal deduction and principal type for M.

Definition 2.2.20. Let $M \in \Lambda$. The ***type inference function*** $\mathit{HM}(M)$ that returns the principal pair (Γ, τ) for M is defined inductively in Figure 2.3.

$$
\begin{array}{rcl}
HM(x) & = & (\{x:\alpha\},\alpha)\ (\alpha \text{ fresh}) \\
\\
HM(M_1M_2) & = & \text{let } \{x_1,\ldots,x_n\} \in FV(M_1)\cap FV(M_2) \\
 & & \text{in let } (\Gamma_1,\tau_1) = HM(M_1) \\
 & & \qquad (\Gamma_2,\tau_2) = HM(M_2) \\
 & & \qquad S_1 = \mathcal{U}(\Gamma_1(x_1),\Gamma_2(x_1)) \\
 & & \qquad S_2 = \mathcal{U}(S_1(\Gamma_1)(x_1), S_1(\Gamma_2)(x_1)) \\
 & & \qquad S_3 = \mathcal{U}(S_2\circ S_1(\Gamma_1)(x_1), S_2\circ S_1(\Gamma_2)(x_1)) \\
 & & \qquad\qquad \vdots \\
 & & \qquad S_n = \mathcal{U}(S_{n-1}\circ\cdots\circ S_1(\Gamma_1)(x_n), S_{n-1}\circ\cdots\circ S_1(\Gamma_2)(x_n)) \\
 & & \qquad S_0 = \mathcal{U}(S_n\circ\cdots S_1(\tau_1), S_n\circ\cdots\circ S_1(\tau_2\to\alpha))\ (\alpha \text{ fresh}) \\
 & & \quad \text{in } (S_n\circ\cdots\circ S_1\circ S_0(\Gamma_1\cup\Gamma_2), S_0(\alpha)) \\
\\
HM(\lambda x.M) & = & \text{let } (\Gamma',\tau_1) = HM(M) \\
 & & \text{in if } x\in dom(\Gamma') \\
 & & \quad \text{then } (\Gamma'_x, \Gamma'(x)\to\tau_1) \\
 & & \quad \text{else } (\Gamma', \alpha\to\tau_1)\ (\alpha \text{ fresh})
\end{array}
$$

Figure 2.3: The type inference algorithm *HM* for the Simply Typed λ-Calculus.

Example 2.2.9.

1)	$HM((\lambda xy.x)(\lambda x.x))$	$=$	$(\emptyset, \alpha_2\to\alpha_3\to\alpha_3)$
1.1)	$HM(\lambda xy.x)$	$=$	$(\emptyset, \alpha_1\to\alpha_2\to\alpha_1)$
1.1.1)	$HM(\lambda y.x)$	$=$	$(\{x:\alpha_1\}, \alpha_2\to\alpha_1)$
1.1.1.1)	$HM(x)$	$=$	$(\{x:\alpha_1\}, \alpha_1)$
1.2)	$HM(\lambda x.x)$	$=$	$(\emptyset, \alpha_3\to\alpha_3)$
1.2.1)	$HM(x)$	$=$	$(\{x:\alpha_3\}, \alpha_3)$
1.3)	$\mathcal{U}(\alpha_1\to\alpha_2\to\alpha_1, (\alpha_3\to\alpha_3)\to\alpha_4)$	$=$	$[\alpha_2\to\alpha_1/\alpha_4]\circ[\alpha_3\to\alpha_3/\alpha_1]$
1.3.1)	$\mathcal{U}(\alpha_2\to\alpha_1, \alpha_4)$	$=$	$[\alpha_2\to\alpha_1/\alpha_4]$
1.3.1.1)	$\mathcal{U}(\alpha_4, \alpha_2\to\alpha_1)$	$=$	$[\alpha_2\to\alpha_1/\alpha_4]$
1.3.2)	$\mathcal{U}([\alpha_2\to\alpha_1/\alpha_4](\alpha_1), [\alpha_2\to\alpha_1/\alpha_4](\alpha_3\to\alpha_3))$	$=$	$[\alpha_3\to\alpha_3/\alpha_1]$

The principal pair of $(\lambda xy.x)(\lambda x.x)$ is $(\emptyset, \alpha_2\to\alpha_3\to\alpha_3)$.

2.3 λ-Calculus with Parametric Polymorphism

From a programming languages point-of-view, there are some terms that are not typeable in the simply typed λ-calculus that we would like to write.

Example 2.3.1. Consider the λ-term that corresponds to the identity function $\lambda x.x$.

If we apply this function to some other $M\in\Lambda$ and β-reduce the application, we get M back

no matter the M:

$$(\lambda x.x)\ M \to_\beta x[M/x] \equiv M.$$

The semantics of functions like the identity function is independent (in a type-free semantics) of the types of their arguments. Such objects are ***type-polymorphic*** in the sense that they belong to (or possess) more than one type. Additionally, since many of the primitives of a programming language are naturally polymorphic, then many of the functions one can define using those primitives will also be type-polymorphic.

Example 2.3.2.

1)	$HM((\lambda y.yy)(\lambda x.x))$	$=$	fail
1.1)	$HM(\lambda y.yy)$	$=$	$(\emptyset, \alpha_2 \to \alpha_2)$
1.1.1)	$HM(yy)$	$=$	$(\{y : \alpha_2\}, \alpha_2)$
1.1.1.1)	$HM(y)$	$=$	$(\{y : \alpha_1\}, \alpha_1)$
1.1.1.2)	$HM(y)$	$=$	$(\{y : \alpha_2\}, \alpha_2)$
1.1.1.3)	$\mathcal{U}(\alpha_1, \alpha_2)$	$=$	$[\alpha_2/\alpha_1]$
1.1.1.4)	$\mathcal{U}(\alpha_2, \alpha_2 \to \alpha_3)$	$=$	fail
1.2)	$HM(\lambda x.x)$	$=$	$(\emptyset, \alpha_4 \to \alpha_4)$
1.2.1)	$HM(x)$	$=$	$(\{x : \alpha_4\}, \alpha_4)$

$(\lambda y.yy)(\lambda x.x)$ is not typeable in the simply typed λ-calculus because y is required to have more than one type.

In [45], Robin Milner introduced a theory of type-polymorphism for programming languages that was originally design for the ML programming language. We will present the simple extension of the λ-calculus in [23] obtained by the introduction of a declarative construct of the form

$$\text{let } x = M \text{ in } N$$

denoting the result of evaluating N with x denoting the value of M.

This construct is semantically equivalent (in the operational sense) to the β-redex $(\lambda x.N)\ M$, but in ML's type-system let $x = M$ in N might have a type even when $(\lambda x.N)\ M$ does not. The reason is that x is used polymorphically in the former, but not it the latter.

Definition 2.3.1. Let x range over an infinite countable set of variables $\mathbb{V}$. The of λ-terms, denoted by Λ_{let}, is given by the following grammar:

M	$::=$	x	(variables)
	$\mid$	(MM)	(function application)
	$\mid$	$\lambda x.M$	(functional abstraction)
	$\mid$	let $x = M$ in M	(let declaration)

Many concepts from the (simply typed) λ-calculus can be extended to this calculus by simply treating an expression of the form let $x = M$ in N as if it was the β-redex $(\lambda x.N)\ M$.

For this reason, the reader may assume that the concepts introduced in the two previous sections are also valid for this calculus, except when stated otherwise.

Instead of introducing a separate reduction rule for let-expressions that will act on let-expressions just like β-reduction, we will treat let-expressions as β-redexes and reduce them using β-reduction as well.

Definition 2.3.2. Let α range over an infinite countable set of type variables $\mathbb{V}$. The set of types, denoted by $\mathbb{T}_{let}$, is given by the following grammar:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(type variables)} \\ & | & \tau \to \tau & \text{(function types)} \\ \\ \sigma & ::= & \tau & \text{(monomorphic types)} \\ & | & \forall\alpha.\sigma & \text{(polymorphic types)} \end{array}$$

Polymorphic types are ***type-schemes*** that represent the set of all possible types that can be assigned to a term.

We will abbreviate a type-scheme $\forall\alpha_1 \cdots \forall\alpha_n.\tau$ to $\forall\alpha_1 \cdots \alpha_n.\tau$ and call $\alpha_1, \ldots, \alpha_n$ the ***generic variables*** of that type-scheme.

Definition 2.3.3. We say that a type-variable ***occurs free*** in a type-scheme $\forall\alpha_1 \ldots \forall\alpha_n.\tau$ if and only if it occurs in τ and is not one of the generic variables $\alpha_1, \ldots, \alpha_n$.

Definition 2.3.4. The set of ***free type-variables*** of $\sigma \in \mathbb{T}_{let}$, denoted by $FTV(\sigma)$, is defined inductively as follows:

$$\begin{aligned} FTV(\alpha) &= \{\alpha\} \\ FTV(\tau \to \tau') &= FTV(\tau) \cup FTV(\tau') \\ FTV(\forall\alpha.\sigma) &= FTV(\sigma) \setminus \{\alpha\} \end{aligned}$$

We say that a type $\sigma \in \mathbb{T}$ is closed if $FTV(\sigma) = \emptyset$.

We must extend the subtype-relation to type-schemes.

Definition 2.3.5. We say that $\sigma \in \mathbb{T}_{let}$ is a ***subtype*** of $\sigma' \in \mathbb{T}_{let}$, denoted by $\sigma \sqsubseteq_{type} \sigma'$, if $\sigma \in \mathit{Subtype}(\sigma')$, where $\mathit{Subtype}(\sigma')$, the collection of subtypes of σ', is defined inductively as follows:

$$\begin{aligned} \mathit{Subtype}(\alpha) &= \{\alpha\} \\ \mathit{Subtype}(\tau \to \tau') &= \mathit{Subtype}(\tau) \cup \mathit{Subtype}(\tau') \cup \{\tau \to \tau'\} \\ \mathit{Subtype}(\forall\alpha.\sigma) &= \mathit{Subtype}(\sigma) \cup \{\forall\alpha.\sigma\} \end{aligned}$$

Definition 2.3.6. We say that a typing environment Γ is ***closed*** if $\forall x \in dom(\Gamma)$, $\Gamma(x)$ is closed.

$$\frac{(x:\sigma)\in\Gamma}{\Gamma\vdash_{let} x:\sigma}\ (\text{Var})$$

$$\frac{\Gamma\vdash_{let} M:\sigma \qquad \alpha\notin FTV(\Gamma)}{\Gamma\vdash_{let} M:\forall\alpha.\sigma}\ (\text{Gen})$$

$$\frac{\Gamma\vdash_{let} M:\forall\alpha.\sigma}{\Gamma\vdash_{let} M:\sigma[\tau/\alpha]}\ (\text{Inst})$$

$$\frac{\Gamma\vdash_{let} M:\tau\to\tau' \qquad \Gamma\vdash_{let} N:\tau}{\Gamma\vdash_{let} (MN):\tau'}\ (\text{App})$$

$$\frac{\Gamma\{x:\tau\}\vdash_{let} M:\tau'}{\Gamma\vdash_{let} \lambda x.M:\tau\to\tau'}\ (\text{Abs})$$

$$\frac{\Gamma\vdash_{let} M:\sigma \qquad \Gamma\cup\{x:\sigma\}\vdash_{let} N:\tau}{\Gamma\vdash_{let} \text{let } x=M \text{ in } N:\tau}\ (\text{Let})$$

Figure 2.4: Typing rules for the λ-Calculus with Parametric Polymorphism.

Given a type-scheme σ we will let a substitution S act on σ by acting on the free variables of σ while renaming (if necessary) the generic variables of σ to avoid clashes with variables involved in S.

Definition 2.3.7. An ***assumption scheme*** is a type-declaration with a type scheme as its predicate.

Definition 2.3.8. We say that a type $\tau\in\mathbb{T}_{let}$ is a ***generic instance*** of a type-scheme $\sigma\in\mathbb{T}_{let}$ if and only if $\sigma=\tau$ or $\sigma=\forall\alpha_1,\ldots,\alpha_n.\tau'$ and there exists a type-substitution S such that $\tau=S(\tau')$.

Definition 2.3.9. Let V be a given set of type-variables and $\tau\in\mathbb{T}_{let}$:

- The ***closure*** of τ under V, denoted by $\overline{V}(\tau)$, is the type-scheme $\forall\alpha_1\cdots\alpha_n.\tau$ where $\alpha_1,\ldots,\alpha_n$ are all the type-variables occurring in τ that are not in V.
- For any typing environment Γ, $\overline{\Gamma}(\tau)$ denotes the closure of τ under (the set of type-variables that occur in) Γ.

Definition 2.3.10. We say that $M\in\Lambda_{let}$ can be assigned the type $\tau\in\mathbb{T}_{let}$ according to the type assignment Γ, if the statement $M:\tau$ can be derived from Γ using the typing rules in Figure 2.4.

Definition 2.3.11. A ***principal type-scheme*** of a term $M\in\Lambda_{let}$ with respect to a typing environment Γ is a type-scheme $\sigma\in\mathbb{T}_{let}$ such that:

- $\Gamma\vdash_{let} M:\sigma$.

$$
\begin{aligned}
W(\Gamma, x) &= \text{if } (x : \forall\alpha_1, \ldots, \alpha_n.\tau) \in \Gamma \ (1 \leq i \leq n) \\
&\quad \text{then } (id, [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau)) \ (\beta_1, \ldots, \beta_n \text{ fresh}) \\
&\quad \text{else fail} \\
\\
W(\Gamma, MN) &= \text{let } (S_1, \tau_1) = W(\Gamma, M) \\
&\quad\quad (S_2, \tau_2) = W(S_1(\Gamma), N) \\
&\quad\quad S_0 = \mathcal{U}(S_2(\tau_1), \tau_2 \to \beta) \ (\beta \text{ fresh}) \\
&\quad \text{in } (S_0 \circ S_2 \circ S_1, S_0(\beta)) \\
\\
W(\Gamma, \lambda x.M) &= \text{let } (S, \tau) = W(\Gamma_x \cup \{x : \beta\}, M) \ (\beta \text{ fresh}) \\
&\quad \text{in } (S, S(\beta) \to \tau) \\
\\
W(\Gamma, \text{let } x = M \text{ in } N) &= \text{let } (S_1, \tau_1) = W(\Gamma, M) \\
&\quad\quad (S_2, \tau_2) = W(S_1(\Gamma_x) \cup \{x : \overline{S_1(\Gamma)}(\tau_1)\}, N) \\
&\quad \text{in } (S_2 \circ S_1, \tau_2)
\end{aligned}
$$

Figure 2.5: The type inference algorithm W for the λ-Calculus with Parametric Polymorphism.

- If $\Gamma \vdash_{let} M : \sigma'$ for some type-scheme $\sigma \in \mathbb{T}_{let}$, then σ' is a generic instance of σ.

In contrast with the type inference algorithm for the simply typed λ-calculus, which could be used to find the principal type of a term, it is not possible for a type-scheme inference algorithm to do the same. We can derive type-schemes for the term $((\lambda xy.x)\ (fz))(fw)$ if we assume, e.g., either $f : \forall\alpha.\alpha \to \alpha$ or $f : \forall\alpha.\alpha \to \beta$. It is however easy to see that there is not a more general assumption scheme about f except for the trivial $f : \forall\alpha.\alpha$, from which a type for $((\lambda xy.x)\ (fz))(fw)$ can be derived using the typing rules, and which includes as generic instances all generic instance of both $\forall\alpha.\alpha \to \alpha$ and $\forall\alpha.\alpha \to \beta$.

Also in contrast with what we have seen for the simply typed λ-calculus, if a type scheme can be inferred for a term from a particular typing environment then it admits a principal type-scheme under those assumptions. For this reason it is natural that a type-scheme inference algorithm should take as arguments not only a λ-term but also a typing environment.

Definition 2.3.12. Let Γ be a type assignment, $M \in \Lambda_{let}$ and $\tau \in \mathbb{T}_{let}$. The ***type-scheme inference function*** $W(\Gamma, M)$ that returns a pair (S, τ) such that $S(\Gamma) \vdash_{let} M : \tau$ is defined inductively in Figure 2.5.

Theorem 2.3.1 (Soundness of W)**.** If $W(\Gamma, M)$ succeeds with (S, τ) then there is a derivation $S(\Gamma) \vdash_{let} M : \tau$.

Proof. See Chapter II, Theorem 2 in [23]. □

Theorem 2.3.2 (Completeness of W)**.** Given a typing environment Γ and a term $M \in \Lambda_{let}$, let Γ' be an instance of Γ and $\sigma \in \mathbb{T}_{let}$ be a type-scheme such that $\Gamma' \vdash_{let} M : \sigma$:

- $W(\Gamma, M)$ succeeds.
- If $W(\Gamma, M) = (S, \tau)$, then, for some substitution S', $\Gamma' = S' \circ S(\Gamma)$ and σ is a generic instance of $S' \circ \overline{S(\Gamma)}(\tau)$.

Proof. See Chapter II, Theorem 3 in [23]. □

Corollary 2.3.2.1. If $W(\Gamma, M) = (S, \tau)$, then $\overline{S(\Gamma)}(\tau)$ is the principal type-scheme of M under Γ.

Proof. See Chapter II, Corollary 1 in [23]. □

Example 2.3.3.

$$
\begin{array}{lrcl}
1) & W(\emptyset, \text{let } y = \lambda x.x \text{ in } yy) & = & ([\alpha_4/\alpha_2, \alpha_3 \to \alpha_3/\alpha_4], \alpha_3 \to \alpha_3) \\
1.1) & W(\emptyset, \lambda x.x) & = & (id, \alpha_1 \to \alpha_1) \\
1.1.1) & W(\{x : \alpha_1\}, x) & = & (id, \alpha_1) \\
1.2) & W(\{y : \forall\alpha_1.\alpha_1 \to \alpha_1\}, yy) & = & ([\alpha_4/\alpha_2, \alpha_3 \to \alpha_3/\alpha_4], \alpha_3 \to \alpha_3) \\
1.2.1) & W(\{y : \forall\alpha_1.\alpha_1 \to \alpha_1\}, y) & = & (id, \alpha_2 \to \alpha_2) \\
1.2.2) & W(\{y : \forall\alpha_1.\alpha_1 \to \alpha_1\}, y) & = & (id, \alpha_3 \to \alpha_3) \\
1.2.3) & [\alpha_4/\alpha_2, \alpha_3 \to \alpha_3/\alpha_4] & = & \mathcal{U}(\alpha_2 \to \alpha_2, (\alpha_3 \to \alpha_3) \to \alpha_4)
\end{array}
$$

let $y = \lambda x.x$ in $yy \in \Lambda_{let}$ is semantically equal to $(\lambda y.yy)\ (\lambda x.x) \in \Lambda$ from Example 2.3.2, but it is typeable in this calculus, while the latter is not.

From Corollary 2.3.2.1, we get that $\forall\alpha_3.\alpha_3 \to \alpha_3$ is the principal type-scheme of let $y = \lambda x.x$ in yy with respect to the empty typing-environment.

2.4 λ-Calculus with Polymorphic Records

Labeled records are widely used data structures and are essential building blocks in various data-intensive applications such as database programming. The ML programming language contains labeled records, but their allowable operations are restricted to monomorphic ones. In [46], Atsushi Ohori introduced a let-polymorphic record calculus that extends the previous calculus with labeled records and polymorphic operations over those records.

Example 2.4.1. Consider the following simple function on records:

$$(\lambda x.x).A$$

where $x.A$ corresponds to selecting the A field from the record x. This function is polymorphic in the sense that it can be applied to terms of any record type containing a A field, such as $\{A : \alpha_1, B : \alpha_2\}$ or $\{A : \alpha_1, C : \alpha_3\}$.

Record-polymorphism is based on the idea that labeled-field access is polymorphic and can therefore be applied to any labeled data structure containing the specified field, just like we have seen in the previous example. This form of polymorphism can be captured by defining a ***subtyping*** relation and allowing a value to have all its "supertypes". However, in the presence of subtyping, a static type (i.e. a type that is know at compile time) no longer represents the exact record structure of a runtime value.

Example 2.4.2. Let $x : \alpha_1$, $y : \alpha_2$ and $z : \alpha_3$, the following term

$$\text{if } \mathit{true} \text{ then } \{A = x, B = y\} \text{ else } \{B = y, C = z\}$$

has a type $\{B : \alpha_2\}$, but its runtime value will presumably be $\{A = x, B = y\}$.

An alternative approach is to extend ML-style polymorphic typing directly to record-polymorphism. In these type-systems, a most general polymorphic type-scheme can be inferred for any typeable untyped term containing operations on records. By an appropriate instantiation of the inferred type-scheme, an untyped term can safely be used as a value of various different types. This approach not only captures the polymorphic nature of functions on records but also integrates record-polymorphism and ML-style type-scheme inference [46].

Most of the proposed type inference systems have been based on the mechanism of ***row variables***, which are variables ranging over finite sets of field types. In [46], instead of row variables, restrictions are placed on possible instantiations of type variables using a kind-system of types that refines ordinary ***type-quantification*** to ***kinded-quantification*** of the form $\forall\alpha :: \kappa.\sigma$ where variable α is constrained to range only over the set of types denoted by a ***kind*** κ. This mechanism is analogous to bounded quantification [16].

Definition 2.4.1. Let x range over an infinite countable set of variables $\mathbb{V}$ and l range over an infinite countable set of labels $\mathbb{L}$. The set of λ-terms, denoted by $\Lambda_{\mathcal{O}}$, is given by the following grammar:

$$\begin{array}{rcll} M & ::= & c^b & \text{(constants)} \\ & | & x & \text{(variables)} \\ & | & (MM) & \text{(function application)} \\ & | & \lambda x.M & \text{(functional abstraction)} \\ & | & \text{let } x = M \text{ in } M & \text{(let declaration)} \\ & | & \{l = M, \ldots, l = M\} & \text{(records)} \\ & | & M.l & \text{(field selection)} \\ & | & \text{modify}(M, l, M) & \text{(field update)} \end{array}$$

where b is a base type from a set of base types $\mathbb{B}$.

Example 2.4.3. The following are all well-formed terms from $\Lambda_{\mathcal{O}}$:

$$\lambda xy.\text{let } \mathit{name} = \lambda z.(z.\mathit{Name}) \text{ in } \mathit{name}\ \{\mathit{Name} = x, \mathit{Address} = y\}$$

$$\lambda xyz.\text{let } \mathit{update} = \lambda xy.\text{modify}(x, \mathit{Address}, y) \text{ in } (\mathit{update}\ \{\mathit{Name} = x, \mathit{Address} = y\})\ z$$

Definition 2.4.2. Let α range over an infinite countable set of type variables $\mathbb{A}$, l range over an infinite countable set of labels $\mathbb{L}$ and b range over a set of base types $\mathbb{B}$. The set of types, denoted by $\mathbb{T}_{\mathcal{O}}$, and the set of kinds, denoted by $\mathbb{K}_{\mathcal{O}}$, are given by the following grammar:

$$\begin{array}{rcll} \tau & ::= & b & \text{(base types)} \\ & | & \alpha & \text{(type variables)} \\ & | & \tau \to \tau & \text{(function types)} \\ & | & \{l : \tau, \ldots, l : \tau\} & \text{(record types)} \\ \\ \sigma & ::= & \tau & \text{(monomorphic types)} \\ & | & \forall \alpha :: \kappa.\sigma & \text{(polymorphic types)} \\ \\ \kappa & ::= & \mathcal{U} & \text{(universal kind)} \\ & | & \{\{l : \tau, \ldots, l : \tau\}\} & \text{(record kinds)} \end{array}$$

The universal kind represents the set of all types and a record kind of the form $\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$ represents the set of structures that have at least fields $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$, respectively. We assume that the labels that appear in any type or kind are always pairwise distinct.

This calculus is an extension of the calculus presented in Section 2.3 and many of the concepts that are valid for that calculus can be extended for this calculus. For this reason, the reader may assume that the concepts introduced in the previous sections are also valid for this calculus, except when stated otherwise.

We need to extend the set of free type variables from Definition 2.3.4 to cover records and kinds.

Definition 2.4.3. The set of ***free type-variables*** of a type $\sigma \in \mathbb{T}_{\mathcal{O}}$, denoted by $FTV(\sigma)$, is defined inductively as follows:

$$\begin{aligned} FTV(\alpha) &= \{\alpha\} \\ FTV(\tau \to \tau') &= FTV(\tau) \cup FTV(\tau') \\ FTV(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) &= FTV(\tau_1) \cup \cdots \cup FTV(\tau_n) \\ FTV(\forall \alpha :: \kappa.\sigma) &= FTV(\kappa) \cup (FTV(\sigma) \setminus \{\alpha\}) \end{aligned}$$

The set of ***free type-variables*** of a kind $\kappa \in \mathbb{K}$, denoted $FTV(\kappa)$, is defined inductively as follows:

$$\begin{aligned} FTV(\mathcal{U}) &= \emptyset \\ FTV(\{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}) &= FTV(\tau_1) \cup \cdots \cup FTV(\tau_n) \end{aligned}$$

Note that the type construct $\forall \alpha :: \kappa.\sigma$ binds the type variable α in σ, but not in κ.

Example 2.4.4. Let $l \in \mathbb{L}$.

$$FTV(\forall \alpha_1 :: \{\{l : \alpha_2 \to \alpha_2\}\}.\alpha_1 \to \alpha_3) = \{\alpha_2, \alpha_3\}$$

We need to extend the subtype relation to record types.

Definition 2.4.4. We say that $\sigma \in \mathbb{T}_{\mathcal{O}}$ is a ***subtype*** of $\sigma' \in \mathbb{T}_{\mathcal{O}}$, denoted by $\sigma \sqsubseteq_{type} \sigma'$, if $\sigma \in \mathit{Subtype}(\sigma')$, where $\mathit{Subtype}(\sigma')$, the collection of subtypes of σ', is defined inductively as follows:

$$\begin{aligned}
\mathit{Subtype}(\alpha) &= \{\alpha\} \\
\mathit{Subtype}(\tau \to \tau') &= \mathit{Subtype}(\tau) \cup \mathit{Subtype}(\tau') \cup \{\tau \to \tau'\} \\
\mathit{Subtype}(\forall\alpha.\sigma) &= \mathit{Subtype}(\sigma) \cup \{\forall\alpha.\sigma\} \\
\mathit{Subtype}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) &= \mathit{Subtype}(\tau_1) \cup \ldots \cup \mathit{Subtype}(\tau_n) \cup \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}
\end{aligned}$$

Definition 2.4.5. Let $\alpha \in \mathbb{A}$ and $\kappa \in \mathbb{K}$. A ***kind-declaration*** is a statement of the form $\alpha :: \kappa$, where α is the subject and κ the predicate of the statement.

Definition 2.4.6. A ***kinding environment***, denoted by K, is any finite (possibly empty) set of kind-declarations $\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ with distinct type-variables as subjects.

Definition 2.4.7. Let $K = \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ be a kinding environment:

- The domain of K, denoted $\mathit{dom}(K)$, is $\{\alpha_1, \ldots, \alpha_n\}$.
- $K(\alpha_i) = \kappa_i, 1 \leq i \leq n$.
- The result of removing the kind-declaration whose subject is α (when it exists) from K, denoted K_α, is $K \setminus \{\alpha :: K(\alpha)\}$.

Any type variables that appear in a kind assignment K must also be properly kinded by K itself.

Definition 2.4.8. We say that a kinding environment K is ***well-formed*** if for all $\alpha \in \mathit{dom}(K)$, $\mathit{FTV}(K(\alpha)) \subseteq \mathit{dom}(K)$.

From now on, every kinding environment can be assumed to be well-formed unless stated otherwise.

Notation 2.4.1. We write $K\{\alpha :: \kappa\}$ for $K \cup \{\alpha :: \kappa\}$ if K is well-formed, $\alpha \notin \mathit{dom}(K)$, and $\mathit{FTV}(\kappa) \subseteq \mathit{dom}(K)$.

Definition 2.4.9. A type $\tau \in \mathbb{T}_{\mathcal{O}}$ is ***well-formed*** under a kinding environment K if $\mathit{FTV}(\sigma) \subseteq \mathit{dom}(K)$.

The previous definition is naturally extended to other syntactic constructs containing types, except for type-substitutions whose well-formedness condition is defined separately.

Definition 2.4.10. A type $\tau \in \mathbb{T}_{\mathcal{O}}$ has a kind $\kappa \in \mathbb{K}$ under a kinding assignment K, denoted by $K \Vdash_{\mathcal{O}} \tau :: \kappa$, if it is derivable by the following set of ***kinding rules***:

$$K \Vdash_{\mathcal{O}} \tau :: \mathcal{U} \text{ if } \tau \text{ is well-formed under } K$$
$$K \Vdash_{\mathcal{O}} \alpha :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \text{ if } K(\alpha) = \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$$
$$K \Vdash_{\mathcal{O}} \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} :: \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\}$$
$$\text{if } \{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots\} \text{ is well-formed under } K$$

Note that, if $K \Vdash_{\mathcal{O}} \tau :: \kappa$ for some kind assignment K, $\sigma \in \mathbb{T}_{\mathcal{O}}$ and $\kappa \in \mathbb{K}$, then τ and κ are well-formed under K.

Example 2.4.5. Let $l_1, l_2 \in \mathbb{L}$:

$$\{\alpha_1 :: \{\{l_1 : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\} \Vdash_{\mathcal{O}} \alpha_1 \to \alpha_2 :: \mathcal{U}$$
$$\{\alpha_1 :: \mathcal{U}\} \nVdash_{\mathcal{O}} \{l_1 : \alpha_1, l_2 : \alpha_2\} :: \{\{l_1 : \alpha_1, l_2 : \alpha_2\}\}$$

Since in this type discipline type-variables are ***kinded*** by a kinding assignment, our previous notion of substitution need to be refined by incorporating kind constraints.

Definition 2.4.11. A type-substitutions S is well-formed under a kinding environment K if for all $\alpha \in dom(S)$, $S(\alpha)$ is well-formed under K.

Definition 2.4.12. A ***kinded substitution*** is a pair (K, S) of a kinding environment K and a substitution S that is well-formed under K.

The kinding environment K in (K, S) specifies kind constraints of the result of the substitution.

A kinded substitution (K, S) is ***ground*** if $K = \emptyset$. We will write S for a ground kind substitution $(\emptyset, S)$.

Definition 2.4.13. A kinded substitution (K_1, S) ***respects*** a kinding environment K_2 if for all $\alpha \in dom(K_2)$, $K_1 \Vdash_{\mathcal{O}} S(\alpha) :: S(K_2(\alpha))$.

This notion specifies the condition under which a substitution can be applied, i.e. if a kinded substitution (K_1, S) respects K then it can be applied to a type σ kinded by K, yielding a type $S(\sigma)$ kinded by K_1.

Example 2.4.6. Let $l_1, l_2 \in \mathbb{L}$, $K = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}$ and $S = [\{l_1 : \alpha_1, l_2 : \alpha_2\}/\alpha_3]$. (K, S) respects $K' = \{\alpha :: \{\{l_1 :: \alpha_1, l_2 : \alpha_2\}\}, \alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}$.

Lemma 2.4.1. If $K \Vdash_{\mathcal{O}} \tau :: \kappa$ and a kinded substitution (K_1, S) respects K, then $K_1 \Vdash_{\mathcal{O}} S(\tau) :: S(\kappa)$.

Proof. See Lemma 2.1.1 in [46]. □

Corollary 2.4.1.1. If (K_1, S_1) respects K and (K_2, S_2) respects K_1, then $(K_2, S_2 \circ S_1)$ respects K.

A proof for this corollary was not provided by Ohori in [46] so we provide one here.

Proof. Since (K_1, S_1) respects K, we know that $\forall \alpha \in dom(K), K_1 \Vdash_{\mathcal{O}} S_1(\alpha) :: S_1(K(\alpha))$. Since (K_2, S_2) respects K_1, we also know that $\forall \alpha \in dom(K_1), K_2 \Vdash_{\mathcal{O}} S_2(\alpha) :: S_2(K_1(\alpha))$. But, since we know that (K_2, S_2) respects K_1, we have that $K_2 \Vdash_{\mathcal{O}} S_2 \circ S_1(\alpha) :: S_2 \circ S_1(K(\alpha))$. Therefore, $\forall \alpha \in dom(K), K_2 \Vdash_{\mathcal{O}} S_2 \circ S_1(\alpha) :: S_2 \circ S_1(K(\alpha))$ and $(K_2, S_2 \circ S_1)$ respects K. □

A kinding environment is regarded as a constraint on possible substitutions of type-variables, i.e., to those that respect it. Definition 2.4.8 allows a cyclic kinding environment like $\{\alpha_1 :: \{\{l_1 : \alpha_2\}\}, \alpha_2 :: \{\{l_2 : \alpha_1\}\}$, which is (in some sense) useless since there is no ground substitution that respects it. A stronger well-formedness condition for kinding environments that would not allow this to happen was not adopted in [46] for performance reasons. This approach however does not change the set of derivable ***closed*** typings and still yields a sound type system that detects all the type errors of a program [46].

Since in this type-system types may depend on type-variables other than their own free type-variables, we need to extend the notion of free type-variables of a type.

Definition 2.4.14. Let $\sigma \in \mathbb{T}_{\mathcal{O}}$ be well-formed under a kinding environment K. The set of ***essentially free type-variables*** of σ under K, denoted by $EFTV(K, \sigma)$, is the smallest set satisfying:

- $FTV(\sigma) \subseteq EFTV(K, \sigma)$.
- If $\alpha \in EFTV(K, \sigma)$, then $FTV(K(\alpha)) \subseteq EFTV(K, \sigma)$.

Intuitively, $\alpha \in EFTV(K, \sigma)$ if $\sigma \in \mathbb{T}_{\mathcal{O}}$ contains α either directly or through kind constrains specified by K.

Example 2.4.7. The type-variable α_1 is essentially free in α_2 under $\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}$.

This notion naturally extends to other syntactic structures containing types.

We need to extend the notion of a generic instance of a type from Definition 2.3.8 by incorporating kinds.

Definition 2.4.15. Let $\sigma_1 \in \mathbb{T}_{\mathcal{O}}$ be a polymorphic type well-formed under a kind assignment K. We say that $\sigma_2 \in \mathbb{T}_{\mathcal{O}}$ is a ***generic instance*** of σ_1 under K, denoted by $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma_2$, if $\sigma_1 = \forall \alpha_1 :: \kappa_1^1 \cdots \forall \alpha_n^1 :: \kappa_n^1 . \tau_1$, $\sigma_2 = \forall \beta_1 :: \kappa_1^2 \cdots \forall \beta_m :: \kappa_m^2 . \tau_2$ and there is a type substitution S such that $dom(S) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\beta_1 :: \kappa_1^2, \ldots, \beta_n :: \kappa_m^2\}, S)$ respects $K\{\alpha_1 :: \kappa_1^1, \ldots, \alpha_n :: \kappa_n^1\}$, and $\tau_2 = S(\tau_1)$.

Example 2.4.8. The type $\alpha_1 \to \{l_1 : \alpha_1, l_2 : \alpha_2\} \in \mathbb{T}_{\mathcal{O}}$ is a generic instance of $\forall\beta_1 :: \mathcal{U}.\forall\beta_2 :: \{\{l_1 : \alpha_1, l_2 : \alpha_2\}\}.\beta_1 \to \beta_2$ under $K = \{\alpha_1 : \mathcal{U}, \alpha_2 : \mathcal{U}\}$.

Lemma 2.4.2. If $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma_2$ and $K \Vdash_{\mathcal{O}} \sigma_2 \geq \sigma_3$ then $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma_3$.

A proof for this lemma was not provided by Ohori in [46] so we provide one here.

Proof. Without loss of generality, let us assume that $\sigma_1 = \forall\alpha_1^1 :: \kappa_1^1 \cdots \forall\alpha_n^1 :: \kappa_n^1.\tau_1$, $\sigma_2 = \forall\alpha_1^2 :: \kappa_1^2 \cdots \forall\alpha_m^2 :: \kappa_m^2.\tau_2$, and $\sigma_3 = \forall\alpha_1^3 :: \kappa_1^3 \cdots \forall\alpha_k^3 :: \kappa_k^3.\tau_3$. Since $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma_2$, we know that there exists a substitution S_1 such that $dom(S_1) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\alpha_1^2 :: \kappa_1^2, \ldots, \alpha_m^2 :: \kappa_m^2\}, S_1)$ respects $K\{\alpha_1^1 :: \kappa_1^1, \ldots, \alpha_n^1 :: \kappa_n^1\}$ and $\tau_2 = S_1(\tau_1)$. Since $K \Vdash_{\mathcal{O}} \sigma_2 \geq \sigma_3$, we know that there exists a substitution S_2 such that $dom(S_2) = \{\alpha_1^2, \ldots, \alpha_m^2\}$, $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_2)$ respects $K\{\alpha_1^2 :: \kappa_1^2, \ldots, \alpha_m^2 :: \kappa_m^2\}$ and $\tau_3 = S_2(\tau_2)$. To show that $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma_3$, we just need to find a substitution S_3, such that $dom(S_3) = \{\alpha_1^1, \ldots, \alpha_n^1\}$, $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_3)$ respects $K\{\alpha_1^1 :: \kappa_1^1, \ldots, \alpha_n^1 :: \kappa_n^1\}$, and $\tau_3 = S_3(\tau_1)$. If we choose $S_3 = S_2 \circ S_1$, then $dom(S_3) = dom(S_1) = \{\alpha_1^1, \ldots, \alpha_n^1\}$. By Corollary 2.4.1.1, we have that $(K\{\alpha_1^3 :: \kappa_1^3, \ldots, \alpha_k^3 :: \kappa_k^3\}, S_2 \circ S_1)$ respects K and $\tau_3 = S_3(\tau_1) = S_2 \circ S_1(\tau_1) = S_2(S_1(\tau_1)) = S_2(\tau_2)$. □

We also need to extend the notion of the closure of a type from Definition 2.3.9 by incorporating kinds.

Definition 2.4.16. The ***closure*** of $\tau \in \mathbb{T}_{\mathcal{O}}$ under a typing environment Γ and a kinding environment K, denoted by $Cls(K, \Gamma, \tau)$, is a pair $(K', \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau)$ such that $K'\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\} = K$ and $\{\alpha_1, \ldots, \alpha_n\} = EFTV(K, \tau) \setminus EFTV(K, \Gamma)$.

Example 2.4.9.

$$Cls(K, \Gamma, \alpha_3 \to \alpha_4) = (\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \forall\alpha_3 :: \mathcal{U}.\forall\alpha_4 :: \{\{l_1 : \alpha_1, l_2 :: \alpha_2\}\}.\alpha_3 \to \alpha_4).$$

Definition 2.4.17. We say that $M \in \Lambda_{\mathcal{O}}$ can be assigned the $\tau \in \mathbb{T}_{\mathcal{O}}$ according to the kinding assignment K and the typing assignment Γ, if the statement $M : \tau$ can be derived form K, Γ using the typing rules in Figure 2.6.

Note that if $K, \Gamma \vdash_{\mathcal{O}} M : \tau$ and $Cls(K, \Gamma, \tau) = (K', \sigma)$ then Γ and σ are well-formed under K'.

Example 2.4.10. Let $\Gamma = \{x : \alpha, y : \beta\}$ and $K = \{\alpha :: \mathcal{U}, \beta :: \mathcal{U}\}$. We can deduce $\{l_1 = x, l_2 = y\}.l_1 : \alpha$ from K, Γ as follows:

$$\dfrac{\dfrac{\dfrac{K \Vdash_{\mathcal{O}} \alpha \geq \alpha}{K, \Gamma \vdash_{\mathcal{O}} x : \alpha}\,(\text{Var}) \quad \dfrac{K \Vdash_{\mathcal{O}} \beta \geq \beta}{K, \Gamma \vdash_{\mathcal{O}} y : \beta}\,(\text{Var})}{K, \Gamma \vdash_{\mathcal{O}} \{l_1 = x, l_2 = y\} : \{l_1 : \alpha, l_2 : \beta\}}\,(\text{Rec}) \quad K \Vdash_{\mathcal{O}} \{l_1 : \alpha, l_2 : \beta\} :: \{\{l_1 :: \alpha\}\}}{K, \Gamma \vdash_{\mathcal{O}} \{l_1 = x, l_2 = y\}.l_1 : \alpha}\,(\text{Sel})$$

The following lemma shows that typings are closed under kind-respecting kinded substitutions.

$$\frac{\Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_{\mathcal{O}} c^b : b} \text{(Const)}$$

$$\frac{K \Vdash_{\mathcal{O}} \Gamma(x) \geq \tau \qquad \Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_{\mathcal{O}} x : \tau} \text{(Var)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M : \tau \qquad \mathit{Cls}(K, \Gamma, \tau) = (K', \sigma)}{K', \Gamma \vdash_{\mathcal{O}} M : \sigma} \text{(Gen)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M : \tau \to \tau' \qquad K, \Gamma \vdash_{\mathcal{O}} N : \tau}{K, \Gamma \vdash_{\mathcal{O}} MN : \tau'} \text{(App)}$$

$$\frac{K, \Gamma\{x : \tau\} \vdash_{\mathcal{O}} M : \tau'}{K, \Gamma \vdash_{\mathcal{O}} \lambda x.M : \tau \to \tau'} \text{(Abs)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M : \sigma \qquad K, \Gamma\{x : \sigma\} \vdash_{\mathcal{O}} N : \tau}{K, \Gamma \vdash_{\mathcal{O}} \text{let } x = M \text{ in } N : \tau} \text{(Let)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M_i : \tau_i \ (1 \leq i \leq n)}{K, \Gamma \vdash_{\mathcal{O}} \{l_1 : M_1, \ldots, l_n : M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{(Rec)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M : \tau \qquad K \Vdash_{\mathcal{O}} \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash_{\mathcal{O}} M.l : \tau'} \text{(Sel)}$$

$$\frac{K, \Gamma \vdash_{\mathcal{O}} M : \tau \qquad K, \Gamma \vdash_{\mathcal{O}} N : \tau' \qquad K \Vdash_{\mathcal{O}} \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash_{\mathcal{O}} \text{modify}(M, l, N) : \tau} \text{(Modif)}$$

Figure 2.6: Typing rules for the λ-Calculus with Polymorphic Records.

Lemma 2.4.3. If $K_1, \Gamma \vdash_{\mathcal{O}} M : \tau$ and (K_2, S) respects K_1, then $K_2, S(\Gamma) \vdash_{\mathcal{O}} M : S(\tau)$.

Proof. See Lemma 2.2.3 in [46]. □

In this type-system, polymorphic generalization and let abstraction are separated into two rules. It is possible to combine the two into a single rule, but it makes it harder to prove various properties that can be easily proved by induction on typing derivations.

The following lemma allows us to strengthen the type assignment.

Lemma 2.4.4. If $K, \Gamma\{x : \sigma_1\} \vdash_{\mathcal{O}} M : \tau$ and $K \Vdash_{\mathcal{O}} \sigma_2 \geq \sigma_1$, then $K, \Gamma\{x : \sigma_2\} \vdash_{\mathcal{O}} M : \tau$.

Proof. See Lemma 3.1.2 in [46]. □

To establish a stronger property of type soundness than the subject reduction property, a call-by-value operational semantics using evaluation contexts was introduced in [46] and its type soundness theorem (with respect to that semantics) was proved.

In order to develop a type inference algorithm for this type system, Robinson's unification algorithm must be refined in order to incorporate kind constraints on type-variables.

Definition 2.4.18. A ***kinded set of equations*** is a pair (K, E) consisting of a kind assignment K and a set E of pairs of types such that E is well-formed under K.

Definition 2.4.19. We say that a type-substitution S ***satisfies*** a set of pairs of types E if $S(\tau_1) = S(\tau_2)$ for all $(\tau_1, \tau_2) \in E$.

We need to extend the notion of ***unifier*** from Definition 2.2.16 and of ***most general unifier*** from Definition 2.2.18 to incorporate kinds.

Definition 2.4.20. A kinded substitution (K_1, S_1) is a ***unifier*** of a kinded set of equations (K_2, E) if it respects K_2 and if S_1 satisfies E.

Definition 2.4.21. A kinded substitution (K_1, S_1) is the ***most general unifier*** of a kinded set of equations (K_2, E) if it is a unifier of (K_2, E) and if for any unifier (K_3, S_2) of (K_2, E) there is some type substitution S_3 such that (K_3, S_3) respects K_1 and $S_2 = S_3 \circ S_1$.

We will present the ***kinded unification algorithm*** in the same style as it was presented in [46], i.e., by transformation. Each rule transforms a 4-tuple of the form (E, K, S, SK) consisting of a set E of type equations, a kinding environment K, a type-substitution S, and a (not necessarily well-formed) kinding environment SK. The roles of these components are the following:

- E keeps the set of equations to be unified;
- K specifies kind constraints to be verified;
- S records "solved" equations as a form of substitution;
- SK record "solved" kind constraints that have already been verified for S.

When specifying rules, we treat K, SK and S as sets of pairs. We also use the following notations.

Notation 2.4.2. Let F range over functions from a finite set of labels to types.

- We write $\{F\}$ and $\{\{F\}\}$ to denote the record type identified by F and the record kind identified by F, respectively.
- For two functions F_1 and F_2 we write $F_1 \pm F_2$ for the function F such that $dom(F) = dom(F_1) \cup dom(F_2)$ and such that for $l \in dom(F)$, $F(l) = F_1(l)$ if $l \in dom(F_1)$; otherwise $F(l) = F_2(l)$.

Definition 2.4.22. Let (K, E) be a given kinded set of equations. The ***kinded unification function*** $\mathcal{U}(E, K)$ that takes a any kinded set of equations, computes a most general unifier if one exists, and reports failure otherwise is defined by the transformation rules in Figure 2.7.

$$
\begin{array}{rcl}
(E \cup \{(\tau,\tau)\}, K, S, SK) & \Rightarrow_{\mathcal{U}} & (E, K, S, SK) \\
\\
(E \cup \{(\alpha,\tau)\}, K \cup \{(\alpha,\mathcal{U})\}, S, SK) & \Rightarrow_{\mathcal{U}} & ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha]S \cup \{(\alpha,\tau)\}, \\
& & [\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\}) \\
& & \text{if } \alpha \in FTV(\tau) \\
\\
(E \cup \{(\alpha_1,\alpha_2)\}, & & \\
K \cup \{(\alpha_1,\{\{F_1\}\}),(\alpha_2,\{\{F_2\}\})\}, S, SK) & \Rightarrow_{\mathcal{U}} & ([\alpha_2/\alpha_1](E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1) \cap dom(F_2)\}), \\
& & [\alpha_2/\alpha_1](K) \cup \{(\alpha_2,[\alpha_2/\alpha_1](\{\{F_1 \pm F_2\}\}))\}, \\
& & [\alpha_2/\alpha_1](S) \cup \{(\alpha_1,\alpha_2)\}, [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1,\{\{F_1\}\})\}) \\
\\
(E \cup \{(\alpha,\{F_2\})\}, K \cup \{(\alpha,\{\{F_1\}\})\}, S) & \Rightarrow_{\mathcal{U}} & ([\{F_2\}/\alpha](E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1)\}), \\
& & [\{F_2\}/\alpha](K), [\{F_2\}/\alpha](S) \cup \{(\alpha,\{F_2\})\}), \\
& & [\{F_2\}(SK) \cup \{(\alpha_1,\{\{F_1\}\})\}) \\
& & \text{if } dom(F_1) \subseteq dom(F_2) \text{ and } \alpha \notin FTV(\{F_2\}) \\
\\
E \cup \{(\{F_1\},\{F_2\})\}, K, S, SK) & \Rightarrow_{\mathcal{U}} & (E \cup \{(F_1(l),F_2(l)) \mid l \in dom(F_1)\}, K, S, SK) \\
& & \text{if } dom(F_1) = dom(F_2) \\
\\
(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2\}, K, S, SK) & \Rightarrow_{\mathcal{U}} & (E \cup \{(\tau_1^1,\tau_2^1),(\tau_1^2,\tau_2^2)\}, K, S, SK)
\end{array}
$$

For a notation of the form $X \cup Y$ on the left hand side of each rule, we assume that X and Y are disjoint.

Figure 2.7: The unification algorithm for the λ-Calculus with Polymorphic Records.

Example 2.4.11.

$$
\begin{array}{l}
(\{(\alpha_1,\alpha_2)\}, \{\alpha_1 :: \{\{l_1 : \alpha_3\}\}, \alpha_2 :: \{\{l_1 : \alpha_4, l_2 : \alpha_3\}\}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \emptyset, \emptyset) \Rightarrow_{\mathcal{U}} \\
(\{(\alpha_3,\alpha_4)\}, \{\alpha_2 :: \{\{l_1 : \alpha_4, l_2 : \alpha_3\}\}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \{(\alpha_1,\alpha_2)\}, \{(\alpha_1,\{\{l_1 : \alpha_3\}\})\}) \Rightarrow_{\mathcal{U}} \\
(\emptyset, \{\alpha_2 :: \{\{l_1 : \alpha_4, l_2 : \alpha_4\}\}, \alpha_4 :: \mathcal{U}\}, \{(\alpha_3,\alpha_4),(\alpha_1,\alpha_2)\}, \{(\alpha_1,\{\{l_1 : \alpha_3\}\})\})
\end{array}
$$

The most general unifier of the kinded set of equations

$$(\{\alpha_1 :: \{\{l_1 : \alpha_3\}\}, \alpha_2 :: \{\{l_1 : \alpha_4, l_2 : \alpha_3\}\}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \{(\alpha_1,\alpha_2)\})$$

is the kinded substitution

$$(\{\alpha_2 :: \{\{l_1 : \alpha_4, l_2 : \alpha_4\}\}, \alpha_4 :: \mathcal{U}\}, \{(\alpha_3,\alpha_4),(\alpha_1,\alpha_2)\}).$$

Theorem 2.4.5. The kinded unification algorithm $\mathcal{U}$ takes any kinded set of equations, computes a most general unifier if one exists, and reports failure otherwise.

Proof. See Theorem 3.4.1 in [46]. □

A stronger occur check condition could have been chosen when eliminating a type variable. This would correspond to not allowing kinding environments with cyclic dependencies but would

increase the complexity of the unification algorithm and the typeability on closed terms does not change [46].

Using the kinded unification algorithm, we can extend the type-scheme inference algorithm we have previously defined, with record-polymorphism. The ***kinded type inference algorithm*** infers the principal typing of a given term in $\Lambda_{\mathcal{O}}$.

Definition 2.4.23. Let Γ be a type assignment, K be a kinding environment, $M \in \Lambda_{\mathcal{O}}$ and $\tau \in \mathbb{T}_{\mathcal{O}}$. The ***kinded type inference function*** $WK(K, \Gamma, M)$ that returns the triple (K', S, τ) such that $K', S(\Gamma) \vdash_{\mathcal{O}} M : \tau$ is defined inductively in Figure 2.5.

Theorem 2.4.6. Given a kinding environment K, a type assignment Γ and $M \in \Lambda_{\mathcal{O}}$. If $WK(K, \Gamma, M) = (K', S, \tau)$ then the following properties hold:

- (K', S) is a kinded substitution that respects K and $K', S(\Gamma) \vdash_{\mathcal{O}} M : \tau$.
- If $K_0, S_0(\Gamma) \vdash_{\mathcal{O}} M : \tau_0$ for some kinded substitutions (K_0, S_0) and $\tau_0 \in \mathbb{T}_{\mathcal{O}}$ such that (K_0, S_0) respects K, then there is some type substitution S' such that the kinded substitution (K_0, S') respects K', $\tau_0 \equiv S'(\tau)$, and $S_0(\Gamma) = S' \circ S(\Gamma)$.

If $WK(K, \Gamma, M)$ fails, then there is no kinded substitution (K_0, S_0) and τ_0 such that (K_0, S_0) respects K and $K_0, S_0(\Gamma) \vdash_{\mathcal{O}} M : \tau_0$.

Proof. See Theorem 3.5.1 in [46]. □

Example 2.4.12. Let $\Gamma = \{x : \alpha, y : \beta\}$ and $K = \{\alpha :: \mathcal{U}, \beta :: \mathcal{U}\}$.

$$
\begin{array}{rrcl}
1) & WK(K, \Gamma, \{l_1 = x, l_2 = y\}.l_1) & = & (K, [\{l_1 : \alpha, l_2 : \beta\}/\alpha_2, \alpha/\alpha_1], \alpha) \\
1.1) & WK(K, \Gamma, \{l_1 = x, l_2 = y\}) & = & (K, id, \{l_1 : \alpha, l_2 : \beta\}) \\
1.1.1) & WK(K, \Gamma, x) & = & (K, id, \alpha) \\
1.1.2) & WK(K, \Gamma, y) & = & (K, id, \beta) \\
1.2) & \mathcal{U}(K\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l_1 : \alpha_1\}\}\}, \{(\alpha_2, \{l_1 : \alpha, l_2 : \beta\}) & = & (K, [\{l_1 : \alpha, l_2 : \beta\}/\alpha_2, \alpha/\alpha_1])
\end{array}
$$

The principal typing of $\{l_1 = x, l_2 = y\}.l_1$ is $(K, [\{l_1 : \alpha, l_2 : \beta\}/\alpha_2, \alpha/\alpha_1], \alpha)$.

This concludes the background chapter. Again, we would like to point the reader to the work of Barendregt [7] for a more complete introduction to the lambda calculus, the work of Hindley [36] for a more complete introduction to the simply typed lambda calculus, both the paper by Damas an Milner [24] and the latter's PhD thesis [23] for a more complete overview of the let-polymorphic lambda calculus, and the work by Ohori [46] for a more complete overview of his polymorphic record calculus. In the next chapter we will introduce the `EVL` language for dealing with events.

$$
\begin{array}{rcl}
WK(K,\Gamma,x) & = & \text{if } x \notin dom(\Gamma) \text{ then } \mathit{fail} \\
& & \text{else let } \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau = \Gamma(x), \\
& & \qquad S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \ (\beta_1, \ldots, \beta_n \text{ are fresh}) \\
& & \quad \text{in } (K\{\beta_1 :: S(\kappa_1), \ldots, \beta_n :: S(\kappa_n)\}, id, S(\tau)) \\
\\
WK(K,\Gamma,M_1\ M_2) & = & \text{let } (K_1,S_1,\tau_1) = WK(K,\Gamma,M_1) \\
& & \quad (K_2,S_2,\tau_2) = WK(K_1,S_1(\Gamma),M_2) \\
& & \quad (K_3,S_3) = \mathcal{U}(K_2\{\alpha :: \mathcal{U}\}, \\
& & \qquad\qquad \{(S_2(\tau_1), \tau_2 \to \alpha)\}) \ (\alpha \text{ is fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha)) \\
\\
WK(K,\Gamma,\lambda x.M) & = & \text{let } (K_1,S_1,\tau) = WK(K\{\alpha :: \mathcal{U}\}, \Gamma\{x : \alpha\}, M) \ (\alpha \text{ fresh}) \\
& & \text{in } (K_1, S_1, S_1(\alpha) \to \tau) \\
\\
WK(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) & = & \text{let } (K_1,S_1,\tau_1) = WK(K,\Gamma,M_1) \\
& & \quad (K_1',\sigma) = \mathit{Cls}(K_1,S_1(\Gamma),\tau_1) \\
& & \quad (K_2,S_2,\tau_2) = WK(K_1',S_1(\Gamma)\{x : \sigma\}, M_2) \\
& & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\\
WK(K,\Gamma,\{l_1 = M_1, \ldots, l_n = M_n\}) & = & \text{let } (K_1,S_1,\tau_1) = WK(K,\Gamma,M_1) \\
& & \quad (K_i,S_i,\tau_i) = WK(K_{i-1}, S_{i-1} \circ \cdots \circ S_1(\Gamma), M_i) \ (2 \le i \le n) \\
& & \text{in } (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \\
& & \quad \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\}) \\
\\
WK(K,\Gamma,M.l) & = & \text{let } (K_1,S_1,\tau_1) = WK(K,\Gamma,M) \\
& & \quad (K_2,S_2) = \mathcal{U}(K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \\
& & \qquad\qquad \{(\alpha_2, \tau_1)\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \text{in } (K_2, S_2 \circ S_1, S_2(\alpha_1)) \\
\\
WK(K,\Gamma,\text{modify}(M_1,l,M_2)) & = & \text{let } (K_1,S_1,\tau_1) = WK(K,\Gamma,M_1) \\
& & \quad (K_2,S_2,\tau_2) = WK(K_1,S_1(\Gamma),M_2) \\
& & \quad (K_3,S_3) = \mathcal{U}(K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \\
& & \qquad\qquad \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\}) \\
& & \quad (\alpha_1, \alpha_2 \text{ are fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha_2))
\end{array}
$$

It is implicitly assumed that the algorithm fails if unification or any of its recursive calls fail.

Figure 2.8: The type inference algorithm *WK* for the λ-Calculus with Polymorphic Records.

Chapter 3

The EVL language for events

In this chapter we define EVL, a higher-order polymorphic typed language, designed to facilitate the specification and processing of events. This language is an extension of a restriction of Ohori's original ML-style polymorphic record calculus. Although our type system is very much based on that system, it is not meant to be a general language, but rather a language purposely designed for dealing with events.

3.1 Terms

Definition 3.1.1. Let x range over an infinite countable set of variables $\mathbb{V}$ and l range over an infinite countable set of labels $\mathbb{L}$. The set of EVL-terms, denoted by $\Lambda_{\texttt{EVL}}$, is given by the following grammar:

$$
\begin{array}{rcll}
M & ::= & c^b & \text{(constants)} \\
 & | & x & \text{(variables)} \\
 & | & (MM) & \text{(function application)} \\
 & | & \lambda x.M & \text{(functional abstraction)} \\
 & | & \text{let } x = M \text{ in } M & \text{(let declaration)} \\
 & | & \{l = M, \ldots, l = M\} & \text{(records)} \\
 & | & M.l & \text{(field selection)} \\
 & | & \text{modify}(M, l, M) & \text{(field update)} \\
 & | & \text{if } M \text{ then } M \text{ else } M & \text{(conditional branching)} \\
 & | & \text{letEv } x = M \text{ in } M & \text{(event declaration)}
\end{array}
$$

where b is a base type from a set of base types $\mathbb{B}$ that will always include at least the boolean base type *Bool*.

Notation 3.1.1. We will assume the following equivalences:

$$\text{let } f x_1 \ldots x_n = M \text{ in } N \equiv \text{let } f = \lambda x_1 \ldots x_n.M \text{ in } N$$
$$\text{letEv } f x_1 \ldots x_n = M \text{ in } N \equiv \text{letEv } f = \lambda x_1 \ldots x_n.M \text{ in } N$$

In addition, we will convention that event-names will always start with a capital letter to help distinguish them from functions.

Other potentially useful constructors to the language like tuples and projections were not added to the language to keep it minimal. Nevertheless, pairs (M, N) can be encoded as records of the form $\{\text{fst} = M, \text{snd} = N\}$ and projections $\pi_1(M, N)$ and $\pi_2(M, N)$ can be encoded using field selections $\{\text{fst} = M, \text{snd} = N\}.\text{fst}$ and $\{\text{fst} = M, \text{snd} = N\}.\text{snd}$. And this encoding can be trivially extended to tuples in general. For this reason, we will use this encoding when writing examples whenever necessary.

Example 3.1.1. Let $\text{location}, \text{fire\_danger} \in \mathbb{L}$ and $\mathit{String} \in \mathbb{B}$. The following is a well-formed term from Λ_{EVL}:

$$\text{letEv FireDanger } l\ d = \{\text{location} = l, \text{fire\_danger} = d\} \text{ in FireDanger } \texttt{"Porto"}^{\mathit{String}}\ \texttt{"low"}^{\mathit{String}}$$

Another convenient encoding we are going to use is for lists. The empty list can be encoded as the record $\{\text{empty} = \texttt{True}^{\mathit{Bool}}\}$ and the list whose head is the $M \in \Lambda_{\text{EVL}}$ and tail is $N \in \Lambda_{\text{EVL}}$ can be encoded as the record $\{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = M, \text{tail} = N\}$.

Notation 3.1.2. We will also consider the following equivalences, for readability purposes:

$$\begin{aligned} \text{nil} &\equiv \{\text{empty} = \texttt{True}^{\mathit{Bool}}\} \\ \text{cons} &\equiv \lambda xy.\{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = x, \text{tail} = y\}. \end{aligned}$$

The fact that a more realistic approach to adding lists to the language is not adopted lays in the fact that it is intended to be minimal. For this reason we refrain from adding lists (and recursive types) as primitive notion of the language and instead assume that $\text{nil}, \text{cons} \in \mathbb{V}$ and that the following type-declarations are present in every typing environment:

$$\begin{aligned} \text{nil} &: \{\text{empty} : \mathit{Bool}\} \\ \text{cons} &: \forall \alpha :: \mathcal{U}.\beta :: \{\{\text{empty} : \mathit{Bool}\}\}.\alpha \to \beta \to \{\text{empty} : \mathit{Bool}, \text{head} : \alpha, \text{tail} : \beta\}. \end{aligned}$$

3.2 Types and Kinds

Definition 3.2.1. Let α range over an infinite countable set of type variables $\mathbb{A}$, l range over an infinite countable set of labels $\mathbb{L}$ and b range over a set of base types $\mathbb{B}$. The set of types,

denoted by $\mathbb{T}_{\text{EVL}}$, and the set of kinds, denoted by $\mathbb{K}_{\text{EVL}}$, are given by the following grammar:

$$
\begin{array}{rcll}
\tau & ::= & b & \text{(base types)} \\
 & | & \alpha & \text{(type variables)} \\
 & | & \tau \to \tau & \text{(function types)} \\
 & | & \{l : \tau, \ldots, l : \tau\} & \text{(record types)} \\
 & & & \\
\sigma & ::= & \tau & \text{(monomorphic types)} \\
 & | & \forall \alpha :: \kappa . \sigma & \text{(polymorphic types)} \\
 & & & \\
\rho & ::= & b \mid \alpha \mid \tau \to \rho & \text{(event field types)} \\
 & & & \\
\gamma & ::= & \tau \to \{l : \rho, \ldots, l : \rho\} \mid \{l : \rho, \ldots, l : \rho\} & \text{(event types)} \\
 & & & \\
\kappa & ::= & \mathcal{U} & \text{(universal kind)} \\
 & | & \{\{l : \tau, \ldots, l : \tau\}\} & \text{(record kinds)}
\end{array}
$$

The distinction of ρ and γ (which are actually both included in τ) is necessary to accurately represent the type of event definitions and its purpose will become clear when we present the typing rules.

We assume that the labels that appear in any type or kind are always pairwise distinct.

Note that event-records cannot be nested but records (in general) can. This is fundamental in order to encode lists using records.

Note that the set of kinds $\mathbb{K}_{\text{EVL}}$ is the same as $\mathbb{K}_{\mathcal{O}}$. For this reason the set of kinding rules is the same as the one in Definition 2.4.10 and the reader may assume that every concept related with kinds introduced in Section 2.4 is also valid for this calculus.

The reader may also assume that any concept related to the typing environment, free type-variables, essentially-free type-variables, substitution, generic instance, closure, and well-formedness introduced in Section 2.4 is also valid for this calculus.

Definition 3.2.2. We say that $M \in \Lambda_{\text{EVL}}$ can be assigned the $\tau \in \mathbb{T}_{\text{EVL}}$ according to the kinding assignment K and the typing assignment Γ, if the statement $M : \tau$ can be derived from K, Γ using the typing rules in Figure 3.1.

Example 3.2.1. Let $M \equiv \{\text{location} = l, \text{fire\_danger} = d\}$, $\tau_1 \equiv \{\text{location} : \alpha_1, \text{fire\_danger} : \alpha_2\}$, $\tau_1' \equiv \{\text{location} : \mathit{String}, \text{fire\_danger} : \mathit{String}\}$, $\tau_2 \equiv \forall \alpha_1 :: \mathcal{U} . \forall \alpha_2 :: \mathcal{U} . \alpha_1 \to \alpha_2 \to \{\text{location} : \alpha_1, \text{fire\_danger} : \alpha_2\}$, and $\tau_2' \equiv \mathit{String} \to \mathit{String} \to \{\text{location} : \mathit{String}, \text{fire\_danger} : \mathit{String}\}$. We can deduce letEv FireDanger $= \lambda l . \lambda d . M$ in FireDanger `"Porto"` `"low"` $: \tau_1'$ from the empty kinding and typing environments as follows:

$$
\Delta_1 = \dfrac{\dfrac{\dfrac{\dfrac{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\} \Vdash_{\mathcal{O}} l : \alpha_1 \geq l : \alpha_1}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \vdash_{\text{EVL}} l : \alpha_1}\,(\text{Var}) \quad \dfrac{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\} \Vdash_{\mathcal{O}} d : \alpha_2 \geq d : \alpha_2}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \vdash_{\text{EVL}} d : \alpha_2}\,(\text{Var})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\} \vdash_{\text{EVL}} M : \tau_1}\,(\text{Rec})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1\} \vdash_{\text{EVL}} \lambda d . M : \alpha_2 \to \tau_1}\,(\text{Abs})}{\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \emptyset \vdash_{\text{EVL}} \lambda l . \lambda d . M : \alpha_1 \to \alpha_2 \to \tau_1}\,(\text{Abs})
$$

$$\frac{\Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_{\texttt{EVL}} c^b : b} \text{ (Const)}$$

$$\frac{K \Vdash_{\mathcal{O}} \Gamma(x) \geq \tau \qquad \Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_{\texttt{EVL}} x : \tau} \text{ (Var)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M : \tau \to \tau' \qquad K, \Gamma \vdash_{\texttt{EVL}} N : \tau}{K, \Gamma \vdash_{\texttt{EVL}} M\ N : \tau'} \text{ (App)}$$

$$\frac{K, \Gamma\{x : \tau\} \vdash_{\texttt{EVL}} M : \tau'}{K, \Gamma \vdash_{\texttt{EVL}} \lambda x.M : \tau \to \tau'} \text{ (Abs)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M_1 : \mathit{Bool} \qquad K, \Gamma \vdash_{\texttt{EVL}} M_2 : \tau \qquad K, \Gamma \vdash_{\texttt{EVL}} M_3 : \tau}{K, \Gamma \vdash_{\texttt{EVL}} \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau} \text{ (Cond)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M : \tau \qquad \mathit{Cls}(K, \Gamma, \tau) = (K', \sigma) \qquad K', \Gamma\{x : \sigma\} \vdash_{\texttt{EVL}} N : \tau'}{K', \Gamma \vdash_{\texttt{EVL}} \text{let } x = M \text{ in } N : \tau'} \text{ (Let)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M : \gamma \qquad \mathit{Cls}(K, \Gamma, \gamma) = (K', \sigma) \qquad K', \Gamma\{x : \sigma\} \vdash_{\texttt{EVL}} N : \tau}{K', \Gamma \vdash_{\texttt{EVL}} \text{letEv } x = M \text{ in } N : \tau} \text{ (LetEv)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M_i : \tau_i, 1 \leq i \leq n}{K, \Gamma \vdash_{\texttt{EVL}} \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{ (Rec)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M : \tau \qquad K \Vdash_{\mathcal{O}} \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash_{\texttt{EVL}} M.l : \tau'} \text{ (Sel)}$$

$$\frac{K, \Gamma \vdash_{\texttt{EVL}} M : \tau \qquad K, \Gamma \vdash_{\texttt{EVL}} N : \tau' \qquad K \Vdash_{\mathcal{O}} \tau :: \{\{l : \tau'\}\}}{K, \Gamma \vdash_{\texttt{EVL}} \text{modify}(M, l, N) : \tau} \text{ (Modif)}$$

Figure 3.1: Typing rules for EVL.

$$\Delta_2 = \mathit{Cls}(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \Gamma, \alpha_1 \to \alpha_2 \to \tau_1) = (\emptyset, \tau_2)$$

$$\Delta_3 = \frac{\dfrac{\emptyset \Vdash_{\mathcal{O}} \text{FireDanger} : \tau_2 \geq \text{FireDanger} : \tau_2'}{\emptyset, \{\text{FireDanger} : \tau_2\} \vdash_{\texttt{EVL}} \text{FireDanger} : \tau_2'} \text{ (Var)} \qquad \dfrac{}{\emptyset, \{\text{FireDanger} : \tau_2\} \vdash_{\texttt{EVL}} \texttt{"Porto"}^{\mathit{String}} : \mathit{String}} \text{ (Const)}}{\emptyset, \{\text{FireDanger} : \tau_2\} \vdash_{\texttt{EVL}} \text{FireDanger } \texttt{"Porto"}^{\mathit{String}} : \mathit{String} \to \tau_1'} \text{ (App)}$$

$$\Delta_4 = \frac{\Delta_3 \qquad \dfrac{}{\emptyset, \{\text{FireDanger} : \tau_2\} \vdash_{\texttt{EVL}} \texttt{"low"}^{\mathit{String}} : \mathit{String}} \text{ (Const)}}{\emptyset, \{\text{FireDanger} : \tau_2\} \vdash_{\texttt{EVL}} \text{FireDanger } \texttt{"Porto"}^{\mathit{String}} : \tau_1'} \text{ (App)}$$

$$\frac{\Delta_1 \qquad \Delta_2 \qquad \Delta_4}{\emptyset, \emptyset \vdash_{\texttt{EVL}} \text{letEv FireDanger} = \lambda l.\lambda d.M \text{ in FireDanger } \texttt{"Porto"}\ \texttt{"low"} : \tau_1'} \text{ (LetEv)}$$

3.3 Operational Semantics

As a model of an ML-style programming language, we require EVL to have a stronger property of type soundness than the subject reduction property, i.e., that evaluation of a closed term of some type always yields a value of that type. For this reason, we define a call-by-value operational semantics using evaluation contexts based on the operational semantics in [46], which serves as an evaluation model of a polymorphically-typed programming language with records.

Definition 3.3.1. Let v range over the set of values $\mathcal{V}$ given by the following grammar:

$$v ::= c^b \mid \lambda x.M \mid \{l = v, \ldots, l = v\}$$

Definition 3.3.2. Let $[\bullet]$ represent the empty context ($\bullet$ is called a hole). Evaluation contexts identify terms that are to be evaluated and the set of evaluation contexts (ranged over by $\mathcal{E}[\,]$) is given by the following grammar:

$$\begin{aligned}\mathcal{E}[\,] \quad ::= \quad & [\bullet] \mid \mathcal{E}[\,]\ M \mid v\ \mathcal{E}[\,] \mid \text{if } \mathcal{E}[\,] \text{ then } M_1 \text{ else } M_2 \\ & \text{let } x = \mathcal{E}[\,] \text{ in } M \mid \text{letEv } x = \mathcal{E}[\,] \text{ in } M \\ & \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = \mathcal{E}[\,], \ldots\} \mid \mathcal{E}[\,].l \\ & \text{modify}(\mathcal{E}[\,], l, M) \mid \text{modify}(v, l, \mathcal{E}[\,])\end{aligned}$$

Definition 3.3.3. Let $\mathcal{E}[M]$ be the term obtained by placing M in the hole of the evaluation context $\mathcal{E}[\,]$. The set of call-by-value context-rewriting rules is given by the following transformation rules:

$$\begin{aligned}\mathcal{E}[(\lambda x.M)\ v] &\to_{\mathcal{E}} \mathcal{E}[M[v/x]] \\ \mathcal{E}[\text{if } \texttt{True} \text{ then } M_1 \text{ else } M_2] &\to_{\mathcal{E}} \mathcal{E}[M_1] \\ \mathcal{E}[\text{if } \texttt{False} \text{ then } M_1 \text{ else } M_2] &\to_{\mathcal{E}} \mathcal{E}[M_2] \\ \mathcal{E}[\text{let } x = v \text{ in } M] &\to_{\mathcal{E}} \mathcal{E}[M[v/x]] \\ \mathcal{E}[\text{letEv } x = v \text{ in } M] &\to_{\mathcal{E}} \mathcal{E}[M[v/x]] \\ \mathcal{E}[\{l_1 = v_1, \ldots, l_n = v_n\}.l_i] &\to_{\mathcal{E}} \mathcal{E}[v_i] \\ \mathcal{E}[\text{modify}(\{l_1 = v_1, \ldots, l_n = v_n\}, l_i, v)] &\to_{\mathcal{E}} \mathcal{E}[\{l_1 = v_1, \ldots, l_i = v, \ldots, l_n = v_n\}]\end{aligned}$$

Definition 3.3.4. Let $M, N \in \Lambda_{\text{EVL}}$.

- We say that M reduces to N in one $\mathcal{E}$-evaluation step, denoted $M \to^1_{\mathcal{E}} N$, if there exists an evaluation context $\mathcal{E}[\,]$ and $M_1, M_2 \in \Lambda_{\text{EVL}}$ such that $M \equiv \mathcal{E}[M_1]$, $\mathcal{E}[M_1] \to_{\mathcal{E}} \mathcal{E}[M_2]$, and $N \equiv \mathcal{E}[M_2]$.
- We define $\to_{\mathcal{E}}$ as the reflexive and transitive closure of $\to^1_{\mathcal{E}}$.
- We write $M \downarrow N$ if $M \to_{\mathcal{E}} N$ and there is no $N' \in \Lambda_{\text{EVL}}$ such that $N \to_{\mathcal{E}} N'$.

To show the type soundness with respect to this operational semantics, we need to define a type-indexed family of predicates on closed values.

Definition 3.3.5. For a closed type $\sigma \in \mathbb{T}_{\text{EVL}}$, let $\mathcal{V}^\sigma = \{v \mid \emptyset, \emptyset \vdash_{\text{EVL}} v : \sigma\}$ and $p^\sigma \subseteq \mathcal{V}^\sigma$ be defined inductively on σ as follows:

$$v \in p^b \quad \Leftrightarrow \quad v = c^b \text{ for some constant } c^b$$

$$v \in p^{\forall\alpha_1::\kappa_1\cdots\forall\alpha_n::\kappa_n.\tau} \quad \Leftrightarrow \quad \text{for any ground substitution } S \text{ such that } dom(S) = \{\alpha_1, \ldots, \alpha_n\} \text{ and it respects } \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}, v \in p^{S(\tau)}$$

$$v \in p^{\tau\to\tau'} \quad \Leftrightarrow \quad \forall v_0 \in p^\tau, \text{ if } (v\ v_0) \downarrow M \text{ then } M \in p^{\tau'}$$

$$v \in p^{\{l_1:\tau_1,\ldots,l_n:\tau_n\}} \quad \Leftrightarrow \quad v \equiv \{l_1 = v_1, \ldots, l_n = v_n\} \text{ such that } v_i \in p^{\tau_i}, (1 \le i \le n)$$

Definition 3.3.6. Let Γ be a closed typing environment.

- A Γ-environment is a function, denoted by η, such that $dom(\eta) = dom(\Gamma)$ and for any $x \in dom(\Gamma)$, $\eta(x) \in v^{\Gamma(x)}$.
- We write $\eta(M)$ for the term obtained from $M \in \Lambda_{\text{EVL}}$ by substituting $\eta(x)$ for each free occurrence of x in M.

For a function f, if $x \notin dom(f)$, we write $f\{x \mapsto v\}$ for the extension f' of f to x such that $f'(x) = v$.

Theorem 3.3.1. If $K, \Gamma \vdash_{\text{EVL}} M : \sigma$ then for any ground substitution S that respects K, and for any $S(\Gamma)$-environment η, if $\eta(M) \downarrow N$, then $N \in p^{S(\sigma)}$.

Proof. Let S be any ground substitution respecting K, and let η be any $S(\Gamma)$-environment. This can be proved by induction on the typing derivation.

- Case (Const): Trivial.
- Case (Var): Suppose $K, \Gamma \vdash_{\text{EVL}} x : \tau$. Then $K, \Gamma \Vdash_{\mathcal{O}} \Gamma(x) \ge \tau$. Let $\forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau_0 = \Gamma(x)$. Then there is some S_0 such that $dom(S_0) = \{\alpha_1, \ldots, \alpha_n\}$, $\tau = S_0(\tau_0)$, and $K \vdash_{\text{EVL}} S_0(\alpha_i) :: S_0(\kappa_i)$. By Lemma 2.4.1, $\emptyset \Vdash_{\mathcal{O}} S(S_0(\alpha_i)) :: S(S_0(\kappa_i))$. By the bound type variable convention, $S(S_0(\tau_0)) = (S \circ S_0)(S(\tau_0))$ and $S(S_0(\kappa_i)) = (S \circ S_0)(S(\kappa_i))$, since S_0 only affects bound variables. This means that $S \circ S_0$ is a ground substitution respecting $\{\alpha_1 :: S(\kappa_1), \ldots, \alpha_n :: S(\kappa_n)\}$. Now, suppose that $\eta(x) \downarrow M'$. Then by the assumption $M' \in p^{\forall\alpha_1::S(\kappa_1)\cdots\forall\alpha_n::S(\kappa_n).S(\tau_0)}$. By the definition of p, we have that $M' \in p^{(S\circ S_0)(S(\tau_0))} = p^{S(S_0(\tau_0))} = p^{S(\tau)}$.
- Case (App): Suppose $K, \Gamma \vdash_{\text{EVL}} M_1\ M_2 : \tau_2$ is derived from $K, \Gamma \vdash_{\text{EVL}} M_1 : \tau_1 \to \tau_2$ and $K, \Gamma \vdash_{\text{EVL}} M_2 : \tau_1$. Now, also suppose that $\eta(M_1\ M_2) \downarrow M'$. By the definition of evaluation contexts, $\eta(M_1) \downarrow M_1'$ and $(M_1'\ \eta(M_2)) \downarrow M$, since $(M_1\ M_2)$ fits $(\mathcal{E}[\ ]\ M)$ and $(M_1'\ M_2)$ fits $(v\ \mathcal{E}[\ ])$. By the induction hypothesis for M_1, we have that $M_1' = v_1 \in p^{S(\tau_1)\to S(\tau_2)}$ for some value v_1. But, by the definition of evaluation contexts, $\eta(M_2) \downarrow M_2'$ and $(v_1\ M_2') \downarrow M'$ and,

by the induction hypothesis for M_2, we have that $M_2' = v_2 \in p^{S(\tau_1)}$ for some value v_2. By the definition of p, we have $M' \in p^{S(\tau_2)}$.

- Case (Abs): Suppose $K, \Gamma \vdash_{\text{EVL}} \lambda x.M_1 : \tau_1 \to \tau_2$ is derived from $K, \Gamma \cup \{x : \tau_1\} \vdash_{\text{EVL}} M_1 : \tau_2$. Then, $\eta(\lambda x.M_1) = \lambda x.\eta(M_1) \downarrow \lambda x.\eta(M_1)$. This means that, if we want to see what happens to the type of $\lambda x.M_1$ during evaluation, we have to apply it to a value of type $S(\tau_1)$. Let v be any element in $p^{S(\tau_1)}$ and suppose $(\lambda x.\eta(M_1))\ v \downarrow M'$. By the definition of evaluation contexts, $[v/x](\eta(M_1)) \downarrow M'$, i. e., $\eta\{x \mapsto v\}(M_1) \downarrow M'$. Since $\eta\{x \mapsto v\}$ is a $S(\Gamma \cup \{x : \tau_1\})$-environment, by the induction hypothesis, $M' \in p^{S(\tau_2)}$. By the definition of p, this proves that $\lambda x.\eta(M_1) \in p^{S(\tau_1) \to S(\tau_2)}$.

- Case (Let): Suppose $K, \Gamma_x \vdash_{\text{EVL}} \text{let } x = M_1 \text{ in } M_2 : \tau$ is derived from $K', \Gamma_x \vdash_{\text{EVL}} M_1 : \tau'$, $\mathit{Cls}(K', \Gamma_x, \tau') = (K, \sigma)$, and $K, \Gamma_x \cup \{x : \sigma\} \vdash_{\text{EVL}} M_2 : \tau$. Then, there are some $\alpha_1, \ldots, \alpha_n$ and $\kappa_1, \ldots, \kappa_n$ such that $K = K' \cup \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ and $\sigma = \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau'$. By the bound type variable convention, we can assume that any $\{\alpha_1, \ldots, \alpha_n\}$ do not appear in S. Then $S(\sigma) = \forall \alpha_1 :: S(\kappa_1) \cdots \forall \alpha_n :: S(\kappa_n).S(\tau')$. Let S' be any ground substitution such that $\mathit{dom}(S') = \{\alpha_1, \ldots, \alpha_n\}$ and S' respects $\{\alpha_1 :: S(\kappa_1), \ldots, \alpha_n :: S(\kappa_n)\}$. Then $S' \circ S$ is a ground substitution that respects $K' \cup \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\} = K$, and η is a $(S' \circ S)(\Gamma_x)$-environment. Therefore, by the induction hypothesis for M_1, if $\eta(M_1) \downarrow M_1'$, then $M_1' = v_1 \in p^{S'(S(\tau'))}$ and, by the definition of p, $v_1 \in p^{S(\sigma)}$. Now, suppose that $\eta(\text{let } x = M_1 \text{ in } M_2) \downarrow M'$. By the definition of evaluation contexts, $\eta(M_1) \downarrow M_1'$ and $(\text{let } x = M_1' \text{ in } \eta(M_2)) \downarrow M'$. This means that $M_1' = v_1 \in p^{S(\sigma)}$. By the definition of evaluation contexts, $[v_1/x](\eta(M_2)) \downarrow M'$, i. e., $\eta\{x \mapsto v_1\}(M_2) \downarrow M'$. Since $\eta\{x \mapsto v_1\}$ is a $S(\Gamma \cup \{x : \sigma\})$-environment, then $M \in p^{S(\tau)}$.

- Case (LetEv): Similar to the case for (Let).

- Case (Rec): Suppose $K, \Gamma \vdash_{\text{EVL}} \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is derived from $K, \Gamma \vdash_{\text{EVL}} M_i : \tau_i, (1 \le i \le n)$. Now, also suppose that $\eta(\{l_1 = M_1, \ldots, l_n = M_n\}) \downarrow M$. By the definition of evaluation contexts, $(\{l_1 = \eta(M_1), \ldots, l_n = \eta(M_n)\}) \downarrow M'$. But, by the induction hypothesis, $\eta(M_i) \downarrow M_i'$, $M_i' = v_i \in p^{S(\tau_i)}$, for some value v_i. Thus, by the definition of p, we have $M' \in p^{\{l_1 : S(\tau_1), \ldots, l_n : S(\tau_n)\}}$.

- Case (Sel): Suppose $K, \Gamma \vdash_{\text{EVL}} M.l : \tau$ is derived from $K, \Gamma \vdash_{\text{EVL}} M : \tau'$ and $K \vdash_{\text{EVL}} \tau' :: \{\{l : \tau\}\}$. Now, also suppose that $\eta(M.l) \downarrow M'$. By the definition of evaluation contexts, we have that $\eta(M) \downarrow M''$ and $M''.l \downarrow M'$. By the induction hypothesis, we have that $M'' = v \in p^{S(\tau)}$ for some value v. Since S is a ground substitution that respects K, by Lemma 2.4.1, we have that $\emptyset \vdash_{\text{EVL}} S(\tau') :: \{\{l : S(\tau)\}\}$. This implies that $S(\tau')$ is a ground record type of the form $\{\ldots, l : S(\tau), \ldots\}$. Thus, by the definition of p, $v = \{\ldots, l = v', \ldots\}$, $v' \in p^{S(\tau)}$. But $\{\ldots, l = v', \ldots\}.l \downarrow v'$.

- Case (Modif): Suppose $K, \Gamma \vdash_{\text{EVL}} \text{modify}(M_1, l, M_2) : \tau$ is derived from $K, \Gamma \vdash_{\text{EVL}} M_1 : \tau$, $K, \Gamma \vdash_{\text{EVL}} M_2 : \tau'$, and $K \Vdash_{\mathcal{O}} \tau :: \{\{l : \tau'\}\}$. Now, also suppose $\eta(\text{modify}(M_1, l, M_2)) \downarrow M'$. By the definition of evaluation contexts, $\eta(M_1) \downarrow M_1'$ and $\text{modify}(M_1', l, \eta(M_2)) \downarrow M'$. By the induction hypothesis for M_1, $M_1' = v_1 \in p^{S(\tau)}$, for some value v_1. Since S is

a ground substitution that respects K, by Lemma 2.4.1, $\emptyset \Vdash_{\mathcal{O}} S(\tau) :: \{\{l : S(\tau')\}\}$. This implies that $S(\tau)$ is a ground record type of the form $\{\ldots, l : S(\tau'), \ldots\}$. By the definition of p, $v_1 = \{\ldots, l = v, \ldots\}, v \in p^{S(\tau')}$. By the definition of evaluation contexts, $\eta(M_2) \downarrow M_2'$ and $\text{modify}(M_1', l, M_2') \downarrow M'$. By the induction hypothesis for M_2, $M_2' = v_2 \in p^{S(\tau')}$, for some value v_2. But $\text{modify}(\{\ldots, l = v, \ldots\}, l, v_2) \downarrow \{\ldots, l : v_2, \ldots\}$ and $\{\ldots, l = v_2, \ldots\} \in p^{\{\ldots, l:S(\tau'), \ldots\}} = p^{S(\tau)}$.

□

Example 3.3.1. Let $K = \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l_1 : \alpha_1\}\}\}$ and $\Gamma = \{x_1 : \alpha_1, x_2 : \alpha_2\}$. Then $K, \Gamma \vdash_{\texttt{EVL}} x_2.l_1 : \alpha_1$ and $S = [Bool/\alpha_1, \{l_1 : \alpha_1, l_2 : \alpha_1\}/\alpha_2]$ respects K.

Consider the following $S(\Gamma)$-environment:

$$\begin{array}{rcl} \eta : \{x_1, x_2\} & \rightarrow & Bool \\ x_1 & \mapsto & \texttt{True}^{Bool} \\ x_2 & \mapsto & \{l_1 = \texttt{True}^{Bool}, l_2 = \texttt{False}^{Bool}\}. \end{array}$$

Then $\eta(x_2.l_1) \equiv \{l_1 = \texttt{True}^{Bool}, l_2 = \texttt{False}^{Bool}\}.l_1$, $\{l_1 = \texttt{True}^{Bool}, l_2 = \texttt{False}^{Bool}\}.l_1 \downarrow \texttt{True}^{Bool}$ and $\texttt{True}^{Bool} \in p^{S(\alpha_1)} = p^{Bool}$.

3.4 Type Inference

The ***kinded type inference algorithm*** for EVL is a natural extension of the kinded type inference algorithm introduced in Section 2.4 of the previous chapter. The ***kinded unification algorithm*** for EVL is exactly the same as the one defined in that section. The reason is that the set of types for the EVL language is also the same as the one defined in that section.

Definition 3.4.1. Let Γ be a type assignment, K be a kinding environment, $M \in \Lambda_{\texttt{EVL}}$ and $\tau \in \mathbb{T}_{\texttt{EVL}}$. The ***EVL type inference function*** $WE(K, \Gamma, M)$ that returns the triple (K', S, τ) such that $K', S(\Gamma) \vdash_{\texttt{EVL}} M : \tau$ is defined inductively in Figure 3.2.

Theorem 3.4.1. Given a kinding environment K, a type assignment Γ and $M \in \Lambda_{\texttt{EVL}}$. If $WE(K, \Gamma, M) = (K', S, \tau)$ then the following properties hold:

- (K', S) is a kinded substitution that respects K and $K', S(\Gamma) \vdash_{\texttt{EVL}} M : \tau$.
- If $K_0, S_0(\Gamma) \vdash_{\texttt{EVL}} M : \tau_0$ for some kinded substitutions (K_0, S_0) and $\tau_0 \in \mathbb{T}_{\texttt{EVL}}$ such that (K_0, S_0) respects K, then there is some type substitution S' such that the kinded substitution (K_0, S') respects K', $\tau_0 \equiv S'(\tau)$, and $S_0(\Gamma) = S' \circ S(\Gamma)$.

If $WE(K, \Gamma, M)$ fails, then there is no kinded substitution (K_0, S_0) and τ_0 such that (K_0, S_0) respects K and $K_0, S_0(\Gamma) \vdash_{\texttt{EVL}} M : \tau_0$.

$$
\begin{array}{rcl}
WE(K,\Gamma,x) & = & \text{if } x \notin dom(\Gamma) \text{ then } \mathit{fail} \\
& & \text{else let } \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau = \Gamma(x), \\
& & \qquad S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \ (\beta_1, \ldots, \beta_n \text{ are fresh}) \\
& & \quad \text{in } (K \cup \{\beta_1 :: S(\kappa_1), \ldots, \beta_n :: S(\kappa_n)\}, id, S(\tau)) \\
\\
WE(K,\Gamma,M_1\ M_2) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M_1) \\
& & \quad (K_2,S_2,\tau_2) = WE(K_1,S_1(\Gamma),M_2) \\
& & \quad (K_3,S_3) = \mathcal{U}(K_2\{\alpha :: \mathcal{U}\}, \\
& & \qquad\qquad \{(S_2(\tau_1), \tau_2 \to \alpha)\}) \ (\alpha \text{ is fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha)) \\
\\
WE(K,\Gamma,\lambda x.M) & = & \text{let } (K_1,S_1,\tau) = WE(K \cup \{\alpha :: \mathcal{U}\}, \Gamma \cup \{x : \alpha\}, M) \ (\alpha \text{ fresh}) \\
& & \text{in } (K_1, S_1, S_1(\alpha) \to \tau) \\
\\
WE(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M_1) \\
& & \quad (K_1',\sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \tau_1) \\
& & \quad (K_2,S_2,\tau_2) = WE(K_1', S_1(\Gamma) \cup \{x : \sigma\}, M_2) \\
& & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\\
WE(K,\Gamma,\text{letEv } x = M_1 \text{ in } M_2) & = & \text{let } (K_1,S_1,\gamma) = WE(K,\Gamma,M_1) \\
& & \quad (K_1',\sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \gamma) \\
& & \quad (K_2,S_2,\tau_2) = WE(K_1', S_1(\Gamma) \cup \{x : \sigma\}, M_2) \\
& & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\\
WE(K,\Gamma,\{l_1 = M_1, \ldots, l_n = M_n\}) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M_1) \\
& & \quad (K_i,S_i,\tau_i) = WE(K_{i-1}, S_{i-1} \circ \cdots \circ S_1(\Gamma), M_i) \ (2 \le i \le n) \\
& & \text{in } (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \\
& & \quad \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\}) \\
\\
WE(K,\Gamma,M.l) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M) \\
& & \quad (K_2,S_2) = \mathcal{U}(K_1 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \\
& & \qquad\qquad \{(\alpha_2, \tau_1)\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \text{in } (K_2, S_2 \circ S_1, S_2(\alpha_1)) \\
\\
WE(K,\Gamma,\text{modify}(M_1,l,M_2)) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M_1) \\
& & \quad (K_2,S_2,\tau_2) = WE(K_1, S_1(\Gamma), M_2) \\
& & \quad (K_3,S_3) = \mathcal{U}(K_2 \cup \{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1\}\}\}, \\
& & \qquad\qquad \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\}) \ (\alpha_1, \alpha_2 \text{ are fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, S_3(\alpha_2)) \\
\\
WE(K,\Gamma,\text{if } M_1 \text{ then } M_2 \text{ else } M_3) & = & \text{let } (K_1,S_1,\tau_1) = WE(K,\Gamma,M_1) \\
& & \quad (K_2,S_2) = \mathcal{U}(K_1, \{(\tau_1, \mathit{Bool})\}) \\
& & \quad (K_3,S_3,\tau_2) = WE(K_2, S_2 \circ S_1(\Gamma), M_2) \\
& & \quad (K_4,S_4,\tau_3) = WE(K_3, S_3 \circ S_2 \circ S_1(\Gamma), M_3) \\
& & \quad (K_5,S_5) = \mathcal{U}(K_4, \{(S_4(\tau_2), \tau_3)\}) \\
& & \text{in } (K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1, S_5 \circ S_4(\tau_2))
\end{array}
$$

Figure 3.2: The type inference algorithm *WE* for EVL.

Proof. We start by proving the soundness of the type inference algorithm by induction on the structure of M. We only show the case for if M_1 then M_2 else M_3 and letEv $x = M_1$ in M_2. The case for c^b is trivial and the remaining cases are similar to the corresponding proof in [46].

- $M \equiv$ if M_1 then M_2 else M_3. Suppose that $WE(K, \Gamma, \text{if } M_1 \text{ then } M_2 \text{ else } M_3) = (K', S, \tau)$. Then $WE(K, \Gamma, \text{if } M_1 \text{ then } M_2 \text{ else } M_3) = (K', S, \tau)$, $\mathcal{U}(K_1, \{(\tau_1, Bool)\}) = (K_2, S_2)$, $WE(K_2, S_2 \circ S_1(\Gamma), M_2) = (K_3, S_3, \tau_2)$, $WE(K_3, S_3 \circ S_2 \circ S_1(\Gamma), M_3) = (K_4, S_4, \tau_3)$, $\mathcal{U}(K_4, \{(S_4(\tau_2), \tau_3)\}) = (K_5, S_5)$, $K' = K_5$, $S = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1$ and $\tau = S_5(\tau_3)$. First, we show that $(K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1)$ respects K. By the correction of the unification algorithm, we know that (K_5, S_5) respects K_4 and (K_2, S_2) respects K_1. We also know that $S_5 \circ S_4(\tau_2) = S_5(\tau_3)$ and $S_2(\tau_1) = S_2(b)$, i.e., $S_2(\tau_1) = b$. By the induction hypothesis, we know that (K_4, S_4) respects K_3, (K_3, S_3) respects K_2 and (K_1, S_1) respects K. By applying Lemma 2.4.1 as many times as needed, we have that $(K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1)$ respects K. Now, we are left to show that $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : S_5(\tau_3)$. By the induction hypothesis, we know that $K_1, S_1(\Gamma) \vdash_{\text{EVL}} M_1 : \tau_1$, $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_2 : \tau_2$ and $K_4, S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_3 : \tau_3$. By Lemma 2.4.3, we have that $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_1 : S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\tau_1)$, i.e., $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_1 : S_5 \circ S_4 \circ S_3(b)$, $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_1 : b$, $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_2 : S_5 \circ S_4(\tau_2)$, i.e., $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_2 : S_5(\tau_3)$, and $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} M_3 : S_5(\tau_3)$. Finally, by (Cond), we have that $K_5, S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) \vdash_{\text{EVL}} \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : S_5(\tau_3)$.

- $M \equiv \text{letEv } x = M_1 \text{ in } M_2$. Suppose that $WK(K, \Gamma, \text{letEv } x = M_1 \text{ in } M_2) = (K', S, \tau)$. Then $WK(K, \Gamma, M_1) = (K_1, S_1, \gamma)$, $Cls(K_1, S_1(\Gamma), \gamma) = (K_1', \sigma)$, $WK(K_1', S_1(\Gamma) \cup \{x : \sigma\}, M_2) = (K_2, S_2, \tau')$, $K' = K_2$, $S = S_2 \circ S_1$, and $\tau = \tau'$. First, we show that $(K_2, S_2 \circ S_1)$ respects K. By the induction hypothesis, we know that (K_2, S_2) respects K_1'. If we show that (K_1', S_1) respects K then, by Lemma 2.4.1, we know that $(K_2, S_2 \circ S_1)$ respects K. Since we want to show that (K_1', S_1) respects K, we need to show that $\forall \alpha \in dom(K), K_1' \Vdash_{\mathcal{O}} S_1(\alpha) :: S_1(K(\alpha))$, i.e., $FTV(S_1(\alpha)) \subseteq dom(K_1')$. We know that $\forall \alpha \in dom(K), K_1 \Vdash_{\mathcal{O}} S(\alpha) :: S_1(K(\alpha))$, i.e., $FTV(S_1(\alpha)) \subseteq dom(K_1)$, and, since $Cls(K_1, S_1(\Gamma), \gamma) = (K_1', \sigma)$, that $K_1 = K_1' \cup \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$. But then we know that $FTV(S_1(\alpha)) \in dom(K_1') \cup dom(\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\})$. Hence, we only need to show that no element of $FTV(S_1(\alpha))$ is in $dom(\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}) = EFTV(K_1, \gamma) \setminus EFTV(K_1, S_1(\Gamma))$. Let $\alpha' \in FTV(S_1(\alpha))$. If $\alpha' \in EFTV(K_1, S_1(\Gamma))$, then $\alpha' \notin dom(\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\})$. If $\alpha' \notin EFTV(K_1, S_1(\Gamma))$, then it is easy to see that $\alpha' \notin EFTV(K_1, \gamma)$. Therefore, we have that (K_1', S_1) respects K and, by Lemma 2.4.1, that $(K_2, S_2 \circ S_1)$ respects K. Now, we have only left to show that $K_2, S_2 \circ S_1(\Gamma) \vdash \text{letEv } x = M_1 \text{ in } M_2 : \tau'$. By the induction hypothesis, we have that $K_1, S_1(\Gamma) \vdash M_1 : \gamma$ and that $K_2, S_2 \circ S_1(\Gamma) \cup \{x : S_2(\sigma)\} \vdash M_2 : \tau'$. Since $Cls(K_1, S_1(\Gamma), \gamma) = (K_1', \sigma)$, then, by (Gen), we have that $K_1, S_1(\Gamma) \vdash M_1 : \sigma$, and, by Lemma 2.4.3, that $K_2, S_2 \circ S_1(\Gamma) \vdash M_1 : S_2(\sigma)$. Finally, by (LetEv), we have that $K_2, S_2 \circ S_1(\Gamma) \vdash \text{letEv } x = M_1 \text{ in } M_2 : \tau'$.

We now prove the completeness of the type inference algorithm by induction on the structure of

M. Again, we only show the case for if M_1 then M_2 else M_3 and letEv $x = M_1$ in M_2. The case for c^b is trivial and the remaining cases are similar to the corresponding proof in [46].

- $M \equiv$ if M_1 then M_2 else M_3. Suppose that $\mathit{WE}(K, \Gamma, \text{if } M_1 \text{ then } M_2 \text{ else } M_3) = (K, S, \tau)$. Then $\mathit{WE}(K, \Gamma, M_1) = (K_1, S_1, \tau_1)$, $\mathcal{U}(K_1, \{(\tau_1, \mathit{Bool})\}) = (K_2, S_2)$, $\mathit{WE}(K_2, S_2 \circ S_1(\Gamma), M_2) = (K_3, S_3, \tau_2)$, $\mathit{WE}(K_3, S_3 \circ S_2 \circ S_1(\Gamma), M_3) = (K_4, S_4, \tau_3)$, $\mathcal{U}(K_4, \{(S_4(\tau_2), \tau_3)\}) = (K_5, S_5)$, and $K' = K_5$, $S = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1$, $\tau_0 = S_5 \circ S_4(\tau_2)$. Now, suppose that (K_0, S_0) respects K, and $K_0, S_0(\Gamma) \vdash_{\text{EVL}}$ if M_1 then M_2 else $M_3 : \tau_0$. Then $K_0, S_0(\Gamma) \vdash_{\text{EVL}} M_1 : \mathit{Bool}$, $K_0, S_0(\Gamma) \vdash_{\text{EVL}} M_2 : \tau_0$, and $K_0, S_0(\Gamma) \vdash_{\text{EVL}} M_3 : \tau_0$. By applying the induction hypothesis to M_2, we conclude that there exists some S_0^3 such that (K_0, S_0^3) respects K_3, and $\tau_0 = S_0^3(\Gamma)$ and $S_0(\Gamma) = S_0^3 \circ S_3 \circ S_2 \circ S_1$. Now, by applying the induction hypothesis to M_3, we conclude that there exists some S_0^4 such that (K_0, S_0^4) respects K_4, and $\tau_0 = S_0^4(\tau_3)$ and $S_0(\Gamma) = S_0^4 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma)$. It is easy to see that S_0^4 is a unifier of $S_4(\tau_2)$ and τ_3. By the correctness and completeness of the unification algorithm, there is a S_0^5 such that (K_0, S_0^5) respects K_5 and $S_0^4 = S_0^5 \circ S_5$. Then we have $\tau_0 = S_0^4(\tau_3) = S_0^5 \circ S_5 \circ \tau_3 = S_0^5 \circ S_5 \circ S_4 \circ \tau_2$, $S_0(\Gamma) = S_0^4 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma) = S_0^5 \circ S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1(\Gamma)$.

- $M \equiv$ letEv $x = M_1$ in M_2. Suppose that $\mathit{WK}(K, \Gamma, \text{letEv } x = M_1 \text{ in } M_2) = (K', S, \tau)$. Then $\mathit{WK}(K, \Gamma, M_1) = (K, S_1, \gamma)$, $\mathit{Cls}(K_1, S_1(\Gamma), \gamma) = (K_1', \sigma_1)$, $\mathit{WK}(K_1', S_1(\Gamma) \cup \{x : \sigma_1\}, M_2) = (K_2, S_2, \tau')$, and $K' = K_2$, $S = S_2 \circ S_1$, $\tau = \tau'$. Now, suppose that (K_0, S_0) respects K, and $K_0, S_0(\Gamma) \vdash \text{letEv } x = M_1 \text{ in } M_2 : \tau_0$. Then $K_0', S_0(\Gamma) \vdash M_1 : \gamma_0^1$, $\mathit{Cls}(K_0', S_0(\Gamma), \gamma_0^1) = (K_0, \sigma_0^1)$ and $K_0, S_0(\Gamma) \cup \{x : \sigma_0^1\} \vdash M_2 : \tau_0$. By the definition of Cls, we know we can write $K_0' = K_0 \cup \{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\}$ such that $\{\alpha_1^0, \ldots, \alpha_m^0\} = \mathit{EFTV}(K_0', \gamma_0^1) \setminus \mathit{EFTV}(K_0', S_0(\Gamma))$ and $\sigma_0^1 = \forall \alpha_1^0 :: \kappa_1^0 \cdots \forall \alpha_m^0 :: \kappa_m^0 . \gamma_0^1$. By the induction hypothesis, there is some S_0^1 such that (K_0', S_0^1) respects K_1, $\sigma_0^1 = S_0^1(\gamma)$ and $S_0(\Gamma) = S_0^1 \circ S_1(\Gamma)$. Again, by the definition of Cls, we know we can write $K_1 = K_1' \cup \{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ such that $\{\alpha_1, \ldots, \alpha_n\} = \mathit{EFTV}(K_1, \gamma) \setminus \mathit{EFTV}(K_1, S_1(\Gamma))$ and $\sigma_1 = \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n . \gamma$. Since we can always rename bound variables as needed, we can assume that $\{\alpha_1, \ldots, \alpha_n\} \cap \{\alpha_1^0, \ldots, \alpha_m^0\} = \emptyset$. Since (K_0', S_0^1) respects K_1, then $K_0 \cup \{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\} \vdash S_0^1(\alpha_i) :: S_0^1(\kappa_i), (1 \le i \le n)$. We can decompose S_0^1 into two substitutions S_0^2 and S_0^3 such that $S_0^1 = S_0^3 \circ S_0^2$ as follows: S_0^2 is the restriction of S_0^1 on $\mathit{dom}(S_0^1) \setminus \{\alpha_1, \ldots, \alpha_n\}$; S_0^3 is $[S_0^1(\alpha_1)/\alpha_1, \ldots, S_0^1(\alpha_n)/\alpha_n]$. Now, we show that, for each $1 \le n$, $S_0^2(\kappa_i)$ is well formed under $K_0 \cup \{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_{i-1} :: S_0^2(\kappa_{i-1})\}$. Since $S_0^1(\kappa_i)$ is well formed under K_0', it is enough to show that $\mathit{FTV}(S_0^2(\kappa_i)) \cap \{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0 = \emptyset$. Suppose $\alpha \in \mathit{FTV}(S_0^2(\kappa_i))$. Then there is some α' such that $\alpha \in \mathit{FTV}(S_0^2(\alpha'))$ and $\alpha' \in \mathit{FTV}(\kappa_i)$. Since $\alpha_i \in \mathit{EFTV}(K_1, \gamma)$, then $\alpha' \in \mathit{EFTV}(K_1, \gamma)$. By our assumption on K_1, for any $j \ge i$, $\alpha_i \notin \mathit{FTV}(\kappa_i)$. Therefore, either $\alpha \in \{\alpha_1, \ldots, \alpha_{i-1}\}$, or $\alpha \in S_0^2(\mathit{EFTV}(K_1, S_1(\Gamma)))$. But it can be shown by induction of the construction of $\mathit{EFTV}(K, S_1(\Gamma))$ that $S_0^2(\mathit{EFTV}(K_1, S_1(\Gamma)) \subseteq \mathit{EFTV}(K_0', S_0(\Gamma))$. Thus $\alpha \notin \{\alpha_1^0, \ldots, \alpha_m^0\}$, and $K_0 \cup \{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$ is well formed. Using a similar argument, it can be shown that $S_0^2(\gamma)$ is well formed under $K_0 \cup \{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$. Then $(K_0 \cup \{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\}, S_0^3)$ respects $K_0 \cup \{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$, and

$\gamma_0^1 = S_0^3(S_0^2(\gamma))$. Thus $K_0 \vdash \forall\alpha_1 :: S_0^2(\kappa_1) \cdots \forall\alpha_n :: S_0^2(\kappa_n).S_0^2(\gamma) \geq \forall\alpha_1^0 :: \kappa_1^0 \cdots \forall\alpha_m^0 :: \kappa_m^0.\gamma_0^1$, i.e. $K_0 \vdash S_0^2(\sigma_1) \geq \sigma_0^1$. By Lemma 2.4.4, $K_0, S_0(\Gamma) \cup \{x : S_0^2(\sigma_1)\} \vdash M_2 : \tau_0$. Since $S_0(\Gamma) = S_0^2(S_1(\Gamma))$, then $K_0, S_0^2(S_1(\Gamma) \cup \{x : \sigma_1\}) \vdash M_2 : \tau_0$. Since (K_0', S_0^1) respects K_1, then (K_0, S_0^2) respects K_1'. Finally, by applying the induction hypothesis to M_2, we can conclude that there is some S_0^4 such that (K_0, S_0^4) respects K_2 and that $\tau_0 = S_0^4(\tau')$ and $S_0^4(S_2(S_1(\Gamma))) = S_0^2(S_1)) = S_0^1(S_1(\Gamma)) = S_0(\Gamma)$.

□

Example 3.4.1. Let $M \equiv \{\text{location} = l, \text{fire\_danger} = d\}$, $N \equiv \text{FireDanger } \texttt{"Porto"}^{String}\ \texttt{"low"}^{String}$, $\tau_1 \equiv \{\text{location} : \alpha_3, \text{fire\_danger} : \alpha_4\}$, $\tau_1' \equiv \{\text{location} : String, \text{fire\_danger} : String\}$, $\tau_2 = \forall\alpha_3 :: \mathcal{U}.\forall\alpha_4 :: \mathcal{U}.\alpha_3 \to \alpha_4 \to \{\text{location} : \alpha_3, \text{fire\_danger} : \alpha_4\}$.

1)	$WE(\emptyset, \emptyset, \text{letEv FireDanger} = \lambda l.\lambda d.M \text{ in } N)$	=	$(\emptyset, [String/\alpha_6, String/\alpha_5, String/\alpha_3, String/\alpha_4, \alpha_4/\alpha_2, String/\alpha_1], \tau_1')$
1.1)	$WE(\emptyset, \emptyset, \lambda l.\lambda d.M)$	=	$(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2, \alpha_3/\alpha_1], \alpha_3 \to \alpha_4 \to \tau_1)$
1.1.1)	$WE(\{\alpha_1 :: \mathcal{U}\}, \{l : \alpha_1\}, \lambda d.M)$	=	$(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2, [\alpha_3/\alpha_1], \alpha_4 \to \tau_1)$
1.1.1.1)	$WE(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, M)$	=	$(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2, \alpha_3/\alpha_1], \tau_1)$
1.1.1.1.1)	$WE(\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \mathcal{U}\}, \{l : \alpha_1, d : \alpha_2\}, l)$	=	$(\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, [\alpha_3/\alpha_1], \alpha_3)$
1.1.1.1.2)	$WE(\{\alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}\}, \{l : \alpha_3, d : \alpha_2\}, d)$	=	$(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, [\alpha_4/\alpha_2], \alpha_4)$
1.2)	$Cls(\{\alpha_3 :: \mathcal{U}, \alpha_4 :: \mathcal{U}\}, \emptyset, \alpha_3 \to \alpha_4 \to \tau_1)$	=	$(\emptyset, \tau_2)$
1.3)	$WE(\emptyset, \{\text{FireDanger} : \tau_2\}, N)$	=	$(\emptyset, [String/\alpha_6, String/\alpha_5, String/\alpha_3, String/\alpha_4], \tau_1')$
1.3.1)	$WE(\emptyset, \{\text{FireDanger} : \tau_2\}, \text{FireDanger } \texttt{"Porto"}^{String})$	=	$(\emptyset, [String/\alpha_5, String/\alpha_3, \alpha_6/\alpha_4], \alpha_6 \to \{\text{location} : String, \text{fire\_danger} : \alpha_6\})$
1.3.1.1)	$WE(\emptyset, \{\text{FireDanger} : \tau_2\}, \text{FireDanger})$	=	$(\emptyset, [\alpha_5/\alpha_3, \alpha_6/\alpha_4], \alpha_5 \to \alpha_6 \to \{\text{location} : \alpha_5, \text{fire\_danger} : \alpha_6\})$
1.3.1.3)	$\mathcal{U}(\emptyset, (\alpha_5, String))$	=	$(\emptyset, [String/\alpha_5])$
1.3.3)	$\mathcal{U}(\emptyset, (\alpha_6, String))$	=	$(\emptyset, [String/\alpha_6])$

The principal typing of letEv FireDanger $= \lambda l.\lambda d.M$ in FireDanger $\texttt{"Porto"}^{String}\ \texttt{"low"}^{String}$ is $\{\text{location} : String, \text{fire\_danger} : String\}$.

For now, EVL does not include more powerful operations on records, such as those for extending a record with a new field or for removing an existing field from a record. These operations allow us to compose events in a more straightforward manner, which is a very important concept both in CEP and in the treatment of obligations. We will explain these kind of operations on records in the next chapter.

Chapter 4

An ML-style Calculus with Extensible Records

While the type inference algorithm introduced by Ohori as an extension of let-polymorphic system for ML [24] allows for a polymorphic treatment of record-based operations such as field selection and modification, it lacks support for extensible records. This is often accepted in practical implementations of languages with record types in exchange for efficiency or due to the difficulties in guaranteeing the correctness of type-inference whenever these operations are considered.

In this chapter we add extensible records (i.e., records that can have new fields added to them, or preexisting fields removed from them) to Ohori's original ML-style polymorphic record calculus by refining the notion of a record kind and type.

4.1 Terms

Definition 4.1.1. Let x range over an infinite countable set of variables $\mathbb{V}$ and l range over an infinite countable set of labels $\mathbb{L}$. The set of λ-terms, denoted by Λ_χ, is given by the following grammar:

$$
\begin{array}{rcll}
M & ::= & c^b & \text{(constants)} \\
 & | & x & \text{(variable)} \\
 & | & (MM) & \text{(function application)} \\
 & | & \lambda x.M & \text{(functional abstraction)} \\
 & | & \text{let } x = M \text{ in } M & \text{(let declaration)} \\
 & | & \{l = M, \ldots, l = M\} & \text{(records)} \\
 & | & M.l & \text{(field selection)} \\
 & | & \text{modify}(M, l, M) & \text{(field update)} \\
 & | & M \setminus\setminus l & \text{(field removal)} \\
 & | & \text{extend}(M, l, M) & \text{(field addition)}
\end{array}
$$

where b is a base type from a set of base type $\mathbb{B}$.

4.2 Types and Kinds

Definition 4.2.1. Let α range over an infinite countable set of type variables $\mathbb{A}$, l range over an infinite countable set of labels $\mathbb{L}$ and b range over a set of base types $\mathbb{B}$. The set of types, denoted by $\mathbb{T}_\chi$, and the set of kinds, denoted by $\mathbb{K}_\chi$, are given by the following grammar:

$$
\begin{array}{rcll}
\tau & ::= & b & \text{(base types)} \\
 & | & \chi & \text{(extensible types)} \\
 & | & \tau \to \tau & \text{(function types)} \\
 & & & \\
\sigma & ::= & \tau & \text{(monomorphic types)} \\
 & | & \forall \alpha :: \kappa . \sigma & \text{(polymorphic types)} \\
 & & & \\
\chi & ::= & \alpha & \text{(type variables)} \\
 & | & \{l : \tau, \ldots, l : \tau\} & \text{(record types)} \\
 & | & \chi + \{l : \tau\} & \text{(field-addition types)} \\
 & | & \chi - \{l : \tau\} & \text{(field-removal types)} \\
 & & & \\
\kappa & ::= & \mathcal{U} & \text{(universal kind)} \\
 & | & \{\{l : \tau, \ldots, l : \tau \parallel l : \tau, \ldots, l : \tau\}\} & \text{(record kinds)}
\end{array}
$$

where the universal kind represents the set of all types and a record kind of the form $\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$ represents the set of structures that have at least fields $l_1^l, \ldots, l_n^l$ with types $\tau_1^l, \ldots, \tau_n^l$, respectively, and do not have at least the fields $l_1^r, \ldots, l_m^r$ with types $\tau_1^r, \ldots, \tau_m^r$, respectively.

We will call a type that is either a field-addition or field-removal type, a ***field-alteration type***.

Note that extensible types are defined recursively but are *not* recursive types. As "base cases", extensible types may have a type variable or a record type. We say that an extensible type χ has τ as its ***root-type***, denoted by $root(\chi)$, if χ was constructed starting with τ.

We assume that the labels that appear in any type or kind are always pairwise distinct and that each label has exactly one type.

In this calculus, we will allow records to be empty and empty records are always assigned the empty record type $\{\}$. An important distinction to make right away is the following. A type variable that has the universal kind $\mathcal{U}$ can be instantiated with any type, but a type variable that has the empty record kind $\{\{\parallel\}\}$ can only be instantiated with a record type.

The set of free type-variables provided in Definition 2.4.3 has to be extended to cover extensible records and the new type of kinds.

Definition 4.2.2. The set of ***free type-variables*** of a type $\sigma \in \mathbb{T}_\chi$, denoted by $FTV(\sigma)$, is defined inductively as follows:

$$\begin{aligned}
FTV(\alpha) &= \{\alpha\} \\
FTV(\tau \to \tau') &= FTV(\tau) \cup FTV(\tau') \\
FTV(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) &= FTV(\tau_1) \cup \cdots \cup FTV(\tau_n) \\
FTV(\forall \alpha :: \kappa.\sigma) &= FTV(\kappa) \cup (FTV(\sigma) \setminus \{\alpha\}) \\
FTV(\chi + \{l : \tau\}) &= FTV(\chi) \cup FTV(\tau) \\
FTV(\chi - \{l : \tau\}) &= FTV(\chi) \cup FTV(\tau)
\end{aligned}$$

The set of ***free type-variables*** of a kind $\kappa \in \mathbb{K}$, denoted $FTV(\kappa)$, is defined inductively as follows:

$$\begin{aligned}
FTV(\mathcal{U}) &= \emptyset \\
FTV(\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}) &= FTV(\tau_1^l) \cup \cdots \cup FTV(\tau_n^l) \\
&\quad \cup FTV(\tau_1^r) \cup \cdots \cup FTV(\tau_m^r)
\end{aligned}$$

We need to extend the subtype relation to extensible types.

Definition 4.2.3. We say that $\sigma \in \mathbb{T}_\chi$ is a ***subtype*** of $\sigma' \in \mathbb{T}_\chi$, denoted by $\sigma \sqsubseteq_{type} \sigma'$, if $\sigma \in Subtype(\sigma')$, where $Subtype(\sigma')$, the collection of subtypes of σ', is defined inductively as follows:

$$\begin{aligned}
Subtype(\alpha) &= \{\alpha\} \\
Subtype(\tau \to \tau') &= Subtype(\tau) \cup Subtype(\tau') \cup \{\tau \to \tau'\} \\
Subtype(\forall \alpha.\sigma) &= Subtype(\sigma) \cup \{\forall \alpha.\sigma\} \\
Subtype(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) &= Subtype(\tau_1) \cup \ldots \cup Subtype(\tau_n) \cup \{\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\} \\
Subtype(\chi + \{l : \tau\}) &= Subtype(\chi) \cup Subtype(\tau) \cup \{\chi + \{l : \tau\}\} \\
Subtype(\chi - \{l : \tau\}) &= Subtype(\chi) \cup Subtype(\tau) \cup \{\chi - \{l : \tau\}\}
\end{aligned}$$

Kinding environments are defined the same way as in Section 2.4, just like any well-formedness criteria applicable to both typing and kinding environments.

Definition 4.2.4. A type $\tau \in \mathbb{T}_\chi$ has a kind $\kappa \in \mathbb{K}_\chi$ under a kinding assignment K, denoted by

$K \Vdash_\chi \tau :: \kappa$, if it is derivable by the following set of ***kinding rules***:

$$
\begin{aligned}
&K \Vdash_\chi \tau :: \mathcal{U} \text{ for any } \tau \text{ well-formed under } K \\
&K \Vdash_\chi \{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n, \ldots\} :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m\}\} \\
&\qquad \text{if } \{l^l_1, \ldots, l^l_n, \ldots\} \cap \{l^r_1, \ldots, l^r_m\} = \emptyset, \\
&\qquad \text{and both } \{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n, \ldots\} \text{ and } \tau^r_i \ (1 \leq i \leq m) \text{ are well-formed under } K \\
&K \Vdash_\chi \alpha :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m\}\} \\
&\qquad \text{if } K(\alpha) = \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n, \ldots \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m, \ldots\}\} \\
&K \Vdash_\chi \chi + \{l : \tau\} :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n, [l : \tau] \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m\}\} \\
&\qquad \text{if } K \Vdash_\chi \chi :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m, l : \tau\}\} \\
&K \Vdash_\chi \chi - \{l : \tau\} :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m, [l : \tau]\}\} \\
&\qquad \text{if } K \Vdash_\chi \chi :: \{\{l^l_1 : \tau^l_1, \ldots, l^l_n : \tau^l_n, l : \tau \parallel l^r_1 : \tau^r_1, \ldots, l^r_m : \tau^r_m\}\}
\end{aligned}
$$

where $[l : \tau]$ means that the inclusion of the field $l : \tau$ in its respective kind is optional.

Example 4.2.1. Let $\tau_1, \tau_2 \in \mathbb{T}_\chi$ and $l_1, l_2 \in \mathbb{L}$:

$$
\begin{aligned}
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} :: \mathcal{U} \\
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} :: \{\{\parallel\}\} \\
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} :: \{\{l_1 : \tau_1 \parallel\}\} \\
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} + \{l_2 : \tau_2\} :: \{\{l_2 : \tau_2 \parallel\}\} \\
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} + \{l_2 : \tau_2\} :: \{\{\parallel l_3 : \tau_3\}\} \\
&\emptyset \Vdash_\chi \{l_1 : \tau_1\} + \{l_2 : \tau_2\} - \{l_1 : \tau_1\} :: \{\{l_2 : \tau_2 \parallel l_1 : \tau_1\}\}
\end{aligned}
$$

The following proposition ensures us that the kind restrictions that are produced by the kinding rules are ***consistent***.

Proposition 4.2.1. The kinding rules ensure the two following properties:

1. If $K \Vdash_\chi \tau :: \{\{\ldots, l : \tau', \ldots \parallel \ldots\}\}$, then $K \not\Vdash_\chi \tau :: \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$;
2. If $K \Vdash_\chi \tau :: \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, then $K \not\Vdash_\chi \tau :: \{\{\ldots, l : \tau', \ldots \parallel \ldots\}\}$.

Proof. Let us assume that $K \Vdash_\chi \tau :: \{\{\ldots, l : \tau', \ldots \parallel \ldots\}\}$. According to the kinding rules, τ is either a record type, a type variable, or an extensible type. Since extensible types have either record types of type variables as root-types, we will first prove that these properties hold for record types and type variables and then prove that these properties hold for extensible types as well. The proof follows by induction on the structure of τ:

Property 1. First let us show that this property holds when τ is either a record type or a type variable:

- If $\tau \equiv \{\ldots, l : \tau', \ldots\}$, then $K \not\Vdash_\chi \{\ldots, l : \tau', \ldots\} :: \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, since $\{\ldots, l, \ldots\} \cap \{\ldots, l, \ldots\} \neq \emptyset$;
- If $\tau \equiv \alpha$, then $K(\alpha) = \{\{\ldots, l : \tau', \ldots \parallel \ldots\}\}$ and $K \not\vdash_\chi \alpha :: \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, since $K(\alpha)$ cannot be of the form $\{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, because l can only appear once in $K(\alpha)$;

Now let us show that this property also holds when τ is an extensible type, assuming that it holds for χ:

- If $\tau \equiv \chi + \{l : \tau'\}$, then $K \Vdash_\chi \chi :: \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$ and $K \not\Vdash_\chi \chi + \{l : \tau'\} :: \{\{\ldots \mid \mid \ldots, l : \tau', \ldots\}\}$, since, by the induction hypothesis, $K \not\Vdash_\chi \chi : \{\{\ldots, l : \tau', \ldots \parallel \ldots\}\}$;
- If $\tau \equiv \chi + \{l' : \tau''\}$, such that $l \neq l'$, then $K \Vdash_\chi \chi :: \{\{\ldots, l : \tau', \ldots \parallel \ldots, l' : \tau'', \ldots\}\}$ and $K \not\Vdash_\chi \chi + \{l' : \tau''\} : \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, since, by the induction hypothesis, $K \not\Vdash_\chi \chi : \{\{\ldots \parallel \ldots, l : \tau', l' : \tau'', \ldots\}\}$;
- If $\tau \equiv \chi - \{l' : \tau''\}$, such that $l \neq l'$, then $K \Vdash_\chi \chi :: \{\{\ldots, l : \tau', \ldots, l' : \tau'', \ldots \parallel \ldots\}\}$ and $K \not\Vdash_\chi \chi - \{l' : \tau''\} : \{\{\ldots \parallel \ldots, l : \tau', \ldots\}\}$, since, by the induction hypothesis, $K \not\Vdash_\chi \chi : \{\{\ldots, l' : \tau'', \ldots \parallel \ldots, l : \tau', \ldots\}\}$.

Property 2. The proof is very similar to the one for Property 1.

□

We now restate and prove the following lemmas for this calculus.

Lemma 4.2.1. If $K \Vdash_\chi \tau :: \kappa$ and (K_1, S) respects K, then $K_1 \Vdash_\chi S(\tau) :: S(\kappa)$.

Proof. The proof follows by induction on the structure of τ:

- Let τ be well-formed under K and $\kappa \equiv \mathcal{U}$. Then, by Definition 2.4.9, this means that $FTV(\tau) \subseteq dom(K)$. Since (K_1, S) respects K, we know that for all $\alpha \in FTV(\tau)$, $K_1 \Vdash_\chi S(\alpha) :: S(K(\alpha))$, i.e., that for all $\alpha \in FTV(\tau)$, $S(\alpha)$ is well-formed under K_1. By Definition 2.4.9, this means that $S(\tau)$ is also well-formed under K_1. Finally, by Definition 4.2.4, we can now conclude that $K_1 \Vdash_\chi S(\tau) :: \mathcal{U}$. Note that $S(\mathcal{U}) = \mathcal{U}$.
- Let $\tau \equiv \{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots\}$ and $\kappa \equiv \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$. By the induction hypothesis, we know that for all $\tau' \in \{\tau_1^l, \ldots, \tau_n^l, \ldots\}$, such that $K \Vdash_\chi \tau' :: \kappa$, $K_1 \Vdash_\chi S(\tau') :: S(\kappa)$, i.e., that for all $\tau' \in \{\tau_1^l, \ldots, \tau_n^l, \ldots\}$, $S(\tau')$ is well-formed under K_1. This means that $S(\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots\})$ is also well-formed under K_1. Since we know that $\tau_i^r, (1 \leq i \leq m)$ are well-formed under K, we know, by Definition 2.4.9, that $FTV(\tau_i^r) \subseteq dom(K)$. Since (K_1, S) respects K, we know that for all $\alpha \in FTV(\tau_i^r)$, $K_1 \Vdash_\chi S(\alpha) :: S(K(\alpha))$, i.e., that for all $\alpha \in FTV(\tau_i^r)$, $S(\alpha)$ is well-formed under K_1. Finally, by Definition 4.2.4, we can now conclude that $K_1 \Vdash_\chi S(\{\tau_1^l, \ldots, \tau_n^l, \ldots\}) :: S(\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\})$. Note that labels are not affected by substitutions.

- Let $\tau \equiv \alpha$ and $\kappa \equiv \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$. We know, by Definition 4.2.4, that $K(\alpha) = \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, \ldots\}\}$. Since (K_1, S) respects K (and $\alpha \in dom(K)$), we can conclude, by Definition 2.4.13, that $K_1 \Vdash_\chi S(\alpha) :: S(\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, \ldots \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, \ldots\}\})$.
- Let $\tau \equiv \chi + \{l : \tau'\}$ and $\kappa \equiv \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, [l : \tau'] \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\}$. By the induction hypothesis, we know that $K_1 \Vdash_\chi S(\chi) :: S(\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, l : \tau'\}\})$. Therefore $K_1 \Vdash_\chi S(\chi + \{l : \tau'\}) :: S(\{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l, [l : \tau'] \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r\}\})$;
- Let $\tau \equiv \chi - \{l : \tau'\}$ and $\kappa \equiv \{\{l_1^l : \tau_1^l, \ldots, l_n^l : \tau_n^l \parallel l_1^r : \tau_1^r, \ldots, l_m^r : \tau_m^r, [l : \tau']\}\}$. The proof for this case is very similar to the one for the previous case.

□

Example 4.2.2. Let $\chi, \tau, \tau' \in \mathbb{T}_\chi$ and $l_1, l_2 \in \mathbb{L}$. The two following types are equal because both represent the set of terms that add a field labelled with l_1 to χ and remove a field labelled with l_2 from χ:

$$\chi + \{l_1 : \tau\} - \{l_2 : \tau'\} \equiv \chi - \{l_2 : \tau'\} + \{l_1 : \tau\}.$$

And the two following types are also equal because they represent the set of terms that add, and then remove, a field labelled with l_1 to and from χ and also remove a field labelled with l_2 from χ:

$$\chi + \{l_1 : \tau\} - \{l_2 : \tau'\} - \{l_1 : \tau\} \equiv \chi + \{l_1 : \tau\} - \{l_1 : \tau\} - \{l_2 : \tau'\}$$

Now, let $\chi_1, \chi_2, \tau, \tau' \in \mathbb{T}_\chi$ such that $root(\chi_1) \not\equiv root(\chi_2)$, and $l_1, l_2 \in \mathbb{L}$. The two following types are not equal because they have different root types:

$$\chi_1 + \{l_1 : \tau\} - \{l_2 : \tau'\} \not\equiv \chi_2 + \{l_1 : \tau\} - \{l_2 : \tau'\}$$

And the two following types are not equal because the one on the left represents the set of terms that first remove a field labelled with l_1 while the one on the right represents the set of terms that first add a field labelled with l_1 from χ_1:

$$\chi_1 - \{l_1 : \tau\} + \{l_2 : \tau'\} + \{l_1 : \tau\} \not\equiv \chi_1 + \{l_1 : \tau\} - \{l_1 : \tau\} + \{l_2 : \tau'\}$$

The reason why we do not consider these last two types equal lays in the fact that χ_1 must either represent the set of terms that contain a field labelled with l_1, or not. One of them must not be well-formed, since the one on the left tells us that a record typed with χ_1 must have a field labelled with l_1, but the type on the right tells us the opposite, i.e., that a record typed with χ_1 must *not* have a field labelled with l_1.

Note that the fact that we cannot change the order of fields-alteration types that have the same label is naturally enforced by the ***kinding rules***.

Definition 4.2.5. We say that two extensible types $\chi, \chi' \in \mathbb{T}_\chi$ are equivalent up to the ordering of their sequences of field-addition or removal types, denoted $\equiv_{ord}$, if the two following conditions are met:

- Their root types are equal.
- Their sequences of field-alteration types only differ on the order of field-alteration types with different labels. In this case, we say that χ and χ' have ***equivalent sequences*** of field-alteration types.

This definition can be trivially extended to any two types $\sigma, \sigma' \in \mathbb{T}_\chi$, in which case we will also write $\sigma \equiv_{ord} \sigma'$.

The definitions provided in Section 2.4 related with type-substitutions, essentially-free type variables, generic instance and closure are also valid for this calculus.

4.2.1 χ-reduction

Definition 4.2.5 allows us to identify extensible types with the same root type built from equivalent sequences of field-alteration types, i.e., to identify extensible types that represent sets of records that are *built* the same way. To identify sets of records that have *the same set of fields*, we will also need to identify extensible types with the same root type and different sequences of field-alteration types that still end up representing sets of records that have the same set of fields.

Example 4.2.3. Let $\chi, \tau, \tau' \in \mathbb{T}_\chi$, $l_1, l_2, l_3 \in \mathbb{L}$ and K be a kinding environment such that $K \Vdash_\chi \chi :: \{\{l_1 : \tau \parallel l_2 : \tau'\}\}$. The two following types are equal because they both represent the set of records that have (at least) a field labelled with l_1 and do not have (at least) a field labelled with l_2:

$$\chi - \{l_1 : \tau\} + \{l_1 : \tau\} \equiv \chi + \{l_2 : \tau'\} - \{l_2 : \tau'\}.$$

We now introduce a reduction system for extensible types that, along with the restriction imposed by the kinding rules, will allow us to identify extensible types that represent sets of records that have the same set of fields.

Definition 4.2.6. Let $\chi, \chi' \in \mathbb{T}_\chi$.

- We say that χ_1 reduces to χ in one ***χ-reduction*** step, denoted by $\chi \to^1_\chi \chi'$, if χ' can be

obtained from χ using one of the following reduction rules:

$$\{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} - \{l_i : \tau_i\} \pm \{l' : \tau'\} \cdots \rightarrow^1_\chi \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \pm \{l' : \tau'\} \cdots$$

$$\{l_1 : \tau_1, \ldots, l_n : \tau_n\} + \{l : \tau\} \pm \{l' : \tau'\} \cdots \rightarrow^1_\chi \{l_1 : \tau_1, \ldots, l_n : \tau_n, l : \tau\} \pm \{l' : \tau'\} \cdots$$

$$\begin{aligned}&\alpha \pm_1 \{l_1 : \tau_1\} \cdots -_i \{l : \tau\} \cdots +_j \{l : \tau\} \cdots \\ &\rightarrow^1_\chi \alpha \pm_1 \cdots \pm_{i-1} \{l_{i-1} : \tau_{i-1}\} \pm_{i+1} \{l_{i+1} : \tau_{i+1}\} \cdots \pm_{j-1} \{l_{j-1} : \tau_{j-1}\} \pm_{j+1} \{l_{j+1} : \tau_{j+1}\} \cdots\end{aligned}$$

$$\begin{aligned}&\alpha \pm_1 \{l_1 : \tau_1\} \cdots +_i \{l : \tau\} \cdots -_j \{l : \tau\} \cdots \\ &\rightarrow^1_\chi \alpha \pm_1 \cdots \pm_{i-1} \{l_{i-1} : \tau_{i-1}\} \pm_{i+1} \{l_{i+1} : \tau_{i+1}\} \cdots \pm_{j-1} \{l_{j-1} : \tau_{j-1}\} \pm_{j+1} \{l_{j+1} : \tau_{j+1}\} \cdots\end{aligned}$$

- We define $\rightarrow_\chi$ as the reflexive and transitive closure of $\rightarrow^1_\chi$.

To ensure that $\rightarrow_\chi$ is confluent, i.e., that if $\chi_1 \leftarrow_\chi \chi \rightarrow_\chi \chi_2$ then there is a χ' such that $\chi_1 \rightarrow_\chi \chi' \leftarrow_\chi \chi_2$, we are going to assume that i and j always correspond to the positions of the *first two occurrences* of those opposite and same-labelled fields in χ_1.

Proposition 4.2.2. $\rightarrow_\chi$ is convergent, i.e., it is both confluent and terminating.

Proof. It is obvious that $\rightarrow_\chi$ is terminating since the number of field-alteration types is reduced after each reduction step. Also, note that $\rightarrow_\chi$ is confluent by construction. □

Proposition 4.2.3. Every rewrite rule $\rightarrow_\chi$, transforms a type $\chi \in \mathbb{T}_\chi$ into a $\chi' \in \mathbb{T}_\chi$, such that $K \vdash_\chi \chi :: \kappa$ if, and only if, $K \vdash_\chi \chi' :: \kappa$.

Proof. Let $root(\chi)$ be a record type:

- If $\chi \equiv \{l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\} - \{l_i : \tau_i\} \pm \{l : \tau\} \cdots$ and $\chi' \equiv \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \pm \{l : \tau\} \cdots$. The only field that is affected by the transformation is $l_i : \tau_i$. This means we only have to show that $K \Vdash_\chi \chi :: \{\{l_i : \tau_i \parallel\}\}$, if, and only if, $K \Vdash_\chi \chi' :: \{\{l_i : \tau_i \parallel\}\}$, and that, $K \Vdash_\chi \chi :: \{\{\parallel l_i : \tau_i\}\}$, if, and only if, $K \Vdash_\chi \chi' :: \{\{\parallel l_i : \tau_i\}\}$. The proofs for each direction of the implication are symmetric, so we are only going to provide the proof for one of the directions. Let us start by assuming that $K \Vdash_\chi \chi :: \{\{l_i : \tau_i \parallel\}\}$. According to Definition 4.2.4, the last occurrence of the field $l_i : \tau_i$ in χ must be a field-alteration type. But, since the transformation does not remove any field-alteration type from χ, the last occurrence of $l_i : \tau_i$ in χ' must also be a field-alteration type, and $K \Vdash_\chi \chi' :: \{\{l_i : \tau_i \parallel\}\}$. Now, let us assume that $K \vdash_\chi \chi :: \{\{\parallel l_i : \tau_i\}\}$. According to Definition 4.2.4, this means that the last occurrence of the field $l_i : \tau_i$ in χ must be a field-removal type. If the field-removal type removed by the transformation is not the last one, then the last occurrence of the field $l_i : \tau_i$ in χ' is also a field-alteration type, and $K \vdash_\chi \chi' :: \{\{\parallel l_i : \tau_i\}\}$. If, on the other hand, the field-removal type removed by the transformation is the last one, then it is easy to see that it contains the only occurrence of that label in χ. After the transformation, that field will no longer appear in χ'. By Definition 4.2.4, this means that $K \Vdash_\chi \chi' :: \{\{\parallel l_i : \tau_i\}\}$.

- The proof for $\{l_1 : \tau_1, \ldots, l_n : \tau_n\} + \{l : \tau\} \pm \{l' : \tau'\} \cdots$ is very similar to the previous one.

Now, let $root(\chi)$ be a type variable, $l : \tau$ be the first field appearing more than once containing label l in χ, and i, j $(1 \leq i \leq j)$ correspond to the positions of the two first occurrences of $l : \tau$ in χ:

- If $\chi \equiv \alpha \pm_1 \cdots -_i \{l : \tau\} \pm_{i+1} \cdots +_j \{l : \tau\} \pm_{j+1} \cdots$ and χ' is of the form $\alpha \pm_1 \cdots \pm_{i+1} \cdots \pm_{j+1} \cdots$. The only field that is affected by the transformation is $l : \tau$. This means that we only have to show that $K \Vdash_\chi \chi :: \{\{l : \tau \parallel\}\}$, if, and only if, $K \Vdash_\chi \chi' :: \{\{l : \tau \parallel\}\}$, and that $K \Vdash_\chi \chi :: \{\{\parallel l : \tau\}\}$, if, and only if, $K \vdash_\chi \chi' :: \{\{\parallel l : \tau\}\}$. The proofs for each direction of the implication are symmetric, so we are only going to provide the proof for the former direction. Let us start by assuming that $K \Vdash_\chi \chi :: \{\{l : \tau \parallel\}\}$. According to Definition 4.2.4, the last occurrence of the field $l : \tau$ in χ must be a field-addition type. If the field-addition type removed by the transformation is not the last one, then the last occurrence of $l : \tau$ in χ' is also a field-addition type, and $K \Vdash_\chi \chi' :: \{\{l : \tau \parallel\}\}$. If, on the other hand, the field-addition type removed by the transformation is the last one, then it is easy to see that i and j correspond to the positions of the only two occurrences of that field in χ. After the transformation, that field will no longer appear in χ'. If we want to show that $K \Vdash_\chi \alpha' :: \{\{l : \tau \parallel\}\}$, then we will have to show that $\{l : \tau\}$ is guaranteed to appear in $K(\alpha)$. But, by Definition 4.2.4, we know that the field-removal type in position i can only appear in χ if that is the case. Now, let us assume that $K \Vdash_\chi \chi :: \{\{\parallel l : \tau\}\}$. According to Definition 4.2.4, the last occurrence of the field $l : \tau$ in χ must be a field-removal type. This means that we only have to consider the case where the field-removal type removed by the transformation is not the last occurrence of field $l : \tau$ in χ. Since no other field-removal type is remove by the transformation, this means that the the last occurrence of $l : \tau$ in χ' is also a field-removal type and $K \Vdash_\chi \chi' :: \{\{l : \tau \parallel\}\}$.
- The proof for $\alpha \pm_1 \{l_1 : \tau_1\} \cdots +_i \{l : \tau\} \cdots -_j \{l : \tau\} \cdots$ is very similar to the previous one.

□

Definition 4.2.7. Let $\sigma \in \mathbb{T}_\chi$.

- σ is a χ-normal form (or is *in* χ-normal form) if for every extensible type $\chi \sqsubseteq_{type} \sigma$, there is no $\chi' \in \mathbb{T}_\chi$ such that $\chi \to_\chi \chi'$.
- We say that σ has a χ-normal form if there exists a $\sigma' \in \chi$ such that $\sigma \equiv_{ord} \sigma'$ and σ' is in χ-normal form.
- We will write $|\sigma|$ for the χ-normal form of σ.

Proposition 4.2.4. If χ is in normal form, then each label that appears in it occurs exactly once.

Proof. Assume $root(\chi)$ is a record type. If no more rules can be applied, then χ does not have any field-alteration types. This means that χ is a record type and, therefore, that every label that appears in χ occurs exactly once. Now, assume $root(\chi)$ is a type variable. If no more rules can be applied, then this means that there are no fields with matching labels appearing in χ. That is, every label that appears in χ occurs exactly once. □

This means that if a type $\sigma \in \mathbb{T}_\chi$ is in χ-normal form, then every extensible type $\chi \sqsubseteq_{type} \sigma$ has a type variable as its root-type and every field-addition or removal in χ has a unique label.

Definition 4.2.8. We say that two types $\sigma, \sigma' \in \mathbb{T}_\chi$ are χ-equivalent (i.e. represent the same sets of records with the same set of fields), denoted $\sigma \equiv_\chi \sigma'$ if $|\sigma| \equiv_{ord} |\sigma'|$.

Proposition 4.2.5. Let S be a substitution and τ be a type. Then $|S(|\tau|)| = |S(\tau)|$.

Proof. Let τ be in χ-normal form, i.e., $|\tau| \equiv_\chi \tau$. Then there exists an extensible type χ in τ that can be χ-normalized to its normal form $|\chi|$. Let $\alpha \in dom(S)$ appear in χ, but not in $|\chi|$. Then α only appears in field-alteration types of χ that were removed by χ-normalization. Clearly, $S(\alpha)$ will appear in $S(\chi)$, but not in $S(|\chi|)$. But then, since α only appeared in field-alteration types of χ that were removed by χ-normalization, $S(\alpha)$ will only appear in those same filed-alterations types. Therefore, $S(\alpha)$ will not appear in $|S(\chi)|$. Thus $S(\alpha)$ will neither appear in $|S(|\chi|)|$ nor in $|S(\chi)|$. Let $\alpha \in dom(S)$ appear in χ and in $|\chi|$. Then α must not appear in any field-alterations types χ that can be removed by χ-normalization. Then $S(\alpha)$ will appear both in $S(\chi)$ and $S(|\chi|)$:

- If $S(|\chi|)$ is in χ-normal form, then $S(\alpha)$ is in χ-normal form and will appear in both $|S(\chi)|$ and $|S(|\chi|)|$;
- If $S(|\chi|)$ is not in χ-normal form, then either $S(\alpha)$ is not in χ-normal form, $S(\alpha)$ is in χ-normal form, but field-alteration types that can be cancelled out by preexisting "opposite" field-alteration types in $|\chi|$ were added to $|\chi|$, or both. In any case, these field-alteration types will not appear in $|S(|\chi|)|$ or $|S(\chi)|$.

This means we can conclude that field-alteration types that appear in $|S(|\chi|)|$ and $|S(\chi)|$ are precisely the same, *i.e.* $|S(|\chi|)| = |S(\chi)|$. □

In this type-system we will consider extensionally equivalent type-substitutions modulo χ-equivalence.

Definition 4.2.9. We say that two type-substitutions S_1 and S_2 are ***extensionally equivalent***, denoted by $S_1 = S_2$, if and only if $S_1(\tau) \equiv_\chi S_2(\tau)$ for all $\tau \in \mathbb{T}_\chi$.

Proposition 4.2.6. Let S_1, S_2, S_1', and S_2' be substitutions, such that $S_1 = S_2$ and $S_1' = S_2'$. Then $S_1 \circ S_1' = S_2 \circ S_2'$.

$$\frac{\Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_\chi c^b : b} \text{ (Const)}$$

$$\frac{K \Vdash_\chi \Gamma(x) \geq \tau \qquad \Gamma \text{ is well-formed under } K}{K, \Gamma \vdash_\chi x : \tau} \text{ (Var)}$$

$$\frac{K, \Gamma\{x : \tau\} \vdash_\chi M : \tau'}{K, \Gamma \vdash_\chi \lambda x.M : \tau \to \tau'} \text{ (Abs)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \to \tau' \qquad K, \Gamma \vdash_\chi N : \tau}{K, \Gamma \vdash_\chi M\ N : \tau'} \text{ (App)}$$

$$\frac{K, \Gamma \vdash_\chi M : \sigma \qquad K, \Gamma\{x : \sigma\} \Vdash_\chi N : \tau}{K, \Gamma \vdash_\chi \text{let } x = M \text{ in } N : \tau} \text{ (Let)}$$

$$\frac{K, \Gamma \vdash_\chi M_i : \tau_i \ (1 \leq i \leq n)}{K, \Gamma \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{ (Rec)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \qquad K \Vdash_\chi \tau :: \{\{l : \tau' \parallel\}\}}{K, \Gamma \vdash_\chi M.l : \tau'} \text{ (Sel)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \qquad K, \Gamma \vdash_\chi N : \tau' \qquad K \Vdash_\chi \tau :: \{\{l : \tau' \parallel\}\}}{K, \Gamma \vdash_\chi \text{modify}(M, l, N) : \tau} \text{ (Modif)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \qquad \mathit{Cls}(K, \Gamma, \tau) = (K', \sigma)}{K', \Gamma \vdash_\chi M : \sigma} \text{ (Gen)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \qquad K \Vdash_\chi \tau :: \{\{l : \tau' \parallel\}\}}{K, \Gamma \vdash_\chi M \setminus\setminus l : \tau - \{l : \tau'\}} \text{ (Contr)}$$

$$\frac{K, \Gamma \vdash_\chi M : \tau \qquad K, \Gamma \vdash_\chi N : \tau' \qquad K \Vdash_\chi \tau :: \{\{\parallel l : \tau'\}\} \qquad \mathit{root}(\tau) \notin \mathit{FTV}(\tau')}{K, \Gamma \vdash_\chi \text{extend}(M, l, N) : \tau + \{l : \tau'\}} \text{ (Ext)}$$

Figure 4.1: Typing rules for the ML-style Calculus with Extensible Records.

Proof. Since $S_1 = S_2$ and $S_1' = S_2'$, we know that $S_1(\tau) \equiv_\chi S_2(\tau)$ and $S_1'(\tau) \equiv_\chi S_2'(\tau)$ for any $\tau \in \mathbb{T}_\chi$. But $|S_1'(\tau)| \equiv_\chi |S_2'(\tau)|$ and $|S_1'(\tau)|, |S_2'(\tau)| \in \mathbb{T}_\chi$. Therefore, $S_1(|S_1'(\tau)|) \equiv_\chi S_2(|S_2'(\tau)|)$ and, by Proposition 4.2.5, $S_1(S_1'(\tau)) \equiv_\chi S_2(S_2'(\tau))$. Thus, $S_1 \circ S_1' = S_2 \circ S_2'$. □

Definition 4.2.10. We say that $M \in \Lambda_\chi$ can be assigned the type $\tau \in \mathbb{T}_\chi$ according to the type assignment Γ, if the statement $M : \tau$ can be derived from Γ using the typing rules in Figure 4.1.

Example 4.2.4. Let $K = \{\alpha_1 :: \{\{\parallel l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}$ and $\Gamma = \{x : \alpha_1, y : \alpha_2\}$. We can deduce $\text{extend}(x, l, y).l : \alpha_2$ from Γ, K as follows:

$$\Delta_1 = \frac{K \Vdash_\chi \alpha_1 \geq \alpha_1}{K, \Gamma \vdash_\chi x : \alpha_1}\ (\text{Var})$$

$$\Delta_2 = \frac{K \Vdash_\chi \alpha_2 \geq \alpha_2}{K, \Gamma \vdash_\chi y : \alpha_2}\ (\text{Var})$$

$$\Delta_3 = \frac{\Delta_1 \qquad \Delta_2 \qquad K \Vdash_\chi \alpha_1 :: \{\{\| \, l : \alpha_2\}\} \qquad \alpha_1 \notin FTV(\alpha_2)}{K, \Gamma \vdash_\chi \text{extend}(x, l, y) : \alpha_1 + \{l : \alpha_2\}}\ (\text{Ext})$$

$$\frac{\Delta_3 \qquad K \Vdash_\chi \alpha_1 + \{l : \alpha_2\} :: \{\{l : \alpha_2 \, \|\}\}}{K, \Gamma \vdash_\chi \text{extend}(x, l, y).l : \alpha_2}\ (\text{Sel})$$

Lemma 4.2.2. If $K_1, \Gamma \vdash_\chi M : \sigma$ and (K_2, S) respects K_1, then $K_2, S(\Gamma) \vdash_\chi M : S(\sigma)$.

Proof. The proof follows by induction on the typing derivation of M:

- Case (Var): $M \equiv x$ and we know that $K_1, \Gamma \vdash_\chi x : \sigma$ and (K_2, S) respects K_1. Then $\sigma = \tau$ for some τ Γ is well-formed under K_1 and $K_1 \Vdash_\chi \Gamma(x) \geq \tau$. Since Γ is well-formed under K_1 and (K_2, S) respects $K_1, S(\Gamma)$ is well-formed under K_2 and $K_2 \Vdash_\chi (S(\Gamma))(x) \geq S(\tau)$. By the rule (Var), $K_2, S(\Gamma) \vdash_\chi x : S(\tau)$.

- Case (Abs): $M \equiv \lambda x.N$ and we know that $K_1, \Gamma \vdash_\chi \lambda x.N : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_1 \to \tau_2$ for some τ_1 and τ_2 such that $K_1, \Gamma\{\alpha : \tau_1\} \vdash_\chi N : \tau_2$. By the induction hypothesis, $K_2, S(\Gamma)\{\alpha : S(\tau_1)\}) \vdash_\chi N : S(\tau_2)$. By the rule (Abs), $K_2, S(\Gamma) \vdash_\chi \lambda x.N : S(\tau_1) \to S(\tau_2)$. Note that $S(\tau_1) \to S(\tau_2) \equiv S(\tau_1 \to \tau_2)$.

- Case (App): $M \equiv M_1\ M_2$ and we know that $K_1, \Gamma \vdash_\chi M_1\ M_2 : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_2$, for some τ_2 such that $K_1, \Gamma \vdash_\chi M_1 : \tau_1 \to \tau_2$ and $K_1, \Gamma \vdash_\chi M_2 : \tau_1$, for some τ_1. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi M_1 : S(\tau_1 \to \tau_2)$ and $K_2, S(\Gamma) \vdash_\chi M_2 : S(\tau_1)$. By the rule (App), $K_2, S(\Gamma) \vdash_\chi M_1\ M_2 : S(\tau_2)$.

- Case (Let): $M \equiv \text{let } x = M_1 \text{ in } M_2$ and we know that $K_1, \Gamma \vdash_\chi \text{let } x = M_1 \text{ in } M_2 : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau$ for some τ such that $K_1, \Gamma \vdash_\chi M_1 : \sigma'$ for some σ' and $K_1, \Gamma\{x : \sigma'\} \vdash_\chi M_2 : \tau$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi M_1 : S(\sigma')$ and $K_2, (S(\Gamma))\{x : S(\sigma')\}$. By the (Let) rule, $K_2, S(\Gamma) \vdash_\chi \text{let } x = M_1 \text{ in } M_2 : S(\tau)$.

- Case (Rec): $M \equiv \{l_1 = M_1, \ldots, l_n = M_n\}$ and we know that $K_1, \Gamma \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : \sigma$. Then $\sigma = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ for some τ_i $(1 \leq i \leq n)$ such that $K_1, \Gamma \vdash_\chi M_i : \tau_i$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi M_i : S(\tau_i)$. By the (Rec) rule, $K_2, S(\Gamma) \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : S(\{l_1 : \tau_1, \ldots, l_n = \tau_n\})$. Note that $\{l_1 : S(\tau_1), \ldots, l_n = S(\tau_n)\} = S(\{l_1 : \tau_1, \ldots, l_n = \tau_n\})$.

- Case (Sel): $M \equiv N.l$ and we know that $K_1, \Gamma \vdash_\chi N.l : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_2$ for some τ_2 such that $K_1, \Gamma \vdash_\chi N : \tau_1$ for some τ_1 and $K_1 \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \, \|\}\}$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi N : S(\tau_1)$. By Lemma 4.2.1, $K_2 \Vdash_\chi S(\tau) :: \{\{l : S(\tau_2) \, \|\}\}$. By the rule (Sel), $K_2, S(\Gamma) \vdash_\chi N.l : S(\tau_2)$.

- Case (Modif): $M \equiv \text{modify}(M_1, l, M_2)$ and we know that $K_1, \Gamma \vdash_\chi \text{modify}(M_1, l, M_2) : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_1$ for some τ_1 such that $K_1, \Gamma \vdash_\chi M_1 : \tau_1$, $K_1, \Gamma \vdash_\chi M_2 : \tau_2$ for some τ_2 and $K_1 \vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi M_1 : S(\tau_1)$ and $K_2, S(\Gamma) \vdash_\chi M_2 : S(\tau_2)$. By Lemma 4.2.1, $K_2 \vdash_\chi S(\tau_1) :: \{\{l : S(\tau_2) \parallel\}\}$. By rule (Modif), $K_2, S(\Gamma) \vdash_\chi \text{modify}(M_1, l, M_2) : S(\tau_1)$.

- Case (Gen): We know that $K_1, \Gamma \vdash_\chi M : \sigma$ and (K_2, S) respects K_1. Then $K_1', \Gamma \vdash_\chi M : \tau$ for some τ such that $\mathit{Cls}(K_1', \Gamma, \tau) = (K_1, \sigma)$. Since Γ is well-formed under K_1 and (K_2, S) respects K_1, $S(\Gamma)$ is well-formed under K_2. Since $\mathit{Cls}(K_1', \Gamma, \tau) = (K_1, \sigma)$, $K_1 = K_1'\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ and $\sigma \equiv \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n.\tau$. By the bound variable convention, $\{\alpha_1, \ldots, \alpha_n\} \notin \mathit{dom}(S)$, therefore (K_2, S) respects K_1', $S(\sigma) \equiv \forall \alpha_1 :: S(\kappa_1) \cdots \forall \alpha_n :: S(\kappa_n).S(\tau)$, and $\mathit{Cls}(K_2', S(\Gamma), S(\tau)) = (K_2, S(\sigma))$ such that $K_2' = K_2\{\alpha :: S(\kappa_1), \ldots, \alpha_n :: S(\kappa_n)\}$. By the induction hypothesis on $K_1', \Gamma \vdash_\chi M : \tau$, $K_1', S(\Gamma) \vdash_\chi M : S(\tau)$. By the rule (Gen), $K_2, S(\Gamma) \vdash_\chi M : S(\sigma)$.

- Case (Contr): $M \equiv N \setminus\setminus l$ and we know that $K_1, \Gamma \vdash_\chi N \setminus\setminus l : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_1 - \{l : \tau_2\}$ for some τ_1 and τ_2 such that $K_1, \Gamma \vdash_\chi N : \tau_1$ and $K_1 \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi N : S(\tau_1)$. By Lemma 4.2.1, $K_2 \Vdash_\chi S(\tau_1) :: \{\{l : S(\tau_2) \parallel\}\}$. By the rule (Contr), $K_2, S(\Gamma) \vdash_\chi N \setminus\setminus l : S(\tau_1 - \{l : \tau_2\})$.

- Case (Ext): $M \equiv \text{extend}(M_1, l, M_2)$ and we know that $K_1, \Gamma \vdash_\chi \text{extend}(M_1, l, M_2) : \sigma$ and (K_2, S) respects K_1. Then $\sigma \equiv \tau_1 + \{l : \tau_2\}$, for some τ_1 and τ_2 such that $K_1, \Gamma \vdash_\chi M_1 : \tau_1$, $K_1, \Gamma \vdash_\chi M_2 : \tau_2$, and $K_1 \Vdash_\chi \tau_1 :: \{\{\parallel l : \tau_2\}\}$. By the induction hypothesis, $K_2, S(\Gamma) \vdash_\chi M_1 : S(\tau_1)$ and $K_2, S(\Gamma) \vdash_\chi M_2 : S(\tau_2)$. By Lemma 4.2.1, $K_2 \Vdash_\chi S(\tau_1) :: \{\{\parallel l : S(\tau_2)\}\}$. By the rule (Ext), $K_2, S(\Gamma) \vdash_\chi \text{extend}(M_1, l, M_2) : S(\tau_1 + \{l : \tau_2\})$.

□

We recall the corollary for the above lemma (see Corollary 2.4.1.1).

Corollary 4.2.2.1. If (K_1, S_1) respects K and (K_2, S_2) respects K_1, then $(K_2, S_2 \circ S_1)$ respects K.

Lemma 4.2.3. If $K, \Gamma\{x : \sigma_1\} \vdash_\chi M : \tau$ and $K \vdash_\chi \sigma_2 \geq \sigma_1$, then $K, \Gamma\{x : \sigma_2\} \vdash_\chi M : \tau$.

Proof. Let $\sigma_1 = \forall \alpha_1^1 :: \kappa_1^1 \cdots \forall \alpha_n^1 :: \kappa_n^1.\tau_1$, and $\sigma_2 = \forall \alpha_1^2 :: \kappa_1^2 \cdots \forall \alpha_m^2 :: \kappa_m^2.\tau_2$. Since $K \Vdash_\chi \sigma_2 \geq \sigma_1$, we know there is a substitution S such that $\mathit{dom}(S) = \{\alpha_1^2, \ldots, \alpha_m^2\}$, $(K\{\alpha_1^1 :: \kappa_1^1, \ldots, \alpha_n^1 :: \kappa_n^1, S)$ respects $K\{\alpha_1^2 :: \kappa_1^2, \ldots, \alpha_m^2 :: \kappa_m^2\}$, and $\tau_1 = S(\tau_2)$. The proof follows by induction on the typing derivation of M:

- Case (Var): $M \equiv y$ and we know that $\Gamma\{x : \sigma_1\}$ is well-formed under K and $K \Vdash_\chi \Gamma\{x : \sigma_1\}(y) \geq \tau$. To show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi y : \tau$, we need to show that $\Gamma\{x : \sigma_2\}$ is well-formed under K and that $K \Vdash_\chi \Gamma\{x : \sigma_2\}(y) \geq \tau$, and then to apply the (Var) rule. To show that $\Gamma\{x : \sigma_2\}$ is well-formed under K, we just need to show that σ_2 is well-formed

under K. But, since $K \Vdash_\chi \sigma_2 \geq \sigma_1$, we already know that σ_2 is well-formed under K. To show that $K \Vdash_\chi \Gamma\{x : \sigma_2\}(y) \geq \tau$, we need to consider two possible scenarios. If $x \equiv y$, then we know that $K \Vdash_\chi \sigma_1 \geq \tau$ and we have to show that $K \Vdash_\chi \sigma_2 \geq \tau$, which is easy to prove using the fact that $K \Vdash_\chi \sigma_2 \geq \sigma_1$. If, on the other hand, $x \not\equiv y$, then the type of x does not change in Γ and we already know that $K \Vdash_\chi \sigma_1 \geq \tau$.

- Case (Const): $M \equiv c^b$, $\tau \equiv b$, and we know that $\Gamma\{x : \sigma_1\}$ is well-formed under K. To show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi c^b : b$, we just need to show that $\Gamma\{x : \sigma_2\}$ is well-formed under K (which we already know to be true) and then to apply the (Const) rule.

- Case (Abs): $M \equiv \lambda y.N$, $\tau \equiv \tau_1 \to \tau_2$, and we know that $K, \Gamma\{x : \sigma_1, y : \tau_1\} \vdash_\chi N : \tau_2$. To show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi \lambda y.N : \tau_1 \to \tau_2$, we need to show that $K, \Gamma\{x : \sigma_2, y : \tau_1\} \vdash_\chi N : \tau_2$ and then to apply the (Abs) rule. But we already know, by the induction hypothesis, that $K, \Gamma\{x : \sigma_2, y : \tau_1\} \vdash_\chi N : \tau_2$.

- Case (App): $M \equiv M_1\ M_2$, $\tau \equiv \tau_2$, and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_1 : \tau_1 \to \tau_2$ and $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_2 : \tau_2$. To show that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_1\ M_2 : \tau_2$, we need to show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_1 : \tau_1 \to \tau_2$ and $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_2 : \tau_1$, and then apply the (App) rule. But we already know, by the induction hypothesis, that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_1 : \tau_1 \to \tau_2$ and $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_2 : \tau_1$.

- Case (Let): $M \equiv \text{let } y = M_1 \text{ in } M_2$ and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_1 : \tau$ and $K, \Gamma\{x : \sigma_1, y : \sigma\} \vdash_\chi M_2 : \tau$. To show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi \text{let } y = M_1 \text{ in } M_2 : \tau$, we need to show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_1 : \sigma$ and $K, \Gamma\{x : \sigma_2, y : \sigma\} \vdash_\chi M_2 : \tau$, and then apply the (Let) rule. But we already know, by the induction hypothesis, that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M_1 : \sigma$ and $K, \Gamma\{x : \sigma_2, y : \sigma\} \vdash_\chi M_2 : \tau$.

- Case (Rec): $M \equiv \{l_1 = M_1, \ldots, l_n = M_n\}$, $\tau \equiv \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, and $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_i$ $(1 \leq i \leq n)$. This case can be easily proved using the induction hypothesis.

- Case (Sel): $M \equiv N.l$, $\tau \equiv \tau_2$, and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi N : \tau_1$ and $K \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$. To show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi N.l$, we need to show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi N : \tau_1$ and $K \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$, and then apply the (Sel) rule. But we already know that $K \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$ and, by the induction hypothesis, that $K, \Gamma\{x : \sigma_2\} \vdash_\chi N : \tau_1$.

- Case (Modif): $M \equiv \text{modify}(M_1, l, M_2)$, $\tau \equiv \tau_1$, and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_1 : \tau_1$, $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_2 : \tau_2$ and $K \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$. The proof for this case is very similar to the proof for (Sel).

- Case (Gen): We know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M : \tau'$ and $\mathit{Cls}(K, \Gamma\{x : \sigma_1\}, \tau') = (K', \tau)$. To show that $K', \Gamma\{x : \sigma_2\} \vdash_\chi M : \tau$, we need to show that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M : \tau'$ and $\mathit{Cls}(K, \Gamma\{x : \sigma_2\}, \tau') = (K', \sigma')$. At first sight, it may seem that we cannot apply the induction hypothesis since we have that $K' \Vdash_\chi \tau_2 \geq \tau_1$ and not that $K \Vdash_\chi \tau_2 \geq \tau_1$. In fact, since the type closure of τ' is a monotype τ, we know, by Definition 2.4.16, that $\tau' = \tau$ and $K' = K$. This means we can apply the induction hypothesis and conclude that $K, \Gamma\{x : \sigma_2\} \vdash_\chi M : \tau'$. Now, we just need to show that $\mathit{Cls}(K, \Gamma\{x : \sigma_2\}, \tau) = (K, \sigma')$.

Since τ is a monotype, $EFTV(K, \tau) = \emptyset$, which means that the closure of τ does not depend on the type of x and $Cls(K, \Gamma\{x : \sigma_2\}, \tau) = Cls(K, \Gamma\{x : \sigma_1\}, \tau) = (K, \tau)$.

- Case (Contr): $M \equiv N \setminus\setminus l$, $\tau \equiv \tau_1 - \{l : \tau_2\}$, and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi N : \tau_1$ and $K \Vdash_\chi \tau_1 :: \{\{l : \tau_2 \parallel\}\}$. The proof for this case is very similar to the proof for (Sel).
- Case (Ext): $M \equiv \text{extend}(M_1, l, M_2)$, $\tau \equiv \tau_1 + \{l : \tau_2\}$, and we know that $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_1 : \tau_1$, $K, \Gamma\{x : \sigma_1\} \vdash_\chi M_2 : \tau_2$ and $K \Vdash_\chi \tau_1 :: \{\{\parallel l : \tau_2\}\}$.The proof for this case is very similar to the proof for (Sel).

□

4.3 Type Inference

In this section, we are going to introduce extended versions of the kinded type inference and unification algorithms that were first introduced in Section 2.4.

4.3.1 Kinded Unification

In this type-system we will also consider satisfiability of kinded sets of equations modulo χ-equivalence.

Definition 4.3.1. A substitution S satisfies E if $S(\tau) \equiv_\chi S(\tau')$, for all $(\tau, \tau') \in E$.

The definitions in Section 2.4 that mention unifiers and most general unifiers are also valid in this type-system.

Notation 4.3.1. Let F range over functions from a finite set of labels to types.

- We write $F_1 - F_2$ for the function F such that $dom(F) = dom(F_1) \setminus dom(F_2)$ and, for $l \in dom(F)$, $F(l) = F_1(l)$.
- For an extensible type χ, we write $F_{e(\chi)}$ and $F_{c(\chi)}$ for the functions that represent the set of field-addition types and field-removal types of χ, respectively.

Example 4.3.1. Let $\chi = \alpha + \{l_1 : \tau_1\} - \{l_2 : \tau_2\} + \{l_3 : \tau_3\}$. Then $F_{e(\chi)}$ is the function (of domain $\{l_1, l_3\}$) that sends l_1 to τ_1 and l_3 to τ_3 and $F_{c(\chi)}$ is the function (of domain $\{l_2\}$) that sends l_2 to τ_2.

The ***kinded unification algorithm*** presented in Section 2.4 is extended to extensible types as follows.

$(E \cup \{(\tau_1, \tau_2)\}, K, S, SK) \Rightarrow_{\mathcal{U}_\chi} (E, K, S, SK)$ if $\tau_1 \equiv_\chi \tau_2$

$(E \cup \{(\alpha, \tau)\}, K \cup \{(\alpha, \mathcal{U})\}, S, SK) \Rightarrow_{\mathcal{U}_\chi} ([\tau/\alpha]E, [\tau/\alpha]K, [\tau/\alpha](S) \cup \{(\alpha, \tau)\}, [\tau/\alpha](SK) \cup \{(\alpha, \mathcal{U})\})$
if $\alpha \notin FTV(\tau)$

$(E \cup \{(\alpha_1, \alpha_2)\}, K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}, S, SK)$
$\Rightarrow_{\mathcal{U}_\chi} ([\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}),$
$[\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\},$
$[\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}, [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\})$
if $dom(F_1^l) \cap dom(F_2^r) = \emptyset$ and $dom(F_1^r) \cap dom(F_2^l) = \emptyset$

$(E \cup \{(\alpha, \{F_2\})\}, K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}, S, SK)$
$\Rightarrow_{\mathcal{U}_\chi} ([\{F_2\}/\alpha](E \cup \{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)\}),$
$[\{F_2\}/\alpha](K), [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}), [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$
if $dom(F_1^l) \subseteq dom(F_2), dom(F_1^r) \cap dom(F_2) = \emptyset,$ and $\alpha \notin FTV(\{F2\})$

$(E \cup \{(\{F_1\}, \{F_2\})\}, K, S, SK) \Rightarrow_{\mathcal{U}_\chi} (E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}, K, S, SK)$ if $dom(F_1) = dom(F_2)$

$(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2\}, K, S, SK) \Rightarrow_{\mathcal{U}_\chi} (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, K, S, SK)$

$(E \cup \{(\alpha, \chi)\}, K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}, S, SK)$
$\Rightarrow_{\mathcal{U}_\chi} ([\chi/\alpha](E \cup \{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\lfloor\chi\rfloor)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\lfloor\chi\rfloor)}))\}$
$\cup \{(F_1^r(l), F_{c(\lfloor\chi\rfloor)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\lfloor\chi\rfloor)})\}),$
$[\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\lfloor\chi\rfloor)}))) \parallel F_2^r + (F_1^r - F_{c(\lfloor\chi\rfloor)})\}\}))\},$
$[\chi/\alpha](S) \cup \{(\alpha, \chi)\}, [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$
if $dom(F_1^l) \cap dom(F_{c(\lfloor\chi\rfloor)}) = \emptyset, dom(F_1^r) \cap dom(F_2^r + (F_2^l - F_{c(\lfloor\chi\rfloor)})) = \emptyset,$ and $\alpha \notin FTV(\chi)$

$(E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^1 : \tau_m^2\})\}, K, S, SK)$
$\Rightarrow_{\mathcal{U}_\chi} (E \cup \{(\tau_i^1, \tau_j^2), (\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \cdots \pm_n^1 \{l_n^1 : \tau_n^1\},$
$\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}, K, S, SK)$
if $(\pm_i^1 = \pm_j^2 \wedge l_i^1 = l_j^2), \forall i < k \leq n : l_k^1 \neq l_i^2,$ and $\forall j < r \leq m : l_r^2 \neq l_j^2$

$(E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\},$
$K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}, S, SK)$
$\Rightarrow_{\mathcal{U}_\chi} (\mathcal{I}(E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}),$
$\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\},$
$\mathcal{I}(S) \cup \{(\alpha_1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha_2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\}$
$\mathcal{I}(SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$
where $\mathcal{I} = [\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}/\alpha_1, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}/\alpha_2]$
if $dom(F_1^l) \cap dom(F_2^r) = \emptyset, dom(F_1^r) \cap dom(F_1^l) = \emptyset, \alpha_1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}),$
$\alpha_2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}), \forall 1 \leq i \leq n, 1 \leq j \leq m, l_i^1 \neq l_j^1,$ and α is fresh

Figure 4.2: The unification algorithm for the ML-style Calculus with Extensible Types.

Definition 4.3.2. Let (K, E) be a given kinded set of equations. The ***kinded unification function for extensible types*** $\mathcal{U}_\chi(E, K)$ that takes a any kinded set of equations, computes a most general unifier if one exists, and reports failure otherwise is defined by the transformation rules in Figure 4.2.

Since the constructions present in rule *vii)* of Definition 4.3.2 are somewhat intricate, we will state the following facts in hope that they will help the reader better understand this rule. Let α be type variable and χ an extensible type, both well formed under some kind assignment K, such that $root(\chi) = \alpha_\chi$, $K(\alpha) = \{\{F_1^l \parallel F_1^r\}\}$, $K(\alpha_\chi) = \{\{F_2^l \parallel F_2^r\}\}$. Also, let us assume that $dom(F_1^l) \cap dom(F_{c(\chi)}) = \emptyset$, $dom(F_1^r) \cap dom(F_2^r + (F_2^l - F_{c(\chi)})) = \emptyset$ and $\alpha \notin FTV(\chi)$:

- Any field that appears in F_2^l was either introduced by the (Sel), (Modif) or (Contr) rule. This means that the set of the labels that appear in type contractions of χ is $dom(F_{c(\chi)}) \subseteq dom(F_2^l)$.
- Any field that appears on F_2^r was introduced by the (Ext) rule. This means that the set of the labels that appear in type extensions of χ is $dom(F_2^r) \subseteq dom(F_{e(\chi)})$.
- The set of fields that are guaranteed to appear in a record typed with χ is represented by the function $F_2^r + (F_2^l - F_{c(\chi)})$ and the set of fields that are guaranteed to *not* appear in a record typed with χ is represented by the function $F_{c(\chi)}$.
- The set of fields that are guaranteed to exist by $K(\alpha)$, but not by χ, is represented by the function $F_1^l - (F_2^r + (F_2^l - F_{c(\chi)}))$ and the set of fields that are guaranteed *not* to exist by $K(\alpha)$, but not by χ, is represented by the function $F_1^r - F_{c(\chi)}$.

Note that we use $|\chi|$ instead of χ in $F_{c(\chi)}$ so that constructing this function is more straightforward. That being said, to keep proofs simpler, we do not normalize extensible types during unification.

Example 4.3.2.

$$(\{(\alpha_1 + \{l : \alpha_3\} - \{l : \alpha_3\}, \alpha_2 - \{l : \alpha_3\})\}, \{\alpha_1 :: \{\{\parallel l : \alpha_3\}\}, \alpha' :: \{\{l : \alpha_3 \parallel\}\}, \alpha_3 :: \mathcal{U}\}, \emptyset, \emptyset) \Rightarrow_{\mathcal{U}_\chi}$$
$$(\{(\alpha_1 + \{l : \alpha_3\}, \alpha_2)\}, \{\alpha_1 :: \{\{\parallel l : \alpha_3\}\}, \alpha_2 :: \{\{l : \alpha_3 \parallel\}\}, \alpha_3 :: \mathcal{U}\}, \emptyset, \emptyset) \Rightarrow_{\mathcal{U}_\chi}$$
$$(\emptyset, \{\alpha_1 :: \{\{\parallel l : \alpha_3\}\}, \alpha_3 :: \mathcal{U}\}, \{(\alpha_2, \alpha_1 + \{l : \alpha_3\})\}, \{(\alpha_2, \{\{l : \alpha_3 \parallel\}\})\})$$

The most general unifier of the kinded set of equations

$$(\{\alpha_1 :: \{\{\parallel l : \alpha_3\}\}, \alpha_2 :: \{\{l : \alpha_3 \parallel\}\}, \alpha_3 :: \mathcal{U}\}, \{(\alpha_1 + \{l : \alpha_3\} - \{l : \alpha_3\}, \alpha_2 - \{l : \alpha_3\})\})$$

is the kinded substitution

$$(\{\alpha_1 :: \{\{l : \alpha_2 \parallel\}\}\}, \{(\alpha_2, \alpha_1 + \{l : \alpha_3\})\}).$$

We could have defined kinded normalization modulo χ-equivalence. This would simplify the description of the unification algorithm, but would also make it more abstract and less

"implementation friendly". To keep the algorithm closer to its implementation, we could have added normalization steps with unification steps. This would ensure the algorithm would be able to unify types that are not in χ-normal form as well. But now this would not only make its description more complicated again, but its proofs as well. Another more pertinent reason for considering normalization modulo χ-equivalence has to do with the fact that it might actually look like it would be the correct way to define the unification algorithm. But this is not true as can be seen in Example 4.3.3. In fact, both alternatives are (in a sense) equivalent. For this reason, we decided both to not consider unification modulo χ-equivalence, nor to normalize extensible types during unification and to accept the fact that its description is going to be more complicated because of that.

Example 4.3.3. Let $K = \{\alpha :: \{\{\| \ l : \alpha_2\}\}, \beta :: \{\{l : \alpha_2 \ \|\}\}\}$, $\chi_1 \equiv \alpha + \{l : \alpha_2\} - \{l : \alpha_2\}$, and $\chi_2 \equiv \beta - \{l : \alpha_2\}$. Then $|\,\chi_1\,| \equiv \alpha$ and $|\,\chi_2\,| \equiv \chi_2$.

If we unify χ_1 and χ_2 under K, we obtain the following kinded substitution:

$$(\{\alpha :: \{\{\| \ l : \alpha_2\}\}\}, [\alpha + \{l : \alpha_2\}/\beta]);$$

and the following common instance

$$\alpha + \{l : \alpha_2\} - \{l : \alpha_2\} \equiv_\chi \alpha.$$

If, on the other hand, we unify $\mid \chi_1 \mid$ and $\mid \chi_2 \mid$ under K, we get the following kinded substitution:

$$(\{\beta :: \{\{l : \alpha_2 \ \|\}\}\}, [\beta - \{l : \alpha_2\}/\alpha]);$$

and the following common instance

$$\beta - \{l : \alpha_2\}.$$

At first sight, $(\{\alpha :: \{\{\| \ l : \alpha_2\}\}\}, [\alpha + \{l : \alpha_2\}/\beta])$ might seem more general than $(\{\beta :: \{\{l : \alpha_2 \ \|\}\}\}, [\beta - \{l : \alpha_2\}/\alpha])$. In fact, the following equalities also appear to suggest that this is true

$$\begin{aligned} [\alpha + \{l : \alpha_2\}/\beta](\alpha) &\equiv \alpha \\ [\alpha + \{l : \alpha_2\}/\beta](\beta - \{l : \alpha_2\}) &\equiv \alpha + \{l : \alpha_2\} - \{l : \alpha_2\} \\ &\equiv_\chi \alpha \end{aligned}$$

$$\begin{aligned} [\beta - \{l : \alpha_2\}/\alpha](\alpha + \{l : \alpha_2\} - \{l : \alpha_2\}) &\equiv \beta - \{l : \alpha_2\} + \{l : \alpha_2\} - \{l : \alpha_2\} \\ &\equiv_\chi \beta - \{l : \alpha_2\} \\ [\beta - \{l : \alpha_2\}/\alpha](\beta - \{l : \alpha_2\}) &\equiv \beta - \{l : \alpha_2\}. \end{aligned}$$

In reality, because equality of substitutions is modulo χ-equivalence (see Definition 4.2.9),

they are both equally general under Definition 2.4.21.

$$\begin{aligned}
&[\beta - \{l : \alpha_2\}/\alpha] \circ [\beta - \{l : \alpha_2\} + \{l : \alpha_2\}/\beta] \\
&\quad = [\beta - \{l : \alpha_2\}/\alpha, \beta - \{l : \alpha_2\} - \{l : \alpha_2\} + \{l : \alpha_2\}/\beta] \\
&\quad = [\beta - \{l : \alpha_2\}/\alpha]
\end{aligned}$$

$$\begin{aligned}
&[\beta - \{l : \alpha_2\} + \{l : \alpha_2\}/\beta] \circ [\beta - \{l : \alpha_2\}/\alpha] \\
&\quad = [\beta - \{l : \alpha_2\} + \{l : \alpha_2\}/\beta, \beta - \{l : \alpha_2\} + \{l : \alpha_2\} - \{l : \alpha_2\}/\alpha] \\
&\quad = [\beta - \{l : \alpha_2\}/\alpha].
\end{aligned}$$

Theorem 4.3.1. Algorithm $\mathcal{U}_\chi$ takes any kinded set of equations and computes a most general unifier, if one exists; otherwise it fails.

Proof. We will only give a sketch of the proof here. The complete proof can be found in the Appendix A.

We first show that if the algorithm returns a kinded substitution, then it is a most general unifier of a given kinded set of equations.

Property 1 is composed out of the following sub-properties:

(1.1) K and $K \cup SK$ are well-formed kind assignments.

(1.2) E is well-formed under K.

(1.3) S is a well-formed substitution under K.

(1.4) $dom(K) \cap dom(SK) = \emptyset$.

(1.5) $dom(SK) = dom(S)$.

It is easily verified that each transformation rule preserves ***Property 1*** on 4-tuples.

We now state two more properties that are also preserved by each transformation rule:

- ***Property 2***: For any kinded substitution (K_0, S_0), if (K_0, S_0) respects K and S_0 satisfies $E \cup S$ then (K_0, S_0) respect SK.
- ***Property 3***: The set of unifiers of $(K \cup SK, E \cup S)$.

We can verify that these two properties are preserved by each transformation rule knowing that ***Property 1*** holds for the 4-tuple.

Using the previous three properties, we can conclude the correctness of the algorithm. Let (K, E) be a given kinded set of equations. Suppose the algorithm terminates with (K', S). Then there is some SK such that $(E, K, \emptyset, \emptyset)$ is transformed to $(\emptyset, K', S, SK)$ by repeated applications

of the transformation rules. ***Property 1*** trivially holds for $(E, K, \emptyset, \emptyset)$, which means that (K', S) is a kinded substitution, and $dom(S) \cap dom(K') = \emptyset$. Therefore (K', S) respects K'. S also trivially satisfies $S \cup \emptyset$, therefore, by ***Property 2***, (K', S) also respects SK, therefore, (K', S) is a unifier of $(K' \cup SK, \emptyset \cup S)$. By ***Property 3***, (K', S) is also a unifier of $(K' \cup SK, \emptyset \cup S)$. Let (K_0, S_0) be any unifier of (K, E). By ***Property 3***, it is also a unifier of $(K' \cup SK, \emptyset \cup S)$. But, then $S_0 = S_0 \circ S$ and (K', S) is more general than (K_0, S_0). Conversely, suppose the algorithm fails. Then $(E, K, \emptyset, \emptyset)$ is transformed to (E', K', S', SK') for some E', K', S', SK' such that $E' \neq \emptyset$, and no rule applies to (E', K', S', SK'). It is clear from the definition of each rule that $(K', SK', E' \cup S')$ has no unifier, and therefore, by ***Property 3***, that (K, E) has no unifier. The termination can be proved by showing that each transformation rule decreases the complexity measure of the lexicographical pair consisting of the size of the set $dom(K)$ and the total number of occurrences of type constructors (including base types) in E. □

4.3.2 Type Inference Algorithm

Using this extended version of the kinded unification algorithm, we extend the type inference algorithm presented in Section 2.4 with extensible types.

Definition 4.3.3. Let Γ be a type assignment, K be a kinding environment, $M \in \Lambda_\chi$ and $\tau \in \mathbb{T}_\chi$. The ***kinded type inference function for extensible types*** $WK_\chi(K, \Gamma, M)$ that returns the triple (K', S, τ) such that $K', S(\Gamma) \vdash_\chi M : \tau$ is defined inductively in Figure 4.3.

Proposition 4.3.1. If $WK_\chi(K, \Gamma, M) = (K', S, \tau)$, then τ is in χ-normal form.

Proof. The proof is by induction on the structure of M.

- Case x: By definition, $|S(\tau)|$ is in normal form.
- Case $\lambda x.M$: By the induction hypothesis, τ_1 is in χ-normal form. Also, by definition, $|S_1(\alpha)|$ is a χ-normal form. Thus $|S_1(\alpha)| \rightarrow \tau_1$ is normal form.
- Case $M_1\ M_2$: By definition, $|S_3(\alpha)|$ is in normal form.
- Case let $x = M_1$ in M_2: By the induction hypothesis, τ_2 is in normal form.
- Case $\{l_1 = M_1, \ldots, l_n = M_n\}$: By the induction hypothesis, τ_n is in χ-normal form. Also, by definition, $|S_n \circ \cdots \circ S_i(\tau_i)|$ $(1 \leq i < n)$, is in χ-normal form. Thus $\{l_1 : |S_n \circ \cdots S_2(\tau_1)|, \ldots, l_i : |S_n \circ \cdots \circ S_{i+1}(\tau_i)|, \ldots, l_n : \tau_n\}$ is in χ-normal form.
- Case $M.l$: By definition, $|S_2(\alpha_1)|$ is in χ-normal form.
- Case modify(M_1, l, M_2): By definition, $|S_3(\alpha_2)|$ is in χ-normal form.
- Case $M \setminus\setminus l$: By definition, $|S_2(\alpha_2 - \{l : \alpha_1\})|$ is in χ-normal form.
- Case extend(M_1, l, M_2): By definition, $|S_2(\alpha_2 + \{l : \alpha_1\})|$ is in χ-normal form.

$$
\begin{array}{rcl}
WK_\chi(K,\Gamma,x) & = & \text{if } x \notin dom(\Gamma) \text{ then } \mathit{fail} \\
& & \text{else let } \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau = \Gamma(x), \\
& & \qquad S = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n] \ (\beta_1, \ldots, \beta_n \text{ are fresh}) \\
& & \quad \text{in } (K\{\beta_1 :: S(\kappa_1), \ldots, \beta_n :: S(\kappa_n)\}, id, |\, S(\tau)\,|) \\
\\
WK_\chi(K,\Gamma,\lambda x.M) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K\{\alpha :: \mathcal{U}\}, \Gamma\{x : \alpha\}, M) \ (\alpha \text{ fresh}) \\
& & \text{in } (K_1, S_1, |\, S_1(\alpha) \,| \to \tau_1) \\
\\
WK_\chi(K,\Gamma,M_1\ M_2) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M_1) \\
& & \quad (K_2, S_2, \tau_2) = WK_\chi(K_1, S_1(\Gamma), M_2) \\
& & \quad (K_3, S_3) = \mathcal{U}(K_2\{\alpha :: \mathcal{U}\}, \\
& & \qquad\qquad \{(S_2(\tau_1), \tau_2 \to \alpha)\}) \ (\alpha \text{ fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, |\, S_3(\alpha)\,|) \\
\\
WK_\chi(K,\Gamma,\text{let } x = M_1 \text{ in } M_2) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M_1) \\
& & \quad (K_1', \sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \tau_1) \\
& & \quad (K_2, S_2, \tau_2) = WK_\chi(K_1', (S_1(\Gamma))\{x : \sigma\}, M_2) \\
& & \text{in } (K_2, S_2 \circ S_1, \tau_2) \\
\\
WK_\chi(K,\Gamma,\{l_1 = M_1, \ldots, l_n = M_n\}) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M_1) \\
& & \quad (K_i, S_i, \tau_i) = WK_\chi(K_{i-1}, S_{i-1} \circ \cdots \circ S_1(\Gamma), M_i) \ (2 \le i \le n) \\
& & \text{in } (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \\
& & \quad \{l_1 :|\, S_n \circ \cdots \circ S_2(\tau_1)\,|, \ldots, l_i :|\, S_n \circ \cdots \circ S_{i+1}(\tau_i)\,|, \ldots, l_n : \tau_n\}) \\
\\
WK_\chi(K,\Gamma,M.l) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M) \\
& & \quad (K_2, S_2) = \mathcal{U}(K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \\
& & \qquad\qquad \{(\alpha_2, \tau_1)\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \text{in } (K_2, S_2 \circ S_1, |\, S_2(\alpha_1)\,|) \\
\\
WK_\chi(K,\Gamma,\text{modify}(M_1, l, M_2)) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M_1) \\
& & \quad (K_2, S_2, \tau_2) = WK_\chi(K_1, S_1(\Gamma), M_2) \\
& & \quad (K_3, S_3) = \mathcal{U}(K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \\
& & \qquad\qquad \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \text{in } (K_3, S_3 \circ S_2 \circ S_1, |\, S_3(\alpha_2)\,|) \\
\\
WK_\chi(K,\Gamma,M \setminus\setminus l) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M) \\
& & \quad (K_2, S_2) = \mathcal{U}(K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \\
& & \qquad\qquad \{(\alpha_2, \tau_1)\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \text{in } (K_2, S_2 \circ S_1, |\, S_2(\alpha_2 - \{l : \alpha_1\})\,|) \\
\\
WK_\chi(K,\Gamma,\text{extend}(M_1, l, M_2)) & = & \text{let } (K_1, S_1, \tau_1) = WK_\chi(K,\Gamma,M_1) \\
& & \quad (K_2, S_2, \tau_2) = WK_\chi(K_1, S_1(\Gamma), M_2) \\
& & \text{in if } \mathit{root}(\tau_1) \in FTV(\tau_2) \text{ then } \mathit{fail} \\
& & \quad \text{else let } (K_3, S_3) = \mathcal{U}(K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{\parallel l : \alpha_1\}\}\}, \\
& & \qquad\qquad\qquad \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\}) \ (\alpha_1, \alpha_2 \text{ fresh}) \\
& & \qquad \text{in } (K_3, S_3 \circ S_2 \circ S_1, |\, S_3(\alpha_2 + \{l : \alpha_1\})\,|)
\end{array}
$$

Figure 4.3: Type inference algorithm WK_χ for the ML-style Calculus with Extensible Records.

□

Theorem 4.3.2. Given a kinding environment K, a type assignment Γ and $M \in \Lambda_\chi$. If $WK_\chi(K, \Gamma, M) = (K', S, \tau)$ then the following properties hold:

- (K', S) is a kinded substitution that respects K and $K', S(\Gamma) \vdash_\chi M : \tau$.
- If $K_0, S_0(\Gamma) \vdash_\chi M : \tau_0$ for some kinded substitutions (K_0, S_0) and $\tau_0 \in \mathbb{T}_\chi$ such that (K_0, S_0) respects K, then there is some type substitution S' such that the kinded substitution (K_0, S') respects K', $\tau_0 \equiv_\chi S'(\tau)$, and $S_0(\Gamma) = S' \circ S(\Gamma)$.

If $WK_\chi(K, \Gamma, M)$ fails, then there is no kinded substitution (K_0, S_0) and τ_0 such that (K_0, S_0) respects K and $K_0, S_0(\Gamma) \vdash_\chi M : \tau_0$.

Proof. Let us start by proving the soundness of the algorithm. The proof is by induction on the structure of M:

- Case x: Let $WK_\chi(K, \Gamma, x) = (K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, id, |\,[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau')\,|)$, where $\kappa'_i = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_i), (1 \leq i \leq n)$. For all $\alpha \in dom(K)$, $K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\} \vdash_\chi id(\alpha) :: id(K(\alpha)) \iff K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\} \vdash_\chi \alpha :: K(\alpha)$, therefore $(K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, id)$ respects K. Also, we know that , $(K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n])$ respects $K\{\alpha_1 :: \kappa_1, \ldots, \beta_n :: \kappa_n\}$, therefore $K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\} \vdash_\chi \forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\tau' \geq [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau')$. By rule (Var), we have that $K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, \Gamma \vdash_\chi x : [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau')$, therefore, $K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, \Gamma \vdash_\chi x :|\,[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau')\,|$.
- Case $\lambda x.M$: Let $WK_\chi(K, \Gamma, \lambda x.M) = (K_1, S_1, |\,S_1(\alpha)\,| \to \tau_1)(\alpha$ fresh). By the induction hypothesis, $WK_\chi(K\{\alpha :: \mathcal{U}\}, \Gamma\{x : \alpha\}, M) = (K_1, S_1, \tau_1)$, (K_1, S_1) respects $K\{\alpha :: \mathcal{U}\}$, and $K_1, S_1(\Gamma\{x : \alpha\}) \vdash_\chi M : \tau_1$. Also, for all for all $\alpha' \in dom(K\{\alpha :: \mathcal{U}\})$, $K_1 \vdash_\chi S_1(\alpha') :: S_1(K\{\alpha :: \mathcal{U}\}(\alpha'))$, which means that, for all $\alpha' \in dom(K)$:
  - $K_1 \Vdash_\chi S_1(\alpha') :: S_1(K(\alpha'))$, if $\alpha' \not\equiv \alpha$;
  - $K_1 \Vdash_\chi S_1(\alpha) :: S_1(\mathcal{U}) \iff K_1 \vdash_\chi S_1(\alpha) :: \mathcal{U}$, if $\alpha' \equiv \alpha$.

  Therefore, (K_1, S_1) respects K as well. Also, by the rule (Abs), we have that $K_1, S_1(\Gamma) \vdash_\chi \lambda x.M : S_1(\alpha) \to \tau_1$, therefore, $K_1, S_1(\Gamma) \vdash_\chi \lambda x.M :|\,S_1(\alpha)\,| \to \tau_1$.
- Case $M_1\ M_2$: Let $WK_\chi(K, \Gamma, M_1\ M_2) = (K_3, S_3 \circ S_2 \circ S_1, |\,S_3(\alpha)\,|)$. By the induction hypothesis, $WK_\chi(K, \Gamma, M_1) = (K_1, S_1, \tau_1)$, (K_1, S_1) respects K, $K_1, S_1(\Gamma) \vdash_\chi M_1 : \tau_1$. Also, by induction hypothesis, $WK_\chi(K_1, S_1(\Gamma), M_2) = (K_2, S_2, \tau_2)$, (K_2, S_2) respects K_1 and $K_2, S_2(S_1(\Gamma)) \vdash_\chi M_2 : \tau_2$. By Theorem 2.4.5, (K_3, S_3) respects K_2. By Lemma 4.2.1, $(K_3, S_3 \circ S_2 \circ S_1)$ respects K. By Lemma 4.2.2, $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3 \circ S_2(\tau_1)$ and $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : S_3(\tau_2)$. By Theorem 2.4.5, $S_3 \circ S_2(\tau_1) \equiv_\chi S_3(\tau_2 \to \alpha) \equiv S_3(\tau_2) \to S_3(\alpha)$, therefore $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3(\tau_2) \to S_3(\alpha)$. By rule (App), we have $K, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1\ M_2 : S_3(\alpha)$, therefore $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1\ M_2 :|\,S_3(\alpha)\,|$.

- Case let $x = M_1$ in M_2: Let $WK_\chi(K, \Gamma, \text{let } x = M_1 \text{ in } M_2) = (K_2, S_2 \circ S_1, \tau_2)$. By the induction hypothesis, (K_1, S_1) respects K, $K_1, S_1(\Gamma) \vdash_\chi M_1 : \tau_1$. Since $(K_1', \sigma) = \mathit{Cls}(K_1, S_1(\Gamma), \tau_1)$, then by rule (Gen), we have $K_1', S_1(\Gamma) \vdash_\chi M_1 : \sigma_1$. By the induction hypothesis, (K_2, S_2) respects K_1', $K_2, S_2 \circ S_1(\Gamma)\{x : S_2(\sigma_1)\} \vdash_\chi M_2 : \tau_2$. Now we show that, for any $\alpha \in \mathit{dom}(K)$, $S_1(\alpha)$ is well formed under K_1'. By the definition of the type inference algorithm and the unification algorithm, if $\alpha \notin \mathit{EFTV}(K, \Gamma)$, then α does not appear in τ_1 or S_1, and, therefore, $\mathit{FTV}(S_1(\alpha)) \subseteq \mathit{dom}(K_1')$. Suppose $\alpha \in \mathit{EFTV}(K, \Gamma)$. Then by a simple induction on the derivation of $\alpha \in \mathit{EFTV}(K, \Gamma)$, it is show that $\mathit{FTV}(S_1(\alpha)) \subseteq \mathit{EFTV}(K_1, S_1(\Gamma))$, and, therefore, $\mathit{FTV}(S_1(\alpha)) \subseteq \mathit{dom}(K_1')$. Thus, in either case, $S_1(\alpha)$ is well formed under K_1'. But, then (K_1', S_1) respects K, and by Lemma 4.2.1, $(K_2, S_2 \circ S_1)$ respects K. By Lemma 4.2.2, $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_2(\sigma_1)$. By rule (Let), $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi \text{let } x = M_1 \text{ in } M_2 : \tau_2$.

- Case $\{l_1 = M_1, \ldots, l_n = M_n\}$: Let $WK_\chi(K, \Gamma, \{l_1 = M_1, \ldots, l_n = M_n\}) = (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \{l_1 =\mid S_n \circ \cdots \circ S_2(\tau_1) \mid, \ldots, l_i :\mid S_n \circ \cdots \circ S_{i+1}(\tau_i) \mid, \ldots, l_n = \tau_n\}$. By the induction hypothesis, (K_1, S_1) respects K, $K_1, S_1(\Gamma) \vdash_\chi M_1 : \tau_1$. Also, by induction hypothesis, (K_i, S_i) respects K_{i-1}, and $K_i, S_i \circ S_{i-1} \circ \cdots \circ S_1(\Gamma) \vdash_\chi \tau_i (2 \leq i \leq n)$. By Lemma 4.2.1, $(K_n, S_n \circ \cdots \circ S_2 \circ S_1)$ respects K. By Lemma 4.2.2, $K_n, S_n \circ \cdots \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_i : S_n \circ \cdots \circ S_{i+1}(\tau_i)$. By rule (Rec), $K_n, S_n \circ \cdots \circ S_2 \circ S_1(\Gamma) \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\}$, therefore $K_n, S_n \circ \cdots \circ S_2 \circ S_1(\Gamma) \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} :\mid \{l_1 : S_n \circ \cdots \circ S_2(\tau_1), \ldots, l_i : S_n \circ \cdots \circ S_{i+1}(\tau_i), \ldots, l_n : \tau_n\} \mid$ and $K_n, S_n \circ \cdots \circ S_2 \circ S_1(\Gamma) \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 :\mid S_n \circ \cdots \circ S_2(\tau_1) \mid, \ldots, l_i :\mid S_n \circ \cdots \circ S_{i+1}(\tau_i) \mid, \ldots, l_n : \tau_n\}$.

- Case $M.l$: Let $WK_\chi(K, \Gamma, M.l) = (K_2, S_2 \circ S_1, \mid S(\alpha_1) \mid)$. By Theorem 4.3.1, (K_2, S_2) respects K_1 and $K_1, S_1(\Gamma) \vdash_\chi M : \tau_1$. By Lemma 4.2.1, $(K_2, S_2 \circ S_1)$ respects K. By Lemma 4.2.1, $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M : S_2(\tau_1)$. By Theorem 2.4.5:

  - $K_2 \Vdash_\chi S_2(\alpha_2) :: S_2(\{\{l : \alpha_1 \parallel\}\})$;
  - $K_2 \Vdash_\chi S_2(\tau_1) :: \{\{l : S_2(\alpha_1) \parallel\}\}$.

  But, then, by rule (Sel), $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M.l : S_2(\alpha_1)$, therefore $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M.l :\mid S(\alpha_1) \mid$.

- Case $\text{modify}(M_1, l, M_2)$: Let $WK_\chi(K, \Gamma, \text{modify}(M_1, l, M_2)) = (K_3, S_3 \circ S_2 \circ S_1, \mid S_3(\alpha_2) \mid)$. By Theorem 2.4.5, (K_3, S_3) respects $K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}$, therefore (K_3, S_3) respects K_2. By the induction hypothesis, (K_1, S_1) respects K and $K_1, S_1(\Gamma) \vdash_\chi M_1 : \tau_1$. Also, by the induction hypothesis, (K_2, S_2) respects K_1 and $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : \tau_2$. By Lemma 4.2.1, $(K_3, S_3 \circ S_2 \circ S_1)$ respects K. By Lemma 4.2.2, $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3 \circ S_2(\tau_1) \iff K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3(\alpha_2)$ and $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : S_3(\tau_2) \iff K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : S_3(\alpha_1)$. By Theorem 2.4.5, $S_3 \circ S_2(\tau_1) \equiv_\chi S_3(\alpha_2)$ and $S_3(\tau_2) \equiv_\chi S_3(\alpha_1)$. Also:

  - $K_3 \Vdash_\chi S_3(\alpha_2) :: S_3(\{\{l : \alpha_1 \parallel\}\})$;

– $K_3 \Vdash_\chi S_3 \circ S_2(\tau_1) :: \{\{l : S_3(\alpha_1) \parallel\}\} \iff K_3 \vdash_\chi S_3 \circ S_2(\tau_1) :: \{\{l : S_3(\tau_2) \parallel\}\}$.

Therefore, by rule (Modif), we have that $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi \text{modify}(M_1, l, M_2) : S_3(\alpha_2)$, and $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi \text{modify}(M_1, l, M_2) :| S_3(\alpha_2) |$.

- Case $M \setminus\setminus l$: Let $WK_\chi(K, \Gamma, M \setminus\setminus l) = (K_2, S_2 \circ S_1, | S_2(\alpha_2 - \{l : \alpha_1\}) |)$. By Theorem 2.4.5, (K_2, S_2) respects $K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}$, therefore (K_2, S_2) respects K_1 as well. By the induction hypothesis, (K_1, S_1) respects K and $K_1, \Gamma \vdash_\chi M : \tau_1$. By Lemma 2.4.1, $(K_2, S_2 \circ S_1)$ respects K. By Lemma 2.4.3, $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M : S_2(\tau_1)$. By Theorem 4.3.1, $S_2(\alpha) \equiv_\chi S_2(\tau_1)$, therefore $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M : S_2(\alpha)$ and $K_2 \vdash_\chi S_2(\tau_1) :: S_2(\{\{l : \alpha_1 \mid |\}\}) \iff K_2 \vdash_\chi S_2(\alpha_2) :: \{\{l : S(\alpha_1) \parallel\}\}$. By rule (Contr), $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M \setminus\setminus l : S_2(\alpha_2) - \{l : S_2(\alpha_1)\}$, therefore $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M \setminus\setminus l :| S_2(\alpha_2 - \{l : \alpha_1\}) |$.

- Case extend(M_1, l, M_2): Let $WK_\chi(K, \Gamma, \text{extend}(M_1, l, M_2)) = (K_3, S_3 \circ S_2 \circ S_1, | S_3(\alpha_2 + \{l : \alpha_1\}) |)$. By Theorem 2.4.5, (K_3, S_3) respects K_2. By the induction hypothesis, (K_1, S_1) respects K and $K_1, S_1(\Gamma) \vdash_\chi M_1 : \tau_1$. Also, by the induction hypothesis, (K_2, S_2) respects K and $K_2, S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : \tau_2$. By Lemma 4.2.1, $(K_3, S_3 \circ S_2 \circ S_1)$ respects K. By Lemma 4.2.2, $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3 \circ S_2(\tau_1)$ and $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : S_3(\tau_2)$. By Theorem 2.4.5, $S_3(\alpha_1) \equiv_\chi S_3(\tau_2)$ and $S_3(\alpha_2) \equiv_\chi S_3 \circ S_2(\tau_1)$, therefore $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_1 : S_3(\alpha_2)$, $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi M_2 : S_3(\alpha_1)$, and $K_3 \vdash_\chi S_3(\alpha_1) :: S_3(\{\{\parallel l : \alpha_1\}\}) \iff K_3 \vdash_\chi S_3(\tau_2) :: \{\{\parallel l : S_3(\alpha_1)\}\}$. By the induction hypothesis, we also know that $root(\tau_1) \notin FTV(\tau_2)$. By rule (Ext), we have $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi \text{extend}(M_1, l, M_2) : S_3(\alpha_2) + \{l : S_3(\alpha_1)\}$, therefore $K_3, S_3 \circ S_2 \circ S_1(\Gamma) \vdash_\chi \text{extend}(M_1, l, M_2) :| S_3(\alpha_2 + \{l : \alpha_1\}) |$.

Now, let us prove the completeness of the algorithm. The proof is also by induction on the structure of M:

- Case x: Let $WK_\chi(K, \Gamma, x) = (K\{\beta_1 :: \kappa'_1, \ldots, \beta_n :: \kappa'_n\}, id, | [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau') |)$, where $\kappa'_i = [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_i), (1 \leq i \leq n)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi x : \tau_0$ and (K_0, S_0) respects K. By the bound variable convention, $\alpha_1, \ldots, \alpha_n$ will not appear in S_0. Since $\beta_1, \ldots, \beta_n$ are fresh, they will not appear in S_0 as well. But, then $(S_0(\Gamma))(x) \equiv \forall\alpha_1 :: S_0(\kappa_1) \cdots \forall\alpha_n :: S_0(\kappa_n).S_0(\tau)$, and there is some S_1 such that $dom(S_1) = \{\alpha_1, \ldots, \alpha_n\}$, $S_1 \circ S_0(\tau) \equiv_\chi \tau_0$, $K_0 \vdash_\chi S_1(\alpha_i) :: S_1 \circ S_0(\kappa_i) (1 \leq i \leq n)$. Let $S_2 = S_1 \circ S_0 \circ [\alpha_1/\beta_1, \ldots, \alpha_n/\beta_n]$. Then (K_0, S_2) respects K, since $dom(K) \cap (dom(S_1) \cup \{\alpha_1, \ldots, \alpha_n\}) = \emptyset$, $S_2(\beta_i) \equiv S_1(\alpha_i)$, and $S_2 \circ [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_i) \equiv S_1 \circ S_0 \circ id(\kappa_i) \equiv S_1 \circ S_0(\kappa_i)$. Therefore $K_0 \vdash_\chi S_2(\beta_i) :: S_2 \circ [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_i)$. Thus (K_0, S_2) respects $K\{\beta_1 :: [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_1), \ldots, \beta_n :: [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\kappa_n)\}$. Also, we have $S_2 \circ [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\tau) \equiv S_1 \circ S_0 \circ id(\tau) \equiv S_1 \circ S_0(\tau) \equiv_\chi S_1 \circ S_0(|\tau|) \equiv_\chi \tau_0$ and $S_0(\Gamma) = S_2 \circ [\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n](\Gamma)$.

- Case $\lambda x.M$: Let $WK_\chi(K, \Gamma, \lambda x.M) = (K_1, S_1, | S_1(\alpha) |\to \tau_1)(\alpha \text{ fresh})$. Additionally, let $K_0, S_0(\Gamma) \vdash_\chi \lambda x.M : \tau_0^1 \to \tau_0^2$ and (K_0, S_0) respect K. But, then $K_0 S_0(\Gamma)\{x : \tau_0^1\} \vdash_\chi M : \tau_0^2$. Since α is fresh, it will not appear in S_0. Let $S'_1 = [\tau_0^1/\alpha] \circ S_0$. Then (K_0, S'_1) respects $K\{\alpha :: \mathcal{U}\}$, since τ_0^1 is well formed under K, and $K_0, S'_1(\Gamma\{x : \alpha\}) \vdash_\chi M : \tau_0^2$. Since $WK_\chi(K\{\alpha :: \mathcal{U}\}, \Gamma, \{x : \alpha\}, M) = (K_1, S_1, \tau_1)$ and $K_0, S'_1(\Gamma\{x : \alpha\}) \vdash_\chi M : \tau_0^2$, by

the induction hypothesis, there is an S_2' such that (K_0, S_2') respects $K_1, S_2'(\tau_1) \equiv_\chi \tau_0^2$, and $S_1'(\Gamma\{x : \alpha\}) = S_2' \circ S_1(\Gamma\{x : \alpha\}) \iff S_1'(\Gamma)\{x : \tau_0^1\} = S_2' \circ S_1(\Gamma)\{x : S_2' \circ S_1(\alpha)\} \Rightarrow \tau_0^1 \equiv S_2' \circ S_1(\alpha)$. But, then $S_2'(|\, S_1(\alpha)\,| \to \tau_1) \equiv_\chi |\, S_2'(|\, S_1(\alpha)\,|) \,| \to | \, S_2'(\tau_1) \,| \equiv_\chi \| \, S_2'(S_1(\alpha)) \,| \to | \, S_2'(\tau_1) \,\| \equiv_\chi | \, \tau_0^1 \,| \to | \, \tau_0^2 \,|$ and $S_0(\Gamma) = S_2' \circ S_1(\Gamma)$.

- Case $M_1\ M_2$: Let $WK_\chi(K, \Gamma, M_1\ M_2) = (K_3, S_3 \circ S_2 \circ S_1, |\, S_3(\alpha)\,|)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi M_1\ M_2 : \tau_0^2$ and (K_0, S_0) respects K. But, then $K_0, S_0(\Gamma) \vdash_\chi M_1 : \tau_0^1 \to \tau_0^2$ and $K_0, S_0(\Gamma) \vdash_\chi M_2 : \tau_0^1$. Since $WK_\chi(K, \Gamma, M_1) = (K_1, S_1, \tau_1)$ and $K_0, S_0(\Gamma) \vdash_\chi M_1 : \tau_0^1 \to \tau_0^2$, we have, by the induction hypothesis, that there is a S_1' such that (K_0, S_1') respects K, $S_1'(\tau_1) \equiv_\chi \tau_0^1 \to \tau_0^2$, and $S_0(\Gamma) = S_1' \circ S_1(\Gamma)$. But, then $K_0, S_1' \circ S_1(\Gamma) \vdash_\chi M_2 : \tau_0^1$. Since $WK_\chi(K_1, S_1(\Gamma), M_2) = (K_2, S_2, \tau_2)$ and $K_0, S_1' \circ S_1(\Gamma) \vdash_\chi M_2 : \tau_0^1$, we have, by the induction hypothesis, there is a S_2' such that (K_0, S_2') respects K_2, $S_2'(\tau_2) \equiv_\chi \tau_0^1$, and $S_1' \circ S_1(\Gamma) = S_2' \circ S_2 \circ S_1(\Gamma) \Rightarrow S_1' = S_2' \circ S_2$. Since α is fresh, it will not appear in S_2'. Let $S_3' = [\tau_0^2/\alpha] \circ S_2'$. Then (K_0, S_3') respects $K_2\{\alpha :: \mathcal{U}\}$, since (K_0, S_2') respects K_1 and τ_0^2 is well formed under K_2. Also, $|\, S_3'(S_2(\tau_1)) \,| \equiv_\chi S_3'(\tau_2 \to \alpha) \iff S_2' \circ S_2(\tau_1) \equiv_\chi S_2'(\tau_2) \to \tau_0^2 \iff S_1'(\tau_1) \equiv_\chi \tau_0^1 \to \tau_0^2$. Therefore (K_0, S_3') is an unifier of $(K_2\{\alpha, \mathcal{U}\}, \{(S_2(\tau_1), \tau_2 \to \alpha)\})$ and, by Theorem 4.3.1, there is a S_4' such that (K_0, S_4') respects K_3 and $S_3' = S_4' \circ S_3$. But, then $S_4'(|\, S_3(\alpha)\,|) \equiv_\chi S_4'(S_3(\alpha)) \equiv_\chi S_3'(\alpha) \equiv \tau_0^2$. Also, $S_0(\Gamma) = S_1' \circ S_1(\Gamma) = S_2' \circ S_2 \circ S_1(\Gamma) = [\tau_0^2/\alpha] \circ S_2' \circ S_2 \circ S_1(\Gamma) = S_3' \circ S_2 \circ S_1(\Gamma) = S_4' \circ S_3 \circ S_2 \circ S_1(\Gamma)$.

- Case let $x = M_1$ in M_2: Let $WK_\chi(K, \Gamma, \text{let } x = M_1 \text{ in } M_2) = (K_2, S_2 \circ S_1, \tau_2)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi \text{let } x = M_1 \text{ in } M_2 : \tau_0^2$ and (K_0, S_0) respect K. But, then $K_0', S_0(\Gamma) \vdash_\chi M_1 : \tau_0^1$, $Cls(K_0', S_0(\Gamma), \tau_0^1) = (K_0, \sigma_0^1)$, and $K_0, (S_0(\Gamma))\{x : \sigma_0^1\} \vdash_\chi M_2 : \tau_0^2$. By Definition 2.4.16 we know that $K_0' = K_0\{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\}$ such that $\{\alpha_1^0, \ldots, \alpha_m^0\} = EFTV(K_0', \tau_0^1) \setminus EFTV(K_0', S_0(\Gamma))$, and $\sigma_0^1 \equiv \forall \alpha_1^0 :: \kappa_1^0 \cdots \forall \alpha_m^0 :: \kappa_m^0 . \tau_0^1$. By the induction hypothesis, there is some S_0^1 such that (K_0', S_0^1) respects K_1, $S_0^1(\tau_1) \equiv_\chi \tau_0^1$, and $S_0(\Gamma) = S_0^1 \circ S_1(\Gamma)$. By Definition 2.4.16, $K_1 = K_1'\{\alpha_1 :: \kappa_1, \ldots, \alpha_n :: \kappa_n\}$ such that $\{\alpha_1, \ldots, \alpha_n\} = EFTV(K_1, \tau_1) \setminus EFTV(K_1, S_1(\Gamma))$, and $\sigma_1 \equiv \forall \alpha_1 :: \kappa_1 \cdots \forall \alpha_n :: \kappa_n . \tau_1$. By the bound variable convention, $\{\alpha_1, \ldots, \alpha_n\} \cap \{\alpha_1^0, \ldots, \alpha_m^0\} = \emptyset$. Since (K_0', S_0^1) respects K_1, $K_0\{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\} \vdash_\chi S_0^1(\alpha_i) :: S_0^1(\kappa_i) (1 \le i \le n)$. Let S_0^2 be the restriction of S_0^1 on $dom(S_0^1) \setminus \{\alpha_1, \ldots, \alpha_n\}$, and let S_0^3 be the substitutions $[S_0^1(\alpha_1)/\alpha_1, \ldots, S_0^1(\alpha_n)/\alpha_n]$. We now show that, for each $1 \le i \le n$, $S_0^2(\kappa_i)$ is well formed under $K_0\{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_{i-1} :: S_0^2(\kappa_{i-1})\}$. Since $S_0^1(\kappa_i)$ is well formed under K_0', it is enough to show that $FTV(S_0^2(\kappa_i)) \cap \{\tau_1^0, \ldots, \tau_m^0\} = \emptyset$. Suppose that $\alpha \in FTV(S_0^2(\kappa_i))$. Then there is some α' such that $\alpha \in FTV(S_0^2(\alpha'))$ and $\alpha' \in FTV(\kappa_i)$. Since $\alpha_i \in EFTV(K_1, \tau_1)$, by Definition 2.4.14, $\alpha' \in EFTV(K_1, \tau_1)$. By our assumption on K_1, for any $j \ge i$, $\alpha_j \notin FTV(\kappa_i)$. Therefore, either $\alpha \in \{\alpha_1, \ldots, \alpha_{i-1}\}$, or $\alpha \in S_0^2(EFTV(K_1, S_1(\Gamma))$. But, it can be shown, by induction on the construction of $EFTV(K_1, S_1(\Gamma))$, that $S_0^2(EFTV(K_1, S_1(\Gamma)) \subseteq EFTV(K_0', S_0(\Gamma))$. Thus, $\alpha \notin \{\alpha_1^0, \ldots, \alpha_m^0\}$, and $K_0\{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$ is well formed. By a similar argument, it can also be shown that $S_0^2(\tau_1)$ is well formed under $K_0\{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$. Then $(K_0\{\alpha_1^0 :: \kappa_1^0, \ldots, \alpha_m^0 :: \kappa_m^0\}, S_0^3)$ respects $K_0\{\alpha_1 :: S_0^2(\kappa_1), \ldots, \alpha_n :: S_0^2(\kappa_n)\}$, and $S_0^3 \circ S_0^2(\tau_1) \equiv \tau_0^1$. Thus, $K_0 \vdash_\chi \forall \alpha_1 :: S_0^2(\kappa_1) \cdots \forall \alpha_n :: S_0^2(\kappa_n) . S_0^2(\tau_1) \ge \forall \alpha_1^0 :: \kappa_1^0 \cdots \forall \alpha_m^0 ::$

$\kappa^0_m.\tau^1_0$. Then, by Lemma 4.2.2, $K_0, (S_0(\Gamma))\{x : S^2_0(\kappa_1) \cdots \forall\alpha_n :: S^2_0(\kappa_n).S^2_0(\tau_1)\} \vdash_\chi M_2 : \tau^2_0$. Since $S^2_0(\sigma_1) \equiv S^2_0(\kappa_1) \cdots \forall\alpha_n :: S^2_0(\kappa_n).S^2_0(\tau_1)$ and $S_0(\Gamma) = S^2_0(S_1(\Gamma))$, we have that $K_0, S^2_0((S_1(\Gamma))\{x : \sigma_1\}) \vdash_\chi M_2 : \tau^2_0$. Since (K_0, S^1_0) respects K_1, we have that (K_0, S^2_0) respects K'_1. Since $WK_\chi(K'_1, S_1(\Gamma)\{x : \sigma\}, M_2) = (K_2, S_2, \tau_2)$ and $K_0, S^2_0((S_1(\Gamma))\{x : \sigma_1\}) \vdash_\chi M_2 : \tau^2_0$, we have, by the induction hypothesis, there is a S^4_0 such that (K_0, S^4_0) respects K_2, $S^4_0(\tau_2) \equiv_\chi \tau^2_0$, and $S^4_0 \circ S_2 \circ S_1(\Gamma) = S^2_0 \circ S_1(\Gamma) = S^1_0 \circ S_1(\Gamma) = S_0(\Gamma)$.

- Case $\{l_1 = M_1, \ldots, l_n = M_n\}$: Let $WK_\chi(K, \Gamma, \{l_1 = M_1, \ldots, l_n = M_n\}) = (K_n, S_n \circ \cdots \circ S_2 \circ S_1, \{l_1 =\mid S_n \circ \cdots \circ S_2(\tau_1) \mid, \ldots, l_i :\mid S_n \circ \cdots \circ S_{i+1}(\tau_i) \mid, \ldots, l_n = \tau_n\}$. Also, let $K_0, S_0(\Gamma) \vdash_\chi \{l_1 = M_1, \ldots, l_n = M_n\} : \{l_1 : \tau^1_0, \ldots, l_n : \tau^n_0\}$ and (K_0, S_0) respects K. But, then $K_0, S_0(\Gamma) \vdash_\chi M_i : \tau^i_0 (1 \leq i \leq n)$. By the induction hypothesis, there is a S'_i such that (K_0, S'_i) respects K_i, $S'_i(\tau_i) \equiv_\chi \tau^i_0$, $S_0(\Gamma) = S'_i \circ S_i \circ \cdots \circ S_2 \circ S_1(\Gamma)$. In particular, for $i = n$, $S'_n(\tau_n) \equiv_\chi \tau^n_0$ and $S_0(\Gamma) = S'_n \circ S_n \circ \cdots \circ S_2 \circ S_1(\Gamma)$. Also, note that $S'_n \circ S_n \circ \cdots \circ S_{i+1}(\tau_i) \equiv_\chi \tau^i_0$.

- Case $M.l$: Let $WK_\chi(K, \Gamma, M.l) = (K_2, S_2 \circ S_1, \mid S(\alpha_1) \mid)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi M.l : \tau^1_0$ and (K_0, S_0) respects K. But, then $K_0, S_0(\Gamma) \vdash_\chi M : \tau^2_0$ and $K \vdash_\chi \tau^2_0 :: \{\{l : \tau^1_0 \parallel\}\}$. Since $WK_\chi(K, \Gamma, M) = (K_1, S_1, \tau_1)$ and $K_0, S_0(\Gamma) \vdash_\chi M : \tau^2_0$, we have, by the induction hypothesis, that there is a S'_1 such that (K_0, S'_1) respects K_1, $S'_1(\tau_1) \equiv_\chi \tau^2_0$, and $S_0(\Gamma) = S^1_0 \circ S_1(\Gamma)$. Also, $K_0 \vdash_\chi S'_1(\tau_1) :: \{\{l : \tau^1_0 \parallel\}\}$. Since α_1 and α_2 are fresh, they will not occur in S'_1. Let $S'_2 = [\tau^1_0/\alpha_1, \tau^2_0/\alpha_2] \circ S'_1$. Then $K_0 \vdash_\chi S'_2(\alpha_2) :: \{\{l : S'_2(\alpha_1) \parallel\}\}$ and $K_0 \vdash_\chi S'_2(\alpha_2) :: S'_2(\mathcal{U}) \iff K_0 \vdash_\chi S'_2(\alpha_2) :: \mathcal{U}$. Note that both τ^1_0 and τ^2_0 are well formed under K_1. Therefore (K_0, S'_2) respects $K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}$, since (K_0, S'_2) respects K_1 and $dom(K_1) \cap \{\alpha_1, \alpha_2\} = \emptyset$, and $S'_2(\alpha_2) \equiv \tau^2_0 \equiv_\chi S'_1(\tau_1) \equiv S'_2(\tau_1)$. But, then (K_0, S'_2) is a unifier of $(K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \{(\alpha_2, \tau_1)\})$ and, by Theorem 4.3.1, there is a S'_3 such that (K, S'_3) respects K_2 and $S'_2 = S'_3 \circ S_2$. But, then $S'_3(\mid S_2(\alpha_1) \mid) \equiv_\chi S'_3(S_2(\alpha_1)) \equiv S'_2(\alpha_1) \equiv \tau^1_0$. Also, $S_0(\Gamma) = S^1_0 \circ S_1(\Gamma) = S^2_0 \circ S_1(\Gamma) = S^3_0 \circ S_2 \circ S_1(\Gamma)$.

- Case modify(M_1, l, M_2): Let $WK_\chi(K, \Gamma, \text{modify}(M_1, l, M_2)) = (K_3, S_3 \circ S_2 \circ S_1, \mid S_3(\alpha_2) \mid)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi \text{modify}(M_1, l, M_2) : \tau^1_0$ and (K_0, S_0) respects K. But, then $K_0, S_0(\Gamma) \vdash_\chi M_1 : \tau^1_0$, $K_0, S_0(\Gamma) \vdash_\chi M_2 : \tau^2_0$, and $K_0 \vdash_\chi \tau^1_0 :: \{\{l : \tau^2_0 \parallel\}\}$. Since $WK_\chi(K, \Gamma, M_1) = (K_1, S_1, \tau_1)$ and $K_0, S_0(\Gamma) \vdash_\chi M_1 : \tau^1_0$, we have, by the induction hypothesis, that there is a S'_1 such that (K_0, S'_1) respects K_1, $S'_1(\tau_1) \equiv_\chi \tau^1_0$, and $S_0(\Gamma) = S'_1 \circ S_1(\Gamma)$. But, then $K_0, S'_1 \circ S_1(\Gamma) \vdash_\chi M_2 : \tau^2_0$. Since $WK_\chi(K_1, S_1(\Gamma), M_2) = (K_2, S_2, \tau_2)$ and $K_0, S'_1 \circ S_1(\Gamma) \vdash_\chi M_2 : \tau^2_0$, we have, by the induction hypothesis, that there is a S'_2 such that (K_0, S'_2) respects K_2, $S'_2(\tau_2) \equiv_\chi \tau^2_0$, and $S_0(\Gamma) = S'_2 \circ S_2 \circ S_1(\Gamma)$. Note that $S'_1 = S'_2 \circ S_2$. Since α_1 and α_2 are fresh, they will not appear in S'_2. Let $S'_3 = [\tau^1_0/\alpha_2, \tau^2_0/\alpha_1] \circ S'_2$. Then $K_0 \vdash_\chi S'_3(\alpha_2) :: \{\{l : S'_3(\alpha_1) \parallel\}\}$ and $K_0 \vdash_\chi S'_3(\alpha_1) :: S'_3(\mathcal{U}) \iff K_0 \vdash_\chi S'_3(\alpha_1) :: \mathcal{U}$. Note that τ^1_0 and τ^2_0 are both well formed under K_2. Therefore (K_0, S'_3) respects $K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}$. Also, $S'_3(\alpha_1) \equiv \tau^2_0 \equiv_\chi S'_2(\tau_2) \equiv S'_3(\tau_2)$ and $S'_3(\alpha_2) \equiv \tau^1_0 \equiv_\chi S'_1(\tau_1) \equiv S'_2 \circ S_2(\tau_1) \equiv S'_3 \circ S_2(\tau_1)$. Therefore, (K_0, S'_3) is an unifier of $(K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\})$ and, by Theorem 4.3.1, there is a S'_4 such that (K_0, S'_4) respects K_3, and $S'_3 = S'_4 \circ S_3$. But, then $S'_4(\mid S_3(\alpha_2) \mid) \equiv_\chi S'_4 \circ S_3(\alpha_2) \equiv S'_3(\alpha_2) \equiv \tau^1_0$. Also,

$S_0(\Gamma) = S'_1 \circ S_1(\Gamma) = S'_2 \circ S_2 \circ S_1(\Gamma) = [\tau_0^1/\alpha_2, \tau_0^2/\alpha_1] \circ S'_2 \circ S_2 \circ S_1(\Gamma) = S'_3 \circ S_2 \circ S_1(\Gamma) = S'_4 \circ S_3 \circ S_2 \circ S_1(\Gamma)$.

- Case $M \setminus\setminus l$: Let $WK_\chi(K, \Gamma, M \setminus\setminus l) = (K_2, S_2 \circ S_1, \mid S_2(\alpha_2 - \{l : \alpha_1\}) \mid)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi M \setminus\setminus l : \tau_0^1 - \{l : \tau_0^2\}$ and (K_0, S_0) respect K. But, then $K_0, S_0(\Gamma) \vdash_\chi M : \tau_0^1$ and $K_0 \vdash_\chi \tau_0^1 :: \{\{l : \tau_0^2 \parallel\}\}$. Since $WK_\chi(K, \Gamma, M) = (K_1, S_1, \tau_1)$ and $K_0, S_0(\Gamma) \vdash_\chi M : \tau_0^1$, we have, by the induction hypothesis, that there is a S'_1 such that (K_0, S'_1) respects K_1, $S'_1(\tau_1) \equiv_\chi \tau_0^1$, and $S_0(\Gamma) = S_0^1 \circ S_1(\Gamma)$. But, then $K_0 \vdash_\chi S'_1(\tau_1) :: \{\{l : \tau_0^2 \parallel\}\}$. Since α_1 and α_2 are fresh, they will not appear in S'_1. Let $S'_2 = [\tau_0^1/\alpha_2, \tau_0^2/\alpha_1] \circ S'_1$. Then:

  - $K_0 \Vdash_\chi S'_2(\alpha_2) :: \{\{l : S'_2(\alpha_1) \parallel\}\}$;
  - $K_0 \Vdash_\chi S'_2(\alpha_1) :: S'_2(\mathcal{U}) \iff K_0 \vdash_\chi S'_2(\alpha_1) :: \mathcal{U}$.

  Note that both τ_0^1 and τ_0^2 are well formed under K_1. Thus, (K_0, S'_2) respects $K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}$. Also, $S'_2(\alpha_2) \equiv \tau_0^1 \equiv_\chi S'_1(\tau_1) \equiv S'_2(\tau_1)$. Therefore (K_0, S'_2) is a unifier of $(K_1\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{l : \alpha_1 \parallel\}\}\}, \{(\alpha_2, \tau_1)\})$ and, by Theorem 4.3.1, there is some S'_3 such that (K_0, S'_3) respects K_2 and $S'_2 = S'_3 \circ S_2$. But, then $S'_3(\mid S_2(\alpha_2 - \{l : \alpha_1\}) \mid) \equiv_\chi S'_3 \circ S_2(\alpha_2) - \{l : S'_3 \circ S_2(\alpha_1)\} \equiv S'_2(\alpha_2) - \{l : S'_2(\alpha_1)\} \equiv \tau_0^1 - \{l : \tau_0^2\}$. Also, $S_0(\Gamma) = S'_1 \circ S_1(\Gamma) = S'_2 \circ S_1(\Gamma) = S'_3 \circ S_2 \circ S_1(\Gamma)$.

- Case extend(M_1, l, M_2): Let $WK_\chi(K, \Gamma, \text{extend}(M_1, l, M_2)) = (K_3, S_3 \circ S_2 \circ S_1, \mid S_3(\alpha_2 + \{l : \alpha_1\}) \mid)$. Also, let $K_0, S_0(\Gamma) \vdash_\chi \text{extend}(M_1, l, M_2) : \tau_0^1 + \{l : \tau_0^2\}$ and (K_0, S_0) respects K. But, then $K_0, S_0(\Gamma) \vdash_\chi M_1 : \tau_0^1$, $K_0, S_0(\Gamma) \vdash_\chi M_2 : \tau_0^2$, $K_0 \vdash_\chi \tau_0^1 :: \{\{\parallel l : \tau_0^2\}\}$, and $root(\tau_0^1) \notin FTV(\tau_0^2)$. Since $WK_\chi(K_1, S_1(\Gamma), M_2) = (K_2, S_2, \tau_2)$ and $K_0, S'_1 \circ S_1(\Gamma) \vdash_\chi M_2 : \tau_0^2$, we have, by the induction hypothesis, that the is a S'_2 such that (K_0, S'_2) respects K_2, $S'_2(\tau_2) \equiv_\chi \tau_0^2$, and $S_0(\Gamma) = S'_2 \circ S_2 \circ S_1(\Gamma) \Rightarrow S'_1 = S'_2 \circ S_2$. Since both α_1 and α_2 are fresh, they will not appear in S'_2. Let $S'_3 = [\tau_0^1/\alpha_2, \tau_0^2/\alpha_1] \circ S'_2$. Then $K_0 \vdash_\chi S'_3(\alpha_2) :: \{\{\parallel l : S'_3(\alpha_1)\}\}$ and (K_0, S'_3) respects $K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{\parallel l : \alpha_1\}\}\}$. Also, $S'_3(\alpha_2) \equiv \tau_0^1 \equiv_\chi S'_1(\tau_1) \equiv S'_2 \circ S_2(\tau_1) \equiv S'_3(\tau_1)$ and $S'_3(\alpha_1) \equiv \tau_0^2 \equiv_\chi S'_2(\tau_2) \equiv S'_3(\tau_2)$. Therefore (K_0, S'_2) is a unifier of $(K_2\{\alpha_1 :: \mathcal{U}, \alpha_2 :: \{\{\parallel l : \alpha_1\}\}\}, \{(\alpha_1, \tau_2), (\alpha_2, S_2(\tau_1))\})$ and, by Theorem 4.3.1, there is some S'_4 such that (K_0, S'_4) respects K_3 and $S'_3 = S'_4 \circ S_3$. But, then $S'_4(\mid S_3(\alpha_2 + \{l : \alpha_1\}) \mid) \equiv_\chi S'_4(S_3(\alpha_2 + \{l : \alpha_1\})) \equiv S'_4 \circ S_3(\alpha_2) + \{l : S'_4 \circ S_3(\alpha_1)\} \equiv \tau_0^1 + \{l : \tau_0^2\}$. Also, $S_0(\Gamma) = S'_1 \circ S_1(\Gamma) = S'_2 \circ S_2 \circ S_1(\Gamma) = [\tau_0^1/\alpha_2, \tau_0^2/\alpha_1] \circ S'_2 \circ S_2 \circ S_1(\Gamma) = S'_3 \circ S_2 \circ S_1(\Gamma) = S'_4 \circ S_3 \circ S_2 \circ S_1(\Gamma)$.

Finally, let us assume that the algorithm fails for some term. Then it is easy to see that there is no typing for that term. □

Example 4.3.4.

$$
\begin{aligned}
1)\ & WK_\chi(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \{x : \alpha_1, y : \alpha_2\}, \mathrm{extend}(x, l, y).l) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_5\}\}, \alpha_5 :: \mathcal{U}\}, [\alpha_5/\alpha_2, \alpha_1/\alpha_6, \alpha_5/\alpha_3, \alpha_1/\alpha_4], \alpha_5) \\
1.1)\ & WK_\chi(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \{x : \alpha_1, y : \alpha_2\}, \mathrm{extend}(x, l, y)) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, [\alpha_2/\alpha_3, \alpha_1/\alpha_4], \alpha_1) \\
1.1.1)\ & WK_\chi(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \{x : \alpha_1, y : \alpha_2\}, x) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \mathit{id}, \alpha_1) \\
1.1.2)\ & WK_\chi(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \{x : \alpha_1, y : \alpha_2\}, x) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, \mathit{id}, \alpha_2) \\
1.1.3)\ & \mathcal{U}(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}, \alpha_3 :: \mathcal{U}, \alpha_4 :: \{\{\| \, l : \alpha_3\}\}\}, \{(\alpha_3, \alpha_2), (\alpha_4, \alpha_1)\}) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\}, [\alpha_2/\alpha_3, \alpha_1/\alpha_4]) \\
1.2)\ & \mathcal{U}(\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}, \alpha_5 :: \mathcal{U}, \alpha_6 :: \{\{\| \, l : \alpha_5\}\}\}, \{(\alpha_6, \alpha_1)\}) \\
& = (\{\alpha_1 :: \{\{\| \, l : \alpha_5\}\}, \alpha_5 :: \mathcal{U}\}, [\alpha_5/\alpha_2, \alpha_1/\alpha_6])
\end{aligned}
$$

Assuming the typing environment

$$\{x : \alpha_1, y : \alpha_2\}$$

and the kinding environment

$$\{\alpha_1 :: \{\{\| \, l : \alpha_2\}\}, \alpha_2 :: \mathcal{U}\},$$

the principal typing of $\mathrm{extend}(x, l, y).l$ is $(\{\alpha_1 :: \{\{\| \, l : \alpha_5\}\}, \alpha_5 :: \mathcal{U}\}, \alpha_5)$.

In this chapter, we have presented an ML-style calculus with with extensible records and developed both a typing system based on the notion of kinded quantification and a sound and complete kinded type inference algorithm based on kinded unification. This will allow us to add extensible records to `EVL`.

Chapter 5

Applications

In this chapter we illustrate some applications of the EVL programming language in the context of CEP and specification of obligation policies and also present some of the applications of the ML-style calculus with extensible records presented in Chapter 4 for CEP.

5.1 Events

In event processing applications, many events have a similar structure and a similar meaning. As an example, consider a temperature sensor. On the one hand, all of the events produced by a temperature sensor have the same kind of information, such as the temperature reading, a timestamp and its location. On the other hand, it is most probable that each reading results in different readings. This relationship was also formally defined in [2] as that between ***Generic*** and ***Specific*** events.

We will consider events as particular actions or happenings that occur at a particular time.

Definition 5.1.1 (Event Specifications)**.** Given a set of terms $M_1, \ldots, M_n$, defined in a particular language, an ***event specification***, denoted spec, is a term of the form $\{l_1 = M_1, \ldots, l_n = M_n\},\ n > 0$, representing a structure with labels $l_1, \ldots, l_n$ and values $M_1, \ldots, M_n$ respectively. An event specification without term variable occurrences is called a ground event specification.

We distinguish between ***events*** and ***generic events*** (or event schemes). The former correspond to specific happenings or occurrences and the latter represent sets of events that can occur in a particular system.

Definition 5.1.2 (Event)**.** A ***(specific) event*** is a ground event specification that represents a particular action/happening, occurring in a system.

Definition 5.1.3 (Generic Event)**.** A ***generic event*** (or ***event scheme***) represents a set of events, defined as $ge[x_1, \ldots, x_n] =$ spec, where $x_1, \ldots, x_n$ are the variables occurring in spec.

Note that a specific event e is associated to a generic event ge by an instantiation relation $\vdash_\theta e :: ge$ and that this relation can either be syntactic (e is obtained from ge by replacing the variables in ge by terms through a substitution mapping θ) or semantic (in which case, instantiation may require some computation).

5.1.1 Complex Event Processing

Complex Event Processing (CEP), or simply event processing, refers to a set of techniques used to deal with event streams that include event identification, classification, and response. A variety of languages to process events have been developed over the years [8, 26, 43]. Event processing is a key component of Internet-of-Things applications, which need to identify and react to events in streams of data generated by sensors. In critical domains such as healthcare, languages with a formal semantics are particularly valuable because there is a need to prove properties such as correctness, security, and safety of applications. In the context of security and, in particular, when modelling access control, it is often the case that granting or denying access to certain resources depends on the occurrence of a particular event [9, 10, 35]. In fact, this is even more crucial in access control systems dealing with obligations. In these systems, the status of a particular obligation is usually defined in terms of occurrences of events in the system and several models that deal with obligations have to deal in some way with the notion of an event.

The area of CEP comprises a series of techniques to deal with streams of events such as event processing, detection of patterns and relationships, filtering, transformation and abstraction, amongst others. See [26] for a detailed reference on the area.

Event processing agents are classified according to the actions that they perform to process incoming events [26].

Definition 5.1.4 (Filter event processing agent)**.** A ***filter agent*** is an event processing agent that performs filtering only, so it does not transform the input event.

Definition 5.1.5 (Transformation event processing agent)**.** A ***transformation agent*** is an event processing agent that includes a derivation step, and optionally also a filtering step.

Definition 5.1.6 (Translate event processing agent)**.** A ***translate agent*** can be used to convert events from one type to another, or to add, remove, or modify the values of an event's attributes.

Definition 5.1.7 (Aggregate event processing agent)**.** An ***aggregate agent*** takes a stream of incoming events and produces an output event that is a map of the incoming events.

Definition 5.1.8 (Compose event processing agent)**.** A ***compose agent*** takes two streams of incoming events and processes them to produce a single output stream of events.

Definition 5.1.9 (Pattern Detect event processing agent)**.** A ***pattern detect agent*** performs a pattern matching function on one or more input streams. It emits one or more derived events if it detects an occurrence of the specified pattern in the input events.

The notions of specific and generic events are also a key aspect in CEP, where instead of defining the structure of each event individually, one wants to be able to specify the structure of an entire class of events. Generic events can then be related to other (generic or specific) events through semantic relations.

5.2 EVL for Event Processing

In EVL, events are represented using records typed with record types. In this section, we are going to present some examples that illustrate how EVL can be used for event processing.

5.2.1 CEP

In this section we explore the higher-order capabilities of EVL to define higher-order parameterised functions that deal with the usual CEP techniques. The canonical model [17, 26] for event processing is based on a producer-consumer model: an event processing agent (EPA) takes events from event producers and distributes them among event consumers. Using EVL we are able to process raw events produced by some event processing system and to generate derived events that can then be passed on to an event consumer as a result.

5.2.1.1 Event Processing Agents

Below we give some examples that show how the standard types of EPAs can be defined in EVL. An event processing agent is any function whose principal type is of the form $\forall\alpha_1 :: \kappa_1 \cdots \forall\alpha_n :: \kappa_n.\gamma$.

Filter agents take an incoming event object and apply a test to decide whether to discard it or whether to pass it on for processing by subsequent agents. The test is usually stateless, i.e. based solely on the content of the event instance.

Example 5.2.1. This example represents an event processing agent that uses a higher-order filter function `filter` to filter events according to their location.

$$\text{let } p = \lambda x.(x.\text{location}) == \texttt{"Porto"}^{String} \text{ in } (\text{filter } p)$$

Assuming that

$$\text{filter} : \forall\alpha_1 :: \mathcal{U}.\forall\alpha_2 :: \{\{\text{empty} : Bool\}\}.\forall\alpha_3 :: \{\{\text{empty} : Bool, \text{head} : \alpha_1, \text{tail} : \alpha_2\}\}.(\alpha_1 \to Bool) \to \alpha_3 \to \alpha_2$$

and

$$(\ ==\ ) : String \to String \to Bool,$$

its principal typing is

$$(\{\alpha_1 :: \{\{\text{empty} : Bool, \text{head} : \alpha_2, \text{tail} : \alpha_3\}\}, \alpha_2 :: \{\{\text{location} : String\}\}, \alpha_3 :: \{\{\text{empty} : Bool\}\}\}, \alpha_1 \to \alpha_3).$$

Let

$$\begin{aligned}
M &\equiv \lambda \mathit{self}.\lambda p.\lambda \mathit{list}.\text{if } \mathit{list}.\text{empty then } \mathit{list} \\
&\qquad\qquad \text{else if } (p\ \mathit{list}.\text{head}) \text{ then cons } \mathit{list}.\text{head } (\mathit{self}\ \mathit{self}\ p\ \mathit{list}.\text{tail}) \\
&\qquad\qquad\quad \text{else } \mathit{self}\ \mathit{self}\ p\ \mathit{list}.\text{tail}, \\
P &\equiv \lambda x.(x.\text{location}) == \texttt{"Porto"}^{\mathit{String}}, \\
N_0 &\equiv \{\text{empty} = \texttt{True}^{\mathit{Bool}}\} \\
N_1 &\equiv \{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = \{\text{location} = \text{tail} =\}, \text{tail} = N_0\},
\end{aligned}$$

and

$$N_2 \equiv \{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}, \text{tail} = N_1\}.$$

Assuming the usual operational semantics for (==), we can evaluate the following program using the operational semantics defined for the EVL language in Chapter 3 as follows:

$$\begin{aligned}
&\mathcal{E}[\text{let filter} = M \text{ in let } p = P \text{ in filter filter } p\ N_2] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{let } p = P \text{ in } M\ M\ p\ N_2] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[M\ M\ P\ N_2] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda p.\lambda \mathit{list}.\text{if } \mathit{list}.\text{empty then } \mathit{list} \\
&\qquad\qquad \text{else if } (p\ \mathit{list}.\text{head}) \text{ then cons } \mathit{list}.\text{head } (M\ M\ p\ \mathit{list}.\text{tail}) \text{ else } M\ M\ p\ \mathit{list}.\text{tail})\ P\ N_2] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda \mathit{list}.\text{if } \mathit{list}.\text{empty then } \mathit{list} \\
&\qquad\qquad \text{else if } (P\ \mathit{list}.\text{head}) \text{ then cons } \mathit{list}.\text{head } (M\ M\ P\ \mathit{list}.\text{tail}) \text{ else } M\ M\ P\ \mathit{list}.\text{tail})\ N_2] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } N_2.\text{empty then } N_2 \\
&\qquad \text{else if } (P\ N_2.\text{head}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail})] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } \texttt{False}^{\mathit{Bool}} \text{ then } N_2 \\
&\qquad \text{else if } (P\ N_2.\text{head}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (P\ N_2.\text{head}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (P\ N_2.\text{head}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (P\ \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (\{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}.\text{location} == \texttt{"Porto"}^{\mathit{String}}) \\
&\qquad \text{then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (\texttt{"Porto"}^{\mathit{String}} == \texttt{"Porto"}^{\mathit{String}}) \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \\
&\qquad \text{else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } \texttt{True}^{\mathit{Bool}} \text{ then cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail}) \text{ else } M\ M\ P\ N_2.\text{tail}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } N_2.\text{head } (M\ M\ P\ N_2.\text{tail})] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}\ (M\ M\ P\ N_2.\text{tail})] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}\ (M\ M\ P\ N_1)] \\
&\Rightarrow_{\mathcal{E}} \cdots \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}\ (M\ M\ P\ N_1.\text{tail})] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}\ (M\ M\ P\ N_0)] \\
&\Rightarrow_{\mathcal{E}} \cdots \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{cons } \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}\ \{\text{empty} = \texttt{True}^{\mathit{Bool}}\}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}, \text{tail} = \{\text{empty} = \texttt{True}^{\mathit{Bool}}\}\}] \\
&\Rightarrow_{\mathcal{E}} \{\text{empty} = \texttt{False}^{\mathit{Bool}}, \text{head} = \{\text{location} = \texttt{"Porto"}^{\mathit{String}}\}, \text{tail} = \{\text{empty} = \texttt{True}^{\mathit{Bool}}\}\}
\end{aligned}$$

Transformation agents can be either stateless (if events are processed without taking into account preceding or following events) or stateful (if the way events are processed is influenced by

preceding or following events). In the former case, events are processed individually. In the latter, the way events are processed can depend on preceding or succeeding events. Transformation events can be further classified as translate, split, aggregate or compose agents.

We now give some examples of transformation agents written in EVL.

Example 5.2.2. This example represents a translate event processing agent that converts the temperature field of an event from degrees Fahrenheit to degrees Celsius.

$$\text{farToCel} \equiv \lambda x.\text{modify}(x, \text{temperature}, ((x.\text{temperature}) \text{ - } 32.0^{Float}) \text{ / } 1.8^{Float}))$$

Assuming that

$$(\text{ - }) : Float \to Float \to Float$$

and

$$(\text{ / }) : Float \to Float \to Float,$$

its principal typing is

$$(\{\alpha_1 :: \{\{\text{temperature} : Float\}\}\}, \alpha_1 \to \alpha_1).$$

Assuming the usual operational semantics for the (-) and (/) operators, we can evaluate the following program using the operational semantics defined for the EVL language in Chapter 3 as follows:

$$\begin{aligned}
&\mathcal{E}[\lambda x.\text{modify}(x, \text{temperature}, ((x.\text{temperature}) \text{ - } 32.0^{Float}) \text{ / } 1.8^{Float}))\ \{\text{temperature} = 50.0^{Float}\}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{modify}(\{\text{temperature} = 50.0^{Float}\}, \text{temperature}, \\
&\qquad\qquad (((\{\text{temperature} = 50.0^{Float}\}.\text{temperature}) \text{ - } 32.0^{Float}) \text{ / } 1.8^{Float}))] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{modify}(\{\text{temperature} = 50.0^{Float}\}, \text{temperature}, ((50.0^{Float} \text{ - } 32.0^{Float}) \text{ / } 1.8^{Float}))] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\{\text{temperature} = 10.0^{Float}\}] \\
&\Rightarrow_{\mathcal{E}} \{\text{temperature} = 10.0^{Float}\}
\end{aligned}$$

Example 5.2.3. This example represents an aggregate event processing agent that receives two events, x and y, and outputs event y with its precipitation level updated with the average of the two.

$$\text{avgPrecip} \equiv \lambda xy.\text{modify}(y, \text{precipitation}, ((x.\text{precipitation}) + (y.\text{precipitation})) \text{ / } 2.0^{Float}))$$

Assuming that

$$(\text{ + }) : Float \to Float \to Float,$$

its principal typing is

$$(\{\alpha_1 :: \{\{\text{precipitation} : Float\}\}, \alpha_2 :: \{\{\text{precipitation} : Float\}\}\}, \alpha_1 \to \alpha_2 \to \alpha_2).$$

Assuming the usual operational semantics for the (+) and (/) operators, we can evaluate the following program using the operational semantics defined for the `EVL` language as follows:

$$
\begin{aligned}
&\mathcal{E}[\lambda xy.\text{modify}(y, \text{precipitation}, ((x.\text{precipitation}) + (y.\text{precipitation})) \ / \ 2.0^{Float}))\\
&\qquad \{\text{precipitation} = 10.0^{Float}\}\ \{\text{precipitation} = 20.0^{Float}\}]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\lambda y.\text{modify}(y, \text{precipitation},\\
&\qquad (((\{\text{precipitation} = 20.0^{Float}\}.\text{precipitation}) +\\
&\qquad\ (y.\text{precipitation})) \ / \ 2.0^{Float}))\ \{\text{precipitation} = 10.0^{Float}\}]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{modify}(\{\text{precipitation} = 10.0^{Float}\}, \text{precipitation},\\
&\qquad (((\{\text{precipitation} = 20.0^{Float}\}.\text{precipitation}) +\\
&\qquad\ (\{\text{precipitation} = 10.0^{Float}\}.\text{precipitation})) \ / \ 2.0^{Float}))]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{modify}(\{\text{precipitation} = 10.0^{Float}\}, \text{precipitation}, ((20.0^{Float}) + (10.0^{Float}) \ / \ 2.0^{Float}))]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{modify}(\{\text{precipitation} = 10.0^{Float}\}, \text{precipitation}, 15.0^{Float})]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\{\text{precipitation} = 15.0^{Float}\}]\\
&\Rightarrow_{\mathcal{E}} \{\text{precipitation} = 15.0^{Float}\}
\end{aligned}
$$

Example 5.2.4. This example represents an event processing agent that composes the partial weather information that is provided by two different sensors. One of the sensors outputs event x, which contains information about the temperature and wind velocity, and the other sensor outputs event y, which contains information about the humidity and precipitation levels. This event processing agent outputs an instance of `WeatherInfo` with the complete weather information.

$$\text{composeInfo} \equiv \lambda xy.\text{WeatherInfo}\ x.\text{temperature}\ x.\text{wind}\ y.\text{humidity}\ y.\text{precipitation}$$

Assuming that

$$
\begin{aligned}
&\text{WeatherInfo}:\\
&\quad \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \mathit{Float} \to \{\text{temperature} : \mathit{Float}, \text{wind} : \mathit{Float}, \text{humidity} : \mathit{Float}, \text{precipitation} : \mathit{Float}\},
\end{aligned}
$$

its principal typing is

$$
\begin{aligned}
&(\{\alpha_1 :: \{\{\text{temperature} : \mathit{Float}, \text{wind} : \mathit{Float}\}\}, \alpha_2 :: \{\{\text{humidity} : \mathit{Float}, \text{precipitation} : \mathit{Float}\}\}\},\\
&\quad \alpha_1 \to \alpha_2 \to \{\text{temperature} : \mathit{Float}, \text{wind} : \mathit{Float}, \text{humidity} : \mathit{Float}, \text{precipitation} : \mathit{Float}\}).
\end{aligned}
$$

Let $M \equiv \{\text{temperature} = 120.0^{Float}, \text{wind} = 40.0^{Float}\}$ and $N \equiv \{\text{humidity} = 70.0^{Float}, \text{precipitation} = 10.0^{Float}\}$. We can evaluate the following program using the operational semantics defined for the `EVL` language in Chapter 3 as follows:

$$
\begin{aligned}
&\mathcal{E}[\text{letEv WeatherInfo}\\
&\qquad = \lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}\\
&\quad \text{in } (\lambda xy.\text{WeatherInfo}\ x.\text{temperature}\ x.\text{wind}\ \ y.\text{humidity}\ y.\text{precipitation})\ M\ N]\\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda xy.(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\})\\
&\qquad x.\text{temperature}\ x.\text{wind}\ \ y.\text{humidity}\ y.\text{precipitation})\ M\ N] \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\cdots \Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda y.(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad M.\text{temperature}\ M.\text{wind}\ \ y.\text{humidity}\ y.\text{precipitation})\ N] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad M.\text{temperature}\ M.\text{wind}\ \ N.\text{humidity}\ N.\text{precipitation})] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 120.0^{Float}\ M.\text{wind}\ \ N.\text{humidity}\ N.\text{precipitation}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 120.0^{Float}\ 40.0^{Float}\ \ N.\text{humidity}\ N.\text{precipitation}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 120.0^{Float}\ 40.0^{Float}\ \ 70.0^{Float}\ N.\text{precipitation}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_1 x_2 x_3 x_4.\{\text{temperature} = x_1, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 120.0^{Float}\ 40.0^{Float}\ \ 70.0^{Float}\ 10.0^{Float}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_2 x_3 x_4.\{\text{temperature} = 120.0^{Float}, \text{wind} = x_2, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 40.0^{Float}\ \ 70.0^{Float}\ 10.0^{Float}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_3 x_4.\{\text{temperature} = 120.0^{Float}, \text{wind} = 40.0^{Float}, \text{humidity} = x_3, \text{precipitation} = x_4\}) \\
&\qquad 70.0^{Float}\ 10.0^{Float}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x_4.\{\text{temperature} = 120.0^{Float}, \text{wind} = 40.0^{Float}, \text{humidity} = 70.0^{Float}, \text{precipitation} = x_4\}) \\
&\qquad 10.0^{Float}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\{\text{temperature} = 120.0^{Float}, \text{wind} = 40.0^{Float}, \text{humidity} = 70.0^{Float}, \text{precipitation} = 10.0^{Float}\}] \\
&\Rightarrow_{\mathcal{E}} \{\text{temperature} = 120.0^{Float}, \text{wind} = 40.0^{Float}, \text{humidity} = 70.0^{Float}, \text{precipitation} = 10.0^{Float}\}
\end{aligned}
$$

Pattern Detect agents take collections of incoming event objects and examine them to see if they can spot the occurrence of particular patterns.

Example 5.2.5. The following function is an event processing agent that generates an instance of the `FireDanger` event depending on the weather conditions.

$$
\begin{aligned}
\text{checkWeather} \equiv \lambda x.\text{if}\ (&x.\text{temperature} > 29.0^{Float}\ \text{and}\ x.\text{wind} > 32.0^{Float} \\
&\text{and}\ x.\text{humidity} < 20.0^{Float}\ \text{and}\ x.\text{precipitation} < 50.0^{Float}) \\
&\text{then FireDanger}\ x.\text{location}\ \texttt{"high"}^{String} \\
&\text{else FireDanger}\ x.\text{location}\ \texttt{"low"}^{String}
\end{aligned}
$$

Assuming that

$$
\begin{aligned}
> &: \mathit{Float} \to \mathit{Float} \to \mathit{Bool}, \\
< &: \mathit{Float} \to \mathit{Float} \to \mathit{Bool}, \\
\text{and} &: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool},
\end{aligned}
$$

and

$$
\text{FireDanger} : \mathit{String} \to \mathit{String} \to \{\text{location} : \mathit{String}, \text{danger} : \mathit{String}\},
$$

its principal typing is

$$(\{\alpha_1 :: \{\{\text{temperature} : \mathit{Float}, \text{wind} : \mathit{Float}, \text{humidity} : \mathit{Float}, \text{precipitation} : \mathit{Float}\}\}, \text{location} : \mathit{String}\},$$
$$\alpha_1 \to \{\text{location} : \mathit{String}, \text{danger} : \mathit{String}\}).$$

Let $M \equiv \{\text{temperature} = 30.0^{\mathit{Float}}, \text{wind} = 33.0^{\mathit{Float}}, \text{humidity} = 18.0^{\mathit{Float}}, \text{precipitation} = 10.0^{\mathit{Float}}, \text{location} = \texttt{"Porto"}^{\mathit{String}}\}$. Assuming the usual operational semantics for the ($<$), ($>$) and logical (and) operators, we can evaluate the following program using the operational semantics defined for the EVL language in Chapter 3 as follows:

$$\begin{aligned}
&\mathcal{E}[\text{letEv FireDanger} = \lambda xy.\{\text{location} = x, \text{danger} = y\} \\
&\quad \text{in } (\lambda x.\text{if } (x.\text{temperature} > 29.0^{\mathit{Float}} \text{ and } x.\text{wind} > 32.0^{\mathit{Float}} \\
&\qquad \text{and } x.\text{humidity} < 20.0^{\mathit{Float}} \text{ and } x.\text{precipitation} < 50.0^{\mathit{Float}}) \\
&\qquad \text{then FireDanger } x.\text{location } \texttt{"high"}^{\mathit{String}} \\
&\qquad \text{else FireDanger } x.\text{location } \texttt{"low"}^{\mathit{String}})\ M] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda x.\text{if } (x.\text{temperature} > 29.0^{\mathit{Float}} \text{ and } x.\text{wind} > 32.0^{\mathit{Float}} \\
&\qquad \text{and } x.\text{humidity} < 20.0^{\mathit{Float}} \text{ and } x.\text{precipitation} < 50.0^{\mathit{Float}}) \\
&\qquad \text{then } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ x.\text{location } \texttt{"high"}^{\mathit{String}} \\
&\qquad \text{else } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ x.\text{location } \texttt{"low"}^{\mathit{String}})\ M] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (M.\text{temperature} > 29.0^{\mathit{Float}} \text{ and } M.\text{wind} > 32.0^{\mathit{Float}} \\
&\qquad \text{and } M.\text{humidity} < 20.0^{\mathit{Float}} \text{ and } M.\text{precipitation} < 50.0^{\mathit{Float}}) \\
&\qquad \text{then } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"high"}^{\mathit{String}} \\
&\qquad \text{else } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"low"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } (30.0^{\mathit{Float}} > 29.0^{\mathit{Float}} \text{ and } 33.0^{\mathit{Float}} > 32.0^{\mathit{Float}} \\
&\qquad \text{and } 18.0^{\mathit{Float}} < 20.0^{\mathit{Float}} \text{ and } 10.0^{\mathit{Float}} < 50.0^{\mathit{Float}}) \\
&\qquad \text{then } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"high"}^{\mathit{String}} \\
&\qquad \text{else } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"low"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\text{if } \texttt{True}^{\mathit{Bool}} \text{ then } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"high"}^{\mathit{String}} \\
&\qquad \text{else } (\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"low"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda xy.\{\text{location} = x, \text{danger} = y\})\ M.\text{location } \texttt{"high"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda xy.\{\text{location} = x, \text{danger} = y\})\ \texttt{"Porto"}^{\mathit{String}}\ \texttt{"high"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[(\lambda y.\{\text{location} = \texttt{"Porto"}^{\mathit{String}}, \text{danger} = y\})\ \texttt{"high"}^{\mathit{String}}] \\
&\Rightarrow_{\mathcal{E}} \mathcal{E}[\{\text{location} = \texttt{"Porto"}^{\mathit{String}}, \text{danger} = \texttt{"high"}^{\mathit{String}}\}] \\
&\Rightarrow_{\mathcal{E}} \{\text{location} = \texttt{"Porto"}^{\mathit{String}}, \text{danger} = \texttt{"high"}^{\mathit{String}}\}
\end{aligned}$$

5.2.1.2 A Higher-Order Library for CEP

Since EVL is a higher-order language, we use higher-order functions that allow us to deal with a sequence of events (represented as a list of events). We now provide some of these useful higher-order functions, which are naturally implemented in a higher-order functional language.

- `filter` is a function that filters the events in the sequence according to some filtering

expression:

$$\begin{aligned}\text{filter} \equiv \lambda p.\lambda \mathit{list}.&\text{if } \mathit{list}.\text{empty then } \mathit{list}\\ &\text{else if } (p\ \mathit{list}.\text{head})\\ &\quad\text{then cons } \mathit{list}.\text{head } (\text{filter } p\ \mathit{list}.\text{tail})\\ &\quad\text{else filter } p\ \mathit{list}.\text{tail}\end{aligned}$$

- `transform` is a function that applies a transformation to all of the events in the sequence:

$$\begin{aligned}\text{transform} \equiv \lambda f.\lambda \mathit{list}.&\text{if } \mathit{list}.\text{empty then } \mathit{list}\\ &\text{else cons } (f\ \mathit{list}.\text{head}) \ (\text{transform } f\ \mathit{list}.\text{tail})\end{aligned}$$

- `aggregater` is a function that produces some output value by aggregating by right association the events of the sequence according to some binary aggregating function:

$$\begin{aligned}\text{aggregater} \equiv \lambda f z.&\text{if } \mathit{list}.\text{empty then } z\\ &\text{else } f\ \mathit{list}.\text{head } (\text{aggregater } f\ z\ \mathit{list}.\text{tail})\end{aligned}$$

- `aggregatel` is very similar to `aggregater` but it aggregates the events by left association:

$$\begin{aligned}\text{aggregater} \equiv \lambda f z.&\text{if } \mathit{list}.\text{empty then } z\\ &\text{else aggregatel } f\ (f\ z\ \mathit{list}.\text{head})\ \mathit{list}.\text{tail}\end{aligned}$$

Note that we cannot assign types to the higher-order functions because they are recursive and we do not have recursive types on our type-system. For Example 5.2.1 we can assume a non-recursive type for filter simply because of the way we decided to encode lists using records. However, we cannot infer a type for `filter`. In fact, if we were to implement the `filter` function in a language that supported recursive types, we would either have to use some kind of `letrec` or to make explicit the recursive nature of the function's definition as was done in Example 5.2.1.

The following example is intended to illustrate several features described in this section.

Example 5.2.6. Consider a sequence of events produced by sensors distributed across some number of locations. The events produced by a particular sensor contains information about the weather conditions at that sensor's location. More specifically, it contains information about the temperature (in degrees Celsius), the humidity level (as a percentage), the wind speed (in km/h) and the amount of precipitation (in mm), as well as information about its location. Now, consider an EPA that infers the fire danger of a particular location based on a given sequence of events produced by an arbitrary number of these sensors. This can be done with varying degrees of accuracy, but this is not the subject of this paper, so let us consider a simple algorithm based on the following three steps:

1. Filtering the events according to the specified location;
2. Aggregating the events according to the latest values of temperature, humidity and wind speed, and by the mean precipitation;
3. Producing an event that indicates if there is fire danger in that particular location considering the values obtained in the previous step and comparing them to their threshold levels.

We now provide an implementation of this algorithm in EVL:

$$
\begin{aligned}
&\text{letEv FireDanger} = \lambda l.\lambda d.\{\text{location} = l, \text{danger} = d\}\\
&\text{in let } p = \lambda x.x.\text{location} == \texttt{"Porto"}^{String}\\
&\quad \text{in let } f = \lambda xy.(x.\text{fst} + 1^{Int},\\
&\qquad\qquad \text{modify}(y, \text{precipitation}, (x.\text{snd.precipitation} + y.\text{precipitation}) \ / \ x.\text{fst}))\\
&\qquad \text{in let checkWeather} = \text{if } (x.\text{temperature} > 29.0^{Float} \text{ and } x.\text{wind} > 32.0^{Float}\\
&\qquad\qquad\qquad \text{and } x.\text{humidity} < 20.0^{Float} \text{ and } x.\text{wind} < 50.0^{Float})\\
&\qquad\qquad\qquad \text{then FireDanger } x.\text{location } \texttt{"high"}^{String}\\
&\qquad\qquad\qquad \text{else FireDanger } x.\text{location } \texttt{"low"}^{String}\\
&\qquad\quad \text{in } \lambda x.\text{checkWeather (aggregatel } f \ (1^{Int}, \{\text{precipitation} = 0\}) \ (\text{filter } p \ x)).\text{snd}
\end{aligned}
$$

5.2.1.3 Typing Relations On Events

We now discuss the different semantic relations between events, which are captured by EVL's type-system.

Specialization The specialization relation indicates that an event is a specialization of another event.

In the type theory of EVL, this relation is given by the instantiation relation $\geq$, given in Definition 2.4.15.

Example 5.2.7. Let ge_1 be a generic event with type $\forall\alpha :: \mathcal{U}\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma$ and ge_2 be a generic event with type $\forall\gamma :: \{\{l_1 : Int\}\}.\gamma$. Then ge_2 is specialization of ge_1 since

$$\emptyset \Vdash \forall\alpha :: \mathcal{U}\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma \geq \forall\gamma :: \{\{l_1 : Int\}\}.\gamma.$$

Generalization The generalization relation indicates that an event is a generalization of another event.

In the type theory of EVL, this relation is also given by the instantiation relation $\geq$, given in Definition 2.4.15.

Example 5.2.8. Let ge_1 be a generic event with type $\forall\alpha :: \mathcal{U}.\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma$ and ge_2 be a generic event with type $\forall\gamma :: \{\{l_1 : Int\}\}.\gamma$. Then ge_1 is generalization of ge_2 since

$$\emptyset \Vdash \forall\alpha :: \mathcal{U}\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma \geq \forall\gamma :: \{\{l_1 : Int\}\}.\gamma.$$

Membership A generic event ge_1 is said to be a member of another generic event ge_2 if the instances of ge_1 are included in the instances of ge_2.

In EVL this notion is verified by the instantiation relation given in Definition 2.4.15, as well.

Let $ge_1 : \sigma_1$ and $ge_2 : \sigma_2$ be such that σ_1 and σ_2 are well-formed under the kinding environment K. We say that ge_1 is a member of ge_2 if, for all $\sigma \in \mathbb{T}_{\text{EVL}}$, if $K \Vdash_{\mathcal{O}} \sigma_1 \geq \sigma$, then $K \Vdash \sigma_2 \geq \sigma$.

Example 5.2.9. Let ge_1 be a generic event with type $\forall\alpha :: \mathcal{U}.\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma$ and ge_2 be a generic event with type $\forall\gamma :: \{\{l_1 : Int\}\}.\gamma$. Then ge_2 is a member of ge_1 since for all σ, if

$$\emptyset \Vdash \forall\gamma :: \{\{l_1 : Int\}\}.\gamma \geq \sigma,$$

then

$$\emptyset \Vdash \forall\alpha :: \mathcal{U}\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma \geq \sigma.$$

But note that ge_1 is not a member of ge_2 since, for $\sigma = \{l_1 : Float\}$,

$$\emptyset \Vdash \forall\alpha :: \mathcal{U}\forall\gamma :: \{\{l_1 : \alpha\}\}.\gamma \geq \{l_1 : Float\},$$

but

$$\emptyset \Vdash \forall\gamma :: \{\{l_1 : Int\}\}.\gamma \not\geq \{l_1 : Float\}.$$

Note that, by Lemma 2.4.2, if a generic event is a *specialization* of another event, then it is also a *member* of that event.

Retraction A retraction event relationship is a property of an event referencing a second event. It indicates that the second event is a logical reversal of the event type that references it. For example, an event that starts a fire alert and the event that stops it. Unlike the previous notions, retraction is not directly addressed by EVL. Retraction is a notion that is also present in access control systems that deal with obligations, where the correct treatment of events is crucial. We will briefly discuss the treatment of events in obligation models in the next section.

5.2.2 Event Processing in Obligation Models

The notion of event and an adequate processing of events is essential to the treatment of obligations in access control models. Obligations are usually associated with some mandatory action that must be performed at a time defined by some temporal constraints or by the occurrence of an event. The Category Based Metamodel for Access Control with Obligations (CBACO) [3] defines an obligation as a tuple $o = (a, r, ge_1, ge_2)$, where a is an action, r a resource, and ge_1, ge_2 two generic events (ge_1 triggers the obligation, and ge_2 ends it). The model relies on two additional relations on events:

- ***Event Instantiation***: denoted $e :: ge$, meaning that the event e is an instance of ge, according to an instance relation between events and generic events.
- ***Event Interval***: denoted $(e_1, e_2, h))$, meaning that the event e_2 closes the interval started by the event e_1 in an history of events h.

As we have discussed in the previous section, the notion of event instantiation is captured by the type-system of EVL. With respect to event intervals, this notion is closely related to the notion of retraction in CEP and was addressed in [2] by the definition of a closing function that describes how events are linked to subsequent events in history. These functions are assumed to be defined for each system and are used to extract intervals from a given history. One of the motivations to develop EVL was to provide a simple language in which we can program such functions.

5.2.3 CEP using Extensible Records

Although EVL allows us to adequately deal with the notion of generic events and has the potential of providing a formal semantics to Complex Event Processing (CEP) systems, the lack of support for extensibility is one of its limitations. In fact, the ability to extend a record with a new field or remove an existing field from a record is often very useful for the processing of complex events. In Chapter 4 we address this limitation and develop a polymorphic record calculus with extensible records, i.e., records that can have new fields added to them, or preexisting fields removed from them. This ML-style calculus still allows us to represent polymorphic versions of various types of record operations such as field selection and modification, but is also powerful enough to represent field addition and removal. In this section we provide some examples of the use of these operations in the context of CEP.

Example 5.2.10. This example represents a translate event processing agent that converts the Fahrenheit field to Celsius and adds that new temperature to the event.

$$\text{addFarCel} \equiv \lambda x.\text{extend}(x, \text{celsius}, (x.\text{fahrenheit}) \text{ - } 32.0^{Float}) \text{ / } 1.8^{Float})$$

Example 5.2.11. This example represents a compose event processing agent that receives two events, x and y, and outputs event y with addition of a field with the average precipitation level of the two.

$$\begin{aligned}\text{addAvgPrecip} \equiv &\lambda xy.\text{extend}(y, \text{avg\_precipitation}, \\ &(x.\text{precipitation}) + (y.\text{precipitation})) \text{ / } 2.0^{Float}))\end{aligned}$$

In future work, we would like to add matching primitives to EVL in order to more easily decompose and process data. This is especially useful for detecting patterns, i.e., the presence of a common set of fields, in a sequences of events, which is a key notion in most CEP systems.

Chapter 6

Related Work

6.1 Type Systems for Record Calculi with Extensible Records

Following Ohori's approach in [46], we were able to give EVL a type system that supports the basic operations on records with a sound and complete type inference algorithm. Nevertheless, we found that adding more powerful operation on records such as field addition and removal could greatly improve the language's applicability in the context of CEP.

There are several alternative type systems that support polymorphic records with some form of record extensibility. The most common techniques are based on either subtyping [14, 37] or row variables [32, 49, 52, 54], but there are also others based on flags [48, 50], predicates [28, 33] and scope variables [42].

6.1.1 Subtyping

This is one of the most commonly used techniques used for building type systems for records [12, 13, 15, 47]. We can defined a subtyping relation, by specifying that a record type $l_1 : \tau_1, \ldots, l_n : \tau_n$ is a subtype of another record type $\{l'_1 : \tau'_1, \ldots, l'_m : \tau'_m\}$, denoted by $l_1 : \tau_1, \ldots, l_n : \tau_n \leq \{l'_1 : \tau'_1, \ldots, l'_m : \tau'_m\}$, if $\{l_1, \ldots, l_n\} \subseteq \{l'_1, \ldots, l'_m\}$. As an example, we have the following

$$\{one : Int, two : Int\} \leq \{one : Int, two : Int, three : Int\}.$$

The intuition here is that we should be able to use a record of type $\{one : Int, two : Int, three : Int\}$, in any context where a record of type $\{one : Int, two : Int\}$ is expected. In particular, the field selection operation on records $M.l$ can be treated as a function of the following type:

$$\forall\alpha.\forall\beta \leq \{l : \alpha\}.\beta \to \alpha.$$

One weakness of this approach is that information about the other fields of the record is lost, so it is harder to describe operations on records such as field addition and removal. This complicates compilation and will require some degree of dynamic typing during runtime to compensate for the loss of information.

6.1.2 Row Variables

Several approaches dealing with the extensibility of records are based on Wand's notion of a row variable [50]. Row variables are variables that range over sets of fields and allow for the incremental construction of records. As an example, a record of type

$$\{l : \tau \mid r\}$$

has all of the fields of a record of type r, together with a field with label l and of type τ. The field selection operation on records $M.l$ would have the type:

$$\forall\alpha\forall r.\{l : \alpha \mid r\} \to \alpha.$$

Wand did not discuss compilation, but his approach supports both polymorphism and extensibility. However, unlike our type system for the ML-style polymorphic record calculus with extensible records, operations are unchecked. As an example, adding a field with label l to a row may either add a completely new field, or simply replace an existing field labelled with l. As a result, some programs do not have principal types [53].

6.1.3 Flags

Flexible treatments for extensible records have been constructed around the concept of row variables [47, 48]. Rémy did this by developing a natural extension of ML. A key feature of his system is the use of flags to encode both positive and negative information on records, i.e., information about what fields must be present, and which must be absent. Row variables were used to deal with fields whose presence or absence is not significant in a particular situation. As an example, the field selection operator has type:

$$\forall\alpha.\forall r.\{l : pre(\alpha) \mid r\} \to \alpha,$$

where $pre(\alpha)$ is a flag indicating the presence of a field of type α, and r is a row variable representing the rest of the record. Unfortunately, despite its flexibility to define various powerful operations on records, this technique does not lead to a simple and efficient implementation. This is partially due to the fact that it still retains the ability to support some unchecked operations.

6.1.4 Predicates

Harper and Pierce [33, 34] have studied type systems for extensible records where negative and positive information on records is captured by predicates on types. In fact, using predicates they were able to develop a system with checked operations. As an example, if we write $r_1\#r_2$ for the assertion that record types r_1 and r_2 have disjoint sets of labels, the field selection operator has type:

$$\forall\alpha\forall r.(r\#\{l : \alpha\} \Rightarrow (r \parallel \{l : \alpha\}) \to \alpha,$$

where $r_1 \parallel r_2$ is the record type obtained by merging r_1 and r_2, and is only defined if $r_1 \# r_2$. Their work does not deal with type inference or compilation, and does not provide an operational interpretation of predicates.

6.1.5 Qualified Types

The use of predicates for the development of a type system for extensible records was one of the motivating examples in Jones' work on qualified types [38, 39]. In his work, a general framework for type inference and compilation was developed, including a type system for extensible records as a special case.

Building on this work is the approach by Gaster and Jones [28, 29]. Their approach combines both the notion of row variables and qualified types. It is perhaps the work that is more closely related to ours. Row variables are used to capture positive information, while predicates are used to capture negative information in order to avoid duplicating labels. As an example, the field operator has type:

$$(r \setminus l) \Rightarrow \{l : \alpha \mid r\} \to \alpha,$$

where the predicate $(r \setminus l)$ restricts instantiation of r to record types without an l-labelled field.

In our approach, kind restrictions capture both negative and positive information on records. Because of this, constraints on both field addition and removal can be treated in an uniform way.

6.1.6 Scoped Labels

Building on the work of Wand, Rémy, and Gaster and Jones [29, 50, 52], Leijen has developed a polymorphic type system for extensible records based on scoped labels [42]. His type system implements free extensions, i.e., the extension of a record with a field that is already present. As an example, consider the following record:

$$\{name = \mathit{"John"}\}$$

Because of free extension, the following extension of the previous record with the field $(name = \mathit{"Joe"})$ is also valid:

$$\{\{name = \mathit{"John"}\} \mid name = \mathit{"Joe"}\}.$$

Approaches using free extensions usually overwrite the previous field with the new field. By introducing a scoping mechanism for labels, Leijen is able to always keep previous fields, both in the value and in the type, while removing ambiguity and ensuring the safety of operations. In this system, the concepts of update and extension are separate operations work on the first matching label in a record. As an example, we should get $\mathit{"Joe"}$ from

$$\{\{name = \mathit{"John"}\} \mid name = \mathit{"Joe"}\}.name,$$

and *"John"* from

$$((\{\{name = \text{"}John\text{"}\} \mid name = \text{"}Joe\text{"}\}) \setminus\setminus name).name.$$

Because of this choice of semantics, it is always possible to unambiguously select a particular field from a record. But a new notion of equality between types where the record types are considered equal up to permutation of distinct labels is required in order to be able to compute more complicated field selections.

Free extensions are susceptible to programming errors where one unknowingly extends a record with a duplicate label. Leijen addresses this problem stating that the type system can always issue a warning if a record with a fixed type contains duplicate labels, but does not develop this idea. Our approach does not allow for duplicate labels and we implement a strict notion of extensibility using kind restrictions.

6.1.7 Compilation

Ohori's compilation method is based on the definition of an implementation calculus in which records are represented as vectors whose elements are accessed by direct indexing based on a total order $\ll$ on the set of labels (e.g., the lexicographical ordering on the string representation of labels) and of a compilation algorithm for translating terms from his original calculus to terms of that calculus. The set of terms of the implementation calculus includes index variables (ranged over by I) and natural numbers (ranged over by i). Given a total order $\ll$ on the set of labels, the index of a label l_i in a record type of the form $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is i, and a record term of the form $\{l_1 = M_1, \ldots, l_n = M_n\}$ and the previous type is a vector whose i^{th} element is the one assigned to the label l_i. To account for polymorphic operations, index types $idx(l, \tau)$ are used to type index values as follows. When τ is a record type, $idx(l, \tau)$ denotes the index of label l in τ and $\mid idx(l, \tau) \mid$ is the index value denoted by $idx(l, \tau)$. When τ is a type variable α, $idx(l, \alpha)$ denotes possible index values depending on the instantiations of α and $\mid idx(l, \alpha) \mid$ is undefined. The set of types of the implementation calculus extends the set of types of the original calculus with $idx(l, \tau_1) \Rightarrow \tau_2$, which denotes functions that take an index valued denoted by $idx(l, \tau_1)$ and return a value of type τ_2. The type system of the original calculus is also extended to include index assignments that map index variables I to index types of the form $idx(l, \tau)$ and allow for both typing and index judgements. Using these new constructions, Ohori's strategy for compiling polymorphic functions containing polymorphic record operations is to insert appropriate index abstractions as needed. As an example, the following polymorphic type

$$\forall \alpha_2 :: \{\{l_1 : Bool, l_2 : String\}\}.\forall \alpha_3 :: \{\{l_1 : l_2\}\}.\alpha_2 \to \alpha_3$$

is transformed to

$$\forall \alpha_2 :: \{\{l_1 : Bool, l_2 : String\}\}.\forall \alpha_3 :: \{\{l_1 : \alpha_2\}\}.idx(l_1, \alpha_2) \Rightarrow idx(l_2, \alpha_2) \Rightarrow idx(l_1, \alpha_3) \Rightarrow \alpha_2 \to \alpha_3$$

such that the order in which each $idx(l, \tau)$ appears for each τ follows the ordering implied by $\ll$.

To prove the correctness of this compilation method, Ohori shows that the compilation algorithm preserves the operational behaviour of the original polymorphic record calculus by proving that the compilation algorithm preserves types, and that the compilation calculus has the subject reduction property.

We believe that an efficient compilation algorithm can also be defined for the calculus developed in this work because variables still range over complete record types and the negative information added to kinds should not play any role in compilation. For these reasons, it should be possible to extend the compilation method in [46] to our approach.

6.2 Languages for Event Processing

When it comes to processing flows of information, there are two main models leading the research done in this area: the data stream processing model [6] and the complex event processing model [43]. The data stream processing model consists on producing new data streams by looking at streams of data coming from different sources, while the complex event processing model consists on looking at sequences of events happening in order to filter and combine them to produce new events.

In [20], several information processing systems were surveyed. This showed a gap between data processing languages and event detection languages, and the need to define a minimal set of language constructors to combine both features in the same language. We believe that `EVL` is a good candidate to explore the gap between these two models.

To deal with event classification in a uniform way, Alves et. al. [2] defined a general term-based language for events. In this language, events are represented as typed-terms built from a user-defined signature, i.e., a particular set of typed function symbols that are specific to the system modelled. With this approach it is possible to define general functions to implement event typing and to compute event intervals, without needing to know the exact type of events. A compound event [2] links a set of events that occur separately in the history, but should be identified as a single event occurrence. For simplicity, in [2] compound events were assumed to appear as a single event in history, leaving a more detailed and realistic treatment of compound events for future work. The notion of compound or composite event is also a key feature in CEP systems, which put great emphasis on the ability to detect complex patterns of incoming streams of events and establish sequencing and ordering relations. Types were used in [2] not only to ensure that terms representing events respect the type signature specific to the system under study, but also to formally define the notion of event instantiation, associating specific events to generic events through an implicit notion of subtyping, inspired by Ohori's system of polymorphic record types [46]. Because of the implicit subtyping rule for typing records, the system defined in [2] allowed for type-checking of event-specification, but not for dealing with most general types for event specifications. One of the main motivations behind the development of `EVL` was precisely to facilitate the specification and processing of (compound) events.

Following the complex event processing model, one of the key features is the ability to derive complex events (composite) from lower-level events and several special purpose Event Query Languages (EQLs) have been proposed for that [25]. One possible categorization of the multitude of EQLs consists in grouping together languages with a similar "style" or "flavor" together. As it turns out, in [25] it was found that most approaches for querying events fell into one of the following five categories:

1. Languages based on composition operators (sometimes also called composite event algebras or event pattern languages);
2. Data stream query languages (usually based on SQL);
3. Production rules;
4. Timed (finite) state machines;
5. Logic languages.

The first, the second and the fifth approaches were composed out of special purpose languages that were specifically designed for specifying event queries. The third and forth approaches are simply clever ways of using established technology to model event queries. That being said, it was also found in [25] that many industry products actually follow approaches where several languages of different flavors are supported or a single language combines aspects of several flavors. As an example, the TESLA language [20] supports content-based event filtering and is also able to establish temporal relations on events while providing a formal semantics based on temporal logic. The main advantage of logic languages is their strong formal foundation, an issue which is neglected by many languages of other styles [25]. In fact, the lack of a simple denotational semantics is a common criticism of CEP query languages [5, 27, 55], with several languages not guaranteeing important language features, such as orthogonality, as well as an overlapping of definitions that make reasoning about these languages much harder. Recently, a formal framework based on a Complex Event Logic was proposed [31], with the purpose of "giving a rigorous and efficient framework to CEP". The authors define well-formed and safe formulas as syntactic restrictions that characterize semantic properties, and argue that only well-formed formulas should be considered and that users should understand that all variables in a formula must be correctly defined. This notion of well-formed formulas and correctly defined variables is naturally guaranteed in a typed language like `EVL`. Therefore we believe that `EVL` can also be used to provide formal semantics to CEP systems.

Our notion of events follows the approach of the Event Calculus, where events are seen as action occurrences, or action happenings in a particular system and at a particular point in time. This notion was initially introduced in [41] then later axiomatised in [44], and has been further used in the context of dynamic access control systems [11] and in dynamic systems dealing with obligations [30]. As in the case of [9], our flexible representation of events is capable of encoding the representation of events in the Event Calculus. The higher-order capabilities of `EVL` allow

us to reason about events and their effects in a particular system by permitting us to define parameterised functions for dealing with the usual CEP techniques.

In the context of access control systems, the Obligation Specification Language defined in [35], presents a language for events to monitor and reason about data usage requirements. It also defines the *refinesEv* instance relation between events, which is based on a subset relation on labels, as is the case for the instance relation in [3]. The instance relation in [2] was defined by implicit subtyping on records but more generally using variable instantiation. In this work we further generalise the notion of instance relation and define it formally using kinded instantiation.

Chapter 7

Final Remarks

In [2], a general typed language to deal with the notion of event in the context of access control systems was defined. As a simplification, this event language did not deal with compound events and a more detailed and realistic treatment of this type of events was left for future work.

We have presented two typed languages for dealing with events and have shown how they can be used in the context of CEP and the specification of obligation policies. These languages are both based on Ohori's original ML-style polymorphic record calculus [46] mostly due to the fact that some aspects of the event language developed in [2] were inspired by that type system. Because of this, we were able to provide sound and complete type inference algorithms for both languages and a call-by-value operational semantics for `EVL` that ensures type-soundness.

In this work, we have also shown that it is possible to define a type system based on kinded quantification capable of capturing more powerful operations on records than the ones considered by Ohori, in order to maintain an efficient compilation method. Although we do not deal with compilation, we believe that an efficient compilation method can also be achieved for both our languages.

In future extensions of `EVL`, we would also like to explore the use of pattern matching. Pattern matching is a powerful mechanism for decomposing and processing data and its ability to detect patterns is a key notion in most CEP systems. For this reason, we believe that the addition of matching primitives to `EVL` would greatly improve its capabilities in the context of CEP.

We believe that most of the proofs presented in this work are easy to follow. With the possible exception of a few that are very technical. One such proof is that of Theorem 2.4.5. We find that some of the results presented in this work would greatly benefit from being validated using a proof assistant like Agda.

In the near future, we would like to continue to develop this work by defining an efficient compilation method for both our languages.

Appendix A

Appendix

A.1 Complete proof of Theorem 2.4.5

Proof. We first show that if the algorithm returns a kinded substitution, then it is a most general unifier of a given kinded set of equations.

Property 1 is composed out of the following sub-properties:

(1.1) K and $K \cup SK$ are well-formed kind assignments.

(1.2) E is well-formed under K.

(1.3) S is a well-formed substitution under K.

(1.4) $dom(K) \cap dom(SK) = \emptyset$.

(1.5) $dom(SK) = dom(S)$.

It is easily verified that each transformation rule preserves ***Property 1*** on 4-tuples:

- Rule *i)*:

  (1.1) K is well-formed and SK is well-formed.

  (1.2) Since $E \cup \{(\tau_1, \tau_2)\}$ is well-formed under K, then E is also well-formed under K.

  (1.3) S is well-formed under K.

  (1.4) $dom(K) \cap dom(SK) = \emptyset$.

  (1.5) $dom(SK) = dom(S)$.

- Rule *ii)*:

  (1.1) Since $K \cup \{(\alpha, \mathcal{U})\}$ is well-formed, we know that for all $\alpha' \in dom(K \cup \{(\alpha, \mathcal{U})\})$, $FTV(K \cup \{(\alpha, \mathcal{U})\}(\alpha')) \subseteq dom(K \cup \{(\alpha, \mathcal{U})\})$. But then $FTV(K(\alpha')) \cup FTV(\mathcal{U}) \subseteq$

$dom(K) \cup \{\alpha\} \iff FTV(K(\alpha')) \subseteq dom(K) \cup \{\alpha\}$, and $\alpha \notin FTV(\tau)$. Therefore, α will not appear in $[\tau/\alpha](K)$ and, for all $\alpha' \in dom([\tau/\alpha](K))$, $FTV([\tau/\alpha](K)(\alpha')) \subseteq dom(K) = dom([\tau/\alpha](K))$. Since $K \cup \{(\alpha,\mathcal{U})\} \cup SK$ is well-formed, we know that for all $\alpha' \in dom(K \cup \{(\alpha,\mathcal{U})\} \cup SK)$, $FTV(K \cup \{(\alpha,\mathcal{U})\} \cup SK(\alpha')) \subseteq dom(K \cup \{(\alpha,\mathcal{U})\} \cup SK)$. But then $FTV(K(\alpha')) \cup FTV(\mathcal{U}) \cup FTV(SK(\alpha')) \subseteq dom(K) \cup \{\alpha\} \cup dom(SK) \iff FTV(K(\alpha')) \cup FTV(SK(\alpha')) \subseteq dom(K) \cup \{\alpha\} \cup dom(SK)$, and $\alpha \notin FTV(\tau)$. Therefore α will not appear in $[\tau/\alpha](K)$ or $[\tau/\alpha](SK)$ and, for all $\alpha' \in dom([\tau/\alpha](K) \cup [\tau/\alpha](SK))$, $FTV([\tau/\alpha](K) \cup [\tau/\alpha](SK))) \subseteq dom([\tau/\alpha](K) \cup [\tau/\alpha](SK))$. But then, for all $\alpha' \in dom([\tau/\alpha](K) \cup [\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\})$, we have $FTV([\tau/\alpha](K) \cup [\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\}(\alpha')) = FTV([\tau/\alpha](K) \cup [\tau/\alpha](SK)(\alpha')) \subseteq dom(K) \cup dom(SK) = dom([\tau/\alpha](K) \cup [\tau/\alpha](SK)) \subseteq dom([\tau/\alpha](K) \cup [\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\})$.

(1.2) Since $E \cup \{(\alpha,\tau)\}$ is well-formed under $K \cup \{(\alpha,\mathcal{U})\}$, we know that E is well-formed under $K \cup \{(\alpha,\mathcal{U})\}$, that is, for all $(\tau_1,\tau_2) \in E$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K \cup \{(\alpha,\mathcal{U})\})$. But $\alpha \notin FTV(\tau)$, which means that α will not appear in $[\tau/\alpha](E)$ and for all $(\tau_1,\tau_2) \in [\tau/\alpha](E)$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) = dom([\tau/\alpha](K))$.

(1.3) Since S is well-formed under $K \cup \{(\alpha,\mathcal{U})\}$, we know that, for all $\alpha' \in dom(S)$, $FTV(S(\alpha')) \subseteq dom(K \cup \{(\alpha,\mathcal{U})\})$. But $FTV(S(\alpha)) \subseteq dom(K) \cup \{\alpha\}$ and $\alpha \notin FTV(\tau)$. Therefore α will not appear in $[\tau/\alpha](S)$ and, for all $\alpha' \in dom([\tau/\alpha](S))$, $FTV([\tau/\alpha](S)(\alpha')) \subseteq dom(K) = dom([\tau/\alpha](K))$. Also, we know that τ is well-formed under K, therefore $FTV(\tau) \subseteq dom(K) = dom([\tau/\alpha](K))$. Thus, for all $\alpha' \in dom([\tau/\alpha](S) \cup \{(\alpha,\tau)\})$, $FTV([\tau/\alpha](S) \cup \{(\alpha,\tau)\}(\alpha')) \subseteq dom([\tau/\alpha](K))$.

(1.4) $dom(K \cup \{(\alpha,\mathcal{U})\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha\}) \cap dom(SK) = \emptyset$. But $\alpha \in FTV(\tau)$, therefore α will not appear in $[\tau/\alpha](K)$ or $[\tau/\alpha](SK)$ and $(dom([\tau/\alpha](K)) \cup \{\alpha\}) \cap dom([\tau/\alpha](SK)) = \emptyset \iff dom([\tau/\alpha](K)) \cap (dom([\tau/\alpha](SK) \cup \{\alpha\})) = \emptyset \iff dom([\tau/\alpha](K)) \cap dom([\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\}) = \emptyset$.

(1.5) $dom(SK) = dom(S) \iff dom([\tau/\alpha](SK)) = dom([\tau/\alpha](S)) \iff dom([\tau/\alpha](SK)) \cup \{\alpha\} = dom([\tau/\alpha](S)) \cup \{\alpha\} \iff dom([\tau/\alpha](SK) \cup \{(\alpha,\mathcal{U})\}) = dom([\tau/\alpha](S) \cup \{(\alpha,\tau)\})$.

- Rule *iii)*:

(1.1) Since $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}$ is well-formed, we know that, for all $\alpha \in dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$, $FTV(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha')) \subseteq dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$ and, therefore, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \subseteq dom(K) \cup \{\alpha_1, \alpha_2\}$. But then α_1 will not appear in $[\alpha_2/\alpha_1](K)$ and, for all $\alpha \in dom([\alpha_2/\alpha_1](K))$, $FTV([\alpha_2/\alpha_1](K)(\alpha)) \subseteq dom(K) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K)) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}$. Also, α_1 will not appear in $[\alpha_2/\alpha_1]\{\{F_1^l \mid \mid F_1^r\}\}$ or $[\alpha_2/\alpha_1]\{\{F_2^l \parallel F_2^r\}\}$ and $FTV([\alpha_2/\alpha_1\{\{F_1^l \parallel F_1^r\}\}) \cup FTV([\alpha_2/\alpha_1]\{\{F_2^l \mid \mid F_2^r\}\}) \subseteq dom(K) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. But $FTV([\alpha_2/\alpha_1]\{\{F_1^l \parallel F_1^r\}\}) \cup FTV([\alpha_2/\alpha_1]\{\{F_2^l \parallel F_2^r\}\})$, therefore, for

all $\alpha \in dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$, $FTV([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}(\alpha)) \subseteq dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \mid \mid F_1^r + F_2^r\}\})\})$.

Since $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ is well-formed, we know that, for all $\alpha \in dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK)$, $FTV(K \cup \{(\alpha_1, \{\{F_1^l \mid \mid F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK(\alpha)) \subseteq dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \mid \mid F_2^r\}\})\} \cup SK)$ and, therefore, $FTV(K(\alpha)) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \cup FTV(SK(\alpha)) \subseteq dom(K) \cup \{\alpha_1, \alpha_2\} \cup dom(SK)$. But α_1 will not appear in $[\alpha_2/\alpha_1](K)$ or $[\alpha_2/\alpha_1](SK)$ and, for all $\alpha \in dom([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK))$, $FTV([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK)(\alpha)) \subseteq dom(K \cup SK) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK)) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. But then, for all $\alpha \in dom([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$, $FTV([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}(\alpha)) \subseteq dom([\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Also, then, for all $\alpha \in dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}\} \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}\})$, $FTV([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}\} \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\}(\alpha)) \subseteq dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}\} \cup [\alpha_2/\alpha_1](SK)) \cup \{\alpha_1\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\} \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}\})$.

(1.2) Since $E \cup \{(\alpha_1, \alpha_2)\}$ is well-formed under $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}$, we know that, for al $(\tau_1, \tau_2) \in E$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K \cup \{(\alpha_1, \{\{F_1^l \mid \mid F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}) = dom(K) \cup \{\alpha_1, \alpha_2\}$. But α_1 will not appear in $[\alpha_2/\alpha_1](E)$, therefore, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](E)$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K)) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Also, α_1 will not appear in any type in $[\alpha_2/\alpha_1](\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\})$ and, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\})$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K)) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Therefore, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\})$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1]\{\{F_1^l + F_2^l \mid \mid F_1^r + F_2^r\}\})\})$.

(1.3) Since S is well-formed under $K \cup \{\}$, we know that, for all $\alpha \in dom(S)$, $FTV(S(\alpha)) \subseteq dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}) = dom(K) \cup \{\alpha_1, \alpha_2\}$. But α_1 will not appear in $[\alpha_2/\alpha_1](S)$ and, for all $\alpha \in dom([\alpha_2/\alpha_1](S))$, $FTV(S(\alpha)) \subseteq dom(K) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K)) \cup \{\alpha_2\} = dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, \{\{F_1^l + F_2^l \mid \mid F_1^r + F_2^r\}\})\})$. Also, $FTV(\alpha_2) = \{\alpha_2\} \subseteq \{\alpha_2\} = dom(\{(\alpha_2, \{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}) \subseteq dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, \{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$, therefore, for all $\alpha \in dom([\alpha_2/\alpha_1](S)) \cup \{(\alpha_1, \alpha_2)\}$, $FTV([\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}(\alpha)) \subseteq dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, \{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$.

(1.4) $dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha_1,$

$\alpha_2\}) \cap dom([\alpha_2/\alpha_1](SK)) = \emptyset \iff (dom([\alpha_2/\alpha_1](K)) \cup \{\alpha_2\}) \cap (dom([\alpha_2/\alpha_1](SK)) \cup \{\alpha_1\}) = \emptyset \iff dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}) \cap dom([\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \alpha_2)\}) = \emptyset$.

(1.5) $dom(SK) = dom(S) \iff dom([\alpha_2/\alpha_1](SK)) = dom([\alpha_2/\alpha_1](S)) \iff dom([\alpha_2/\alpha_1](SK)) \cup \{\alpha_1\} = dom([\alpha_2/\alpha_1](S)) \cup \{\alpha_1\} \iff dom([\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\}) = dom([\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\})$.

- Rule *iv)*:

(1.1) Since $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$ is well-formed, we know that, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$, $FTV(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$ and, therefore, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \subseteq dom(K) \cup \{\alpha\}$. But, since $\alpha \notin FTV(\{F_2\})$, α will not appear in $[\{F_2\}/\alpha](K)$ and, for all $\alpha' \in dom([\{F_2\}/\alpha](K))$, $FTV([\{F_2\}/\alpha](K)(\alpha')) \subseteq dom(K) = dom([\{F_2\}/\alpha](K))$.
Since $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup SK$ is well-formed, we know that, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup SK)$, $FTV(K \cup \{(\alpha, \{\{F_1^l \parallel F_2^r\}\})\} \cup SK(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}\} \cup SK)$ and, therefore, $FTV(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup SK(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup SK)$. Also, $dom(K(\alpha')) \cup dom(\{\{F_1^l \parallel F_1^r\}\}) \cup dom(\alpha') \subseteq dom(K) \cup \{\alpha\} \cup dom(SK)$. But, since $\alpha \notin FTV(\{F_2\})$, α will not appear in $[\{F_2\}/\alpha](K)$ or $[\{F_2\}/\alpha](SK)$ and, for all $\alpha' \in dom([\{F_2\}/\alpha] \cup [\{F_2\}/\alpha](SK))$, $FTV([\{F_2\}/\alpha](K) \cup [\{F_2\}/\alpha](SK)(\alpha')) \subseteq dom([\{F_2\}/\alpha](K) \cup [\{F_2/\alpha](SK))$. Also, $FTV(\{\{F_1^l \parallel F_1^r\}\}) \subseteq dom(K) \cup \{\alpha\} \cup dom(SK) = dom([\{F_2\}/\alpha](K)) \cup \{\alpha\} \cup dom([\{F_2/\alpha\}/\alpha](SK))$. Therefore, for all $\alpha' \in dom([\{F_2\}/\alpha](K) \cup [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$, $FTV([\{F_2/\alpha\}](K) \cup [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}(\alpha')) \subseteq dom([\{F_2\}/\alpha](K) \cup [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}\})$.

(1.2) Since $E \cup \{(\alpha, \{F_2\}\}$ is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}\}$, we know that, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha, \{F_2\})\}$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}) = dom(K) \cup \{\alpha\}$. But, since $\alpha \notin FTV(\{F_2\})$, α will not appear in $[\{F_2\}/\alpha](E)$ and, for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](E)$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) = dom([\{F_2\}/\alpha](K))$. Also, since $\alpha \notin FTV(\{F_2\})$, α will not appear in any type in $[\{F_2\}/\alpha](\{(F_1^l(l), F_2^r(l)) \mid l \in dom(F_1^l)\})$ and, for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](\{(F_1^l(l), F_1^r(l)) \mid l \in dom(F_1^l)\})$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) = dom([\{F_2\}/\alpha](K))$.

(1.3) Since S is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$, we know that, for all $\alpha' \in dom(S)$, $FTV(S(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}) \subseteq dom(K) \cup \{\alpha\}$. Since $\alpha \notin FTV(\{F_2\})$, α will not appear in $[\{F_2\}/\alpha](S)$ and, for all $\alpha' \in dom([\{F_2\}/\alpha](S))$, $FTV([\{F_2\}/\alpha](S)(\alpha')) \subseteq dom(K) = dom([\{F_2\}/\alpha](K))$. Also, we know that $\{F_2\}$ is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$, therefore $FTV(\{F_2\}) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}\})\}) = dom(K) \cup \{\alpha\}$. But, $\alpha \notin dom([\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\})$, $FTV([\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}(\alpha')) \subseteq dom([\{F_2\}/\alpha](K))$.

(1.4) $dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha\}) \cap dom(SK) = \emptyset \iff (dom([\{F_2\}/\alpha](K)) \cup \{\alpha\}) \cap dom([\{F_2\}/\alpha](SK)) = \emptyset \iff dom([\{F_2\}/\alpha](K))$

$\cap\,(dom([\{F_2\}/\alpha](SK) \cup \{\alpha\})) = \emptyset \iff dom([\{F_2\}/\alpha](K)) \cap (dom([\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})) = \emptyset$.

(1.5) $dom(SK) = dom(S) \iff dom([\{F_2\}/\alpha](SK)) = dom([\{F_2\}/\alpha](S)) \iff dom([\{F_2\}/\alpha](SK)) \cup \{\alpha\} = dom([\{F_2\}/\alpha](S)) \cup \{\alpha\} \iff dom([\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}) = dom([\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\})$.

- Rule *v)*:

  (1.1) K is well-formed and $K \cup SK$ is well-formed.

  (1.2) Since $E \cup \{(\{F_1\}, \{F_2\})\}$ is well-formed under K, we know that E is well-formed under K. Since both $\{F_1\}$ and $\{F_1\}$ are well-formed under K, we know that, for all $(\tau_1, \tau_2) \in \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K)$. But then, $E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}$ is well-formed under K.

  (1.3) S is well-formed under K.

  (1.4) $dom(K) \cap dom(SK) = \emptyset$.

  (1.5) $dom(SK) = dom(S)$.

- Rule *vi)*:

  (1.1) K is well-formed and $K \cup SK$ is well-formed.

  (1.2) Since $E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}$ is well-formed under K, we know that E is well-formed under K. Also, $FTV(\tau_1^1 \to \tau_1^2) \cup FTV(\tau_2^1 \to \tau_2^2) \subseteq dom(K)$ and, therefore, $FTV(\tau_1^1) \cup FTV(\tau_1^2) \cup FTV(\tau_2^1) \cup FTV(\tau_2^2) \subseteq dom(K)$. But, then $FTV(\tau_1^1) \cup FTV(\tau_2^1) \subseteq dom(K)$ and $FTV(\tau_1^2) \cup FTV(\tau_2^2) \subseteq dom(K)$. Therefore $E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}$ is well-formed under K.

  (1.3) S is well-formed under K.

  (1.4) $dom(K) \cap dom(SK) = \emptyset$.

  (1.5) $dom(SK) = dom(S)$.

- Rule *vii)*:

  (1.1) Since $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}$ is well-formed, we know that, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\})$, $FTV(K \cup \{(\alpha, \{\{F_1^l \mid \mid F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\})$. Also, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \subseteq dom(K) \cup \{\alpha, root(\chi)\}$. But, since $\alpha \notin FTV(\chi)$, α will not appear in $[\chi/\alpha](K)$ and, for all $\alpha' \in dom([\chi/\alpha](K))$, $FTV([\chi/\alpha](K)(\alpha')) \subseteq dom(K) \cup \{root(\chi)\} = dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$. Also, $FTV(\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}) \subseteq dom(K) \cup \{\alpha, root(\chi)\}$. Since α will not appear in $[\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\})$, we have that $FTV([\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\})) \subseteq dom(K) \cup \{root(\chi)\} = dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$. Thus, for all $\alpha' \in dom([\chi/\alpha](K) \cup$

$\{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$, $FTV([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\}(\alpha')) \subseteq dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \mid \mid F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$.

Since $K \cup \{(\alpha, \{\{F_1 \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ is well-formed, we know that, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\} \cup SK)$, $FTV(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \mid \mid F_2^r\}\})\} \cup SK)$. Also, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \cup FTV(SK(\alpha')) \subseteq dom(K) \cup \{\alpha, root(\chi)\} \cup dom(SK)$. But, since $\alpha \notin FTV(\chi)$, α will not appear in $[\chi/\alpha](K)$ or $[\chi/\alpha](SK)$ and, for all $\alpha' \in dom([\chi/\alpha](K) \cup [\chi/\alpha](SK))$, $FTV([\chi/\alpha](K) \cup [\chi/\alpha](SK)(\alpha')) \subseteq dom(K) \cup \{root(\chi)\} \cup dom(SK) = dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\} \cup [\chi/\alpha](SK))$. Also, $FTV(\{\{F_1^l \parallel F_1^r\}\}) \subseteq dom(K) \cup \{\alpha, root(\chi)\} \cup dom(SK) = dom([\chi/\alpha](K)) \cup \{\alpha, root(\chi)\} \cup dom([\chi/\alpha](SK))$ and $dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\} \cup [\chi/\alpha](SK)) \subseteq dom([\chi/\alpha](SK)) \cup \{\alpha, root(\chi)\} \cup [\chi/\alpha](SK)$. Therefore, for all $\alpha' \in dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\} \cup [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \mid \mid F_1^r\}\}), FTV([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\} \cup [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})(\alpha')) \subseteq dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\} \cup [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\})$.

(1.2) Since $E \cup \{(\alpha, \chi)\}$ is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$, we know that E is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$. But $\alpha \notin FTV(\chi)$, therefore α will not appear in $[\chi/\alpha](E)$ and, for all $(\tau_1, \tau_2) \in [\chi/\alpha](E)$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{root(\chi)\} = dom([\chi/\alpha](K)) \cup \{root(\chi)\}$. Also, since $\alpha \notin FTV(\chi)$, α will not appear in any of the types in $[\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)})(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)}\}))$, therefore, for all $(\tau_1, \tau_2) \in [\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)})(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)}\}))$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{root(\chi)\} = dom([\chi/\alpha](K)) \cup \{root(\chi)\}$.

(1.3) Since S is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$, we know that, for all $\alpha' \in dom(S)$, $FTV(S(\alpha')) \subseteq dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup \{(root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}) = dom(K) \cup \{\alpha, root(\chi)\}$. But, since $\alpha \notin FTV(\chi)$, then α will not appear in $[\chi/\alpha](S)$ and, for all $\alpha' \in dom([\chi/\alpha](S))$, $FTV([\chi/\alpha](S)(\alpha')) \subseteq dom(K) \cup \{root(\chi)\} = dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$. Also, χ is well-formed under $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}$, which means that $FTV(\chi) \subseteq dom((K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\})$. But, $\alpha \notin FTV(\chi)$, therefore $FTV(\chi) \subseteq dom(K \cup \{(root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}) = dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$. Thus, for all $\alpha' \in dom([\chi/\alpha](S) \cup \{(\alpha, \chi)\})$, $FTV([\chi/\alpha](S) \cup \{(\alpha, \chi)\}(\alpha')) \subseteq dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\})$.

(1.4) $dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha, root(\chi)\}) \cap dom(SK) = \emptyset \iff (dom([\chi/\alpha](K)) \cup \{\alpha, root(\chi)\}) \cap dom([\chi/\alpha](SK)) = \emptyset \iff (dom([\chi/\alpha](K)) \cup \{root(\chi)\}) \cap (dom([\chi/\alpha](SK) \cap \{\alpha\})) = \emptyset \iff dom([\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\not\downarrow)}))) \parallel F_2^r + (F_1^r - F_{c(\not\downarrow)})\}\}))\}) \cap dom([\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}) = \emptyset$.

(1.5) $dom(SK) = dom(S) \iff dom([\chi/\alpha](SK)) = dom([\chi/\alpha](S)) \iff dom([\chi/\alpha](SK)) \cup \{\alpha\} = dom([\chi/\alpha](S)) \cup \{\alpha\} \iff dom([\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} = dom([\chi/\alpha](S) \cup \{(\alpha, \chi)\})$.

- Rule *viii)*:

(1.1) K is well-formed and $K \cup SK$ is well-formed.

(1.2) Since $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}$ is well-formed under K, we know that E is well-formed under K. Also, $FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \subseteq dom(K)$ and $FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \subseteq dom(K)$. But, then $FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \{l_{i+1}^1 : \tau_{i+1}^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \subseteq FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \subseteq dom(K)$ and $FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \{l_{j+1}^2 : \tau_{j+1}^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \subseteq FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \subseteq dom(K)$.

(1.3) S is well-formed under K.

(1.4) $dom(K) \cap dom(SK) = \emptyset$.

(1.5) $dom(SK) = dom(S)$.

- Rule *ix)*:

(1.1) Since $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}$ is well-formed, we know that $\mathcal{I}(K) \cup \{(\alpha, \{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}$ is well-formed and that, for all $\alpha' \in dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\})$, $FTV(K \cup \{K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha')) \subseteq dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\})$ and, therefore, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \subseteq dom(K) \cup \{\alpha^1, \alpha^2\}$. But, since α is fresh, $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \cup FTV(\tau_1^1) \cup \cdots \cup FTV(\tau_n^1)$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \cup \cdots \cup FTV(\tau_1^2) \cdots FTV(\tau_m^2)$, both α^1 and α^2 will not appear in $\mathcal{I}(K)$ and, for all $\alpha' \in dom(\mathcal{I}(K))$, $FTV(\mathcal{I}(K)(\alpha')) \subseteq dom(K) = dom(\mathcal{I}(K))$. Also, $FTV(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}) \subseteq FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \subseteq dom(K) \cup \{\alpha_1, \alpha_2\}$. But, again, α^1 and α^2 will not appear in $\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})$ and $FTV(\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K)) \cup \{\alpha\}$. Since $dom(\mathcal{I}(K)) \subseteq dom(\mathcal{I}(K)) \cup \{\alpha\}$, we have that, for all $\alpha' \in dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\})$, $FTV(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}(\alpha')) \subseteq dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\})$.
Since $K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ is well-formed, we know that, for all $\alpha' \in dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK)$, $FTV(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK(\alpha')) \subseteq dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel$

$| F_2^r\}\})\} \cup SK)$ and, therefore, $FTV(K(\alpha')) \cup FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \mid | F_2^r\}\}) \cup FTV(SK(\alpha')) \subseteq dom(K) \cup \{\alpha^1, \alpha^2\} \cup dom(SK)$. But, since α is fresh, $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \cup FTV(\tau_1^1) \cup \cdots \cup FTV(\tau_n^1)$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \cup \cdots \cup FTV(\tau_1^2) \cdots FTV(\tau_m^2)$ and α^2 will not appear in $\mathcal{I}(K)$ or $\mathcal{I}(SK)$ and, for all $\alpha' \in dom(\mathcal{I}(K) \cup \mathcal{I}(SK))$, $FTV(\mathcal{I}(K) \cup \mathcal{I}(SK)(\alpha')) \subseteq dom(K) \cup dom(SK) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, S(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Also, $FTV(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}) \subseteq FTV(\{\{F_1^l \parallel F_1^r\}\}) \cup FTV(\{\{F_2^l \parallel F_2^r\}\}) \subseteq dom(K) \cup dom(SK) \cup \{\alpha^1, \alpha^2\}$ and α^1 and α^2 will not appear in $\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})$, therefore $FTV(\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})) \subseteq dom(K) \cup dom(SK) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Finally, since $dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \mid | F_1^r + F_2^r\}\})\}) \subseteq dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\}) \cup \{\alpha^1, \alpha^2\}$, we have that, for all $\alpha' \in dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\} \cup \{(\alpha^1, \{\{F_1^l \mid | F_1^r\}\}\} \cup \{(\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\})$, $FTV(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\} \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}\} \cup \{(\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha')) \subseteq dom(\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \mid | F_1^r + F_2^r\}\})\} \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$.

(1.2) We know that $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}$ is well-formed under $K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F^l \parallel F^r\}\})\}$. But then, since α is fresh, $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \cup FTV(\tau_1^1) \cup \cdots \cup FTV(\tau_n^1)$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \cup \cdots \cup FTV(\tau_1^2) \cdots FTV(\tau_m^2)$ and α^2 will not appear in any type in $\mathcal{I}(E)$, therefore, for all $(\tau_1, \tau_2) \in \mathcal{I}(E)$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})\})$. Also, for all $(\tau_1, \tau_2) \in \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\}$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{\alpha_1, \alpha_2\}$, but α_1 and α_2 will not appear in $\mathcal{I}(\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\})$, therefore, for all $(\tau_1, \tau_2) \in \mathcal{I}(\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\})$, $FTV(\tau_1) \cup FTV(\tau_2) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\})$.

(1.3) Since S is well-formed under $K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}$, we know that, for all $\alpha' \in dom(S)$, $FTV(S(\alpha')) \subseteq dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}) = dom(K) \cup \{\alpha^1, \alpha^2\}$. But, since α is fresh, $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \cup FTV(\tau_1^1) \cup \cdots \cup FTV(\tau_n^1)$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \cup \cdots \cup FTV(\tau_1^2) \cdots FTV(\tau_m^2)$ and α^2 will not appear in $\mathcal{I}(S)$, therefore, for all $\alpha' \in dom(\mathcal{I}(S))$, $FTV(\mathcal{I}(S)(\alpha')) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\})$. Also, since $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \cup FTV(\tau_1^1) \cup \cdots \cup FTV(\tau_n^1)$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \cup \cdots \cup FTV(\tau_1^2) \cdots FTV(\tau_m^2)$, we have that $FTV(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \mid | F_1^r + F_2^r\}\}))\})$ and $FTV(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \pm_m^2 \{l_m^2 : \tau_m^2\}) \subseteq dom(K) \cup \{\alpha\} = dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_1^l \parallel F_1^r + F_2^r\}\}))\})$. Therefore, for all $\alpha' \in dom(\mathcal{I}(S) \cup \{(\alpha^1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha^2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\})$.

(1.4) $dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha^1, \alpha^2\}) \cap dom(SK) = \emptyset \overset{\alpha \text{ fresh}}{\iff} (dom(K) \cup \{\alpha^1, \alpha^2, \alpha\}) \cap dom(SK) = \emptyset \iff (dom(K) \cup \{\alpha\}) \cap (dom(SK) \cup \{\alpha^1, \alpha^2\}) = \emptyset \iff dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}) \cap dom(\mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}) = \emptyset$.

(1.5) $dom(SK) = dom(S) \iff dom(SK) \cup \{\alpha^1, \alpha^2\} = dom(S) \cup \{\alpha^1, \alpha^2\} \iff dom(SK \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}) = dom(S \cup \{(\alpha^1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha^2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\})$.

We now state two more properties that are also preserved by each transformation rule:

- ***Property 2***: For any kinded substitution (K_0, S_0), if (K_0, S_0) respects K and S_0 satisfies $E \cup S$ then (K_0, S_0) respect SK.
- ***Property 3***: The set of unifiers of $(K \cup SK, E \cup S)$.

We can verify that these two properties are preserved by each transformation rule knowing that ***Property 1*** holds for the 4-tuple.

Preservation of ***Property 2***:

- Rule *i)*: Assume that (K_0, S_0) respects K. Assume that $\tau_1 \equiv_\chi \tau_2$ and S_0 satisfies $E \cup \{(\tau_1, \tau_2)\} \cup S$. Then, (K_0, S_0) respects SK.
- Rule *ii)*: If we assume that (K_0, S_0) respects $[\tau/\alpha](K)$, then we know that, for all $\alpha' \in dom([\tau/\alpha](K))$, $K_0 \vdash_\chi S_0([\tau/\alpha](K)(\alpha'))$. If we assume that S_0 satisfies $[\tau/\alpha](E) \cup [\tau/\alpha](S) \cup \{(\alpha, \tau)\}$, then we know that, for all $(\tau_1, \tau_2) \in [\tau/\alpha](E) \cup [\tau/\alpha](S) \cup \{(\alpha, \tau)\}$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:
  - for all $(\tau_1, \tau_2) \in [\tau/\alpha](E)$, $S_0(\tau_1) = S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in [\tau/\alpha](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - $S_0(\alpha) \equiv_\chi S_0(\tau)$.

  But then, since $S_0(\alpha) \equiv_\chi S_0(\tau)$, we have that $S_0(\alpha) \equiv_\chi S_0 \circ [\tau/\alpha](\alpha) \equiv_\chi S_0(\tau)$. Also, since $\alpha \notin FTV(\tau)$, α will not appear in $[\alpha/\alpha](K)$ and $dom(K) = dom([\tau/\alpha](K))$, which means that, for all $\alpha' \in dom(K \cup \{(\alpha, \mathcal{U})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(K(\alpha'))$ and, therefore, $K_0 \vdash_\chi S_0 \circ [\tau/\alpha](\alpha') :: S_0 \circ [\tau/\alpha](K(\alpha'))$, that is:
  - $K_0 \vdash_\chi S_0([\tau/\alpha](\alpha')) :: S_0([\tau/\alpha](K)(\alpha'))$, if $\alpha' \neq \alpha$;
  - $K_0 \vdash_\chi S_0([\tau/\alpha](\alpha)) :: S_0([\tau/\alpha](K)(\alpha)) \iff K_0 \vdash_\chi S_0(\alpha) :: \mathcal{U}$, if $\alpha' = \alpha$ - which is valid since τ is well-formed under K, $S_0(\alpha) \equiv_\chi S_0(\tau)$, and $S_0(\tau)$ is well-formed under K_0.

  Also, since $S_0 = S_0 \circ [\tau/\alpha]$, then, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha, \tau)\} \cup S$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, since:
  - if, for all $(\tau_1, \tau_2) \in [\tau/\alpha](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ [\tau/\alpha](\tau_1) \equiv_\chi S_0 \circ [\tau/\alpha](\tau_2)$;
  - if, for all $(\tau_1, \tau_2) \in [\tau/\alpha](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ [\tau/\alpha](\tau_1) \equiv_\chi S_0 \circ [\tau/\alpha](\tau_2)$;

– $S_0(\alpha) \equiv_\chi S_0 \circ [\tau/\alpha](\alpha) \equiv_\chi S_0 \circ [\tau/\alpha](\tau) \equiv_\chi S_0(\tau)$.

That is, (K_0, S_0) respects $K \cup \{(\alpha, \mathcal{U})\}$ and S_0 satisfies $E \cup \{(\alpha, \mathcal{U})\} \cup S$. But then, (K_0, S_0) respects SK, that is, for all $\alpha' \in dom(SK)$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(SK(\alpha'))$. But, since $\alpha \notin FTV(\tau)$, α will not appear in $[\tau/\alpha](SK)$, therefore, for all $\alpha' \in dom([\tau/\alpha](SK))$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(SK(\alpha'))$. Also, $K_0 \vdash_\chi S_0(\alpha) :: \mathcal{U}$, therefore, for all $\alpha' \in dom([\tau/\alpha](SK) \cup \{(\alpha, \mathcal{U})\})$, $K \vdash_\chi S_0(\alpha') :: S_0(SK(\alpha'))$, which means that (K_0, S_0) respects $[\tau/\alpha](SK) \cup \{(\alpha, \tau)\}$.

- Rule *iii)*: If we assume that (K_0, S_0) respects $[\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}$, then we know that, for all $\alpha' \in dom([\alpha_2/\alpha_1](K) \cup \{(\alpha_2, [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}(\alpha'))$, that is:

  – $K_0 \vdash_\chi S_0(\alpha') :: S_0([\alpha_2/\alpha_1](K)(\alpha'))$;

  – $K_0 \vdash_\chi S_0(\alpha_2) :: S_0([\alpha_2/\alpha_1](\{\{F_1^l + F_1^l \parallel F_1^r + F_2^r\}\}))$.

  If we assume that S_0 satisfies $[\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l) \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}) \cup [\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}$, then we know that, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l) \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}) \cup [\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:

  – for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;

  – for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;

  – for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](\{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;

  – for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;

  – $S_0(\alpha_1) \equiv_\chi S_0(\alpha_2)$.

  But then, since $S_0(\alpha_1) \equiv_\chi S_0(\alpha_2)$, we have that $S_0 \circ [\alpha_2/\alpha_1](\alpha_1) \equiv_\chi S_0(\alpha_2)$, therefore, $S_0 = S_0 \circ [\alpha_2/\alpha_1]$. Also, α_1 will not appear in $[\alpha_2/\alpha_1](K)$ and $dom(K) = dom([\alpha_2/\alpha_1](K))$, which means that, for all $\alpha \in dom(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\})$, $K_0 \vdash_\chi S_0(K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha))$, since:

  – $K_0 \vdash_\chi S_0 \circ [\alpha_2/\alpha_1](\alpha) :: S_0 \circ [\alpha_2/\alpha_1](K(\alpha))$, if $\alpha \notin \{\alpha_1, \alpha_2\}$;

  – $KS_2 \circ [\alpha_2/\alpha_1](\alpha_1) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_1^l \parallel F_1^r\}\})$, if $\alpha = \alpha_1$;

  – $KS_2 \circ [\alpha_2/\alpha_1](\alpha_2) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_2^l \parallel F_2^r\}\})$, if $\alpha = \alpha_2$.

  The two last points are valid since $dom(F_1^l) \cap dom(F^r) = \emptyset = dom(F_1^r) \cap dom(F_2^l)$ and $K_0 \vdash_\chi S_0 \circ [\alpha_2/\alpha_1](\alpha_2) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})$. Also, since $S_0 = S_0 \circ [\alpha_2/\alpha_1]$, then, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha_1, \alpha_2)\} \cup S$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, since:

  – if, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ [\alpha_2/\alpha_1](\tau_1) \equiv_\chi S_0 \circ [\alpha_2/\alpha_1](\tau_2)$;

  – if, for all $(\tau_1, \tau_2) \in [\alpha_2/\alpha_1](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in S$, $S_0 \circ [\alpha_2/\alpha_1](\tau_1) \equiv_\chi S_0 \circ [\alpha_2/\alpha_1](\tau_2)$;

– $S_0(\alpha_1) \equiv_\chi S_0 \circ [\alpha_2/\alpha_1](\alpha_1) \equiv_\chi S_0(\alpha_2)$.

That is, (K_0, S_0) respects $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\}$ and S_0 satisfies $E \cup \{(\alpha_1, \alpha_2)\} \cup S$. But, then (K_0, S_0) respects SK, that is, for all $\alpha \in dom(SK)$, $K \vdash_\chi S_0(\alpha) :: S_0(SK(\alpha))$. But α_1 will not appear in $[\alpha_2/\alpha_1](SK)$ and $dom(SK) = dom([\alpha_2/\alpha_1](SK))$, therefore, for all $\alpha \in dom([\alpha_2/\alpha_1](SK))$, $K_0 \vdash_\chi S_0(\alpha) :: S_0([\alpha_2/\alpha_1](SK))$. Also, we know that $K \vdash_\chi S_0(\alpha_1) :: S_0([\alpha_2/\alpha_1](\{\{F_1^l \parallel F_2^r\}\})) \iff K \vdash_\chi S_0 \circ [\alpha_2/\alpha_1](\alpha_1) :: S_0([\alpha_2/\alpha_1](\{\{F_1^l \parallel F_2^r\}\})) \iff K \vdash_\chi S_0(\alpha_2) :: S_0([\alpha_2/\alpha_1](\{\{F_1^l \parallel F_2^r\}\}))$, which means that (K_0, S_0) respects $[\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\}$.

- Rule *iv)*: If we assume that (K_0, S_0) respects $[\{F_2\}/\alpha](K)$, then we know that, for all $\alpha' \in dom([\{F_2\}/\alpha](K))$, $K_0 \vdash_\chi S_0(\alpha') :: S_0([\{F_2\}/\alpha](K)(\alpha'))$. If we assume that S_0 satisfies $[\{F_2\}/\alpha](E \cup \{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)\}) \cup [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}$, then we know that, for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](E \cup \{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)) \cup [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:

  – for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  – for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](\{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  – $S_0(\alpha) \equiv_\chi S_0(\{F_2\})$.

  But then, since $S_0(\alpha) \equiv_\chi S_0(\{F_2\})$, and $\alpha \notin FTV(\{F_2\})$, we have that $S_0 \circ [\{F_2\}/\alpha](\alpha) \equiv_\chi S_0(\{F_2\}) \equiv_\chi S_0 \circ [\{F_2\}/\alpha](\{F_2\})$, that is, that $S_0 = S_0 \circ [\{F_2\}/\alpha]$. Also, α will not appear in $[\{F_2\}/\alpha](K)$ and $dom([\{F_2\}/\alpha](K)) = dom(K)$, therefore, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \mid \mid F_1^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(K(\alpha'))$, since:

  – $K \vdash_\chi S_0 \circ [\{F_2\}/\alpha](\alpha') :: S_0 \circ [\{F_2\}/\alpha](K(\alpha')) \iff K_0 \vdash_\chi S_0(\alpha') :: S_0([\{F_2\}/\alpha](K)(\alpha'))$, if $\alpha' \neq \alpha$;
  – $K_0 \vdash_\chi S_0 \circ [\{F_2\}/\alpha](\alpha) :: S_0 \circ [\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(\{F_2\}) :: S_0 \circ [\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\})$, if $\alpha' = \alpha$.

  And the last point is valid since S_0 satisfies $[\{F_2/\alpha](\{(F_1^l(l), F_2(l)) \mid l \in dom(F_1^l)\})$, $dom(F_1^l) \subseteq dom(F_2)$, and $dom(F_1^r) \cap dom(F_2) = \emptyset$.

  Also, since $S_0 = S_0 \circ [\{F_2\}/\alpha]$, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha, \{F_2\})\} \cup S$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, since:

  – if, for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ [\{F_2\}/\alpha](\tau_1) \equiv_\chi S_0 \circ [\{F_2\}/\alpha](\tau_2)$;
  – if, for all $(\tau_1, \tau_2) \in [\{F_2\}/\alpha](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in S$, $S_0 \circ [\{F_2\}/\alpha](\tau_1) \equiv_\chi S_0 \circ [\{F_2\}/\alpha](\tau_2)$;
  – $S_0(\alpha) \equiv_\chi S \circ [\{F_2\}/\alpha](\alpha) \equiv_\chi S_0(\{F_2\})$.

  That is, (K_0, S_0) respects $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$ and S_0 satisfies $E \cup \{(\alpha, \{F_2\})\} \cup S$. But, then (K_0, S_0) respects SK, that is, for all $\alpha' \in dom(SK)$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(SK(\alpha'))$. But, α will not appear in $[\{F_2\}/\alpha](SK)$ and $dom(SK) = dom([\{F_2\}/\alpha](SK))$, therefore, for all

$\alpha' \in dom([\{F_2\}/\alpha](SK))$, $K_0 \vdash_\chi S_0(\alpha') :: S_0([\{F_2\}/\alpha](SK)(\alpha'))$. Also, we know that $K_0 \vdash_\chi S_0(\alpha) :: S_0([\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\})) \iff K_0 \vdash_\chi S_0 \circ [\{F_2\}/\alpha](\alpha) :: S_0([\{F_2\}/\alpha](\{\{F_1^l \mid \mid F_1^r\}\})) \iff K_0 \vdash_\chi S_0(\{F_2\}) :: S_0([\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\}))$, therefore, (K_0, S_0) respects $[\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$.

- Rule *v)*: Assume that (K_0, S_0) respects K. If we assume that S_0 satisfies $E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\} \cup S$, then we know that, for all $(\tau_1, \tau_2) \in E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\} \cup S$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:

  - for all $(\tau_1, \tau_2) \in E$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}$.

  But, then S_0 satisfies E. Also, since S_0 satisfies $\{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}$ and $dom(F_1) = dom(F_2)$, we have that S_0 satisfies $\{(\{F_1\}, \{F_2\})\}$. This means that S_0 satisfies $E \cup \{(\{F_1\}, \{F_2\})\}$. Therefore, (K_0, S_0) respects SK.

- Rule *vi)*: Assume that (K_0, S_0) respects K. If we assume that S_0 satisfies $E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\} \cup S$, then we know that:

  - S_0 satisfies E;
  - S_0 satisfies S;
  - S_0 satisfies $\{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}$.

  Therefore, (K_0, S_0) satisfies SK.

- Rule *vii)*: If we assume that (K_0, S_0) respects $[\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l \parallel F_2^r\}\})\}$, then we know that, for all $\alpha' \in dom([\chi/\alpha](K) \cup \{(root(\chi), \{\{F_2^l \parallel F_2^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0([\chi/\alpha](K) \cup \{(root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}$, that is:

  - $K_0 \vdash_\chi S_0(\alpha') :: S_0([\chi/\alpha](K)(\alpha'))$, if $\alpha' = root(\chi)$;
  - $K_0 \vdash_\chi S_0(root(\chi)) :: S_0([\chi/\alpha](\{\{F_2^l \parallel F_2^r\}\}))$, if $\alpha' = root(\chi)$.

  If we assume that S_0 satisfies $[\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)})\}) \cup [\chi/\alpha](S) \cup \{(\alpha, \chi)\}$, then we know that, for all $(\tau_1, \tau_2) \in [\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)})\}) \cup [\chi/\alpha](S) \cup \{(\alpha, \chi)\}$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:

  - for all $(\tau_1, \tau_2) \in [\chi/\alpha](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in [\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in [\chi/\alpha](\{(F_1^r(l), F_{c(\chi)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)})\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in [\chi/\alpha](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - $S_0(\alpha) \equiv_\chi S_0(\chi)$.

But, then, since $S_0(\alpha) \equiv_\chi S_0(\chi)$ and $\alpha \notin FTV(\chi)$, we have that $S_0 \circ [\chi/\alpha](\alpha) \equiv_\chi S_0(\chi) \equiv_\chi S_0 \circ [\chi/\alpha](\chi)$. Also, α will not appear in $[\chi/\alpha](K)$ and $dom([\chi/\alpha](K)) = dom(K)$, therefore, for all $\alpha' \in dom(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup \{(root(\chi), \{\{F_2^l \parallel F_2^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}(\alpha'))$, that is:

- $K_0 \vdash_\chi S_0(\alpha') :: S_0(K(\alpha')) \iff K_0 \vdash_\chi S_0 \circ [\chi/\alpha](\alpha') :: S_0 \circ [\chi/\alpha](K(\alpha'))$, if $\alpha' \in \{\alpha, root(\chi)\}$;
- $K_0 \vdash_\chi S_0 \circ [\chi/\alpha](\alpha) :: S_0 \circ [\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ \chi :: S_0 \circ [\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\})$, if $\alpha' = \alpha$;
- $K_0 \vdash_\chi S_0 \circ [\chi/\alpha](root(\chi)) :: S_0 \circ [\chi/\alpha](\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0(root(\chi)) :: S_0 \circ [\chi/\alpha](\{\{F_2^l \parallel F_2^r\}\})$, if $\alpha' = root(\chi)$.

The last two points are valid since $dom(F_1^l) \cap dom(F_{c(\chi)}) = \emptyset$, $dom(F_1^r) \cap dom(F_2^r + (F_2^l - F_{c(\chi)})) = \emptyset$, and $K_0 \vdash_\chi S_0(root(\chi)) :: S_0([\chi/\alpha](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))$, if $\alpha' = root(\chi)$. Also, since $S_0 = S_0 \circ [\chi/\alpha]$, we have that, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha, \chi)\} \cup S$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, that is:

- if, for all $(\tau_1, \tau_2) \in [\chi/\alpha](E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ [\chi/\alpha](\tau_1) \equiv_\chi S_0 \circ [\chi/\alpha](\tau_2)$;
- $S_0(\alpha) \equiv_\chi S_0 \circ [\chi/\alpha](\alpha) \equiv_\chi S_0(\chi)$;
- if, for all $(\tau_1, \tau_2) \in [\chi/\alpha](S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in S$, $S_0 \circ [\chi/\alpha](\tau_1) = S_0 \circ [\chi/\alpha](\tau_2)$.

That is, (K_0, S_0) respects $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\}$ and S_0 satisfies $E \cup \{(\alpha, \chi)\} \cup S$. But, then (K_0, S_0) respects SK, that is, for all $\alpha' \in dom(SK)$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(SK(\alpha'))$. But, since $\alpha \notin FTV(\chi)$, then α will not appear in $[\chi/\alpha](SK)$ and $dom([\chi/\alpha](SK)) = dom(SK)$, therefore, for all $\alpha' \in dom([\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0([\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}(\alpha'))$, since we know that $K_0 \vdash_\chi S_0(\alpha) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ [\chi/\alpha](\alpha) :: S_0 \circ [\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0\chi :: S_0 \circ [\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\})$. Therefore, (K_0, S_0) respects $[\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}\}$.

- Rule *viii)*: Assume (K_0, S_0) respects K. If we assume that S_0 satisfies $E \cup \{(\alpha_i^1, \tau_j^2), (\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \{l_{i+1}^1 : \tau_{i+1}^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \{l_{j+1}^2 : \tau_{j+1}^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$, then we know that:
  - S_0 satisfies E;
  - S_0 satisfies S;
  - S_0 satisfies $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}$.

  Therefore, S_0 satisfies $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \{l_{i+1}^1 : \tau_{i+1}^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \{l_{j+1}^2 : \tau_{j+1}^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$ and (K_0, S_0) respects SK.

- Rule *ix)*: If we assume that (K_0, S_0) respects $\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}$, then, for all $\alpha' \in dom(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\}(\alpha'))$, that is:

  - $K_0 \vdash_\chi S_0(\alpha') :: S_0(\mathcal{I}(K)(\alpha'))$, if $\alpha' = \alpha$;
  - $K_0 \vdash_\chi S_0(\alpha) :: S_0(\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))$.

  If we assume that S_0 satisfies, $\mathcal{I}(E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}) \cup \mathcal{I}(S) \cup \{(\alpha^1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha^2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\}$, that is:

  - for all $(\tau_1, \tau_2) \in \mathcal{I}(E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in \mathcal{I}(\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\})$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - for all $(\tau_1, \tau_2) \in \mathcal{I}(S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$;
  - $S_0(\alpha^1) \equiv_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})$;
  - $S_0(\alpha^2) \equiv_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$.

  But then, since $S_0(\alpha^1) \equiv_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})$, $S_0(\alpha^2) \equiv_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$, $\alpha^1 \notin FTV(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})$, and $\alpha^2 \notin FTV(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$, we have that $S_0 \circ \mathcal{I}(\alpha^1) \equiv_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \equiv_\chi S_0 \circ \mathcal{I}(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})$, and $S_0 \circ \mathcal{I}(\alpha^2) \equiv_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0 \circ \mathcal{I}(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$. Also, both α^1 and α^2 will not appear in $\mathcal{I}(K)$ and $dom(\mathcal{I}(K)) = dom(K)$, therefore, for all $\alpha' \in dom(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha'))$, since:

  - $K_0 \vdash_\chi S_0(\alpha') :: S_0(K(\alpha')) \iff K_0 \vdash_\chi S_0 \circ \mathcal{I}(\alpha') :: S_0(\mathcal{I}(K)(\alpha'))$, if $\alpha' \notin \{\alpha^1, \alpha^2\}$;
  - $K_0 \vdash_\chi S_0(\alpha^1) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(S_{ix}(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}))S_0(\mathcal{I}(\{\{F_1^l \parallel F_1^r\}\}))$;
  - $K_0 \vdash_\chi S_0(\alpha^2) :: S_0(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0(\mathcal{I}(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}))S_0(\mathcal{I}(\{\{F_2^l \parallel F_2^r\}\}))$.

  And the two last points are valid, since $dom(F_1^l) \cap dom(F_2^r) = \emptyset$, $dom(F_1^r) \cap dom(F_2^l) = \emptyset$, and $S_0(\alpha) :: S_0(\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))$. Also, since $S_0 = S_0 \circ \mathcal{I}$, we have that, for all $(\tau_1, \tau_2) \in E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$, $S_0(\tau_1) = S_0(\tau_2)$, since:

  - if, for all $(\tau_1, \tau_2) \in \mathcal{I}(E)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in E$, $S_0 \circ \mathcal{I}(\tau_1) \equiv_\chi S_0 \circ \mathcal{I}(\tau_2)$;
  - if, for all $(\tau_1, \tau_2) \in \mathcal{I}(S)$, $S_0(\tau_1) \equiv_\chi S_0(\tau_2)$, then, for all $(\tau_1, \tau_2) \in S$, $S_0 \circ \mathcal{I}(\tau_1) \equiv_\chi S_0 \circ \mathcal{I}(\tau_2)$;
  - $S_0(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0 \circ \mathcal{I}(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\} \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0 \circ \mathcal{I}(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) \equiv_\chi S_0(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})$.

This means that (K_0, S_0) respects $K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}$, and S_0 satisfies $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$. Therefore (K_0, S_0) respects SK. But, α^1 and α^2 will not appear in $\mathcal{I}(SK)$ and $dom(\mathcal{I}(SK)) = dom(SK)$, therefore, for all $\alpha' \in dom(\mathcal{I}(SK))$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(\mathcal{I}(SK))$. Also, we know that:

- $K_0 \vdash_\chi S_0(\alpha^1) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ \mathcal{I}(\alpha^1) :: S_0(\mathcal{I}(\{\{F_1^l \parallel F_1^r\}\})) \iff K_0 \vdash_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}) :: S_0(\mathcal{I}(\{\{F_1^l \parallel F_1^r\}\}))$;
- $K_0 \vdash_\chi S_0(\alpha^2) :: S_0(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0 \circ \mathcal{I}(\alpha^2) :: S_0(\mathcal{I}(\{\{F_2^l \parallel F_2^r\}\})) \iff K_0 \vdash_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) :: S_0(\mathcal{I}(\{\{F_2^l \parallel F_2^r\}\}))$.

Therefore, for all $\alpha' \in dom(\mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\})$, $K_0 \vdash_\chi S_0(\alpha') :: S_0(\mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}(\alpha'))$, that is, (K_0, S_0) respects $\mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}$.

We only show that the set of unifiers of $(K \cup SK, E \cup S)$ is preserved by the transformation rules from left to right, but similar arguments can be used to show that it is also preserved by the transformation rules from right to left.

Preservation of ***Property 3***:

- Rule *i)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup SK$ and such that S_0 satisfies $E \cup \{(\tau_1, \tau_2)\} \cup S$, if $\tau_1 \equiv_\chi \tau_2$. Then S_0 satisfies $E \cup S$.
- Rule *ii)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup \{(\alpha, \mathcal{U})\} \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha, \tau)\} \cup S$. Since $\alpha \notin FTV(\tau)$, $dom(K) = dom([\tau/\alpha](K))$ and $dom(SK) = dom([\tau/\alpha](SK))$, therefore, (K_0, S_0) respects $[\tau/\alpha](K) \cup [\tau/\alpha](SK) \cup \{(\alpha, \mathcal{U})\}$. Also, since $S_0(\alpha) \equiv_\chi S_0(\tau)$, then $S_0 = S_0 \circ [\tau/\alpha]$, therefore S_0 also satisfies $[\tau/\alpha](E) \cup [\tau/\alpha](S) \cup \{(\alpha, \tau)\}$.
- Rule *iii)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha_2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha_1, \alpha_2)\} \cup S$. But, $dom(K) = dom([\alpha_2/\alpha_1](K))$ and $dom(SK) = dom([\alpha_2/\alpha_1](SK))$, therefore (K_0, S_0) respects $[\alpha_2/\alpha_1](SK) \cup [\alpha_2/\alpha_1](SK)$. Also, since $S_0(\alpha_1) \equiv_\chi S_0(\alpha_2)$, then $S_0 = S_0 \circ [\alpha_2/\alpha_1]$, and:
  - $K_0 \vdash_\chi S_0(\alpha_1) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(\alpha_1) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_1^l \parallel F_1^r\}\})$;
  - $K_0 \vdash_\chi S_0(\alpha_2) :: S_0(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0(\alpha_2) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_2^l \parallel F^r\}\})$.

  But then $K_0 \vdash_\chi S_0(\alpha_2) :: S_0 \circ [\alpha_2/\alpha_1](\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\})$ and S_0 satisfies $[\alpha_2/\alpha_1](\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}$, since $dom(F_1^l) \cap dom(F_2^r) = \emptyset$, and $dom(F_1^r) \cap dom(F_2^l) = \emptyset$. Therefore (K_0, S_0) respects $[\alpha_2/\alpha_1](K) \cup [\alpha_2/\alpha_1](SK) \cup \{(\alpha_1, \{\{F_1^l \parallel F_1^r\}\})\}$. Also, S_0 satisfies $[\alpha_2/\alpha_1](E)$, $[\alpha_2/\alpha_1](S)$, and $\{(\alpha_1, \alpha_2)\}$, therefore, S_0 satisfies $[\alpha_2/\alpha_1](E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\}) \cup [\alpha_2/\alpha_1](S) \cup \{(\alpha_1, \alpha_2)\}$.

- Rule *iv)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\} \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha, \{F_2\})\} \cup S$. Since $\alpha \notin FTV(\{F_2\})$, we have that $dom(K) = dom([\{F_2\}/\alpha](K))$ and that $dom(SK) = dom([\{F_2\}/\alpha](SK))$, therefore (K_0, S_0) respects $[\{F_2\}/\alpha](K) \cup [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$. Also, since $S_0(\alpha) \equiv_\chi S(\{F_2\})$, then $S_0 = S_0 \circ [\{F_2\}/\alpha]$ and S_0 satisfies $[\{F_2\}/\alpha](E)$, $[\{F_2\}\alpha](S)$, and $\{(\alpha, \{F_2\})\}$. Finally, we know that $K_0 \vdash_\chi S_0(\alpha) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ [\{F_2\}/\alpha](\alpha) :: S_0 \circ [\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(\{F_2\}) :: S_0([\{F_2\}/\alpha](\{\{F_1^l \parallel F_1^r\}\}))$, therefore, S_0 satisfies $[\{F_2\}/\alpha](\{(F_1^l(l), F_1^r(l)) \mid l \in dom(F_1^l)\}$, since $dom(F_1^l) \subseteq dom(F_2)$ and $dom(F_1^r) \cap dom(F_2) = \emptyset$. This means that (K_0, S_0) respects $[\{F_2\}/\alpha](K) \cup [\{F_2\}/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$ and S_0 satisfies $[\{F_2\}/\alpha](E) \cup [\{F_2\}/\alpha](S) \cup \{(\alpha, \{F_2\})\}$.

- Rule *v)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup SK$ and such that S_0 satisfies $E \cup \{(\{F_1\}, \{F_2\})\} \cup S$. Since $S_0(\{F_1\}) \equiv_\chi S_0(\{F_2\})$ and $dom(\{F_1\}) = dom(\{F_2\})$, then S_0 satisfies $\{(F_1(l), F_2(l)) \mid l \in dom(F_1)\}$, therefore S_0 satisfies $E \cup \{(F_1(l), F_2(l)) \mid l \in dom(F_1)\} \cup S$.

- Rule *vi)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup SK$ and such that S_0 satisfies $E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\} \cup S$. Since S_0 satisfies $\{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}$, then it also satisfies $\{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}$, therefore S_0 satisfies $E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\} \cup S$.

- Rule *vii)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\}), (root(\chi), \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha, \chi)\} \cup S$. Since $\alpha \notin FTV(\chi)$, we have that $dom(K) = dom([\chi/\alpha](K))$, $dom(SK) = dom([\chi/\alpha](SK))$, therefore (K_0, S_0) respects $[\chi/\alpha](K) \cup [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$. Also, since $S_0(\alpha) \equiv_\chi S_0(\chi)$, then $S_0 = S_0 \circ [\chi/\alpha]$, and:

  - $K_0 \vdash_\chi S_0(\alpha) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ [\chi/\alpha](\alpha) :: S_0 \circ [\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(\chi) :: S_0([\chi/\alpha](\{\{F_1^l \parallel F_1^r\}\}))$;
  - $K_0 \vdash_\chi S_0(root(\chi)) :: S_0(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0(root(\chi)) :: S_0([\chi/\alpha](\{\{F_2^l \parallel F_2^r\}\}))$.

  But then, S_0 satisfies $[\chi/\alpha](\{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)})\})$ and $K_0 \vdash_\chi S_0(root(\chi)) :: S_0([\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2 - F_{c(\chi)}))) \parallel F_2^r + (F_1^r - F_{c(\chi)})\}\})$, since $dom(F_1^l) \cap dom(F_{c(\chi)}) = \emptyset$ and $dom(F_1^r) \cap dom(F^r + (F_2^l - F_{c(\chi)})) = \emptyset$. Finally, since $\alpha \notin FTV(\chi)$, then α will not appear in $[\chi/\alpha](E)$ or $[\chi/\alpha](S)$, therefore S_0 satisfies $[\chi/\alpha](E) \cup [\chi/\alpha](S) \cup \{(\alpha, \chi)\}$. But then, (K_0, S_0) respects $[\chi/\alpha](K) \cup \{(root(\chi), [\chi/\alpha](\{\{F_2^l + (F_1^l - (F_2^r + (F_2^l - F_{c(\chi)}))) \mid \mid F_2^r + (F_1^r - F_{c(\chi)})\}\}))\} \cup [\chi/\alpha](SK) \cup \{(\alpha, \{\{F_1^l \parallel F_1^r\}\})\}$ and S_0 satisfies $[\chi/\alpha](E \cup \{(F_1^l(l), (F_2^r + (F_2^l - F_{c(\chi)}))(l)) \mid l \in dom(F_1^l) \cap dom(F_2^r + (F_2^l - F_{c(\chi)}))\} \cup \{(F_1^r(l), F_{c(\chi)}(l)) \mid l \in dom(F_1^r) \cap dom(F_{c(\chi)})\}) \cup [\chi/\alpha](S) \cup \{(\alpha, \chi)\}$.

- Rule *viii)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$. Since S_0 satisfies $\{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_i^1 \{l_i^1 : \tau_i^1\} \cdots \pm_n^1 \{l_n^1 :$

$\tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_j^2 \{l_j^2 : \tau_j^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}$, then it also satisfies $\{(\tau_i^1, \tau_j^2), (\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \{l_{i+1}^1 : \tau_{i+1}^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \{l_{j+1}^2 : \tau_{j+1}^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\}$, since $(\pm_i^1 = \pm_j^2 \wedge l_i^2 = l_j^2)$, $\forall i < k \leq n : l_k^1 \neq l_i^1$, and $\forall j < r \leq m : l_r^2 \neq l_j^2$. Therefore, S_0 satisfies $E \cup \{(\tau_i^1, \tau_j^2), (\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_{i-1}^1 \{l_{i-1}^1 : \tau_{i-1}^1\} \pm_{i+1}^1 \{l_{i+1}^1 : \tau_{i+1}^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_{j-1}^2 \{l_{j-1}^2 : \tau_{j-1}^2\} \pm_{j+1}^2 \{l_{j+1}^2 : \tau_{j+1}^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\})\} \cup S$.

- Rule *ix)*: Let (K_0, S_0) be a kinded substitution that respects $K \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\} \cup SK$ and such that S_0 satisfies $E \cup \{(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})\} \cup S$. Since $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$ and $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}, \alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$, we have that $dom(K) = dom(\mathcal{I}(K))$, $dom(SK) = dom(\mathcal{I}(SK))$ and (K_0, S_0) respects $\mathcal{I}(K) \cup \mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1 \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}$. Also, since $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$, $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$, α fresh, and $S_0(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$, then $S_0 = S_0 \circ \mathcal{I}$, and:

  - $K_0 \vdash_\chi S_0(\alpha^1) :: S_0(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0 \circ \mathcal{I}(\alpha^1) :: S_0 \circ \mathcal{I}(\{\{F_1^l \parallel F_1^r\}\}) \iff K_0 \vdash_\chi S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\} \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) :: S_0(\mathcal{I}(\{\{F_1^l \parallel F_1^r\}\}))$;
  - $K_0 \vdash_\chi S_0(\alpha^2) :: S_0(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0 \circ \mathcal{I}(\alpha^2) :: S_0 \circ \mathcal{I}(\{\{F_2^l \parallel F_2^r\}\}) \iff K_0 \vdash_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\} \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\}) :: S_0(\mathcal{I}(\{\{F_2^l \parallel F_2^r\}\}))$.

  Note that, since $\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\} \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\} \equiv_\chi \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\} \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\}$, then $S_0(\alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\} \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\}) \equiv_\chi S_0(\alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\} \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$. But then, S_0 satisfies $\mathcal{I}(\{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l \in dom(F_1^r) \cap dom(F_2^r)\})$ and $K_0 \vdash_\chi S_0(\alpha) :: S_0(\mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))$, since $dom(F_1^l) \cap dom(F_2^r) = \emptyset$ and $dom(F_1^r) \cap dom(F_1^l) = \emptyset$. Finally, since $\alpha^1 \notin FTV(\alpha^2 \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\})$, $\alpha^2 \notin FTV(\alpha^1 \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})$, α fresh, and $\forall 1 \leq i \leq n, 1 \leq j \leq m : l_i^1 \neq l_j^1$, S_0 satisfies $\mathcal{I}(E) \cup \mathcal{I}(S) \cup \{(\alpha^1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau^2\}), (\alpha^2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\}$. But then, (K_0, S_0) respects $\mathcal{I}(K) \cup \{(\alpha, \mathcal{I}(\{\{F_1^l + F_2^l \parallel F_1^r + F_2^r\}\}))\} \cup \mathcal{I}(SK) \cup \{(\alpha^1, \{\{F_1^l \parallel F_1^r\}\}), (\alpha^2, \{\{F_2^l \parallel F_2^r\}\})\}$, and S_0 satisfies $\mathcal{I}(E \cup \{(F_1^l(l), F_2^l(l)) \mid l \in dom(F_1^l) \cap dom(F_2^l)\} \cup \{(F_1^r(l), F_2^r(l)) \mid l dom(F_1^r) \cap dom(F_2^r)\}) \cup \mathcal{I}(S) \cup \{(\alpha^1, \alpha \pm_1^2 \{l_1^2 : \tau_1^2\} \cdots \pm_m^2 \{l_m^2 : \tau_m^2\}), (\alpha^2, \alpha \pm_1^1 \{l_1^1 : \tau_1^1\} \cdots \pm_n^1 \{l_n^1 : \tau_n^1\})\}$.

Using the three previous properties, we can conclude the correctness of the algorithm. Let (K, E) be a given kinded set of equations. Suppose the algorithm terminates with (K', S). Then there is some SK such that $(E, K, \emptyset, \emptyset)$ is transformed to $(\emptyset, K', S, SK)$ by repeated applications of the transformation rules. ***Property 1*** trivially holds for $(E, K, \emptyset, \emptyset)$, which means that (K', S) is a kinded substitution, and $dom(S) \cap dom(K') = \emptyset$. Therefore (K', S) respects K'. S also trivially satisfies $S \cup \emptyset$, therefore, by ***Property 2***, (K', S) also respects SK, therefore, (K', S) is a unifier of $(K' \cup SK, \emptyset \cup S)$. By ***Property 3***, (K', S) is also a unifier of $(K' \cup SK, \emptyset \cup S)$. Let

(K_0, S_0) be any unifier of (K, E). By ***Property 3***, it is also a unifier of $(K' \cup SK, \emptyset \cup S)$. But, then $S_0 = S_0 \circ S$ and (K', S) is more general than (K_0, S_0). Conversely, suppose the algorithm fails. Then $(E, K, \emptyset, \emptyset)$ is transformed to (E', K', S', SK') for some E', K', S', SK' such that $E' \neq \emptyset$, and no rule applies to (E', K', S', SK'). It is clear from the definition of each rule that $(K', SK', E' \cup S')$ has no unifier, and therefore, by ***Property 3***, that (K, E) has no unifier. The termination can be proved by showing that each transformation rule decreases the complexity measure of the lexicographical pair consisting of the size of the set $dom(K)$ and the total number of occurrences of type constructors (including base types) in E. □

Bibliography

[1] Sandra Alves and Miguel Ramos. An ml-style record calculus with extensible records (long version), 2021. (https://arxiv.org/abs/2108.06296).

[2] Sandra Alves, Sabine Broda, and Maribel Fernández. A typed language for events. In Moreno Falaschi, editor, ***Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers***, volume 9527 of ***Lecture Notes in Computer Science***, pages 107–123. Springer, 2015. doi:10.1007/978-3-319-27436-2_7.

[3] Sandra Alves, Anatoli Degtyarev, and Maribel Fernández. Access Control and Obligations in the Category-Based Metamodel: A Rewrite-Based Semantics. In ***Proceedings of LOPSTR'14***, volume 8981 of ***LNCS***, pages 148–163. Springer, 2015.

[4] Sandra Alves, Maribel Fernández, and Miguel Ramos. EVL: A typed higher-order functional language for events. ***Electronic Notes in Theoretical Computer Science***, 351:3–23, September 2020. doi:10.1016/j.entcs.2020.08.002.

[5] Alexander Artikis, Alessandro Margara, Martin Ugarte, Stijn Vansummeren, and Matthias Weidlich. Complex event recognition languages: Tutorial. In ***Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems***, DEBS '17, page 7–10, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450350655. doi:10.1145/3093742.3095106.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In ***Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems***, PODS '02, page 1–16, New York, NY, USA, 2002. Association for Computing Machinery. ISBN: 1581135076. doi:10.1145/543613.543615.

[7] Hendrik Pieter Barendregt. ***The lambda calculus - its syntax and semantics***, volume 103 of ***Studies in logic and the foundations of mathematics***. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[8] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In ***CIDR 2007, Third***

Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings, pages 363–374. www.cidrdb.org, 2007.

[9] Steve Barker, Marek J. Sergot, and Duminda Wijesekera. Status-Based Access Control. ***ACM Transactions on Information and System Security***, 12(1):1:1–1:47, 2008.

[10] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A Temporal Role-based Access Control Model. ***ACM Transactions on Information and System Security***, 4 (3):191–233, August 2001.

[11] Clara Bertolissi, Maribel Fernández, and Steve Barker. Dynamic event-based access control as term rewriting. In ***Proceedings of the 21st Annual IFIP WG 11.3 Working Conference on Data and Applications Security***, page 195–210, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN: 9783540735335.

[12] Luca Cardelli. A semantics of multiple inheritance. ***Inf. Comput.***, 76(2/3):138–164, 1988. doi:10.1016/0890-5401(88)90007-7.

[13] Luca Cardelli. ***Extensible Records in a Pure Calculus of Subtyping***, page 373–425. MIT Press, Cambridge, MA, USA, 1994. ISBN: 026207155X.

[14] Luca Cardelli and John C. Mitchell. Operations on records. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, ***Mathematical Foundations of Programming Semantics***, pages 22–52, New York, NY, 1990. Springer New York. ISBN: 978-0-387-34808-7.

[15] Luca Cardelli and John C. Mitchell. Operations on records. ***Math. Struct. Comput. Sci.***, 1(1):3–48, 1991. doi:10.1017/S0960129500000049.

[16] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ***ACM Comput. Surv.***, 17(4):471–522, 1985. doi:10.1145/6041.6042.

[17] Mani K. Chandy, Opher Etzion, and Rainer von Ammon. The event processing manifesto. In K. Mani Chandy, Opher Etzion, and Rainer von Ammon, editors, ***Event Processing***, number 10201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[18] Alonzo Church. A set of postulates for the foundation of logic. ***The Annals of Mathematics***, 33(2):346, April 1932. doi:10.2307/1968337.

[19] Alonzo Church. A formulation of the simple theory of types. ***Journal of Symbolic Logic***, 5(2):56–68, June 1940. doi:10.2307/2266170.

[20] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In ***Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems***, DEBS '10, page 50–61, New York, NY, USA, 2010. Association for Computing Machinery. ISBN: 9781605589275. doi:10.1145/1827418.1827427.

[21] H. B. Curry. Functionality in combinatory logic. ***Proceedings of the National Academy of Sciences***, 20(11):584–590, November 1934. doi:10.1073/pnas.20.11.584.

[22] H. B. Curry and R. Feys. ***Combinatory Logic Vol. 1***. North-Holland Publishing Company, 1958.

[23] Luís Damas. ***Type assignment in programming languages***. PhD thesis, University of Edinburgh, UK, 1984.

[24] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In Richard A. DeMillo, editor, ***Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982***, pages 207–212. ACM Press, 1982. doi:10.1145/582153.582176.

[25] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. A CEP babelfish: Languages for complex event processing and querying surveyed. In ***Reasoning in Event-Based Distributed Systems***, pages 47–70. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-19724-6_3.

[26] Opher Etzion and Peter Niblett. ***Event Processing in Action***. Manning Publications Co., USA, 1st edition, 2010. ISBN: 1935182218.

[27] Antony Galton and Juan Carlos Augusto. Two approaches to event definition. In Abdelkader Hameurlain, Rosine Cicchetti, and Roland Traunmüller, editors, ***Database and Expert Systems Applications***, pages 547–556, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN: 978-3-540-46146-3.

[28] Benedict R. Gaster. ***Records, variants and qualified types***. PhD thesis, University of Nottingham, UK, 1998.

[29] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, 1996.

[30] Michael Gelfond and Jorge Lobo. Authorization and obligation policies in dynamic systems. In ***ICLP***, pages 22–36, 2008.

[31] Alejandro Grez, Cristian Riveros, and Martín Ugarte. A Formal Framework for Complex Event Processing. In Pablo Barcelo and Marco Calautti, editors, ***22nd International Conference on Database Theory (ICDT 2019)***, volume 127 of ***Leibniz International Proceedings in Informatics (LIPIcs)***, pages 5:1–5:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-101-6. doi:10.4230/LIPIcs.ICDT.2019.5.

[32] Robert Harper and John C. Mitchell. On the type structure of standard ML. ***ACM Trans. Program. Lang. Syst.***, 15(2):211–252, 1993. doi:10.1145/169701.169696.

[33] Robert Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. In David S. Wise, editor, ***Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991***, pages 131–142. ACM Press, 1991. doi:10.1145/99583.99603.

[34] Robert William Harper and Benjamin C. Pierce. Extensible records without subsumption, 2000. doi:10.1184/R1/6605507.V1.

[35] Manuel Hilty, Alexander Pretschner, David A. Basin, Christian Schaefer, and Thomas Walter. A Policy Language for Distributed Usage Control. In ***Proceedings of ESORICS'07***, pages 531–546, 2007.

[36] J. Roger Hindley. ***Basic Simple Type Theory***. Cambridge University Press, July 1997. doi:10.1017/cbo9780511608865.

[37] Lalita A. Jategaonkar and John C. Mitchell. Type inference with extended pattern matching and subtypes. ***Fundam. Inf.***, 19(1–2):127–165, September 1993. ISSN: 0169-2968.

[38] Mark P. Jones. A theory of qualified types. ***Sci. Comput. Program.***, 22(3):231–256, 1994. doi:10.1016/0167-6423(94)00005-0.

[39] Mark P. Jones. ***Qualified Types: Theory and Practice***. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994. doi:10.1017/CBO9780511663086.

[40] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. ***The Annals of Mathematics***, 36(3):630, July 1935. doi:10.2307/1968646.

[41] R. Kowalski and M. Sergot. A logic-based calculus of events. ***New Generation Computing***, 4(1):67–95, 1986.

[42] Daan Leijen. Extensible records with scoped labels. In Marko C. J. D. van Eekelen, editor, ***Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005***, volume 6 of ***Trends in Functional Programming***, pages 179–194. Intellect, 2005.

[43] David Luckham. ***The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems***. Addison-Wesley, Boston, MA, 2002. ISBN: 978-0-201-72789-0.

[44] Rob Miller and Murray Shanahan. The event calculus in classical logic - alternative axiomatisations. ***Electron. Trans. Artif. Intell.***, 3(A):77–105, 1999.

[45] Robin Milner. A theory of type polymorphism in programming. ***Journal of Computer and System Sciences***, 17(3):348–375, December 1978. doi:10.1016/0022-0000(78)90014-4.

[46] Atsushi Ohori. A polymorphic record calculus and its compilation. ***ACM Trans. Program. Lang. Syst.***, 17(6):844–895, 1995. doi:10.1145/218570.218572.

[47] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. ***Journal of Functional Programming***, 4(2):207–247, April 1994. doi:10.1017/s0956796800001040.

[48] Didier Rémy. Typechecking records and variants in a natural extension of ML. In ***Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989***, pages 77–88. ACM Press, 1989. doi:10.1145/75277.75284.

[49] Didier Rémy. Typing record concatenation for free. In Ravi Sethi, editor, ***Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992***, pages 166–176. ACM Press, 1992. doi:10.1145/143165.143202.

[50] Didier Rémy. ***Type Inference for Records in Natural Extension of ML***, page 67–95. MIT Press, Cambridge, MA, USA, 1994. ISBN: 026207155X.

[51] J. A. Robinson. A machine-oriented logic based on the resolution principle. ***Journal of the Association for Computing Machinery (ACM)***, 12:23–41, 1965.

[52] Mitchell Wand. Complete type inference for simple objects. In ***Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987***, pages 37–44. IEEE Computer Society, 1987.

[53] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In ***Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988***, page 132. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5111.

[54] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In ***Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989***, pages 92–97. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39162.

[55] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In ***Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)***, pages 392–399, 1999.