Kind Inference for the FreeST Programming Language

Bernardo Almeida Andreia Mordido
Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
{bpdalmeida,afmordido,vmvasconcelos}@ciencias.ulisboa.pt

We present a kind inference algorithm for the FREEST programming language. The input to the algorithm is FREEST source code with (possibly part of) kind annotations replaced by kind variables. The algorithm infers concrete kinds for all kind variables. We ran the algorithm on the FREEST test suite by first replacing kind annotation on all type variables by fresh kind variables, and concluded that the algorithm correctly infers all kinds. Non surprisingly, we found out that programmers do not choose the most general kind in 20% of the cases.

1 Introduction

Software systems usually handle resources such as files and communication channels. The correct usage of such resources generally follows a protocol that describes valid patterns of interactions. For example a file should be opened and eventually closed, after which no read or write operations should ever be performed. The case for communication channels is similar: channels are opened, messages are exchanged, channels may eventually be closed, after which no more messages should be exchanged. Session types [6, 7, 15] allow expressing elaborate protocols (for files and channels, for example) guaranteeing that protocols are obeyed by programs.

FREEST [1, 2, 3] is a concurrent functional programming language based on System F where processes communicate via heterogeneously typed-channels governed by context-free session types [16]. Context-free session types allow describing protocols such as the serialization of arithmetic expressions. Consider the following datatype for arithmetic expressions.

```
1 data Exp = Lit Int | Plus Exp Exp | Times Exp Exp
```

An Exp is either a literal with an integer (Lit Int), a sum of two sub-expressions (Plus Exp Exp) or the product of two sub-expressions (Times Exp Exp). To serialise a value of type Exp we use a session type such as the following.

```
2 type ExpC = ⊕{LitC: !Int, PlusC: ExpC;ExpC, TimesC: ExpC;ExpC}
```

The abbreviation ExpC defines the type of a channel as seen from the point of view of the writer. A channel of type ExpC offers a set of options LitC, PlusC and TimesC. If the first option is chosen, an integer must be sent (!Int), while, in the others, two (sub-) expressions are expected to be sent.

Now, suppose that serialise is a function that serialises an Exp on a channel ExpC.

```
3 serialise : Exp → ExpC;a → a
```

The function expects a channel whose initial part is of type ExpC and then behaves as a: serialise is thus polymorphic on a. It consumes the front of the channel (of type ExpC) and returns the unused part of the channel (of type a).

Ilaria Castellani and Alceste Scalas (Ed.): 14th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2023).
EPTCS 378, 2023, pp. 1–13, doi:10.4204/EPTCS.378.1

As simple as it may seem, the above code is not valid in the current version of FreeST. The actual code requires further annotations allowing to distinguish functional from session types as well as linear from unrestricted types. The distinction is materialised by classifying types with kinds.

In FreeST kinds are composed of a multiplicity and a basic kind. Multiplicities control the number of times a value may be used: exactly once (linear, 1) or zero or more (unrestricted, $*$). Basic kinds distinguish functional types (T) from session types (S). The reason why FreeST requires kinds lies on polymorphism. If !**Int**;?**Int** is undoubtedly a session type and **Int** $\rightarrow$ **Bool** a functional type, the same does not apply to the polymorphic variable a. Is it a session type or a functional type? The answer depends on the base kind of a: if S or then it is a session type, if T then it is a functional type. Kinds are thus necessary to decide whether the types such as a;!**Int** are well-formed.

The datatype defined in line 1 is currently written in annotated form as follows.

```
4 data Exp:*T = Lit Int | Plus Exp Exp | Times Exp Exp
```

The kind annotation $*$T, says that the datatype is functional. As for the multiplicity, we chose the unrestricted usage so that it may be used as often as required. Notwithstanding, one may declare Exp of kind 1T, in which case serialise must become a linear function (of type Exp $\rightarrow$ ExpC;a $1\rightarrow$ a).

Expanding the abbreviation and annotating the datatype in line 2 we get the following type.

```
5 type ExpC:1S = rec a:1S . ⊕{LitC: !Int, PlusC: a;a, TimesC: a;a}
```

ExpC defines a recursive type that is well-formed when the kind of its body, the external choice ($\oplus$), is a subkind of the kind for the recursion variable. In this case, the recursion variable ExpC is annotated with 1S, given that its body is itself a linear session.

Finally, the function serialise is currently written as follows.

```
6 serialise : ∀a:1S . Exp → ExpC;a → a
```

The polymorphic variable a stands for the continuation channel; it must be a linear session. Annotating a with the unrestricted session $*$S would dictate that it can only be instantiated with Skip, the only unrestricted session type.

Even if kinds are necessary in the underlying theory of the FreeST language, they clutter the code. The code in lines 1–3 is easier to understand and quicker to write; programmers need not fight the subtleties of each kind. Note that once kinds are inferred, the prenex occurrences of $\forall$ can be omitted. The algorithm that we present in this paper annotates all type variables with their kinds, converting the code in lines 1–3 to that in lines 4–6.

The works more closely related to FreeST are Quill [9], Affe [13], Alms [17], F$^\circ$ [8], FuSe$^{\{\}}$ [11] and Linear Haskell [4]. All these languages feature substructural type systems for dealing with linear, functional and affine types (in the case of Affe).

Quill [9] is a language with linear types and a syntax similar to that of Haskell. Quill features a novel design that combines linear and functional types. Contrarily to FreeST, Quill does not use kind mechanisms to distinguish between linear and functional types, instead it uses type predicates (or, qualified types) to reason about linearity. Furthermore, Quill does not support subkinding. Quill also has a type inference algorithm which was proven sound and complete. Affe [13] is an ML-like language with support to linear, affine and unrestricted types. Like Quill, Affe uses kinds and constrained types to distinguish between linear and affine types. Affe supports subkinding and it is equipped with full principal type inference. Like Affe, Alms [17] is an ML-like language but is based on System $F^{\omega}_{<:}$, the higher-order polymorphic λ-calculus with subtyping. Alms supports affine and unrestricted types. It features a rich kind system with dependent kinds, unions, and intersections. Moreover, Alms supports ML modules, allows to expose unrestricted types as affine which gives flexibility to library programmers

$$
\begin{array}{rcll}
m & ::= & * \mid \mathtt{1} \mid \varphi & \text{Multiplicity} \\
\upsilon & ::= & \mathtt{S} \mid \mathtt{T} & \text{Prekind} \\
\kappa & ::= & m\upsilon \mid \chi & \text{Kind} \\
\sharp & ::= & ! \mid ? & \text{Polarity} \\
\star & ::= & \oplus \mid \& & \text{View} \\
(\!|\cdot|\!) & ::= & \{\cdot\} \mid \langle\cdot\rangle & \text{Record} \\
T & ::= & \mathsf{Skip} \mid \mathsf{End} \mid \sharp T \mid \star(\!|\ell\colon T_\ell|\!)_{\ell\in L} \mid T;T \mid ()_m & \text{Type} \\
 & & \mid T\ m\!\!\rightarrow T \mid (\!|\ell\colon T_\ell|\!)_{\ell\in L} \mid \forall a^\kappa.T \mid \mu\, a^\kappa.T \mid a & \\
e & ::= & ()_m \mid x \mid \lambda_m x\colon T.e \mid \Lambda a^\kappa.v \mid e\ e \mid \{\ell{=}e_\ell\}_{\ell\in L} \mid \mathsf{let}\ \{\ell{=}x_\ell\}_{\ell\in L} = e\ \mathsf{in}\ e & \text{Expression} \\
 & & \mid \ell\ e \mid \mathsf{let}\ ()_m = e\ \mathsf{in}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L} \mid e[T] \mid \mathsf{match}\ e\ \mathsf{with}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L} &
\end{array}
$$

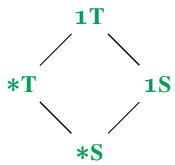
Figure 1: The syntax of kinds and types with support for kind inference

and it is equipped with local type inference. F° [8] is an extension of System F that uses kinds to distinguish between linear and unrestricted types. Similarly to Affe and Alms, it supports subkinding. Similarly to FreeST, but unlike Affe, F° does not support quantification over kinds. The work closest to FreeST in terms of context-free session types is FuSe$^{\{\}}$ [11]. Padovani proposed an alternative formulation of context-free session types in which code and types are aligned via extra annotations, something we decided to avoid in FreeST. Linear Haskell [4] is a proposal to bring linear types to Haskell. In Linear Haskell functions $T \rightarrow U$ and $T \multimap U$ describe how the arguments of the function are used. The latter form, inspired by linear logic [5], uses the argument T exactly once. In FreeST, annotated arrows $T *\!\!\rightarrow U$ or $T\ \mathtt{1}\!\!\rightarrow U$ describe how the function is used (unbounded usage or exactly once). FreeST kinding system differentiates session from functional types. It also classifies types according to their usage, linear or unrestricted. Other systems consider these notions separately (or only one of them). The ideas behind our inference algorithm are similar to Quill and Affe, but the details are quite different since we do not use type qualifiers to reason about linearity.

2 The Syntax of Kinds, Types and Expressions

This section briefly introduces the notions of kinds, types and expressions; we refer the interested reader to previous work for details [1]. FreeST relies on a base set for type variables (denoted by a, b, c) and another for labels (denoted by k, ℓ). For the purpose of kind inference, we further use *multiplicity variables* (denoted by φ) and *kind variables* (denoted by χ). The syntax of kinds, types and expressions is in fig. 1.

Multiplicities are used to indicate the number of times a value can be used. They are either unrestricted ($*$), which denotes an arbitrary number of usages, linear ($\mathtt{1}$), indicating precisely one usage, or a multiplicity variable (φ). The kinding system relies on two base kinds: $\mathtt{S}$ for session types and $\mathtt{T}$ for arbitrary types. *Kinds* are either the combination of a base kind and a multiplicity or a kind variable χ. Since a value of an unrestricted type may be used zero or more times, and one with a linear type must be used exactly once, it should be clear that an unrestricted value can used where



a linear one is expected. Similarly, the interpretation of base kinds should be such that a session type ($*\mathbf{S}$, $\mathbf{1S}$) can be used in place of an arbitrary type ($\mathbf{1T}$). The subkind relation for non variables (denoted $\kappa <: \kappa$) forms a lattice, as exhibited in the diagram.

Session types include Skip indicating no communication, End representing channels ready to be closed, output ($!T$) and input ($?T$) messages, internal ($\&\{\ell\colon T_\ell\}_{\ell\in L}$) and external choices ($\oplus\{\ell\colon T_\ell\}_{\ell\in L}$) and sequential composition ($T;U$). *Functional types* are composed of linear $()_{\mathbf{1}}$ and unrestricted unit types $()_*$, linear $T\ \mathbf{1}\!\!\rightarrow U$ and unrestricted $T\ *\!\!\rightarrow U$ functions, records $\{\ell\colon T_\ell\}_{\ell\in L}$, variants $\langle\ell\colon T_\ell\rangle_{\ell\in L}$ and universal types $\forall a^\kappa.T$. Recursive types $\mu\, a^\kappa.T$ are either session or functional depending on κ. Type variables a may refer to recursion variables in recursive types or to polymorphic variables in universal types. A function capturing in its body a free linear variable must itself be linear.

Expressions include variables x, term abstraction $\lambda_m x\colon T.e$ and application $e\,e$, type abstraction $\Lambda a^\kappa.v$ and application $e[T]$, record $\{\ell{=}e_\ell\}_{\ell\in L}$ and record elimination $\mathsf{let}\ \{\ell{=}x_\ell\}_{\ell\in L} = e\ \mathsf{in}\ e$, unit $()_m$ and unit elimination $\mathsf{let}\ ()_m = e\ \mathsf{in}\ e$, injection in a variant $\ell\,e$ and variant elimination $\mathsf{case}\ e\ \mathsf{of}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L}$. The expressions for channel operations include channel creation, $\mathsf{new}\ T$, and branching on a choice, $\mathsf{match}\ e\ \mathsf{with}\ \{\ell{\rightarrow}x_\ell\}_{\ell\in L}$. The remaining operations on channels—namely new, send, receive and select ℓ—are all understood as constants (pre-defined variables).

Given that our goal is to infer kind annotations, the reader may wonder why we allow them in the source language, namely in polymorphic types $\forall a^\kappa.T$, in recursive types $\mu\, a^\kappa.T$ and in type abstractions $\Lambda a^\kappa.v$. Programmers may, if they so wish, provide kind annotations in the source code. Such annotations are passed to the algorithm. For those omitted, a fresh kind variable χ is generated in its place.

3 Kind Inference

Our approach to kind inference follows the established two-step process, wherein the first gathers constraints and the second resolves the constraints. The constraint generation step produces constraints in two forms: $\kappa <: \kappa$ and $\varphi = \bigsqcup_{\ell\in L}\mathtt{mult}(\kappa_\ell)$. The first form represents subkinding constraints, while the second represents equalities between multiplicity variables and the least upper bound of a given set of multiplicities. To enhance readability, we use shorthand notation $\varphi = \mathtt{mult}(\kappa)$ for $\varphi = \bigsqcup\mathtt{mult}(\kappa)$ and use $\bigsqcup$ in infix format for binary sets.

Constraint Generation from Types Kind and multiplicity constraints are captured by judgement $\Delta_{\text{in}} \vdash T_{\text{in}}\colon \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}$. The judgement states that type T has kind κ under kinding context Δ (a map from type variables to kinds), producing constraint set $\mathscr{C}$. To clarify the distinction between input and output, we use the subscript **in** for parameters and **out** for results.

We explain a core subset of the constraint generation rules, those in fig. 2 (the complete set is in fig. 4). Rule CG-Var reads the kind for type variable a (recursive or polymorphic) from the kinding context, generating no additional restrictions. Rule CG-Rec governs recursive types which can either be session or functional. The kind of the recursion variable is copied to the kinding context when analysing type T. A constraint $\kappa' <: \kappa$ is generated to ensure that the kind κ' of the body of the recursive type is a subkind of the kind κ of the recursion variable. Rule CG-Arrow, deals with functions $T\ m\!\!\rightarrow U$. It applies the algorithm recursively to T and U, and assigns the kind $m\mathbf{T}$ to the function type, where m comes from the arrow annotation. Rule CG-Rcd builds kinds and constraints for all elements in the record. It generates a new fresh multiplicity variable φ. The result is kind $\varphi\mathbf{T}$ and the constraint set is composed of the union of $\mathscr{C}_\ell$ for all $\ell \in L$ and a new constraint associating variable φ to the least upper bound of the multiplicities of κ_ℓ. In order to ensure that φ gets the expected multiplicity, all elements

$\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}}$

CG-VAR
$$\Delta, a : \kappa \vdash a : \kappa \Rightarrow \varnothing$$

CG-REC
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C}}{\Delta \vdash \mu\, a^{\kappa} . T : \kappa' \Rightarrow \mathscr{C} \cup \{\kappa' <: \kappa\}}$$

CG-ARROW
$$\frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2}{\Delta \vdash T\ m \to U : m\mathbf{T} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2}$$

CG-RCD
$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \varphi \text{ fresh} \qquad (\forall \ell \in L)}{\Delta \vdash \{\ell : T_\ell\}_{\ell \in L} : \varphi\mathbf{T} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell), \kappa_\ell <: \varphi\mathbf{T}\}}$$

CG-TABS
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C} \qquad \varphi \text{ fresh}}{\Delta \vdash \forall a^{\kappa} . T : \varphi\mathbf{T} \Rightarrow \mathscr{C} \cup \{\varphi = \mathtt{mult}(\kappa')\}}$$

Figure 2: Constraint generation from types

must be subkinds of the kind of the record, that is $\varphi\mathbf{T}$. Thus, if at least one entry in the record is linear, then φ is also constrained to be linear. Rule CG-TAbs adds the kind of the polymorphic variable to the typing context when checking the body T. It then assigns kind $\varphi\mathbf{T}$ to the incoming type $\forall a^{\kappa} . T$, where the fresh multiplicity variable φ denotes the multiplicity of the kind of type T.

Type operator `mult` is fully resolved only after analysing expressions. At this point it can only be partially resolved. When applied to a kind of the form $m\upsilon$ operator `mult` rewrites into multiplicity m, that is, $\mathtt{mult}(m\upsilon) = m$.

As an example, let us consider the function that extracts the first element of a pair.

$$\mathtt{fst} : \forall a^{\chi_a} . \forall b^{\chi_b} . \{\mathsf{fst} : a, \mathsf{snd} : b\} \ast\!\!\to a$$

The application of the rules in fig. 2, yields the constraint set $\{\varphi_1 = \mathtt{mult}(\varphi_2\mathbf{T}), \varphi_2 = \mathtt{mult}(\ast\mathbf{T}), \varphi_3 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. Solving the constraint set one obtains $\{\varphi_1 = \ast, \varphi_2 = \ast, \varphi_3 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. We resolve the indeterminacy of kind variables χ_a and χ_b by assuming that they both are $\mathbf{1T}$, the maximum of the kind lattice. The solution would then be $\{\varphi_1 = \ast, \varphi_2 = \ast, \varphi_3 = \mathbf{1}, \chi_a = \mathbf{1T}, \chi_b = \mathbf{1T}\}$.

We argue that assigning $\mathbf{1T}$ (the maximum) to χ_a and χ_b is the preferred solution, since it is the less restrictive of all solutions. If we were to choose another kind, such as $\ast\mathbf{T}$, then it would be impossible to call function `fst` on linear values (of types with kind $\mathbf{1T}$). We would, undesirably, be ruling out some perfectly well-behaved programs.

But is $\mathbf{1T}$ the best kind for variables χ_a and χ_b? The answer depends on the definition of `fst`.

$$\mathtt{fst} = \Lambda a^{\chi_a} . \Lambda b^{\chi_b} . \lambda_{\ast}\, p : \{\mathsf{fst} : a, \mathsf{snd} : b\} . \mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$$

An examination of expression $\mathsf{let}\ \{\mathsf{fst} = x, \mathsf{snd} = y\} = p\ \mathsf{in}\ x$ reveals that the second element of the pair, y, is discarded. Hence, χ_b must be unrestricted. Would $\chi_b = \mathbf{1T}$ be chosen, then FREEST would complain about a linearity violation when type checking the function. In other words, constraint $\chi_b <: \ast\mathbf{T}$ must be added to the constraint set, but an inspection of the type of `fst` alone does not provide enough information to generate such a constraint. In the following, we present rules that allow generating constraints such as $\chi_b <: \ast\mathbf{T}$ by inspecting variable usage in expressions.

Constraint Generation from Expressions Constraints for expressions are derived from judgement $\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}$. The judgement states that expression e has type T under kinding context Δ and typing context Γ. It generates a constraint set $\mathscr{C}$ and a usage context Σ. Typing contexts map term variables x to types T; usage contexts map term variables x to the kind κ of their types. Usage

$$\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}}$$

$$\textsc{Inf-Var}\ \frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma, x : T \vdash x : T \Rightarrow \mathscr{C} \mid \{x : \kappa\}}$$

$$\textsc{Inf-Abs}\ \frac{\Delta \vdash T_1 : \kappa \Rightarrow \mathscr{C}_1 \quad \Delta \mid \Gamma, x : T_1 \vdash e : T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \quad \mathscr{C}_3 = \mathsf{if}\ \mathtt{isAbs}\ e\ \mathsf{then}\ \{\kappa <: m\mathbf{T}\}\ \mathsf{else}\ \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x : T_1 . e : T_1\ m \to T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus x}$$

$$\textsc{Inf-App}\ \frac{\Delta \mid \Gamma \vdash e_1 : T_1\ m \to T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1\ m \to T_2 : \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1\ e_2 : T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

$$\textsc{Inf-RcdElim}\ \frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 : \{\ell : T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, (x_\ell : T_\ell)_{\ell \in L} \vdash e_2 : T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T : \kappa \Rightarrow \mathscr{C}_3 \\ \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \mathscr{C}_\ell \cup \mathtt{Merge}(\Sigma_1, \Sigma_2) \cup \mathtt{Weaken}(\Sigma_2, x_\ell, \kappa_\ell) \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma \vdash \mathsf{let}\ \{\ell = x_\ell\}_{\ell \in L} = e_1\ \mathsf{in}\ e_2 : T \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell\}_{\ell \in L}}$$

Figure 3: Constraint generation from expressions

contexts enable reasoning about variable usage: if the variable is used exactly once, it may be linear, otherwise it must be unrestricted. Next, we define functions `Weaken` and `Merge`. The former checks whether variables are used in expressions. If a variable is not used, then the set with constraint $\kappa <: *\mathbf{T}$ is returned. The latter checks whether a variable is used more than once: if it appears in multiple usage contexts, it must also be unrestricted.

$$\mathtt{Weaken}(\Sigma, x, \kappa) = \begin{cases} \varnothing & \text{if } x \in \Sigma \\ \{\kappa <: *\mathbf{T}\} & \text{otherwise} \end{cases} \qquad \mathtt{Merge}(\Sigma_1, \Sigma_2) = \{\kappa <: *\mathbf{T} \mid x : \kappa \in \Sigma_1 \cap \Sigma_2\}$$

We are now in a position to explain the rules for expressions, in fig. 3 (the complete set is in fig. 5). Rule Inf-Var is used to assign a type to a variable in a given typing context. The rule requires the type context Γ to contain an entry $x : T$. The constraints pertaining to type T are gathered in $\mathscr{C}$. To reflect the usage of x, the rule returns a singleton map $x : \kappa$, where κ is the kind of T. Rule Inf-Abs deals with abstractions $\lambda_m x : T_1 . e$. It recursively calls the judgments on T_1 and on e to collect constraint sets $\mathscr{C}_1, \mathscr{C}_2$ and usage context Σ. The rule uses a new predicate, $\mathtt{isAbs}\ e$, which holds when e is an abstraction. Then, if e is a closure the kind of T_1 must be a subkind of $m\mathbf{T}$, where m is the multiplicity of the abstraction. This restriction ensures that unrestricted abstractions do not close over linear values. The result is type $T_1\ m \to T_2$ together with a constraint set composed of the union of $\mathscr{C}_1$, $\mathscr{C}_2$, $\mathscr{C}_3$ and the result of `Weaken`. The `Weaken` function checks whether a variable is unused at the end of its scope. In this case, the lambda abstraction introduces term variable x and therefore, at the end of the scope, we have to check its usage. Rule Inf-App states that if e_1 has type $T_1\ m \to T_2$ and e_2 has type T_1, then the expression $e_1\ e_2$ has type T_2. The constraints $\mathscr{C}$ and usage context Σ are computed by combining the results of the kind inference of e_1, e_2 and T. The final constraint set is the union of Σ_1, Σ_2, Σ_3, and the result of the `Merge` function which imposes that any variable found in both Σ_1 and Σ_2 must be unrestricted. The final usage context is $\Sigma_1 \cup \Sigma_2$. Rule Inf-RcdElim combines all previously discussed concepts: it evaluates expressions e_1 and e_2, collecting $\mathscr{C}_1, \mathscr{C}_2$ and Σ_1, Σ_2. The result is the type of e_2, a constraint set $\mathscr{C}$, which is the union of

$\mathscr{C}_1, \mathscr{C}_2, \mathscr{C}_3$, the result of `Merge` on Σ_1 and Σ_2, and the application of `Weaken` on Σ_2 for all $x_\ell : \kappa_\ell$ to check for unused variables. The resulting usage context is the combination of Σ_1 and Σ_2 with all entries for x_ℓ removed.

When analysing constraint generation from the type for function `fst`, we intuitively concluded that the second element in the pair must be unrestricted because it is discarded. The application of rules in fig. 3, yield the constraint set $\{\chi_b <: *\mathbf{T}, \chi_a <: \varphi_1\mathbf{T}, \chi_b <: \varphi_1\mathbf{T}, \chi_a <: \varphi_0\mathbf{T}, \chi_b <: \varphi_0\mathbf{T}, \varphi_0 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b), \varphi_1 = \mathtt{mult}(\chi_a) \sqcup \mathtt{mult}(\chi_b)\}$. A solution for this set is $\{\varphi_0 = \mathbf{1}, \varphi_1 = \mathbf{1}, \chi_a = \mathbf{1T}, \chi_b = *\mathbf{T}\}$. The kind variable χ_b is set to $*\mathbf{T}$ as we predicted. The constraint set is computed by combining the constraint sets generated resulting from applying the judgement to all sub-expressions and the result of functions `Merge` and `Weaken`. First, we examine the `Merge` function: it takes contexts $\{p : \kappa_p\}$ and $\{x : \chi_a\}$ as input and calculates the intersection of the two contexts, adding a constraint $\kappa <: *\mathbf{T}$ for each element in the intersection. This process ensures that any variable that is used in both contexts is unrestricted. The `Weaken` function is used to verify if any newly introduced variable is eventually discarded. In our example, `Weaken` is applied to $x : \chi_a$ and $y : \chi_b$ against usage context $\{p : \kappa_p, x : \chi_a\}$. For $y : \chi_b$ function `Weaken` proceeds as follows: since y is not present in the context, a new constraint $\{\chi_b <: *\mathbf{T}\}$ is added. On the other hand, since x is already in the context, no constraint is created.

Constraint Solving We now describe an algorithm to solve constraint sets.

1. Initialise all kind variables χ to the maximum of the kind lattice, $\mathbf{1T}$. Likewise initialize all multiplicity variables φ to the maximum of multiplicities, $\mathbf{1}$. Store them in σ.
2. Iterate over each constraint in the set:
   (a) If the constraint is of the form $\chi <: \kappa$, then update the entry for χ in σ with the greatest lower bound of κ and $\sigma(\chi)$. For example, if $\sigma = [\chi \mapsto \mathbf{1T}]$ and we are analysing constraint $\chi <: *\mathbf{T}$, then the value for χ in σ must be updated to $\mathbf{1T} \sqcap *\mathbf{T} = *\mathbf{T}$. After this step, we would have $\sigma = [\chi \mapsto *\mathbf{T}]$.
   (b) If the constraint is of the form $\kappa <: \chi$, then check whether κ and the kind for χ in σ is in the subkind relation; if not then fail. For example, if $\sigma = [\chi \mapsto \mathbf{1T}]$ and we are analysing constraint $*\mathbf{T} <: \chi$, then we find that it is in the subkind relation since $*\mathbf{T} <: \mathbf{1T}$. A failure would happen with $\sigma = [\chi \mapsto \mathbf{1T}]$.
   (c) If the constraint is of the form $\kappa_1 <: \kappa_2$ and neither of the elements is a kind variable, then check whether $\kappa_1 <: \kappa_2$ is in the subkind relation; if not then fail. If not fail, then remove constraint $\kappa_1 <: \kappa_2$ from the constraint set.
   (d) If the constraint is a multiplicity constraint $\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell)$, then compute the least upper bound of the multiplicities. If any κ_ℓ is a kind variable (χ) or a base kind with a multiplicity variable ($\varphi\mathbf{T}$), we get its kind from σ (recall that all variables are in σ as per step 1). If the thus obtained kind is more restrictive than that for φ in σ (e.g. $*$ against $\sigma(\varphi) = \mathbf{1}$), then store it in σ. If $\varphi = *$, then remove the constraint from the set.
3. Repeat the process in step 2 until there are no further updates to be made.
4. If all constraints have been satisfied, then return the solution σ. Otherwise, the constraint set is unsatisfiable.

In the case of function `fst`, the constraints gathered by the rules in fig. 3 are as follows.

$$\chi_1 <: \varphi_0\mathbf{T}, \chi_0 <: \varphi_0\mathbf{T}, \chi_1 <: \varphi_1\mathbf{T}, \chi_0 <: \varphi_1\mathbf{T}, \chi_1 <: *\mathbf{T},$$
$$\varphi_0 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1), \varphi_1 = \mathtt{mult}(\chi_0) \sqcup \mathtt{mult}(\chi_1)$$

Category of annotation	Number of annotations in the source code	Number of more general annotations generated
Datatypes	129	0
Type abbreviations	206	7
Universal types	282	94
Explicit recursive types	23	10
Type abstractions	30	25
Total	670	136

Table 1: Distribution of annotations

We start with $\sigma = [\chi_0 \mapsto \mathtt{1T}, \chi_1 \mapsto \mathtt{1T}, \varphi_0 \mapsto \mathtt{1}, \varphi_1 \mapsto \mathtt{1}]$. Next we pick constraint $\chi_1 <: \varphi_0\mathtt{T}$ and use item 2(a). We have, $\chi_1 <: \mathtt{1T}$ since $\sigma(\varphi_0) = \mathtt{1}$. Given that $\sigma(\chi_0)$ is equal to $\mathtt{1T}$, and subkinding is reflexive, $\sigma(\varphi_0)$ remains as $\mathtt{1T}$. The process for the second constraint, $\chi_1 <: \varphi_0\mathtt{T}$, is similar. We analyse the constraint $\chi_1 <: \mathtt{1T}$ since $\sigma(\varphi_0) = \mathtt{1}$. Also in this case item 2(a) does not change σ. The next two constraints, $\chi_1 <: \varphi_1\mathtt{T}$ and $\chi_0 <: \varphi_1\mathtt{T}$, are also handled by item 2(a). Once again, σ is subject to no update. Now we pick constraint $\chi_1 <: *\mathtt{T}$. Under item 2(a) the algorithm computes the greatest lower bound of $*\mathtt{T}$ and $\mathtt{1T}$, which is $*\mathtt{T}$, so σ is updated accordingly. For the last two constraints we use item 2(d). We read the values of χ_0 and χ_1 from σ and compute the least upper bound of $\mathtt{mult}(\mathtt{1T})$ and $\mathtt{mult}(*\mathtt{T})$ which yields $\mathtt{1}$. Both entries for χ_0 and χ_1 are already $\mathtt{1}$ and therefore no update to σ is done. Since we analysed all constraints and σ was updated in this iteration of the algorithm, the fixed-point is not reached yet and so we go through each constraint once again. This time no update is made and therefore we terminate with $\sigma = [\chi_0 \mapsto \mathtt{1T}, \chi_1 \mapsto *\mathtt{T}, \varphi_0 \mapsto \mathtt{1}, \varphi_0 \mapsto \mathtt{1}]$.

The algorithm iteratively updates the values of the kind and multiplicity variables until no further updates can be made, that is, until a fixed point is reached. Since the kind lattice is finite, any sequence of updates must eventually converge to a fixed point. For the same reason, each constraint can only be updated a finite number of times. Therefore, the algorithm terminates after a finite number of iterations.

The running time of the constraint generation algorithm is linear on the size of the input expression; that of the constraint satisfaction algorithm is quadratic. In the worst case scenario the number of constraints is equal to the size of the expression. Each constraint can only update σ twice (when a more restrictive solution is found). The worst case happens when a different constraint performs an update in each iteration, forcing the algorithm to analyse all the constraints in each iteration. A sensible optimization removes the constraints from the constraint set also in items 2(a) and 2(b), after concluding that they cannot update σ to a more restrictive solution. Since the update can only be performed a constant number of times, the algorithm becomes linear on the size of the input expression.

Evaluation We implemented the algorithm and incorporated it in the FREEST interpreter. Then we conducted an evaluation to check the behaviour of the algorithm when used on FREEST source code. The evaluation consisted of replacing all the 670 kind annotations by fresh kind variables in the 232 valid programs in the FREEST test suite and standard library (total of 9131 lines of code), running the algorithm and checking whether the algorithm infers the annotations back.

Kind annotations are spread over datatypes, type abbreviations, universal types, recursive types, and type abstractions. The distribution of annotations is as in table 1. The small number of annotations in recursive types and type abstractions comes from the fact that they are usually introduced implicitly, either via type abbreviations (as in the code in line 2) or through compiler elaboration introducing type

abstractions $\Lambda a^{\kappa}.\, v$ for functions accompanied by their signatures.

We concluded that the algorithm correctly inferred all annotations and found that 136 of the 670 annotations (that is, 20%) were too specific and could be relaxed to a more general kind. The largest number of more general annotations found by the algorithm come from universal types. We attribute this to the conservative nature of programmers: if we are developing Church encodings (heavy on polymorphism), why would one require linear type variables? The algorithm did not improve the kind for datatypes: datatypes are usually used in an unrestricted manner in programs. Moreover, in the test suite, they usually appear as the first argument (to be pattern-matched) of functions with unrestricted closures and therefore they cannot be linear.

For an example where the algorithm suggests a more general kind, consider function composition.

```
dot : ∀ a:*T b:*T c:*T . (b → c) → (a → b) → a → c
dot f g x = f (g x)
```

If we only provide unrestricted arguments to dot, then there is no reason why the polymorphic variables a, b and c could not have kind *T. However, we would be ruling out programs that apply dot to linear arguments. Consider the following program.

```
dot : (b → c) → (a → b) → a → c
dot f g x = f (g x)

g : ?Int;End → Int
g c = let (x, c) = receive c in close c ; x

main : Int
main =
  let (w,r) = new () in
  fork (\_ 1→ let w = send 5 w in close w);
  dot id g r
```

This program would be flagged as untypable because we instantiate the polymorphic variable a with the linear session type ?Int;End. Since there is no reason why a, b and c should be unrestricted, the algorithm assigns kind 1T to the three polymorphic variables.

4 Future Work

There are several avenues for future work. The most immediate is to prove the correctness of the algorithm with respect to the typing system. Then, equipped with kind inference, we may think of introducing a third base kind, that for session types that must be eventually closed (that reach type End). In this case we would require the kind of the argument to function new to be of the newly introduced kind. We further plan to study the possibility of quantifying over kinds or multiplicities for extra flexibility in programming.

Acknowledgements We thank the anonymous reviewers for their detailed comments that greatly contributed to improve the paper. This work was supported by FCT through project SafeSessions, ref. PTDC/CCI-COM/6453/2020, and the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020.

References

[1] Bernardo Almeida, Andreia Mordido, Peter Thiemann & Vasco T. Vasconcelos (2022): *Polymorphic lambda calculus with context-free session types*. *Inf. Comput.* 289(Part), p. 104948, doi:10.1016/j.ic.2022.104948.

[2] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST, a concurrent programming language with context-free session types*. https://freest-lang.github.io. Last accessed 2023.

[3] Bernardo Almeida, Andreia Mordido & Vasco T. Vasconcelos (2019): *FreeST: Context-free Session Types in a Functional Language*. In: *PLACES*, *EPTCS* 291, pp. 12–23, doi:10.4204/EPTCS.291.2.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones & Arnaud Spiwack (2018): *Linear Haskell: practical linearity in a higher-order polymorphic language*. *Proc. ACM Program. Lang.* 2(POPL), pp. 5:1–5:29, doi:10.1145/3158093.

[5] Jean-Yves Girard (1987): *Linear Logic*. *Theor. Comput. Sci.* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.

[6] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[7] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP*, *LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[8] Karl Mazurak, Jianzhou Zhao & Steve Zdancewic (2010): *Lightweight linear types in system fdegree*. In Andrew Kennedy & Nick Benton, editors: *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, ACM, pp. 77–88, doi:10.1145/1708016.1708027.

[9] J. Garrett Morris (2016): *The best of both worlds: linear functional programming without compromise*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 448–461, doi:10.1145/2951913.2951925.

[10] Martin Odersky, Christoph Zenger & Matthias Zenger (2001): *Colored local type inference*. In Chris Hankin & Dave Schmidt, editors: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, ACM, pp. 41–53, doi:10.1145/360204.360207.

[11] Luca Padovani (2019): *Context-Free Session Type Inference*. *ACM Trans. Program. Lang. Syst.* 41(2), pp. 9:1–9:37, doi:10.1145/3229062.

[12] Benjamin C. Pierce & David N. Turner (2000): *Local type inference*. *ACM Trans. Program. Lang. Syst.* 22(1), pp. 1–44, doi:10.1145/345099.345100.

[13] Gabriel Radanne, Hannes Saffrich & Peter Thiemann (2020): *Kindly bent to free us*. *Proc. ACM Program. Lang.* 4(ICFP), pp. 103:1–103:29, doi:10.1145/3408985.

[14] John C. Reynolds (1974): *Towards a theory of type structure*. In Bernard J. Robinet, editor: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, *Lecture Notes in Computer Science* 19, Springer, pp. 408–423, doi:10.1007/3-540-06859-7_148.

[15] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, *LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[16] Peter Thiemann & Vasco T. Vasconcelos (2016): *Context-free session types*. In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, ACM, pp. 462–475, doi:10.1145/2951913.2951926.

[17] Jesse A. Tov & Riccardo Pucella (2011): *Practical affine types*. In Thomas Ball & Mooly Sagiv, editors: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, ACM, pp. 447–458, doi:10.1145/1926385.1926436.

[18] J. B. Wells (1994): *Typability and Type-Checking in the Second-Order lambda-Calculus are Equivalent and Undecidable*. In: *LICS*, IEEE Computer Society, pp. 176–185, doi:10.1109/LICS.1994.316068.

[19] Andrew K. Wright (1995): *Simple Imperative Polymorphism*. *LISP Symb. Comput.* 8(4), pp. 343–355.

$\boxed{\Delta_{\text{in}} \vdash T_{\text{in}} : \kappa_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}}}$

CG-Unit
$$\Delta \vdash ()_m : m\mathbf{T} \Rightarrow \varnothing$$

CG-Var
$$\Delta, a : \kappa \vdash a : \kappa \Rightarrow \varnothing$$

CG-Skip
$$\Delta \vdash \mathsf{Skip} : *\mathbf{S} \Rightarrow \varnothing$$

CG-End
$$\Delta \vdash \mathsf{End} : \mathbf{1S} \Rightarrow \varnothing$$

CG-Msg
$$\frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \vdash \sharp T : \mathbf{1S} \Rightarrow \mathscr{C}}$$

CG-Ch
$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad (\forall \ell \in L)}{\Delta \vdash \star\langle\!|\ell : T_\ell|\!\rangle_{\ell \in L} : \mathbf{1S} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\kappa_\ell <: \mathbf{1S}\}}$$

CG-Seq
$$\frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2 \qquad \varphi \text{ fresh}}{\Delta \vdash T;U : \varphi\mathbf{S} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \{\kappa_1 <: \mathbf{1S}, \kappa_2 <: \mathbf{1S}, \varphi = \mathtt{mult}(\kappa_1) \sqcup \mathtt{mult}(\kappa_2)\}}$$

CG-Rec
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C}}{\Delta \vdash \mu\, a^{\kappa}.T : \kappa' \Rightarrow \mathscr{C} \cup \{\kappa' <: \kappa\}}$$

CG-Arrow
$$\frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \vdash U : \kappa_2 \Rightarrow \mathscr{C}_2}{\Delta \vdash T\ m{\to}\ U : m\mathbf{T} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2}$$

CG-Rcd
$$\frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \varphi \text{ fresh} \qquad (\forall \ell \in L)}{\Delta \vdash \{\ell : T_\ell\}_{\ell \in L} : \varphi\mathbf{T} \Rightarrow \bigcup_{\ell \in L} \mathscr{C}_\ell \cup \{\varphi = \bigsqcup_{\ell \in L} \mathtt{mult}(\kappa_\ell), \kappa_\ell <: \varphi\mathbf{T}\}}$$

CG-TAbs
$$\frac{\Delta, a : \kappa \vdash T : \kappa' \Rightarrow \mathscr{C} \qquad \varphi \text{ fresh}}{\Delta \vdash \forall\, a^{\kappa}.T : \varphi\mathbf{T} \Rightarrow \mathscr{C} \cup \{\varphi = \mathtt{mult}(\kappa')\}}$$

Figure 4: Constraint generation from types (complete set of rules)

$$\boxed{\Delta_{\text{in}} \mid \Gamma_{\text{in}} \vdash e_{\text{in}} : T_{\text{out}} \Rightarrow \mathscr{C}_{\text{out}} \mid \Sigma_{\text{out}}}$$

$$\textsc{Inf-Const}\ \frac{\Delta \vdash \texttt{typeof}(c) : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma \vdash c : \texttt{typeof}(c) \Rightarrow \mathscr{C} \mid \varnothing} \qquad \textsc{Inf-Var}\ \frac{\Delta \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma, x : T \vdash x : T \Rightarrow \mathscr{C} \mid \{x : \kappa\}}$$

$$\textsc{Inf-Abs}\ \frac{\Delta \vdash T_1 : \kappa \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma, x : T_1 \vdash e : T_2 \Rightarrow \mathscr{C}_2 \mid \Sigma \qquad \mathscr{C}_3 = \mathsf{if}\ \texttt{isAbs}\ e\ \mathsf{then}\ \{\kappa <: m\mathbf{T}\}\ \mathsf{else}\ \varnothing}{\Delta \mid \Gamma \vdash \lambda_m x : T_1 . e : T_1\ m \to T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Weaken}(\Sigma, x, \kappa) \mid \Sigma \setminus \{x : \kappa\}}$$

$$\textsc{Inf-App}\ \frac{\Delta \mid \Gamma \vdash e_1 : T_1\ m \to T_2 \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_2 : T_1 \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T_1\ m \to T_2 : \kappa \Rightarrow \mathscr{C}_3}{\Delta \mid \Gamma \vdash e_1\ e_2 : T_2 \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Merge}(\Sigma_1, \Sigma_2) \mid \Sigma_1 \cup \Sigma_2}$$

$$\textsc{Inf-TAbs}\ \frac{\Delta, a : \kappa \mid \Gamma \vdash v : T \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T : \kappa' \Rightarrow \mathscr{C}_2}{\Delta \mid \Gamma \vdash \Lambda a^{\kappa} . v : \forall a^{\kappa} . T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

$$\textsc{Inf-TApp}\ \frac{\Delta \vdash T : \kappa_1 \Rightarrow \mathscr{C}_1 \qquad \Delta \mid \Gamma \vdash e : \forall a^{\kappa_2} . U \Rightarrow \mathscr{C}_2 \mid \Sigma}{\Delta \mid \Gamma \vdash e\,[T] : U[T/a] \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_2 \mid \Sigma}$$

$$\textsc{Inf-RcdElim}\ \frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 : \{\ell : T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma, (x_\ell : T_\ell)_{\ell \in L} \vdash e_2 : T \Rightarrow \mathscr{C}_2 \mid \Sigma_2 \qquad \Delta \vdash T : \kappa \Rightarrow \mathscr{C}_3 \\ \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \mathscr{C} = \mathscr{C}_1 \cup \mathscr{C}_2 \cup \mathscr{C}_3 \cup \texttt{Merge}(\Sigma_1, \Sigma_2) \cup \texttt{Weaken}(\Sigma_2, x_\ell, \kappa_\ell) \qquad (\forall \ell \in L)\end{array}}{\Delta \mid \Gamma \vdash \mathsf{let}\ \{\ell = x_\ell\}_{\ell \in L} = e_1\ \mathsf{in}\ e_2 : T \Rightarrow \mathscr{C} \mid (\Sigma_1 \cup \Sigma_2) \setminus \{x_\ell : \kappa_\ell\}_{\ell \in L}}$$

$$\textsc{Inf-Rcd}\ \frac{\Delta \mid \Gamma \vdash e_\ell : T_\ell \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \{\ell = v_\ell\}_{\ell \in L} : \{\ell : T_\ell\}_{\ell \in L} \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \cup \texttt{Merge}(\Sigma_\ell) \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

$$\textsc{Inf-Variant}\ \frac{\Delta \mid \Gamma \vdash e : T_k \Rightarrow \mathscr{C}_1 \mid \Sigma \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash k\ e : \langle \ell : T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \mid \Sigma}$$

$$\textsc{Inf-Case}\ \frac{\Delta \mid \Gamma \vdash e : \langle \ell : T_\ell \rangle_{\ell \in L} \Rightarrow \mathscr{C}_1 \mid \Sigma_1 \qquad \Delta \mid \Gamma \vdash e_\ell : T_\ell\ m \to T \Rightarrow \mathscr{C}_\ell \mid \Sigma_\ell \qquad \Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}'_\ell \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\ell \to x_\ell\}_{\ell \in L} : T \Rightarrow \mathscr{C}_1 \cup \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \Sigma_1 \cup \Sigma_\ell}$$

$$\textsc{Inf-Sel}\ \frac{\Delta \vdash T_\ell : \kappa_\ell \Rightarrow \mathscr{C}_\ell \qquad \Delta \mid \Gamma \vdash e_\ell : T_\ell\ m \to T \Rightarrow \mathscr{C}'_\ell \mid \Sigma_\ell \qquad k \in L \qquad (\forall \ell \in L)}{\Delta \mid \Gamma \vdash \mathsf{select}\ k : \oplus \langle\!| \ell : T_\ell |\!\rangle_{\ell \in L}\ m \to T_k \Rightarrow \mathscr{C}_\ell \cup \mathscr{C}'_\ell \mid \bigcup_{\ell \in L} \Sigma_\ell}$$

$$\textsc{Inf-New}\ \frac{\varnothing \vdash T : \kappa \Rightarrow \mathscr{C}}{\Delta \mid \Gamma \vdash \mathsf{new}\ T : \{\mathsf{fst} : T, \mathsf{snd} : \overline{T}\} \Rightarrow \mathscr{C} \mid \emptyset}$$

Figure 5: Constraint generation from expressions (complete set of rules)