1 Inference

Definition 1 (Type variables). *$\mathbb{V}$ is the space of type variables given as follows, each (sub)type considered by the inference algorithm will be assigned a type variable representing it:*
$$\begin{aligned}\mathbb{V} &\Rightarrow v\\ &\mid \mathbb{V}'\end{aligned}$$

The variables follow the obvious lexical ordering (based on their length).

Definition 2 (Primitive patterns). *$\mathbb{P}$ is the space of primitive patterns given as follows, they represent the topmost level of a type's structure:*

$\mathbb{P} \Rightarrow$	Primitive pattern
$[\mathbb{V}] \to \mathbb{V}$	Function
$\mid [\mathbb{V}]$	Tuple
$\mid \mathbb{V}\ \mathbb{V}$	Application
$\ldots$	

Observation 1 (Primitive patterns). *Every type $\tau \in \mathbb{T}$ can be equally represented by a type variable and a set of primitive patterns $P \in \mathbb{P}$. This also provides us the way to unambiguously identify the type τ' of a subexpression x' ($x' : \tau'$) of the expression x ($x : \tau$).*

Definition 3 (Constnesses). *$\mathbb{C}$ is the space of constnesses given as follows:*

$\mathbb{C} \Rightarrow$	Constness
`Regular`	*for regular runtime variables*
$\mid$ `Linkexpr`	*for variables that are to be known during linkage*
$\mid$ `Constexpr`	*for variables that are to be known during compilation*

Constness of a type specifies the time when an object assigned the type is to be computed.

We define ordering on constnesses: `Constexpr` $<$ `Linkexpr` $<$ `Regular`.

Definition 4 (Data kinds). *$\mathbb{K}$ is the space of data kinds, it is a subset of the power set of a set of registers.*

The kinds are ordered by the relation of being a subset. We put `GenericKind` *as the superset of all possible kinds and* `EmptyKind` *as the subset of all possible kinds.*

Definition 5 (Type properties). *$\mathbb{D}$ is the space of type properties given as follows:*

$\mathbb{D} \Rightarrow (\mathit{typing} : \mathbb{T}, \mathit{constness} : \mathbb{V}, \mathit{kind} : \mathbb{V})$

This states that a type has a certain typing, constness of a certain type and a kind of a certain type.

The functions that project objects from $\mathcal{D}$ to the respective properties are called t, c, k.

Definition 6 (Facts)**.** *$\mathbb{F}$ is the space of flat facts given as follows:*

$\mathbb{F} \Rightarrow$	Flat facts
$\mathbb{T} \sim \mathbb{T}$	Unification: *specifies that the types have to be identical*
$\mathbb{T} \sim_T \mathbb{T}$	Typing unification: *specifies that the types have to follow the same typing*
$\mathbb{T} \leq_K \mathbb{T}$	SubKind: *specifies that the right operand has more general data kind than the first operand*
$\mathbb{T} \leq_C \mathbb{T}$	SubConst: *specifies that the right operand has more general constness than the first operand*
$\mathbb{K} \leq_K \mathbb{T} \leq_K \mathbb{K}$	Kind bounds: *specifies that the type has a data kind from the given range*
$\mathbb{C} \leq_C \mathbb{T} \leq_C \mathbb{C}$	Const bounds: *specifies that the type has a constness from the given range*
$C\ \mathbb{T}$	Class constraint: *specifies that the given constraint type (applied to a certain number of types - unless nullary) is valid*
$\mathbb{T} \leq_M \mathbb{T}$	Instantiation: *specifies that the left operand is monotype instance of the polytype right operand; note: this is satisfiable only if the right operand is indeed a polytype*

Sometimes we will use derived facts like $\mathbb{T} \leq_K \mathbb{K}$, $\mathbb{T} =_K \mathbb{T}$, etc. These are either special cases of the aforementioned facts, or compositions of multiple facts. The most complicated derived fact we will use is the "SubType" fact, $\mathbb{T} \leq \mathbb{T}$, which is a combination of $\mathbb{T} \sim_T \mathbb{T}$, $\mathbb{T} \leq_K \mathbb{T}$ and $\mathbb{T} \leq_C \mathbb{T}$.

It should be noted that, within the defined language, we cannot state the facts $\mathbb{T} \not\sim_T \mathbb{T}$, $\mathbb{T} \not\leq_K \mathbb{T}$, $\mathbb{T} \not\leq_C \mathbb{T}$ and $\mathbb{T} \not\leq_M \mathbb{T}$ and thus these (and similar ones) are not derived facts.

$\mathbb{W}$ is the space of facts given as follows:

$\mathbb{W} \Rightarrow$	Facts
$\mathbb{F}$	Fact
$\forall \alpha_1, \ldots \alpha_n.[\mathbb{F}] \Rightarrow [\mathbb{W}]$	Nested facts: *specifies that satisfying the given facts requires the nested facts to be satisfied as well*

Definition 7 (Bounds)**.** *We call the rightmost operand of a bounds fact the upper-bound and the leftmost operand the lower-bound.*

Definition 8 (Trivial bounds)**.** *If a fact F states $k_1 \leq_K \tau \leq_K k_2$ and $k_2 \leq k_1$, or $c_1 \leq_C \tau \leq_C c_2$ and $c_2 \leq c_1$, we call such a bounds fact trivial.*

Observation 2 (Trivial bounds)**.**

1. *If the bounds of a trivial bounds fact are equal, the type constesses of the types constrained by such a fact have to be the same and we can unify them.*
2. *If, for a bounds fact, the upperbound is lower than the lowerbound, the fact cannot be satisfied.*

Definition 9 (Algorithm state)**.** *The algorithm state consists of the following finite variables*

- *Facts:* $\mathcal{F} :: [\mathbb{W}]$
- *Active type variables:* $\mathcal{V} ::\subset \mathbb{V}$
- *Forgotten type variables:* $\mathcal{G} ::\subset \mathbb{V}$
- *primitive patterns:* $\mathcal{P} ::\subset \mathbb{P}$
- *Type properties:* $\mathcal{D} ::\subset \mathbb{D}$
- *SubKinds:* $\mathcal{K} ::$ *Directional graph* $\mathbb{V}$
- *KindBounds:* $b_K :: \mathbb{V} \to$ *interval* $[\mathbb{K}, \mathbb{K}]$
- *SubConsts:* $\mathcal{C} ::$ *Directional graph* $\mathbb{V}$
- *ConstBounds:* $b_C :: \mathbb{V} \to$ *interval* $[\mathbb{C}, \mathbb{C}]$
- *Type explanation:* $p :: \mathcal{V}_p \to \mathcal{P}; \mathcal{V}_p \subseteq \mathcal{V}$
- *Type definition:* $d :: \mathcal{V} \to \mathcal{D}$
- *Result function:* $u :: \mathcal{G} \to \mathcal{V}$

By $<_\mathcal{K}$ and $<_\mathcal{C}$, we understand that there exist a directional path between the given distinct operands in the respective graphs. We naturally extend this definitions to the other comparison operators (for example: $x =_\mathcal{K} y$ if there are paths back and forth between x and y in the graph $\mathcal{K}$ or $x = y$).

Definition 10 (Algorithm intrastate invariants)**.** *The algorithm is designed to have the following intrastate invariants:*

1. *Active and forgotten variables:* $Var(\mathcal{F}) \cup Var(\mathcal{P}) \cup Var(\mathcal{D}) \cup Var(\mathcal{K}) \cup Var(\mathcal{C}) = Var(\mathcal{V})$ *and* $Var(\mathcal{V}) \cap Var(\mathcal{G}) = \emptyset$.

2. *$\mathcal{V}$ and $\mathcal{D}$ are bijected by the type definition function d with its inverse d^{-1}. The subset $\mathcal{V}_p \subseteq \mathcal{V}$ and the set $\mathcal{P}$ are bijected by the type explanation function p and its inverse p^{-1}. Note that there generally are many types with no known explanation.*

   We expand the function applicability of b_K and b_C to type definitions is such a way that for an arbitrary type variable $v \in \mathcal{V}$, we have $b_K(d(v)) = b_K(c(d(v)))$, and similarly for b_C. In other words: applying b_K to a type definition is equivalent to applying it to the type's kind (and similarly for b_C).

   We expand the applicability of p and d in such a way that for an arbitrary type variable $v \in \mathcal{V}_p$ it holds that $d(v) = d(p(v))$ and $p(v) = p(d(v))$.

   And finally, we expand the applicability of t, k, c in such a way that for an arbitrary $v \in \mathcal{V}$ it holds that $t(v) = t(d(v))$, $k(v) = k(d(v))$, and $c(v) = c(d(v))$.

3.

4. *If we have two type variables $v, v' \in \mathcal{V}$ representing two types, then:*

   (a) *If $k(v) =_{\mathcal{K}} k(v')$, then: $k(v) = k(v')$. In other words, graph $\mathcal{K}$ has no strongly connected components (this will be true for $\mathcal{C}$ as well).*

   (b) *If $c(v) =_{\mathcal{C}} c(v')$, then: $c(v) = c(v')$.*

   (c) *If $b_K(k(v)) = b_K(k(v')) = [k_1, k_1]$ for some kind $k_1 \in \mathbb{K}$, then $k(v) = k(v')$.*

   (d) *If $b_K(k(v)) = b_K(k(v')) = [k_1, k_2]$ for some kinds $k_1, k_2 \in \mathbb{K}$, $k_2 < k_1$, then $k(v) = k(v')$ and it is an invalid kind.*

   (e) *If $b_C(c(v)) = b_C(c(v')) = [c_1, c_1]$ for some constness $c_1 \in \mathbb{C}$, then $c(v) = c(v')$.*

   (f) *If $b_C(c(v)) = b_C(c(v')) = [c_1, c_2]$ for some constnesses $c_1, c_2 \in \mathbb{K}$, $c_2 < c_1$, then $c(v) = c(v')$ and it is an invalid constness.*

   (g) *If $k(v) <_{\mathcal{K}} k(v')$, then for bounds k_1, k_2, k_3, k_4 such that $b_K(k(v)) = [k_1, k_3]$, and $b_K(k(v')) = [k_2, k_4]$ it holds that $k_1 \leq k_2$ and $k_3 \leq k_4$. Note that for new (and any non-stored) variables we will assume the least restrictive bounds, so adhering to this invariant means we update the bounds monotonically.*

   (h) *If $c(v) <_{\mathcal{C}} c(v')$, then for bounds c_1, c_2, c_3, c_4 such that $b_C(c(v)) = [c_1, c_3]$, and $b_C(c(v')) = [c_2, c_4]$ it holds that $c_1 \leq c_2$ and $c_3 \leq c_4$.*

5. *If a type represented by a type variable v is explained ($v \in \mathcal{V}_p$), then $t(d(v)) = p(v)\,[\tau := t(d(\tau))|\tau \in \mathrm{free}(p(d))]$. In other words, the typing of the type represented by v follows the typing generated from the typings of the types via which it is explained.*

Definition 11 (Algorithm interstate invariants)**.** *The algorithm is designed to have the following interstate invariants: (we use the indices to distinguish a successor state variable from its predecessor, we will always assume $n < m \in \mathbb{N}$, we then use these indices in all derived functions too).*

1. *Forgotten variables stay forgotten: $v \in \mathcal{G}_n \Rightarrow v \in \mathcal{G}_m$.*

2. *Forgotten variables used to be active: $v \in \mathcal{G}_m \Rightarrow \exists n.v \in \mathcal{V}_n$.*

3. *All free variables from facts are considered by the algorithm: $v \in \mathrm{free}(\mathcal{F}_0) \Rightarrow v \in \mathcal{V}_0$.*

4. *Assumed subtype constraints are monotonically getting restricted. For kinds: $v \in \mathcal{V}_n \Rightarrow b_{Km}(k_m(u_m(v))) \subseteq b_{Kn}(k_n(v))$ and for constnesses: $v \in \mathcal{V}_n \Rightarrow b_{Cm}(c_m(u_m(v))) \subseteq b_{Cn}(c_n(v))$.*

5. *Typings do not get forgotten: $v \in \mathcal{V}_n \Rightarrow \exists s.s(t_n(v)) = t_m(u_m(v))$.*

6. *Explanations do not get forgotten: $v \in \mathcal{V}_{p_n} \Rightarrow \exists s.s(p_n(v)) = p_m(u_m(v))$.*

7. *Results are results of results: $u_m = u_m(u_n)$.*

Definition 12 (Unifications)**.** *In the scope of this algorithm we distinguish 4 types of unification. The algorithms for each will be the same, but their meaning is different.*

1. *Type unification: we unify two type variables that represent types, we then apply the resulting unification to all type variables considered by the algorithm.*

2. *Typing unification: we unify two typings of certain types, we then apply the resulting unification to all other typings considered by the algorithm.*

3. *Kind unification: we unify two type variables that represent kinds of certain types, we then apply the resulting unification to all other variables representing kinds considered by the algorithm.*

4. *Constness unification: we unify two type variables that represent constnesses of certain types, we then apply the resulting unification to all other variables representing constnesses considered by the algorithm.*

Definition 13 (Fresh variable). *A fresh variable is chosen according to:* $\inf(\mathbb{V} \setminus \mathcal{V} \setminus \mathcal{G})$.

Definition 14 (Algorithm middlesteps). *After each step of the algorithm that transforms one of the state variables (mainly $\mathcal{G}$,$\mathcal{P}$ or $\mathcal{D}$) into a new state, we perform the minimal necessary unifications to "repair" the bijections d and p (intrastate invariant 2, all unifications), to collapse any strongly connected components of the graphs $\mathcal{K}$ and $\mathcal{C}$ and to remove any duplicate images of trivial bounds from b_K and b_C (intrastate invariant 4, unifications 3 and 4). This will be often performed recursively. We then apply the resulting combined unification (limited to just unifications 1) to the function u so it satisfies the intrastate invariant 1.*

Observation 3 (Algorithm middlesteps). *The algorithm may (correctly) fail during the middlesteps. This can be shown by the observation* 2.

Definition 15 (Introducing new (fresh) variable). *We introduce a new (fresh) variable by extending the sets $\mathcal{V}$ and $\mathcal{D}$ and the type definition function d by the argument v and its image $i = (typing : v, constness : v, kind : v)$, formally: $\mathcal{V}' = \mathcal{V} \cup \{v\}$, $\mathcal{D}' = \mathcal{D} \cup \{i\}$, $d' = d[v := i]$. Where v is the new (fresh) variable.*

Definition 16 (Algorithm presteps). *All the (flat) facts generally use complicated types, before each step of the algorithm we replace the types in the observed fact with type variables that represent the types.*

The transformation is performed according to the observation 1, changing the state variables $\mathcal{P}$, $\mathcal{D}$, $\mathcal{V}$, p and d accordingly and we set the typing of the type to adhere to the intrastate invariant 5. For the yet unnamed types we perform the steps explained in definitions 13 and 15.

Definition 17 (Algorithm initialization). *The initial state is: $\mathcal{V} = \mathcal{G} = \mathcal{P} = \mathcal{D} = \mathcal{K} = \mathcal{C} = \emptyset$, $p = d = u = b_K = b_C = \emptyset \to \emptyset$ and $\mathcal{F}$ is the list of facts to be satisfied.*

Before the algorithm starts, we introduce the variables encountered in $\mathcal{F}$ according to the definition 15.

Definition 18 (Algorithm steps). *After the prestep (definition 16), we retrieve the first fact f of $\mathcal{F}$ and then:*

- *If $f \equiv t_1 \sim t_2$, we unify t_1 and t_2*
- *If $f \equiv t_1 \sim_T t_2$, we unify the typings $t(d(t_1))$ and $t(d(t_2))$, resulting in the unification m which we then apply to all typings in $\mathcal{D}$.*

- *If* $f \equiv t_1 \leq_K t_2$, *we add an edge* (t_1, t_2) *to* $\mathcal{K}$.
- *If* $f \equiv t_1 \leq_C t_2$, *we add an edge* (t_1, t_2) *to* $\mathcal{C}$.

Then we perform the middlestep (definition 14) and continue to the next step.