Online appendix to:

Polymorphic Functions with Set-Theoretic Types

Part 2: Local Type Inference and Type Reconstruction

Giuseppe Castagna[1] Kim Nguyễn[2] Zhiwu Xu[1,3] Pietro Abate[1]

[1]CNRS, PPS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France [2]LRI, Université Paris-Sud, Orsay, France
[3]Training and Evaluation Center, Guangdong Power Grid Company, Guangzhou, China

POPL '15, January 15–17 2015, Mumbai, India. Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676991

A. Examples of code

In this section we show examples of real code that follows the same pattern as the `even` function we defined in the introduction.

A.1 Red-Black trees

As a first example we show that the use of polymorphic set-theoretic types yields a better definition of Okasaki's implementation of red-black trees.

A red and black tree is a colored binary search tree in which all nodes are colored either black or red and that satisfies 4 invariants:

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from the root to a leaf contains the same number of black nodes

Thanks to our type system (and contrary to Okasaki's version) the implementation below ensures *by typing* that the operations on red-black trees (notably, the insertion) satisfy the first three invariants, as well as, that the `ins_aux` function, local to insertion, never returns empty trees (yet another important property that, in ML/Haskell Okasaki's version, types cannot ensure).

```
type RBtree(α) = Btree(α) | Rtree(α)

(* Black rooted RB tree: *)
type Btree(α)  = [] | <black elem=α>[ RBtree(α) RBtree(α) ]

(* Red rooted RB tree: *)
type Rtree(α)  =  <red elem=α>[ Btree(α) Btree(α) ]

type Wrongtree(α)  =   <red elem=α>( [ Rtree(α) Btree(α) ]
                                   | [ Btree(α) Rtree(α) ])

type Unbalanced(α) = <black elem=α>( [ Wrongtree(α) RBtree(α) ]
                                   | [ RBtree(α) Wrongtree(α) ])


let balance ( Unbalanced(α) -> Rtree(α) ; β\Unbalanced(α) -> β\Unbalanced(α) )
  | (<black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ] |
     <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ] |
     <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ] |
     <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ] ) & Unbalanced(α)
       -> <red (y)>[  <black (x)>[ a b ]   <black (z)>[ c d ]  ]
  |  x -> x


let insert (x : α) (t : Btree(α)) : Btree(α) =
  let ins_aux ( [] -> Rtree(α);
                Btree(α)\[] -> RBtree(α)\[];
                Rtree(α) -> Rtree(α)|Wrongtree(α) )
    | [] -> <red elem=x>[ [] [] ]
    | (<(color) elem=y>[ a b ]) & z ->
          if x << y then balance <(color) elem=y>[ (ins_aux a) b ]
          else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
          else z
   in match ins_aux t with
      | <_ (y)>[ a b ] -> <black (y)>[ a b ]
```

We invite the reader to refer to the excellent Okasaki's monograph [13] for details about Okasaki's algorithm —that our code faithfully follows— and the documentation of the language ℂDuce for details about the syntax we used, which is standard ℂDuce syntax apart from the presence of type variables.[8] Let us instead spend some words to comment the types, since they are the real novelty and the added value of our definition. First, notice that we used the full palette of our types: unions, intersections, negations (difference), and type variables. Red-black trees (`Btree`s) are black-rooted `RBtree`s (invariant 1), which

[8] For the reader convenience we recall that in ℂDuce XML types/pattern/expressions may have the form

`<`*tag attr=type/pattern/expression* `...` *attr=type/pattern/expression*`>[` *sequence of types/patterns/expression* `]`

and that possibly recursive functions can be defined as

```
let name (type->type; ... ; type->type)
  | pattern -> expression | ... | pattern -> expression
```

where the list of arrow types that follow the function name form the intersection type (*ie*, the interface) of the function.

are themselves black-rooted trees or red-rooted trees. The difference between the last two is that the latter cannot be leaves (invariant 2) and their children can only be black-rooted trees (invariant 3).

The `insert` function takes an element `x` of type α, and returns a function that maps red-black trees into red-black trees.

```
insert :: α -> Btree(α) -> Btree(α)
```

By examining the code of `insert` it is easy to see that if the argument tree is empty a red-rooted tree is returned, otherwise the element is inserted in the appropriate subtree and the whole tree is then balanced by the function `balance`. This function has the following type (which follows the same typing pattern as the function `even` defined in the introduction):

```
balance :: ( Unbalanced(α)->Rtree(α) ) & ( β\Unbalanced(α)->β\Unbalanced(α) )
```

This type states that `balance` transforms an unbalanced tree into a (balanced) red-rooted tree and leaves all other trees (in particular the balanced ones) unchanged. The core of our definition is the type of `ins_aux`:

```
ins_aux :: ( [] -> Rtree(α) )
         & ( Btree(α)\[] -> RBtree(α)\[] )
         & ( Rtree(α) -> Rtree(α)|Wrongtree(α) )
```

which precisely describes the behaviour of the function. Notice that the domain of `ins_aux` (which is the union of the three domains of the arrows forming its intersection type) is exactly `RBtree(`α`)`. The intersection type describes the behaviour of `ins_aux` for each form of an `RBtree` —*ie*, empty, black-rooted, and red-rooted—. The type system needs the full precision of this type to infer whether the calls to `balance` in the body of `ins_aux` are applied to a balanced or an unbalanced tree: even a slight approximation of this type, such as

```
ins_aux :: ( Btree(α)\[] -> RBtree(α)\[] )
         & ( Rtree(α)|[] -> Rtree(α)|Wrongtree(α) )
```

makes type-checking fail. By examining the type of `ins_aux` it is easy to see that `ins_aux` always returns either a (balanced) black-rooted tree or a tree with a red root in which one of the children may be a `Rtree`. In case of a tree with a red root, a (balanced) red-black tree is then obtained by changing the color of the root to black, as it is done in the last line of `insert`.

The implementation above must be compared with the corresponding version in monomorphic ℂDuce:

```
type RBtree = Btree | Rtree
type Btree  = [] | <black elem=Int>[ RBtree RBtree ]
type Rtree  =  <red elem=Int>[ Btree Btree ];;

type Wrongtree  =  Wrongleft | Wrongright
type Wrongleft  =  <red elem=Int>[ Rtree Btree ]
type Wrongright =  <red elem=Int>[ Btree Rtree ]
type Unbalanced =  <black elem=Int>([ Wrongtree RBtree ] | [ RBtree Wrongtree ]);;

let balance ( Unbalanced -> Rtree ; Rtree -> Rtree ; Btree\[] -> Btree\[] ;
              [] -> [] ; Wrongleft -> Wrongleft ; Wrongright -> Wrongright)
  | <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ] |
    <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ] |
    <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ] |
    <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ]
      ->  <red (y)>[  <black (x)>[ a b ]   <black (z)>[ c d ]  ]
  | x -> x

let insert (x : Int) (t : Btree) : Btree =
 let ins_aux ( [] -> Rtree ; Btree\[] -> RBtree\[]; Rtree -> Rtree|Wrongtree)
    | [] -> <red elem=x>[ [] [] ]
    | (<(color) elem=y>[ a b ]) & z ->
           if x << y then balance <(color) elem=y>[ (ins_aux a) b ]
           else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
           else z
   in match ins_aux t with
      | <_ (y)>[ a b ] -> <black (y)>[ a b ]
```

which, besides being monomorphic, requires the introduction of several intermediate types (in particular `Wrongleft` and `Wrongright`) in order to describe the polymorphic behavior of `balance` —whose type results, thus, much more obscure—. Our implementation must also be compared with the version given by Rowan Davies in his PhD Thesis [7] which uses polymorphic intersection types and type inference. Contrary to our definition, Davies's implementation (i) does not statically verify invariant 1, (ii) it introduces —as our monomorphic version does— several intermediate type definitions to specify the behavior of local functions, and (iii) it does not faithfully reproduce Okasaki implementation since it needs the definition of several auxiliary functions absent from Okasaki's (and our) formulation.

Likewise, there exist other implementations that are able to ensure/verify the first invariants of red-black trees (*eg*, by using GADTs or finite tree automata) but they all need extra definitions of intermediate functions or operations: as far as we know ours types are the only system that can statically ensure the invariants above simply by decorating (with types) the original Okasaki's code without any further modification.

Notice that the definition of `balance` given above and the one in Section 3.3 differ for a couple of details. First, the name of the types require mandatory type parameters when their definitions contain free type variable (*eg*, we have to write `RBtree(α)` rather than just `RBtree`): this is the behavior implemented in the current development version of ℂDuce, and we omitted this detail in the code of Section 3.3 just for space reasons. More importantly, the union pattern of the first branch of the pattern matching is intersected with the type `Unbalanced(α)`. The reason is that this type is *strictly* contained in the type accepted by the union pattern and, thereforere, this branch can be selected for values that are not in `Unbalanced(α)` (notice that `balance` can be applied to any value). Now for these values the interfaces of `balance` declares that a result of the same type is returned, which is not true since the first rather than the second branch is selected (and the latter transforms a black-rooted tree into a red-rooted one). This is why without the intersection in the pattern the type-checker rejects the definition by pointing out the problem. By adding the intersection we force the first branch to be selected only for values of type `Unbalanced(α)`, and the function type-checks.

There is still a last glitch, at least to run the example on the current development version of ℂDuce. Notice that we used `Unbalanced(α)` in the pattern and that this type contains the monomorphic variable α. The current develoment version of ℂDuce does not allow monomorphic variables to occur in patterns, yet (this is listed as future work in Section 7). In order to execute `balance` on the current development version of ℂDuce there are (at least) three solutions, which are all as valid as the one presented before. The first solution is to restrict the domain of `balance` to `Unbalanced(α) | RBtree(α)`. This can be done by declaring for `balance` the interface `( Unbalanced(α) -> Rtree(α) ; β&RBtree(α) -> β&RBtree(α) )` (notice that the intersection of `Unbalanced(α)` and `RBtree(α)` is empty so the difference in the interface and the intersection in the pattern are no longer necessary). The second solution is to use as intersection in the pattern the type `Unbalanced(Any)` to overapproximate `Unbalanced(α)`. This however requires to modify the interface of `balance` accordingly into `( Unbalanced(α)->Rtree(α) ; β\Unbalanced(Any) -> β\Unbalanced(Any) )` to capture the precise cases in which the second branch is selected. The third, more verbose, solution is to get rid again of the intersection pattern by declaring in the interface that the second arrow type applies only to values that are not in the accepted type of the first union pattern, that is, `( Unbalanced(α) -> Rtree(α) ; β\UTree -> β\UTree)`, where `UTree` is the accepted type of the first union pattern, which is obtained by replacing in the pattern `Any` for every capture variable occurring in it, namely:

```
type UTree = <black (Any)>[ <red (Any)>[ <red (Any)>[ Any Any ] Any ] Any ]
           | <black (Any)>[ <red (Any)>[ Any <red (Any)>[ Any Any ] ] Any ]
           | <black (Any)>[ Any <red (Any)>[ <red (Any)>[ Any Any ] Any ] ]
           | <black (Any)>[ Any <red (Any)>[ Any <red (Any)>[ Any Any ] ] ]
```

A.2 Soap envelopes

As a second usage example of the typing pattern followed by `even` we explore a typical XML application to process envelopes. Soap envelopes are a standardized format to communicate information wrapped in XML. An envelope contains a body and an optional header as described by the following type definitions.

```
type Envelope(α,β) = <Envelope>[ Header(β)? Body(α) ]
type Header(β) = <Header>β
type Body(α) = <Body>α
```

We define an `enrich` function which maps functions into a function with the same typing pattern as `even` (NOTE: in the current development version of ℂDuce this function cannot be executed because neither the `::` notation, nor monomorphic varibles in patterns are implemented, yet).

```
(* enrich envelope headers with info computed by applying *)
(* the argument function f to the body of the envelope    *)

enrich :: ( α -> β ) ->
                ( ( Envelope(α,β) -> <Envelope>[Header(β) Body(α)] )
                & ( γ\Envelope(α,β) -> γ\Envelope(α,β) )
                )

enrich f x = match x with
  | (<Envelope>[ <Body> b ]) & Envelope(α,β) ->
        <Envelope> [ <Header>(f b)   <Body>(enrich f b) ]
  | (<Envelope>[ <Header> h <Body> b ]) & Envelope(α,β) ->
        <Envelope> [ <Header>((f b)@h) <Body>(enrich f b) ]
  | lst & [ (AnyXML\Envelope(α,β)|Char)* ] -> (map lst with y -> enrich f y)
```

```
    | <(x)>y -> <(x)>(enrich f y)
    | y -> y
```

When applied to a function `f`, `enrich` returns a function that adds to the header of an envelope the result obtained by applying `f` to the body of the envelope, and recursively applies this transformation inside the body and in possible subtrees. Once more, in the definition of `enrich` we used pattern with monomorphic variables, therefore the same considerations as for the red-black tree example apply, too. The function `enrich` can then be typically used as in

```
  xtransform anXMLdoc with
    | x & T ->  enrich f_T x
    | x & <Envelope>_ -> enrich f x
```

where `f_T` is specific for type `T` and `f` is generic. The expression above transforms all the envelopes in the *anXMLdoc* document by preserving the type of all its subcomponents with the addition of the information on the headers, when it is missing.

Again this must be contrasted with the monomorphic version which must list all possible alternatives for the input type and in which the types of the contents of the envelope and the header are not preserved since they are subsumed to `[(AnyXml|Char)*]`:

```
type Envelope = <Envelope>[ Header? Body ]
type Header = <Header> B
type Body = <Body> A

type A = [(AnyXml|Char)*]
type B = [(AnyXml|Char)*]

let enrich (f: A -> B):  ( (Envelope -> <Envelope>[ Header Body ])
                         & (A -> A) & (Char -> Char) & (AnyXml -> AnyXml)) =
   (fun ( Envelope ->  <Envelope>[ Header Body ] ;
          A -> A ; Char -> Char ; AnyXml -> AnyXml)
     | <Envelope>[ <Body> b ] ->
           <Envelope> [ <Header>(f b)   <Body>(enrich f b) ]
     | <Envelope>[ <Header> h <Body> b ] ->
           <Envelope> [ <Header>((f b)@h) <Body>(enrich f b) ]
     | lst & [ (AnyXml|Char)* ] -> (map lst with y -> enrich f y)
     | <(x)>y -> <(x)> (enrich f y)
     | y -> y);;
```

B. Implicitly-Typed Calculus

We want sets of type-substitutions to be inferred by the system, not written by the programmer. To this end, we define a calculus without type substitutions (called *implicitly-typed*, in contrast to the calculus in (7) in Section 2, which we henceforth call *explicitly-typed*), for which we define a type-substitutions inference system. As explained in Section 3, we do not try to infer decorations in λ-abstractions, and we therefore look for completeness of the type-substitutions inference system with respect to the expressions written according to the following grammar:

$$e ::= c \mid x \mid (e,e) \mid \pi_i(e) \mid e\ e \mid \lambda^{\wedge_{i\in I} t_i \to s_i} x.e \mid e{\in}t\,?\,e : e \mid e[\sigma_j]_{j\in J}.$$

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

B.1 Implicitly-typed Calculus

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

$$a ::= c \mid x \mid (a,a) \mid \pi_i(a) \mid a\ a \mid \lambda^{\wedge_{i\in I} t_i \to s_i} x.a \mid a{\in}t\,?\,a : a$$

where t_i, s_i range over types and $t \in \mathscr{T}_0$ is a ground type. We write $\mathscr{E}_A$ to denote the set of all implicitly-typed expressions.

Clearly, $\mathscr{E}_A$ is a proper subset of $\mathscr{E}_0$.

The erasure of explicitly-typed expressions to implicitly-typed expressions is defined as follows:

Definition B.2. *The* erasure *is the mapping from $\mathscr{E}_0$ to $\mathscr{E}_A$ defined as*

$$\begin{array}{rcl}
\mathit{erase}(c) & = & c \\
\mathit{erase}(x) & = & x \\
\mathit{erase}((e_1, e_2)) & = & (\mathit{erase}(e_1), \mathit{erase}(e_2)) \\
\mathit{erase}(\pi_i(e)) & = & \pi_i(\mathit{erase}(e)) \\
\mathit{erase}(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) & = & \lambda^{\wedge_{i\in I} t_i \to s_i} x.\mathit{erase}(e) \\
\mathit{erase}(e_1 e_2) & = & \mathit{erase}(e_1)\mathit{erase}(e_2) \\
\mathit{erase}(e{\in}t\,?\,e_1 : e_2) & = & \mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2) \\
\mathit{erase}(e[\sigma_j]_{j\in J}) & = & \mathit{erase}(e)
\end{array}$$

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

Definition B.3. *Let s and t be two types and Δ a set of type variables. We define the following relations:*

$$[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \quad \overset{\text{def}}{\Longleftrightarrow} \quad \bigwedge_{i\in I} s\sigma_i \leq t \text{ and } \forall i{\in}I.\ \sigma_i \sharp \Delta$$

$$s \sqsubseteq_\Delta t \quad \overset{\text{def}}{\Longleftrightarrow} \quad \exists[\sigma_i]_{i\in I} \text{ such that } [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$$

We write $s \not\sqsubseteq_\Delta t$ if it does not exist a set of type substitutions $[\sigma_i]_{i\in I}$ such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$. We now prove some properties of the preorder $\sqsubseteq_\Delta$.

Lemma B.4. *Let t_1 and t_2 be two types and Δ a set of type variables. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $(t_1 \wedge t_2) \sqsubseteq_\Delta (s_1 \wedge s_2)$ and $(t_1 \times t_2) \sqsubseteq_\Delta (s_1 \times s_2)$.*

Proof. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2\in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then

$$\begin{array}{rcl}
\bigwedge_{i\in I_1\cup I_2}(t_1 \wedge t_2)\sigma_i & \simeq & (\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \wedge (\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i) \\
& \leq & (\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \wedge (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2}) \\
& \leq & s_1 \wedge s_2
\end{array}$$

and

$$\begin{array}{rcl}
\bigwedge_{i\in I_1\cup I_2}(t_1 \times t_2)\sigma_i & \simeq & ((\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \times (\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i)) \\
& \leq & ((\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \times (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2})) \\
& \leq & (s_1 \times s_2)
\end{array}$$

□

Lemma B.5. *Let t_1 and t_2 be two types and Δ a set of type variables such that $(\mathsf{var}(t_1) \setminus \Delta) \cap (\mathsf{var}(t_2) \setminus \Delta) = \emptyset$. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $t_1 \vee t_2 \sqsubseteq_\Delta s_1 \vee s_2$.*

Proof. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2\in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then we construct another set of type substitutions $[\sigma_{i_1,i_2}]_{i_1\in I_1, i_2\in I_2}$ such that

$$\sigma_{i_1,i_2}(\alpha) = \begin{cases} \sigma_{i_1}(\alpha) & \text{if } \alpha \in (\mathsf{var}(t_1) \setminus \Delta) \\ \sigma_{i_2}(\alpha) & \text{if } \alpha \in (\mathsf{var}(t_2) \setminus \Delta) \\ \alpha & \text{otherwise} \end{cases}$$

So we have

$$\begin{array}{rcl}
\bigwedge_{i_1\in I_1, i_2\in I_2}(t_1 \vee t_2)\sigma_{i_1,i_2} & \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}(t_1 \vee t_2)\sigma_{i_1,i_2}) \\
& \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}((t_1\sigma_{i_1,i_2}) \vee (t_2\sigma_{i_1,i_2}))) \\
& \simeq & \bigwedge_{i_1\in I_1}(\bigwedge_{i_2\in I_2}(t_1\sigma_{i_1} \vee t_2\sigma_{i_2})) \\
& \simeq & \bigwedge_{i_1\in I_1}(t_1\sigma_{i_1} \vee (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2})) \\
& \simeq & (\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \vee (\bigwedge_{i_2\in I_2} t_2\sigma_{i_2}) \\
& \leq & s_1 \vee s_2
\end{array}$$

□

Notice that two successive instantiations can be safely merged into one (see Lemma B.6). Henceforth, we assume that there are no successive instantiations in a given derivation tree. In order to guess where to insert sets of type-substitutions in an implicitly-typed expression, we consider each typing rule of the explicitly-typed calculus used in conjunction with the instantiation rule (ALG-INST). If instantiation can be moved through a given typing rule without affecting typability or changing the result type, then it is not necessary to infer type substitutions at the level of this rule.

Lemma B.6. *Let e be an explicitly-typed expression and $[\sigma_i]_{i\in I}$, $[\sigma_j]_{j\in J}$ two sets of type substitutions. Then*

$$\Delta \,{}_{\S}\, \Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : t \Longleftrightarrow \Delta \,{}_{\S}\, \Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : t$$

Proof. $\Rightarrow$**:** consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e : s} \quad \sigma_i \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i\in I} : \bigwedge_{i\in I} s\sigma_i} \quad \sigma_j \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j}$$

As $\sigma_i \sharp \Delta$, $\sigma_j \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_i) \cup \mathsf{dom}(\sigma_j)$, we have $\sigma_j \circ \sigma_i \sharp \Delta$. Then by (ALG-INST), we have $\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e([\sigma_j \circ \sigma_i]_{j\in J, i\in I}) : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)$, that is $\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j$.

$\Leftarrow$**:** consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e : s} \quad \sigma_j \circ \sigma_i \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)}$$

As $\sigma_j \circ \sigma_i \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_i) \cup \mathsf{dom}(\sigma_j)$, we have $\sigma_i \sharp \Delta$ and $\sigma_j \sharp \Delta$. Then applying the rule (ALG-INST) twice, we have $\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{i\in I} s\sigma_i)\sigma_j$, that is $\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_i]_{i\in I})[\sigma_j]_{j\in J} : \bigwedge_{j\in J, i\in I} s(\sigma_j \circ \sigma_i)$.

□

First of all, consider a typing derivation ending with (ALG-PAIR) where both of its sub-derivations end with (ALG-INST)[9]:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_1 : t_1} \quad \forall j_1 \in J_1.\, \sigma_{j_1} \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} t_1\sigma_{j_1}} \quad \dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_2 : t_2} \quad \forall j_2 \in J_2.\, \sigma_{j_2} \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} t_1\sigma_{j_2}}}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e_1[\sigma_{j_1}]_{j_1\in J_1}, e_2[\sigma_{j_2}]_{j_2\in J_2}) : (\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})}$$

We rewrite such a derivation as follows:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e_2 : t_2}}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2} \quad \forall j \in J_1 \cup J_2.\, \sigma_j \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} (e_1, e_2)[\sigma_j]_{j\in J_1\cup J_2} : \bigwedge_{j\in J_1\cup J_2}(t_1 \times t_2)\sigma_j}$$

Clearly, $\bigwedge_{j\in J_1\cup J_2}(t_1 \times t_2)\sigma_j \leq (\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})$. Then by subsumption we can deduce that $(e_1, e_2)[\sigma_j]_{j\in J_1\cup J_2}$ also has the type $(\bigwedge_{j_1\in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2\in J_2} t_1\sigma_{j_2})$. Therefore, we can disregard the sets of type substitutions that are applied inside a pair, since inferring them outside the pair is equivalent. Hence, we can use the following inference rule for pairs.

$$\dfrac{\Delta\,⨟\,\Gamma \vdash_{\mathscr{I}} a_1 : t_1 \quad \Delta\,⨟\,\Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}$$

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

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e : t} \quad \forall j \in J.\, \sigma_j \sharp \Delta}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j} \quad (\bigwedge_{j\in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{A}} \pi_i(e[\sigma_j]_{j\in J}) : \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)}$$

According to Lemma C.8 in the companion paper [3], we have $\boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j) \leq \bigwedge_{j\in J} \boldsymbol{\pi}_i(t)\sigma_j$, but the converse does not necessarily hold. For example, $\boldsymbol{\pi}_1(((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2))\{{}^{s_2}\!/_{\alpha}\}) = t_1\{{}^{s_2}\!/_{\alpha}\}$ while $(\boldsymbol{\pi}_1((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2)))\{{}^{s_2}\!/_{\alpha}\} = (t_1 \vee s_1)\{{}^{s_2}\!/_{\alpha}\}$. So we cannot exchange the instantiation and projection rules without losing completeness. However, as $(\bigwedge_{j\in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}$ and $\forall j \in J.\, \sigma_j \sharp \Delta$, we have $t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$. This indicates that for an implicitly-typed expression $\pi_i(a)$, if the inferred type for a is t and there exists $[\sigma_j]_{j\in J}$ such that $[\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$, then we infer the type $\boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)$ for $\pi_i(a)$. Let $\amalg^i_\Delta(t)$ denote the set of such result types, that is,

$$\amalg^i_\Delta(t) = \{u \mid [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}, u = \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)\}$$

Formally, we have the following inference rule for projections

$$\dfrac{\Delta\,⨟\,\Gamma \vdash_{\mathscr{I}} a : t \quad u \in \amalg^i_\Delta(t)}{\Delta\,⨟\,\Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

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

Lemma B.7. *Let t be a type and Δ a set of type variables. If $u_1 \in \amalg^i_\Delta(t)$ and $u_2 \in \amalg^i_\Delta(t)$, then $\exists u_0 \in \amalg^i_\Delta(t).\ u_0 \leq u_1 \wedge u_2$.*

[9] If one of the sub-derivations does not end with (ALG-INST), we can apply a trivial instance of (ALG-INST) with an identity substitution σ_{id}.

Proof. Let $[\sigma_{j_k}]_{j_k\in J_k} \Vdash t \sqsubseteq_\Delta \mathbb{1}\times\mathbb{1}$ and $u_k = \boldsymbol{\pi}_i(\bigwedge_{j_k\in J_k} t\sigma_{j_k})$ for $k=1,2$. Then $[\sigma_j]_{j\in J_1\cup J_2} \Vdash t \sqsubseteq_\Delta \mathbb{1}\times\mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{j\in J_1\cup J_2} t\sigma_j) \in \amalg^i_\Delta(t)$. Moreover, by Lemma C.6 in the companion paper [3], we have

$$\boldsymbol{\pi}_i(\bigwedge_{j\in J_1\cup J_2} t\sigma_j) \le \boldsymbol{\pi}_i(\bigwedge_{j_1\in J_1} t\sigma_{j_1}) \wedge \boldsymbol{\pi}_i(\bigwedge_{j_2\in J_2} t\sigma_{j_2}) = u_1\wedge u_2$$

□

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

$$\frac{\forall i\in I.\ \Delta\cup\mathsf{var}(\bigwedge_{i\in I}(t_i\to s_i))\,\S\,\Gamma, x:t_i \vdash_{\mathscr{A}} e: s_i' \text{ and } s_i'\le s_i}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I}t_i\to s_i}x.e : \bigwedge_{i\in I}(t_i\to s_i)}\ (\text{ALG-ABSTR0})$$

Suppose the last rule used in the sub-derivations is (ALG-INST).

$$\frac{\forall i\in I.\ \left\{\begin{array}{l} \dfrac{\dfrac{\cdots}{\Delta'\,\S\,\Gamma, x:t_i\vdash_{\mathscr{A}} e:s_i'}\quad \forall j\in J.\ \sigma_j\sharp\Delta'}{\Delta'\,\S\,\Gamma, x:t_i\vdash_{\mathscr{A}} e[\sigma_j]_{j\in J}:\bigwedge_{j\in J}s_i'\sigma_j} \\ \bigwedge_{j\in J}s_i'\sigma_j\le s_i\end{array}\right. \\ \Delta' = \Delta\cup\mathsf{var}(\bigwedge_{i\in I}(t_i\to s_i))}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}}\lambda^{\wedge_{i\in I}t_i\to s_i}x.e[\sigma_j]_{j\in J}:\bigwedge_{i\in I}(t_i\to s_i)}$$

From the side conditions, we deduce that $s_i'\sqsubseteq_{\Delta'} s_i$ for all $i\in I$. Instantiation may be necessary to bridge the gap between the computed type s_i' for e and the type s_i required by the interface, so inferring type substitutions at this stage is mandatory. Therefore, we propose the following inference rule for abstractions.

$$\frac{\forall i\in I.\ \left\{\begin{array}{l}\Delta\cup\mathsf{var}(\bigwedge_{i\in I}t_i\to s_i)\,\S\,\Gamma,(x:t_i)\vdash_{\mathscr{I}} a:s_i' \\ s_i'\sqsubseteq_{\Delta\cup\mathsf{var}(\bigwedge_{i\in I}t_i\to s_i)} s_i\end{array}\right.}{\Delta\,\S\,\Gamma\vdash_{\mathscr{I}}\lambda^{\wedge_{i\in I}t_i\to s_i}x.a:\bigwedge_{i\in I}t_i\to s_i}$$

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

$$\frac{\begin{array}{ll}\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}} e_1:t}\quad\forall j_1\in J_1.\ \sigma_{j_1}\sharp\Delta}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1}:\bigwedge_{j_1\in J_1}t\sigma_{j_1}} & \dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}} e_2:s}\quad\forall j_2\in J_2.\ \sigma_{j_2}\sharp\Delta}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2}:\bigwedge_{j_2\in J_2}s\sigma_{j_2}} \\ \bigwedge_{j_1\in J_1}t\sigma_{j_1}\le\mathbb{0}\to\mathbb{1} & \bigwedge_{j_2\in J_2}s\sigma_{j_2}\le\mathsf{dom}(\bigwedge_{j_1\in J_1}t\sigma_{j_1})\end{array}}{\Delta\,\S\,\Gamma\vdash_{\mathscr{A}}(e_1[\sigma_{j_1}]_{j_1\in J_1})(e_2[\sigma_{j_2}]_{j_2\in J_2}):(\bigwedge_{j_1\in J_1}t\sigma_{j_1})\cdot(\bigwedge_{j_2\in J_2}s\sigma_{j_2})}$$

Instantiation may be needed to bridge the gap between the (domain of the) function type and its argument (e.g., to apply $\lambda^{\alpha\to\alpha}x.x$ to `42`). The side conditions imply that $[\sigma_{j_1}]_{j_1\in J_1}\Vdash t\sqsubseteq_\Delta\mathbb{0}\to\mathbb{1}$ and $[\sigma_{j_2}]_{j_2\in J_2}\Vdash s\sqsubseteq_\Delta\mathsf{dom}(\bigwedge_{j_1\in J_1}t\sigma_{j_1})$. Therefore, given an implicitly-typed application a_1a_2 where a_1 and a_2 are typed with t and s respectively, we have to find two sets of substitutions $[\sigma_{j_1}]_{j_1\in J_1}$ and $[\sigma_{j_2}]_{j_2\in J_2}$ verifying the above preorder relations to be able to type the application. If such sets of substitutions exist, then we can type the application with $(\bigwedge_{j_1\in J_1}t\sigma_{j_1})\cdot(\bigwedge_{j_2\in J_2}s\sigma_{j_2})$. Let $t\bullet_\Delta s$ denote the set of such result types, that is,

$$t\bullet_\Delta s \stackrel{\text{def}}{=} \left\{ u\ \middle|\ \begin{array}{l}[\sigma_i]_{i\in I}\Vdash t\sqsubseteq_\Delta\mathbb{0}\to\mathbb{1}\\ [\sigma_j]_{j\in J}\Vdash s\sqsubseteq_\Delta\mathsf{dom}(\bigwedge_{i\in I}t\sigma_i)\\ u=\bigwedge_{i\in I}t\sigma_i\cdot\bigwedge_{j\in J}s\sigma_j\end{array}\right\}$$

This set is closed under intersection (see Lemma B.8). Formally, we get the following inference rule for applications

$$\frac{\Delta\,\S\,\Gamma\vdash_{\mathscr{I}} a_1:t\quad\Delta\,\S\,\Gamma\vdash_{\mathscr{I}} a_2:s\quad u\in t\bullet_\Delta s}{\Delta\,\S\,\Gamma\vdash_{\mathscr{I}} a_1a_2:u}$$

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

Proof. Let $u_k=(\bigwedge_{i_k\in I_k}t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k}s\sigma_{j_k})$ for $k=1,2$. According to Lemma C.18 in the companion paper [3], we have $(\bigwedge_{i\in I_1\cup I_2}t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2}s\sigma_j)\in t\bullet_\Delta s$ and $(\bigwedge_{i\in I_1\cup I_2}t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2}s\sigma_j)\le\bigwedge_{k=1,2}(\bigwedge_{i_k\in I_k}t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k}s\sigma_{j_k})=u_1\wedge u_2$.

□

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

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \qquad \bigwedge_{j\in J} t'\sigma_j \leq \mathbb{0}}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2 : \mathbb{0}}$$

Clearly, the side conditions implies $t' \sqsubseteq_\Delta \mathbb{0}$. The type inference rule for implicitly-typed expressions corresponding to this case is then

$$\dfrac{\Delta\,;\,\Gamma \vdash_{\mathscr{I}} a : t' \quad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta\,;\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \mathbb{0}}$$

For case (ii), consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \sigma_j \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq t \quad \dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1} \quad \sigma_{j_1} \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : e_2 : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

First, such a derivation can be rewritten as

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \sigma_j \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq t \quad \dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1}}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2 : s_1} \qquad \sigma_{j_1} \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_j]_{j\in J}){\in}t\,?\,e_1 : e_2)[\sigma_{j_1}]_{j_1\in J_1}) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

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

$$\dfrac{\Delta\,;\,\Gamma \vdash_{\mathscr{I}} a : t' \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \Delta\,;\,\Gamma \vdash_{\mathscr{I}} a_1 : s}{\Delta\,;\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}$$

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

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

$$\dfrac{\left\{\begin{array}{l} \dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_2 : s_2} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}} \end{array}\right.}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}{\in}t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : (e_2[\sigma_{j_2}]_{j_2\in J_2})) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}}$$

Using α-conversion if necessary, we can assume that the polymorphic type variables of e_1 and e_2 are distinct, and therefore we have $(\mathsf{var}(s_1) \setminus \Delta) \cap (\mathsf{var}(s_2) \setminus \Delta) = \emptyset$. According to Lemma B.5, we get $s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Let $[\sigma_{j_{12}}]_{j_{12}\in J_{12}} \Vdash s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. We can rewrite this derivation as

$$\dfrac{\dfrac{\left\{\begin{array}{l} \dfrac{\dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_1 : s_1} \\[2ex] \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\cdots}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} e_2 : s_2} \end{array}\right.}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} (e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2} \qquad \forall j_{12} \in J_{12}.\ \sigma_{j_{12}} \sharp \Delta}{\Delta\,;\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}) : \bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}}}$$

As $\bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}} \leq \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$, by subsumption, we can deduce that $(e[\sigma_j]_{j\in J}{\in}t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}$ has the type $\bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Hence, we eliminate the substitutions that are applied to these two branches.

We now consider the part of the derivation tree which concerns e. With the specific set of substitutions $[\sigma_j]_{j\in J}$, we have $\bigwedge_{j\in J} t'\sigma_j \not\leq \neg t$ and $\bigwedge_{j\in J} t'\sigma_j \not\leq t$, but it does not mean that we have $t' \not\sqsubseteq_\Delta t$ and

$t' \not\sqsubseteq_\Delta \neg t$ in general. If $t' \sqsubseteq_\Delta t$ and/or $t' \sqsubseteq_\Delta \neg t$ hold, then we are in one of the previous cases $(i) - (iii)$ (i.e., we type-check at most one branch), and the inferred result type for the whole type case belongs to $\mathbb{0}$, s_1 or s_2. We can then use subsumption to type the whole type-case expression with $s_1 \vee s_2$. Otherwise, both branches are type-checked, and we deduce the corresponding inference rule as follows.

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \quad \begin{cases} t' \not\sqsubseteq_\Delta \neg t & \text{and} \quad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 : s_1 \\ t' \not\sqsubseteq_\Delta t & \text{and} \quad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_2 : s_2 \end{cases}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

From the study above, we deduce the type-substitution inference rules for implicitly-typed expressions given in Figure 5, which are the same as those in Section 3 except for the rules for products.

$$\frac{}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} c : b_c}(\textsc{Inf-const}) \qquad \frac{}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} x : \Gamma(x)}(\textsc{Inf-var})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 : t_1 \qquad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}(\textsc{Inf-pair}) \qquad \frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t \qquad u \in \amalg^i_\Delta(t)}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}(\textsc{Inf-proj})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 : t \qquad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_2 : s \qquad u \in t \bullet_\Delta s}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}(\textsc{Inf-appl})$$

$$\frac{\forall i \in I. \begin{cases} \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,\S\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)} s_i \end{cases}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}(\textsc{Inf-abstr})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \mathbb{0}}(\textsc{Inf-case-none})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_\Delta t \qquad t' \not\sqsubseteq_\Delta \neg t \qquad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 : s}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}(\textsc{Inf-case-fst})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \qquad t' \sqsubseteq_\Delta \neg t \qquad t' \not\sqsubseteq_\Delta t \qquad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_2 : s}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}(\textsc{Inf-case-snd})$$

$$\frac{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \quad \begin{cases} t' \not\sqsubseteq_\Delta \neg t & \text{and} \quad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_1 : s_1 \\ t' \not\sqsubseteq_\Delta t & \text{and} \quad \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_2 : s_2 \end{cases}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}(\textsc{Inf-case-both})$$

Figure 5. Type-substitution inference rules

B.2 Soundness and Completeness

We now prove that the inference rules of the implicitly-typed calculus given in Figure 5 are sound and complete with respect to the type system of the explicitly-typed calculus (*i.e.*, Figure 1 extended with the standard rules for products).

To construct an explicitly-typed expression from an implicitly-typed one a, we have to insert sets of substitutions in a each time a preorder check is performed in the rules of Figure 5. For an abstraction $\lambda^{\wedge_{i\in I} t_i \to s_i} x.a$, different sets of substitutions may be constructed when type checking the body under the different hypotheses $x : t_i$. For example, let $a = \lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})} x.(\lambda^{\alpha\to\alpha} y.y)x$. When a is type-checked against $\texttt{Int} \to \texttt{Int}$, that is, x is assumed to have type $\texttt{Int}$, we infer the type substitution $\{\texttt{Int}/\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$. Similarly, we infer $\{\texttt{Bool}/\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$, when a is type-checked against $\texttt{Bool} \to \texttt{Bool}$. We have to collect these two different substitutions when constructing the explicitly-typed expression e which corresponds to a. To this end, we introduce an intersection operator $e \sqcap e'$ of expressions which is defined only for pair of expressions that have similar structure but different type substitutions. For example, the intersection of $(\lambda^{\alpha\to\alpha} y.y)[\{\texttt{Int}/\alpha\}]x$ and $(\lambda^{\alpha\to\alpha} y.y)[\{\texttt{Bool}/\alpha\}]x$ will be $(\lambda^{\alpha\to\alpha} y.y)[\{\texttt{Int}/\alpha\}, \{\texttt{Bool}/\alpha\}]x$.

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

$$\begin{array}{rcll}
c \sqcap c & = & c & \\
x \sqcap x & = & x & \\
(e_1, e_2) \sqcap (e'_1, e'_2) & = & ((e_1 \sqcap e'_1), (e_2 \sqcap e'_2)) & \\
\pi_i(e) \sqcap \pi_i(e') & = & \pi_i(e \sqcap e') & \\
e_1 e_2 \sqcap e'_1 e'_2 & = & (e_1 \sqcap e'_1)(e_2 \sqcap e'_2) & \\
(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e') & = & \lambda^{\wedge_{i\in I} t_i \to s_i} x.(e \sqcap e') & \\
(e_0 {\in} t \,?\, e_1 : e_2) \sqcap (e'_0 {\in} t \,?\, e'_1 : e'_2) & = & e_0 \sqcap e'_0 {\in} t \,?\, e_1 \sqcap e'_1 : e_2 \sqcap e'_2 & \\
(e_1[\sigma_j]_{j\in J}) \sqcap (e'_1[\sigma_j]_{j\in J'}) & = & (e_1 \sqcap e'_1)[\sigma_j]_{j\in J\cup J'} & \\
e \sqcap (e'_1[\sigma_j]_{j\in J'}) & = & (e[\sigma_{id}]) \sqcap (e'_1[\sigma_j]_{j\in J'}) & \textit{if } e \neq e_1[\sigma_j]_{j\in J} \\
(e_1[\sigma_j]_{j\in J}) \sqcap e' & = & (e_1[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_{id}]) & \textit{if } e' \neq e'_1[\sigma_j]_{j\in J'}
\end{array}$$

where σ_{id} is the identity type substitution and is undefined otherwise.

The intersection of the same constant or the same variable is the constant or the variable itself. If e and e' have the same form, then their intersection is defined if the intersections of their corresponding sub-expressions are defined. In particular when e is of the form $e_1[\sigma_j]_{j\in J}$ and e' is of the form form $e'_1[\sigma_j]_{j\in J'}$, we merge the sets of substitutions $[\sigma_j]_{j\in J}$ and $[\sigma_j]_{j\in J'}$ into one set $[\sigma_j]_{j\in J\cup J'}$. Otherwise, e and e' have different forms. The only possible case where their intersection is well-defined is when they have similar structures but one with instantiations and the other without (*i.e.*, $e = e_1[\sigma_j]_{j\in J}, e' \neq e'_1[\sigma_j]_{j\in J'}$ or $e \neq e_1[\sigma_j]_{j\in J}, e' = e'_1[\sigma_j]_{j\in J'}$). In order not to lose any inferred information and be able to reuse the cases defined above, we add the identity substitution σ_{id} to the expression without substitutions (*i.e.*, $e[\sigma_{id}]$ or $e'[\sigma_{id}]$). Let us infer the substitutions for the abstraction $\lambda^{(t_1\to s_1)\wedge(t_2\to s_2)}x.e$. Assume that we have inferred some substitutions for the body e under $t_1 \to s_1$ and $t_2 \to s_2$ respectively, yielding two explicitly-typed expressions e_1 and $e_2[\sigma_j]_{j\in J}$. If we did not add the identity substitution σ_{id} for the intersection of e_1 and $e_2[\sigma_j]_{j\in J}$, that is, $e_1 \sqcap (e_2[\sigma_j]_{j\in J})$ were $(e_1 \sqcap e_2)[\sigma_j]_{j\in J}$ rather than $(e_1 \sqcap e_2)([\sigma_{id}] \cup [\sigma_j]_{j\in J})$, then the substitutions we inferred under $t_1 \to s_1$ would be lost since they may be modified by $[\sigma_j]_{j\in J}$.

Lemma B.10. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, then $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$.*

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

$\underline{c, c}$**:** straightforward.

$\underline{x, x}$**:** straightforward.

$\underline{(e_1, e_2), (e'_1, e'_2)}$**:** we have $erase(e_i) = erase(e'_i)$. By induction, $e_i \sqcap e'_i$ exists and $erase(e_i \sqcap e'_i) = erase(e_i) = erase(e'_i)$. Therefore $(e_1, e_2) \sqcap (e'_1, e'_2)$ exists and

$$\begin{array}{rcl}
erase((e_1, e_2) \sqcap (e'_1, e'_2)) & = & erase(((e_1 \sqcap e'_1), (e_2 \sqcap e'_2))) \\
& = & (erase(e_1 \sqcap e'_1), erase(e_2 \sqcap e'_2)) \\
& = & (erase(e_1), erase(e_2)) \\
& = & erase((e_1, e_2))
\end{array}$$

Similarly, we also have $erase((e_1, e_2) \sqcap (e'_1, e'_2)) = erase((e'_1, e'_2))$.

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

$$\begin{array}{rcl}
erase(\pi_i(e) \sqcap \pi_i(e')) & = & erase(\pi_i(e \sqcap e')) \\
& = & \pi_i(erase(e \sqcap e')) \\
& = & \pi_i(erase(e)) \\
& = & erase(\pi_i(e))
\end{array}$$

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

$\underline{e_1 e_2, e'_1 e'_2}$**:** we have $erase(e_i) = erase(e'_i)$. By induction, $e_i \sqcap e'_i$ exists and $erase(e_i \sqcap e'_i) = erase(e_i) = erase(e'_i)$. Therefore $e_1 e_2 \sqcap e'_1 e'_2$ exists and

$$\begin{array}{rcl}
erase((e_1 e_2) \sqcap (e'_1 e'_2)) & = & erase((e_1 \sqcap e'_1)(e_2 \sqcap e'_2)) \\
& = & erase(e_1 \sqcap e'_1) erase(e_2 \sqcap e'_2) \\
& = & erase(e_1) erase(e_2) \\
& = & erase(e_1 e_2)
\end{array}$$

Similarly, we also have $erase((e_1 e_2) \sqcap (e'_1 e'_2)) = erase(e'_1 e'_2)$.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i} x.e, \lambda^{\wedge_{i\in I} t_i \to s_i} x.e'}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e')$ exists and

$$\begin{array}{rcl}
erase((\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e')) & = & erase(\lambda^{\wedge_{i\in I} t_i \to s_i} x.(e \sqcap e')) \\
& = & \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase((e \sqcap e')) \\
& = & \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase(e) \\
& = & erase(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e)
\end{array}$$

Similarly, we also have

$$erase((\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) \sqcap (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e')) = erase(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e')$$

<u>$e_0{\in}t\,?\,e_1 : e_2, e'_0{\in}t\,?\,e'_1 : e'_2$</u>**:** we have $erase(e_i) = erase(e'_i)$. By induction, $e_i \sqcap e'_i$ exists and $erase(e_i \sqcap e'_i) = erase(e_i) = erase(e'_i)$. Therefore $(e_0{\in}t\,?\,e_1 : e_2) \sqcap (e'_0{\in}t\,?\,e'_1 : e'_2)$ exists and

$$\begin{array}{rcl} erase((e_0{\in}t\,?\,e_1 : e_2) \sqcap (e'_0{\in}t\,?\,e'_1 : e'_2)) & = & erase((e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)) \\ & = & erase(e_0 \sqcap e'_0){\in}t\,?\,erase(e_1 \sqcap e'_1) : erase(e_2 \sqcap e'_2) \\ & = & erase(e_0){\in}t\,?\,erase(e_1) : erase(e_2) \\ & = & erase(e_0{\in}t\,?\,e_1 : e_2) \end{array}$$

Similarly, we also have

$$erase((e_0{\in}t\,?\,e_1 : e_2) \sqcap (e'_0{\in}t\,?\,e'_1 : e'_2)) = erase(e'_0{\in}t\,?\,e'_1 : e'_2)$$

<u>$e[\sigma_j]_{j\in J}, e'[\sigma_j]_{j\in J'}$</u>**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})$ exists and

$$\begin{array}{rcl} erase((e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})) & = & erase((e \sqcap e')[\sigma_j]_{j\in J\cup J'}) \\ & = & erase(e \sqcap e') \\ & = & erase(e) \\ & = & erase(e[\sigma_j]_{j\in J}) \end{array}$$

Similarly, we also have $erase((e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})) = erase(e'[\sigma_j]_{j\in J'})$.

<u>$e, e'[\sigma_j]_{j\in J'}$</u>**:** a special case of $e[\sigma_j]_{j\in J}$ and $e'[\sigma_j]_{j\in J'}$ where $[\sigma_j]_{j\in J} = [\sigma_{id}]$.

<u>$e[\sigma_j]_{j\in J}, e'$</u>**:** a special case of $e[\sigma_j]_{j\in J}$ and $e'[\sigma_j]_{j\in J'}$ where $[\sigma_j]_{j\in J'} = [\sigma_{id}]$.

□

Lemma B.11. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, $\Delta\,\S\,\Gamma \vdash e : t$, $\Delta'\,\S\,\Gamma' \vdash e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then $\Delta\,\S\,\Gamma \vdash e \sqcap e' : t$ and $\Delta'\,\S\,\Gamma' \vdash e \sqcap e' : t'$.*

Proof. According to Lemma B.10, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. We only prove $\Delta\,\S\,\Gamma \vdash e \sqcap e' : t$ as the other case is similar. For simplicity, we just consider one set of type substitutions. For several sets of type substitutions, we can either compose them or apply (*instinter*) several times. The proof proceeds by induction on $\Delta\,\S\,\Gamma \vdash e : t$.

<u>(*const*)</u>**:** $\Delta\,\S\,\Gamma \vdash c : b_c$. As $erase(e') = c$, e' is either c or $c[\sigma_j]_{j\in J}$. If $e' = c$, then $e \sqcap e' = c$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = c[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,\S\,\Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c \wedge \bigwedge_{j\in J} b_c\sigma_j$, that is, $\Delta\,\S\,\Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c$.

<u>(*var*)</u>**:** $\Gamma \vdash x : \Gamma(x)$. As $erase(e') = x$, e' is either x or $x[\sigma_j]_{j\in J}$. If $e' = x$, then $e \sqcap e' = x$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = x[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,\S\,\Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x) \wedge \bigwedge_{j\in J} \Gamma(x)\sigma_j$, that is, $\Delta\,\S\,\Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x)$.

<u>(*pair*)</u>**:** consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash e_2 : t_2}}{\Delta\,\S\,\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

As $erase(e') = (erase(e_1), erase(e_2))$, e' is either (e'_1, e'_2) or $(e'_1, e'_2)[\sigma_j]_{j\in J}$ such that $erase(e'_i) = erase(e_i)$. By induction, we have $\Delta\,\S\,\Gamma \vdash e_i \sqcap e'_i : t_i$. Then by (*pair*), we have $\Delta\,\S\,\Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2) : (t_1 \times t_2)$. If $e' = (e'_1, e'_2)$, then $e \sqcap e' = (e_1 \sqcap e'_1, e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,\S\,\Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2) \wedge \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$. Finally, by (*subsum*), we get $\Delta\,\S\,\Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2)$.

<u>(*proj*)</u>**:** consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash e_0 : t_1 \times t_2}}{\Delta\,\S\,\Gamma \vdash \pi_i(e_0) : t_i}\ (proj)$$

As $erase(e') = \pi_i(erase(e_0))$, e' is either $\pi_i(e'_0)$ or $\pi_i(e'_0)[\sigma_j]_{j\in J}$ such that $erase(e'_0) = erase(e_0)$. By induction, we have $\Delta\,\S\,\Gamma \vdash e_0 \sqcap e'_0 : (t_1 \times t_2)$. Then by (*proj*), we have $\Delta\,\S\,\Gamma \vdash \pi_i(e_0 \sqcap e'_0) : t_i$. If $e' = \pi_i(e'_0)$, then $e \sqcap e' = \pi_i(e_0 \sqcap e'_0)$. So the result follows.
Otherwise, $e \sqcap e' = \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,\S\,\Gamma \vdash \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : t_i \wedge \bigwedge_{j\in J} t_i\sigma_j$. Finally, by (*subsum*), we get $\Delta\,\S\,\Gamma \vdash \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : t_i$.

<u>(*appl*)</u>**:** consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash e_1 : t \to s} \quad \dfrac{\cdots}{\Delta\,\S\,\Gamma \vdash e_2 : t}}{\Delta\,\S\,\Gamma \vdash e_1 e_2 : s}\ (pair)$$

As *erase*$(e') = $ *erase*(e_1)*erase*(e_2), e' is either $e'_1e'_2$ or $(e'_1e'_2)[\sigma_j]_{j\in J}$ such that *erase*$(e'_i) = $ *erase*(e_i). By induction, we have $\Delta\,{}_{\S}\,\Gamma \vdash e_1 \sqcap e'_1 : t \to s$ and $\Delta\,{}_{\S}\,\Gamma \vdash e_2 \sqcap e'_2 : t$. Then by (*appl*), we have $\Delta\,{}_{\S}\,\Gamma \vdash (e_1 \sqcap e'_1)(e_2 \sqcap e'_2) : s$. If $e' = e'_1e'_2$, then $e \sqcap e' = (e_1 \sqcap e'_1)(e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,{}_{\S}\,\Gamma \vdash ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{}_{\S}\,\Gamma \vdash ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*abstr*)**:** consider the following derivation:

$$\frac{\begin{array}{c}\forall i \in I.\ \dfrac{\ldots}{\Delta''\,{}_{\S}\,\Gamma, (x : t_i) \vdash e_0 : s_i} \\ \Delta'' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)\end{array}}{\Delta\,{}_{\S}\,\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 : \bigwedge_{i\in I} t_i \to s_i}\ (abstr)$$

As *erase*$(e') = \lambda^{\wedge_{i\in I} t_i \to s_i} x.$*erase*$(e_0)$, e' is either $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$ or $(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0)[\sigma_j]_{j\in J}$ such that *erase*$(e'_0) = $ *erase*(e_0). As $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$ is well-typed under Δ' and Γ', $e'_0 \sharp \Delta' \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, we have $\Delta''\,{}_{\S}\,\Gamma, (x : t_i) \vdash e_0 \sqcap e'_0 : s_i$. Then by (*abstr*), we have $\Delta\,{}_{\S}\,\Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0 : \bigwedge_{i\in I} t_i \to s_i$. If $e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$, then $e \sqcap e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0$. So the result follows.
Otherwise, $e \sqcap e' = (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,{}_{\S}\,\Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : (\bigwedge_{i\in I} t_i \to s_i) \wedge \bigwedge_{j\in J}(\bigwedge_{i\in I} t_i \to s_i)\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{}_{\S}\,\Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : \bigwedge_{i\in I} t_i \to s_i$.

(*case*)**:** consider the following derivation

$$\frac{\dfrac{\ldots}{\Delta\,{}_{\S}\,\Gamma \vdash e_0 : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta\,{}_{\S}\,\Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta\,{}_{\S}\,\Gamma \vdash e_2 : s}\end{cases}}{\Delta\,{}_{\S}\,\Gamma \vdash (e_0{\in}t\,?\,e_1 : e_2) : s}\ (case)$$

As *erase*$(e') = $ *erase*$(e_0){\in}t\,?\,$*erase*$(e_1) : $ *erase*(e_2), e' is either $e'_0{\in}t\,?\,e'_1 : e'_2$ or $(e'_0{\in}t\,?\,e'_1 : e'_2)[\sigma_j]_{j\in J}$ such that *erase*$(e'_i) = $ *erase*(e_i). By induction, we have $\Delta\,{}_{\S}\,\Gamma \vdash e_0 \sqcap e'_0 : t'$ and $\Delta\,{}_{\S}\,\Gamma \vdash e_i \sqcap e'_i : s$. Then by (*case*), we have $\Delta\,{}_{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)) : s$. If $e' = e'_0{\in}t\,?\,e'_1 : e'_2$, then $e \sqcap e' = (e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = ((e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta\,{}_{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{}_{\S}\,\Gamma \vdash ((e_0 \sqcap e'_0){\in}t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*instinter*)**:** consider the following derivation:

$$\frac{\dfrac{\ldots}{\Delta\,{}_{\S}\,\Gamma \vdash e_0 : t} \quad \sigma_j \sharp \Delta}{\Delta\,{}_{\S}\,\Gamma \vdash e_0[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (instinter)$$

As *erase*$(e') = $ *erase*(e_0), e' is either e'_0 or $e'_0[\sigma_j]_{j\in J'}$ such that *erase*$(e'_0) = $ *erase*(e_0). By induction, we have $\Delta\,{}_{\S}\,\Gamma \vdash e_0 \sqcap e'_0 : t$. If $e' = e'_0$, then $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J}$. By (*instinter*), we have $\Delta\,{}_{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j \wedge t$. Finally, by (*subsum*), we get $\Delta\,{}_{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.
Otherwise, $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$ for all $j \in J'$. By (*instinter*), we have $\Delta\,{}_{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J\cup J'} t\sigma_j$. Finally, by (*subsum*), we get $\Delta\,{}_{\S}\,\Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J} t\sigma_j$.

(*subsum*)**:** there exists a type s such that

$$\frac{\dfrac{\ldots}{\Delta\,{}_{\S}\,\Gamma \vdash e : s} \quad s \leq t}{\Delta\,{}_{\S}\,\Gamma \vdash e : t}\ (subsum)$$

By induction, we have $\Delta\,{}_{\S}\,\Gamma \vdash e \sqcap e' : s$. Then the rule (*subsum*) gives us $\Delta\,{}_{\S}\,\Gamma \vdash e \sqcap e' : t$.

□

Corollary B.12. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If erase$(e) = $ erase(e'), $\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$, $\Delta'\,{}_{\S}\,\Gamma' \vdash_{\mathscr{A}} e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then*

1. *there exists s such that $\Delta\,{}_{\S}\,\Gamma \vdash_{\mathscr{A}} e \sqcap e' : s$ and $s \leq t$.*
2. *there exists s' such that $\Delta'\,{}_{\S}\,\Gamma' \vdash_{\mathscr{A}} e \sqcap e' : s'$ and $s' \leq t'$.*

Proof. Immediate consequence of Lemma B.11 and Theorems C.22 and C.23 in the companion paper [3]. □

These type-substitution inference rules are sound and complete with respect to the typing algorithm in Section C.2 in the companion paper [3], modulo the restriction that all the decorations in the λ-abstractions are empty.

Theorem B.13 (Soundness). *If $\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t$, then there exists an explicitly-typed expression $e \in \mathscr{E}_0$ such that $\mathit{erase}(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : t$.*

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t$. We proceed by a case analysis of the last rule used in the derivation.

(INF-CONST)**:** straightforward (take e as c).
(INF-VAR)**:** straightforward (take e as x).
(INF-PAIR)**:** consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 : t_1} \qquad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_2 : t_2}}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}$$

Applying the induction hypothesis, there exists an expression e_i such that $\mathit{erase}(e_i) = a_i$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_i : t_i$. Then by (ALG-PAIR), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2$. Moreover, according to Definition B.2, we have $\mathit{erase}((e_1, e_2)) = (\mathit{erase}(e_1), \mathit{erase}(e_2)) = (a_1, a_2)$.
(INF-PROJ)**:** consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t} \qquad u \in \mathbf{\Pi}^i_{\Delta}(t)}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

By induction, there exists an expression e such that $\mathit{erase}(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : t$. Let $u = \boldsymbol{\pi}_i(\bigwedge_{i \in I} t\sigma_i)$. As $\sigma_i \sharp \Delta$, by (ALG-INST), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i \in I} : \bigwedge_{i \in I} t\sigma_i$. Moreover, since $\bigwedge_{i \in I} t\sigma_i \leq \mathbb{1} \times \mathbb{1}$, by (ALG-PROJ), we get $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} \pi_i(e[\sigma_i]_{i \in I}) : \boldsymbol{\pi}_i(\bigwedge_{i \in I} t\sigma_i)$. Finally, according to Definition B.2, we have $\mathit{erase}(\pi_i(e[\sigma_i]_{i \in I})) = \pi_i(\mathit{erase}(e[\sigma_i]_{i \in I})) = \pi_i(\mathit{erase}(e)) = \pi_i(a)$.
(INF-APPL)**:** consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 : t} \qquad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_2 : s} \qquad u \in t \bullet_{\Delta} s}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}$$

By induction, we have that (i) there exists an expression e_1 such that $\mathit{erase}(e_1) = a_1$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1 : t$ and (ii) there exists an expression e_2 such that $\mathit{erase}(e_2) = a_2$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_2 : s$. Let $u = (\bigwedge_{i \in I} t\sigma_i) \cdot (\bigwedge_{j \in J} s\sigma_j)$. As $\sigma_h \sharp \Delta$ for $h \in I \cup J$, applying (ALG-INST), we get $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1[\sigma_i]_{i \in I} : \bigwedge_{i \in I} t\sigma_i$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_2[\sigma_j]_{j \in J} : \bigwedge_{j \in J} s\sigma_j$. Then by (ALG-APPL), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e_1[\sigma_i]_{i \in I})(e_2[\sigma_j]_{j \in J}) : (\bigwedge_{i \in I} t\sigma_i) \cdot (\bigwedge_{j \in J} s\sigma_j)$. Furthermore, according to Definition B.2, we have $\mathit{erase}((e_1[\sigma_i]_{i \in I})(e_2[\sigma_j]_{j \in J})) = \mathit{erase}(e_1)\mathit{erase}(e_2) = a_1 a_2$.
(INF-ABSTR)**:** consider the derivation

$$\frac{\forall i \in I. \left\{ \begin{array}{l} \dfrac{\dots}{\Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i} \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i)} s_i \end{array} \right.}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.a : \bigwedge_{i \in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i)$ and $[\sigma_{j_i}]_{j_i \in J_i} \Vdash s'_i \sqsubseteq_{\Delta'} s_i$. By induction, there exists an expression e_i such that $\mathit{erase}(e_i) = a$ and $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathscr{A}} e_i : s'_i$ for all $i \in I$. Since $\sigma_{j_i} \sharp \Delta'$, by (ALG-INST), we have $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathscr{A}} e_i[\sigma_{j_i}]_{j_i \in J_i} : \bigwedge_{j_i \in J_i} s'_i\sigma_{j_i}$. Clearly, $e_i[\sigma_{j_i}]_{j_i \in J_i} \sharp \Delta'$ and $\mathit{erase}(e_i[\sigma_{j_i}]_{j_i \in J_i}) = \mathit{erase}(e_i) = a$. Then by Lemma B.10, the intersection $\bigsqcap_{i \in I}(e_i[\sigma_{j_i}]_{j_i \in J_i})$ exists and we have $\mathit{erase}(\bigsqcap_{i \in I'}(e_i[\sigma_{j_i}]_{j_i \in J_i})) = a$ for any non-empty $I' \subseteq I$. Let $e = \bigsqcap_{i \in I}(e_i[\sigma_{j_i}]_{j_i \in J_i})$. According to Corollary B.12, there exists a type t'_i such that $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathscr{A}} e : t'_i$ and $t'_i \leq \bigwedge_{j_i \in J_i} s'_i\sigma_{j_i}$ for all $i \in I$. Moreover, since $t'_i \leq \bigwedge_{j_i \in J_i} s'_i\sigma_{j_i} \leq s_i$, by (ALG-ABSTR), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.e : \bigwedge_{i \in I}(t_i \to s_i)$. Finally, according to Definition B.2, we have

$$\mathit{erase}(\lambda^{\wedge_{i \in I} t_i \to s_i} x.e) = \lambda^{\wedge_{i \in I} t_i \to s_i} x.\mathit{erase}(e) = \lambda^{\wedge_{i \in I} t_i \to s_i} x.a.$$

(INF-CASE-NONE)**:** consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t'} \qquad t' \sqsubseteq_{\Delta} \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} (a \in t \,?\, a_1 : a_2) : \mathbb{0}}$$

By induction, there exists an expression e such that $\mathit{erase}(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : t'$. Let $[\sigma_i]_{i \in I} \Vdash t' \sqsubseteq_{\Delta} \mathbb{0}$. Since $\sigma_i \sharp \Delta$, by (ALG-INST), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e[\sigma_i]_{i \in I} : \bigwedge_{i \in I} t'\sigma_i$. Let e_1 and e_2 be two expressions such that $\mathit{erase}(e_1) = a_1$ and $\mathit{erase}(e_2) = a_2$. Then we have

$$\mathit{erase}((e[\sigma_i]_{i \in I}) \in t \,?\, e_1 : e_2) = (a \in t \,?\, a_1 : a_2).$$

Moreover, since $\bigwedge_{i \in I} t'\sigma_i \leq \mathbb{0}$, by (ALG-CASE-NONE), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} ((e[\sigma_i]_{i \in I}) \in t \,?\, e_1 : e_2) : \mathbb{0}.$$

(INF-CASE-FST): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a : t'} \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a_1 : s}}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s}$$

By induction, there exist e, e_1 such that $\mathit{erase}(e) = a$, $\mathit{erase}(e_1) = a_1$, $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t'$, and $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_1 : s$. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t' \sqsubseteq_\Delta t$. Since $\sigma_{i_1} \sharp \Delta$, applying (ALG-INST), we get $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e[\sigma_{i_1}]_{i_1\in I_1} : \bigwedge_{i_1\in I_1} t'\sigma_{i_1}$. Let e_2 be an expression such that $\mathit{erase}(e_2) = a_2$. Then we have

$$\mathit{erase}((e[\sigma_{i_1}]_{i_1\in I_1}){\in}t\,?\,e_1 : e_2) = (a{\in}t\,?\,a_1 : a_2).$$

Finally, since $\bigwedge_{i_1\in I_1} t'\sigma_{i_1} \leq t$, by (ALG-CASE-FST), we have

$$\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} ((e[\sigma_{i_1}]_{i_1\in I_1}){\in}t\,?\,e_1 : e_2) : s.$$

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

(INF-CASE-BOTH): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a : t'} \quad \begin{cases} t' \not\sqsubseteq_\Delta \neg t \quad \text{and} \quad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a_1 : s_1} \\ t' \not\sqsubseteq_\Delta t \quad \text{and} \quad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} a_2 : s_2} \end{cases}}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

By induction, there exist e, e_i such that $\mathit{erase}(e) = a$, $\mathit{erase}(e_i) = a_i$, $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t'$, and $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_i : s_i$. According to Definition B.2, we have $\mathit{erase}((e{\in}t\,?\,e_1 : e_2)) = (a{\in}t\,?\,a_1 : a_2)$. Clearly $t' \not\simeq \mathbb{0}$. We claim that $t' \not\leq \neg t$. Let σ_{id} be any identity type substitution. If $t' \leq \neg t$, then $t'\sigma_{id} \simeq t' \leq \neg t$, i.e., $t' \sqsubseteq_\Delta \neg t$, which is in contradiction with $t' \not\sqsubseteq_\Delta \neg t$. Similarly we have $t' \not\leq t$. Therefore, by (ALG-CASE-SND), we have $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2$.

□

The proof of the soundness property constructs along the derivation for a some expression e that satisfies the statement of the theorem. We denote by $\mathit{erase}^{-1}(a)$ the set of expressions e that satisfy the statement.

Theorem B.14 (Completeness). *Let $e \in \mathscr{E}_0$ be an explicitly-typed expression. If $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$, then there exists a type t' such that $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t'$ and $t' \sqsubseteq_\Delta t$.*

Proof. By induction on the typing derivation of $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t$. We proceed by a case analysis on the last rule used in the derivation.

(ALG-CONST): take t' as b_c.

(ALG-VAR): take t' as $\Gamma(x)$.

(ALG-PAIR): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_2 : t_2}}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2}$$

Applying the induction hypothesis twice, we have

$$\exists t'_1.\ \Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_\Delta t_1,$$
$$\exists t'_2.\ \Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_2) : t'_2 \text{ and } t'_2 \sqsubseteq_\Delta t_2.$$

Then by (INF-PAIR), we have $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e_1), \mathit{erase}(e_2)) : t'_1 \times t'_2$, that is, $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}((e_1, e_2)) : t'_1 \times t'_2$. Finally, Applying Lemma B.4, we have $(t'_1 \times t'_2) \sqsubseteq_\Delta (t_1 \times t_2)$.

(ALG-PROJ): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e : t} \quad t \leq \mathbb{1} \times \mathbb{1}}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_\Delta t.$$

It is clear that $\bigwedge_{k\in K} t'\sigma_k \leq \mathbb{1} \times \mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \in \amalg^i_\Delta(t')$. Then by (INF-PROJ), we have $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \pi_i(\mathit{erase}(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k)$, that is, $\Delta\,{\S}\,\Gamma \vdash_{\mathscr{I}} \mathit{erase}(\pi_i(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k)$. According to Lemma C.5 in the companion paper [3], $t \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Then $\bigwedge_{k\in K} t'\sigma_k \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Finally, applying Lemma C.5 again, we get $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \leq \boldsymbol{\pi}_i(t)$ and *a fortiori* $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \sqsubseteq_\Delta \boldsymbol{\pi}_i(t)$.

(ALG-APPL): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_1 : t} \quad \dfrac{\cdots}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_2 : s} \quad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta\,{\S}\,\Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s}$$

Applying the induction hypothesis twice, we have

$$\begin{array}{l}\exists t_1', [\sigma_k^1]_{k\in K_1}.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1) : t_1' \text{ and } [\sigma_k^1]_{k\in K_1} \Vdash t_1' \sqsubseteq_\Delta t,\\ \exists t_2', [\sigma_k^2]_{k\in K_2}.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_2) : t_2' \text{ and } [\sigma_k^2]_{k\in K_2} \Vdash t_2' \sqsubseteq_\Delta s.\end{array}$$

It is clear that $\bigwedge_{k\in K_1} t_1'\sigma_k^1 \leq \mathbb{0} \to \mathbb{1}$, that is, $\bigwedge_{k\in K_1} t_1'\sigma_k^1$ is a function type. So we get $\mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t_1'\sigma_k^1)$. Then we have $\bigwedge_{k\in K_2} t_2'\sigma_k^2 \leq s \leq \mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t_1'\sigma_k^1)$. Therefore, $(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \in t_2' \bullet_\Delta t_1'$. Then applying (INF-APPL), we have

$$\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1)\mathit{erase}(e_2) : (\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2),$$

that is, $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1e_2) : (\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2)$. Moreover, as $\bigwedge_{k\in K_2} t_2'\sigma_k^2 \leq \mathsf{dom}(t)$, $t \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2)$ exists. According to Lemma C.14 in the companion paper [3], we have

$$(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \leq t \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \leq t \cdot s.$$

Thus, $(\bigwedge_{k\in K_1} t_1'\sigma_k^1) \cdot (\bigwedge_{k\in K_2} t_2'\sigma_k^2) \sqsubseteq_\Delta t \cdot s$.

(ALG-ABSTR0): consider the derivation

$$\frac{\forall i \in I.\ \dfrac{\ldots}{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,⨟\, \Gamma, (x : t_i) \vdash_{\mathscr{A}} e : s_i'} \quad \text{and } s_i' \leq s_i}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.e : \bigwedge_{i\in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, for each $i \in I$, we have

$$\exists t_i'.\ \Delta' \,⨟\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} \mathit{erase}(e) : t_i' \text{ and } t_i' \sqsubseteq_{\Delta'} s_i'.$$

Clearly, we have $t_i' \sqsubseteq_{\Delta'} s_i$. By (INF-ABSTR), we have

$$\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.\mathit{erase}(e) : \bigwedge_{i\in I} t_i \to s_i,$$

that is, $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) : \bigwedge_{i\in I} t_i \to s_i$.

(ALG-CASE-NONE): consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : \mathbb{0}}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : \mathbb{0}}$$

By induction, we have

$$\exists t_0'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t_0' \text{ and } t_0' \sqsubseteq_\Delta \mathbb{0}.$$

By (INF-CASE-NONE), we have $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0}$, that is, $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$.

(ALG-CASE-FST): consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad t' \leq t \quad \dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1}$$

Applying the induction hypothesis twice, we have

$$\begin{array}{l}\exists t_0'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t_0' \text{ and } t_0' \sqsubseteq_\Delta t',\\ \exists t_1'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1) : t_1' \text{ and } t_1' \sqsubseteq_\Delta s_1.\end{array}$$

Clearly, we have $t_0' \sqsubseteq_\Delta t$. If $t_0' \sqsubseteq_\Delta \neg t$, then by Lemma B.4, we have $t_0' \leq_\Delta \mathbb{0}$. By (INF-CASE-NONE), we get

$$\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0},$$

that is, $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_\Delta s_1$.
Otherwise, by (INF-CASE-FST), we have

$$\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t_1',$$

that is, $\Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t_1'$. The result follows as well.

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

(ALG-CASE-BOTH): consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e : t'} \quad \begin{cases} t' \not\leq \neg t & \text{and} \quad \dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_1 : s_1} \\ t' \not\leq t & \text{and} \quad \dfrac{\ldots}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} e_2 : s_2} \end{cases}}{\Delta \,⨟\, \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}$$

By induction, we have

$$\begin{array}{l}\exists t_0'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t_0' \text{ and } t_0' \sqsubseteq_\Delta t',\\ \exists t_1'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_1) : t_1' \text{ and } t_1' \sqsubseteq_\Delta s_1,\\ \exists t_2'.\ \Delta \,⨟\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e_2) : t_2' \text{ and } t_2' \sqsubseteq_\Delta s_2.\end{array}$$

If $t'_0 \sqsubseteq_\Delta \mathbb{0}$, then by (INF-CASE-NONE), we get

$$\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0},$$

that is, $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_\Delta s_1 \vee s_2$.
If $t'_0 \sqsubseteq_\Delta t$, then by (INF-CASE-FST), we get

$$\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t'_1,$$

that is, $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1$. Moreover, it is clear that $t'_1 \sqsubseteq_\Delta s_1 \vee s_2$, the result follows as well. Similarly for $t'_0 \sqsubseteq_\Delta \neg t$.
Otherwise, by (INF-CASE-BOTH), we have

$$\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t'_1 \vee t'_2,$$

that is, $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1 \vee t'_2$. Using α-conversion, we can assume that the polymorphic type variables of t'_1 and t'_2 (and of e_1 and e_2) are distinct, i.e., $(\mathsf{var}(t'_1) \setminus \Delta) \cap (\mathsf{var}(t'_1) \setminus \Delta) = \emptyset$. Then applying Lemma B.5, we have $t'_1 \vee t'_2 \sqsubseteq_\Delta t_1 \vee t_2$.

(ALG-INST): consider the derivation

$$\dfrac{\dfrac{\cdots}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e : t} \qquad \forall j \in J.\ \sigma_j \sharp \Delta \qquad |J| > 0}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_\Delta t.$$

Since $\mathit{erase}(e[\sigma_j]_{j\in J}) = \mathit{erase}(e)$, we have $\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \mathit{erase}(e[\sigma_j]_{j\in J}) : t'$. As $\bigwedge_{k\in K} t'\sigma_k \leq t$, we have $\bigwedge_{j\in J}(\bigwedge_{k\in K} t'\sigma_k)\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$, that is $\bigwedge_{k\in K, j\in J} t'(\sigma_j \circ \sigma_k) \leq \bigwedge_{j\in J} t\sigma_j$. Moreover, it is clear that $\sigma_j \circ \sigma_k \sharp \Delta$. Therefore, we get $t' \sqsubseteq_\Delta \bigwedge_{j\in J} t\sigma_j$.

□

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for $\sqsubseteq_\Delta$, $\amalg^i_\Delta(t)$ and $t \bullet_\Delta s$. The problem of deciding them is tackled in Section C.2.

Finally, notice that we did not give any reduction semantics for the implicitly typed calculus. The reason is that its semantics is defined in terms of the semantics of the explicitly-typed calculus: the relabeling at run-time is an essential feature —independently from the fact that we started from an explicitly typed expression or not— and we cannot avoid it. The (big-step) semantics for a is then given in expressions of $\mathit{erase}^{-1}(a)$: if an expression in $\mathit{erase}^{-1}(a)$ reduces to v, so does a. As we see the result of computing an implicitly-typed expression is a value of the explicitly typed calculus (so λ-abstractions may contain non-empty decorations) and this is unavoidable since it may be the result of a partial application. Also notice that the semantics is not deterministic since different expressions in $\mathit{erase}^{-1}(a)$ may yield different results. However this may happen only in one particular case, namely, when an occurrence of a polymorphic function flows into a type-case and its type is tested. For instance the application $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Bool}} f.f{\in}\mathtt{Bool}{\to}\mathtt{Bool}\,?\,\mathtt{true} : \mathtt{false})(\lambda^{\alpha\to\alpha}x.x)$ results into `true` or `false` according to whether the polymorphic identity at the argument is instantiated by $[\{{}^{\mathtt{Int}}/_{\alpha}\}]$ or by $[\{{}^{\mathtt{Int}}/_{\alpha}\}, \{{}^{\mathtt{Bool}}/_{\alpha}\}]$. Once more this is unavoidable in a calculus that can dynamically test the types of polymorphic functions that admit several sound instantiations.

B.3 A More Tractable Type Inference System

With the rules of Figure 5, when type-checking an implicitly-typed expression, we have to compute sets of type substitutions for projections, applications, abstractions and type cases. Because type substitutions inference is a costly operation, we would like to perform it as less as possible. To this end, we give in this section a restricted version of the inference system, which is not complete but still sound and powerful enough to be used in practice.

First, we want to simplify the type inference rule for projections:

$$\dfrac{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} a : t \quad u \in \amalg^i_\Delta(t)}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : u}$$

where $\amalg^i_\Delta(t) = \{u \mid [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1}\times\mathbb{1},\ u = \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)\}$. Instead of picking any type in $\amalg^i_\Delta(t)$, we would like to simply project t, i.e., assign the type $\boldsymbol{\pi}_i(t)$ to $\pi_i(a)$. By doing so, we lose completeness on pair types that contain top-level variables. For example, if $t = (\mathtt{Int}\times\mathtt{Int}) \wedge \alpha$, then $\mathtt{Int} \wedge \mathtt{Bool} \in \amalg^i_\Delta(t)$ (because α can be instantiated with $(\mathtt{Bool}\times\mathtt{Bool})$), but $\boldsymbol{\pi}_t(t) = \mathtt{Int}$. We also lose typability if t is not a pair type, but can be instantiated in a pair type. For example, the type of $(\lambda^{\alpha\to(\alpha\vee((\beta\to\beta)\setminus(\mathtt{Int}\to\mathtt{Int})))}x.x)(42, 3)$ is $(\mathtt{Int}\times\mathtt{Int}) \vee ((\beta\to\beta)\setminus(\mathtt{Int}\to\mathtt{Int}))$, which is not a pair type, but can be instantiated in $(\mathtt{Int}\times\mathtt{Int})$ by taking $\beta = \mathtt{Int}$. We believe these kinds of types will not be written by programmers, and it is safe to use the following projection rule in practice.

$$\dfrac{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} a : t \qquad t \leq \mathbb{1}\times\mathbb{1}}{\Delta \,{}_{9}^{\circ}\, \Gamma \vdash_{\mathscr{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)}\,(\text{INF-PROJ'})$$

We now look at the type inference rules for the type case $a{\in}t\,?\,a_1 : a_2$. The four different rules consider the different possible instantiations that make the type t' inferred for a fit t or not. For the sake of simplicity, we decide not to infer type substitutions for polymorphic arguments of type cases. Indeed, in the expression $(\lambda^{\alpha\to\alpha}x.x) \in \texttt{Int} \to \texttt{Int}\ ?\ \texttt{true} : \texttt{false}$, we assume the programmer wants to do a type case on the polymorphic identity, and not on one of its instance (otherwise, he would have written the instantiated interface directly), so we do not try to instantiate it. And in any case there is no real reason for which the inference system should choose to instantiate the identity by $\texttt{Int}{\to}\texttt{Int}$ (and thus make the test succeed) rather than $\texttt{Bool}{\to}\texttt{Bool}$ (and thus make the test fail). If we decide not to infer types for polymorphic arguments of type-case expression, then since $\alpha \to \alpha$ is not a subtype of $\texttt{Int} \to \texttt{Int}$ (we have $\alpha \to \alpha \sqsubseteq_{\emptyset} \texttt{Int} \to \texttt{Int}$ but $\alpha \to \alpha \not\leq \texttt{Int} \to \texttt{Int}$) the expression evaluates to $\texttt{false}$. With this choice, we can merge the different inference rules into the following one.

$$\frac{\begin{array}{c}\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a : t' \quad t_1 = t' \wedge t \quad t_2 = t' \wedge \neg t \\ t_i \not\simeq \mathbb{0} \ \Rightarrow\ \Delta \,\S\, \Gamma \vdash_{\mathscr{I}} a_i : s_i\end{array}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \bigvee_{t_i \not\simeq \mathbb{0}} s_i}\ (\text{INF-CASE'})$$

Finally, consider the inference rule for abstractions:

$$\frac{\forall i \in I.\ \begin{cases} \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,\S\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta\cup\mathsf{var}(\bigwedge_{i\in I} t_i\to s_i)} s_i \end{cases}}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}$$

We verify that the abstraction can be typed with each arrow type $t_i \to s_i$ in the interface. Meanwhile, we also infer a set of type substitutions to tally the type s'_i we infer for the body expression with s_i. In practice, similarly, we expect that the abstraction is well-typed only if the type s'_i we infer for the body expression is a subtype of s_i. For example, the expression

$$\lambda^{\texttt{Bool}\to(\texttt{Int}\to\texttt{Int})} x.x{\in}\texttt{true}\,?\,(\lambda^{\alpha\to\alpha}y.y) : (\lambda^{\alpha\to\alpha}y.(\lambda^{\alpha\to\alpha}z.z)y)$$

is not well-typed while

$$\lambda^{\texttt{Bool}\to(\alpha\to\alpha)} x.x{\in}\texttt{true}\,?\,(\lambda^{\alpha\to\alpha}y.y) : (\lambda^{\alpha\to\alpha}y.(\lambda^{\alpha\to\alpha}z.z)y)$$

is well-typed. So we use the following restricted rule for abstractions instead:

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \,\S\, \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i \text{ and } s'_i \leq s_i}{\Delta \,\S\, \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}\ (\text{INF-ABSTR'})$$

In conclusion, we restrict the inference of type substitutions to applications. We give in Figure 6 the inference rules of the system which respects the above restrictions. With these new rules, the system remains sound, but it is not complete.

Theorem B.15. *If $\Gamma \vdash_{\mathscr{I}} a : t$, then there exists an expression $e \in \mathscr{E}_0$ such that erase$(e) = a$ and $\Gamma \vdash_{\mathscr{A}} e : t$.*

Proof. Similar to the proof of Theorem B.13. □

C. Type Tallying

Given two types t and s, the goal of this section is to find pairs of sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $\bigwedge_{j\in J} s\sigma_j \leq \bigvee_{i\in I} t\sigma_i$. Assuming that the cardinalities of I and J are known, then this problem can be reduced to a *type tallying* problem, that we define and solve first. We then explain how we can reduce the original problem to the type tallying problem, and provide a semi-algorithm for the original problem. Finally, we give some heuristics to establish upper bounds (which depend on t and s) for the cardinalities of I and J.

C.1 Type Tallying Problem

Given a finite set C of pairs of types and a finite set Δ of type variables, the tallying problem for C and Δ consists in verifying whether there exists a substitution σ such that $\sigma \sharp \Delta$ and for all $(s,t) \in C$, $s\sigma \leq t\sigma$ holds. In this section we denote constraints as triples. The notation is different from the one used in Section 3 in that it also specifies the symbol of the relation. So a pair of types $(s,t) \in C$ corresponds to the constraint $(s, \leq, t)$:

Definition C.1 (Constraints). *A* constraint (t,c,s) *is a triple belonging to $\mathscr{T} \times \{\leq,\geq\} \times \mathscr{T}$. Let $\mathscr{C}$ denote the set of all constraints. Given a constraint-set $C \subseteq \mathscr{C}$, the* set of type variables *occurring in C is defined as*

$$\mathit{var}(C) = \bigcup_{(t,c,s)\in C} \mathit{var}(t) \cup \mathit{var}(s)$$

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} c : b_c}\,(\textsc{Inf-const}) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} x : \Gamma(x)}\,(\textsc{Inf-var})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_2 : t_2}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} (a_1, a_2) : t_1 \times t_2}\,(\textsc{Inf-pair}) \qquad \frac{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)}\,(\textsc{Inf-proj'})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 : t \qquad \Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_2 : s \qquad u \in t \bullet_{\Delta} s}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 a_2 : u}\,(\textsc{Inf-appl})$$

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathscr{I}} a : s'_i \text{ and } s'_i \leq s_i}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.a : \bigwedge_{i \in I} t_i \to s_i}\,(\textsc{Inf-abstr'})$$

$$\frac{\begin{array}{l}\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a : t' \quad t_1 = t' \wedge t \quad t_2 = t' \wedge \neg t \\ t_i \not\simeq \mathbb{0} \ \Rightarrow\ \Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_i : s_i\end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} (a{\in}t\,?\,a_1 : a_2) : \bigvee_{t_i \not\simeq \mathbb{0}} s_i}\,(\textsc{Inf-case'})$$

Figure 6. Restricted type-substitution inference rules

Definition C.2 (**Normalized constraint).** *A constraint (t, c, s) is said to be* normalized *if t is a type variable. A constraint-set $C \subseteq \mathscr{C}$ is said to be normalized if every constraint $(t, c, s) \in C$ is normalized. Given a normalized constraint-set C, its domain is defined as $\mathsf{dom}(C) = \{\alpha \mid \exists c, s.\ (\alpha, c, s) \in C\}$.*

Definition C.3 (**Constraint solution).** *Let $C \subseteq \mathscr{C}$ be a constraint-set. A solution to C is a substitution σ such that*

$$\forall (t, \leq, s) \in C\ .\ t\sigma\ \leq\ s\sigma \textit{ holds} \quad \textit{and} \quad \forall (t, \geq, s) \in C\ .\ s\sigma\ \leq\ t\sigma \textit{ holds.}$$

If σ is a solution to C, we write $\sigma \Vdash C$.

Definition C.4. *Given two sets of constraint-sets $\mathscr{S}_1, \mathscr{S}_2 \subseteq \mathscr{P}(\mathscr{C})$, we define their* union *as*

$$\mathscr{S}_1 \sqcup \mathscr{S}_2 = \mathscr{S}_1 \cup \mathscr{S}_2$$

and their intersection as

$$\mathscr{S}_1 \sqcap \mathscr{S}_2 = \{C_1 \cup C_2 \mid C_1 \in \mathscr{S}_1, C_2 \in \mathscr{S}_2\}$$

Given a constraint-set C, the constraint solving algorithm produces the set of all the solutions of C by following the algorithm given in Section 3.2.1. Let us examine each step of the algorithm on some examples.

Step 1: constraint normalization.
Because normalized constraints are easier to solve than regular ones, we first turn each constraint into an equivalent set of normalized constraint-sets according to the decomposition rules in [4]. For example, the constraint $c_1 = (\alpha \times \alpha) \leq ((\mathtt{Int} \times \mathbb{1}) \times (\mathbb{1} \times \mathtt{Int}))$ can be normalized into the set $\mathscr{S}_1 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\mathtt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \mathtt{Int}))\}\}$. Another example is the constraint $c_2 = ((\beta \times \beta) \to (\mathtt{Int} \times \mathtt{Int}), \leq, \alpha \to \alpha)$, which is equivalent to the following set of normalized constraint-sets $\mathscr{S}_2 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\mathtt{Int} \times \mathtt{Int}))\}\}$. Then we join all the sets of constraint-sets by (constraint-set) intersections, yielding the normalization of the original constraint-set. For instance, the normalization $\mathscr{S}$ of $\{c_1, c_2\}$ is $\mathscr{S}_1 \sqcap \mathscr{S}_2$. It is easy to see that the constraint-set $C_1 = \{(\alpha, \leq, (\mathtt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \mathtt{Int})), (\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\mathtt{Int} \times \mathtt{Int}))\}$ is in $\mathscr{S}$ (see Definition C.4).

Step 2: constraint merging.

Step 2.1: merge the constraints with a same type variable.
In each constraint-set of the normalization of the original constraint-set, there may be several constraints of the form $(\alpha, \geq, t_i)$ (resp. $(\alpha, \leq, t_i)$), which give different lower bounds (resp. upper bounds) for α. We merge all these constraints into one using unions (resp. intersections). For example, the constraint-set C_1 of the previous step can be merged as $C_2 = \{(\alpha, \leq, (\mathtt{Int} \times \mathbb{1}) \wedge (\mathbb{1} \times \mathtt{Int}) \wedge (\beta \times \beta)), (\alpha, \geq, (\mathtt{Int} \times \mathtt{Int}))\}$, which is equivalent to $\{(\alpha, \leq, (\mathtt{Int} \wedge \beta \times \mathtt{Int} \wedge \beta)), (\alpha, \geq, (\mathtt{Int} \times \mathtt{Int}))\}$.

Step 2.2: saturate the lower and upper bounds of a same type variable.
If a type variable has both a lower bound s and an upper bound t in a constraint-set, then the solutions we are looking for must satisfy the constraint $(s, \leq, t)$ as well. Therefore, we have to saturate the

constraint-set with $(s, \leq, t)$, which has to be normalized, merged, and saturated itself first. Take C_2 for example. We have to saturate C_2 with $((\texttt{Int}\times\texttt{Int}), \leq, (\texttt{Int}\wedge\beta\times\texttt{Int}\wedge\beta))$, whose normalization is $\{\{(\beta, \geq, \texttt{Int})\}\}$. Thus, the saturation of C_2 is $\{C_2\} \sqcap \{\{(\beta, \geq, \texttt{Int})\}\}$, which contains only one constraint-set $C_3 = \{(\alpha, \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int})), (\beta, \geq, \texttt{Int})\}$.

Step 3: constraint solving.

Step 3.1: transform each constraint-set into an equation system.
To transform constraints into equations, we use the property that some set of constraints is satisfied for all assignments of α included between s and t if and only if the same set in which we replace α by $(s \vee \alpha') \wedge t$[10] is satisfied for all possible assignments of α' (with α' fresh). Of course such a transformation works only if $s \leq t$, but remember that we "checked" that this holds at the moment of the saturation. By performing this replacement for each variable we obtain a system of equations. For example, the constraint set C_3 is equivalent to the following equation system E:

$$\begin{aligned} \alpha &= \quad ((\texttt{Int} \times \texttt{Int}) \vee \alpha') \wedge (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta) \\ \beta &= \quad \texttt{Int} \vee \beta' \end{aligned}$$

where α', β' are fresh type variables.

Step 3.2: extract a substitution from each equation system.
Finally, using the Courcelle's work on infinite trees [5], we solve each equation system, which gives us a substitution which is a solution of the original constraint-set. For example, we can solve the equation system E, yielding the type-substitution $\{{}^{(\texttt{Int} \times \texttt{Int})}\!/\!_{\alpha}, {}^{\texttt{Int} \vee \beta'}\!/\!_{\beta}\}$, which is a solution of C_3 and thus of $\{c_1, c_2\}$.

In the following subsections we study in details each step of the algorithm.

C.1.1 Constraint Normalization

The type tallying problem is quite similar to the subtyping problem presented in [4]. We therefore reuse most of the technology developed in [4] such as, for example, the transformation of the subtyping problem into an emptiness decision problem, the elimination of top-level constructors, and so on. One of the main differences is that we do not want to eliminate top-level type variables from constraints, but, rather, we want to isolate them to build sets of normalized constraints (from which we then construct sets of substitutions).

In general, normalizing a constraint generates a set of constraints. For example, $(\alpha\vee\beta, \geq, \mathbb{0})$ holds if and only if $(\alpha, \geq, \mathbb{0})$ or $(\beta, \geq, \mathbb{0})$ holds; therefore the constraint $(\alpha \vee \beta, \geq, \mathbb{0})$ is equivalent to the normalized constraint-set $\{(\alpha, \geq, \mathbb{0}), (\beta, \geq, \mathbb{0})\}$. Consequently, the normalization of a constraint-set C yields a set $\mathscr{S}$ of normalized constraint-sets.

Several normalized sets may be suitable replacements for a given constraint; for example, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ and $\{(\alpha, \leq, \beta \vee t_1), (\alpha, \geq, \beta \setminus t_2)\}$ are clearly equivalent normalized sets. However, the equation systems generated by the algorithm for these two sets are completely different, and different equation systems yield different substitutions (see Section C.1.3 for more details). Concretely, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ generates the equation system $\{\alpha = \alpha' \wedge (\beta \vee t_1), \beta = \beta' \wedge (\alpha \vee t_2)\}$, which in turn gives the substitution σ_1 such that

$$\begin{aligned} \sigma_1(\alpha) &= \mu x.\ ((\alpha' \wedge \beta' \wedge x) \vee (\alpha' \wedge \beta' \wedge t_2) \vee (\alpha' \wedge t_1)) \\ \sigma_1(\beta) &= \mu x.\ ((\beta' \wedge \alpha' \wedge x) \vee (\beta' \wedge \alpha' \wedge t_1) \vee (\beta' \wedge t_2)) \end{aligned}$$

where α' and β' are fresh type variables and we used the μ notation to denote regular recursive types. These recursive types are not valid in our calculus, because x does not occur under a type constructor (this means that the unfolding of the type does not satisfy the property that every infinite branch contains infinitely many occurrences of type constructors). In contrast, the equation system built from $\{(\alpha, \leq, \beta\vee t_1), (\alpha, \geq, \beta\setminus t_2)\}$ is $\alpha = ((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)$, and the corresponding substitution is $\sigma_2 = \{{}^{((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)}\!/\!_{\alpha}\}$, which is valid since it maps the type variable α into a well-formed type. Ill-formed recursive types are generated when there exists a chain $\alpha_0 = \alpha_1\ B_1\ t_1, \ldots, \alpha_i = \alpha_{i+1}\ B_{i+1}\ t_{i+1}, \ldots, \alpha_n = \alpha_0\ B_{n+1}\ t_{n+1}$ (where $B_i \in \{\wedge, \vee\}$ for all i, and $n \geq 0$) in the equation system built from the normalized constraint-set. This chain implies the equation $\alpha_0 = \alpha_0\ B\ t'$ for some $B \in \{\wedge, \vee\}$ and t', and the corresponding solution for α_0 will be an ill-formed recursive type. To avoid this issue, we give an arbitrary ordering on type variables occurring in the constraint-set C such that different type variables have different orders. Then we always select the normalized constraint (α, c, t) such that the order of α is smaller than all the orders of the top-level type variables in t. As a result, the transformed equation system does not contain any problematic chain like the one above.

Definition C.5 (Ordering). *Let V be a set of type variables. An* ordering *O on V is an injective map from V to $\mathbb{N}$.*

We formalize normalization as a judgement $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, which states that under the environment Σ (which, informally, contains the types that have already been processed at this point), C is normalized to $\mathscr{S}$. The judgement is derived according the rules of Figure 7. These rules describe the same algorithm

[10] Or by $s \vee (\alpha' \wedge t)$.

as the function norm given in Figure 3 (*ie*, $\Sigma \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathsf{norm}(t, \Sigma)$ is provable in the system of Figure 7) but extended to handle also product types. We just switched to a deduction systems since it eases the formal treatment.

$$\frac{}{\Sigma \vdash_{\mathscr{N}} \emptyset \rightsquigarrow \{\emptyset\}}\,(\textsc{Nempty}) \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(t_i\ c_i\ t'_i)\} \rightsquigarrow S_i}{\Sigma \vdash_{\mathscr{N}} \{(t_i\ c_i\ t'_i) \mid i \in I\} \rightsquigarrow \bigsqcap_{i \in I} S_i}\,(\textsc{Njoin})$$

$$\frac{\Sigma \vdash_{\mathscr{N}} \{(t \leq t')\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(t' \geq t)\} \rightsquigarrow \mathscr{S}}\,(\textsc{Nsym}) \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(t \wedge \neg t' \leq \mathbb{0})\} \rightsquigarrow \mathscr{S} \qquad t' \neq \mathbb{0}}{\Sigma \vdash_{\mathscr{N}} \{(t \leq t')\} \rightsquigarrow \mathscr{S}}\,(\textsc{Nzero})$$

$$\frac{\Sigma \vdash_{\mathscr{N}} \{(\mathsf{dnf}(t) \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(t \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\,(\textsc{Ndnf}) \qquad \frac{\Sigma \vdash_{\mathscr{N}} \{(\tau_i \leq \mathbb{0}) \mid i \in I\} \rightsquigarrow \mathscr{S}}{\Sigma \vdash_{\mathscr{N}} \{(\bigvee_{i \in I} \tau_i \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\,(\textsc{Nunion})$$

$$\frac{\mathsf{tlv}(\tau_0) = \emptyset \qquad \alpha' \lightning_O P \cup N \qquad \mathscr{S} = \begin{cases} \{\{(\alpha', \leq, \neg t_{\alpha'})\}\} & \alpha' \in P \\ \{\{(\alpha', \geq, t_{\alpha'})\}\} & \alpha' \in N \end{cases}}{\Sigma \vdash_{\mathscr{N}} \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\,(\textsc{Ntlv})$$

$$\frac{\tau_0 \in \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}\,(\textsc{Nhyp}) \qquad \frac{\Sigma \cup \{\tau_0\} \vdash^*_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S} \qquad \tau_0 \notin \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathscr{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathscr{S}}\,(\textsc{Nassum})$$

$$\frac{\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}\,(\textsc{Nbasic-T}) \qquad \frac{\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \not\leq \mathbb{0}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \emptyset}\,(\textsc{Nbasic-F})$$

$$\frac{\forall N' \subseteq N. \begin{cases} \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i \in P} t^1_i \wedge \bigwedge_{j \in N'} \neg t^1_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^1_{N'} \\ \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i \in P} t^2_i \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^2_{N'} \end{cases}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i \in P} (t^1_i \times t^2_i) \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\} \rightsquigarrow \bigsqcap_{N' \subseteq N} (\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})}\,(\textsc{Nprod})$$

$$\frac{\exists j \in N\ \forall P' \subseteq P. \begin{cases} \Sigma \vdash_{\mathscr{N}} \{t^1_j \wedge \bigwedge_{i \in P'} \neg t^1_i \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^1_{P'} \\ \begin{cases} \Sigma \vdash_{\mathscr{N}} \{\bigwedge_{i \in P \setminus P'} t^2_i \wedge \neg t^2_j \leq \mathbb{0}\} \rightsquigarrow \mathscr{S}^2_{P'} & P' \neq P \\ \mathscr{S}^2_{N'} = \emptyset & \text{otherwise} \end{cases} \end{cases}}{\Sigma \vdash^*_{\mathscr{N}} \{(\bigwedge_{i \in P} (t^1_i \to t^2_i) \wedge \bigwedge_{j \in N} \neg(t^1_j \to t^2_j) \leq \mathbb{0})\} \rightsquigarrow \bigsqcup_{j \in N} \bigsqcap_{P' \subseteq P} (\mathscr{S}^1_{P'} \sqcup \mathscr{S}^2_{P'})}\,(\textsc{Narrow})$$

where t and t with scripts are types, c_i belongs to $\{\leq, \geq\}$, τ_0 and τ_i are single normal forms, $\alpha \lightning_O P \cup N$ denotes α has the smallest order in $P \cup N$ under the ordering O, and $t_{\alpha'}$ is the type obtained from $\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0$ by eliminating α'.

Figure 7. Normalization rules

If the constraint-set is empty, then clearly any substitution is a solution, and, the result of the normalization is simply the singleton containing the empty set (rule (Nempty)). Otherwise, each constraint is normalized separately, and the normalization of the constraint-set is the intersection of the normalizations of each constraint (rule (Njoin)). By using rules (Nsym), (Nzero), and (Ndnf) repeatedly, we transform any constraint into the constraint of the form $(\tau, \leq, \mathbb{0})$ where τ is disjunctive normal form: the first rule reverses $(t', \geq, t)$ into $(t, \leq, t')$, the second rule moves the type t' from the right of $\leq$ to the left, yielding $(t \wedge \neg t', \leq, \mathbb{0})$, and finally the last rule puts $t \wedge \neg t'$ in disjunctive normal form. Such a type τ is the type to be normalized. If τ is a union of single normal forms, the rule (Nunion) splits the union of single normal forms into constraints featuring each of the single normal forms. Then the results of each constraint normalization are joined by the rule (Njoin).

The following rules handle constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. If there are some top-level type variables, the rule (Ntlv) generates a normalized constraint for the top-level type variable whose order is the smallest. Otherwise, there are no top-level type variables. If τ has already been normalized (i.e., it belongs to Σ), then it is not processed again (rule (Nhyp)). Otherwise, we memoize it and then process it using the predicate for single normal forms $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ (rule (Nassum)). Note that switching from $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ to $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ prevents the incorrect use of (Nhyp) just after (Nassum), which would wrongly say that any type is normalized without doing any computation.

Finally, the last four rules state how to normalize constraints of the form $(\tau, \leq, \mathbb{0})$ where τ is a single normal form and contains no top-level type variables. Thereby τ should be an intersection of atoms with the same constructor. If τ is an intersection of basic types, normalizing is equivalent to checking whether τ is empty or not: if it is (rule (NBASIC-T)), we return the singleton containing the empty set (any substitution is a solution), otherwise there is no solution and we return the empty set (rule (NBASIC-F)). When τ is an intersection of products, the rule (NPROD) decomposes τ into several candidate types (following Lemma 3.11 in [4]), which are to be further normalized. The case when τ is an intersection of arrows (rule (NARROW)) is treated similarly. Note that, in the last two rules, we switch from $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ back to $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ in the premises to ensure termination.

If $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, then $\mathscr{S}$ is the result of the normalization of C. We now prove soundness, completeness, and termination of the constraint normalization algorithm.

To prove soundness, we use a family of subtyping relations $\leq_n$ that layer $\leq$[11] (*i.e.*, such that $\bigcup_{n\in\mathscr{N}} \leq_n = \leq$) and a family of satisfaction predicates $\Vdash_n$ that layer $\Vdash$ (*i.e.*, such that $\bigcup_{n\in\mathscr{N}} \Vdash_n = \Vdash$), which are defined as follows.

Definition C.6. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support $([\![\_]\!], \mathscr{D})$. We define the family $(\leq_n)_{n\in\mathscr{N}}$ of subtyping relations as*

$$t \leq_n s \overset{\text{def}}{\iff} \forall\eta.\ [\![t]\!]_n\eta \subseteq [\![s]\!]_n\eta$$

where $[\![\_]\!]_n$ is the rank n interpretation of a type, defined as

$$[\![t]\!]_n\eta = \{d \in [\![t]\!]\eta \mid \mathsf{height}(d) \leq n\}$$

and $\mathsf{height}(d)$ *is the height of an element d in $\mathscr{D}$, defined as*

$$\begin{array}{rl} \mathsf{height}(c) & = 1 \\ \mathsf{height}((d, d')) & = max(\mathsf{height}(d), \mathsf{height}(d')) + 1 \\ \mathsf{height}(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}) & = \begin{cases} 1 & n = 0 \\ max(\mathsf{height}(d_i), \mathsf{height}(d'_i), \ldots) + 1 & n > 0 \end{cases} \end{array}$$

Lemma C.7. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) $t \leq_0 s$ *for all* $t, s \in \mathscr{T}$.
(2) $t \leq s \iff \forall n.\ t \leq_n s$.
(3)

$$\bigwedge_{i\in I}(t_i \times s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \times s_j) \iff \forall J' \subseteq J\ .\ \begin{cases} \bigwedge_{i\in I} t_i \leq_n \bigvee_{j\in J'} t_j \\ \vee \\ \bigwedge_{i\in I} s_i \leq_n \bigvee_{j\in J\setminus J'} s_j \end{cases}$$

(4)

$$\bigwedge_{i\in I}(t_i \to s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \to s_j) \iff \exists j_0{\in}J\,.\,\forall I'{\subseteq}I\,.\ \begin{cases} t_{j_0} \leq_n \bigvee_{i\in I'} t_i \\ \vee \\ \begin{cases} I \neq I' \\ \wedge \\ \bigwedge_{i\in I\setminus I'} s_i \leq_n s_{j_0} \end{cases} \end{cases}$$

Proof. (1) straightforward.
(2) straightforward.
(3) the result follows by Lemma 3.11 in [4] and Definition C.6.
(4) the result follows by Lemma 3.12 in [4] and Definition C.6.

□

Definition C.8. *Given a constraint-set C and a type substitution σ, we define the rank n satisfaction predicate $\Vdash_n$ as*

$$\sigma \Vdash_n C \overset{\text{def}}{\iff} \forall(t, \leq, s) \in C.\ t \leq_n s \text{ and } \forall(t, \geq, s) \in C.\ s \leq_n t$$

Lemma C.9. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) $\sigma \Vdash_0 C$ *for all σ and C.*
(2) $\sigma \Vdash C \iff \forall n.\ \sigma \Vdash_n C$.

Proof. Consequence of Lemma C.7. □

Given a set Σ of types, we write $C(\Sigma)$ for the constraint-set $\{(t, \leq, \mathbb{0}) \mid t \in \Sigma\}$.

[11] See [4] for the definitions of the notions of models, interpretations, and assignments

Lemma C.10 (Soundness). *Let C be a constraint-set. If $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, then for all normalized constraint-set $C' \in \mathscr{S}$ and all substitution σ, we have $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all $C' \in \mathscr{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_n C$.
(2) Assume $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all $C' \in \mathscr{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_{n+1} C$.

Before proving these statements, we explain how the first property implies the lemma. Suppose $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$, $C' \in \mathscr{S}$ and $\sigma \Vdash C'$. It is easy to check that $\sigma \Vdash_n C(\emptyset)$ holds for all n. From $\sigma \Vdash C'$, we deduce $\sigma \Vdash_n C'$ for all n (by Lemma C.9). By Property (1), we have $\sigma \Vdash_n C$ for all n, and we have then the required result by Lemma C.9.

We prove these two properties simultaneously by induction on the derivations of $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ and $\Sigma \vdash^*_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$.

(NEMPTY): straightforward.

(NJOIN): according to Definition C.4, if there exists $C_i \in \mathscr{S}_i$ such that $C_i = \emptyset$, then $\bigsqcap_{i \in I} \mathscr{S}_i = \emptyset$, and the result follows immediately. Otherwise, we have $C' = \bigcup_{i \in I} C_i$, where $C_i \in \mathscr{S}_i$. As $\sigma \Vdash_n C'$, then clearly $\sigma \Vdash_n C_i$. By induction, we have $\sigma \Vdash_n \{(t_i\ c_i\ t'_i)\}$. Therefore, we get $\sigma \Vdash_n \{(t_i\ c_i\ t'_i) \mid i \in I\}$.

(NSYM): by induction, we have $\sigma \Vdash_n \{(t \leq t')\}$. Then clearly $\sigma \Vdash_n \{(t' \geq t)\}$.

(NZERO): by induction, we have $\sigma \Vdash_n \{(t \wedge \neg t' \leq \mathbb{0})\}$. According to set-theory, we have $\sigma \Vdash_n \{(t \leq t')\}$.

(NDNF): similar to the case of (NZERO).

(NUNION): similar to the case of (NZERO).

(NTLV): assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then we have $C' = (\alpha', \leq, \neg t_{\alpha'})$. From $\sigma \Vdash_n C'$, we deduce $\sigma(\alpha') \leq_n \neg t_{\alpha'}\sigma$. Intersecting both sides of the inequality by the same type, we obtain $\sigma(\alpha') \wedge t_{\alpha'}\sigma \leq_n \mathbb{0}$, that is, $\sigma \Vdash_n \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

(NHYP): since we have $\tau_0 \in \Sigma$ and $\sigma \Vdash_n C(\Sigma)$, then $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0})\}$ holds.

(NASSUM): if $n = 0$, then $\sigma \Vdash_0 \{(\tau_0 \leq \mathbb{0})\}$ holds. Suppose $n > 0$. From $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_k C'$, it is easy to prove that $\sigma \Vdash_k C(\Sigma)$ (*) and $\sigma \Vdash_k C'$ (**) hold for all $0 \leq k \leq n$. We now prove that $\sigma \Vdash_k \{(\tau_0 \leq \mathbb{0}\})$ (***) holds for all $1 \leq k \leq n$. By definition of $\Vdash_0$, we have $\sigma \Vdash_0 C(\Sigma \cup \{\tau_0\})$ and $\sigma \Vdash_0 C'$. Consequently, by the induction hypothesis (item (2)), we have $\sigma \Vdash_1 \{\tau_0 \leq \mathbb{0}\}$. From this and (*), we deduce $\sigma \Vdash_1 C(\Sigma \cup \{\tau_0\})$. Because we also have $\sigma \Vdash_1 C'$ (by (**)), we can use the induction hypothesis (item (2)) again to deduce $\sigma \Vdash_2 \{(\tau_0 \leq \mathbb{0}\})$. Hence, we can prove (***) by induction on $1 \leq k \leq n$. In particular, we have $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0}\})$, which is the required result.

(NBASIC): straightforward.

(NPROD): If $\bigsqcap_{N' \subseteq N} (\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})$ is $\emptyset$, then the result follows straightforwardly. Otherwise, we have $C' = \bigcup_{N' \subseteq N} C_{N'}$, where $C_{N'} \in (\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})$. Since $\sigma \Vdash_n C'$, we have $\sigma \Vdash_n C_{N'}$ for all subset $N' \subseteq N$. Moreover, following Definition C.4, either $C_{N'} \in \mathscr{S}^1_{N'}$ or $C_{N'} \in \mathscr{S}^2_{N'}$. By induction, we have either $\sigma \Vdash_n \{\bigwedge_{i \in P} t^1_i \wedge \bigwedge_{j \in N'} \neg t^1_j \leq \mathbb{0}\}$ or $\sigma \Vdash_n \{\bigwedge_{i \in P} t^2_i \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j \leq \mathbb{0}\}$. That is, for all subset $N' \subseteq N$, we have

$$\bigwedge_{i \in P} t^1_i\sigma \wedge \bigwedge_{j \in N'} \neg t^1_j\sigma \leq_n \mathbb{0} \text{ or } \bigwedge_{i \in P} t^2_i\sigma \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j\sigma \leq_n \mathbb{0}$$

Applying Lemma C.7, we have

$$\bigwedge_{i \in P} (t^1_i \times t^2_i)\sigma \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j)\sigma \leq_{n+1} \mathbb{0}$$

Thus, $\sigma \Vdash_{n+1} \{(\bigwedge_{i \in P}(t^1_i \times t^2_i) \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\}$.

(NARROW): similar to the case of (NPROD).

□

Given a normalized constraint-set C and a set X of type variables, we define the restriction $C|_X$ of C by X to be $\{(\alpha, c, t) \in C \mid \alpha \in X\}$.

Lemma C.11. *Let t be a type and $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathscr{S}$. Then for all normalized constraint-set $C \in \mathscr{S}$, all substitution σ and all n, if $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$ and $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$.*

Proof. By applying the rules (NDNF) and (NUNION), the constraint-set $\{(t, \leq, \mathbb{0})\}$ is normalized into a new constraint-set C', consisting of the constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. That is, $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \{C'\}$. Let $C'_1 = \{(\tau, \leq, \mathbb{0}) \in C' \mid \mathsf{tlv}(\tau) \neq \emptyset\}$ and $C'_2 = C' \setminus C'_1$. It is easy to deduce that all the constraints in $C \setminus C|_{\mathsf{tlv}(t)}$ are generated from C'_2 and must pass at least one instance of $\vdash^*_{\mathscr{N}}$ (*i.e.*, being decomposed at least once). Since $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then according to the statement (2) in the proof of Lemma C.10, we have $\sigma \Vdash_n C'_2$. Moreover, from $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$, we have $\sigma \Vdash_n C'_1$. Thus, $\sigma \Vdash_n C'$ and *a fortiori* $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$. □

Lemma C.12 (**Completeness**)**.** *Let C be a constraint-set such that $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.
(2) Assume $\Sigma \vdash^{*}_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.

The result is then a direct consequence of the first item (indeed, we have $\sigma \Vdash C(\emptyset)$ for all σ). We prove the two items simultaneously by induction on the derivations of $\Sigma \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$ and $\Sigma \vdash^{*}_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$.

(NEMPTY)**:** straightforward.
(NJOIN)**:** as $\sigma \Vdash \{(t_i\ c_i\ t'_i) \mid i \in I\}$, we have in particular $\sigma \Vdash \{(t_i\ c_i\ t'_i)\}$ for all i. By induction, there exists $C_i \in \mathscr{S}_i$ such that $\sigma \Vdash C_i$. So $\sigma \Vdash \bigcup_{i\in I} C_i$. Moreover, according to Definition C.4, $\bigcup_{i\in I} C_i$ must be in $\bigsqcap_{i\in I} \mathscr{S}_i$. Therefore, the result follows.
(NSYM)**:** if $\sigma \Vdash \{(t' \geq t)\}$, then $\sigma \Vdash \{(t \leq t')\}$. By induction, the result follows.
(NZERO)**:** since $\sigma \Vdash \{(t \leq t')\}$, we can substract t' from both sides obtain $\sigma \Vdash \{(t \wedge \neg t' \leq \mathbb{0})\}$. By induction, the result follows.
(NDNF)**:** similar to the case of (NZERO).
(NUNION)**:** similar to the case of (NZERO).
(NTLV)**:** assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then according to set-theory, we have $\alpha'\sigma \leq \neg(\bigwedge_{\alpha\in(P\setminus\{\alpha'\})} \alpha \wedge \bigwedge_{\alpha\in N} \neg\alpha \wedge \tau_0)$, that is $\sigma \Vdash \{(\alpha' \leq \neg t_{\alpha'})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.
(NHYP)**:** it is clear that $\sigma \Vdash \emptyset$.
(NASSUM)**:** as $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash \{(\tau_0 \leq \mathbb{0})\}$, we have $\sigma \Vdash C(\Sigma \cup \{\tau_0\})$. By induction, the result follows.
(NBASIC)**:** straightforward.
(NPROD)**:** as

$$\sigma \Vdash \{(\bigwedge_{i\in P}(t^1_i \times t^2_i) \wedge \bigwedge_{j\in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\}$$

we have

$$\bigwedge_{i\in P}(t^1_i \times t^2_i)\sigma \wedge \bigwedge_{j\in N} \neg(t^1_j \times t^2_j)\sigma \leq \mathbb{0}$$

Applying Lemma 3.11 in [4], for all subset $N' \subseteq N$, we have

$$\bigwedge_{i\in P} t^1_i\sigma \wedge \bigwedge_{j\in N'} \neg t^1_j\sigma \leq \mathbb{0} \text{ or } \bigwedge_{i\in P} t^2_i\sigma \wedge \bigwedge_{j\in N\setminus N'} \neg t^2_j\sigma \leq \mathbb{0}$$

that is,

$$\sigma \Vdash \{(\bigwedge_{i\in P} t^1_i \wedge \bigwedge_{j\in N'} \neg t^1_j \leq \mathbb{0})\} \text{ or } \sigma \Vdash \{(\bigwedge_{i\in P} t^2_i \wedge \bigwedge_{j\in N\setminus N'} \neg t^2_j \leq \mathbb{0})\}$$

By induction, either there exists $C^1_{N'} \in \mathscr{S}^1_{N'}$ such that $\sigma \Vdash C^1_{N'}$ or there exists $C^2_{N'} \in \mathscr{S}^2_{N'}$ such that $\sigma \Vdash C^2_{N'}$. According to Definition C.4, we have $C^1_{N'}, C^2_{N'} \in \mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'}$. Thus there exists $C'_{N'} \in \mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'}$ such that $\sigma \Vdash C'_{N'}$. Therefore $\sigma \Vdash \bigcup_{N'\subseteq N} C'_{N'}$. Moreover, according to Definition C.4 again, $\bigcup_{N'\subseteq N} C'_{N'} \in \bigsqcap_{N'\subseteq N}(\mathscr{S}^1_{N'} \sqcup \mathscr{S}^2_{N'})$. Hence, the result follows.
(NARROW)**:** similar to the case (NPROD) except we use Lemma 3.12 in [4].

□

We now prove termination of the algorithm.

Definition C.13 (**Plinth**)**.** *A* plinth $\beth \subset \mathscr{T}$ *is a set of types with the following properties:*

- *$\beth$ is finite;*
- *$\beth$ contains $\mathbb{1}$, $\mathbb{0}$ and is closed under Boolean connectives ($\wedge, \vee, \neg$);*
- *for all types $(t_1 \times t_2)$ or $(t_1 \to t_2)$ in $\beth$, we have $t_1 \in \beth$ and $t_2 \in \beth$.*

As stated in [9], every finite set of types is included in a plinth. Indeed, we already know that for a regular type t the set of its subtrees S is finite. The definition of the plinth ensures that the closure of S under Boolean connective is also finite. Moreover, if t belongs to a plinth $\beth$, then the set of its subtrees is contained in $\beth$. This is used to show the termination of algorithms working on types.

Lemma C.14 (**Termination**)**.** *Let C be a finite constraint-set. Then the normalization of C terminates.*

Proof. Let T be the set of type occurring in C. As C is finite, T is finite as well. Let $\sqsupset$ be a plinth such that $T \subseteq \sqsupset$. Then when we normalize a constraint $(t \leq \mathbb{0})$ during the process of $\emptyset \vdash_{\mathscr{N}} C$, t would belong to $\sqsupset$. We prove the lemma by induction on $(|\sqsupset \setminus \Sigma|, U, |C|)$ lexicographically ordered, where Σ is the set of types we have normalized, U is the number of unions $\vee$ occurring in the constraint-set C plus the number of constraints $(t, \geq, s)$ and the number of constraint $(t, \leq, s)$ where $s \neq \mathbb{0}$ or t is not in disjunctive normal form, and C is the constraint-set to be normalized.

<u>(NEMPTY)</u>**:** it terminates immediately.
<u>(NJOIN)</u>**:** $|C|$ decreases, and neither $|\sqsupset \setminus \Sigma|$ nor U increase.
<u>(NSYM)</u>**:** U decreases and Σ is unchanged
<u>(NZERO)</u>**:** U decreases and Σ is unchanged.
<u>(NDNF)</u>**:** U decreases and Σ is unchanged.
<u>(NUNION)</u>**:** although $|C|$ increases, U decreases and Σ is unchanged
<u>(NTLV)</u>**:** it terminates immediately.
<u>(NHYP)</u>**:** it terminates immediately.
<u>(NASSUM)</u>**:** as $\tau_0 \in \sqsupset$ and $\tau_0 \notin \Sigma$, the number $|\sqsupset \setminus \Sigma|$ decreases.
<u>(NBASIC)</u>**:** it terminates immediately.
<u>(NPROD)</u>**:** although $(|\sqsupset \setminus \Sigma|, U, |C|)$ may not change, the next rule to apply must be one of (NEMPTY), (NJOIN), (NSYM), (NZERO), (NDNF), (NUNION), (NTLV), (NHYP) or (NASSUM). Therefore, either the normalization terminates or the triple decreases in the next step.
<u>(NARROW)</u>**:** similar to Case (NPROD).

□

Lemma C.15 (Finiteness). *Let C be a constraint-set and $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. Then $\mathscr{S}$ is finite.*

Proof. It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints. □

Definition C.16. *Let C be a normalized constraint-set and O an ordering on $\mathsf{var}(C)$. We say C is* well-ordered *if for all normalized constraint $(\alpha, c, t_\alpha) \in C$ and for all $\beta \in \mathsf{tlv}(t_\alpha)$, $O(\alpha) < O(\beta)$ holds.*

Lemma C.17. *Let C be a constraint-set and $\emptyset \vdash_{\mathscr{N}} C \rightsquigarrow \mathscr{S}$. Then for all normalized constraint-set $C' \in \mathscr{S}$, C' is well-ordered.*

Proof. The only way to generate normalized constraints is Rule (NTLV), where we have selected the normalized constraint for the type variable α whose order is minimum as the representative one, that is, $\forall \beta \in \mathsf{tlv}(t_\alpha) \,.\, O(\alpha) < O(\beta)$. Therefore, the result follows. □

Definition C.18. *A general renaming ρ is a special type substitution that maps each type variable to another (fresh) type variable.*

Lemma C.19. *Let t, s be two types and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. Then if $\emptyset \vdash_{\mathscr{N}} \{(s \wedge t, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathscr{N}} \{((\bigwedge_{j\in J} s\rho_j) \wedge (\bigwedge_{i\in I} t\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.*

Proof. By induction on the number of (NPROD) and (NARROW) used in the derivation of $\emptyset \vdash_{\mathscr{N}} \{(s \wedge \neg t, \leq, \mathbb{0})\}$ and by cases on the disjunctive normal form τ of $s \wedge \neg t$. The failure of the normalization of $(s \wedge t, \leq, \mathbb{0})$ is essentially due to (NBASIC-F), (NPROD) and (NARROW), where there are no top-level type variables to make the type empty.
The case of arrows is a little complicated, as we need to consider more than two types: one type for the negative parts and two types for the positive parts from t and s respectively. Indeed, what we prove is the following stronger statement:

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{k\in K} t_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \Longrightarrow \emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{k\in K} (\bigwedge_{i_k\in I_k} t_k\rho_{i_k}), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $|K| \geq 2$ and ρ_{i_k}'s are general renamings. For simplicity, we only consider $|K| = 2$, as it is easy to extend to the case of $|K| > 2$.

<u>**Case 1**</u>**:** $\tau = \tau_{b_s} \wedge \tau_{b_t}$ and $\tau \not\simeq \mathbb{0}$, where τ_{b_s} (τ_{b_t} resp.) is an intersection of basic types from s (t resp.). Then the expansion of τ is

$$(\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i) \simeq \tau_{b_s} \wedge \tau_{b_t} \not\simeq \mathbb{0}$$

So $\emptyset \vdash_{\mathscr{N}} \{((\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.

Case 2: $\tau = \bigwedge_{p_s\in P_s}(w_{p_s}\times v_{p_s})\wedge\bigwedge_{n_s\in N_s}\neg(w_{n_s}\times v_{n_s})\wedge\bigwedge_{p_t\in P_t}(w_{p_t}\times v_{p_t})\wedge\bigwedge_{n_t\in N_t}\neg(w_{n_t}\times v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset\vdash_{\mathscr{N}}\{\tau,\leq,\mathbb{0})\}\rightsquigarrow\emptyset$, by the rule (NPROD), there exist two sets $N'_s\subseteq N_s$ and $N'_t\subseteq N_t$ such that

$$\begin{cases} \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{p_s\in P_s}w_{p_s}\wedge\bigwedge_{n_s\in N'_s}\neg w_{n_s}\wedge\bigwedge_{p_t\in P_t}w_{p_t}\wedge\bigwedge_{n_t\in N'_t}\neg w_{n_t},\leq,\mathbb{0})\}\rightsquigarrow\emptyset\\ \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{p_s\in P_s}v_{p_s}\wedge\bigwedge_{n_s\in N_s\setminus N'_s}\neg v_{n_s}\wedge\bigwedge_{p_t\in P_t}v_{p_t}\wedge\bigwedge_{n_t\in N_t\setminus N'_t}\neg v_{n_t},\leq,\mathbb{0})\}\rightsquigarrow\emptyset \end{cases}$$

By induction, we have

$$\begin{cases} \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\bigwedge_{p_s\in P_s}w_{p_s}\wedge\bigwedge_{n_s\in N'_s}\neg w_{n_s})\rho_j\wedge\bigwedge_{i\in I}(\bigwedge_{p_t\in P_t}w_{p_t}\wedge\bigwedge_{n_t\in N'_t}\neg w_{n_t})\rho_i,\leq,\mathbb{0})\}\rightsquigarrow\emptyset\\ \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\bigwedge_{p_s\in P_s}v_{p_s}\wedge\bigwedge_{n_s\in N_s\setminus N'_s}\neg v_{n_s})\rho_j\wedge\bigwedge_{i\in I}(\bigwedge_{p_t\in P_t}v_{p_t}\wedge\bigwedge_{n_t\in N_t\setminus N'_t}\neg v_{n_t})\rho_i,\leq,\mathbb{0})\}\rightsquigarrow\emptyset \end{cases}$$

Then by the rule (NPROD) again, we get

$$\emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\tau_s)\rho_j\wedge\bigwedge_{i\in I}(\tau_t)\rho_i,\leq,\mathbb{0})\}\rightsquigarrow\emptyset$$

where $\tau_s = \bigwedge_{p_s\in P_s}(w_{p_s}\times v_{p_s})\wedge\bigwedge_{n_s\in N_s}\neg(w_{n_s}\times v_{n_s})$ and $\tau_t = \bigwedge_{p_t\in P_t}(w_{p_t}\times v_{p_t})\wedge\bigwedge_{n_t\in N_t}\neg(w_{n_t}\times v_{n_t})$.

Case 3: $\tau = \bigwedge_{p_s\in P_s}(w_{p_s}\to v_{p_s})\wedge\bigwedge_{n_s\in N_s}\neg(w_{n_s}\to v_{n_s})\wedge\bigwedge_{p_t\in P_t}(w_{p_t}\to v_{p_t})\wedge\bigwedge_{n_t\in N_t}\neg(w_{n_t}\to v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset\vdash_{\mathscr{N}}\{\tau,\leq,\mathbb{0})\}\rightsquigarrow\emptyset$, by the rule (NARROW), for all $w\to v\in N_s\cup N_t$, there exist a set $P'_s\subseteq P_s$ and a set $P'_t\subseteq P_t$ such that

$$\begin{cases} \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{p_s\in P'_s}\neg w_{p_s}\wedge\bigwedge_{p_t\in P'_t}\neg w_{p_t}\wedge w,\leq,\mathbb{0})\}\rightsquigarrow\emptyset\\ P'_s=P_s\wedge P'_t=P_t \text{ or } \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{p_s\in P_s\setminus P'_s}v_{p_s}\wedge\bigwedge_{p_t\in P_t\setminus P'_t}v_{p_t}\wedge\neg v,\leq,\mathbb{0})\}\rightsquigarrow\emptyset \end{cases}$$

By induction, for all $\rho\in[\rho_i]_{i\in I}\cup[\rho_j]_{j\in J}$, we have

$$\begin{cases} \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\bigwedge_{p_s\in P'_s}\neg w_{p_s})\rho_j\wedge\bigwedge_{i\in I}(\bigwedge_{p_t\in P'_t}\neg w_{p_t})\rho_i\wedge w\rho,\leq,\mathbb{0})\}\rightsquigarrow\emptyset\\ \begin{cases} P'_s=P_s\wedge P'_t=P_t\\ \text{or}\\ \emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\bigwedge_{p_s\in P_s\setminus P'_s}v_{p_s})\rho_j\wedge\bigwedge_{i\in I}(\bigwedge_{p_t\in P_t\setminus P'_t}v_{p_t})\rho_i\wedge\neg v\rho,\leq,\mathbb{0})\}\rightsquigarrow\emptyset \end{cases} \end{cases}$$

Then by the rule (NARROW) again, we get

$$\emptyset\vdash_{\mathscr{N}}\{\bigwedge_{j\in J}(\tau_s)\rho_j\wedge\bigwedge_{i\in I}(\tau_t)\rho_i,\leq,\mathbb{0})\}\rightsquigarrow\emptyset$$

where $\tau_s = \bigwedge_{p_s\in P_s}(w_{p_s}\to v_{p_s})\wedge\bigwedge_{n_s\in N_s}\neg(w_{n_s}\to v_{n_s})$ and $\tau_t = \bigwedge_{p_t\in P_t}(w_{p_t}\to v_{p_t})\wedge\bigwedge_{n_t\in N_t}\neg(w_{n_t}\to v_{n_t})$.

Case 4: $\tau = (\bigvee_{k_s\in K_s}\tau_{k_s})\wedge(\bigvee_{k_t\in K_t}\tau_{k_t})$, where τ_{k_s} and τ_{k_t} are single normal forms. As $\emptyset\vdash_{\mathscr{N}}\{(\tau,\leq,\mathbb{0})\}\rightsquigarrow\emptyset$, there must exist at least one $k_s\in K_s$ and at least one $k_t\in K_t$ such that $\emptyset\vdash_{\mathscr{N}}\{(\tau_{k_s}\wedge\tau_{k_t},\leq,\mathbb{0})\}\rightsquigarrow\emptyset$. By Cases $(1)-(3)$, the result follows.

□

The type tallying problem is parameterized with a set Δ of type variables that cannot be instantiated, but so far, we have not considered these monomorphic variables in the normalization procedure. Taking Δ into account affects only the (NTLV) rule, where a normalized constraint is built by singling out a variable α. Since the type substitution σ we want to construct must not touch the type variables in Δ (*i.e.*, $\sigma\sharp\Delta$), we cannot choose a variable α in Δ. To avoid this, we order the variables in C so that those belonging to Δ are always greater than those not in Δ. If, by choosing the minimum top-level variable α, we obtain $\alpha\in\Delta$, it means that all the top-level type variables are contained in Δ. According to Lemmas C.3 and C.11 in the companion paper [3], we can then safely eliminate these type variables. So taking Δ into account, we amend the (NTLV) rule as follows.

$$\frac{\mathsf{tlv}(\tau_0)=\emptyset\qquad \alpha'\,\lessdot_O P\cup N\qquad \mathscr{S}=\begin{cases}\{\{(\alpha',\leq,\neg t_{\alpha'})\}\} & \alpha'\in P\setminus\Delta\\ \{\{(\alpha',\geq,t_{\alpha'})\}\} & \alpha'\in N\setminus\Delta\\ \Sigma\vdash_{\mathscr{N}}\{(\tau_0\leq\mathbb{0})\} & \alpha'\in\Delta\end{cases}}{\Sigma\vdash_{\mathscr{N}}\{(\bigwedge_{\alpha\in P}\alpha\wedge\bigwedge_{\alpha\in N}\neg\alpha\wedge\tau_0\leq\mathbb{0})\}\rightsquigarrow\mathscr{S}}\,(\text{NTLV})$$

Furthermore, it is easy to prove the soundness, completeness, and termination of the algorithm extended with Δ.

C.1.2 Constraint merging

A normalized constraint-set may contain several constraints for a same type variable, which can eventually be merged together. For instance, the constraints $\alpha \geq t_1$ and $\alpha \geq t_2$ can be replaced by $\alpha \geq t_1 \vee t_2$, and the constraints $\alpha \leq t_1$ and $\alpha \leq t_2$ can be replaced by $\alpha \leq t_1 \wedge t_2$. That is to say, we can merge all the lower bounds (resp. upper bounds) of a type variable into only one by unions (resp. intersections).

$$\frac{\forall i \in I \,.\, (\alpha \geq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathscr{M}} C \rightsquigarrow (C \setminus \{(\alpha \geq t_i) \mid i \in I\}) \cup \{(\alpha \geq \bigvee_{i \in I} t_i)\}}\text{(MLB)}$$

$$\frac{\forall i \in I \,.\, (\alpha \leq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathscr{M}} C \rightsquigarrow (C \setminus \{(\alpha \leq t_i) \mid i \in I\}) \cup \{(\alpha \leq \bigwedge_{i \in I} t_i)\}}\text{(MUB)}$$

Figure 8. Merging rules

After repeated uses of the merging rules, a set C contains at most one lower bound constraint and at most one upper bound constraint for each type variable. If both lower and upper bounds exist for a given α, that is, $\alpha \geq t_1$ and $\alpha \leq t_2$ belong to C, then the substitution we want to construct from C must satisfy the constraint $(t_1, \leq, t_2)$ as well. For that, we first normalize the constraint $(t_1, \leq, t_2)$, yielding a set of constraint-sets $\mathscr{S}$, and then saturate C with any normalized constraint-set $C' \in \mathscr{S}$. Formally, we describe the saturation process as the saturation rule $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$, where Σ_p is a set of type pairs (if $(t_1, t_2) \in \Sigma_p$, then the constraint $t_1 \leq t_2$ has already been treated at this point), C_Σ is a normalized constraint-set (which collects the treated original constraints, like $(\alpha, \geq, t_1)$ and $(\alpha, \leq, t_2)$, that generate the additional constraints), C is the normalized constraint-set we want to saturate, and $\mathscr{S}$ is a set of sets of normalized constraints (the result of the saturation of C joined with C_Σ). The saturation rules are given in Figure 9, which describe the same algorithm as *Step 2* of the function merge given in Subsection 3.2.1.

$$\frac{\Sigma_p, C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S} \qquad (t_1, t_2) \in \Sigma_p}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \mathscr{S}}\text{(SHYP)}$$

$$\frac{\begin{array}{l}(t_1, t_2) \notin \Sigma_p \quad \emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S} \\ \mathscr{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathscr{S} \\ \forall C' \in \mathscr{S}' .\, \Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}\end{array}}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \bigsqcup_{C' \in \mathscr{S}'} \mathscr{S}_{C'}}\text{(SASSUM)}$$

$$\frac{\forall \alpha, t_1, t_2 \nexists \{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C}{\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \{C \cup C_\Sigma\}}\text{(SDONE)}$$

where $\Sigma_p, C_\Sigma \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$ means that there exists C' such that $\vdash_{\mathscr{M}} C \rightsquigarrow C'$ and $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C' \rightsquigarrow \mathscr{S}$.

Figure 9. Saturation rules

If $\alpha \geq t_1$ and $\alpha \leq t_2$ belongs to the constraint-set C that is being saturated, and $t_1 \leq t_2$ has already been processed (i.e., $(t_1, t_2) \in \Sigma_p$), then the rule (SHYP) simply extends C_Σ (the result of the saturation so far) with $\{\alpha \geq t_1, \alpha \leq t_2\}$. Otherwise, the rule (SASSUM) first normalizes the fresh constraint $\{t_1 \leq t_2\}$, yielding a set of normalized constraint-sets $\mathscr{S}$. It then saturates (joins) C and C_Σ with each constraint-set $C_{\mathscr{S}} \in \mathscr{S}$, the union of which gives a new set $\mathscr{S}'$ of normalized constrain-sets. Each C' in $\mathscr{S}'$ may contain several constraints for the same type variable, so they have to be merged and saturated themselves. Finally, if C does not contain any couple $\alpha \geq t_1$ and $\alpha \leq t_2$ for a given α, the process is over and the rule (SDONE) simply returns $C \cup C_\Sigma$.

If $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$, then the result of the merging of C is $\mathscr{S}$.

Lemma C.20 (Soundness). *Let C be a normalized constraint-set. If $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$, then for all normalized constraint-set $C' \in \mathscr{S}$ and all substitution σ, we have $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathscr{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C'$, then $\sigma \Vdash C$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$. For all σ and $C_0 \in \mathscr{S}$, if $\sigma \Vdash C_0$, then $\sigma \Vdash C_\Sigma \cup C$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$.

(SHYP): by induction, we have $\sigma \Vdash (C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\}) \cup C$, that is $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SASSUM): according to Definition C.4, $C_0 \in \mathscr{S}_{C'}$ for some $C' \in \mathscr{S}'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}$, we have $\sigma \Vdash C'$. Moreover, the equation $\mathscr{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathscr{S}$ gives us $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma \subseteq C'$. Therefore, we have $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SDONE): straightforward.

□

Lemma C.21 (Completeness). *Let C be a normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathscr{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathscr{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C$, then $\sigma \Vdash C'$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$. For all σ, if $\sigma \Vdash C_\Sigma \cup C$, then there exists $C_0 \in \mathscr{S}$ such that $\sigma \Vdash C_0$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$.

(SHYP): the result follows by induction.

(SASSUM): as $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$, we have $\sigma \Vdash \{(t_1 \leq t_2)\}$. As $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$, applying Lemma C.12, there exists $C_0' \in \mathscr{S}$ such that $\sigma \Vdash C_0'$. Let $C' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \cup C_0'$. Clearly we have $\sigma \Vdash C'$ and $C' \in S'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C' \rightsquigarrow \mathscr{S}_{C'}$, there exists $C_0 \in \mathscr{S}_{C'}$ such that $\sigma \Vdash C_0$. Moreover, it is clear that $C_0 \in \bigsqcup_{C' \in S'} \mathscr{S}_{C'}$. Therefore, the result follows.

(SDONE): straightforward.

□

Lemma C.22 (Termination). *Let C be a finite normalized constraint-set. Then $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C$ terminates.*

Proof. Let T be the set of types occurring in C. As C is finite, T is finite as well. Let $\beth$ be a plinth such that $T \subseteq \beth$. Then when we saturate a fresh constraint $(t_1, \leq, t_2)$ during the process of $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C$, (t_1, t_2) would belong to $\beth \times \beth$. According to Lemma C.14, we know that $\emptyset \vdash_{\mathscr{N}} \{(t_1, \leq, t_2)\}$ terminates. Moreover,the termination of the merging of the lower bounds or the upper bounds of a same type variable is straightforward. Finally, we have to prove termination of the saturation process. The proof proceeds by induction on $(|(\beth \times \beth)| - |\Sigma_p|, |C|)$ lexicographically ordered:

$(Shyp)$: $|C|$ decreases.

$(Sassum)$: as $(t_1, t_2) \notin \Sigma_p$ and $t_1, t_2 \in \beth$, $|(\beth \times \beth)| - |\Sigma_p|$ decreases.

$(Sdone)$: it terminates immediately.

□

Definition C.23 (Sub-constraint). *Let $C_1, C_2 \subseteq \mathscr{C}$ be two normalized constraint-sets. We say C_1 is a* sub-constraint *of C_2, denoted as $C_1 \lessdot C_2$, if for all $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ such that $t' \; c \; t$, where $c \in \{\leq, \geq\}$.*

Lemma C.24. *Let $C_1, C_2 \subseteq \mathscr{C}$ be two normalized constraint-sets and $C_1 \lessdot C_2$. Then for all substitution σ, if $\sigma \Vdash C_2$, then $\sigma \Vdash C_1$.*

Proof. Considering any constraint $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ and $t' \; c \; t$, where $c \in \{\leq, \geq\}$. Since $\sigma \Vdash C_2$, then $\sigma(\alpha) \; c \; t'\sigma$. Moreover, as $t' \; c \; t$, we have $t'\sigma \; c \; t\sigma$. Thus $\sigma(\alpha) \; c \; t\sigma$. □

Definition C.25. *Let $C \subseteq \mathscr{C}$ be a normalized constraint-set. We say C is* saturated *if for each type variable $\alpha \in \mathsf{dom}(C)$,*

(1) *there exists at most one form $(\alpha \geq t_1) \in C$,*

(2) *there exists at most one form* $(\alpha \leq t_2) \in C$,
(3) *if* $(\alpha \geq t_1), (\alpha \leq t_2) \in C$, *then* $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$ *and there exists* $C' \in \mathscr{S}$ *such that* C' *is a sub-constraint of* C *(*i.e., $C' \lessdot C$*).*

Lemma C.26. *Let* C *be a finite normalized constraint-set and* $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. *Then for all normalized constraint set* $C' \in \mathscr{S}$, C' *is saturated.*

Proof. We prove a stronger statement: assume $\Sigma_p, C_\Sigma \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. If

(i) for all $(t_1, t_2) \in \Sigma_p$ there exists $C' \in (\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\})$ such that $C' \lessdot C_\Sigma \cup C$ and
(ii) for all $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C_\Sigma$ the pair (t_1, t_2) is in Σ_p,

then C_0 is saturated for all $C_0 \in \mathscr{S}$.

The proof of conditions (1) and (2) for a saturated constraint-set is straightforward for all $C_0 \in \mathscr{S}$. The proof of the condition (3) proceeds by induction on the derivation $\Sigma_p, C_\Sigma \vdash_{\mathscr{S}} C \rightsquigarrow \mathscr{S}$ and a case analysis on the last rule used in the derivation.

(SHYP): as $(t_1, t_2) \in \Sigma_p$, the conditions (i) and (ii) hold for the premise. By induction, the result follows.
(SASSUME): take any premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathscr{S}} C'' \rightsquigarrow \mathscr{S}_{C'}$, where $C' \in \mathscr{S}'$ and $\vdash_{\mathscr{M}} C' \rightsquigarrow C''$. For any $(s_1, s_2) \in \Sigma_p$, the condition (i) gives us that there exists $C_0 \in (\emptyset \vdash_{\mathscr{N}} \{(s_1 \leq s_2)\})$ such that $C_0 \lessdot C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$. Since $\mathscr{S}' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \sqcap \mathscr{S}$, we have $C_0 \lessdot C''$. Moreover, consider (t_1, t_2). As $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$, there exists $C_0 \in \mathscr{S}$ such that $C_0 \lessdot C''$. Thus the condition (i) holds for the premise. Moreover, the condition (ii) holds straightforwardly for premise. By induction, the result follows.
(SDONE): the result follows by the conditions (i) and (ii).

$\square$

Lemma C.27 (Finiteness). *Let* C *be a constraint-set and* $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. *Then* $\mathscr{S}$ *is finite.*

Proof. It follows by Lemma C.15. $\square$

Lemma C.28. *Let* C *be a well-ordered normalized constraint-set and* $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. *Then for all normalized constraint-set* $C' \in \mathscr{S}$, C' *is well-ordered.*

Proof. The merging of the lower bounds (or the upper bounds) of a same type variable preserves the orders. The result of saturation is well-ordered by Lemma C.17. $\square$

Normalization and merging may produce redundant constraint-sets. For example, consider the constraint-set $\{(\alpha \times \beta), \leq, (\texttt{Int} \times \texttt{Bool})\}$. Applying the rule (NPROD), the normalization of this set is

$$\{\{(\alpha, \leq, \mathbb{0})\}, \{(\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \texttt{Int}), (\beta, \leq, \texttt{Bool})\}\}.$$

Clearly each constraint-set is a saturated one. Note that $\{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}$ is redundant, since any solution of this constraint-set is a solution of $\{(\alpha, \leq, \mathbb{0})\}$ and $\{(\beta, \leq, \mathbb{0})\}$. Therefore it is safe to eliminate it. Generally, for any two different normalized constraint sets $C_1, C_2 \in \mathscr{S}$, if $C_1 \lessdot C_2$, then according to Lemma C.24, any solution of C_2 is a solution of C_1. Therefore, C_2 can be eliminated from $\mathscr{S}$.

Definition C.29. *Let* $\mathscr{S}$ *be a set of normalized constraint-sets. We say that* $\mathscr{S}$ *is* minimal *if for any two different normalized constraint-sets* $C_1, C_2 \in \mathscr{S}$, *neither* $C_1 \lessdot C_2$ *nor* $C_2 \lessdot C_1$. *Moreover, we say* $\mathscr{S} \simeq \mathscr{S}'$ *if for all substitution* σ *such that* $\exists C \in \mathscr{S} . \sigma \Vdash C \iff \exists C' \in \mathscr{S}' . \sigma \Vdash C'$.

Lemma C.30. *Let* C *be a well-ordered normalized constraint-set and* $\emptyset, \emptyset \vdash_{\mathscr{M}\mathscr{S}} C \rightsquigarrow \mathscr{S}$. *Then there exists a minimal set* $\mathscr{S}_0$ *such that* $\mathscr{S}_0 \simeq \mathscr{S}$.

Proof. By eliminating the redundant constraint-sets in $\mathscr{S}$. $\square$

C.1.3 Constraint solving

From constraints to equations. Given a well-ordered saturated constraint-set, we transform it into an equivalent equation system. This shows that the type tallying problem is essentially a unification problem.

Definition C.31 (Equation system). *An* equation system E *is a set of equations of the form* $\alpha = t$ *such that there exists at most one equation in* E *for every type variable* α. *We define the domain of* E, *written* $\mathsf{dom}(E)$, *as the set* $\{\alpha \mid \exists t . \alpha = t \in E\}$.

Definition C.32 (Equation system solution). *Let* E *be an equation system. A solution to* E *is a substitution* σ *such that*

$$\forall \alpha = t \in E . \sigma(\alpha) \simeq t\sigma \text{ holds}$$

If σ *is a solution to* E, *we write* $\sigma \Vdash E$.

From a normalized constraint-set C, we obtain some explicit conditions for the substitution σ we want to construct from C. For instance, from the constraint $\alpha \leq t$ (resp. $\alpha \geq t$), we know that the type substituted for α must be a subtype of t (resp. a super type of t).

We assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound t_1 and a upper bound t_2 using, if necessary, the fact that $\mathbb{0} \leq \alpha \leq \mathbb{1}$. Formally, we rewrite C as follows:

$$\begin{cases} t_1 \leq \alpha \leq \mathbb{1} & \text{if } \alpha \geq t_1 \in C \text{ and } \nexists t.\ \alpha \leq t \in C \\ \mathbb{0} \leq \alpha \leq t_2 & \text{if } \alpha \leq t_2 \in C \text{ and } \nexists t.\ \alpha \geq t \in C \\ t_1 \leq \alpha \leq t_2 & \text{if } \alpha \geq t_1, \alpha \leq t_2 \in C \end{cases}$$

We then transform each constraint $t_1 \leq \alpha \leq t_2$ in C into an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$[12], where α' is a fresh type variable. The type $(t_1 \vee \alpha') \wedge t_2$ ranges from t_1 to t_2, so the equation $\alpha = (t_1 \vee \alpha') \wedge t_2$ expresses the constraint that $t_1 \leq \alpha \leq t_2$, as wished. We prove the soundness and completeness of this transformation.

To prove soundness, we define the rank n satisfaction predicate $\Vdash_n$ for equation systems, which is similar to the one for constraint-sets.

Lemma C.33 (**Soundness**). *Let $C \subseteq \mathscr{C}$ be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash E$ then $\sigma \Vdash C$.*

Proof. Without loss of generality, we assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound and an upper bound, that is $t_1 \leq \alpha \leq t_2 \in C$. We write $O(C_1) < O(C_2)$ if $O(\alpha) < O(\beta)$ for all $\alpha \in \mathsf{dom}(C_1)$ and all $\beta \in \mathsf{dom}(C_2)$. We first prove a stronger statement:

(*) for all σ, n and $C_\Sigma \subseteq C$, if $\sigma \Vdash_n E$, $\sigma \Vdash_n C_\Sigma$, $\sigma \Vdash_{n-1} C \setminus C_\Sigma$, and $O(C \setminus C_\Sigma) < O(C_\Sigma)$, then $\sigma \Vdash_n C \setminus C_\Sigma$.

Here C_Σ denotes the set of constraints that have been checked. The proof proceeds by induction on $|C \setminus C_\Sigma|$.

$C \setminus C_\Sigma = \emptyset$: straightforward.

$C \setminus C_\Sigma \neq \emptyset$: take the constraint $(t_1 \leq \alpha \leq t_2) \in C \setminus C_\Sigma$ such that $O(\alpha)$ is the maximum in $\mathsf{dom}(C \setminus C_\Sigma)$. Clearly, there exists a corresponding equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. As $\sigma \Vdash_n E$, we have $\sigma(\alpha) \simeq_n ((t_1 \vee \alpha') \wedge t_2)\sigma$. Then,

$$\begin{array}{rcl} \sigma(\alpha) \wedge \neg t_2\sigma & \simeq_n & ((t_1 \vee \alpha') \wedge t_2)\sigma \wedge \neg t_2\sigma \\ & \simeq_n & \mathbb{0} \end{array}$$

Therefore, $\sigma(\alpha) \leq_n t_2\sigma$.

Consider the constraint $(t_1, \leq, \alpha)$. We have

$$\begin{array}{rcl} t_1\sigma \wedge \neg\sigma(\alpha) & \simeq_n & t_1\sigma \wedge \neg((t_1 \vee \alpha') \wedge t_2)\sigma \\ & \simeq_n & t_1\sigma \wedge \neg t_2\sigma \end{array}$$

What remains to do is to check the subtyping relation $t_1\sigma \wedge \neg t_2\sigma \leq_n \mathbb{0}$, that is, to check that the judgement $\sigma \Vdash_n \{(t_1 \leq t_2)\}$ holds. Since the whole constraint-set C is saturated, according to Definition C.25, we have $\emptyset \vdash_{\mathscr{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathscr{S}$ and there exists $C' \in \mathscr{S}$ such that $C' \lessdot C$, that is $C' \lessdot C_\Sigma \cup C \setminus C_\Sigma$. Moreover, as C is well-ordered, $O(\{\alpha\}) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$ and thus $O(C \setminus C_\Sigma) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$. Therefore, we can deduce that $C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C_\Sigma$ and $C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C \setminus C_\Sigma$. From the premise and Lemma C.24, we have $\sigma \Vdash_n C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$ and $\sigma \Vdash_{n-1} C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$. Then, by Lemma C.11, we get $\sigma \Vdash_n \{(t_1 \leq t_2)\}$.

Finally, consider the constraint-set $C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. By induction, we have $\sigma \Vdash_n C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. Thus the result follows.

Finally, we explain how to prove the lemma with the statement (*). Take $C_\Sigma = \emptyset$. Since $\sigma \Vdash E$, we have $\sigma \Vdash_n E$ for all n. Trivially, we have $\sigma \Vdash_0 C$. This can be used to prove $\sigma \Vdash_1 C$. Since $\sigma \Vdash_1 E$, by (*), we get $\sigma \Vdash_1 C$, which will be used to prove $\sigma \Vdash_2 C$. Consequently, we can get $\sigma \Vdash_n C$ for all n, which clearly implies the lemma. $\square$

Lemma C.34 (**Completeness**). *Let $C \subseteq \mathscr{C}$ be a saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash C$ then there exists σ' such that $\sigma' \sharp \sigma$ and $\sigma \cup \sigma' \Vdash E$.*

Proof. Let $\sigma' = \{{}^{\sigma(\alpha)}/_{\alpha'} \mid \alpha \in \mathsf{dom}(C)\}$. Consider each equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. As $\sigma \Vdash C$, then $t_1\sigma \leq \sigma(\alpha)$ and $\sigma(\alpha) \leq t_2\sigma$. Thus

$$\begin{array}{rcl} ((t_1 \vee \alpha') \wedge t_2)(\sigma \cup \sigma') & = & (t_1(\sigma \cup \sigma') \vee \alpha'(\sigma \cup \sigma')) \wedge t_2(\sigma \cup \sigma') \\ & = & (t_1\sigma \vee \sigma(\alpha)) \wedge t_2\sigma \\ & \simeq & \sigma(\alpha) \wedge t_2\sigma \quad (t_1\sigma \leq \sigma(\alpha)) \\ & \simeq & \sigma(\alpha) \quad (\sigma(\alpha) \leq t_2\sigma) \\ & = & (\sigma \cup \sigma')(\alpha) \end{array}$$

[12] Or, equivalently, $\alpha = t_1 \vee (\alpha' \wedge t_2)$. Besides, in practice, if only $\alpha \geq t_1$ ($\alpha \leq t_2$ resp.) and all the occurrences of α in the co-domain of the function type are positive (negative resp.), we can use $\alpha = t_1$ ($\alpha = t_2$ resp.) instead, and the completeness is ensured by subsumption.

□

Definition C.35. *Let E be an equation system and O an ordering on* $\mathsf{dom}(E)$*. We say that E is* well ordered *if for all $\alpha = t_\alpha \in E$, we have $O(\alpha) < O(\beta)$ for all $\beta \in \mathit{tlv}(t_\alpha) \cap \mathsf{dom}(E)$.*

Lemma C.36. *Let C be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then E is well ordered.*

Proof. Clearly, $\mathsf{dom}(E) = \mathsf{dom}(C)$. Consider an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. By Definition C.16, for all $\beta \in (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. $O(\alpha) < O(\beta)$. Moreover, α' is a fresh type variable in C, that is $\alpha' \notin \mathsf{dom}(C)$. And then $\alpha' \notin \mathsf{dom}(E)$. Therefore, $\mathsf{tlv}((t_1 \vee \alpha') \wedge t_2) \cap \mathsf{dom}(E) = (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. Thus the result follows. □

Solution of Equation Systems. We now extract a solution (i.e., a substitution) from the equation system we build from C. In an equation $\alpha = t_\alpha$, α may also appear in the type t_α; such an equality reminds the definition of a recursive type. As a first step, we introduce a recursion operator μ in all the equations of the system, transforming $\alpha = t_\alpha$ into $\alpha = \mu x_\alpha.\ t_\alpha\{^{x_\alpha}/_{\alpha}\}$. This ensures that type variables do not appear in the right-hand side of the equalities, making the whole solving process easier. If some recursion operators are in fact not needed in the solution (*i.e.*, we have $\alpha = \mu x_\alpha . t_\alpha$ with $x_\alpha \notin \mathsf{fv}(t_\alpha)$), then we can simply eliminate them.

If the equation system contains only one equation, then this equation is immediately a substitution. Otherwise, consider the equation system $\{\alpha = \mu x_\alpha.\ t_\alpha\} \cup E$, where E contains only equations closed with the recursion operator μ as explained above. The next step is to substitute the content expression $\mu x_\alpha.\ t_\alpha$ for all the occurrences of α in equations in E. In detail, let $\beta = \mu x_\beta.\ t_\beta \in E$. Since t_α may contain some occurrences of β and these occurrences are clearly bounded by μx_β, we in fact replace the equation $\beta = \mu x_\beta.\ t_\beta$ with $\beta = \mu x_\beta.\ t_\beta\{^{\mu x_\alpha.\ t_\alpha}/_{\alpha}\}\{^{x_\beta}/_{\beta}\}$, yielding a new equation system E'. Finally, assume that the equation system E' (which has fewer equations) has a solution σ'. Then the substitution $\{^{t_\alpha \sigma'}/_{\alpha}\} \oplus \sigma'$ is a solution to the original equation system $\{\alpha = \mu x_\alpha.\ t_\alpha\} \cup E$. The solving algorithm $\mathsf{Unify}()$ is given in Figure 10.

Require: an equation system E
Ensure: a substitution σ

```
1.  let $e2mu$ $(\alpha, t_\alpha)$ = $(\alpha,\ \mu x_\alpha.\ t_\alpha\{^{x_\alpha}/_{\alpha}\})$ in
2.  let $subst$ $(\alpha, t_\alpha)$ $(\beta, t_\beta)$ = $(\beta,\ t_\beta\{^{t_\alpha}/_{\alpha}\}\{^{x_\beta}/_{\beta}\})$ in
3.  let rec $mu2sub$ $E$ =
4.    match $E$ with
5.     |[ ] → [ ]
6.     |$(\alpha, t_\alpha) :: E'$ →
7.       let $E''$ = $List.map$ $(subst\ (\alpha, t_\alpha))$ $E'$ in
8.       let $\sigma'$ = $mu2sub$ $E''$ in $\{^{t_\alpha \sigma'}/_{\alpha}\} \oplus \sigma'$
9.  in
10. let $e2sub$ $E$ =
11.    let $E'$ = $List.map$ $e2mu$ $E$ in
12.    $mu2sub$ $E'$
```

Figure 10. Equation system solving algorithm $\mathsf{Unify}()$

Definition C.37 (General solution). *Let E be an equation system. A* general solution *to E is a substitution σ from* $\mathsf{dom}(E)$ *to $\mathscr{T}$ such that*

$$\forall \alpha \in \mathsf{dom}(\sigma)\ .\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$$

and

$$\forall \alpha = t \in E\ .\ \sigma(\alpha) \simeq t\sigma \text{ holds}$$

Lemma C.38. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then $\forall \alpha \in \mathsf{dom}(\sigma).\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$ and $\mathsf{dom}(\sigma) = \mathsf{dom}(E)$.*

Proof. The algorithm $\mathsf{Unify}()$ consists of two steps: (i) transform types into recursive types and (ii) extract the substitution. After the first step, for each equation $(\alpha = t_\alpha) \in E$, we have $\alpha \notin \mathsf{var}(t_\alpha)$. Consider the second step. Let $\mathsf{var}(E) = \bigcup_{(\alpha = t_\alpha) \in E} \mathsf{var}(t_\alpha)$ and $\overline{S} = \mathscr{V} \setminus S$, where S is a set of type variables. We prove a stronger statement:

$$\forall \alpha \in \mathsf{dom}(\sigma).\ \mathsf{var}(\sigma(\alpha)) \cap (\mathsf{dom}(\sigma) \cup \overline{\mathsf{var}(E)}) = \emptyset \text{ and } \mathsf{dom}(\sigma) = \mathsf{dom}(E).$$

The proof proceeds by induction on E:

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' = \{(\beta = t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma = \{{}^{t_\alpha\sigma''}/_{\alpha}\} \oplus \sigma''$. By induction, we have $\forall\beta \in \mathsf{dom}(\sigma'')$. $\mathsf{var}(\sigma''(\beta)) \cap (\mathsf{dom}(\sigma'') \cup \mathsf{var}(E'')) = \emptyset$ and $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, we have $\alpha \notin \mathsf{dom}(\sigma'')$ and then $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\} = \mathsf{dom}(E)$.

Moreover, $\alpha \notin \mathsf{var}(E'')$, then $\mathsf{dom}(\sigma) \subset \mathsf{dom}(\sigma'') \cup \mathsf{var}(E'')$. Thus, for all $\beta \in \mathsf{dom}(\sigma'')$, we have $\mathsf{var}(\sigma''(\beta)) \cap \mathsf{dom}(\sigma) = \emptyset$. Consider $t_\alpha\sigma''$. It is clear that $\mathsf{var}(t_\alpha\sigma'') \cap \mathsf{dom}(\sigma) = \emptyset$. Besides, the algorithm does not introduce any fresh variable, then for all $\beta \in \mathsf{dom}(\sigma)$, we have $\mathsf{var}(t_\beta) \cap \mathsf{var}(E) = \emptyset$. Therefore, the result follows.

□

Lemma C.39 (Soundness). *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then $\sigma \Vdash E$.*

Proof. By induction on E.

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' = \{(\beta = t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma = \{{}^{t_\alpha\sigma''}/_{\alpha}\} \oplus \sigma''$. By induction, we have $\sigma'' \Vdash E''$. According to Lemma C.38, we have $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. So $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Considering any equation $(\beta = t_\beta) \in E$ where $\beta \in \mathsf{dom}(\sigma'')$. Then

$$\begin{array}{rll}
\sigma(\beta) = & \sigma''(\beta) & \text{(apply } \sigma\text{)} \\
\simeq & t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}\sigma'' & \text{(as } \sigma'' \Vdash E''\text{)} \\
= & t_\beta\{{}^{t_\alpha\{{}^{x_\beta}/_{\beta}\}}/_{\alpha}, {}^{x_\beta}/_{\beta}\}\sigma'' & \\
= & t_\beta\{{}^{t_\alpha\{{}^{x_\beta}/_{\beta}\}\sigma''}/_{\alpha}, {}^{x_\beta\sigma''}/_{\beta}\} \oplus \sigma'' & \\
= & t_\beta\{{}^{t_\alpha(\{{}^{x_\beta\sigma''}/_{\beta}\} \oplus \sigma'')}/_{\alpha}, {}^{x_\beta\sigma''}/_{\beta}\} \oplus \sigma'' & \\
\simeq & t_\beta\{{}^{t_\alpha(\{{}^{t_\beta\sigma''}/_{\beta}\} \oplus \sigma'')}/_{\alpha}, {}^{t_\beta\sigma''}/_{\beta}\} \oplus \sigma'' & \text{(expand } x_\beta\text{)} \\
\simeq & t_\beta\{{}^{t_\alpha(\{{}^{\beta\sigma''}/_{\beta}\} \oplus \sigma'')}/_{\alpha}, {}^{\beta\sigma''}/_{\beta}\} \oplus \sigma'' & \text{(as } \sigma'' \Vdash E''\text{)} \\
= & t_\beta\{{}^{t_\alpha\sigma''}/_{\alpha}\} \oplus \sigma'' & \\
= & t_\beta\sigma &
\end{array}$$

Finally, consider the equation $(\alpha = t_\alpha)$. As

$$\begin{array}{rll}
\sigma(\alpha) = & t_\alpha\sigma'' & \text{(apply } \sigma\text{)} \\
= & t_\alpha\{{}^{\beta\sigma''}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'')\} & \text{(expand } \sigma''\text{)} \\
= & t_\alpha\{{}^{\beta\sigma}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'')\} & \text{(as } \beta\sigma = \beta\sigma''\text{)} \\
= & t_\alpha\{{}^{\beta\sigma}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'') \cup \{\alpha\}\} & \text{(as } \alpha \notin \mathsf{var}(t_\alpha)\text{)} \\
= & t_\alpha\{{}^{\beta\sigma}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma)\} & \text{(as } \mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}\text{)} \\
= & t_\alpha\sigma &
\end{array}$$

Thus, the result follows.

□

Lemma C.40. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then σ is a general solution to E.*

Proof. Immediate consequence of Lemmas C.38 and C.39. □

Clearly, given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates with a substitution σ.

Lemma C.41 (Termination). *Given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates.*

Proof. By induction on the number of equations in E. □

Definition C.42. *Let σ, σ' be two substitutions. We say $\sigma \simeq \sigma'$ if and only if $\forall\alpha.\ \sigma(\alpha) \simeq \sigma'(\alpha)$.*

Lemma C.43 (Completeness). *Let E be an equation system. For all substitution σ, if $\sigma \Vdash E$, then there exist σ_0 and σ' such that $\sigma_0 = \mathsf{Unify}(E)$ and $\sigma \simeq \sigma' \circ \sigma_0$.*

Proof. According to Lemma C.41, there exists σ_0 such that $\sigma_0 = \mathsf{Unify}(E)$. For any $\alpha \notin \mathsf{dom}(\sigma_0)$, clearly we have $\alpha\sigma_0\sigma = \alpha\sigma$ and then $\alpha\sigma_0\sigma \simeq \alpha\sigma$. What remains to prove is that if $\sigma \Vdash E$ and $\sigma_0 = \mathsf{Unify}(E)$ then $\forall\alpha \in \mathsf{dom}(\sigma_0)$. $\alpha\sigma_0\sigma \simeq \alpha\sigma$. The proof proceeds by induction on E:

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$: let $E'' = \{(\beta = t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma_0 = \{{}^{t_\alpha\sigma''}/_{\alpha}\} \oplus \sigma''$. Considering each equation $(\beta = t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}) \in E''$, we have

$$\begin{array}{rll}
t_\beta\{{}^{t_\alpha}/_{\alpha}\}\{{}^{x_\beta}/_{\beta}\}\sigma = & t_\beta\{{}^{t_\alpha\{{}^{x_\beta}/_{\beta}\}}/_{\alpha}, {}^{x_\beta}/_{x_\beta}\}\sigma & \\
= & t_\beta\{{}^{t_\alpha\{{}^{x_\beta}/_{\beta}\}\sigma}/_{\alpha}, {}^{x_\beta\sigma}/_{\beta}\} \oplus \sigma & \\
= & t_\beta\{{}^{t_\alpha(\{{}^{x_\beta\sigma}/_{\beta}\} \oplus \sigma)}/_{\alpha}, {}^{x_\beta\sigma}/_{\beta}\} \oplus \sigma & \\
\simeq & t_\beta\{{}^{t_\alpha(\{{}^{t_\beta\sigma}/_{\beta}\} \oplus \sigma)}/_{\alpha}, {}^{t_\beta\sigma}/_{\beta}\} \oplus \sigma & (\text{expand } x_\beta) \\
\simeq & t_\beta\{{}^{t_\alpha(\{{}^{\beta\sigma}/_{\beta}\} \oplus \sigma)}/_{\alpha}, {}^{\beta\sigma}/_{\beta}\} \oplus \sigma & (\text{as } \sigma \Vdash E) \\
= & t_\beta\{{}^{t_\alpha\sigma}/_{\alpha}\} \oplus \sigma & \\
\simeq & t_\beta\{{}^{\alpha\sigma}/_{\alpha}\} \oplus \sigma & \\
= & t_\beta\sigma & \\
\simeq & \beta\sigma &
\end{array}$$

Therefore, $\sigma \Vdash E''$. By induction on E'', we have $\forall \beta \in \mathsf{dom}(\sigma'')$. $\beta\sigma''\sigma \simeq \beta\sigma$. According to Lemma C.38, $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, then $\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Therefore for any $\beta \in \mathsf{dom}(\sigma'') \cap \mathsf{dom}(\sigma_0)$, $\beta\sigma_0\sigma \simeq \beta\sigma''\sigma \simeq \beta\sigma$. Finally, considering α, we have

$$\begin{array}{rll}
\alpha\sigma_0\sigma = & t_\alpha\sigma''\sigma & (\text{apply } \sigma_0) \\
= & t_\alpha\{{}^{\beta\sigma''}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'')\}\sigma & (\text{expand } \sigma'') \\
= & t_\alpha\{{}^{\beta\sigma''\sigma}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & \\
\simeq & t_\alpha\{{}^{\beta\sigma}/_{\beta} \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & (\text{as } \forall \beta \in \sigma''.\ \beta\sigma \simeq \beta\sigma''\sigma) \\
= & t_\alpha\sigma & \\
\simeq & \alpha\sigma & (\text{as } \sigma \Vdash E)
\end{array}$$

Therefore, the result follows.

□

In our calculus, a type is well-formed if and only if the recursion traverses a constructor. In other words, the recursive variable should not appear at the top level of the recursive content. For example, the type $\mu x.\ x \vee t$ is not well-formed. To make the substitutions usable, we should avoid these substitutions with ill-formed types. Fortunately, this can been done by giving an ordering on the domain of an equation system to make sure that the equation system is well-ordered.

Lemma C.44. *Let E be a well-ordered equation system. If $\sigma = \mathsf{Unify}(E)$, then for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof. Assume that there exists an ill-formed $\sigma(\alpha)$. That is, $\sigma(\alpha) = \mu x.\ t$ where x occurs at the top level of t. According to the algorithm $\mathsf{Unify}()$, there exists a sequence of equations $(\alpha =)\alpha_0 = t_{\alpha_0}, \alpha_1 = t_{\alpha_1}, \ldots, \alpha_n = t_{\alpha_n}$ such that $\alpha_i \in \mathsf{tlv}(t_{\alpha_{i-1}})$ and $\alpha_0 \in \mathsf{tlv}(t_{\alpha_n})$ where $i \in \{1, \ldots, n\}$ and $n \geq 0$. According to Definition C.35, $O(\alpha_{i-1}) < O(\alpha_i)$ and $O(\alpha_n) < O(\alpha_0)$. Therefore, we have $O(\alpha_0) < O(\alpha_1) < \ldots < O(\alpha_n) < O(\alpha_0)$, which is impossible. Thus the result follows. □

As mentioned above, there may be some useless recursion constructor μ. They can be eliminated by checking whether the recursive variable appears in the content expression or not. Moreover, if a recursive type is empty (which can be checked with the subtyping algorithm), then it can be replaced by $\mathbb{0}$.

C.1.4 The complete algorithm

To conclude, we now describe the solving procedure $\mathsf{Sol}_\Delta(C)$ for the type tallying problem C. We first normalize C into a finite set $\mathscr{S}$ of well-ordered normalized constraint-sets (`Step 1`). If $\mathscr{S}$ is empty, then there are no solutions to C. Otherwise, each constraint-set $C_i \in \mathscr{S}$ is merged and saturated into a finite set $\mathscr{S}_{C_i}$ of well-order saturated normalized constraint-sets (`Step 2`). Then all these sets are collected into another set $\mathscr{S}'$ (*i.e.*, $\mathscr{S}' = \bigsqcup_{C_i \in \mathscr{S}} \mathscr{S}_{C_i}$). If $\mathscr{S}'$ is empty, then there are no solutions to C. Otherwise, for each constraint-set $C'_i \in \mathscr{S}'$, we transform C'_i into an equation system E_i and then construct a general solution σ_i from E_i (`Step 3`). Finally, we collect all the solutions σ_i, yielding a set Θ of solutions to C. We write $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$ if $\mathsf{Sol}_\Delta(C)$ terminates with Θ, and we call Θ the solution of the type tallying problem C.

Theorem C.45 (Soundness). *Let C be a constraint-set. If $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \Vdash C$.*

Proof. Consequence of Lemmas C.10, C.17, C.20, C.26, C.28, C.33 and C.39. □

Theorem C.46 (Completeness). *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \approx \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas C.12, C.21, C.34 and C.43. □

Theorem C.47 (Termination). *Let C be a constraint-set. Then $\mathsf{Sol}_\Delta(C)$ terminates.*

Proof. Consequence of Lemmas C.14, C.22 and C.41. □

Lemma C.48. *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then*

(1) *Θ is finite.*
(2) *for all $\sigma \in \Theta$ and for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof.(1): Consequence of Lemmas C.15 and C.27.
(2): Consequence of Lemmas C.17, C.28, C.36 and C.44.
□

C.2 Type-Substitution Inference Algorithm

In Section B, we presented a sound and complete inference system, which is parametric in the decision procedures for $\sqsubseteq_\Delta$, $\amalg^i_\Delta()$, and $\bullet_\Delta$. In this section we tackle the problem of computing these operators. We focus on the application problem $\bullet_\Delta$, since the other two can be solved similarly. Recall that to compute $t \bullet_\Delta s$, we have to find two sets of substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $\forall h \in I \cup J.\ \sigma_h \sharp \Delta$ and

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0} \to \mathbb{1} \tag{18}$$

$$\bigwedge_{j\in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) \tag{19}$$

This problem is more general than the other two problems. If we are able to decide inequation (19), it means that we are able to decide $s' \sqsubseteq_\Delta t'$ for any s' and t', just by considering t' ground. Therefore we can decide $\sqsubseteq_\Delta$. We can also decide $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$ for all s, and therefore compute $\amalg^i_\Delta(s)$.

Let the cardinalities of I and J be p and q respectively. We first show that for fixed p and q, we can reduce the application problem to a type tallying problem. Note that if we increase p, the type on the right of Inequality (19) is larger, and if we increase q the type on the left is smaller. Namely, the larger p and q are, the higher the chances that the inequality holds. Therefore, we can search for cardinalities that make the inequality hold by starting from $p = q = 1$, and then by increasing p and q in a dove-tail order until we get a solution. This gives us a semi-decision procedure for the general application problem. In order to ensure termination, we give some heuristics based on the shapes of s and t to set upper bounds for p and q.

C.2.1 Application problem with fixed cardinalities

We explain how to reduce the application problem with fixed cardinalities for I and J to a type tallying problem. Without loss of generality, we can split each substitution σ_k ($k \in I \cup J$) into two substitutions: a renaming substitution ρ_k that maps each variable in the domain of σ_k into a fresh variable and a second substitution σ'_k such that $\sigma_k = \sigma'_k \circ \rho_k$. The two inequalities then can be rewritten as

$$\bigwedge_{i\in I} (t\rho_i)\sigma'_i \leq \mathbb{0} \to \mathbb{1}$$

$$\bigwedge_{j\in J} (s\rho_j)\sigma'_j \leq \mathsf{dom}(\bigwedge_{i\in I} (t\rho_i)\sigma'_i)$$

The domains of the substitutions σ'_k are pairwise distinct, since they are composed by fresh type variables. We can therefore merge the σ'_k into one substitution $\sigma = \bigcup_{k\in I\cup J} \sigma'_k$. We can then further rewrite the two inequalities as

$$(\bigwedge_{i\in I} (t\rho_i))\sigma \leq \mathbb{0} \to \mathbb{1}$$

$$(\bigwedge_{j\in J} (s\rho_j))\sigma \leq \mathsf{dom}((\bigwedge_{i\in I} (t\rho_i))\sigma)$$

which are equivalent to

$$t'\sigma \leq \mathbb{0} \to \mathbb{1}$$
$$s'\sigma \leq \mathsf{dom}(t'\sigma)$$

where $t' = (\bigwedge_{i\in I} t\rho_i)$ and $s' = (\bigwedge_{j\in J} s\rho_j)$. As $t'\sigma \leq \mathbb{0} \to \mathbb{1}$, then $t'\sigma$ must be a function type. Then according to Lemmas C.12 and C.13 in the companion paper [3], we can reduce these two inequalities to the constraint set[13]:

$$C = \{(t', \leq, \mathbb{0} \to \mathbb{1}), (t', \leq, s' \to \gamma)\}$$

where γ is a fresh type variable. We have reduced the original application problem $t \bullet_\Delta s$ to solving C, which can be done as explained in Section C.1. We write $\mathsf{AppFix}_\Delta(t, s)$ for the algorithm of the application problem with fixed cardinalities $t \bullet_\Delta s$ and $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$ if $\mathsf{AppFix}_\Delta(t, s)$ terminates with Θ.

Lemma C.49. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathit{var}(t) \cup \mathit{var}(s)$. Then for all substitution σ, if $t\sigma \leq s\sigma \to \gamma\sigma$, then $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\sigma(\gamma) \geq t\sigma \cdot s\sigma$.*

[13] The first constraint $(t', \leq, \mathbb{0} \to \mathbb{1})$ can be eliminated since it is implied by the second one.

Proof. Consider any substitution σ. As $t\sigma \leq s\sigma \to \gamma\sigma$, by Lemma C.12 in the companion paper [3], we have $s\sigma \leq \mathsf{dom}(t\sigma)$. Then by Lemma C.13 in the companion paper [3], we get $\sigma(\gamma) \geq t\sigma \cdot s\sigma$. □

Lemma C.50. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathit{var}(t) \cup \mathit{var}(s)$. Then for all substitution σ, if $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\gamma \notin \mathsf{dom}(\sigma)$, then there exists σ' such that $\sigma' \sharp \sigma$ and $t(\sigma \cup \sigma') \leq (s \to \gamma)(\sigma \cup \sigma')$.*

Proof. Consider any substitution σ. As $s\sigma \leq \mathsf{dom}(t\sigma)$, by Lemma C.13 in the companion paper [3], the type $(t\sigma) \cdot (s\sigma)$ exists and $t\sigma \leq s\sigma \to ((t\sigma) \cdot (s\sigma))$. Let $\sigma' = \{{}^{(t\sigma)\cdot(s\sigma)}/_{\gamma}\}$. Then

$$\begin{array}{rcl} t(\sigma \cup \sigma') & = & t\sigma \\ & \leq & s\sigma \to ((t\sigma) \cdot (s\sigma)) \\ & = & s\sigma \to \gamma\sigma' \\ & = & (s \to \gamma)(\sigma \cup \sigma') \end{array}$$

□

Note that the solution of the γ introduced in the constraint $(t, \leq, s \to \gamma)$ represents a result type for the application of t to s. In particular, completeness for the tallying problem ensures that each solution will assign to γ (which occurs in a covariant position) the minimum type for that solution. So the minimum solutions for γ are in $t \bullet_\Delta s$ (see the substitution $\sigma'(\gamma) = (t\sigma) \cdot (s\sigma)$ in the proof of Lemma C.50).

Theorem C.51 (Soundness). *Let t and s be two types. If $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, we have $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$.*

Proof. Consequence of Lemmas C.49 and C.45. □

Theorem C.52 (Completeness). *Let t and s be two types and $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$. For all substitution σ, if $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \simeq \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas C.50 and C.46. □

C.2.2 General application problem

Now we take the cardinalities of I and J into account to solve the general application problem. We start with I and J both of cardinality 1 and explore all the possible combinations of the cardinalities of I and J by, say, a dove-tail order until we get a solution. More precisely, the algorithm consists of two steps:

Step A: we generate a constraint set as explained in Section C.2.1 and apply the tallying solving algorithm described in Section C.1, yielding either a solution or a failure.

Step B: if all attempts to solve the constraint sets have failed at `Step 1` of the tallying solving algorithm given at the beginning of Section C.1.1, then fail (the expression is not typable). If they all failed but at least one did not fail in `Step 1`, then increment the cardinalities I and J to their successor in the dove-tail order and start from `Step A` again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if the solving of the constraint-set fails at `Step 1` of the algorithm for the tallying problem. The reason is that up to `Step 1` all the constraints at issue are on distinct occurrences of type variables: if they fail there is no possible expansion that can make the constraint-set satisfiable (see Lemma C.53). For example, the function `map` can not be applied to any integer, as the normalization of $\{(\texttt{Int}, \leq, \alpha \to \beta)\}$ is empty (and even for any expansion of $\alpha \to \beta$). In `Step 2` instead constraints of different occurrences of a same variable are merged. Thus even if the constraints fail it may be the case that they will be satisfied by expanding different occurrences of a same variable into different variables. Therefore an expansion is tried. For example, consider the application of a function of type $((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})) \to t$ to an argument of type $\alpha \to \alpha$. We start with the constraint

$$(\alpha \to \alpha, \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})).$$

The tallying algorithm first normalizes it into the set

$$\{(\alpha, \leq, \texttt{Int}), (\alpha, \geq, \texttt{Int}), (\alpha, \leq, \texttt{Bool}), (\alpha, \geq, \texttt{Bool})\}\ (\texttt{Step 1}).$$

But it fails at `Step 2` as neither $\texttt{Int} \leq \texttt{Bool}$ nor $\texttt{Bool} \leq \texttt{Int}$ hold. However, if we expand $\alpha \to \alpha$, the constraint to be solved becomes

$$((\alpha_1 \to \alpha_1) \wedge (\alpha_2 \to \alpha_2), \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}))$$

and one of the constraint-set of its normalization is

$$\{(\alpha_1, \leq, \texttt{Int}), (\alpha_1, \geq, \texttt{Int}), (\alpha_2, \leq, \texttt{Bool}), (\alpha_2, \geq, \texttt{Bool})\}$$

The conflict between `Int` and `Bool` disappears and we can find a solution to the expanded constraint.

Note that we keep trying expansion without giving any bound on the cardinalities I and J, so the procedure may not terminate, which makes it only a semi-algorithm. The following lemma justifies why we do not try to expand if normalization (i.e., Step 1 of the tallying algorithm) fails.

Lemma C.53. *Let t, s be two types, γ a fresh type variable and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. If $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$.*

Proof. As $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then either $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$ or $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. If the first one holds, then according to Lemma C.19, we have $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$, and *a fortiori*

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Assume that $\emptyset \vdash_{\mathscr{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. Without loss of generality, we consider the disjunctive normal form τ of t:

$$\tau = \bigvee_{k_b\in K_b} \tau_{k_b} \vee \bigvee_{k_p\in K_p} \tau_{k_p} \vee \bigvee_{k_a\in K_a} \tau_{k_a}$$

where τ_{k_b} (τ_{k_p} and τ_{k_a} resp.) is an intersection of basic types (products and arrows resp.) and type variables. Then there must exist $k \in K_b \cup K_p \cup K_a$ such that $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$. If $k \in K_b \cup K_p$, then the constraint $(\tau_k, \leq, s \to \gamma)$ is equivalent to $(\tau_k, \leq, \mathbb{0})$. By Lemma C.19, we get $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, that is, $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows.
Otherwise, it must be that $k \in K_a$ and $\tau_k = \bigwedge_{p\in P}(w_p \to v_p) \wedge \bigwedge_{n\in N} \neg(w_n \to v_n)$. We claim that $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$ (otherwise, $\emptyset \vdash_{\mathscr{N}} \{(\tau_k, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$ does not hold). Applying Lemma C.19 again, we get $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$. Moreover, following the rule (NARROW), there exists a set $P' \subseteq P$ such that

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p\in P'} \neg w_p \wedge s, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathscr{N}} \{ \bigwedge_{p\in P\setminus P'} v_p \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

Applying C.19, we get

$$\begin{cases} \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{i\in I}(\bigwedge_{p\in P'} \neg w_p)\rho_i \wedge \bigwedge_{j\in J} s\rho_j, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathscr{N}} \{\bigwedge_{i\in I}(\bigwedge_{p\in P\setminus P'} v_p)\rho_i \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By the rule (NARROW), we have

$$\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I}(\bigwedge_{p\in P} (w_p \to v_p))\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Therefore, we have $\emptyset \vdash_{\mathscr{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows.

□

Let $\mathsf{App}_\triangle(t, s)$ denote the semi-algorithm for the general application problem.

Theorem C.54. *Let t, s be two types and γ the special fresh type variable introduced in $(\bigwedge_{i\in I} t\sigma_i, \leq, (\bigwedge_{j\in J} s\sigma_j) \to \gamma)$. If $\mathsf{App}_\triangle(t, s)$ terminates with Θ, then*

(1) **(Soundness)** *if $\Theta \neq \emptyset$, then for each $\sigma \in \Theta$, $\sigma(\gamma) \in t \bullet_\triangle s$.*
(2) **(Weak completeness)** *if $\Theta = \emptyset$, then $t \bullet_\triangle s = \emptyset$.*

Proof. (1)**:** consequence of Theorem C.51 and Lemma C.49.
(2)**:** consequence of Lemma C.53.

□

Consider the application `map` `even`, whose types are

$$\begin{array}{rcl} \texttt{map} & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \texttt{even} & :: & (\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int})) \end{array}$$

We start with the constraint-set

$$C_1 = \{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq ((\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int}))) \to \gamma\}$$

where γ is a fresh type variable (and where we α-converted the type of `map`). Then the algorithm $\mathsf{Sol}_{\Delta}(C_1)$ generates a set of eight constraint-sets at `Step 2` :

$$
\begin{array}{l}
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \mathtt{Bool}\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \alpha \setminus \mathtt{Int}\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \mathtt{Bool} \vee (\alpha \setminus \mathtt{Int})\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \mathtt{Bool} \wedge (\alpha \setminus \mathtt{Int})\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathtt{Int}, \beta_1 \geq \mathtt{Bool}\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \setminus \mathtt{Int}, \beta_1 \geq \alpha \setminus \mathtt{Int}\} \\
\{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathtt{Int} \vee \alpha, \beta_1 \geq \mathtt{Bool} \vee (\alpha \setminus \mathtt{Int})\}
\end{array}
$$

Clearly, the solutions to the 2nd-5th constraint-sets are included in those to the first constraint-set. For the other four constraint-sets, by minimum instantiation, we can get four solutions for γ (*i.e.*, the result types of `map even`): $[\,] \to [\,]$, or $[\mathtt{Int}] \to [\mathtt{Bool}]$, or $[\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]$, or $[\mathtt{Int} \vee \alpha] \to [\mathtt{Bool} \vee (\alpha \setminus \mathtt{Int})]$. Of these solutions only the last two are minimal (the first type is an instance of the third one and the second is an instance of the fourth one) and since both are valid we can take their intersection, yielding the (minimum) solution

$$([\alpha \setminus \mathtt{Int}] \to [\alpha \setminus \mathtt{Int}]) \wedge ([\mathtt{Int} \vee \alpha] \to [\mathtt{Bool} \vee (\alpha \setminus \mathtt{Int})]) \tag{20}$$

Alternatively, we can dully follow the algorithm, perform an iteration, expand the type of the function, yielding the constraint-set

$$
\begin{array}{c}
\{((\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1]) \wedge ((\alpha_2 \to \beta_2) \to [\alpha_2] \to [\beta_2]) \\
\leq ((\mathtt{Int} \to \mathtt{Bool}) \wedge ((\alpha \setminus \mathtt{Int}) \to (\alpha \setminus \mathtt{Int}))) \to \gamma\}
\end{array}
$$

from which we get the type (20) directly.

As stated in Section C.1, we chose an arbitrary ordering on type variables, which affects the generated substitutions and then the resulting types. Assume that σ_1 and σ_2 are two type substitutions generated by different orders. Thanks to the completeness of the tallying problem, there exist σ'_1 and σ'_2 such that $\sigma_2 \simeq \sigma'_1 \circ \sigma_1$ and $\sigma_1 \simeq \sigma'_2 \circ \sigma_2$. Therefore, the result types corresponding to σ_1 and σ_2 are equivalent under $\sqsubseteq_{\Delta}$, that is $\sigma_1(\gamma) \sqsubseteq_{\Delta} \sigma_2(\gamma)$ and $\sigma_2(\gamma) \sqsubseteq_{\Delta} \sigma_1(\gamma)$. However, this does not imply that $\sigma_1(\gamma) \simeq \sigma_2(\gamma)$. For example, $\alpha \sqsubseteq_{\Delta} \mathbb{0}$ and $\mathbb{0} \sqsubseteq_{\Delta} \alpha$, but $\alpha \not\simeq \mathbb{0}$. Moreover, some result types are easier to understand or more precise than some others. Which one is better is a language design and implementation problem[14]. For example, consider the `map even` again. The type (20) is obtained under the ordering $o(\alpha_1) < o(\beta_1) < o(\alpha)$. While under the ordering $o(\alpha) < o(\alpha_1) < o(\beta_1)$, we would instead get

$$([\beta \setminus \mathtt{Int}] \to [\beta]) \wedge ([\mathtt{Int} \vee \mathtt{Bool} \vee \beta] \to [\mathtt{Bool} \vee \beta]) \tag{21}$$

It is clear that (20) $\sqsubseteq_{\emptyset}$ (21) and (21) $\sqsubseteq_{\emptyset}$ (20). However, compared with (20), (21) is less precise and less comprehensible, if we look at the type $[\mathtt{Int} \vee \mathtt{Bool} \vee \beta] \to [\mathtt{Bool} \vee \beta]$: (1) there is a `Bool` in the domain which is useless here and (2) we know that `Int` cannot appear in the returned list, but this is not expressed in the type.

There is a final word on completeness, which states that for every solution of the application problem, our algorithm finds a solution that is more general. However this solution is not necessarily the first one found by the algorithm: even if we find a solution, continuing with a further expansion may yield a more general solution. We have just seen that, in the case of `map even`, the good solution is the second one, although this solution could have already been deduced by intersecting the first minimal solutions we found. Another simple example is the case of the application of a function of type $(\alpha \times \beta) \to (\beta \times \alpha)$ to an argument of type $(\mathtt{Int}\times\mathtt{Bool}) \vee (\mathtt{Bool}\times\mathtt{Int})$. For this application our algorithm returns after one iteration the type $(\mathtt{Int}\vee\mathtt{Bool}) \times (\mathtt{Int}\vee\mathtt{Bool})$ (since it unifies α with β) while one further iteration allows the system to deduce the more precise type $(\mathtt{Int}\times\mathtt{Bool})\vee(\mathtt{Bool}\times\mathtt{Int})$. Of course this raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist? This is a problem we did not tackle in this work, but if the answer to the previous question were negative then it would be easy to prove the existence of a principal type: since at each iteration there are only finitely many solutions, then the principal type would be the intersection of the minimal solutions of the last iteration (how to decide that an iteration is the last one is yet another problem).

C.2.3 Heuristics to stop type-substitution inference

We only have a semi-algorithm for $t \bullet_{\Delta} s$ because, as long as we do not find a solution, we may increase the cardinalities of I and J (where I and J are defined as in the previous sections) indefinitely. In this section, we propose two heuristic numbers p and q for the cardinalities of I and J that are established according to the form of s and t. These heuristic numbers set the upper limit for the procedure: if no solution is found when the cardinalities of I and J have reached these heuristic numbers, then the procedure stops returning failure. This yields a terminating algorithm for $t \bullet_{\Delta} s$ which is clearly sound but, in our case, not complete. Whether it is possible to define these boundaries so that they ensure termination *and* completeness is still an open issue.

[14] In the current implementation we assume that the type variables in the function type always have smaller orders than those in the argument type.

Through some examples, we first analyze the reasons why one needs to expand the function type t and/or the argument type s: the intuition is that type connectives are what makes the expansions necessary. Then based on this analysis, we give some heuristic numbers for the copies of types that are needed by the expansions. These heuristics follow some simple (but, we believe, reasonable) guidelines. First, when the substitutions found for a given p and q yield a useless type (*e.g.*, "$\mathbb{0} \to \mathbb{0}$" the type of a function that cannot be applied to any value), it seems sensible to expand the types (*i.e.*, increase p or q), in order to find more informative substitutions. Second, if iterating the process does not give a more precise type (in the sense of $\sqsubseteq$), then it seems sensible to stop. Last, when the process continuously yields more and more precise types, we choose to stop when the type is "good enough" for the programmer. In particular we choose to avoid to introduce too many new fresh variables that make the type arbitrarily more precise but at the same time less "programmer friendly". We illustrate these behaviours for three strategies: increasing p (that is, expanding the domain of the function), increasing q (that is, expanding the type of the argument) or lastly increasing both p and q at the same time.

Expansion of t. A simple reason to expand t is the presence of (top-level) unions in s. Generally, it is better to have as many copies of t as there are disjunctions in s. Consider the example,

$$\begin{array}{l} t = (\alpha \to \alpha) \to (\alpha \to \alpha) \\ s = (\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool}) \end{array} \tag{22}$$

If we do not expand t (*ie*, if p is 1), then the result type computed for the application of t to s is $\mathbb{0} \to \mathbb{0}$. However, this result type cannot be applied hereafter, since its domain is $\mathbb{0}$, and is therefore useless (more precisely, it can be applied only to expressions that are provably diverging). When p is 2, we get an extra result type, $(\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool})$, which is obtained by instantiating t twice, by `Int` and `Bool` respectively. Carrying on expanding t does not give more precise result types, as we always select only two copies of t to match the two summands in s, according to the decomposition rule for arrows [4].

A different example that shows that the cardinality of the summands in the union type of the argument is a good heuristic choice for p is the following one:

$$\begin{array}{l} t = (\alpha \times \beta) \to (\beta \times \alpha) \\ s = (\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int}) \end{array} \tag{23}$$

Without expansion, the result type is $((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Bool} \vee \texttt{Int}))$ (α unifies `Int` and `Bool`). If we expand t, there exists a more precise result type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$, each summand of which corresponds to a different summand in s. Besides, due to the decomposition rule for product types [4], there also exist some other result types which involve type variables, like $((\texttt{Int} \vee \texttt{Bool}) \times \alpha) \vee ((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Int} \vee \texttt{Bool}) \setminus \alpha)$. Further expanding t makes more product decompositions possible, which may in turn generate new result types. However, the type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$ is informative enough, and so we set the heuristic number to 2, that is, the number of summands in s.

We may have to expand t also because of intersection. First, suppose s is an intersection of basic types; it can be viewed as a single basic type. Consider the example

$$t = \alpha \to (\alpha \times \alpha) \text{ and } s = \texttt{Int} \tag{24}$$

Without expansion, the result type is $\gamma_1 = (\texttt{Int} \times \texttt{Int})$. With two copies of t, besides γ_1, we get another result type $\gamma_2 = (\beta \times \beta) \vee (\texttt{Int} \setminus \beta \times \texttt{Int} \setminus \beta)$, which is more general than γ_1 (*eg*, $\gamma_1 = \gamma_2\{^{\mathbb{0}}/_{\beta}\}$). Generally, with k copies, we get k result types of the form

$$\gamma_k = (\beta_1 \times \beta_1) \vee \ldots \vee (\beta_{k-1} \times \beta_{k-1}) \vee (\texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i) \times \texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i))$$

It is clear that $\gamma_{k+1} \sqsubseteq_\emptyset \gamma_k$. Moreover, it is easy to find two substitutions $[\sigma_1, \sigma_2]$ such that $[\sigma_1, \sigma_2] \Vdash \gamma_k \sqsubseteq_\emptyset \gamma_{k+1}$ ($k \geq 2$). Therefore, γ_2 is the minimum (with respect to $\sqsubseteq_\emptyset$) of $\{\gamma_k, k \geq 1\}$, so expanding t more than once is useless (we do not get a type more precise than γ_2). However, we think the programmer expects $(\texttt{Int} \times \texttt{Int})$ as a result type instead of γ_2. So we take the heuristic number here as 1.

An intersection of product types is equivalent to $\bigvee_{i \in I}(s_1^i \times s_2^i)$, so we consider just a single product type (and then use union for the general case). For instance,

$$\begin{array}{l} t = ((\alpha \to \alpha) \times (\beta \to \beta)) \to ((\beta \to \beta) \times (\alpha \to \alpha)) \\ s = (((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})) \times (\texttt{Bool} \to \texttt{Bool})) \end{array} \tag{25}$$

For the application to succeed, we have a constraint generated for each component of the product type, namely $(\alpha \to \alpha \geq (\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd}))$ and $(\beta \to \beta \geq \texttt{Bool} \to \texttt{Bool})$. As with Example (22), it is better to expand $\alpha \to \alpha$ once for the first constraint, while there is no need to expand $\beta \to \beta$ for the second one. As a result, we expand the whole type t once, and get the result type $((\texttt{Bool} \to \texttt{Bool}) \times ((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})))$ as expected. Generally, if the heuristic numbers of the components of a product type are respectively p_1 and p_2, we take $p_1 * p_2$ as the heuristic number for the whole product.

Finally, suppose s is an intersection of arrows, like for example `map even`.

$$\begin{array}{l} t = (\alpha \to \beta) \to [\alpha] \to [\beta] \\ s = (\texttt{Int} \to \texttt{Bool}) \wedge ((\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})) \end{array} \tag{26}$$

When $p = 1$, the constraint to solve is $(\alpha \to \beta \geq s)$. As stated in Subsection C.2.2, we get four possible result types: $[\,] \to [\,]$, $[\texttt{Int}] \to [\texttt{Bool}]$, $[\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]$, or $[\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]$, and

Table 1. Heuristic number $H_p(s)$ for the copies of t

Shape of s	**Number** $H_p(s)$
$\bigvee_{i\in I} s_i$	$\Sigma_{i\in I} H_p(s_i)$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(s_i^1 \times s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \times s_i^2)$	$\Sigma_{N'\subseteq N} H_p(s^1_{N'} \times s^2_{N'})$
$(s_1 \times s_2)$	$H_p(s_1) * H_p(s_2)$
$\bigwedge_{i\in P}(s_i^1 \to s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \to s_i^2)$	1

where $(s^1_{N'} \times s^2_{N'}) = (\bigwedge_{i\in P} s_i^1 \wedge \bigwedge_{i\in N'} \neg s_i^1 \times \bigwedge_{i\in P} s_i^2 \wedge \bigwedge_{i\in N\setminus N'} \neg s_i^2)$.

we can build the minimum one by taking the intersection of them. If we continue expanding t, any result type we obtain is an intersection of some of the result types we have deduced for $p = 1$. Indeed, assume we expand t so that we get p copies of t. Then we would have to solve either $(\bigvee_{i=1..p} \alpha_i \to \beta_i \geq s)$ or $(\bigwedge_{i=1..p} \alpha_i \to \beta_i \geq s)$. For the first constraint to hold, by the decomposition rule of arrows, there exists i_0 such that $s \leq \alpha_{i_0} \to \beta_{i_0}$, which is the same constraint as for $p = 1$. The second constraint implies $s \leq \alpha_i \to \beta_i$ for all i; we recognize again the same constraint as for $p = 1$ (except that we intersect p copies of it). Consequently, expanding does not give us more information, and it is enough to take $p = 1$ as the heuristic number for this case.

Following the discussion above, we propose in Table 1 a heuristic number $H_p(s)$ that, according to the shape of s, sets an upper bound to the number of copies of t. We assume that s is in normal form. This definition can be easily extended to recursive types by memoization.

The next example shows that performing the expansion of t with $H_p(s)$ copies may not be enough to get a result type, confirming that this number is a heuristic that does not ensure completeness. Let

$$\begin{array}{l} t = ((\mathtt{true} \times (\mathtt{Int} \to \alpha)) \to t_1) \wedge ((\mathtt{false} \times (\alpha \to \mathtt{Bool})) \to t_2) \\ s = (\mathtt{Bool} \times (\mathtt{Int} \to \mathtt{Bool})) \end{array} \tag{27}$$

Here $\mathsf{dom}(t)$ is $(\mathtt{true} \times (\mathtt{Int} \to \alpha)) \vee (\mathtt{false} \times (\alpha \to \mathtt{Bool}))$. The type s cannot be completely contained in either summand of $\mathsf{dom}(t)$, but it can be contained in $\mathsf{dom}(t)$. Indeed, the first summand requires the substitution of α to be a supertype of `Bool` while the second one requires it to be a subtype of `Int`. As `Bool` is not a subtype of `Int`, to make the application possible, we have to expand the function type at least once. However, according to Table 1, the heuristic number in this case is 1 (*ie*, no expansions).

Expansion of s**.** For simplicity, we assume that $\mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) = \bigvee_{i\in I} \mathsf{dom}(t)\sigma_i$, so that the tallying problem for the application becomes $\bigwedge_{j\in J} s\sigma'_j \leq \bigvee_{i\in I} \mathsf{dom}(t)\sigma_i$. We now give some heuristic numbers for $|J|$ depending on $\mathsf{dom}(t)$.

First, consider the following example where $\mathsf{dom}(t)$ is a union:

$$\begin{array}{rl} \mathsf{dom}(t) = & (\mathtt{Int} \to ((\mathtt{Bool} \to \mathtt{Bool}) \wedge (\mathtt{Int} \to \mathtt{Int}))) \\ & \vee(\mathtt{Bool} \to ((\mathtt{Bool} \to \mathtt{Bool}) \wedge (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Real} \to \mathtt{Real}))) \\ s = & (\mathtt{Int} \to (\alpha \to \alpha)) \vee (\mathtt{Bool} \to (\beta \to \beta)) \end{array} \tag{28}$$

For the application to succeed, we need to expand $\mathtt{Int} \to (\alpha \to \alpha)$ with two copies (so that we can make two distinct instantiations $\alpha = \mathtt{Bool}$ and $\alpha = \mathtt{Int}$) and $\mathtt{Bool} \to (\beta \to \beta)$ with three copies (for three instantiations $\beta = \mathtt{Bool}$, $\beta = \mathtt{Int}$, and $\beta = \mathtt{Real}$), corresponding to the first and the second summand in $\mathsf{dom}(t)$ respectively. Since the expansion distributes the union over the intersections, we need to get six copies of s. In detail, we need the following six substitutions: $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Bool}\}$, $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Int}\}$, $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Real}\}$, $\{\alpha = \mathtt{Int}, \beta = \mathtt{Bool}\}$, $\{\alpha = \mathtt{Int}, \beta = \mathtt{Int}\}$, and $\{\alpha = \mathtt{Int}, \beta = \mathtt{Real}\}$, which are the Cartesian products of the substitutions for α and β.

If $\mathsf{dom}(t)$ is an intersection of basic types, we use 1 for the heuristic number. If it is an intersection of product types, we can rewrite it as a union of products and we only need to consider the case of just a single product type. For instance,

$$\begin{array}{rl} \mathsf{dom}(t) = & ((\mathtt{Int} \to \mathtt{Int}) \times (\mathtt{Bool} \to \mathtt{Bool})) \\ s = & ((\alpha \to \alpha) \times (\alpha \to \alpha)) \end{array} \tag{29}$$

It is easy to infer that the substitution required by the left component needs α to be `Int`, while the one required by the right component needs α to be `Bool`. Thus, we need to expand s at least once. Assume that $s = (s_1 \times s_2)$ and we need q_i copies of s_i with the type substitutions: $\sigma^i_1, \ldots, \sigma^i_{q_i}$. Generally, we can expand the whole product type so that we get $s_1 \times s_2$ copies as follows:

$$\begin{array}{rl} & \bigwedge_{j=1..q_1}(s_1 \times s_2)\sigma^1_j \wedge \bigwedge_{j=1..q_2}(s_1 \times s_2)\sigma^2_j \\ = & ((\bigwedge_{j=1..q_1} s_1\sigma^1_j \wedge \bigwedge_{j=1..q_2} s_1\sigma^2_j) \times (\bigwedge_{j=1..q_1} s_2\sigma^1_j \wedge \bigwedge_{j=1..q_2} s_2\sigma^2_j)) \end{array}$$

Clearly, this expansion type is a subtype of $(\bigwedge_{j=1..q_1} s_1\sigma^1_j \times \bigwedge_{j=1..q_2} s_2\sigma^2_j)$ and so the type tallying succeeds.

Next, consider the case where $\mathsf{dom}(t)$ is an intersection of arrows:

$$\begin{array}{rl} \mathsf{dom}(t) = & (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool}) \\ s = & \alpha \to \alpha \end{array} \tag{30}$$

Table 2. Heuristic number $H_q(\mathsf{dom}(t))$ for the copies of s

Shape of $\mathsf{dom}(t)$	**Number** $H_q(\mathsf{dom}(t))$
$\bigvee_{i\in I} t_i$	$\prod_{i\in I} H_q(t_i)+1$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(t_i^1\times t_i^2)\wedge\bigwedge_{i\in N}\neg(t_i^1\times t_i^2)$	$\prod_{N'\subset N} H_q(t^1_{N'}\times t^2_{N'})$
$(t_1\times t_2)$	$H_q(t_1)+H_q(t_2)$
$\bigwedge_{i\in P}(t_i^1\to t_i^2)\wedge\bigwedge_{i\in N}\neg(t_i^1\to t_i^2)$	$\lvert P\rvert * (H_q(t_i^1)+H_q(t_i^2))$

where $(t^1_{N'}\times t^2_{N'}) = (\bigwedge_{i\in P} t_i^1 \wedge \bigwedge_{i\in N'} \neg t_i^1 \times \bigwedge_{i\in P} t_i^2 \wedge \bigwedge_{i\in N\setminus N'} \neg t_i^2)$,

Without expansion, we need $(\alpha\to\alpha)\le(\mathtt{Int}\to\mathtt{Int})$ and $(\alpha\to\alpha)\le(\mathtt{Bool}\to\mathtt{Bool})$, which reduce to $\alpha=\mathtt{Int}$ and $\alpha=\mathtt{Bool}$; this is impossible. Thus, we have to expand s once, for the two conjunctions in $\mathsf{dom}(t)$.

Note that we may also have to expand s because of unions or intersections occurring under arrows. For example,

$$\begin{array}{rl}\mathsf{dom}(t) = & t'\to((\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}))\\ s = & t'\to(\alpha\to\alpha)\end{array}\qquad(31)$$

As in Example (30), expanding once the type $\alpha\to\alpha$ (which is under an arrow in s) makes type tallying succeed. Because $(t'\to s_1)\wedge(t'\to s_2)\simeq t'\to(s_1\wedge s_2)$, we can in fact perform the expansion on s and then use subsumption to obtain the desired result. Likewise, we may have to expand s if $\mathsf{dom}(t)$ is an arrow type and contains an union in its domain. Therefore, we have to look into $\mathsf{dom}(t)$ and s deeply if they contain both arrow types.

Following these intuitions, we define in Table 2 a heuristic number $H_q(\mathsf{dom}(t))$ that, according to the sharp of $\mathsf{dom}(t)$, sets an upper bound to the number of copies of s.

Together. Up to now, we have considered the expansions of t and s separately. However, it might be the case that the expansions of t and s are interdependent, namely, the expansion of t causes the expansion of s and vice versa. Here we informally discuss the relationship between the two, and hint as why decidability is difficult to prove.

Let $\mathsf{dom}(t)=t_1\vee t_2$, $s=s_1\vee s_2$, and suppose the type tallying between $\mathsf{dom}(t)$ and s requires that $t_i\sigma_i\ge s_i$, where σ_1 and σ_2 are two conflicting type substitutions. Then we can simply expand $\mathsf{dom}(t)$ with σ_1 and σ_2, yielding $t_1\sigma_1\vee t_2\sigma_1\vee t_1\sigma_2\vee t_2\sigma_2$. Clearly, this expansion type is a supertype of $t_1\sigma_1\vee t_2\sigma_2$ and thus a supertype of s. Note that as t is on the bigger side of $\le$, then the extra chunk of type brought by the expansion (*i.e.*, $t_2\sigma_1\vee t_1\sigma_2$) does not matter. That is to say, the expansion of t would not cause the expansion of s.

However, the expansion of s could cause the expansion of t, and even a further expansion of s itself. Assume that $s=s_1\vee s_2$ and s_i requires a different substitution σ_i (*i.e.*, $s_i\sigma_i\le\mathsf{dom}(t)$ and σ_1 is in conflict with σ_2). If we expand s with σ_1 and σ_2, then we have

$$\begin{array}{rl} & (s_1\vee s_2)\sigma_1\wedge(s_1\vee s_2)\sigma_2\\ = & (s_1\sigma_1\wedge s_1\sigma_2)\vee(s_1\sigma_1\wedge s_2\sigma_2)\vee(s_2\sigma_1\wedge s_1\sigma_2)\vee(s_2\sigma_1\wedge s_2\sigma_2)\end{array}$$

It is clear that $s_1\sigma_1\wedge s_1\sigma_2$, $s_1\sigma_1\wedge s_2\sigma_2$ and $s_2\sigma_1\wedge s_2\sigma_2$ are subtypes of $\mathsf{dom}(t)$. Consider the extra type $s_1\sigma_2\wedge s_2\sigma_1$. If this extra type is empty (e.g., because s_1 and s_2 have different top-level constructors), or if it is a subtype of $\mathsf{dom}(t)$, then the type tallying succeeds. Otherwise, in some sense, we need to solve another type tallying between $s\wedge(s_2\sigma_1\wedge s_1\sigma_2)$ and $\mathsf{dom}(t)$, which would cause the expansion of t or s. This is the main reason why we fail to prove the decidability of the application problem (that is, deciding $\bullet_\Delta$) so far.

To illustrate this phenomenon, consider the following example:

$$\begin{array}{rl}\mathsf{dom}(t) = & ((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Int}\to\mathtt{Int}))\\ & \vee((\mathtt{Bool}\to\mathtt{Bool})\vee(\mathtt{Int}\to\mathtt{Int}))\to((\beta\to\beta)\vee(\mathtt{Bool}\to\mathtt{Bool}))\\ & \vee(\beta\times\beta)\\ s = & (\alpha\to(\mathtt{Int}\to\mathtt{Int}))\vee((\mathtt{Bool}\to\mathtt{Bool})\to\alpha)\vee(\mathtt{Bool}\times\mathtt{Bool})\end{array}\qquad(32)$$

Let us consider each summand in s respectively. A solution for the first summand is $\alpha\ge\mathtt{Bool}\to\mathtt{Bool}$, which corresponds to the first summand in $\mathsf{dom}(t)$. The second one requires $\alpha\le\mathtt{Int}\to\mathtt{Int}$ and the third one $\beta\ge\mathtt{Bool}$. Since $(\mathtt{Bool}\to\mathtt{Bool})$ is not subtype of $(\mathtt{Int}\to\mathtt{Int})$, we need to expand s once, that is,

$$\begin{array}{rl} s' = & s\{{}^{\mathtt{Bool}\to\mathtt{Bool}}/_{\alpha}\}\wedge s\{{}^{\mathtt{Int}\to\mathtt{Int}}/_{\alpha}\}\\ = & ((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Int}\to\mathtt{Int}))\wedge((\mathtt{Int}\to\mathtt{Int})\to(\mathtt{Int}\to\mathtt{Int}))\\ & \vee((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Int}\to\mathtt{Int}))\wedge((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Int}\to\mathtt{Int}))\\ & \vee((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Bool}\to\mathtt{Bool}))\wedge((\mathtt{Int}\to\mathtt{Int})\to(\mathtt{Int}\to\mathtt{Int}))\\ & \vee((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Bool}\to\mathtt{Bool}))\wedge((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Int}\to\mathtt{Int}))\\ & \vee(\mathtt{Bool}\times\mathtt{Bool})\end{array}$$

Almost all the summands of s' are contained in $\mathsf{dom}(t)$ except the extra type

$$((\mathtt{Bool}\to\mathtt{Bool})\to(\mathtt{Bool}\to\mathtt{Bool}))\wedge((\mathtt{Int}\to\mathtt{Int})\to(\mathtt{Int}\to\mathtt{Int}))$$

Therefore, we need to consider another type tallying involving this extra type and $\mathsf{dom}(t)$. By doing so, we obtain $\beta = \texttt{Int}$; however we have inferred before that β should be a supertype of $\texttt{Bool}$. Consequently, we need to expand $\mathsf{dom}(t)$; the expansion of $\mathsf{dom}(t)$ with $\{{}^{\texttt{Bool}}\!/\!_{\beta}\}$ and $\{{}^{\texttt{Int}}\!/\!_{\beta}\}$ makes the type tallying succeed.

In day-to-day examples, the extra type brought by the expansion of s is always a subtype of (the expansion type of) $\mathsf{dom}(t)$, and we do not have to expand $\mathsf{dom}(t)$ or s again. The heuristic numbers we gave seem to be enough in practice.

D. Type reconstruction

We define an *implicit* calculus without interfaces, for which we define a reconstruction system.

Definition D.1. *An* implicit expression m *is an expression without any interfaces (or type substitutions). It is inductively generated by the following grammar:*

$$m ::= c \mid x \mid (m,m) \mid \pi_i(m) \mid m\,m \mid \lambda x.m \mid m{\in}t\,?\,m : m$$

The type reconstruction for expressions has the form $\Gamma \vdash_{\mathscr{R}} e : t \rightsquigarrow \mathscr{S}$, which states that under the typing environment Γ, e has type t if there exists at least one constraint-set C in the set of constraint-sets $\mathscr{S}$ such that C are satisfied. The type reconstruction rules are given in Figure 11.

$$\frac{}{\Gamma \vdash_{\mathscr{R}} c : b_c \rightsquigarrow \{\varnothing\}}\,(\textsc{Recon-const}) \qquad \frac{}{\Gamma \vdash_{\mathscr{R}} x : \Gamma(x) \rightsquigarrow \{\varnothing\}}\,(\textsc{Recon-var})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_1, m_2) : t_1 \times t_2 \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2}\,(\textsc{Recon-pair})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}}{\Gamma \vdash_{\mathscr{R}} \pi_i(m) : \alpha_i \rightsquigarrow \mathscr{S} \sqcap \{\{(t, \leq, \alpha_1 \times \alpha_2)\}\}}\,(\textsc{Recon-proj})$$

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} m_1\, m_2 : \alpha \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1, \leq, t_2 \to \alpha)\}\}}\,(\textsc{Recon-appl})$$

$$\frac{\Gamma, (x : \alpha) \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}}{\Gamma \vdash_{\mathscr{R}} \lambda x.m : \alpha \to \beta \rightsquigarrow \mathscr{S} \sqcap \{\{(t, \leq, \beta)\}\}}\,(\textsc{Recon-abstr})$$

$$\frac{\begin{array}{lll}
\Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 & & (m_0 \notin \mathscr{X}) \\
\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 & & \\
\Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 & & \\
\quad \mathscr{S} = & (\mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) & \\
\qquad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t), (t_1, \leq, \alpha)\}\}) & \\
\qquad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t), (t_2, \leq, \alpha)\}\}) & \\
\qquad \sqcup & (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}) &
\end{array}}{\Gamma \vdash_{\mathscr{R}} (m_0{\in}t\,?\,m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}\,(\textsc{Recon-case})$$

$$\frac{\begin{array}{ll}
\Gamma, (x : \Gamma(x) \wedge t) \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 & \\
\Gamma, (x : \Gamma(x) \wedge \neg t) \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 & \\
\quad \mathscr{S} = & (\{\{(\Gamma(x), \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\
\qquad \sqcup & (\mathscr{S}_1 \sqcap \{\{(\Gamma(x), \leq, t), (t_1, \leq, \alpha)\}\}) \\
\qquad \sqcup & (\mathscr{S}_2 \sqcap \{\{(\Gamma(x), \leq, \neg t), (t_2, \leq, \alpha)\}\}) \\
\qquad \sqcup & (\mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(\Gamma(x), \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\})
\end{array}}{\Gamma \vdash_{\mathscr{R}} (x \in t\,?\,m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}\,(\textsc{Recon-case-var})$$

where α, α_i and β in each rule are fresh type variables.

Figure 11. Type reconstruction rules

Most of the rules, except the rules for type cases, are standard but differ from most of the type inference of other work in that they generate a set of constraint-sets rather than a single constraint-set. This is due to the type inference for type-cases. There are four possible cases for type-cases ((Recon-case)): (i) if no branch is selected, then the type t_0 inferred for the argument m_0 should be $\mathbb{0}$ (and the result type can be any type); (ii) if the first branch is selected, then the type t_0 should be a subtype of t and the result type α for the whole type-case should be a super-type of the type t_1 inferred for the first branch m_1; (iii) if the second branch is selected, then the type t_0 should be a subtype of $\neg t$ and the result type α should be a super-type of the type t_2 inferred for the second branch m_2; and (iv) both branches are selected, then the result type α should be a super-type of the union of t_1 and t_2 (note that the condition for t_0 is the one that does not satisfy (i), (ii) and (iii)). Therefore, there are four possible solutions for type-cases and thus four possible constraint-sets. Finally, the rule (RECON-CASE-VAR) deals with the type inference for the special binding type-case introduced in Appendix E in the companion paper [3].

Let m be an implicit expression such that $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. By inserting into m those types form of $\alpha \to \beta$ introduced by the derivation of $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$ for the λ-abstractions in m correspondingly, we obtain an explicit expression e for m, denoted as *insert*(m). In particular, for λ-abstraction $\lambda x.\ m$, we have

$$\textit{insert}(\lambda x.\ m) = \lambda^{\alpha\to\beta} x.\textit{insert}(m)$$

where $\alpha \to \beta$ is a fresh type introduced for $\lambda x.\ m$.

Theorem D.2 (Soundness). *Let m be an implicit expression such that $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. Then for all $C \in \mathscr{S}$ and for all σ, if $\sigma \Vdash C$, then $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m)@[\sigma] : t\sigma$.*

Proof. By induction on the derivation of $\Gamma \vdash_{\mathscr{R}} m : t \rightsquigarrow \mathscr{S}$. We proceed by a case analysis of the last rule used in the derivation.

(RECON-CONST): straightforward .

(RECON-VAR): straightforward.

(RECON-PAIR): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} (m_1, m_2) : t_1 \times t_2 \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2}$$

Since $C \in \mathscr{S}_1 \sqcap \mathscr{S}_2$, according to Definition C.4, there exists $C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $C = C_1 \cup C_2$. Thus, we have $\sigma \Vdash C_1$ and $\sigma \Vdash C_2$. By induction, we have $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_2\sigma$. By (*pair*), we get $\emptyset \mathbin{;} \Gamma\sigma \vdash (\textit{insert}(m_1)@[\sigma], \textit{insert}(m_2)@[\sigma]) : (t_1\sigma \times t_2\sigma)$, that is $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}((m_1, m_2))@[\sigma] : (t_1 \times t_2)\sigma$.

(RECON-PROJ): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m' : t' \rightsquigarrow \mathscr{S}'}{\Gamma \vdash_{\mathscr{R}} \pi_i(m') : \alpha_i \rightsquigarrow \mathscr{S}' \sqcap \{\{(t', \leq, \alpha_1 \times \alpha_2)\}\}}$$

According to Definition C.4, there exists $C' \in \mathscr{S}'$ such that $C = C' \cup \{(t', \leq, \alpha_1 \times \alpha_2)\}$. Thus, we have $\sigma \Vdash C'$ and $t'\sigma \leq (\alpha_1\sigma \times \alpha_2\sigma)$. By induction, we have $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m')@[\sigma] : t'\sigma$. By subsumption, we have $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m')@[\sigma] : (\alpha_1\sigma \times \alpha_2\sigma)$. Then by (*proj*), we get $\emptyset \mathbin{;} \Gamma\sigma \vdash (\pi_i(\textit{insert}(m')@[\sigma])) : \alpha_i\sigma$, that is $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(\pi_i(m'))@[\sigma] : \alpha_i\sigma$.

(RECON-APPL): consider the following derivation:

$$\frac{\Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \qquad \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2}{\Gamma \vdash_{\mathscr{R}} m_1\ m_2 : \alpha \rightsquigarrow \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_1, \leq, t_2 \to \alpha)\}\}}$$

According to Definition C.4, there exists $C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $C = C_1 \cup C_2 \cup \{(t_1, \leq, t_2 \to \alpha)\}$. Thus, we have $\sigma \Vdash C_1$, $\sigma \Vdash C_2$ and $t_1\sigma \leq t_2\sigma \to \alpha\sigma$. By induction, we have $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_2\sigma$. By subsumption, we can get $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_2\sigma \to \alpha\sigma$. Then by (*appl*), we get $\emptyset \mathbin{;} \Gamma\sigma \vdash (\textit{insert}(m_1)@[\sigma]\ \textit{insert}(m_2)@[\sigma]) : \alpha\sigma$, that is $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(m_1\ m_2)@[\sigma] : \alpha\sigma$.

(RECON-ABSTR): consider the following derivation:

$$\frac{\Gamma, (x : \alpha) \vdash_{\mathscr{R}} m' : t' \rightsquigarrow \mathscr{S}'}{\Gamma \vdash_{\mathscr{R}} \lambda x.m' : \alpha \to \beta \rightsquigarrow \mathscr{S}' \sqcap \{\{(t', \leq, \beta)\}\}}$$

According to Definition C.4, there exists $C' \in \mathscr{S}'$ such that $C = C' \cup \{(t', \leq, \beta)\}$. Thus, we have $\sigma \Vdash C'$ and $t'\sigma \leq \beta\sigma$. By induction, we have $\emptyset \mathbin{;} \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : t'\sigma$. By subsumption, we can get $\emptyset \mathbin{;} \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : \beta\sigma$. It is clear that there are no sub-terms form of $e[\sigma_j]_{j\in J}$ in $\textit{insert}(m')@[\sigma]$, so $\textit{insert}(m')@[\sigma] \sharp \mathsf{var}(\alpha\sigma \to \beta\sigma)$. Then according to weakening (*i.e.*, Lemma B.8 in the companion paper [3]), we have $\mathsf{var}(\alpha\sigma \to \beta\sigma) \mathbin{;} \Gamma\sigma, (x : \alpha\sigma) \vdash \textit{insert}(m')@[\sigma] : \beta\sigma$. Finally, by (*abstr*), we get $\emptyset \mathbin{;} \Gamma\sigma \vdash (\lambda^{\alpha\to\beta}_{[\sigma]} x.\ \textit{insert}(m')) : \alpha\sigma \to \beta\sigma$, that is $\emptyset \mathbin{;} \Gamma\sigma \vdash \textit{insert}(\lambda x.\ m')@[\sigma] : \alpha\sigma \to \beta\sigma$.

(RECON-CASE): consider the following derivation:

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathscr{R}} m_0 : t_0 \rightsquigarrow \mathscr{S}_0 \qquad (m_0 \notin \mathscr{X}) \\ \Gamma \vdash_{\mathscr{R}} m_1 : t_1 \rightsquigarrow \mathscr{S}_1 \\ \Gamma \vdash_{\mathscr{R}} m_2 : t_2 \rightsquigarrow \mathscr{S}_2 \\ \quad \mathscr{S} = \quad (\mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t'), (t_1, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t'), (t_2, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}) \end{array}}{\Gamma \vdash_{\mathscr{R}} (m_0{\in}t'\ ?\ m_1 : m_2) : \alpha \rightsquigarrow \mathscr{S}}$$

Since $C \in \mathscr{S}$, according to Definition C.4, there are four possible cases for C: (i) $C \in \mathscr{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}$, (ii) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \{\{(t_0, \leq, t'), (t_1, \leq, \alpha)\}\}$, (iii) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \neg t'), (t_2, \leq, \alpha)\}\}$, and (iv) $C \in \mathscr{S}_0 \sqcap \mathscr{S}_1 \sqcap \mathscr{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\}$.

Case (i)**:** there exists $C_0 \in \mathscr{S}_0$ such that $\sigma \Vdash C_0$, $t_0\sigma \leq \mathbb{0}$ and $\mathbb{0} \leq \alpha\sigma$. By induction, we have $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$. Since $t_0\sigma \leq \mathbb{0}$, we have $t_0\sigma \leq \neg t'$ and $t_0\sigma \leq t'$. Then applying the rule (*case*), we have $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t' \,?\, \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : \mathbb{0}$, that is, $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m'{\in}t' \,?\, m_1 : m_2)@[\sigma] : \mathbb{0}$. Finally, by subsumption, the result follows.

Case (ii)**:** there exists $C_0 \in \mathscr{S}_0$ and $C_1 \in \mathscr{S}_1$ such that $\sigma \Vdash C_0$, $\sigma \Vdash C_1$, $t_0\sigma \leq t'$ (t' is ground) and $t_1\sigma \leq \alpha\sigma$. By induction, we have $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$ and $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$. If $t_0\sigma \leq \neg t'$, then $t_0\sigma \leq t' \wedge (\neg t') \simeq \mathbb{0}$ (*i.e.*, Case (i)), and thus the result follows by subsumption. Otherwise, we have $t_0\sigma \leq \neg t'$. Then applying the rule (*case*), we have

$$\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t' \,?\, \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : t_1\sigma$$

that is, $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m'{\in}t' \,?\, m_1 : m_2)@[\sigma] : t_1\sigma$. Finally, by subsumption, the result follows.

Case (iii)**:** similar to Case (ii).

Case (iv)**:** there exists $C_0 \in \mathscr{S}_0$, $C_1 \in \mathscr{S}_1$ and $C_2 \in \mathscr{S}_2$ such that $\sigma \Vdash C_0$, $\sigma \Vdash C_1$, $\sigma \Vdash C_2$ and $t_1\sigma \vee t_2\sigma \leq \alpha\sigma$. By induction, we have $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_0)@[\sigma] : t_0\sigma$, $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma$ and $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_2\sigma$. By subsumption, we have $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_1)@[\sigma] : t_1\sigma \vee t_2\sigma$ and $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$. If $t_0\sigma \leq t'$ or $t_0\sigma \leq \neg t'$, then we are in Case $(i) - (iii)$, thus the result follows by subsumption. Otherwise, applying the rule (*case*), we have

$$\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m')@[\sigma]{\in}t' \,?\, \textit{insert}(m_1)@[\sigma] : \textit{insert}(m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$$

that is, $\emptyset \,⨾\, \Gamma\sigma \vdash \textit{insert}(m'{\in}t' \,?\, m_1 : m_2)@[\sigma] : t_1\sigma \vee t_2\sigma$. Finally, by subsumption, the result follows.

(RECON-CASE-VAR)**:** similar to (RECON-CASE).

□

Consider the implicit version of `map`, which can be defined as:

$$\mu m\ \lambda f\ .\ \lambda \ell\ .\ \ell \in \texttt{nil} \,?\, \texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell))$$

The type inferred for `map` by the type reconstruction system is $\alpha_1 \to \alpha_2$ and the generated set $\mathscr{S}$ of constraint-sets is:

$$\begin{array}{ll}\{ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{0}, \mathbb{0} \leq \alpha_5\},\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \texttt{nil}, \texttt{nil} \leq \alpha_5\},\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \neg\texttt{nil}, (\alpha_6 \times \alpha_9) \leq \alpha_5\} \cup C,\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{1}, (\alpha_6 \times \alpha_9) \vee \texttt{nil} \leq \alpha_5\} \cup C \quad \}\end{array}$$

where C is $\{\alpha_1 \leq \alpha_7 \to \alpha_6, \alpha_3 \setminus \texttt{nil} \leq (\alpha_7 \times \alpha_8), \alpha_1 \to \alpha_2 \leq \alpha_1 \to \alpha_{10}, \alpha_3 \setminus \texttt{nil} \leq (\alpha_{11} \times \alpha_{12}), \alpha_{10} \leq \alpha_{12} \to \alpha_9\}$. Then applying the tallying algorithm to the sets, we get the following types for `map`:

$$\begin{array}{l}\alpha_1 \to (\mathbb{0} \to \alpha_5)\\ \alpha_1 \to (\texttt{nil} \to \texttt{nil})\\ \mathbb{0} \to ((\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12}) \to (\alpha_6 \times \alpha_9))\\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to (\alpha_6 \times \alpha_9))\\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to [\alpha_6])\\ \mathbb{0} \to ((\texttt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12})) \to (\texttt{nil} \vee (\alpha_6 \times \alpha_9)))\\ (\alpha_7 \to \alpha_6) \to (\texttt{nil} \to (\texttt{nil} \vee (\alpha_6 \times \alpha_9)))\\ (\alpha_7 \to \alpha_6) \to ((\mu x.\, \texttt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge x)) \to [\alpha_6])\end{array}$$

By replacing type variables that only occur positively by $\mathbb{0}$ and those only occurring negatively by $\mathbb{1}$, we obtain

$$\begin{array}{l}\mathbb{1} \to (\mathbb{0} \to \mathbb{0})\\ \mathbb{1} \to (\texttt{nil} \to \texttt{nil})\\ \mathbb{0} \to ((\mathbb{1} \times \mathbb{1}) \to \mathbb{0})\\ (\mathbb{0} \to \mathbb{1}) \to (\mathbb{0} \to \mathbb{0})\\ (\mathbb{1} \to \beta) \to (\mathbb{0} \to [\beta])\\ \mathbb{0} \to ((\texttt{nil} \vee (\mathbb{1} \times \mathbb{1})) \to \texttt{nil})\\ (\mathbb{0} \to \mathbb{1}) \to (\texttt{nil} \to \texttt{nil})\\ (\alpha \to \beta) \to ([\alpha] \to [\beta])\end{array}$$

All the types, except the last two, are useless [15], as they provide no further information. Thus we deduce the following type for `map`:

$$((\alpha \to \beta) \to ([\alpha] \to [\beta])) \wedge ((\mathbb{0} \to \mathbb{1}) \to (\texttt{nil} \to \texttt{nil}))$$

[15] These useless types are generated from the fact that $\mathbb{0} \to t$ contains all the functions, or the fact that $(\mathbb{0} \times t)$ or $(t \times \mathbb{0})$ is a subtype of any type, or the fact that Case (i) in type-cases is useless in practice.

which is more precise than $(\alpha \to \beta) \to ([\alpha] \to [\beta])$ since it states that the application of `map` to any function and the empty list returns the empty list.

E. Application to ℂDuce

We give a rough overview of the modifications that are necessary in order to transpose the algorithms and the results of this work to the implementation of the polymorphic extension of ℂDuce. In particular, we show how to generalize the static and dynamic semantics of explicit type-case expressions of this work to ℂDuce's pattern matching expressions. Details about the syntax and semantics of ℂDuce can be found in [2] or, better, in the online documentation available at `www.cduce.org`.

E.1 Intermediate language

The ℂDuce compiler includes three different languages (source, typed, and lambda) that are refined one into the other in different passes of the compiler. The first language corresponds to parsed ℂDuce expressions the last is closer to ℂDuce bytecode.

The source language is defined in the module *ast.ml* of the ℂDuce's source distribution. It is the representation of the source code.

$$a ::= c \mid x \mid aa \mid (a,a) \mid \lambda^t p.a \mid \texttt{match}\ a\ \texttt{with}\ p \to a \mid p \to a \tag{33}$$

The source language is composed of variables, constants, tuples, application of two expressions, lambda abstractions where p is a pattern, and match expressions. Patterns are defined as follows

$$p ::= t \mid (p,p) \mid p\&p \mid p|p \mid (x := c) \mid x$$

with types, tuples of patterns, intersection, union, constants, and capture variables (plus recursive patterns here omitted).

The typed language is the result of the type inference performed on the source language and it is defined in the module *typed.ml* in the source distribution of ℂDuce.

$$e ::= c \mid x \mid \underline{x} \mid ee \mid (e,e) \mid \lambda^t p.e \mid e\sigma_I \mid \texttt{match}\ e\ \texttt{with}\ p_1 : \Xi_1 \to e_1 \mid p_2 : \Xi_2 \to e_2 \tag{34}$$

The typed language is similar to the source language. The notable differences are the presence of polymorphic variables $\underline{x}$ (*cf.* Part 1,§5.3 [3]), the application of substitutions to expressions $e\sigma_I$ and the Ξ's associated to patterns in the match expressions. Each Ξ_i is a mapping from the capture variables of the pattern p_i to sets of type variable and will be used to compile away `let`-polymorphic expression variables (compiling a source expression variable into a monomorphic variable is much less expennsive in terms of run-time performance). Also since this language is the target of type-inference and we do not infer the decorations of lambdas, then $\lambda^t p.e$ stands for $\lambda^t_{\mathbb{i}} p.e$.

The typed language is transformed in the intermediate language as result of the compilation step. The evaluation language is defined in the module *lambda.ml*

$$e ::= c \mid x \mid x_\Sigma \mid ee \mid (e,e) \mid \lambda^t_\Sigma x\&p.e \mid \lambdabar^t_\Sigma x\&p.e \mid \texttt{match}\ e\ \texttt{with}\ p_1 \to e_2 \mid p_2 \to e_3 \tag{35}$$

In the intermediate (or compiled) language, (lazy) type-substitutions "Σ" are now associated to polymorphic variables and to polymorphic lambda expressions. The Ξ annotations present in the patterns of the match expressions are now removed since they were used to determine whether a variable x has to be compiled as x_Σ or just as x. The symbol $\lambdabar$ is a compiler optimization that is explained at the end of Section 5.3 in Part 1 [3]. Notice that we added in lambda abstractions an explicit variable to capture the argument of the function. This is used to compile lambda-expressions with lazy type-substitutions (in particular $\texttt{sel}(x,t,\Sigma)$ defined right below); in the actual implementation these variables are nameless and compiled by reserving a special slot in the type-environment of closures. Σ ranges over expressions that denote sets of type substitutions.

$$\Sigma ::= \sigma_I \mid \texttt{comp}(\Sigma,\Sigma') \mid \texttt{sel}(x,t,\Sigma)$$

we use $\mathbb{i}$ to denote the identity of these expressions that is the empty set of type substitutions. We use $\mathsf{dom}(\Sigma)$ to denote the domain Σ. It is inductively defined as follows:

$$\begin{array}{rcll} \mathsf{dom}([\sigma_i]_{i\in I}) & = & \bigcup_{i\in I} \mathsf{dom}(\sigma_i) & \\ \mathsf{dom}(\texttt{comp}(\Sigma,\Sigma')) & = & \mathsf{dom}(\Sigma) \cup \mathsf{dom}(\Sigma') & \text{(note: this is a rough approximation)} \\ \mathsf{dom}(\texttt{sel}(x,t,\Sigma)) & = & \mathsf{dom}(\Sigma) & \end{array}$$

Note that $\mathsf{dom}(\mathbb{i}) = \varnothing$. We use $\mathsf{var}(t)$ to denote the set of all type variables occurring in the type t.

We use two containment relations. The first $s \leq t$ is the semantic subtyping relation that states that for all substitutions s is a subtype of t. The second $s \sqsubseteq_\Delta t$ specifies that there exists a substitution for the variables not in Δ (Δ is a set containing all monomorphic variables) such that $s \leq t$. We denote the set of all polymorphic variables as $\overline{\Delta}$.

E.2 Type-directed translation

The ℂDuce compiler translates one internal language into another.

(INF-VAR-MONO)

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} x : \Gamma(x) \rhd x}\,\mathsf{var}(\Gamma(x)) \setminus \Delta = \emptyset$$

(INF-VAR-POLY)

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} x : \Gamma(x) \rhd \underline{x}}\,\mathsf{var}(\Gamma(x)) \setminus \Delta \neq \emptyset$$

(INF-MATCH)

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash_{\mathscr{I}} a : t_0 \rhd e \\ \begin{array}{l} t_0 \leq \vee_{j\in J} \llparenthesis p_j \rrparenthesis \\ t_j = (t_0 \setminus \vee_{h=1}^{j-1} \llparenthesis p_h \rrparenthesis) \wedge \llparenthesis p_j \rrparenthesis \end{array} \quad \left\{ \begin{array}{ll} s_j = \mathbb{0}\ ,\ e_j = a_j & \text{if } t_j \leq \mathbb{0} \\ \Delta \fatsemi \Gamma, t_j /\!\!/ p_j \vdash a_j : s_j \rhd e_j & \text{otherwise} \end{array}\right.\end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \mathtt{match}\ a\ \mathtt{with}\ (p_j \to a_j)_{j\in J} : \vee_{j\in J} s_j \rhd \mathtt{match}\ e\ \mathtt{with}\ (p_j : \Xi_j \to e_j)_{j\in J}}$$

$$\text{where } \Xi_j(x) = \left\{ \begin{array}{ll} \mathsf{var}((t_j /\!\!/ p_j)(x)) \setminus \Delta & \text{if } x \in \mathsf{var}(p_j) \text{ and } s_j \neq \mathbb{0} \\ \varnothing & \text{otherwise} \end{array}\right.$$

(INF-ABSTR)

$$\frac{\begin{array}{l} u_i \leq \vee_{j\in J} \llparenthesis p_j \rrparenthesis \\ t_1^i = u_i \wedge \llparenthesis p_1 \rrparenthesis \\ t_j^i = (u_i \setminus \vee_{h<j} \llparenthesis p_h \rrparenthesis) \wedge \llparenthesis p_j^i \rrparenthesis \end{array} \quad \left\{ \begin{array}{ll} s_j^i = \mathbb{0}\ ,\ e_i = a_i & \text{if } t_j^i \leq \mathbb{0} \\ \Delta' \fatsemi \Gamma, t_j^i /\!\!/ p_j \vdash_{\mathscr{I}} a_j : s_j^i \rhd e_j^i & \text{otherwise} \end{array}\right.}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} \lambda^{\wedge_{i\in I} u_i \to v_i}(p_j \to a_j)_{j\in J} : \bigwedge_{i\in I}(u_i \to v_i) \rhd \lambda^{\wedge_{i\in I} u_i \to v_i}(p_j : \cup_i \Xi_j^i \to \sqcap_i e_j^i \sigma_{H_j^i})_{j\in J}} \quad \begin{array}{l} \Delta' = \Delta \cup \mathsf{var}(\wedge_{i\in I} t_j^i \to s_j^i) \\ \sigma_{H_j} \Vdash \vee_j s_j^i \sqsubseteq_{\Delta'} v_i \\ \Xi_j^i = \mathsf{var}(\Gamma_j^i) \setminus \Delta' \\ \sigma_{H_j^i} = \left\{ \begin{array}{ll} \mathring{\imath} & \text{if } t_j^i \leq \mathbb{0} \\ \sigma_{H_j} & \text{otherwise} \end{array}\right. \end{array}$$

(INF-APPL)

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 : t \rhd e_1 \qquad \Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_2 : s \rhd e_2}{\Delta \fatsemi \Gamma \vdash_{\mathscr{I}} a_1 a_2 : (t\sigma_J) \bullet (s\sigma_I) \rhd (e_1\sigma_J)(e_2\sigma_I)} \quad \begin{array}{l} \sigma_J \Vdash \mathbb{0} \to \mathbb{1} \\ \sigma_I \Vdash s \sqsubseteq_{\Delta} \mathsf{dom}(t\sigma_J) \end{array}$$

Figure 12. Explicit Inference system for type-substitutions

The translation from the language in (33) to the language in (34) is given contextually to the typing relation. In particular we extend the typing rules in order to prove judgments of the form $\Delta, \Gamma \vdash_{\mathscr{I}} a : t \rhd e$ where a is a term of the source language (33) and e its translation in the intermediate language (34).

The type inference rules that perform the translation from (33) to the language in (34) are specified in Figure 12. The rules (INF-VAR-*) translate variables into polymorphic or monomorphic ones according to whether their type contains polymorphic type variables or not. The rule for application, (INF-APPL) simply applies the sets of type-substitutions inferred for the function and for its argument to them. The rule for `match` (INF-MATCH) is the standard $\mathbb{C}$Duce rule (see [9]) except that it stores in Ξ_j the type variables occurring in the types of each capture variable of p_j. The rule (INT-ABSTR) is standard too, except that it merges the different Ξ's and σ's found for the same branch while checking the type for different arrows of the interface. Notice that these last two rules use the standard $\mathbb{C}$Duce meta-operator "$/\!\!/$" to compute the type environment for pattern's capture variables (see [9]). Formally, let t and p be a type and a parameter such that $t \leq \llparenthesis p \rrparenthesis$. We define $(t /\!\!/ p)(x) = \{(v/p)(x) \mid v \in t\}$, that is:

$$\begin{array}{rcl}
t /\!\!/ x & = & \{x \mapsto t\} \\
t /\!\!/ t_0 & = & \{\} \\
t /\!\!/ (p_1 \& p_2) & = & \pi_1(t) /\!\!/ p_1 \cup \pi_2(t) /\!\!/ p_2 \\
(t /\!\!/ (p_1, p_2))(x) & = & \left\{ \begin{array}{ll} (\pi_1(t) /\!\!/ p_1)(x) & \text{if } x \in \mathsf{var}(p_1) \setminus \mathsf{var}(p_2) \\ (\pi_2(t) /\!\!/ p_2)(x) & \text{if } x \in \mathsf{var}(p_2) \setminus \mathsf{var}(p_1) \\ \bigcup_{(t_1,t_2)\in\pi(t)}((t_1 /\!\!/ p_1)(x), (t_2 /\!\!/ p_2)(x)) & \text{if } x \in \mathsf{var}(p_2) \cap \mathsf{var}(p_1) \end{array}\right. \\
t /\!\!/ p_1 | p_2 & = & (t \wedge \llparenthesis p_1 \rrparenthesis) /\!\!/ p_1 \cup (t \setminus \llparenthesis p_1 \rrparenthesis) /\!\!/ p_2 \\
t /\!\!/ (x := c) & = & \left\{ \begin{array}{ll} \{x \mapsto b_c\} & \text{if } t \not\leq \mathbb{0} \\ \{\} & \text{otherwise} \end{array}\right.
\end{array}$$

where the pairwise union of mappings assumes that the domains are distinct

$$(\Gamma_1 \cup \Gamma_2)(x) = \left\{ \begin{array}{ll} \Gamma_1(x) & \text{if } x \in \mathsf{dom}(\Gamma_1) \\ \Gamma_2(x) & \text{if } x \in \mathsf{dom}(\Gamma_2) \end{array}\right.$$

Finally, the compilation of the explicitly-typed language (34) into the intermediate language (35) is given by the following rules:

$$
\begin{array}{rcl}
[\![x]\!]_{\Sigma,\Xi} & = & x \\
[\![\underline{x}]\!]_{\Sigma,\Xi} & = & \left\{ \begin{array}{ll} x & \text{if } \Xi(x) \cap \mathsf{dom}(\Sigma) = \emptyset^{16} \\ x_\Sigma & \text{otherwise} \end{array} \right. \\
[\![\lambda^t p.e]\!]_{\Sigma,\Xi} & = & \left\{ \begin{array}{lll} \lambda^t_\Sigma x\&p.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma),\Xi} & \text{if } \mathsf{var}(t) \cap \mathsf{dom}(\Sigma) = \emptyset^{16} & (x \text{ fresh}) \\ \lambdabar^t_\Sigma x\&p.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma),\Xi} & \text{otherwise} & (x \text{ fresh}) \end{array} \right. \\
[\![(e_1,e_2)]\!]_{\Sigma,\Xi} & = & ([\![e_1]\!]_{\Sigma,\Xi}, [\![e_2]\!]_{\Sigma,\Xi}) \\
[\![e_1 e_2]\!]_{\Sigma,\Xi} & = & [\![e_1]\!]_{\Sigma,\Xi}[\![e_2]\!]_{\Sigma,\Xi} \\
[\![e\sigma_I]\!]_{\Sigma,\Xi} & = & [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I),\Xi} \\
[\![\mathtt{match}\ e\ \mathtt{with}\ p_1{:}\Xi_1{\to}e_1 \mid p_2{:}\Xi_2{\to}e_2]\!]_{\Sigma,\Xi} & = & \mathtt{match}\ [\![e]\!]_{\Sigma,\Xi}\ \mathtt{with}\ p_1 \to [\![e_1]\!]_{\Sigma,(\Xi\cup\Xi_1)} \mid p_2 \to [\![e_2]\!]_{\Sigma,(\Xi\cup\Xi_2)}
\end{array}
$$

These rules are mostly straightforward except that we try to compile into monomorphic expression variables as many capture variables as possible. In particular, we compile as monomorphic also those polymorphic expression variables for which we can statically determine that type substitutions will have no effect at run-time (*ie*, every variable $\underline{x}$ for which $\Xi(x) \cap \mathsf{dom}(\Sigma) = \emptyset$ holds).

E.3 Evaluation Rules

The evaluation procedure transforms the evaluation language into values of the following form :

$$v ::= c \mid (v,v) \mid (v,v)_\Sigma \mid \langle \lambda^s_\Sigma p.e, x, \mathscr{E} \rangle$$

Notice that closures now include a slot for a variable. This slot stores the fresh variables that were introduced in the translations of lambdas and it is used at the application (rule (OE-APPLY).

The operational semantics must be modified to take into account new constructions and to lazily propagate type substitutions for all constructed values.

(OE-CONST) $$\frac{}{\mathscr{E} \vdash_{\circ} c \Downarrow c}$$

(OE-CLOSURE) $$\frac{}{\mathscr{E} \vdash_{\circ} \lambda^t_\Sigma x\&p.e \Downarrow \langle \lambda^t_\Sigma p.e, x, \mathscr{E} \rangle}$$

(OE-PAIRVALUE) $$\frac{}{\mathscr{E} \vdash_{\circ} (v_1,v_2) \Downarrow (v_1,v_2)}$$

(OE-VAR) $$\frac{}{\mathscr{E} \vdash_{\circ} x \Downarrow \mathscr{E}(x)}$$

(OE-PVAR-C) $$\frac{\mathscr{E}(x) = c}{\mathscr{E} \vdash_{\circ} x_\Sigma \Downarrow c}$$

(OE-PVAR-F) $$\frac{\mathscr{E}(x) = \lambda^t_{\Sigma'} x'\&p.e}{\mathscr{E} \vdash_{\circ} x_\Sigma \Downarrow \langle \lambda^t_{\mathsf{comp}(\Sigma,\Sigma')} p.e, x', \mathscr{E} \rangle}$$

(OE-PVAR-P) $$\frac{\mathscr{E}(x) = (v_1,v_2)}{\mathscr{E} \vdash_{\circ} x_\Sigma \Downarrow (v_1,v_2)_\Sigma}$$

(OE-PAIR) $$\frac{\mathscr{E} \vdash_{\circ} e_1 \Downarrow v_1 \qquad \mathscr{E} \vdash_{\circ} e_2 \Downarrow v_2}{\mathscr{E} \vdash_{\circ} (e_1,e_2) \Downarrow (v_1,v_2)}$$

(OE-APPLY) $$\frac{\mathscr{E} \vdash_{\circ} e_1 \Downarrow \langle \lambda^t_\Sigma p.e, x, \mathscr{E}' \rangle \qquad \mathscr{E} \vdash_{\circ} e_2 \Downarrow v_0 \qquad \mathscr{E}', x \mapsto v_0 \vdash_{\circ} e \Downarrow v}{\mathscr{E} \vdash_{\circ} e_1 e_2 \Downarrow v}$$

(OE-MATCH 1) $$\frac{\mathscr{E} \vdash_{\circ} e \Downarrow v' \qquad v'/p_1 \neq \Omega \qquad \mathscr{E}, v'/p_1 \vdash_{\circ} e_1 \Downarrow v}{\mathscr{E} \vdash_{\circ} \mathtt{match}\ e\ \mathtt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \Downarrow v}$$

(OE-MATCH 2) $$\frac{\mathscr{E} \vdash_{\circ} e \Downarrow v' \qquad v'/p_1 = \Omega \qquad v'/p_2 \neq \Omega \qquad \mathscr{E}, v'/p_2 \vdash_{\circ} e_2 \Downarrow v}{\mathscr{E} \vdash_{\circ} \mathtt{match}\ e\ \mathtt{with}\ p_1 \to e_1 \mid p_2 \to e_2 \Downarrow v}$$

Pattern matching is defined as follows

$$
\begin{array}{rcl}
v/x & = & \{x \mapsto v\} \\
v/t & = & \left\{ \begin{array}{ll} \{\} & \text{if } v \in_{\circ} t \\ \Omega & \text{otherwise} \end{array} \right. \\
(v_1,v_2)/(p_1,p_2) & = & v_1/p_1 \oplus v_2/p_2 \\
(v_1,v_2)_\Sigma/(p_1,p_2) & = & v_1@\Sigma/p_1 \oplus v_2@\Sigma/p_2 \\
v/p_1\&p_2 & = & v/p_1 \oplus v/p_2 \\
v/p_1|p_2 & = & \left\{ \begin{array}{ll} v/p_1 & \text{if } v/p_1 \neq \Omega \\ v/p_2 & \text{otherwise} \end{array} \right. \\
v/(x := c) & = & \{x \mapsto c\}
\end{array}
$$

[16] or $\Sigma = \mathbb{I}$ which is a special case of the condition (since $\mathsf{dom}(\mathbb{I}) = \emptyset$) that can be checked more easily.

where the $\oplus$ operator has the following definition (γ ranges over value substitutions, *ie* mappings from expression variables to values):

$$(\gamma_1 \oplus \gamma_2)(x) = \begin{cases} \gamma_1(x) & \text{if } x \in \mathsf{dom}(\gamma_1) \setminus \mathsf{dom}(\gamma_2) \\ \gamma_2(x) & \text{if } x \in \mathsf{dom}(\gamma_2) \setminus \mathsf{dom}(\gamma_1) \\ (\gamma_1(x), \gamma_2(x)) & \text{if } x \in \mathsf{dom}(\gamma_1) \cap \mathsf{dom}(\gamma_2) \end{cases}$$

Notice that in the fourth rule of the definition of pattern matching when we deconstruct a pair that is annotated by a lazy type-substitution we do not immediately propagate the substitution to the sub-components. Instead we delay it until this substitution is needed. This is implemented by the "delay substitution" operation "@" defined as

$$(v@\Sigma) = \begin{cases} c@\Sigma & = c \\ \langle \lambda^t_{\Sigma'} p.e, x, \mathscr{E} \rangle @\Sigma & = \langle \lambda^t_{\mathsf{comp}(\Sigma,\Sigma')} p.e, x, \mathscr{E} \rangle \\ (v_1, v_2)@\Sigma & = (v_1, v_2)_\Sigma \\ (v_1, v_2)_{\Sigma'}@\Sigma & = (v_1, v_2)_{\mathsf{comp}(\Sigma,\Sigma')} \end{cases}$$

This requires a modification of the rules used to check the type of a value:

$$\begin{array}{rcl} c \in_\circ t & \Leftrightarrow & b_c \leq t \\ \langle \lambda^s_\Sigma p.e, x, \mathscr{E} \rangle \in_\circ t & \Leftrightarrow & s \leq t \\ \langle \lambda^s_\Sigma p.e, x, \mathscr{E} \rangle \in_\circ t & \Leftrightarrow & s(\mathsf{eval}(\mathscr{E}, \Sigma)) \leq t \\ (v_1, v_2) \in_\circ t & \Leftrightarrow & v_i \in_\circ \pi_i(t), i \in 1, 2 \\ (v_1, v_2)_\Sigma \in_\circ t & \Leftrightarrow & v_i@\Sigma \in_\circ \pi_i(t), i \in 1, 2 \end{array}$$

where, we recall, the evaluation of the symbolic set of type-substitutions is inductively defined as

$$\begin{array}{c} \mathsf{eval}(\mathscr{E}, \sigma_I) = \sigma_I \\ \mathsf{eval}(\mathscr{E}, \mathsf{comp}(\Sigma, \Sigma')) = \mathsf{eval}(\mathscr{E}, \Sigma) \circ \mathsf{eval}(\mathscr{E}, \Sigma') \\ \mathsf{eval}(\mathscr{E}, \mathtt{sel}(x, \bigwedge_{i \in I} t_i \to s_i, \Sigma)) = [\ \sigma_j \in \mathsf{eval}(\mathscr{E}, \Sigma) \mid \exists i{\in}I : \mathscr{E}(x) \in_\circ t_i\sigma_j\ ] \end{array}$$

F. Experiments

To gauge the practicality of our local type inference algorithm, we performed extensive experiments, applying higher-order polymorphic function. To that end, we automatically generated function applications from the List module of the OCaml standard distribution. More specifically we considered the following functions:

```
val length : 'a list -> int
val hd : 'a list -> 'a
val tl : 'a list -> 'a list
val nth : 'a list -> int -> 'a
val rev : 'a list -> 'a list
val append : 'a list -> 'a list -> 'a list
val rev_append : 'a list -> 'a list -> 'a list
val concat : 'a list list -> 'a list
val flatten : 'a list list -> 'a list
val iter : ('a -> unit) -> 'a list -> unit
val iteri : (int -> 'a -> unit) -> 'a list -> unit
val map : ('a -> 'b) -> 'a list -> 'b list
val mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val mem : 'a -> 'a list -> bool
val memq : 'a -> 'a list -> bool
val find : ('a -> bool) -> 'a list -> 'a
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
val assoc : 'a -> ('a * 'b) list -> 'b
val assq : 'a -> ('a * 'b) list -> 'b
val mem_assoc : 'a -> ('a * 'b) list -> bool
```

```
    val mem_assq : 'a -> ('a * 'b) list -> bool
    val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
    val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
    val split : ('a * 'b) list -> 'a list * 'b list
    val combine : 'a list -> 'b list -> ('a * 'b) list
    val sort : ('a -> 'a -> int) -> 'a list -> 'a list
    val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
    val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
    val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list
```

We then devised a series of tests as follows. First, we generated all the applications that were well typed in OCaml from one function against all the others. This gave, for instance, applications such as "`map length`" or "`map hd`", that is performing local type inference for the applications

$$(\alpha{\to}\beta){\to}[\alpha]{\to}[\beta] \bullet\_\, \alpha{\to}\texttt{Int}$$

and

$$(\alpha{\to}\beta){\to}[\alpha]{\to}[\beta] \bullet\_\, [\alpha]{\to}\alpha$$

Then, for each function f of type t and $f_1, \ldots, f_n$ of type $t_1, \ldots, t_n$ such that all the applications "$f f_1$", …, "$f f_n$" are well-typed, we performed the local type inference on

$$t \bullet\_\, t_1 \wedge \ldots \wedge t_k \ \text{ for all } \ k \leq n$$

Remark that these applications are well typed since a function of type $t_1 \wedge \ldots \wedge t_n$ has also type t_i ($i = 1..n$), and any of these type is in the domain of t (since each individual application $t t_i$ is well-typed. Notice also that intersection of arrow types are never empty (all arrow types contain the type $\mathbb{1} \to \mathbb{0}$. Conversely, for all triple of functions f, f_1, f_2 such as "$f_1 f$" and "$f_2 f$" are well-typed, we also typed performed local type inference for $t_1 \vee t_2 \bullet\_\, t$.

Lastly, we added to our test suite some ill-typed applications (such as $(\alpha{\to}\beta){\to}[\alpha]{\to}[\beta] \bullet\_\,\texttt{Int}$) to ensure our implementation indeed detects these as invalid applications. Our test machine is an average laptop with 64bit Intel Core i3-2367M, 1.4Ghz, 4 cores and 8GB of RAM.

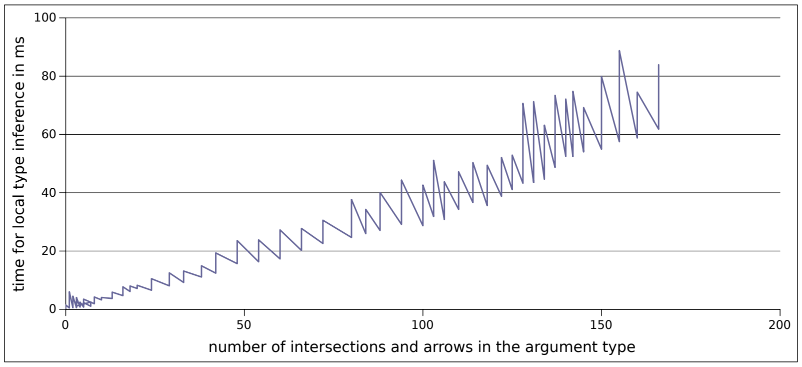
The results of our experiments are summarized in the following table:

# of tests	Total Time	Average Time	Median Time	Min. Time	Max. Time
1 859	27s	14ms	2.1ms	0.1ms	2.090s

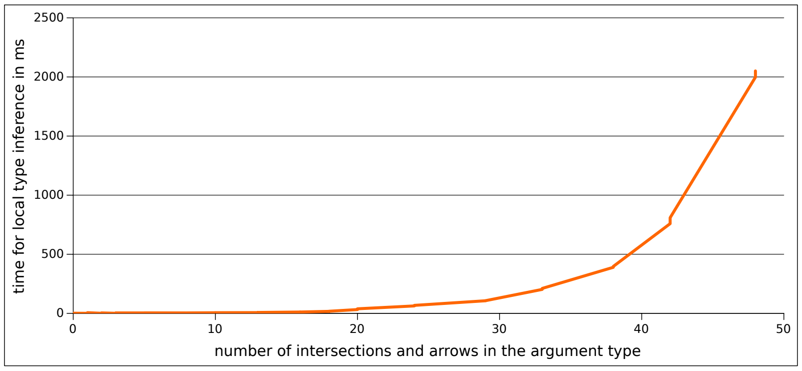
The worst time (2.09s) is the one for the local type inference of

```
  map  (length & hd & tl & nth & rev &
         append & rev_append & concat & flatten &
         iter & iteri & map & mapi & rev_map & fold_left)
```

As expected, the behavior of our algorithm is exponential (subtyping is already EXPTIME-complete, although our implementation performs very well even for large types). We illustrate the general behavior of our algorithm on two kinds of application. First, given a function of type $t \to s$ where t does not contain any arrows (but may contain products, sequences and so forth), local type inference scales linearly with the sum of the size of types $t \to s$ and u, when computing $(t \to s) \bullet\_\, u$.



However (and as expected) if we consider types $t \to s$ where t contains one, two, or more arrows, then local type inference becomes exponential with respect to the size of the argument u



While these tests already represent cases that are unlikely to happen in practice (the worst time case features more than 45 connectives/constructors, namely 15 intersections and 30 arrows), we conjecture that standard optimization techniques (hash-consing, memoization, laziness) will make our semi-naïve implementation even more tractable. During the experiment, memory usage was negligible (few megabytes).

Finally we also tested the type inference for applications of curried functions to several arguements. We added (by hand) to our test suite a set of functions that accept up to n arguments. More precisely for each arity n we added functions with the following types

$$\begin{array}{rcl}
\alpha_1 \to & \cdots & \to \alpha_n \to \texttt{Int} \\
\alpha_1 \to & \cdots & \to \alpha_n \to \alpha_1 \\
\alpha_1 \to & \cdots & \to \alpha_n \to (\alpha_1 \times \ldots \times \alpha_n) \\
(\beta_1 \to \alpha_1) \to & \cdots & \to (\beta_n \to \alpha_n) \to \texttt{Int} \\
(\beta_1 \to \alpha_1) \to & \cdots & \to (\beta_n \to \alpha_n) \to \alpha_1 \\
(\beta_1 \to \alpha_1) \to & \cdots & \to (\beta_n \to \alpha_n) \to (\alpha_1 \times \ldots \times \alpha_n)
\end{array}$$

each of these functions, if its arity is k, was then applied to k other randomly selected functions of this set. The test showed that our implementation can smoothly handle inference for the application of up to 20 arguments (for $n = 20$ the 120 tests take less than one second of cpu on a desktop workstation), then the exponential blowup becomes too important (in particular because of the memory footprint). The following table reports a sample of the cpu times for different n's

arity n	# of tests	Total Time for all the tests
10	60	0m0.033s
15	90	0m0.272s
20	120	0m0.768s
25	150	2m39.689s

Consider that in the standard library of OCaml export all functions have at most 5 arguments, and that there is margin for important improvement since we did not simplify the types of partial applications (whose intersection types are in general quite redundant).

Our implementation is already included in the development branch of the the ℂDuce distribution which can be retrieved at `https://www.cduce.org/redmine/projects/cduce`. It currently is in alpha-testing therefore we recommend the user to check the bug-tracker for open issues.

Also available is a prototype which implements the work described in both papers: type inference/reconstruction for implicitly-typed expressions, constraint solving with basic simplification algorithms, evaluation. The implementation is naive, not optimized, and implements very naive simplification heuristics, but it permits a smoother and friendlier evaluation and testing of our system since it is stable, includes an interactive toplevel and contains, a different test suite based on the examples used in both papers. It is available at `http://www.pps.univ-paris-diderot.fr/~gc/misc/polyduce.tar.gz`