GADTs for Invariants and Postconditions

BY ŁUKASZ STAFINIAK

Institute of Computer Science
University of Wrocław

Abstract

We implemented a system that infers invariants as types of recursive definitions, and postconditions as existential types. We present a Generalized Algebraic Data Types type system MMG(X) based on Francois Pottier and Vincent Simonet's HMG(X) but without type annotations. We extend it to a language with existential types represented as implicitly defined and used GADTs. We present the type inference problem as satisfaction of second order constraints over a multi-sorted domain. The InvarGenT system solves the constraints by iterated constraint abduction and disjunction elimination. It uses a Joint Constraint Abduction under Quantifier Prefix algorithm for free terms, linear equations and inequalities over rationals, and a "plug-in" algorithm for multisorted domains. Disjunction elimination in case of free terms computes anti-unification and in case of rationals computes extended convex hull.

Keywords: invariant inference, type inference, GADTs, constraint abduction

1 Introduction

Type systems are established natural deduction-style means to reason about programs. Dependent types can represent arbitrarily complex properties as they use the same language for both types and programs, the type of value returned by a function can itself be a function of the argument. Generalized Algebraic Data Types bring some of that expressivity to type systems that deal with data-types. Type systems with GADTs introduce the ability to reason about return type by case analysis of the input value, while keeping the benefits of a simple semantics of types, for example deciding equality can be very simple. Existential types are types that hide some information conveyed in a type, usually when that information cannot be reconstructed in the type system. A part of the type will often fail to be expressible in the simple language of types, it might even depend on input to the program. GADTs express existential types by using local type variables for the hidden parts of the type encapsulated in a GADT.

Our type system for GADTs differs from all others in that we do not require any type (or invariant) annotations on expressions, even on recursive functions. Our implementation: InvarGenT, see [19], differs from type systems in mainstream functional languages also in that we include linear equations and inequalities over rational numbers in the language of types, with the possibility to introduce more domains in the future.

1.1 Demonstration

The concrete syntax of InvarGenT is similar to that of OCaml. The sort of a type variable is identified by the first letter of the variable. `a`,`b`,`c`,`r`,`s`,`t`,`a1`,... are in the sort of terms, i.e. "types proper". `i`,`j`,`k`,`l`,`m`,`n`,`i1`,... are in the sort of linear arithmetics over rational numbers. Type constructors and value constructors have the same syntax: capitalized name followed by a tuple of arguments. They are introduced by `newtype` and `newcons` respectively. Values assumed into the environment are introduced by `external`.

`equal` is a function comparing values provided representation of their types:

```
newtype Ty : type   newtype Int   newtype List : type
newcons Zero : Int   newcons TInt : Ty Int
newcons Nil : ∀a. List a
newcons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
newcons TList : ∀a. Ty a ⟶ Ty (List a)
newtype Bool   newcons True : Bool   newcons False : Bool
external eq_int : Int → Int → Bool
external b_and : Bool → Bool → Bool
external b_not : Bool → Bool
external forall2 : ∀a, b. (a→b→Bool) → List a → List b → Bool

let rec equal = function
  | TInt, TInt -> fun x y -> eq_int x y
  | TPair (t1, t2), TPair (u1, u2) ->
    (fun (x1, x2) (y1, y2) ->
        b_and (equal (t1, u1) x1 y1)
              (equal (t2, u2) x2 y2))
  | TList t, TList u -> forall2 (equal (t, u))
  | _ -> fun _ _ -> False
```

InvarGenT returns an unexpected type: `equal`: `∀a,b.(Ty a, Ty b)→b→b→Bool`, one of four maximally general types of `equal`. This illustrates that unrestricted type systems with GADTs lack principal typing property.

InvarGenT commits to a type of a toplevel definition before proceeding to the next one, so sometimes we need to provide more information in the program. Besides type annotations, there are two means to enrich the generated constraints: `assert false` syntax for providing negative constraints, and `test` syntax for including constraints of use cases with constraint of a definition. To ensure only one maximally general type for `equal`, we use both. We add the lines:

```
  | TInt, TList l -> (function Nil -> assert false)
  | TList l, TInt -> (fun _ -> function Nil -> assert false)
test b_not (equal (TInt, TList TInt) Zero Nil)
```

Actually, InvarGenT returns the expected type `equal:∀a,b.(a,b)→a→b→Bool` when either the two `assert false` clauses or the `test` clause is added.

Now we demonstrate numerical invariants:

```
newtype Binary : num   newtype Carry : num
newcons Zero : Binary 0
newcons PZero : ∀n[0≤n]. Binary(n) ⟶ Binary(n+n)
newcons POne : ∀n[0≤n]. Binary(n) ⟶ Binary(n+n+1)
newcons CZero : Carry 0   newcons COne : Carry 1

let rec plus =
  function CZero ->
    (function Zero -> (fun b -> b)
      | PZero a1 as a ->
        (function Zero -> a
          | PZero b1 -> PZero (plus CZero a1 b1)
          | POne b1 -> POne (plus CZero a1 b1))
```
[...truncated...]

We get `plus:∀i,j,k.Carry i→Binary j→Binary k→Binary (i+j+k)`.

We can introduce existential types directly in type declarations. To have an existential type inferred, we have to use `efunction` or `ematch` expressions, which differ from `function` and `match` only in that the (return) type is an existential type. To use a value of an existential type, we have to bind it with a `let..in` expression. Otherwise, the existential type will not be unpacked. An existential type will be automatically unpacked before being "repackaged" as another existential type.

```
newtype Room   newtype Yard   newtype Village
newtype Castle : type   newtype Place : type
newcons Room : Room ⟶ Castle Room
newcons Yard : Yard ⟶ Castle Yard
newcons CastleRoom : Room ⟶ Place Room
newcons CastleYard : Yard ⟶ Place Yard
newcons Village : Village ⟶ Place Village
external wander : ∀a. Place a → ∃b. Place b

let rec find_castle = efunction
  | CastleRoom x -> Room x
  | CastleYard x -> Yard x
  | Village _ as x ->
    let y = wander x in
    find_castle y
```

We get `find_castle:∀a. Place a→ ∃b. Castle b`.

We end with a more practical existential type example:

```
newtype Bool   newcons True : Bool   newcons False : Bool
newtype List : type * num
newcons LNil : ∀a. List(a, 0)
newcons LCons : ∀n,a[0≤n]. a * List(a, n) ⟶ List(a, n+1)
```

```
let rec filter = fun f ->
  efunction LNil -> LNil
    | LCons (x, xs) ->
      ematch f x with
        | True ->
          let ys = filter f xs in
          LCons (x, ys)
        | False ->
          filter f xs
```

We get `filter:∀a,i.(a→Bool)→List (a, i)→ ∃j[j≤i].List (a, j)`.

Besides displaying types of toplevel definitions, InvarGenT also exports an OCaml source file with all the required GADT definitions and type annotations.

1.2 Contributions

We present the type inference problem for MMG(X), a Milner-Mycroft style variant of the HMG(X) type system without subtyping, as satisfaction of second order constraints over a multi-sorted domain. We provide a minimal extension of this type system that enables inference and easy use of existential types. Although introduction and elimination of existential types is not automated by the inference process, it is seamlessly integrated into expressions. Due to space constraints, the proofs are delegated to the appendix. We demonstrate several use cases using the *InvarGenT* system, see [19]. This concludes contributions of this publication. Below we list contributions brought by the *InvarGenT* system.

We revise our early work on abduction for multi-sorted domains from [18]. Our Joint Constraint Abduction under Quantifier Prefix algorithm builds on the fully maximal SCA answers algorithm from [8], but thanks to backtracking it can find answers to joint problems that are not fully maximal answers to each implication in the joint problem. Our JCA algorithm for linear arithmetics is novel.

We define the Constraint Disjunction Elimination problem. In case of free terms it is equivalent to anti-unification and in case of linear equations and inequalities it is equivalent to finding extended convex hull. As we do for abduction, we provide a combination-of-domains algorithm for disjunction elimination.

We design and implement an algorithm solving for predicate variables of the existential second order constraints generated for our type system. Details of all algorithms can be found in [19].

1.3 Related work

In the tradition of the Milner-Mycroft type system (see [3]), we modify the HMG(X) type system from [15] to MMG(X) by dropping the type specifications on recursive definitions from program terms. We also naturally restrict it by limiting the user-specified and inferred invariant constraints to use conjunction as the only logical connective. The traditional framework for loop invariant generation of [2] inspired the iterative aspect of our solver. While undecidability of type inference for polymorphic recursion suggests that an unbounded number of iterations might be needed, in practice abduction solves type inference for polymorphic recursion in one go. Still, with an arithmetic sort, we need 3 to 5 iterations. If a bound on the number of iterations could be derived, it would provide a proof of undecidability of constraint abduction.

Initially we were only aware of the work [4], which applies Dijkstra's weakest precondition calculus to refinement types. A work similar to ours could be done by application of the weakest precondition calculus to the Hoare logic of [12], with the conditions inserted by type inference.

The work in [17], although it is advertised as focused on dependent types, can be seen as extending [4] with reasoning by Boolean cases. Their programming language and type system is in several ways less expressive than the ML language with polymorphic recursion and the full GADTs type system: no inductive types (and therefore no pattern matching), refinement predicates over integers only instead of over arbitrary domains including types. Still, the inclusion of reasoning by cases and development of methods to actually find the refinement predicates, make [17] closer to our results.

Our algorithm eliminates implications in a way similar to [16], but using a slightly different definition of abduction. Use of abduction in [16] is related to the work in [7] and [8], where a more complete abduction algorithm is provided. Our algorithm is extensible to any constraint domain, by providing an abduction algorithm and a quantified conjunctive constraints solution algorithm. It necessarily includes the domain of equations over (free) algebraic terms.

There is a surge of recent work on type inference for GADTs, not contributing to our approach. Works such as [11] (older), [14], [6] and [5] modify the GADTs type system to make it more amenable to type inference (rejecting some reasonable programs as untypable), and develop less declarative inference algorithms. These works also do not allow other domains (than the free term algebra) to express invariants. [5] stands out from our point of view as it handles type inference for polymorphic recursion (by iteration).

Abduction algorithm for the term algebra is provided in [8], and for the linear arithmetic in [9], although further work driven by practical issues was needed.

In case of the free algebra of terms, constraint disjunction elimination reduces to anti-unification. Anti-unification was first introduced by Plotkin [10] and Reynolds [13]. [1] is a recent work on anti-unification, with an example application to invariant inference.

2 The Type System

We start by introducing notation. By the bar $\bar{e}$ we denote a sequence (or a set, depending on context) of elements e, by # we denote disjointness. With a free index i, $\bar{e_i}$ denotes $(e_1, ..., e_n)$ for some n associated with the index i; similarly, $\wedge_i \Phi_i$ denotes $\Phi_1 \wedge ... \wedge \Phi_n$. For convenience, we treat a conjunction of atoms $\wedge_i c_i$ as a set of atoms $\{c_1, ..., c_n\}$.

In some contexts, for a quantifier prefix $\mathcal{Q}$ we write $\mathcal{Q}$ to denote the set of variables quantified by $\mathcal{Q}$. Let FV be a generic function returning the free variables of any expression. For a quantifier prefix $\mathcal{Q}$ and variables x, y in $\mathcal{Q}$, by $x <_{\mathcal{Q}} y$ we denote that x is to the left of y in $\mathcal{Q}$ and they are separated by a quantifier alternation, by $x \leqslant_{\mathcal{Q}} y$ that it is not the case that $y <_{\mathcal{Q}} x$.

By $\Phi[\bar{\alpha} := \bar{t}\,]$, $\Phi[\overline{\alpha := t}\,]$, or $\Phi[\alpha_1 := t_1; ...; \alpha_n := t_n]$, we denote a substitution of terms $\bar{t}$ for corresponding variables $\bar{\alpha}$ in the formula Φ (where $\bar{\alpha}$ and $\bar{t}$ are finite sequences of the same length). By $\bar{s} \doteq \bar{t}$ we denote $\wedge_i s_i \doteq t_i$, where $\bar{s} = (s_1, ..., s_n)$ and $\bar{t} = (t_1, ..., t_n)$ for some n. When a substitution has a name, for example $S = [\bar{\alpha} := \bar{t}\,]$, we write substitution application as $S(\Phi) = \Phi[\bar{\alpha} := \bar{t}\,]$; we write $\dot{S} = \bar{\alpha} \doteq \bar{t}$; and we denote the substitution S corresponding to a formula $A = \dot{S} = \bar{\alpha} \doteq \bar{t}$ by $\tilde{A}$. We say that a substitution $[\bar{\alpha} := \bar{t}\,]$ agrees with a quantifier prefix $\mathcal{Q}$, when $\vDash \mathcal{Q}.\bar{\alpha} \doteq \bar{t}$ and in case of $\alpha_1 \doteq \alpha_2 \in \bar{\alpha} \doteq \bar{t}$ for variables α_1, α_2, we have $\alpha_2 \leqslant_{\mathcal{Q}} \alpha_1$.

2.1 The Language of Constraints

We are interested in a multisorted first-order language with equality $\mathcal{L}$, interpreted in a given model $\mathcal{M}$. The sort of terms or "types proper", denoted s_{ty}, plays a special role. In the current presentation, we will abstract from details of the language, posing the necessary properties as assumptions.

Consider a (first-order) language $\mathcal{L}$ with a model $\mathcal{M}$, the language of constraints for our type inference problem. Let ρ be an interpretation of types, that is an assignment of elements of $\mathcal{M}$ to variables in the corresponding sort, extended homomorphically to terms in the standard way. For $\Phi \in \mathcal{L}$, let $\mathcal{M}, \rho \vDash \Phi$ denote the interpretation of a formula Φ in the model $\mathcal{M}$ under the interpretation ρ, in the standard way, for example $\mathcal{M}, \rho \vDash \pi(t)$ if and only if $\pi(\rho(t))$ holds in $\mathcal{M}$, where predicate symbol π in $\mathcal{L}$ corresponds to predicate π in $\mathcal{M}$, etc.

Add to $\mathcal{L}$ a set of unary predicates $\chi(\cdot)$, which stand for invariants of recursive definitions in the constraints we will derive for type inference problems. Add a set of binary predicates $\chi_K(\cdot, \ \cdot)$, which will be put as constraints of data constructors K when we introduce inferred existential types. We call χ and χ_K *predicate variables*. Let $\mathrm{PV}^1(\cdot)$, resp. $\mathrm{PV}^2(\cdot)$ be the set of unary, resp. binary predicate variables in any expression, and $\mathrm{PV}(\Phi) = \mathrm{PV}^1(\Phi) \cup \mathrm{PV}^2(\Phi)$. We define *solved form formulas* to be existentially quantified conjunctions of atoms $\exists \bar{\alpha}.A$ without predicate variables.

For a formula Φ, let $\bar{\chi} = \mathrm{PV}^1(\Phi)$, resp. $\overline{\chi_K} = \mathrm{PV}^2(\Phi)$, and let $\overline{\chi(\tau_{\chi,k})}$, resp. $\overline{\chi_K(\tau_{K,k}, \tau'_{K,k})}$ be all occurrences of χ, resp. χ_K in Φ. We call an assignment $\mathcal{I} = \bar{\chi} := \overline{\exists \bar{\alpha}_\chi.F_\chi}\,; \overline{\chi_K} := \overline{\exists \bar{\alpha}_K.F_K}$ an *interpretation of predicate variables for* Φ when

1. $\overline{\exists \bar{\alpha}_i.F_i}\ \overline{\exists \bar{\alpha}_j.F_j}$ are solved form formulas,
2. $\delta \bar{\alpha}_\chi \# \mathrm{FV}(\wedge_k \tau_{\chi,k})$ and $\delta \delta' \bar{\alpha}_K \# \mathrm{FV}(\wedge_k \tau_{K,k} \wedge_k \tau'_{K,k})$,
3. for every variable $\beta \in \mathrm{FV}(F_\chi) \backslash \delta \bar{\alpha}_\chi$, there is a quantifier that binds β at every position of $\chi(\tau_{\chi,k})$ in Φ,

4. $\mathrm{FV}(F_K) \subseteq \delta\delta'\bar{\alpha}_K$.

Define a statement $\mathcal{M}, \mathcal{I}, \rho \vDash \Phi$ by: $\mathcal{I}$ is an interpretation of predicate variables for Φ, ρ is an interpretation of types, and $\mathcal{M}, \rho \vDash \mathcal{I}(\Phi)$. Define $\mathcal{M}, \mathcal{I} \vDash \Phi$ as: for all interpretations of types ρ, $\mathcal{M}, \mathcal{I}, \rho \vDash \Phi$. Define $\mathcal{M} \vDash \Phi$ as: for all interpretations of predicate variables $\mathcal{I}$ for Φ, $\mathcal{M}, \mathcal{I} \vDash \Phi$. Often we write $\mathcal{I} \vDash \Phi$, resp. $\vDash \Phi$, instead of $\mathcal{M}, \mathcal{I} \vDash \Phi$, resp. $\mathcal{M} \vDash \Phi$, since the model is fixed. We write $\mathcal{I}, C \vDash \Phi$, resp. $C \vDash \Phi$, for $\mathcal{I} \vDash C \Rightarrow \Phi$, resp. $\vDash C \Rightarrow \Phi$.

We say that a formula Φ is *satisfiable*, if and only if there exists an interpretation of predicate variables $\mathcal{I}$ for Φ, such that $\mathcal{I} \vDash \exists \mathrm{FV}(\Phi).\Phi$. As seen above, we extend the notion of substitution to handle predicate variable atoms, where the replacement of each occurrence of a variable depends on the argument of that variable. For interpretations of predicate variables $\mathcal{I}_1, \mathcal{I}_2$ with disjoint domains, we write their composition $\mathcal{I}_1\mathcal{I}_2(\cdot) = \mathcal{I}_1(\mathcal{I}_2(\cdot))$.

Above we in effect introduce a Henkin semantics for existential second order logic, tailored to our needs of invariant and postcondition inference.

2.2 The GADT Type System

By *types* τ we mean terms of sort s_{ty}. Define *type schemes* σ as $\forall\beta[D].\beta$, where D is either a solved form formula $\exists\bar{\alpha}.E$ or a predicate variable $\chi(\beta)$, and β is a variable of sort s_{ty}. A *simple environment* (or *monomorphic environment*) maps variables x to types τ. An *environment* (or *polymorphic environment*) maps variables x to type schemes σ. When a simple environment is appended to an environment, we identify τ and $\forall\beta[\beta \doteq \tau].\beta$ for $\beta \notin \mathrm{FV}(\tau)$. When operations pertaining to formulas are applied to a type scheme $\forall\beta[\exists\bar{\alpha}.E].\beta$ or $\forall\beta[\chi(\beta)].\beta$, they are performed on the formula $\exists\bar{\alpha}.E$ or $\chi(\beta)$. When operations pertaining to type schemes (types) are applied to (simple) environments Γ, they are performed on the image of Γ. Define *environment fragments* Δ to be triples $\exists\bar{\alpha}[D].\Gamma$ of variables $\bar{\alpha}$, atomic conjunctions D in $\mathcal{L}$ and simple environment Γ.

Unfortunately, our type inference algorithm does not handle disjunctive patterns. We therefore do not introduce them in our type system.

First, we present the type system in the standard, natural deduction style. The *type judgement* $C, \Gamma \vdash e{:}\,\tau$ or $C, \Gamma \vdash e{:}\,\sigma$ is composed of a formula C without predicate variables, an environment Γ, an expression e and a type τ or type scheme σ. Not mentioned explicitly is a set of data constructors Σ, which is fixed when typing an expression. If alternative sets of constructors are considered, we make them explicit by writing $C, \Gamma, \Sigma \vdash e{:}\,\tau$. The intended meaning of the type judgement $C, \Gamma, \Sigma \vdash e{:}\,\tau$ is: for every interpretation $\mathcal{I}$, ρ, if $\mathcal{I}, \rho \vDash C$, then the expression e has a ground type $\rho(\tau)$ in a ground environment $\rho(\mathcal{I}(\Gamma))$; and with constructors $\mathcal{I}(\Sigma)$ but this only becomes relevant starting from subsection 2.4. We define validity of type judgements in table 5, where D is a conjunction of atoms.

Note that the lack of the standard type schemes $\forall\bar{\alpha}[E].\tau$ is only for the simplicity of presentation, as they are equivalent to $\forall\beta[\exists\bar{\alpha}.E \wedge \beta \doteq \tau].\beta$.

A *data constructor* K for a *datatype* ε (recall that the sort s_{ty} holds two categories of elements: datatypes and function types) has definition

$K :: \forall\bar{\alpha}\bar{\beta}\,[D].\tau_1 \times ... \times \tau_n \rightarrow \varepsilon(\bar{\alpha})$ where $\mathrm{FV}(D, \tau_1, ..., \tau_n) \subseteq \bar{\alpha}\bar{\beta}$. D is a solved form formula $\exists\bar{\beta}\,'.A$.

– Patterns (syntax-directed)

`p-Empty`
$C \vdash 0{:}\, \tau \longrightarrow \exists\varnothing[\boldsymbol{F}]\{\}$

`p-Wild`
$C \vdash 1{:}\, \tau \longrightarrow \exists\varnothing[\boldsymbol{T}]\{\}$

`p-And`
$$\frac{\forall i \quad C, \Sigma \vdash p_i{:}\, \tau \longrightarrow \Delta_i}{C \vdash p_1 \wedge p_2{:}\, \tau \longrightarrow \Delta_1 \times \Delta_2}$$

`p-Var`
$C \vdash x{:}\, \tau \longrightarrow \exists\varnothing[\boldsymbol{T}]\{x \mapsto \tau\}$

`p-Cstr`
$$\frac{\forall i \quad C \wedge D \vdash p_i{:}\, \tau_i \longrightarrow \Delta_i \quad K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times ... \times \tau_n \rightarrow \varepsilon(\bar{\alpha}) \quad \bar{\beta}\#\mathrm{FV}(C)}{C \vdash K p_1 ... p_n{:}\, \varepsilon(\bar{\alpha}) \longrightarrow \exists\bar{\beta}[D](\Delta_1 \times ... \times \Delta_n)}$$

– Patterns (non-syntax-directed)

`p-EqIn`
$$\frac{C \vdash p{:}\, \tau' \longrightarrow \Delta \qquad C \vDash \tau \doteq \tau'}{C \vdash p{:}\, \tau \longrightarrow \Delta}$$

`p-SubOut`
$$\frac{C \vdash p{:}\, \tau \longrightarrow \Delta' \qquad C \vDash \Delta' \leqslant \Delta}{C \vdash p{:}\, \tau \longrightarrow \Delta}$$

`p-Hide`
$$\frac{C \vdash p{:}\, \tau \longrightarrow \Delta \qquad \bar{\alpha}\#\mathrm{FV}(\tau, \Delta)}{\exists\bar{\alpha}.C \vdash p{:}\, \tau \longrightarrow \Delta}$$

– Expressions (syntax-directed)

`Var`
$$\frac{\Gamma(x) = \forall\beta[\exists\bar{\alpha}.D].\beta \quad C \vDash D}{C, \Gamma \vdash x{:}\, \beta}$$

`Cstr`
$$\frac{\forall i C, \Gamma \vdash e_i{:}\, \tau_i \qquad C \vDash D \qquad K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 ... \tau_n \rightarrow \varepsilon(\bar{\alpha})}{C, \Gamma \vdash K e_1 ... e_n{:}\, \varepsilon(\bar{\alpha})}$$

`LetIn`
$$\frac{C, \Gamma \vdash \lambda(p.e_2)\, e_1{:}\, \tau}{C, \Gamma \vdash \mathbf{let}\; p = e_1\; \mathbf{in}\; e_2{:}\, \tau}$$

`App`
$$\frac{C, \Gamma \vdash e_1{:}\, \tau' \rightarrow \tau \qquad C, \Gamma \vdash e_2{:}\, \tau'}{C, \Gamma \vdash e_1\, e_2{:}\, \tau}$$

`LetRec`
$$\frac{C, \Gamma' \vdash e_1{:}\, \sigma \qquad C, \Gamma' \vdash e_2{:}\, \tau \qquad \sigma = \forall\beta[\exists\bar{\alpha}.D].\beta \quad \Gamma' = \Gamma\{x \mapsto \sigma\}}{C, \Gamma \vdash \mathbf{letrec}\; x = e_1\; \mathbf{in}\; e_2{:}\, \tau}$$

`Abs`
$$\frac{\forall i C, \Gamma \vdash c_i{:}\, \tau_1 \rightarrow \tau_2}{C, \Gamma \vdash \lambda(c_1 ... c_n){:}\, \tau_1 \rightarrow \tau_2}$$

– Expressions (non-syntax-directed)

`Gen`
$$\frac{C \wedge D, \Gamma \vdash e{:}\, \beta \qquad \beta\bar{\alpha}\#\mathrm{FV}(\Gamma, C)}{C \wedge \exists\beta\bar{\alpha}.D, \Gamma \vdash e{:}\, \forall\beta[\exists\bar{\alpha}.D].\beta}$$

`Inst`
$$\frac{C, \Gamma \vdash e{:}\, \forall\bar{\alpha}[D].\tau' \qquad C \vDash D[\bar{\alpha} := \bar{\tau}]}{C, \Gamma \vdash e{:}\, \tau'[\bar{\alpha} := \bar{\tau}]}$$

`DisjElim`
$$\frac{C, \Gamma \vdash e{:}\, \tau \qquad D, \Gamma \vdash e{:}\, \tau}{C \vee D, \Gamma \vdash e{:}\, \tau}$$

`Hide`
$$\frac{C, \Gamma \vdash e{:}\, \tau \qquad \bar{\alpha}\#\mathrm{FV}(\Gamma, \tau)}{\exists\bar{\alpha}.C, \Gamma \vdash e{:}\, \tau}$$

`Equ`
$$\frac{C, \Gamma \vdash e{:}\, \tau \qquad C \vDash \tau \doteq \tau'}{C, \Gamma \vdash e{:}\, \tau'}$$

`FElim`
$$\frac{}{\boldsymbol{F}, \Gamma \vdash e{:}\, \tau}$$

– Clauses

`Clause`
$$\frac{C \vdash p{:}\, \tau_1 \longrightarrow \exists\bar{\beta}[D]\Gamma' \quad C \wedge D, \Gamma\Gamma' \vdash e{:}\, \tau_2 \quad \bar{\beta}\#\mathrm{FV}(C, \Gamma, \tau_2)}{C, \Gamma \vdash p.e{:}\, \tau_1 \rightarrow \tau_2}$$

Table 1. Typing rules

At this point the construction `LetIn` is a syntactic sugar for single branch patterns – if polymorphic `let` is needed, use `LetRec`. Note that `DisjElim` is unrelated to Constraint Disjunction Elimination we introduce in a later section.

An expression e is *well typed* given Γ,Σ when $\mathrm{PV}(\Gamma,\Sigma)=\varnothing$ and $C,\Gamma,\Sigma\vdash e:\sigma$ holds for some satisfiable constraint C. For simplicity, InvarGenT only admits type and invariant annotations from the user on toplevel definitions. Toplevel definitions in InvarGenT can be seen as a nesting of subsequent `LetRec` and `LetIn` constructions in the scope of previous definitions, with the restriction that the body of each definition is a well typed expression given Γ,Σ with $\mathrm{FV}(\Gamma)=\varnothing$.

Now, we present type judgements declaratively by reducing them to constraints. For $\bar{c}=\overline{p_i.e_i}$, $[\![\Gamma\vdash\bar{c}:\tau_1\rightarrow\tau_2]\!]:=\wedge_i[\![\Gamma\vdash p_i.e_i:\tau_1\rightarrow\tau_2]\!]$. (The presentation is a little bit heavy due to explicit capture-avoidance conditions.)

– Patterns (constraint generation)

$$
\begin{aligned}
[\![\vdash 0\downarrow\tau]\!] &= \boldsymbol{T}\\
[\![\vdash 1\downarrow\tau]\!] &= \boldsymbol{T}\\
[\![\vdash x\downarrow\tau]\!] &= \boldsymbol{T}\\
[\![\vdash p_1\wedge p_2\downarrow\tau]\!] &= [\![\vdash p_1\downarrow\tau]\!]\wedge[\![\vdash p_2\downarrow\tau]\!]\\
[\![\vdash K p_1 \ldots p_n\downarrow\tau]\!] &= \exists\bar{\alpha}'.(\varepsilon(\bar{\alpha}')\doteq\tau\wedge\\
&\qquad \forall\bar{\beta}'.D[\bar{\alpha}\bar{\beta}:=\bar{\alpha}'\bar{\beta}']\Rightarrow\wedge_i[\![p_i\downarrow\tau_i[\bar{\alpha}\bar{\beta}:=\bar{\alpha}'\bar{\beta}']]\!])\\
&\quad \text{where } K::\forall\bar{\alpha}\bar{\beta}[D].\tau_1\times\ldots\times\tau_n\rightarrow\varepsilon(\bar{\alpha}),\\
&\qquad\quad \bar{\alpha}'\bar{\beta}'\#\mathrm{FV}(\Sigma,\tau)
\end{aligned}
$$

– Patterns (environment fragment generation)

$$
\begin{aligned}
[\![\vdash 0\uparrow\tau]\!] &= \exists\varnothing[\boldsymbol{F}]\{\}\\
[\![\vdash 1\uparrow\tau]\!] &= \exists\varnothing[\boldsymbol{T}]\{\}\\
[\![\vdash x\uparrow\tau]\!] &= \exists\varnothing[\boldsymbol{T}]\{x\mapsto\tau\}\\
[\![\vdash p_1\wedge p_2\uparrow\tau]\!] &= [\![\vdash p_1\uparrow\tau]\!]\times[\![\vdash p_2\uparrow\tau]\!]\\
[\![\vdash K p_1 \ldots p_n\uparrow\tau]\!] &= \exists\bar{\alpha}'\bar{\beta}'[\varepsilon(\bar{\alpha}')\doteq\tau\wedge D[\bar{\alpha}\bar{\beta}:=\bar{\alpha}'\bar{\beta}']]\\
&\qquad (\times_i[\![p_i\uparrow\tau_i[\bar{\alpha}\bar{\beta}:=\bar{\alpha}'\bar{\beta}']]\!])\\
&\quad \text{where } K::\forall\bar{\alpha}\bar{\beta}[D].\tau_1\times\ldots\times\tau_n\rightarrow\varepsilon(\bar{\alpha}),\\
&\qquad\quad \bar{\alpha}'\bar{\beta}'\#\mathrm{FV}(\Sigma,\tau)
\end{aligned}
$$

Table 2. Type inference for patterns

$$
\begin{array}{rcl}
[\![\Gamma \vdash x\colon \tau]\!] & = & \boldsymbol{F} \quad \text{when} \quad x \notin \mathrm{Dom}(\Gamma) \\
[\![\Gamma \vdash x\colon \tau]\!] & = & \exists \beta' \bar{\alpha}'.D[\beta\bar{\alpha} := \beta'\bar{\alpha}'] \wedge \beta' \doteq \tau \\
 & & \text{where } \Gamma(x) = \forall\beta[\exists\bar{\alpha}.D].\beta,\ \beta'\bar{\alpha}'\#\mathrm{FV}(\Gamma,\tau) \\
[\![\Gamma \vdash \lambda\bar{c} : \tau]\!] & = & \exists\alpha_1\alpha_2.[\![\Gamma \vdash \bar{c} : \alpha_1 \rightarrow \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 \doteq \tau, \\
 & & \alpha_1\alpha_2\#\mathrm{FV}(\Gamma,\tau) \\
[\![\Gamma \vdash e_1\, e_2\colon \tau]\!] & = & \exists\alpha.[\![\Gamma \vdash e_1\colon \alpha \rightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha]\!], \alpha\#\mathrm{FV}(\Gamma,\tau) \\
[\![\Gamma \vdash K e_1 \ldots e_n\colon \tau]\!] & = & \exists\bar{\alpha}'\bar{\beta}'.(\wedge_i [\![\Gamma \vdash e_i\colon \tau_i[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}']]\!] \wedge \\
 & & \quad D[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}'] \wedge \varepsilon(\bar{\alpha}') \doteq \tau) \\
 & & \text{where } \Sigma \ni K :: \forall\bar{\alpha}\bar{\beta}\,[D].\tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon(\bar{\alpha}), \\
 & & \qquad \bar{\alpha}'\bar{\beta}'\#\mathrm{FV}(\Gamma,\tau) \\
[\![\Gamma \vdash \mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2\colon \tau]\!] & = & (\forall\beta(\chi(\beta) \Rightarrow [\![\Gamma\{x \mapsto \forall\beta[\chi(\beta)].\beta\} \vdash e_1\colon \beta]\!])) \wedge \\
 & & (\exists\alpha.\chi(\alpha)) \wedge [\![\Gamma\{x \mapsto \forall\beta[\chi(\beta)].\beta\} \vdash e_2\colon \tau]\!] \\
 & & \text{where } \beta\#\mathrm{FV}(\Gamma,\tau), \chi\#\mathrm{PV}(\Gamma) \\
[\![\Gamma \vdash p.e\colon \tau_1 \rightarrow \tau_2]\!] & = & [\![\vdash p\downarrow\tau_1]\!] \wedge \forall\bar{\beta}.D \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!] \\
 & & \text{where } \exists\bar{\beta}\,[D]\Gamma' \text{ is } [\![\vdash p\uparrow\tau_1]\!], \bar{\beta}\#\mathrm{FV}(\Gamma,\tau_2) \\
[\![\Gamma \vdash \mathrm{ce}\colon \forall\bar{\alpha}\,[D].\tau]\!] & = & \forall\bar{\alpha}'.D[\bar{\alpha} := \bar{\alpha}'] \Rightarrow [\![\Gamma \vdash \mathrm{ce}\colon \tau[\bar{\alpha} := \bar{\alpha}']]\!], \\
 & & \bar{\alpha}'\#\mathrm{FV}(\Gamma)
\end{array}
$$

Table 3. Type inference for expressions and clauses

The two presentations are equivalent, in the sense of theorems *correctness* and *completeness* below.

Theorem 1. Correctness *(expressions).* $[\![\Gamma \vdash \mathrm{ce} : \tau]\!], \Gamma \vdash \mathrm{ce} : \tau$.

Theorem 2. Completeness *(expressions). If* $\mathrm{PV}(C, \Gamma) = \varnothing$ *and* $C, \Gamma \vdash \mathrm{ce} : \tau$*, then there exists an interpretation of predicate variables* $\mathcal{I}$ *such that* $\mathcal{I}, C \vDash [\![\Gamma \vdash \mathrm{ce} : \tau]\!]$.

Corollary 3. *If* $C, \Gamma \vdash \mathrm{ce} : \forall \bar{\alpha}[D].\tau$ *and* $\bar{\alpha} \# \mathrm{FV}(\Gamma)$*, then there is an interpretation* $\mathcal{I}$ *such that* $\mathcal{I}, C \vDash \forall \bar{\alpha}.D \Rightarrow [\![\Gamma \vdash \mathrm{ce} : \tau]\!]$.

2.3 Example: `eval`

Consider a short example function `eval`:

```
newtype Term : type   newtype Int   newtype Bool
external plus : Int → Int → Int
external is_zero : Int → Bool
external if : ∀a. Bool → a → a → a
newcons Lit : Int ⟶ Term Int
newcons Plus : Term Int * Term Int ⟶ Term Int
newcons IsZero : Term Int ⟶ Term Bool
newcons If : ∀a. Term Bool * Term a * Term a ⟶ Term a

let rec eval = function
  | Lit i -> i
  | IsZero x -> is_zero (eval x)
  | Plus (x, y) -> plus (eval x) (eval y)
  | If (b, t, e) -> if (eval b) (eval t) (eval e)
```

Constraint, with indentation showing scope of implication conclusions:

```
∀t1.χ1(t1) ⟹
  ∃t3, t4. t3 → t4 = t1 ∧ ∃t5. Term t5 = t3 ∧
  ∀t6. Term t6 = t3 ∧ Int = t6 ⟹ ∃. Int = t4 ∧
  ∃t7. Term t7 = t3 ∧
  ∀t8. Term t8 = t3 ∧ Bool = t8 ⟹
    ∃t9. Int → Bool = t9 → t4 ∧
    ∃t10. ∃t11. t11 = t10 → t9 ∧ χ1(t11) ∧ Term Int = t10 ∧
  ∃t12. (Term t12) = t3 ∧
  ∀t13. Term t13 = t3 ∧ Int = t13 ⟹
```

```
    ∃t14. ∃t17. Int → Int → Int = t17 → t14 → t4 ∧
    ∃t18. ∃t19. t19 = t18 → t17 ∧ χ1(t19) ∧ Term Int = t18 ∧
    ∃t15. ∃t16. t16 = t15 → t14 ∧ χ1(t16) ∧ Term Int = t15 ∧
  ∃t20. Term t20 = t3 ∧
  ∀t21. Term t21 = t3 ⟹
    ∃t22. ∃t25. ∃t28.
    ∃t31. Bool → t31 → t31 → t31 = t28 → t25 → t22 → t4 ∧
    ∃t29. ∃t30. t30 = t29 → t28 ∧ χ1(t30) ∧ Term Bool = t29 ∧
    ∃t26. ∃t27. t27 = t26 → t25 ∧ χ1(t27) ∧ Term t21 = t26 ∧
    ∃t23. ∃t24. t24 = t23 → t22 ∧ χ1(t24) ∧ Term t21 = t23 ∧
∃t2. χ1(t2)
```

Normalized and simplified constraint, schematically $\mathcal{Q}.\wedge_i (D_i \Longrightarrow C_i)$:

```
1| χ1(t2)
2| χ1(t1) ⟹ t3 = Term t5 ∧ t1 = Term t5 → t4
3| (Term t21) = t3 ∧ χ1(t1) ⟹ t24 = Term t21 → t4 ∧
    t27 = (Term t21 → t4) ∧ t30 = Term Bool → Bool ∧ χ1(t30) ∧
    χ1(t27) ∧ χ1(t24)
4| Term t6 = t3 ∧ Int = t6 ∧ χ1(t1) ⟹ t4 = Int
5| Term t8 = t3 ∧ Bool = t8 ∧ χ1(t1) ⟹ t11 = Term Int → Int ∧
    t4 = Bool ∧ χ1(t11)
6| Term t13 = t3 ∧ Int = t13 ∧ χ1(t1) ⟹ t16 = Term Int → Int
    ∧ t19 = Term Int → Int ∧ t4 = Int ∧ χ1(t19) ∧ χ1(t16)
```

Quantifier structure is preserved separately. Implication branch 1 (with empty premise) makes sure that the invariant for `eval` is satisfiable. Branch 2 records that the argument of `eval` is a `Term`. Branch 3 covers the recursive calls in `if`, ensuring that `Term Bool → Bool` satisfies the invariant. Branch 4 says that the result for input `Lit i` is of type `Int`. Branch 5 is derived for the case computing `is_zero (eval x)` given input `IsZero x`, and branch 6 for computing `plus`.

2.4 Existential Types

In context of GADTs, existential types play a prominent role, beyond the traditional role of abstraction in software engineering. Without existential types, computations would need to express parameters of the output datatype invariant as a function of parameters of the input datatype invariant. Since GADTs are introduced to curtail the expressivity of types compared to full dependent type systems, opportunities for such functional dependency are rare by design. We need the capacity in the type system to express whatever relations it can of the resulting datatype parameters to the input datatype parameters. Traditionally in GADTs we package the result into a custom datatype. This is tedious and contrary to the benefits of type inference. We automate this process, in effect introducing inferred existential types to our type system. Since the modification of the type system is minimal, formal guarantees carry over to it and it will be familiar to users of GADTs.

Existential quantifiers in argument positions of function types are redundant: they can be lifted to be traditional, polymorphic variables constrained by the invariant of the function. We prohibit the use of inferred existential types in argument positions: it could only result from a mistake.

We introduce a new expression construct $\lambda[K]\bar{c}$, where K is a value constructor, but is not available in concrete syntax, and $\bar{c}$ are pattern matching clauses. In the implementation, the parser introduces a fresh K and forms $\lambda[K]\bar{c}$ for `efunction` $\bar{c}$. $\lambda[K]\bar{c}$ is eliminated by a normalization step. We also introduce a rule `ExLetIn` to the type system, responsible for elimination of existential types. When $K :: \forall\bar{\alpha}\bar{\beta}\gamma[E].\gamma \rightarrow \varepsilon_K(\bar{\alpha}) \in \Sigma$ is such a data constructor absent from concrete syntax, the pretty-printer for types prints $\varepsilon_K(\bar{\tau})$ as $(\exists\bar{\beta}\gamma[E[\bar{\alpha} := \bar{\tau}]].\gamma)$, or $(\exists\bar{\beta}\,[E[\bar{\alpha} := \bar{\tau}]].\tau_e)$ when $\gamma \doteq \tau_e \in E$.

Let $l(e)$ defined in table 4 determine whether an expression introduces or eliminates an existential type.

$$
\begin{array}{rcl}
l(x) & = & \boldsymbol{F} \\
l(\lambda\bar{c}) & = & \boldsymbol{F} \\
l(e_1\, e_2) & = & l(e_1) \\
l(K e_1 ... e_n) & = & \boldsymbol{F} \\
l(\mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2) & = & l(e_2) \\
l(\lambda[K]\overline{p_i.e_i}) & = & \boldsymbol{T} \\
l(\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2) & = & \boldsymbol{T}
\end{array}
$$

Table 4. Does the expression introduce or eliminate an existential type?

Let all occurrences of $\lambda[K]$ in e use distinct K. Let $n(e) := n(e, \bot)$, defined in table 5, flatten nested introductions of existential types. Let $\mathcal{E}(e) := \mathcal{E}(e, \boldsymbol{F})$, defined in table 6, collect value constructors introduced for existential types.

$$
\begin{array}{rcl}
n(e, K') & = & \mathbf{let}\, x = n(e, \bot)\, \mathbf{in}\, K'\, x \\
\text{when } K' \neq \bot \wedge l(e) = \boldsymbol{F} & & \\
n(x, \bot) & = & x \\
n(\lambda\bar{c}, \bot) & = & \lambda(\overline{n(c, \bot)}) \\
n(e_1\, e_2, K') & = & n(e_1, K')\, n(e_2, \bot) \\
n(K e_1 ... e_n, \bot) & = & K n(e_1, \bot) ... n(e_n, \bot) \\
n(\mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2, K') & = & \mathbf{letrec}\, x = n(e_1, \bot)\, \mathbf{in}\, n(e_2, K') \\
n(p.e, K') & = & p.\underline{n(e, K')} \\
n(\lambda[K]\bar{c}, \bot) & = & \lambda(\overline{n(c, K)}) \\
n(\lambda[K]\bar{c}, K') & = & \lambda(\overline{n(c, K')}) \\
\text{when } K' \neq \bot & & \\
n(\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2, K') & = & \mathbf{let}\, p = n(e_1, \bot)\, \mathbf{in}\, n(e_2, K')
\end{array}
$$

Table 5. Flatten nested introductions of existential types

$$\begin{array}{rcl}\mathcal{E}(x,v) & = & \varnothing\\ \mathcal{E}(\lambda\bar{c}\,,v) & = & \cup\overline{\mathcal{E}(c,\boldsymbol{F})}\\ \mathcal{E}(e_1\,e_2,v) & = & \mathcal{E}(e_1,v)\cup\mathcal{E}(e_2,\boldsymbol{F})\\ \mathcal{E}(Ke_1...e_n,v) & = & \cup_i\mathcal{E}(e_i,\boldsymbol{F})\\ \mathcal{E}(\mathbf{letrec}\,x=e_1\,\mathbf{in}\,e_2,v) & = & \mathcal{E}(e_1,\boldsymbol{F})\cup\mathcal{E}(e_2,v)\\ \mathcal{E}(p.e,v) & = & \mathcal{E}(e,v)\\ \mathcal{E}(\lambda[K]\bar{c}\,,\boldsymbol{F}) & = & \{K\}\cup\overline{\mathcal{E}(c,\boldsymbol{T})}\\ \mathcal{E}(\lambda[K]\bar{c}\,,\boldsymbol{T}) & = & \cup\overline{\mathcal{E}(c,\boldsymbol{T})}\\ \mathcal{E}(\mathbf{let}\,p=e_1\,\mathbf{in}\,e_2,v) & = & \mathcal{E}(e_1,\boldsymbol{F})\cup\mathcal{E}(e_2,v)\end{array}$$

Table 6. Collect introduced value constructors

We put the normalization step into the type system as rule `ExIntro`. W.l.o.g. `ExIntro` can be used once at the beginning of derivation. We add rule `ExLetIn`. Although `LetIn` and `ExLetIn` resemble "syntactic sugar", their application is non-deterministic. We include value constructor environment in judgements to faciliate the completeness proof. Me modify the rule `App` to exclude existential types from function positions. We achieve that by introducing a new atomic predicate $\not{E}$ to the sort of terms, i.e. $\not{E}(\tau)\equiv\wedge_K\neg\exists\bar{\alpha}\,.\tau\dot{=}\varepsilon_K(\bar{\alpha}\,)$.

`App`
$$\frac{C,\Gamma,\Sigma\vdash e_1\colon\tau'\rightarrow\tau\quad C,\Gamma,\Sigma\vdash e_2\colon\tau'\quad C\vDash\not{E}(\tau')}{C,\Gamma,\Sigma\vdash e_1\,e_2\colon\tau}$$

`ExLetIn`
$$\frac{\varepsilon_K(\bar{\alpha})\text{ in }\Sigma\quad C,\Gamma,\Sigma\vdash e_1\colon\tau'\quad C,\Gamma,\Sigma\vdash Kp.e_2\colon\tau'\rightarrow\tau}{C,\Gamma,\Sigma\vdash\mathbf{let}\,p=e_1\,\mathbf{in}\,e_2\colon\tau}$$

`ExIntro`
$$\frac{\mathrm{Dom}(\Sigma')\backslash\mathrm{Dom}(\Sigma)=\mathcal{E}(e)\quad C,\Gamma,\Sigma'\vdash n(e)\colon\tau}{C,\Gamma,\Sigma\vdash e\colon\tau}$$

Table 7. Added typing rules

Definition 4. *Let* $\Sigma=\Sigma_0\cup\Sigma_e$ *and* $\Sigma'=\Sigma_0\cup\Sigma'_e$ *be sets of value constructors related to each other as follows:*

- $\mathrm{PV}^2(\Sigma_0)=\varnothing$,
- $\Sigma_e=\overline{K::\forall\alpha_K\gamma_K[\chi_K(\gamma_K,\alpha_K)].\gamma_K\rightarrow\varepsilon_K(\alpha_K)}$,
- *and* $\Sigma'_e=\overline{K::\forall\bar{\alpha}'_K\bar{\beta}'_K\gamma_K[E_K].\gamma_K\rightarrow\varepsilon_K(\bar{\alpha}'_K)}$

where $\exists\bar{\alpha}'_K\bar{\beta}'_K\gamma_K.E_K$ *are solved form formulas. Define* $\Sigma'/\Sigma=\mathcal{I}_e=[\overline{\chi_K}:=\overline{\exists\bar{\alpha}_K.F_K}\,]$ *be* $F_K=E_K\wedge\alpha_K\dot{=}\overrightarrow{\alpha_K}'$ *and* $\bar{\alpha}_K=\bar{\alpha}'_K\bar{\beta}'_K$.

Note that by proposition 8, we do not lose generality by using single-argument datatypes $\varepsilon_K(\alpha)$ rather than the general form $\varepsilon_K(\bar{\alpha}\,)$.

Normalization defined in table 5 is responsible for introduction of existential types, but it also ensures that inferred existential types never directly contain other existential types. This flattening of existential types has no downsides, and enables the use of all information available to derive the postcondition, i.e. the existential type. To flatten nested existential types, we rename constructors K to K' in $n(\lambda[K]\bar{c}\,,K')$, and eliminate potential existential type before introducing one in $n(e,K')$ when $K'\neq\bot\wedge l(e)=\boldsymbol{F}$.

2.5 Type Inference Constraints for Existential Types

The type inference uses predicate variables to determine the existential condition. For the non-recursive call to $[\![\cdot]\!]$, we normalize the expression. We shorten $[\![\Gamma, \Sigma \vdash \cdot : \tau]\!]$ to $[\![\Gamma \vdash \cdot : \tau]\!]$.

$$
\begin{array}{rcl}
[\![\Gamma \vdash e_1\, e_2 : \tau]\!] & = & \exists \alpha . [\![\Gamma \vdash e_1 : \alpha \rightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!] \wedge \not{E}(\alpha), \alpha \# \mathrm{FV}(\Gamma, \tau) \\
[\![\Gamma, \Sigma_0 \vdash e : \tau]\!] & = & [\![\Gamma, \Sigma \vdash n(e) : \tau]\!] \\
\text{when } \mathcal{E}(e) \neq \varnothing & & \text{where } \Sigma = \\
 & & \Sigma_0 \overline{K :: \forall \alpha_K \gamma_K [\chi_K(\gamma_K, \alpha_K)] . \gamma_K \rightarrow \varepsilon_K(\alpha_K)}_{K \in \mathcal{E}(e)} \\
[\![\Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 : \tau]\!] & = & \exists \alpha_0 . [\![\Gamma \vdash e_1 : \alpha_0]\!] \wedge \\
 & & ([\![\Gamma \vdash p.e_2 : \alpha_0 \rightarrow \tau]\!] \wedge \not{E}(\alpha_0) \vee_{\mathcal{E}} [\![\Gamma \vdash K p.e_2 : \alpha_0 \rightarrow \tau]\!]) \\
 & & \text{where } \mathcal{E} = \{K \mid K :: \forall \bar{\alpha} \bar{\beta}\, [E] . \tau \rightarrow \varepsilon_K(\bar{\alpha}) \in \Sigma\}
\end{array}
$$

Table 8. Type inference for the added expressions

Our tools for solving second order constraints only handle conjunctions of implications. We solve disjunctions early, which is problematic as selecting a disjunct may require information hidden in other disjunctions or in predicate variables. For example, in the normalization of constraints we need to associate each unary predicate variable with at most one inferred existential type that can occur as return type in its solution. The pragmatics we adopt in InvarGenT is that whenever the $[\![\Gamma \vdash p.e_2 : \alpha_0 \rightarrow \tau]\!]$ disjunct coming from the `LetIn` rule is satisfiable with the rest of the constraint, we select it for the solution. One can turn the pragmatics into semantics by adding premise $C, \Gamma, \Sigma \nvdash \lambda(p.e_2)\, e_1 : \tau$ to the `ExLetIn` rule, but it makes the formalism a bit more complex.

Theorem 5. *Theorems 1 (Correctness) and 2 (Completeness) hold for the type system extended with* `ExIntro` *and* `ExLetIn` *in the following sense.*

Correctness: $[\![\Gamma, \Sigma \vdash \mathrm{ce} : \tau]\!], \Gamma, \Sigma \vdash \mathrm{ce} : \tau$.

Completeness: If $\mathrm{PV}(C, \Gamma, \Sigma) = \varnothing$ *and* $C, \Gamma, \Sigma \vdash \mathrm{ce} : \tau$*, then there exist interpretations of predicate variables* $\mathcal{I}_u, \mathcal{I}_e$ *such that* $\mathrm{Dom}(\mathcal{I}_u)$ *are unary,* $\mathrm{Dom}(\mathcal{I}_e) = \{\chi_K \mid K \in \mathcal{E}(\mathrm{ce})\}$*, and* $\mathcal{I}_u, C \vDash \mathcal{I}_e([\![\Gamma, \Sigma \vdash \mathrm{ce} : \tau]\!])[\overline{\varepsilon_K(\vec{\tau})} := \overline{\varepsilon_K(\bar{\tau})}]$.

The set of value constructors is updated in InvarGenT after a toplevel definition with a well typed body: from Σ_0 to Σ', using the notation from definition 4.

2.6 Example: `filter`

Consider the function `filter` from the end of demonstration subsection. Constraint with disjunctions already pruned, for conciseness:

```
∀t1.χ2(t1) ⟹
  ∃t3, t4. t3 → t4 = t1 ∧[...truncated...]
  ∀n25, n26, t27.
  List (t27, n25) = t5 ∧ (n26 + 1) = n25 ∧ 0 ≤ n26 ⟹
    ∃t28.∃t30, t31. t30 → t31 = t28 → t6 ∧ ∃. Bool = t30 ∧
    Bool = t30 ⟹
      ∃t32.∃t33.∃t34.
      ∃t35. t35 = t34 → t33 → t32 ∧ χ2(t35) ∧ t3 = t34 ∧
      Ɇ(t34) ∧ List (t27, n26) = t33 ∧ Ɇ(t33) ∧
      ∃t37. (∃2:δ[χ1(δ, t37)].δ) = t32 ∧ ∀t36.
      ∀t38, t39. (∃2:δ[χ1(δ, t39)].δ) = t32 ∧ χ1(t38, t39) ⟹
        ∃t40.∃n41, n42, t43.
        List (t43, n41) = t40 ∧ n42 + 1 = n41 ∧ 0 ≤ n42 ∧
        t27 = t43 ∧ t38 = List (t43, n42) ∧
        ∃t50, t51. (∃2:δ[χ1(δ, t51)].δ) = t31 ∧
        χ1(t50, t51) ∧ t40 = t50 ∧ Ɇ(t40) ∧[...truncated...]
```

Notation such as (∃2:δ[χ1(δ, t51)].δ) identifies an occurrence of existential type, here $\varepsilon_{K_2}(t_{51})$ such that $K_2 :: \forall\delta\alpha[\chi_1(\delta,\alpha)].\delta \rightarrow \varepsilon_{K_2}(\alpha)$. Normalized and simplified constraint:

```
1| χ2(t2)
2| χ2(t1) ⟹ t5 = List (t8, n7) ∧ t1 = t3 → List (t8, n7) → t6
3| (∃2:δ[χ1(δ, t39)].δ) = t32 ∧ χ1(t38, t39) ∧
    List (t27, n25) = t5 ∧ n26 + 1 = n25 ∧ 0 ≤ n26 ∧ χ2(t1) ⟹
    t40 = t50 ∧ t31 = (∃2:δ[χ1(δ, t51)].δ) ∧ χ1(t50, t51) ∧
    t38 = List (t27, n42) ∧ t40 = List (t27, n41) ∧
    n42 + 1 = n41 ∧ 0 ≤ n42
4| (∃2:δ[χ1(δ, t71)].δ) = t64 ∧ χ1(t70, t71) ∧
    List (t27, n25) = t5 ∧ n26 + 1 = n25 ∧ 0 ≤ n26 ∧ χ2(t1) ⟹
    t72 = t70 ∧ t31 = (∃2:δ[χ1(δ, t73)].δ) ∧ χ1(t72, t73)
5| List (t10, n9) = t5 ∧ 0 = n9 ∧ χ2(t1) ⟹ t11 = t20 ∧
    t6 = (∃2:δ[χ1(δ, t21)].δ) ∧ χ1(t20, t21) ∧
    t11 = List (t13, n12) ∧ 0 = n12
6| List (t27, n25) = t5 ∧ n26 + 1 = n25 ∧ 0 ≤ n26 ∧ χ2(t1) ⟹
    t32 = (∃2:δ[χ1(δ, t37)].δ) ∧ t64 = (∃2:δ[χ1(δ, t69)].δ) ∧
    t3 = t27 → Bool ∧ t31 = t6 ∧
    t35 = t3 → List (t27, n26) → t32 ∧ χ2(t35) ∧
    t67 = t3 → List (t27, n26) → t64 ∧ χ2(t67)
```

Branch 1 ensures that the invariant is satisfiable. Branch 2 decomposes the type of the recursive definition and ensures that the second argument is a list. Branch 3 is the case of passed element: `t38` is the type of the recursive call, and the length of resulting list `n41` is increased. Branch 4 is the case of dropped element: the result `t72` and the recursive call result `t70` coincide. Branch 5 is the case of empty list. Branch 6 provides invariant information for branches 3 and 4: `t35` is the type of recursive call with result `t38` thanks to χ`1(t38, t39)`, and `t67` of call with result `t70` thanks to χ`1(t70, t71)`.

3 Solving Second Order Constraints

Least Upper Bounds and Greatest Lower Bounds computations are the standard tools for finding unknowns involved in an order structure. In case of implicational constraints, constraint abduction and constraint "disjunction elimination" belong to this toolset. *Simple Constraint Abduction* under Quantifier Prefix is the task of finding for an implication $\mathcal{Q}.D \Rightarrow C$, where $\mathcal{Q}$ is a quantifier prefix and D, C are conjunctions of atoms, a weakest solved form formula $\exists\bar{\alpha}.A$ such that $\vDash(\exists\bar{\alpha}.A) \Rightarrow (D \Rightarrow C)$, equivalently $\vDash(\exists\bar{\alpha}.A) \wedge D \Rightarrow C$, $\vDash\exists\mathrm{FV}(A, D, C).A \wedge D \wedge C$ and $\vDash\mathcal{Q}.A[\bar{\alpha} := \bar{t}\,]$ for some $\bar{t}$. *Joint Constraint Abduction* under Q.P. handles several implications, i.e. $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)$, simultaneously. We need for each i: $\vDash(\exists\bar{\alpha}.A) \wedge D_i \Rightarrow C_i$, $\vDash\exists\mathrm{FV}(A, D_i, C_i).A \wedge D_i \wedge C_i$ and $\vDash\mathcal{Q}.A[\bar{\alpha} := \bar{t}\,]$ for some $\bar{t}$. *Constraint Disjunction Elimination* answer to a disjunction $\vee_i D_i$ of conjunctions of atoms is a solved form formula $\exists\bar{\alpha}.A$ such that for each i, $\vDash D_i \Rightarrow \exists\bar{\alpha}.A$. The task of constraint disjunction elimination is simple: in case of terms, it is anti-unification, and in case of linear inequalities, it is extended convex hull computation.

Short of enumerating all formulas, algorithms for finding any constraint abduction answer in the domain of (non-unary) free term algebra, and the domain of linear equations, are not known to the author. The task becomes easier when we restrict attention to *fully maximal answers* to $\mathcal{Q}.D \Rightarrow C$: those $\exists\bar{\alpha}.A$ for which $(\exists\bar{\alpha}.A \wedge D) \Leftrightarrow (C \wedge D)$. The algorithms look at various combinations of atoms from $D \wedge C$, and their "abstracted" variants.

Equipped with these tools, consider first solving for invariants – unary predicates $\chi(\cdot)$. We want the invariants to be as weak as possible, to make the use of the corresponding definitions as easy as possible: the weaker the invariant, the more general the type of definition. We perform joint constraint abduction, and divide the atoms of the answer $\exists\bar{\alpha}.A$ into solutions to the predicate variables A_χ and a remainder $A_{\mathrm{res}} = A \setminus \cup_\chi A_\chi$, depending on the variables in the atoms and so that the residuum holds under the quantifiers: $\vDash\mathcal{Q}.A_{\mathrm{res}}$. Note that a predicate takes only one variable $\chi(\beta_\chi)$ in premises. We substitute the result $\mathcal{Q}. \wedge_i (D_i \Rightarrow C_i)[\bar{\chi} := \overline{A_\chi[\beta_\chi := \delta]}]$ and repeat abduction – perform another iteration of the main algorithm – just in case some the occurrences of $\exists\alpha.\chi(\alpha) \wedge \Phi$ in conclusions, for example, bind α inside Φ with a term containing a universally quantified variable. It might be that the added constraints cannot all fit in next iteration's $\vDash\mathcal{Q}.A'_{\mathrm{res}}$ and have to be part of next iteration's A'_χ. It seems to never happen in practice.

For postconditions we want the strongest possible solutions, because stronger postcondition provides more information at use sites of a definition. Therefore we use disjunction elimination to initialize binary predicate variables $\chi_K(\cdot,\cdot)$ without "hurting" the constraint. If required to make the residuum hold: $\vDash \mathcal{Q}.A_{\text{res}}$, more atoms A_{χ_K} can be added to a postcondition. Detailed documentation of the algorithms can be found in [19].

4 Concluding Remarks

We have set out to develop an invariant and postcondition inference framework around constraint based type inference for GADTs, utilizing a formulation parametric w.r.t. the domain of constraints, leaving open what data properties can be expressed. For the difficult task of inference, rather than verification, of arbitrary invariants, we have given up decidability and principal types. Realizing that flexibility of invariant inference requires abstract postconditions, we have introduced implicitly generated existential types into the system.

As in traditional invariant inference, we allow the invariants be built in several iteration steps. It turns out abduction usually finds the invariants at once. For technical reasons – collecting all information, we only start inference for sorts other than terms in the second iteration. Some inference tasks, e.g.

$$\texttt{flatten\_pairs}: \forall \alpha, n[0 \leq n].\text{List}((\alpha,\alpha),n) \rightarrow \text{List}(\alpha, n+n)$$

require that our abduction algorithm, here for numerical equations, starts with non-recursive branches only, and with the bootstrapped solution considers all branches in the next iteration. But the reason is that our abduction algorithms are built on fully maximal simple constraint abduction. If any maximally general abduction answer could be considered, inference would again be solved in a single (i.e. in the second) iteration. One could try justifying this effectiveness of abduction by analysing what constraints are generated for recursive calls. On the other hand, given an oracle for joint constraint abduction problems, a formal argument could be made about semi-completeness of the solver (with oracle for abduction) for unary predicate variables (i.e. without existential types) in the single-sorted case, and correctness in general case. By correctness we mean that when the algorithm stops iterating, if it returns "not solvable", there is no answer, and if it returns an answer, it is a correct answer; by semi-completeness, that it stops if there is an answer.

In case of solving for both invariants and postconditions, the situation is more complex. The postconditions are not guaranteed to change monotonically between iterations. In practice, postconditions for terms are solved "at once", but convergence in the numerical domain has to be enforced by at some point (e.g. in 5th iteration) dropping the atoms that change between iterations. At the time of writing, inferring postconditions in *InvarGenT* is still work in progress. Moreover, the implementation of *InvarGenT* leaves plenty of opportunities for optimization.

The author wishes to thank patient anonymous reviewers of an early attempt to present these ideas.

Bibliography

[1] Peter Bulychev, Egor Kostylev and Vladimir Zakharov. Anti-unification algorithms and their applications in program analysis. In Amir Pnueli, Irina Virbitskaite and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11486-$1_3$5.

[2] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, aug 1977.

[3] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[4] Kenneth W. Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 505–519. Springer, 2007.

[5] Chuan-kai Lin. *Practical type inference for the GADT type system*. PhD dissertation, Portland State University, Department of Computer Science, 2010.

[6] Chuan-kai Lin and Tim Sheard. Pointwise generalized algebraic data types. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '10, pages 51–62. New York, NY, USA, 2010. ACM.

[7] Michael Maher. Herbrand constraint abduction. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 397–406. Washington, DC, USA, 2005. IEEE Computer Society.

[8] Michael Maher and Ge Huang. On computing constraint abduction answers. In Iliano Cervesato, Helmut Veith and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 421–435. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89439-$1_3$0.

[9] MichaelJ. Maher. Abduction of linear arithmetic constraints. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2005.

[10] G.D. Plotkin. A note on inductive generalization. *Machine Intelligence*, , 1969.

[11] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 232–244. New York, NY, USA, 2006. ACM.

[12] Yann Regis-Gianas and Francois Pottier. A Hoare logic for call-by-value functional programs. In *Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08)*, volume 5133 of *Lecture Notes in Computer Science*, pages 305–335. Springer, JUL 2008.

[13] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*. 1970.

[14] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 341–352. New York, NY, USA, 2009. ACM.

[15] Vincent Simonet and Francois Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems*, 29(1), JAN 2007.

[16] M. Sulzmann, T. Schrijvers and P. J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Manuscript, July 2006.

[17] Hiroshi Unno and Naoki Kobayashi. Dependent type inference with interpolants. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '09, pages 277–288. New York, NY, USA, 2009. ACM.

[18] Łukasz Stafiniak. Joint constraint abduction problems. 2011. The International Workshop on Unification.

[19] Łukasz Stafiniak. Invargent: implementation. Manuscript, 2013. https://github.com/lukstafi/invargent/blob/master/doc/invargent.pdf

5 Appendix

5.1 The GADT Type System

Set $\Delta := \exists\bar{\beta}\,[D].\Gamma$ and $\Delta' := \exists\bar{\beta}\,'[D'].\Gamma'$ such that $\bar{\beta}\,\#\mathrm{FV}(\Gamma')$, $\bar{\beta}\,'\#\mathrm{FV}(\Delta)$ and $\bar{\beta}\,'\#C$. Let $C \vDash \Delta' \leqslant \Delta$ denote $C \wedge D' \vDash \exists\bar{\beta}\,.(D \wedge_{x\in\mathrm{Dom}(\Gamma)} \Gamma(x) \doteq \Gamma'(x))$ when $\mathrm{Dom}(\Gamma) = \mathrm{Dom}(\Gamma')$, and otherwise a falsehood (compare lemma 3.5 of [15]). Let $\Delta \times \Delta'$ denote $\exists\bar{\beta}\bar{\beta}'[D \wedge D'].\Gamma\dot{\cup}\Gamma'$, and $\exists\bar{\beta}\,'[D']\Delta$ denote $\exists\bar{\beta}\bar{\beta}'[D \wedge D'].\Gamma$.

Proposition 6. *Properties of environment fragments (see [15] lemma 3.15).*

f-Hide. $\vDash \Delta \leqslant \exists\bar{\alpha}\,.\Delta$.

f-Imply. $C_1 \Rightarrow C_2 \vDash [C_1]\Delta \leqslant [C_2]\Delta$.

f-Enrich. $C \Rightarrow \Delta_1 \leqslant \Delta_2 \vDash [C]\Delta_1 \leqslant [C]\Delta_2$.

f-Ex. $\forall\bar{\alpha}\,.\Delta_1 \leqslant \Delta_2 \vDash (\exists\bar{\alpha}\,.\Delta_1) \leqslant (\exists\bar{\alpha}\,.\Delta_2)$.

f-And. $\Delta_1 \leqslant \Delta_2 \vDash \Delta \times \Delta_1 \leqslant \Delta \times \Delta_2$.

Proposition 7. *Constructor* $K :: \forall\bar{\alpha}\bar{\beta}\,[D].\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha}\,)$ *where* $D = \exists\bar{\beta}\,'.A$, *is equivalent to* $K :: \forall\bar{\alpha}\bar{\gamma_i}[\exists\bar{\beta}\bar{\beta}\,'.\bar{\gamma_i} \doteq \bar{\tau_i} \wedge A].\gamma_1 \times \ldots \times \gamma_n \to \varepsilon(\bar{\alpha}\,)$.

Proposition 8. *Constructors of the form* $K :: \forall\bar{\alpha_i}\bar{\beta}\,[D].\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha_i})$ *where* $D = \exists\bar{\beta}\,'.A$, *are equivalent to constructors of the form* $K :: \forall\alpha\bar{\beta}\,[\exists\bar{\alpha_i}\bar{\beta}\,'.\alpha \doteq \alpha_1 \to \ldots \to \alpha_m \wedge A].\gamma_1 \times \ldots \times \gamma_n \to \varepsilon(\alpha)$ *when all uses of* $\varepsilon(\tau_1, \ldots, \tau_m)$ *are translated to* $\varepsilon(\tau_1 \to \ldots \to \tau_m)$.

Lemma 9. Weakening *(patterns and expressions). Assume* $C_1 \vDash C_2$. *If* $C_2 \vdash p$: $\tau \longrightarrow \Delta$ *(resp.* $C_2, \Gamma \vdash \mathrm{ce}\colon \tau$, $C_2, \Gamma \vdash \mathrm{ce}\colon \sigma$*) is derivable, then there exists a derivation of* $C_1 \vdash p\colon \tau \longrightarrow \Delta$ *(resp.* $C_1, \Gamma \vdash \mathrm{ce}\colon \tau$, $C_1, \Gamma \vdash \mathrm{ce}\colon \sigma$*) of the same structure.*

The lemma follows from transitivity of $\vDash$ ($A \vDash B$ and $B \vDash C$ imply $A \vDash C$) by induction on the structure of the derivation.

Lemma 10. *If* $\Sigma \subset \Sigma'$ *and* $C \vdash p\colon \tau \longrightarrow \Delta$ *(resp.* $C, \Gamma \vdash \mathrm{ce}\colon \tau$, $C, \Gamma \vdash \mathrm{ce}\colon \sigma$*) is derivable with constructors* Σ*, then the same derivation works with constructors* Σ'.

Lemma 11. Correctness *(patterns).* $[\![\vdash p\downarrow\tau]\!] \vdash p\colon \tau \longrightarrow [\![\vdash p\uparrow\tau]\!]$.

Proof. By induction on the structure of p.

- Cases 0, 1 and x: follow directly from `p-Empty`, `p-Wild` and `p-Var` respectively.
- Case $p_1 \wedge p_2$.
    1. By the induction hypothesis, $[\![\vdash p_i\downarrow\tau]\!] \vdash p_i\colon \tau \longrightarrow [\![\vdash p_i\uparrow\tau]\!]$ for $i = 1, 2$.
    2. By weakening and `p-And` we have the goal.

- Case $Kp_1...p_n$.
  1. Let $\Sigma \ni K :: \forall\bar{\alpha}\bar{\beta}\,[D].\tau_1 \times ... \times \tau_n \rightarrow \varepsilon(\bar{\alpha}\,)$.
  2. By the induction hypothesis, $[\![\vdash p_i{\downarrow}\tau_i]\!] \vdash p_i\colon \tau_i \longrightarrow [\![\vdash p_i{\uparrow}\tau_i]\!]$ for $i = 1, ..., n$.
  3. The `p-Cstr` rule says $\forall i\ (C \wedge D \vdash p_i\colon \tau_i \longrightarrow \Delta_i)/_{\text{p-Cstr}} C \vdash p\colon \varepsilon(\bar{\alpha}\,) \longrightarrow \exists\bar{\beta}\,[D](\Delta_1 \times ... \times \Delta_n)$, where $\Delta_i := [\![\vdash p_i{\uparrow}\tau_i]\!]$. Applying it to (2) we get $C \vdash p\colon \varepsilon(\bar{\alpha}\,) \longrightarrow \exists\bar{\beta}\,[D](\Delta_1 \times ... \times \Delta_n)$ as long as $C \wedge D \vDash [\![\vdash p_i{\downarrow}\tau_i]\!]$.
  4. Let $\bar{\alpha}\,'\bar{\beta}\,'\#\mathrm{FV}(\Sigma, \tau)$ and $\tau_i' := \tau_i[\bar{\alpha}\bar{\beta} := \bar{\alpha}\,'\bar{\beta}\,'], D' := D[\bar{\alpha}\bar{\beta} := \bar{\alpha}\,'\bar{\beta}\,']$. Let Δ_i' be Δ_i with unbound occurrences of $\bar{\alpha}\bar{\beta}$ renamed to $\bar{\alpha}\,'\bar{\beta}\,'$.
  5. By weakening and `p-EqIn`, (3) gives $\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}\,'\bar{\beta}\,' \wedge \varepsilon(\bar{\alpha}\,') \doteq \tau \wedge C \vdash p\colon \tau \longrightarrow \exists\bar{\beta}\,[D](\Delta_1 \times ... \times \Delta_n)$.
  6. By proposition 6, transitivity of $\leqslant$, and `p-SubOut`, we get $\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}\,'\bar{\beta}\,' \wedge \varepsilon(\bar{\alpha}\,') \doteq \tau \wedge C \vdash p\colon \tau \longrightarrow \exists\bar{\alpha}\,'\bar{\beta}\,'[D'](\Delta_1' \times ... \times \Delta_n')$.
  7. By applying `p-Hide` to (6) with $C = \bar{\alpha} \doteq \bar{\alpha}\,' \wedge \forall\bar{\beta}\,'.D' \Rightarrow \wedge_i [\![\vdash p_i{\downarrow}\tau_i']\!]$ and weakening, since w.l.o.g. $\bar{\alpha}\bar{\beta}$ do not appear unbound in the goal, and $C \wedge D \vDash [\![\vdash p_i{\downarrow}\tau_i]\!]$, we get the goal $\exists\bar{\alpha}\,'.\varepsilon(\bar{\alpha}\,') \doteq \tau \wedge \forall\bar{\beta}\,'.D' \Rightarrow \wedge_i [\![\vdash p_i{\downarrow}\tau_i']\!] \vdash p\colon \tau \longrightarrow \exists\bar{\alpha}\,'\bar{\beta}\,'[D'](\Delta_1' \times ... \times \Delta_n')$. □

Proof of theorem 1.

Proof. By induction on the structure of ce.

- Case ce is x.
  1. If $x \notin \mathrm{Dom}(\Gamma)$, then the goal follows by applying `FElim`. Otherwise, let $\Gamma(x)$ be $\forall\beta[\exists\bar{\alpha}.D].\beta$. By `Var`, $D, \Gamma \vdash x\colon \beta$.
  2. Let $\beta'\bar{\alpha}\,'\#\mathrm{FV}(\Gamma, \tau)$. By (1), weakening and `Equ`, $\beta\bar{\alpha} \doteq \beta'\bar{\alpha}\,' \wedge D' \wedge \beta' \doteq \tau, \Gamma \vdash x\colon \tau$, where $D' := D[\beta\bar{\alpha} := \beta'\bar{\alpha}\,']$.
  3. By `Hide` and weakening, since w.l.o.g. $\beta\bar{\alpha}$ do not appear unbounded in the goal, this implies the goal $\exists\beta'\bar{\alpha}\,'.(D' \wedge \beta' \doteq \tau), \Gamma \vdash x\colon \tau$.
- Case ce is $\lambda\bar{c}$ where $\bar{c} = (c_1, ..., c_n)$.
  1. Let $\alpha_1\alpha_2\#\mathrm{FV}(\Gamma, \tau)$.
  2. Induction hypothesis yelds $[\![\Gamma \vdash c_i\colon \alpha_1 \rightarrow \alpha_2]\!], \Gamma \vdash c_i\colon \alpha_1 \rightarrow \alpha_2$.
  3. By (2), weakening and `Abs`, $[\![\Gamma \vdash \bar{c} : \alpha_1 \rightarrow \alpha_2]\!], \Gamma \vdash \lambda\bar{c} : \alpha_1 \rightarrow \alpha_2$.
  4. By weakening and `Equ`, (3) implies $[\![\Gamma \vdash \bar{c} : \alpha_1 \rightarrow \alpha_2]\!] \wedge \alpha_1 \rightarrow \alpha_2 \doteq \tau$, $\Gamma \vdash \lambda\bar{c} : \tau$.
  5. By (1) and `Hide`, this implies $[\![\Gamma \vdash \lambda\bar{c} : \tau]\!], \Gamma \vdash \lambda\bar{c} : \tau$.
- Case ce is $e_1\, e_2$.
  1. Let $\alpha\#\mathrm{FV}(\Gamma, \tau)$.

2. By the induction hypothesis, we have $[\![\Gamma \vdash e_1 \colon \alpha \to \tau]\!], \Gamma \vdash e_1 \colon \alpha \to \tau$ and $[\![\Gamma \vdash e_2 \colon \alpha]\!], \Gamma \vdash e_2 \colon \alpha$.
3. By weakening and `App`, this yields $[\![\Gamma \vdash e_1 \colon \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 \colon \alpha]\!]$, $\Gamma \vdash e_1\, e_2 \colon \tau$.
4. By `Hide` using (1), $[\![\Gamma \vdash e_1\, e_2 \colon \tau]\!], \Gamma \vdash e_1\, e_2 \colon \tau$.

– Case ce is $K\, e_1 \dots e_n$.

1. Let $\Sigma \ni K :: \forall \bar{\alpha}\bar{\beta}\, [D].\tau_1 \times \dots \times \tau_n \to \varepsilon(\bar{\alpha})$.
2. By induction hypothesis and weakening for each $i = 1, \dots, n$
$$\wedge_j [\![\Gamma \vdash e_j \colon \tau_j]\!] \wedge D \wedge \varepsilon(\bar{\alpha}) \doteq \tau, \Gamma \vdash e_i \colon \tau_i$$
3. Applying `Cstr` to (1) and (3) we obtain
$$\wedge_i [\![\Gamma \vdash e_i \colon \tau_i]\!] \wedge D \wedge \varepsilon(\bar{\alpha}) \doteq \tau, \Gamma \vdash K\, e_1 \dots e_n \colon \varepsilon(\bar{\alpha})$$
4. Let $\bar{\alpha}'\bar{\beta}'\#\mathrm{FV}(\Gamma, \tau)$ and $\tau_i' := \tau_i[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}']$, $D' := D[\beta\bar{\alpha} := \beta'\bar{\alpha}']$.
$$\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}'\bar{\beta}' \wedge_i [\![\Gamma \vdash e_i \colon \tau_i']\!] \wedge D \wedge \varepsilon(\bar{\alpha}') \doteq \tau, \Gamma \vdash K\, e_1 \dots e_n \colon \varepsilon(\bar{\alpha}')$$
5. By `Equ`, (1) `Hide` and weakening, since w.l.o.g. $\bar{\alpha}\bar{\beta}$ do not appear unbounded in the goal, $[\![\Gamma \vdash K\, e_1 \dots e_n \colon \tau]\!], \Gamma \vdash K\, e_1 \dots e_n \colon \tau$.

– Case ce is $\mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2$.

1. Let $\alpha\beta\#\mathrm{FV}(\Gamma, \tau)$ and $\chi\#\mathrm{PV}(\Gamma)$.
2. Let $\sigma = \forall\beta[\chi(\beta)].\beta$, $\Gamma' = \Gamma\{x \mapsto \sigma\}$. By the induction hypothesis, $[\![\Gamma' \vdash e_1 \colon \beta]\!], \Gamma' \vdash e_1 \colon \beta$ and $[\![\Gamma' \vdash e_2 \colon \tau]\!], \Gamma' \vdash e_2 \colon \tau$.
3. Let $D = \forall\beta.(\chi(\beta) \Rightarrow [\![\Gamma' \vdash e_1 \colon \beta]\!])$. Since $D \wedge \chi(\beta)$ implies $[\![\Gamma' \vdash e_1 \colon \beta]\!]$, by weakening of (2), we have $D \wedge \chi(\beta), \Gamma' \vdash e_1 \colon \beta$. From (1) we have $\alpha\#\mathrm{FV}(D, \Gamma', \tau)$, by `Gen` we have $D \wedge \exists\beta.\chi(\beta)$, $\Gamma' \vdash e_1 \colon \forall\beta[\chi(\beta)].\beta$, by (1) and renaming we have
$$D \wedge \exists\alpha.\chi(\alpha), \Gamma' \vdash e_1 \colon \sigma.$$
4. By weakening of both (2) and (3), and by `LetRec`, we have $[\![\Gamma \vdash \mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2 \colon \tau]\!], \Gamma \vdash \mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2 \colon \tau$.

– Case ce is $p.e$.

1. τ is of the form $\tau_1 \to \tau_2$. Write $[\![\vdash p\uparrow\tau_1]\!]$ as $\exists\bar{\beta}\,[D]\Gamma'$, where $\bar{\beta}\,\#\mathrm{FV}(\Gamma, \tau_1, \tau_2)$.
2. By induction hypothesis, $[\![\Gamma\Gamma' \vdash e \colon \tau_2]\!], \Gamma\Gamma' \vdash e \colon \tau_2$.
3. By lemma 11 and (1), we have $[\![\vdash p\downarrow\tau_1]\!] \vdash p \colon \tau_1 \longrightarrow \exists\bar{\beta}\,[D]\Gamma'$.
4. By instantiation of $\bar{\beta}$ and weakening, (2) implies
$$[\![\Gamma \vdash p.e \colon \tau]\!] \wedge D, \Gamma\Gamma' \vdash e \colon \tau_2$$

5. By weakening, (3) implies $[\![\Gamma \vdash p.e\colon \tau]\!] \vdash p\colon \tau_1 \longrightarrow \exists\bar\beta\,[D]\Gamma'$.
6. By (4), (5), (1), and `Clause`, we obtain $[\![\Gamma \vdash p.e\colon \tau]\!], \Gamma \vdash p.e\colon \tau$. $\square$

$\Gamma' \doteq \Gamma''$ stands for $\forall x \in \mathrm{Dom}(\Gamma') \cup \mathrm{Dom}(\Gamma'').\Gamma'(x) \doteq \Gamma''(x)$ and is false when $\mathrm{Dom}(\Gamma') \neq \mathrm{Dom}(\Gamma'')$. Recall that for $\Delta := \exists\bar\beta\,[D].\Gamma$ and $\Delta' := \exists\bar\beta\,'[D'].\Gamma'$ such that $\bar\beta\,\#\mathrm{FV}(\Gamma')$, $\bar\beta\,'\#\mathrm{FV}(\Delta)$ and $\bar\beta\,'\#C$, $C \vDash \Delta' \leqslant \Delta$ denotes $C \wedge D' \vDash \exists\bar\beta\,.D \wedge \Gamma \doteq \Gamma'$. Observe, that $C \vDash \Delta' \leqslant \Delta$ iff $C \vDash \forall\bar\beta\,'.D' \Rightarrow \exists\bar\beta\,.D \wedge \Gamma \doteq \Gamma'$.

Lemma 12. Completeness *(patterns). Let* $\Delta = \exists\bar\beta\,'[D']\Gamma'$ *and* $[\![\vdash p\uparrow\tau]\!] = \exists\bar\beta\,''[D'']\Gamma'' = \Delta'$. $C \vdash p\colon \tau \longrightarrow \Delta$ *implies* $C \vDash [\![\vdash p\downarrow\tau]\!]$ *and* $C \vDash \forall\bar\beta\,''.D'' \Rightarrow \exists\bar\beta\,'.(D' \wedge \Gamma'' \doteq \Gamma')$, *i.e.* $C \vDash \Delta' \leqslant \Delta$.

Proof. By induction on the derivation of $C \vdash p\colon \tau \longrightarrow \Delta$. To slightly simplify the proof, the induction is actually on the lexicographic ordering: (# of applications of `p-Cstr`, # of other rules applications).

- Cases `p-Empty`, `p-Wild`, `p-Var`. $[\![\vdash p\downarrow\tau]\!] = \boldsymbol{T}$. $[\![\vdash p\uparrow\tau]\!]$ and Δ coincide: $\Gamma'' = \Gamma'$, $D' = D'' = \boldsymbol{T}$ and $\vDash \exists\bar\beta\,.\Gamma' \doteq \Gamma'$ holds because sorts are nonempty.
- Case `p-And`. In this case $\Delta = \Delta_1 \times \Delta_2$, $\bar\beta\,' = \bar\beta_1'\bar\beta_2'$, $D' = D_1' \wedge D_2'$, $\Gamma' = \Gamma_1' \dot\cup \Gamma_2'$.
  1. `p-And`'s premises are $C \vdash p_i\colon \tau \longrightarrow \Delta_i$, which by induction hypothesis gives $C \vDash [\![\vdash p_i\downarrow\tau]\!]$ and $C \vDash \forall\bar\beta_i''.D_i'' \Rightarrow \exists\bar\beta_i'.(D_i' \wedge \Gamma_i'' \doteq \Gamma_i')$ for $i = 1, 2$.
  2. (1) gives $C \vDash [\![\vdash p_1 \wedge p_2\downarrow\tau]\!]$ as $[\![\vdash p_1 \wedge p_2\downarrow\tau]\!] = [\![\vdash p_1\downarrow\tau]\!] \wedge [\![\vdash p_2\downarrow\tau]\!]$.
  3. $[\![\vdash p_1 \wedge p_2\uparrow\tau]\!] = [\![\vdash p_1\uparrow\tau]\!] \times [\![\vdash p_2\uparrow\tau]\!] = \exists\bar\beta_1''\bar\beta_2''[D_1'' \wedge D_2'']\Gamma_1'' \dot\cup \Gamma_2''$. We will show $C \vDash \forall\bar\beta_1''\bar\beta_2''.D_1'' \wedge D_2'' \Rightarrow \exists\bar\beta_1'\bar\beta_2'.(D_1' \wedge D_2' \wedge \Gamma_1'' \dot\cup \Gamma_2'' \doteq \Gamma_1' \dot\cup \Gamma_2')$.
  4. Assume w.l.o.g. $\bar\beta_1'\#\bar\beta_2'$, $\bar\beta_1''\#\bar\beta_2''$. Applying (1) for $i = 1, 2$ gives $C \vDash \forall\bar\beta_1''\bar\beta_2''.D_1'' \wedge D_2'' \Rightarrow \exists\bar\beta_1'\bar\beta_2'.(D_1' \wedge D_2' \wedge \Gamma_1'' \doteq \Gamma_1' \wedge \Gamma_2'' \doteq \Gamma_2')$, which completes the goal.
- Case `p-Cstr`. In this case $\Delta = \exists\bar\beta_0[D_0](\Delta_1 \times \ldots \times \Delta_n)$, and $\tau = \varepsilon(\bar\alpha_0)$, where $D_0 := D_K[\bar\alpha\bar\beta := \bar\alpha_0\bar\beta_0]$ for $\Sigma \ni K :: \forall\bar\alpha\bar\beta\,[D_K].\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar\alpha)$ and $\bar\beta_0\#\mathrm{FV}(C)$.
  1. `p-Cstr`'s premises are $C \wedge D_0 \vdash p_i\colon \tau_i[\bar\alpha\bar\beta := \bar\alpha_0\bar\beta_0] \longrightarrow \Delta_i$.
  2. Let $\bar\alpha_0'\bar\beta_0'\#\mathrm{FV}(\tau, \bar\alpha\bar\beta\,, \bar\alpha_0\bar\beta_0, C)$.
  3. Let $\tau_i' := \tau_i[\bar\alpha\bar\beta := \bar\alpha_0'\bar\beta_0']$. By weakening and `p-EqIn`, (1) gives $C \wedge D_0 \wedge \bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \vdash p_i\colon \tau_i' \longrightarrow \Delta_i$.
  4. By induction hypothesis we have $C \wedge D_0 \wedge \bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \vDash [\![\vdash p_i\downarrow\tau_i']\!]$ and $C \wedge D_0 \wedge \bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \vDash \forall\bar\beta_i''.D_i'' \Rightarrow \exists\bar\beta_i'.(D_i' \wedge \Gamma_i'' \doteq \Gamma_i')$ for $i = 1, \ldots, n$.
  5. Let $D_0' := D_K[\bar\alpha\bar\beta := \bar\alpha_0'\bar\beta_0']$. From (4) follows $C \wedge \bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \vDash D_0' \Rightarrow \wedge_i [\![\vdash p_i\downarrow\tau_i']\!]$.

6. W.l.o.g. $\bar\alpha_0\bar\beta_0\#\mathrm{FV}(D_0' \Rightarrow \wedge_i[\![\vdash p_i{\downarrow}\tau_i']\!])$. (5) gives $C \wedge \bar\alpha_0 \doteq \bar\alpha_0' \vDash \forall\bar\beta_0'.D_0' \Rightarrow \wedge_i[\![\vdash p_i{\downarrow}\tau_i']\!]$ because we can drop $\bar\beta_0 \doteq \bar\beta_0'$ from premises.
7. (6) is equivalent to $C \wedge \bar\alpha_0 \doteq \bar\alpha_0' \vDash \varepsilon(\bar\alpha_0) \doteq \varepsilon(\bar\alpha_0') \wedge \forall\bar\beta_0'.D_0' \Rightarrow \wedge_i[\![p_i{\downarrow}\tau_i']\!]$ which by the nonempty domain property implies $C \wedge \bar\alpha_0 \doteq \bar\alpha_0' \vDash \exists\bar\alpha_0'.\varepsilon(\bar\alpha_0) \doteq \varepsilon(\bar\alpha_0') \wedge \forall\bar\beta_0'.D_0' \Rightarrow \wedge_i[\![\vdash p_i{\downarrow}\tau_i']\!]$.
8. Because by (6) we can drop $\bar\alpha_0 \doteq \bar\alpha_0'$ from premises, (7) is equivalent to $C \vDash \exists\bar\alpha_0'.\varepsilon(\bar\alpha_0) \doteq \varepsilon(\bar\alpha_0') \wedge \forall\bar\beta_0'.D_0' \Rightarrow \wedge_i[\![\vdash p_i{\downarrow}\tau_i']\!]$, which is the first part of the goal.
9. From (4), $C \wedge \bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \vDash D_0' \Rightarrow \forall\bar\beta_1''...\bar\beta_n''.\ \wedge_i\ D_i'' \Rightarrow \exists\bar\beta_1'...\bar\beta_n'.\wedge_i (D_i' \wedge \Gamma_i'' \doteq \Gamma_i')$.
10. From (9) by (2) and (6), $C \vDash \forall\bar\alpha_0'\bar\beta_0'.\bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \wedge D_0' \Rightarrow \forall\bar\beta_1''...\bar\beta_n''.\ \wedge_i\ D_i'' \Rightarrow \exists\bar\beta_1'...\bar\beta_n'.\ \wedge_i\ (D_i' \wedge \Gamma_i'' \doteq \Gamma_i')$, which is equivalent to

$$C \vDash \forall\bar\alpha_0'\bar\beta_0'\bar\beta_1''...\bar\beta_n''.\bar\alpha_0\bar\beta_0 \doteq \bar\alpha_0'\bar\beta_0' \wedge D_0' \wedge_i D_i'' \Rightarrow \exists\bar\beta_1'...\bar\beta_n'.\wedge_i (D_i' \wedge \Gamma_i'' \doteq \Gamma_i')$$

11. Observe, that w.l.o.g. $\bar\beta'' := \bar\alpha_0'\bar\beta_0'\bar\beta_1''...\bar\beta_n''$. Note by definition of $[\![\vdash p{\uparrow}\tau]\!]$, that $D'' = \varepsilon(\bar\alpha_0) \doteq \varepsilon(\bar\alpha_0') \wedge D_0' \wedge_i D_i''$. By the free generation property, $\vDash D'' \Rightarrow \bar\alpha_0 \doteq \bar\alpha_0'$.
12. Observe, that $\Gamma'' \doteq \Gamma' \equiv \wedge_i(\Gamma_i'' \doteq \Gamma_i')$ and $D' = D_0 \wedge_i D_i'$. (10) and (11) imply

$$C \vDash \forall\bar\beta''.\bar\beta_0 \doteq \bar\beta_0' \wedge D'' \Rightarrow \exists\bar\beta_1'...\bar\beta_n'.D' \wedge \Gamma'' \doteq \Gamma'$$

13. Also, $\bar\beta' = \bar\beta_0\bar\beta_1'...\bar\beta_n'$. Because $\bar\beta_0\#\mathrm{FV}(D'')$, because sorts are nonempty (12) gives $C \vDash \forall\bar\beta''.\bar\alpha_0 \doteq \bar\alpha_0' \wedge D'' \Rightarrow \exists\bar\beta'.D' \wedge \Gamma'' \doteq \Gamma'$, the other part of the goal.

– Case `p-EqIn`.
    1. `p-EqIn`'s premises are: $C \vdash p\colon \tau' \longrightarrow \Delta$, which by induction hypothesis gives $C \vDash [\![\vdash p{\downarrow}\tau']\!]$ and $C \vDash \Delta_1' \leqslant \Delta$, for $\Delta_1' = \exists\bar\beta_1''[D_1'']\Gamma_1''$
    2. and $C \vDash \tau \doteq \tau'$.
    3. Observe by induction on p, that $C \wedge \tau \doteq \tau' \vDash [\![\vdash p{\downarrow}\tau']\!]$ iff $C \wedge \tau \doteq \tau' \vDash [\![\vdash p{\downarrow}\tau]\!]$, which by (1) and (2) gives the first part of the goal.
    4. Observe by induction on p, that $C \wedge \tau \doteq \tau' \vDash [\![\vdash p{\uparrow}\tau]\!] \leqslant [\![\vdash p{\uparrow}\tau']\!]$, i.e. $C \wedge \tau \doteq \tau' \vDash \Delta' \leqslant \Delta_1'$, which by (1), (2) and transitivity of $\leqslant$, proves the second part of the goal.

– Case `p-SubOut` follows by transitivity of $\leqslant$.

– Case `p-Hide`.
    1. `p-Hide`'s premises are $C' \vdash p\colon \tau \longrightarrow \Delta$ and $\bar\alpha_0\#\mathrm{FV}(\tau,\ \Delta)$ for $C = \exists\bar\alpha_0.C'$.

2. By inductive hypothesis, $C' \vDash [\![\vdash p \downarrow \tau]\!]$ and $C' \vDash \Delta' \leqslant \Delta$.
3. By induction on p, $\mathrm{FV}([\![\vdash p \downarrow \tau]\!]) = \mathrm{FV}(\tau)$.
4. By (1), (2) and (3) we have $C \vDash [\![\vdash p \downarrow \tau]\!]$.
5. By induction on p, $\mathrm{FV}(D'', \Gamma'') \subseteq \mathrm{FV}(\tau) \cup \bar{\beta}''$.
6. By (1), (2) and (3) we have $C \vDash \Delta' \leqslant \Delta$.

□

Lemma 13. *Let Γ be an environment and Γ', Γ'' be simple (i.e. monomorphic) environments. For any e, τ, $C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma' \vdash e \colon \tau$ iff $C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma'' \vdash e \colon \tau$.*

Proof. Consider a derivation of $C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma' \vdash e \colon \tau$. The only case where Γ' is referred to, is in the `Var` rule, which for a monomorphic environment simplifies to: $\Gamma'(x) = \tau' / C, \Gamma\Gamma' \vdash x \colon \tau'$. Replace Γ' with Γ'' in judgements throughout the derivation. $\Gamma'(x) = \tau' /_{\mathrm{Var}} C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma'' \vdash x \colon \tau'$ is not valid, correct it as $\Gamma''(x) = \tau'' /_{\mathrm{Var}} C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma'' \vdash x \colon \tau'' /_{\mathrm{Equ}} C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma'' \vdash x \colon \tau'$. Analogically follows the other direction of the equivalence of $C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma' \vdash e \colon \tau$ and $C \wedge \Gamma' \doteq \Gamma'', \Gamma\Gamma'' \vdash e \colon \tau$. □

Proof of theorem 2.

Proof. We proceed by induction on the derivation of $C, \Gamma \vdash \mathrm{ce} \colon \tau$. To slightly simplify the proof, the induction is actually on the lexicographic ordering: (# of structural rule applications `Var`, `Cstr`, `Abs`, `App`, `LetRec`, `Clause`; # of non-structural rule applications `Equ`, `Hide`, `FElim`, `DisjElim`). (The rules `FElim` and `DisjElim` are not needed when deriving the syntax-directed rules.)

- Case `Var`.
  1. `Var`'s first premise is $\Gamma(x) = \forall \beta [\exists \bar{\alpha}. D]. \beta$.
  2. `Var`'s second premise is $C \vDash D$.
  3. The goal is: $\mathcal{I}, C \vDash \exists \beta' \bar{\alpha}'. (D[\beta \bar{\alpha} := \beta' \bar{\alpha}'] \wedge \beta' \doteq \tau)$, where w.l.o.g. $\beta' \bar{\alpha}' \# \mathrm{FV}(C, \Gamma, \tau, \beta, \bar{\alpha})$.
  4. (3) follows from (2) by instantiating β to τ, because we assume that all sorts in $\mathcal{M}$ are non-empty. We can take an empty interpretation $\mathcal{I} = \epsilon$.
- Case `Cstr`.
  1. `Cstr`'s premises are $C, \Gamma \vdash e_i \colon \tau_i$, $i = 1, \ldots, n$, $C \vDash D$ and $K :: \forall \bar{\alpha} \bar{\beta} [D]. \tau_1 \ldots \tau_n \rightarrow \varepsilon(\bar{\alpha})$. $\tau = \varepsilon(\bar{\alpha})$.
  2. Let w.l.o.g. $\bar{\alpha}' \bar{\beta}' \# \mathrm{FV}(C, \Gamma, \tau)$. By weakening and `Equ`, (1) gives $C \wedge \bar{\alpha}' \bar{\beta}' \doteq \bar{\alpha} \bar{\beta}, \Gamma \vdash e_i \colon \tau_i [\bar{\alpha} \bar{\beta} := \bar{\alpha}' \bar{\beta}']$.
  3. Let $\Phi_i = [\![\Gamma \vdash e_i \colon \tau_i [\bar{\alpha} \bar{\beta} := \bar{\alpha}' \bar{\beta}']]\!]$. By induction hypothesis, $\mathcal{I}_i$, $C \wedge \bar{\alpha}' \bar{\beta}' \doteq \bar{\alpha} \bar{\beta} \vDash \Phi_i$, $i = 1, \ldots, n$.

4. Observe, that (1) and (3) imply $\mathcal{I}_i$, $C \wedge \bar{\alpha}'\bar{\beta}' \doteq \bar{\alpha}\bar{\beta} \vDash \wedge_i \Phi_i \wedge D[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}'] \wedge \varepsilon(\bar{\alpha}') \doteq \varepsilon(\bar{\alpha})$.
5. By non-emptiness of sorts and because the premise $\mathrm{PV}(C, \Gamma) = \varnothing$ gives disjoint domains for the $\mathcal{I}_i$, (4) and (2) imply $\mathcal{I}_1 ... \mathcal{I}_n$, $C \vDash \exists \bar{\alpha}'\bar{\beta}'. \wedge_i \Phi_i \wedge D[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}'] \wedge \varepsilon(\bar{\alpha}') \doteq \varepsilon(\bar{\alpha})$.
6. By (1) and (5), $\mathcal{I}, C \vDash [\![\Gamma \vdash K e_1 ... e_n : \tau]\!]$ for $\mathcal{I} = \mathcal{I}_1 ... \mathcal{I}_n$.

– Case `Abs`. In this case, $\tau := \tau_1 \rightarrow \tau_2$.

1. `Abs`' premise is $C, \Gamma \vdash \bar{c} : \tau_1 \rightarrow \tau_2$, which by induction hypothesis implies $\mathcal{I}_i, C \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash p_i.e_i : \tau_1 \rightarrow \tau_2]\!]$, $i = 1, ..., n$.
2. Let $\alpha_1 \alpha_2 \# \mathrm{FV}(C, \tau_1, \tau_2)$. Then, because sorts are nonempty, $C \vDash \exists \alpha_1 \alpha_2.(C \wedge \alpha_1 \doteq \tau_1 \wedge \alpha_2 \doteq \tau_2)$.
3. (1) and the premise implies $\mathcal{I}_1 \mathcal{I}_2$, $C \wedge \alpha_1 \doteq \tau_1 \wedge \alpha_2 \doteq \tau_2 \vDash \wedge_i \Phi_i \wedge \alpha_1 \rightarrow \alpha_2 \doteq \tau_1 \rightarrow \tau_2$.
4. Combining (2) and (3), $\mathcal{I}_1 \mathcal{I}_2, C \vDash \exists \alpha_1 \alpha_2.(\wedge_i \Phi_i \wedge \alpha_1 \rightarrow \alpha_2 \doteq \tau_1 \rightarrow \tau_2)$.
5. By (1) and (4), $\mathcal{I}_1 \mathcal{I}_2, C \vDash [\![\Gamma \vdash \lambda \bar{c} : \tau]\!]$.

– Case `App`.

1. `App`'s premises are $C, \Gamma \vdash e_1 : \tau' \rightarrow \tau$ and $C, \Gamma \vdash e_2 : \tau'$.
2. Pick w.l.o.g. $\alpha \notin \mathrm{FV}(C, \tau', \Gamma, \tau)$. By rule `Equ`, (1) implies $C \wedge \alpha \doteq \tau'$, $\Gamma \vdash e_1 : \alpha \rightarrow \tau$ and $C \wedge \alpha \doteq \tau', \Gamma \vdash e_2 : \alpha$.
3. By induction hypothesis, (2) implies $\mathcal{I}_i$, $C \wedge \alpha \doteq \tau' \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash e_i : \tau_i]\!]$, $i = 1, 2, \tau_1 := \tau' \rightarrow \tau, \tau_2 := \tau'$.
4. By (2) and nonemptiness of sorts, we have $C \vDash \exists \alpha.(C \wedge \alpha \doteq \tau')$.
5. By (3), the premise and because $C \vDash D$ implies $\exists \alpha.C \vDash \exists \alpha.D$, we have $\mathcal{I}_1 \mathcal{I}_2, \exists \alpha.(C \wedge \alpha \doteq \tau') \vDash \exists \alpha.(\Phi_1 \wedge \Phi_2)$.
6. By (4) and (5), we have the goal $\mathcal{I}, C \vDash [\![\Gamma \vdash e_1\, e_2 : \tau]\!]$ with $\mathcal{I} = \mathcal{I}_1 \mathcal{I}_2$.

– Case `LetRec`. Let $\Gamma' := \Gamma\{x \mapsto \sigma\}$.

1. `LetRec`'s premises are C, $\Gamma' \vdash e_1 : \sigma$, which can only be derived by `Gen` from $C' \wedge D$, $\Gamma' \vdash e_1 : \beta$, where $\sigma = \forall \beta[\exists \bar{\alpha}.D].\beta$ and $C = C' \wedge \exists \beta \bar{\alpha}.D$; by induction hypothesis we get $\mathcal{I}_1, C' \wedge D \vDash \Phi_1$ for $\Phi_1 = [\![\Gamma' \vdash e_1 : \beta]\!]$;
2. and C, $\Gamma' \vdash e_2 : \tau$; by induction hypothesis we get $\mathcal{I}_2$, $C \vDash \Phi_2$ for $\Phi_2 = [\![\Gamma' \vdash e_2 : \tau]\!]$.
3. $\beta \bar{\alpha} \# \mathrm{FV}(\Gamma, C')$. W.l.o.g., assume additionally that $\beta \bar{\alpha} \# \mathrm{FV}(\tau)$.
4. $\mathcal{I}_1$, $C' \vDash \forall \beta.(\exists \bar{\alpha}.D) \Rightarrow \Phi_1$ iff $\mathcal{I}_1$, $C' \vDash (\exists \bar{\alpha}.D) \Rightarrow \Phi_1$ iff $\mathcal{I}_1$, $C' \vDash \forall \bar{\alpha}.D \Rightarrow \Phi_1$ iff $\mathcal{I}_1$, $C' \vDash D \Rightarrow \Phi_1$ iff $\mathcal{I}_1$, $C' \wedge D \vDash \Phi_1$, which is exactly (1).
5. $\mathcal{I}_2$, $C' \wedge \exists \beta \bar{\alpha}.D \vDash \forall \beta.(\exists \bar{\alpha}.D) \Rightarrow \Phi_1$ follows from (5), $\mathcal{I}_2$, $C' \wedge \exists \beta \bar{\alpha}.D \vDash \exists \beta.\exists \bar{\alpha}.D$, and $\mathcal{I}_2, C \vDash \Phi_2$ is exactly (2).

6. From (4), (5) and the premise, $\mathcal{I}_1\mathcal{I}_2, C \vDash (\forall\beta.(\exists\bar{\alpha}.D) \Rightarrow \Phi_1) \wedge (\exists\beta.\exists\bar{\alpha}.D) \wedge \Phi_2$.
7. Let $\mathcal{I} = \mathcal{I}_1\mathcal{I}_2$; $\chi := \exists\bar{\alpha}.D[\beta := \delta]$, where $\chi\#\mathrm{PV}(\Gamma, \Phi_1, \Phi_2)$. (6) gives $\mathcal{I}, C \vDash (\forall\beta.\chi(\beta) \Rightarrow \Phi_1) \wedge (\exists\beta.\chi(\beta)) \wedge \Phi_2$, which is $\mathcal{I}, C \vDash [\![\Gamma \vdash \mathbf{letrec}\, x = e_1\, \mathbf{in}\, e_2 \colon \tau]\!]$.

– Case `Clause`.

1. `Clause`'s premises are: $C \vdash p\colon \tau_1 \longrightarrow \exists\bar{\beta}\,[D]\Gamma'$,
2. $C \wedge D, \Gamma\Gamma' \vdash e\colon \tau_2$,
3. and $\bar{\beta}\,\#\mathrm{FV}(C, \Gamma, \tau_2)$.
4. Assume w.l.o.g. that $\bar{\beta}\,\#\mathrm{FV}(\tau_1)$.
5. Let $[\![\vdash p\uparrow\tau_1]\!] = \exists\bar{\beta}'[D']\Gamma''$, where $\bar{\beta}'\#\mathrm{FV}(\Gamma, C, \tau_1, \tau_2, \bar{\beta}\,)$.
6. By lemma 12, (1) and (5) gives $C \vDash [\![\vdash p\downarrow\tau_1]\!]$
7. and $C \vDash \forall\bar{\beta}'.D' \Rightarrow \exists\bar{\beta}\,.D \wedge \Gamma''\doteq\Gamma'$, which is equivalent to $C \wedge D' \vDash \exists\bar{\beta}\,.D \wedge \Gamma''\doteq\Gamma'$.
8. By lemma 13, (2) implies $C \wedge D \wedge \Gamma''\doteq\Gamma', \Gamma\Gamma'' \vdash e\colon \tau_2$.
9. By (3) and `Hide`, (8) implies $C \wedge \exists\bar{\beta}\,.D \wedge \Gamma''\doteq\Gamma', \Gamma\Gamma'' \vdash e\colon \tau_2$.
10. (7) and (9) imply $C \wedge D', \Gamma\Gamma'' \vdash e\colon \tau_2$.
11. Which by induction hypothesis implies $\mathcal{I}, C \wedge D' \vDash \Phi_1$ for $\Phi_1 = [\![\Gamma\Gamma'' \vdash e\colon \tau_2]\!]$.
12. (6) and (11) give $\mathcal{I}, C \vDash [\![\vdash p\downarrow\tau_1]\!] \wedge \forall\bar{\beta}'.D' \Rightarrow \Phi_1$.

– Case `Equ`.

1. `Equ`'s premises are $C, \Gamma \vdash \mathrm{ce}\colon \tau'$, which by induction hypothesis gives $\mathcal{I}, C \vDash [\![\Gamma \vdash e\colon \tau']\!]$,
2. and $C \vDash \tau'\doteq\tau$.
3. Let $\Phi_\tau := [\![\Gamma \vdash e\colon \tau]\!]$. Observe, that τ occurs in Φ_τ only as a subterm in a side of equation: $\doteq\tau$, $\doteq\ldots \to \tau$, $\doteq(\ldots \to (\ldots \to \tau)\ldots)$. Therefore, $\tau'\doteq\tau \vDash \Phi_{\tau'} \Leftrightarrow \Phi_\tau$.
4. (1), (2) and (3) imply that $\mathcal{I}, C \vDash [\![\Gamma \vdash e\colon \tau]\!]$.

– Case `Hide`.

1. `Hide`'s premises are $C, \Gamma \vdash e\colon \tau$, that by induction hypothesis gives $\mathcal{I}, C \vDash [\![\Gamma \vdash e\colon \tau]\!]$,
2. and $\bar{\beta}\,\#\mathrm{FV}(\Gamma, \tau)$.
3. By (2), w.l.o.g. $\bar{\beta}\,\#\mathrm{FV}([\![\Gamma \vdash e\colon \tau]\!])$.
4. (1) implies that $\mathcal{I} \vDash \forall\bar{\beta}\,.(C \Rightarrow \Phi_1)$ which by (3) is equivalent to $\mathcal{I}, \exists\bar{\beta}\,.C \vDash [\![\Gamma \vdash e\colon \tau]\!]$.

- Case `FElim`. $\mathcal{I}, \boldsymbol{F} \vDash \Phi$ holds for any Φ.
- Case `DisjElim`.
  1. `DisjElim` premises are $C, \Gamma \vdash e\colon \tau$ and $D, \Gamma \vdash e\colon \tau$. Induction hypothesis gives $\mathcal{I}_1, C \vDash [\![\Gamma \vdash \mathrm{ce}\colon \tau]\!]$ and $\mathcal{I}_2, D \vDash [\![\Gamma \vdash \mathrm{ce}\colon \tau]\!]$ for some interpretations of predicate variables $\mathcal{I}_1, \mathcal{I}_2$.
  2. Therefore, we have $\mathcal{I}, C \vee D \vDash [\![\Gamma \vdash \mathrm{ce}\colon \tau]\!]$, for both $\mathcal{I} = \mathcal{I}_1$ and $\mathcal{I} = \mathcal{I}_2$. □

Proof of corollary 3.

Proof. $C, \Gamma \vdash \mathrm{ce}\colon \forall \bar{\alpha}[D].\tau$ can only be derived by the `Gen` rule, therefore we have $C' \wedge D, \Gamma \vdash e\colon \tau$ for $\bar{\alpha} \# \mathrm{FV}(\Gamma, C')$ and $C = C' \wedge \exists \bar{\alpha}.D$. By theorem 2, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I}, C' \wedge D \vDash [\![\Gamma \vdash e\colon \tau]\!]$. $\mathcal{I}, C' \wedge D \vDash [\![\Gamma \vdash e\colon \tau]\!]$ iff $\mathcal{I} \vDash C' \wedge D \Rightarrow [\![\Gamma \vdash e\colon \tau]\!]$ iff $\mathcal{I} \vDash \forall \bar{\alpha}.C' \wedge D \Rightarrow [\![\Gamma \vdash e\colon \tau]\!]$ iff $\mathcal{I}, C' \vDash \forall \bar{\alpha}.D \Rightarrow [\![\Gamma \vdash e\colon \tau]\!]$. Therefore $\mathcal{I}, C \vDash \forall \bar{\alpha}.D \Rightarrow [\![\Gamma \vdash e\colon \tau]\!]$. □

5.2 Existential Types

Proof of theorem 5.

Proof. By inspecting table 5, note that $\lambda[K]e$ subexpressions are absent from $n(e)$. Thus $\mathcal{I}_e$ is empty in all cases other than `ExIntro`. We therefore shorten these cases by not mentioning $\mathcal{I}_e$ and Σ. Below we extend the inductive proofs with the cases for expressions introduced by, or rule applications of, `ExIntro`, `LetIn` and `ExLetIn`.

- Theorem 1 (Correctness) $[\![\Gamma, \Sigma_0 \vdash \mathrm{ce}\colon \tau]\!], \Gamma, \Sigma_0 \vdash \mathrm{ce}\colon \tau$. Case: $\mathcal{E}(\mathrm{ce}) \neq \varnothing$.
  1. Induction hypothesis states $[\![\Gamma, \Sigma \vdash n(e)\colon \tau]\!], \Gamma, \Sigma \vdash n(e)\colon \tau$.
  2. The goal follows by `ExIntro`.
- Theorem 1 (Correctness) Case: ce is $\mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2$.
  1. Induction hypothesis yields $[\![\Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau]\!], \Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau$, $[\![\Gamma \vdash p.e_2\colon \alpha_0 \to \tau]\!], \Gamma \vdash p.e_2\colon \alpha_0 \to \tau$ and $[\![\Gamma \vdash e_1\colon \alpha_0]\!], \Gamma \vdash e_1\colon \alpha_0$.
  2. By weakening, (1), `Abs` and `App`, we get $[\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2\colon \alpha_0 \to \tau]\!] \wedge \not{E}(\alpha_0), \Gamma \vdash \lambda(p.e_2)e_1\colon \tau$.
  3. By `ExLetIn` we get $[\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge [\![\Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau]\!], \Gamma \vdash \mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2\colon \tau$, and by `LetIn`: $[\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2\colon \alpha_0 \to \tau]\!] \wedge \not{E}(\alpha_0)$, $\Gamma \vdash \mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2\colon \tau$.
  4. By (3) and `DisjElim` we get $([\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2\colon \alpha_0 \to \tau]\!] \wedge \not{E}(\alpha_0)) \vee_{\mathcal{E}} ([\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge [\![\Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau]\!]), \Gamma \vdash \mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2\colon \tau$ for $\mathcal{E} = \{K | K :: \forall \overline{\alpha_K} \bar{\beta}[E].\tau \to \varepsilon_K(\overline{\alpha_K})\}$.
  5. By (4), weakening and `Hide`, we get the goal.

– Theorem 1 (Correctness) Case: ce is $e_1\, e_2$.

1. Let $\alpha \# \mathrm{FV}(\Gamma, \tau)$.
2. By the induction hypothesis, we have $[\![\Gamma \vdash e_1 \colon \alpha \to \tau]\!], \Gamma \vdash e_1 \colon \alpha \to \tau$ and $[\![\Gamma \vdash e_2 \colon \alpha]\!], \Gamma \vdash e_2 \colon \alpha$.
3. By weakening and `App`, this yields $[\![\Gamma \vdash e_1 \colon \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 \colon \alpha]\!] \wedge \not{E}(\alpha), \Gamma \vdash e_1\, e_2 \colon \tau$.
4. By `Hide` using (1), $[\![\Gamma \vdash e_1\, e_2 \colon \tau]\!], \Gamma \vdash e_1\, e_2 \colon \tau$.

– Theorem 2 (Completeness) Case `ExIntro`: premise $C, \Gamma, \Sigma' \vdash n(e) \colon \tau$ for $\mathrm{Dom}(\Sigma') \backslash \mathrm{Dom}(\Sigma) = \mathcal{E}(e)$.

1. By induction hypothesis we have $\mathcal{I}_u, C \vDash [\![\Gamma, \Sigma' \vdash n(e) \colon \tau]\!]$.
2. Let $\Sigma_1 = \Sigma\overline{K :: \forall \alpha_K \gamma_K [\chi_K(\gamma_K, \alpha_K)].\gamma_K \to \varepsilon_K(\alpha_K)}$. The goal is $\mathcal{I}_u$, $C \vDash \mathcal{I}_e([\![\Gamma, \Sigma_1 \vdash n(e) \colon \tau]\!])[\overline{\varepsilon_K(\vec{\tau})} := \overline{\varepsilon_K(\bar{\tau})}]$.
3. The goal follows by setting $\mathcal{I}_e = \Sigma' / \Sigma$.

– Theorem 2 (Completeness) Case `App`.

1. `App`'s premises are $C, \Gamma \vdash e_1 \colon \tau' \to \tau$, $C, \Gamma \vdash e_2 \colon \tau'$ and $C \vDash \not{E}(\tau')$.
2. Pick w.l.o.g. $\alpha \notin \mathrm{FV}(C, \tau', \Gamma, \tau)$. (1) implies $C \wedge \alpha \doteq \tau' \vDash \not{E}(\alpha)$. By rule `Equ`, (1) implies $C \wedge \alpha \doteq \tau', \Gamma \vdash e_1 \colon \alpha \to \tau$ and $C \wedge \alpha \doteq \tau', \Gamma \vdash e_2 \colon \alpha$.
3. By induction hypothesis, (2) implies $\mathcal{I}_i$, $C \wedge \alpha \doteq \tau' \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash e_i \colon \tau_i]\!]$, $i = 1, 2, \tau_1 := \tau' \to \tau, \tau_2 := \tau'$.
4. By (2) and nonemptiness of sorts, we have $C \vDash \exists \alpha.(C \wedge \alpha \doteq \tau')$.
5. By (2), (3), and because $C \vDash D$ implies $\exists \alpha.C \vDash \exists \alpha.D$, we have $\mathcal{I}_1 \mathcal{I}_2$, $\exists \alpha.(C \wedge \alpha \doteq \tau') \vDash \exists \alpha.(\Phi_1 \wedge \Phi_2 \wedge \not{E}(\alpha))$.
6. By (4) and (5), we have the goal $\mathcal{I}, C \vDash [\![\Gamma \vdash e_1\, e_2 \colon \tau]\!]$ with $\mathcal{I} = \mathcal{I}_1 \mathcal{I}_2$.

– Theorem 2 (Completeness) Case `LetIn`: premise $C, \Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 \colon \tau$.

1. `LetIn`'s premise is: $C, \Gamma \vdash \lambda(p.e_2)\, e_1 \colon \tau$,
2. derived by `App` and `Abs` from $C, \Gamma \vdash p.e_2 \colon \tau' \to \tau$, $C, \Gamma \vdash e_1 \colon \tau'$ and $C \vDash \not{E}(\tau')$.
3. Inductive hypothesis gives $\mathcal{I}_1$, $C \vDash [\![\Gamma \vdash p.e_2 \colon \tau' \to \tau]\!]$ and $\mathcal{I}_2$, $C \vDash [\![\Gamma \vdash e_1 \colon \tau']\!]$.
4. (1) and (3) imply $\mathcal{I}_1, C \vDash [\![\Gamma \vdash p.e_2 \colon \tau' \to \tau]\!] \wedge \not{E}(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash K p.e_2 \colon \alpha_0 \to \tau]\!]$ as the first disjunct holds.
5. As the premise $\mathrm{PV}(C, \Gamma) = \varnothing$ gives disjoint domains for the $\mathcal{I}_i$, we have $\mathcal{I}, C \vDash [\![\Gamma \vdash e_1 \colon \tau']\!] \wedge ([\![\Gamma \vdash p.e_2 \colon \tau' \to \tau]\!] \wedge \not{E}(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash K p.e_2 \colon \tau' \to \tau]\!])$ for $\mathcal{I} = \mathcal{I}_1 \mathcal{I}_2$.
6. $\mathcal{I}, C \vDash \exists \alpha_0.[\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge ([\![\Gamma \vdash p.e_2 \colon \alpha_0 \to \tau]\!] \wedge \not{E}(\alpha_0) \vee_{\mathcal{E}} [\![\Gamma \vdash K p.e_2 \colon \alpha_0 \to \tau]\!])$ by abstracting $\alpha_0 = \tau'$.

– Theorem 2 (Completeness) Case `ExLetIn`:
premise $C, \Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 \colon \tau$.

1. `ExLetIn`'s premises are: $C, \Gamma \vdash K p.e_2 \colon \tau' \to \tau$ and $C, \Gamma \vdash e_1 \colon \tau'$,
2. Inductive hypothesis gives $\mathcal{I}_1$, $C \vDash [\![\Gamma \vdash K p.e_2 \colon \tau' \to \tau]\!]$ and $\mathcal{I}_2$, $C \vDash [\![\Gamma \vdash e_1 \colon \tau']\!]$.
3. (3) implies $\mathcal{I}_1, C \vDash [\![\Gamma \vdash p.e_2 \colon \tau' \to \tau]\!] \wedge \not{E}(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash K p.e_2 \colon \tau' \to \tau]\!]$ as one of the $\vee_{\mathcal{E}}$ disjuncts holds. The proof concludes as in the `LetIn` case. □