GADTs for Reconstruction of Invariants and Postconditions

Łukasz Stafiniak

University of Wrocław
Department of Mathematics and Computer Science
Institute of Computer Science

Wrocław 2015

PhD Thesis
Doctoral advisor: prof. dr hab. Leszek Pacholski

Typy GADT dla rekonstrukcji niezmienników i warunków końcowych

Łukasz Stafiniak

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Wrocław 2015

Rozprawa doktorska napisana pod kierunkiem prof. dra hab. Leszka Pacholskiego

Abstract

Type systems for programming languages are both a first line of defense against programmer mistakes, and an aid in structuring programs around data structures and functions that manipulate them. Type systems are a natural formalism for language extensions that express those aspects of program specifications which can be automatically, statically checked. Type inference enhances programmer efficiency and code adaptability by providing the types of expressions to the programmer, rather than asking the programmer for the types. There is a long line of research on automated program analysis and verification by generating specifications that a program does obey, for example generating loop invariants. This thesis extends a familiar type system for functional programming languages and provides a type inference algorithm that generates invariants and postconditions of recursive functions.

Generalized Algebraic Data Types (GADTs) extend polymorphic type systems by introducing reasoning by cases into type-checking of definitions by cases. We present a GADTs type system MMG(X) based on François 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 show that the type inference problem reduces to satisfaction of second order constraints.

We solve the constraints by iterating: (1) constraint generalization, which finds most specific common consequences; and (2) joint constraint abduction, which finds most general common explanations. Abduction is used to generate invariants, infer and check types. Generalization is used to generate existential types, which serve as postconditions. The constraints include linear arithmetic (equations and inequalities).

We called the system implementing these techniques InvarGenT. It solves a vast majority of inference tasks we attempted, without type annotations. InvarGenT solves nearly all meaningful GADTs inference tasks we tried, clearly more than all earlier type inference implementations for GADTs. It also solves clearly more invariant and postcondition inference tasks than the *Liquid Types* approach, except for some programs with higher-order functions. In addition, we present a selection of new inference tasks, for programs manipulating lists with length, binary numbers and AVL trees of imbalance 2. These programs can serve as baseline tests for future research on invariant and postcondition inference.

Streszczenie

Systemy typów dla języków programowania są zarówno pierwszą linią obrony przed błędami programistycznymi, jak i pomocą przy strukturowaniu programów wokół struktur danych oraz funkcji które nimi manipulują. Systemy typów to dobry formalizm do wyrażania tych aspektów specyfikacji programów, które mogą być automatycznie, statycznie sprawdzone. Inferencja typów zwiększa wydajność programisty i adaptowalność kodu dostarczając programiście typy wyrażeń, zamiast wymagać podawania typów. Badania nad automatyczną analizą i weryfikacją programów od dawna obejmowały między innymi generowanie specyfikacji spełnianych przez dane programy, na przykład generowanie niezmienników pętli. Ta praca rozszerza znany system typów dla języków funkcyjnych i podaje algorytm inferencji typów, generujący niezmienniki i warunki końcowe funkcji rekurencyjnych.

Generalized Algebraic Data Types (GADTs) rozszerzają systemy typów polimorficznych o wnioskowanie przez przypadki podczas sprawdzania typu definicji przez przypadki. Prezentuję system typów MMG(X) z GADTs, oparty o system typów HMG(X) François Pottiera i Vincenta Simoneta ale bez annotacji typami. Rozszerzam go do języka z typami egzystencjalnymi reprezentowanymi jako domyślnie definiowane i używane struktury GADTs. Pokazuję redukcję problemu inferencji typów do spełnialności więzów drugiego rzędu.

Więzy drugiego rzędu rozwiązuję iterując dwa algorytmy: (1) generalizację więzów, znajdującą najbardziej specyficzną wspólną konsekwencję więzów; (2) łączną abdukcję więzów, znajdująca najogólniejsze wspólne objaśnienie, przesłankę implikującą więzy. Abdukcji używamy głównie do generowania niezmienników, inferencji i sprawdzania typów. Generalizacji używamy do generowania typów egzystencjalnych, służących jako warunki końcowe. Więzy obejmują arytmetykę liniową (równania i nierówności).

System implementujący te techniki nazwałem InvarGenT. Rozwiązuje on zdecydowaną większość zadań inferencji które opracowałem lub zaadaptowałem, bez annotacji typami. InvarGenT rozwiązuje zdecydowanie więcej zadań inferencji typów dla GADTs niż dotychczasowe systemy. Rozwiązuje też więcej problemów inferencji niezmienników i warunków końcowych niż podejście *Liquid Types*, jeśli ograniczymy się do zadań nie potrzebujących inferencji niezmienników dla argumentów funkcji wyższego rzędu. Dodatkowo, prezentuję kilka programów operujących na listach z długością, liczbach binarnych i drzewach AVL o niezbalansowaniu nie przekraczającym 2. Te programy mogą służyć jako część testów dla przyszłych prac nad wszechstronnymi systemami inferencji niezmienników i warunków końcowych.

Acknowledgements

I thank my advisor Leszek Pacholski for supporting the great opportunity to pursue this work despite my excursions to other topics and projects. I thank Łukasz Kaiser for feedback and support through the years. My office colleague Jan Otop offered discussions early in my work. Dariusz Biernacki reviewed and helped improve my conference paper. Filip Sieczkowski, Jakub Michaliszyn and Marek Materzok remarks helped improve the thesis presentation. All the technical errors are my sole responsibility. Most of all, I dedicate this thesis to my parents Maria and Piotr, without whose support this work would not have been possible.

Every problem that is interesting is also soluble.
David Deutsch's Law

Table of contents

Chapter 1

Introduction

Type systems are established natural deduction-style means to specify programs. Dependent types can represent arbitrarily complex properties as they use the same language for both types and programs. The type of the value returned by a function can itself be a function of the argument. Generalized Algebraic Data Types (GADTs) bring some of that expressiveness to type systems with data-types and parametric polymorphism, by introducing the ability to reason about the return type by case analysis on the input value. Work over the past decade (Pottier and Régis-Gianas [36], Schrijvers, Peyton Jones, Sulzmann and Vytiniotis [45], Lin and Sheard [23]) shows that type systems with GADTs can remain tractable from a compiler implementer's perspective if appropriately constrained, i.e. have principal typings and efficient type inference. Our work is based on Simonet and Pottier [47] instead, which is the most general presentation of GADTs. Our methods are computationally intensive and involve search with backtracking. We expect to be able to infer types for more programs than all of the efficiency-oriented approaches. Our work could bear the title *Type Inference for GADTs and Existentials*, but we stress that we do not compete with the work of [36], [45] in particular. We are concerned with type inference for recursive definitions which are not given their type beforehand. This polymorphic recursion problem has been tackled by [23] with GADTs, by Schrijvers and Bruynooghe [44] without GADTs. The line of work following Unno and Kobayashi [51] also tackles reconstruction of types for recursive definitions, but without ADTs and without polymorphic recursion. Our intended usage is that the generated types of recursive definitions be integrated into the source code, via integration with an IDE.

Existential types hide some information conveyed in a type. We may opt to use them to expose a more abstract interface. However, sometimes we are forced to use existential types to hide what cannot be expressed in the type system. GADTs provide existential types by using local type variables for the hidden parts of the type encapsulated in a GADT. With no other way to express existential types, programmers need to introduce by hand spurious data-types for them. For example, if we want to hide the length of a list in OCaml, we need to define `type _ elist = List : ('a, 'b) llist -> 'a elist`, where `('a, 'b) llist` is the type of lists of length `'b` with elements of type `'a`. The type system in Xi and Pfenning [57] for a language called Dependent ML, a precursor for GADTs, has a more light-weight approach: explicit existential types. But [57] has limited type inference, and we find its type system harder to grasp than the more recent presentations of GADTs. Knowles and Flanagan [20] and Unno and Kobayashi [51] can be seen as performing type inference for forms of existential types. We provide full reconstruction of existential types.

The feedback provided by type inference makes strongly typed programming more convenient in several ways. It softens the learning curve by enabling learning of types by experimentation. It facilitates rapid prototyping and code refactoring by reporting the types of functions when the programmer experiments with invariants of data-types. Besides providing certain correctness guarantees, types also serve as documentation – it is good to keep them around. If type inference results are incorporated in the source code, type checking might suffice during slow evolution of a code base, while the rich types document the code.

Our type system for GADTs differs from others in that we do not require any type annotations on expressions, even on recursive functions. Inference in our implementation sometimes requires guidance by `assert` clauses. Our implementation InvarGenT includes linear arithmetic in the language used to express invariants and postconditions, with the possibility to introduce more domains in the future. The arithmetic properties are conjunctions of equations and inequalities, where a side can be either a linear combination, or a maximum or minimum of (at most two) linear combinations.

Example 1.1. We can easily specify the data-type of AVL trees with height imbalance of at most 2:

```
datatype Avl : type * num
datacons Empty : ∀a. Avl (a, 0)
datacons Node : ∀a,k,m,n
  [k=max(m,n) ∧ 0⩽m ∧ 0⩽n ∧ n⩽m+2 ∧ m⩽n+2].
     Avl (a, m) * a * Avl (a, n) * Num (k+1)
     ⟶ Avl (a, k+1)
```

A similar definition can be given in the DML and ATS languages by Hongwei Xi (see [57]). Given this type definition and AVL tree algorithms, InvarGenT automatically finds out that the height of a tree with added element can be the same as original tree or bigger by 1, and that removing an element can decrease the height of a tree by at most 1, returning, among others, the types:

```
add :
  ∀a,n.a→Avl(a, n)→∃k[k⩽n+1 ∧ 1⩽k ∧ n⩽k].Avl(a, k)
remove :
  ∀a,n.a→Avl(a, n)→∃k[n⩽k+1 ∧ 0⩽k ∧ k⩽n].Avl(a, k)
```

There are no places requiring type annotations or informative assertions in the source code of AVL tree algorithms including the above functions and their helper functions.

We reduce the type inference task to *second order* constraint satisfaction, which in addition to finding substitutions for first order variables, finds solutions to the predicate unknowns. As the predicate-finding techniques that we present in Chapter 4 (especially abduction) improve, type inference will work for more programs. The downside is that we do not specify declaratively the limitations of type inference. That is, while we provide some guarantees that our approach to inference does not overlook solutions (completeness-like result at the end of Section 4.5), there is no concise formulation of when type inference succeeds.

Constraints appear in three contexts in programs and inferred signatures:

- In specifications of value constructors (symbolically $K :: \forall \bar{\beta}[D].\tau_1 \times ... \times \tau_n \longrightarrow \varepsilon(\bar{\tau})$). We call the constraint (i.e. D) the invariant (of K).
- In type schemes, which usually are signatures of values (symbolically $x{:}\forall \bar{a}[D].\tau$). We call the constraint (i.e. D) the invariant or the precondition (of x), interchangeably.
- In existential types (symbolically $\exists \bar{\alpha}[D].\tau$); existential types usually occur in result positions of function types in type schemes. We call the constraint (i.e. D) the post-condition (of the corresponding function or computation).

1.1. Contributions

In order to implement InvarGenT, we needed to resolve several issues, leading to our contributions:

- Design a type system that captures invariants (Section 3.2) and postconditions (Section 3.4).
- Reduce type inference to satisfaction of second order constraints (Sections 3.3 and 3.5).
- Employ invariant-finding abduction and postcondition-finding generalization in an algorithm that reconstructs correct types (Section 4.1).
- Develop constraint abduction algorithms that handle universally quantified variables (Section 4.2).
- Develop constraint generalization algorithms. Constraint generalization computes anti-unification in case of free terms and extended convex hull in case of linear inequalities (Section 4.3).

To summarize, the novelty of our work lies in:

1. performing type inference for GADTs with polymorphic recursion for more programs than in prior work ([23]),
2. introducing existentials into the GADT type system with minimal added complexity, by using GADT encodings, and performing type inference for them,
3. implementing the type inference over a constraint domain with linear arithmetics, thus performing numerical precondition and postcondition generation for recursive functions (in a different manner than in Rondon, Kawaguchi and Jhala [41]).

We discuss related work in Chapter 2 and Section 6.1. There remains work to be done, as we discuss in Section 5.2 and Section 6.2.

InvarGenT can be found at https://github.com/lukstafi/invargent. It solves a vast majority of inference tasks we attempted, without type annotations. InvarGenT solves all but 3 tasks from Lin [22] (2 unsolved are practical, one of them is solved after a slight meaning-preserving modification); the system from [22] does not solve 8 of those tasks (7 unsolved are practical). We translated into InvarGenT all but 3 tasks from Rondon, Kawaguchi and Jhala [41] – all tasks that [41] uses for comparison with Xi and Pfenning's DML system [57], [56]. InvarGenT solves all but 2 of these inference tasks without any type annotation, and solves the remaining 2 tasks after a slight meaning-preserving change to the programs (still without any type annotation). The system DSolve from [41] implementing the *Liquid Types* approach needs a type annotation for 3 of these inference tasks. The inference times of InvarGenT and DSolve are of the same order. Overall, the InvarGenT approach solves clearly more inference tasks than the *Liquid Types* approach, when the programs do not have higher-order functions that require universally quantified invariants for arguments, including existentially quantified invariants for arguments of arguments. In addition, we present a selection of new inference tasks, for programs manipulating lists with length, binary numbers and AVL trees of imbalance 2. They can serve as a baseline for future research. We expect that some of them are beyond the capabilities of any current type inference system, except InvarGenT. DSolve does not introduce new linear combinations into generated constraints, therefore will not solve some of these inference tasks. The recent system SpecLearn, see He Zhu, Aditya Nori and Suresh Jagannathan [60], cannot solve tasks like the AVL trees example, without any type annotations or assertions.

1.2. Examples

To motivate the language constructs and type system rules, we start with some examples. 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 "proper" types. `i`,`j`,`k`,`l`,`m`,`n`,`i1`,... are in the sort of linear arithmetic over rational numbers. Type constructors and value constructors have the same syntax: capitalized name followed by a tuple of arguments. They are introduced by the keywords `datatype` and `datacons` respectively. The result sort of `datatype` is always `type` and is omitted. By *toplevel* of a source file we mean the environment of names available for potential code appended to the end of the file. Values assumed into the toplevel without InvarGenT definition are introduced by the keyword `external`.

We can introduce existential types directly in type declarations. To have an existential type inferred, we have to use `efunction`, `ematch` or `eif` expressions, which differ from `function`, `match` and `if` correspondingly only in that the (return) type is an existential type. To use, i.e. unpack, a value of an existential type, we have to bind it with a `let..in` expression. An existential type will be automatically unpacked before being "repackaged" as another existential type. In a future version, it might be possible to use existential types without explicit introduction (the `efunction`, `ematch` or `eif` syntax) and explicit elimination (the need of `let..in` expressions) at a cost of longer inference times.

```
datatype Z
datatype I : type
datatype O : type
datatype Binary : type
datacons BinaryZ : Binary Z
datacons BinaryO : ∀a. Binary a ⟶ Binary (O a)
datacons BinaryI : ∀a. Binary a ⟶ Binary (I a)

let rec erase_zeros = efunction
  | BinaryZ -> BinaryI BinaryZ
  | BinaryO x -> erase_zeros x
  | BinaryI x -> let y = erase_zeros x in BinaryI y
```

Table 1.1. Use of existential types

Example 1.2. Table 1.1 defines an artificial example using phantom types (i.e. types used for type information, without values), resembling binary numbers. The function `erase_zeros` erases the "zeros" `BinaryO` and adds one `BinaryI`.

We get `erase_zeros:∀a.Binary a→∃a.Binary (I a)`. The resulting type forgets the shape of the content, other than the fact that it starts with a `BinaryI`. The first branch of `erase_zeros` returns a value directly. The second branch directly performs the recursive call – the existential type of the result is unpacked and "repackaged". The third branch unpacks the result of the recusrive call explicitly. By design, the inlined variant `| BinaryI x -> BinaryI (erase_zeros x)` would not type-check. The type system does not allow passing values of existential types as arguments.

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 three means to enrich the generated constraints: `assert false` indicates an unreachable code location and provides a negative constraint, `assert num` e_1`<=`e_2 and `assert type` e_1`=`e_2 provide positive constraints, and the `test` syntax includes constraints of the use cases appearing after `test` with the constraint of a toplevel definition.

Example 1.3. Table 1.2 defines a function `equal` comparing values provided representation of their types. The value constructors `TInt`, `TPair`, `TList` are used to represent the types of values compared. The function `equal` checks whether the second and third argument are of the same type, and if so, whether they are equal.

We get `equal: ∀a,b.(TypeRepr a, TypeRepr b)→a→b→Bool`. To ensure only one maximally general type for `equal`, it is sufficient to provide either the two `assert false` clauses, or the `test` clause; both are illustrated above. The first assertion excludes independence of the first encoded type and the second argument. The second assertion excludes independence of the second encoded type and the third argument. The test ensures that arguments of distinct types can be given.

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

```
datatype List : type
datacons Nil : ∀a. List a
datacons Cons : ∀a. a * List a ⟶ List a
datatype TypeRepr : type
datacons TInt : TypeRepr Int
datacons TPair : ∀a, b. TypeRepr a * TypeRepr b ⟶ TypeRepr (a, b)
datacons TList : ∀a. TypeRepr a ⟶ TypeRepr (List a)
external let eq_int : Int → Int → Bool = "(=)"
external let b_and : Bool → Bool → Bool = "fun a b -> a && b"
external let b_not : Bool → Bool = "fun b -> not b"
external let forall2 :
  ∀a, b. (a→b→Bool) → List a → List b → Bool =
  "fun f a b -> List.for_all2 f a b"
external let zero : Int = "0"

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

Table 1.2. Two ways of constraining types

Chapter 2

Background and Related Work

In this chapter, we provide background knowledge and context by taking an in-depth look at a selection of related work. Sections 2.2 and 2.6 constitute a proper background for later chapters. The remaining sections describe alternative approaches to type inferece for GADTs, and more broadly to the task of automatic generation and verification of program specifications for functional programming languages like OCaml. Readers familiar with the works cited below may skip directly to the subsections *Relevance*, where we contrast the corresponding work with InvarGenT. The final chapter's Section 6.1 provides a broader perspective on related work.

The theoretical underpinnings of type systems for program specification trace back to the work on dependent types by Per Martin-Löf, for example [30]. An early example of a practical programming language based on dependent types is Cayenne by Lennart Augustsson, [2]. A currently popular example of such language is Idris by Edwin Brady [5], see also Brady, Herrmann and Hammond [6]. But the family of approaches our thesis belongs to, maintains the separation of types and values characteristic of languages like Pascal, C, OCaml and Haskell. This separation allows to employ domain-specific decision procedures to reason about types, and also to automatically infer types.

We start with Hongwei Xi and Frank Pfenning [57], which not only is a precursor of GADTs research, but also stresses the importance of existential types and type inference. We present the HMG(X) system from Simonet and Pottier [47], to motivate and gain familiarity with the formalism of our constraint-based type system. We then present Schrijvers, Peyton Jones, Sulzmann and Vytiniotis [53], and Lin and Sheard [23], to illustrate one thread of approaches to type inference for type systems with GADTs. Then, we switch gears to discuss inference of invariants, known as *liquid types* from Rondon, Kawaguchi and Jhala [41], expanded in [42] and [40]. Finally, we describe Maher and Huang [29], which provides the basis for our constraint abduction algorithm for terms.

We take liberties with the presentation of the systems described, except HMG(X) where we stay close to the letter of [47]. We apologize for any unfortunate resulting errors and misunderstandings, and encourage interested readers to consult the original publications.

2.1. The DML System

The work of Hongwei Xi and Frank Pfenning [57], see also [54], was at the forefront of research introducing type system features modeled on dependent types into practical programming languages of the ML family. These features mediate the dependence of types on terms by

the use of strongly constrained types, including singleton types, i.e. types inhabited by single terms. They were later simplified and further developed to fit within the polymorphic (non-dependent) type systems. Together with an independent work at the time by Christoph Zenger [58], and an earlier work by Konstantin Läufer and Martin Odersky [24], these efforts are the origins of Generalized Algebraic Data Types.

Hongwei Xi's DML(X) is parameterized by a constraint domain X, and, like InvarGenT, the implementation includes linear arithmetics. We will discuss a small selection of type system rules and then the approach to type inference taken by the DML system. The reader should not expect to gain understanding of DML from this short exposition, in part because it is a complex system.

2.1.1. The Type System

Type judgments in DML(X) have naturally a different form for patterns than for expressions. For pattern p, type τ, formula φ, and environment assigning expression variables to types Γ, we write $p{\downarrow}\tau \rhd (\varphi; \Gamma)$ to mean that pattern p, when matched against an expression of type τ, allows us to type-check its corresponding branch in the context of an environment enriched by Γ and under a constraint enriched by φ. In our notation: $p{\downarrow}\tau \longrightarrow \exists \bar{\alpha}[\varphi]\Gamma$. The fresh type variables $\bar{\alpha}$ are in fact introduced inside φ in DML(X). For expression e, type τ, formula φ, type environment Δ assigning sorts to type variables, and environment Γ assigning types to expression variables, we write $\varphi; \Delta; \Gamma \vdash e\colon \tau$ to mean that expression e has type τ in the context of the environments Δ, Γ and the constraint formula φ. In our notation: $\varphi; \Gamma \vdash e\colon \tau$, because we discriminate the sorts of type variables by the names of the variables.

One of the most interesting rules of DML(X) connects these two forms of judgments:

$$\frac{p{\downarrow}\tau_1 \rhd (\varphi'; \Gamma') \qquad \varphi \wedge \varphi'; \Delta; \Gamma\Gamma' \vdash e\colon \tau}{\varphi; \Delta; \Gamma \vdash p \Rightarrow e\colon \tau_1 \Rightarrow \tau_2}$$

The notation $\tau_1 \Rightarrow \tau_2$ is used instead of a function type $\tau_1 \to \tau_2$ to indicate an "internal" use of a pattern matching branch: it is always a part of a pattern matching syntactic construct. The premise $\varphi \wedge \varphi'; \Delta; \Gamma\Gamma' \vdash e\colon \tau$ means that for type-checking the body of a pattern matching branch, we have available not only the pattern variables from Γ', but also properties φ' of their types.

Another interesting aspect of DML(X) is the presence of both universal and existential types. Universal types are as in system F. Rules for existential types:

$$\frac{\varphi; \Delta; \Gamma \vdash e\colon \tau[\alpha := i] \qquad \varphi \vdash i\colon \gamma}{\varphi; \Delta; \Gamma \vdash \langle i \,|\, e\rangle\colon \exists \alpha\colon \gamma.\tau} \text{ introduction}$$

$$\frac{\varphi; \Delta; \Gamma \vdash e_1\colon \exists \alpha\colon \gamma.\tau_1 \qquad \varphi \wedge \alpha\colon \gamma; \Gamma\{x\colon \tau_1\} \vdash e_2\colon \tau_2}{\varphi; \Delta; \Gamma \vdash \mathbf{let}\ \langle \alpha \,|\, x\rangle = e_1\ \mathbf{in}\ e_2\colon \tau_2} \quad \text{elimination}$$

where γ is a sort in the multisorted logic of X. The $\langle i \,|\, e\rangle$ construct pairs an expression e with a witness i for the existential type $\exists \alpha\colon \gamma.\tau$. In fact, the concrete language of DML does not have this construct. Rather, it is introduced by type inference, as discussed below.

2.1.2. Bidirectional Type Inference

To facilitate type inference for the concrete language of DML, the system is equipped with two mutually recursive kinds of judgements called *elaboration judgments*: the *synthesizing* judgment $\varphi; \Gamma \vDash e{\uparrow}\tau \Rightarrow e^*$, and the *checking* judgment $\varphi; \Gamma \vDash e{\downarrow}\tau \Rightarrow e^*$. The synthesizing judgments introduce fresh type variables when needed, which are solved from the constraints φ. The types for lambda expressions and recursive definition expressions are not synthesized. To see the interaction between synthesizing and checking judgments, consider the rule for function application:

$$\frac{\varphi; \Gamma \vdash e_1{\uparrow}\tau_1 \to \tau_2 \Rightarrow e_1^* \qquad \varphi; \Gamma \vdash e_2{\downarrow}\tau_1 \Rightarrow e_2^*}{\varphi; \Gamma \vdash e_1\, e_2{\uparrow}\tau_2 \Rightarrow e_1^*\, e_2^*}$$

Since τ_1 is synthesized as part of the function type by $\varphi; \Gamma \vdash e_1{\uparrow}\tau_1 \to \tau_2 \Rightarrow e_1$, subsequently it only needs to be checked for the argument e_2. Synthesis for variables is done by referring to the environment, and for data constructors by referring to the respective datatype definition. Checking for variables and data constructors refers back to synthesis, and we employ constraint solving to decide whether the types agree:

$$\frac{\varphi; \Gamma \vdash x{\uparrow}\tau_1 \Rightarrow e^* \qquad \varphi \vDash \tau_1 \doteq \tau_2}{\varphi; \Gamma \vdash x{\downarrow}\tau_2 \Rightarrow e^*}$$

With existential types, the situation is further complicated by the need to introduce, during elaboration, "witnesses" i into expressions of existential type. DML relies on a judgment coerce, whose one of most interesting rules is:

$$\frac{\varphi \vdash \mathrm{coerce}(\tau_1, \tau[\alpha := i]) \Rightarrow E \qquad \varphi \vdash i : \gamma}{\varphi \vdash \mathrm{coerce}(\tau_1, \Sigma(\alpha : \gamma).\tau) \Rightarrow \langle i \,|E\rangle}$$

where we see existential witness introduced when an existential type is encountered. The coerce judgment is connected to synthesizing judgments via checking rules. Let us look at the other rule for function application:

$$\frac{\varphi; \Gamma \vdash e_1\, e_2{\uparrow}\tau_1 \Rightarrow e^* \qquad \varphi \vdash \mathrm{coerce}(\tau_1, \tau_2) \Rightarrow E}{\varphi; \Gamma \vdash e_1\, e_2{\downarrow}\tau_2 \Rightarrow E[e^*]}$$

where E is an expression context with a single hole.

2.1.3. Relevance

The work on DML(X) is foundational to InvarGenT. While we chose to build on the more elegant formalism of HMG(X), we share some of the goals of DML, diverging on one:

1. The formalism behind InvarGenT is parametrized by the domain of constraints, just as DML(X). The implementation handles linear arithmetic constraints, and can be extended to other domains, similarly to DML.
2. Both DML and InvarGenT place emphasis on the seamless use of existential types, aided by type inference. However, existential types are not synthesized by DML.
3. DML and InvarGenT share concern with covering a sufficient portion of ML-family language features, in particular pattern matching with deep patterns.

4. DML requires type annotations on all functions that cannot be typed in the Hindley-Milner type system. In exchange, DML(X) has the principal type property and type inference in DML is efficient. INVARGENT takes the opposite stance, not requiring any type annotations.

Unlike DML, INVARGENT does not currently support first-class universal types, i.e. function arguments which can be used polymorphically, with different types within a function. Introducing inferred universal types to INVARGENT is left as future work. The type inference for universal types of function arguments would work similarly to how type inference for recursive definitions (which admits polymorphic recursion) works currently.

DML performs A-transformation, which is a source level transformation introducing **let** bindings for subexpressions. In this way, all occurrences of expressions with existential types are unpacked, with the existential type eliminated. In interests of type inference times, INVARGENT does not perform A-transformation. However, passing expressions of an existential type as arguments to functions is prohibited in INVARGENT's type system, therefore A-transformation would be a conservative extension. This captures errors where the programmer forgets to unpack the existential type, without limiting expressivity.

In INVARGENT, introductions of existential types need to be marked in the source by adding a single character to the corresponding concrete syntax keywords: `function`, `match` and `if`. In DML, existential type introductions do not figure in the concrete syntax, but existential types need to be spelled out in type annotations. In INVARGENT, inference of existential types will find out how many existential parameters are needed, including the case when none are needed, i.e. the type is not in fact existential. Therefore, a more sophisticated approach might introduce existential types automatically, at cost of increased type inference times.

2.2. THE HMG(X) FORMALIZATION

In parallel to the development of DML, Martin Odersky, Martin Sulzmann and Martin Wehr in [34] developed a general framework HM(X) for expressing type systems extending the Hindley-Milner type system and parameterized by a constraint domain. Hongwei Xi and collaborators further developed the ideas behind DML into an extension of type systems for languages in the ML family – Standard ML, Haskell, OCaml – they originally called *Guarded Recursive Data Types*, see [55], later to become known as *Generalized Algebraic Data Types* or GADTs. GADTs were independently investigated at that time by James Cheney and Ralf Hinze as *first-class phantom types* in [8], and by Tim Sheard as *equality qualified types* in [46]. Vincent Simonet and François Pottier in [47] brought these two lines of research, HM(X) and GADTs, together, by designing an elegant type system HMG(X).

2.2.1. The Untyped Calculus

While the *indexed types* of Christoph Zenger [58] and dependent types of DML refine ML but do not make more ML programs typeable, GADTs extend ML, i.e. make some programs typeable – provided appropriate datatype definitions – that would not type-check otherwise.

$$
\begin{array}{l}
p := x \mid 0 \mid 1 \mid p \wedge p \mid p \vee p \mid K\,\bar{p} \\
c := p.e \\
e := x \mid K\,\bar{e} \mid \mathbf{let}\, x = e\, \mathbf{in}\, e \mid e\,e \mid \lambda(\bar{c}) \mid \mu x.v \\
v := K\,\bar{v} \mid \lambda(\bar{c})
\end{array}
$$

Table 2.1. Syntax of the calculus for HMG(X)

$$
\begin{array}{rcl}
[0 := v] & = & \text{undefined} \\
[1 := v] & = & \varnothing \\
[p_1 \wedge p_2 := v] & = & [p_1 := v] \otimes [p_2 := v] \\
[p_1 \vee p_2 := v] & = & [p_1 := v] \oplus [p_2 := v] \\
[K p_1 \cdots p_n := K v_1 \cdots v_n] & = & [p_1 := v_1] \otimes \cdots \otimes [p_n := v_n]
\end{array}
$$

Table 2.2. Extended substitution for HMG(X)

We present the call-by-value λ-calculus behind HMG(X) since we will defer the semantics of InvarGenT to HMG(X). The rest of this section follows closely [47], starting with pages 12-14.

Let x and K range over disjoint denumerable sets of variables and data constructors, respectively. For every data constructor K, we assume a fixed nonnegative arity. The syntax of patterns, expressions, clauses, and values is given in Figure 2.1. Patterns include the empty pattern 0, the wildcard pattern 1, variables, conjunction and disjunction patterns, and data constructor applications. Defined program variables $\mathrm{dpv}(p)$ are the unique variables in p. The pattern p is considered ill-formed for nonlinear patterns (i.e. patterns with repeating variables). Expressions include variables, functions, data constructor or function applications, recursive definitions, and local variable definitions. Functions are defined by cases: a λ-abstraction, written $\lambda(c_1, ..., c_n)$, consists of a sequence of clauses. A clause c is made up of a pattern p and an expression e and is written $p.e$; the variables in $\mathrm{dpv}(p)$ are bound within e. We occasionally use ce to stand for a clause or an expression. Values include functions and applications of a data constructor to values. Within patterns, expressions, and values, all applications of a data constructor must respect its arity: data constructors cannot be partially applied.

Whether a pattern p matches a value v is defined by an extended substitution $[p := v]$ that is either undefined, which means that p does not match v, or a mapping of $\mathrm{dpv}(p)$ to values, which means that p does match v and describes how its variables become bound. Of course, when p is a variable x, the extended substitution $[x := v]$ coincides with the ordinary substitution $[x := v]$, which justifies our abuse of notation. Extended substitution for other pattern forms is defined in Figure 2.2. Let us briefly review the definition. The pattern 0 matches no value, so $[0 := v]$ is always undefined. Conversely, the pattern 1 matches every value, but binds no variables, so $[1 := v]$ is the empty substitution. In the case of conjunction patterns, $\otimes$ stands for (disjoint) set-theoretic union, so that the bindings produced by $p_1 \wedge p_2$ are the union of those independently produced by p_1 and p_2. The operator $\otimes$ is strict—that is, its result is undefined if either of its operands is undefined—which means that a conjunction pattern matches a value if and only if both of its members do. In the case of disjunction

patterns, $\oplus$ stands for a nonstrict, angelic choice operator with left bias: when o_1 and o_2 are two possibly undefined mathematical objects that belong to the same space when defined, $o_1 \oplus o_2$ stands for o_1 if it is defined and for o_2 otherwise. As a result, a disjunction pattern matches a value if and only if either of its members does. The set of bindings thus produced is that produced by p_1, if defined, otherwise that produced by p_2. Last, the pattern $Kp_1 \cdots p_n$ matches values of the form $Kv_1 \cdots v_n$ only; it matches such a value if and only if p_i matches v_i for every $i \in \{1, ..., n\}$.

The call-by-value small-step semantics, written $\rightarrow$, is defined by the rules of Figure 2.3. It is standard. The first rule governs function application and pattern-matching: $\lambda(p_1.e_1 \cdots p_n.e_n)$ reduces to $e_i[p_i := v_i]$, where i is the least element of $\{1, ..., n\}$ such that p_i matches v. Note that this expression is stuck (does not reduce) when no such i exists. The last rule lifts reduction to arbitrary evaluation contexts.

2.2.2. The HMG(X) Type System

HMG(X) supports subtyping. The language of constraints is as follows:

$$\begin{array}{rcl} \tau & := & \alpha \mid \tau \rightarrow \tau \mid \varepsilon(\tau, ..., \tau) \\ \pi & := & \preccurlyeq \mid ... \\ C, D & := & \pi\, \bar{\tau} \mid C \wedge C \mid C \vee C \mid \exists \alpha.C \mid \forall \alpha.C \mid C \Rightarrow C \end{array}$$

where τ are types, π are constraint relations and $\preccurlyeq$ is a subtyping relation, C, D are constraint formulas. Environments Γ assign variables to type schemes, e.g. $\{x \mapsto \sigma\}$, where type schemes are triples:

$$\sigma := \forall \bar{\alpha}[C].\tau$$

indicating a value of type τ polymorphic wrt. variables $\bar{\alpha}$, constrained by formula C. A novel concept is that of an *environment fragment*, a triple:

$$\Delta := \exists \bar{\beta}[D]\Gamma$$

where Γ is a *simple environment* which assigns variables to types (not type schemes). Environment fragments are used to describe the static knowledge that is gained by successfully matching a value against a pattern. We write $\exists \bar{\alpha}[C]\Delta$ for $\exists \bar{\alpha}\bar{\beta}[C \wedge D]\Gamma$, and $\Delta_1 \times \Delta_2$ for:

$$\exists \bar{\beta}_1 \bar{\beta}_2 [D_1 \wedge D_2](\Gamma_1 \cup \Gamma_2)$$

Structures in which types can be interpreted have to meet three requirements, see [47] pages 18-19, of which we give two. Every constraint of the form $\tau_1 \rightarrow \tau_2 \preccurlyeq \varepsilon(\bar{\tau})$ or $\varepsilon(\bar{\tau}) \preccurlyeq \tau_1 \rightarrow \tau_2$ or $\varepsilon(\bar{\tau}) \preccurlyeq \varepsilon'(\bar{\tau}')$ for $\varepsilon \neq \varepsilon'$, is unsatisfiable. $\tau_1 \rightarrow \tau_2 \preccurlyeq \tau_1' \rightarrow \tau_2'$ entails $\tau_1' \preccurlyeq \tau_1 \wedge \tau_2 \preccurlyeq \tau_2'$.

Judgments about expressions retain the same form as in HM(X): they are written $C, \Gamma \vdash e\colon \sigma$, where C represents an assumption about the judgment's free type variables, Γ assigns type schemes to variables, and σ is the type scheme assigned to e.

$$\begin{array}{rcl}
\lambda(p_1.e_1 \cdots p_n.e_n)\, v & \to & \bigoplus_{i=1}^{n} e_i[p_i := v] \\
\mu x.v & \to & v[x := \mu x.v] \\
\mathbf{let}\, x = v\, \mathbf{in}\, e & \to & e[x := v] \\
E[e] & \to & E[e'] \quad \text{if} \quad e \to e' \\
\\
E & := & K\, \bar{v}\, [] \, \bar{e} \mid [] \, e \mid v \, [] \mid \mathbf{let}\, x = [] \, \mathbf{in}\, e
\end{array}$$

Table 2.3. Operational semantics for HMG(X)

Judgments about patterns are written $C \vdash p\colon \tau \rightsquigarrow \exists \bar{\beta}[D]\Gamma$, where the domain of Γ is $\mathrm{dpv}(p)$. Such a judgment can be read: under assumption C, it is legal to match a value of type τ against p; furthermore, if successful, this test guarantees that there exist types $\bar{\beta}$ that satisfy D such that Γ is a valid description of the values that the variables in $\mathrm{dpv}(p)$ receive. D carries the information unpacked from guarded data constructors K:

$$\frac{\forall i \quad C \wedge D \vdash p_i\colon \tau_i \rightsquigarrow \Delta_i \qquad K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha}) \qquad \bar{\beta} \# \mathrm{FV}(C)}{C \vdash K p_1 \cdots p_n\colon \varepsilon(\bar{\alpha}) \rightsquigarrow \exists \bar{\beta}[D](\Delta_1 \times \cdots \times \Delta_n)}$$

Subsequently, the unpacked information is available when type-checking the expression in the corresponding branch:

$$\frac{C \vdash p\colon \tau' \rightsquigarrow \exists \bar{\beta}[D]\Gamma' \qquad C \wedge D, \Gamma\Gamma' \vdash e\colon \tau \qquad \bar{\beta} \# \mathrm{FV}(C, \Gamma, \tau)}{C, \Gamma \vdash p.e\colon \tau' \to \tau} \tag{2.1}$$

The type judgments involving clauses are used to derive types of λ-abstractions, i.e. anonymous functions defined by cases:

$$\frac{\forall i \quad C, \Gamma \vdash c_i\colon \tau}{C, \Gamma \vdash \lambda(c_1 \cdots c_n)\colon \tau}$$

where τ is always of the form $\tau_1 \to \tau_2$.

Let us look at a selection of remaining rules. Patterns in a disjunction need to agree on the information they bring about:

$$\frac{\forall i \quad C \vdash p_i\colon \tau \rightsquigarrow \Delta}{C \vdash p_1 \vee p_2\colon \tau \rightsquigarrow \Delta}$$

However, we can make them agree by discarding irrelevant information:

$$\frac{C \vdash p\colon \tau \rightsquigarrow \Delta' \qquad C \Vdash \Delta' \leqslant \Delta}{C \vdash p\colon \tau \rightsquigarrow \Delta}$$

where $C \Vdash \Delta' \leqslant \Delta$ is defined in terms of interpreting environment fragments as sets of ground environments. Fortunately, we can equivalently define $C \Vdash \Delta' \leqslant \Delta$ as $C \wedge D' \Vdash \exists \bar{\beta}.D \wedge \Gamma' \preccurlyeq \Gamma$, where $\Gamma' \preccurlyeq \Gamma$ is a shorthand for $\mathrm{Dom}(\Gamma') = \mathrm{Dom}(\Gamma) \wedge_{x \in \mathrm{Dom}(\Gamma')} \Gamma'(x) \preccurlyeq \Gamma(x)$ and $\Delta = \exists \bar{\beta}[D]\Gamma$, $\Delta' = \exists \bar{\beta}'[D']\Gamma'$. Note that $C \Vdash D$ is defined as $\vDash C \Rightarrow D$, i.e. holding in the model, rather than $\vdash C \Rightarrow D$, i.e. provability.

To construct a value of a GADT, we need to check that the guard, or invariant, holds:

$$\frac{\forall i \quad C, \Gamma \vdash e_i : \tau_i \qquad K :: \forall \bar{\alpha} \bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}) \qquad C \Vdash D}{C, \Gamma \vdash K e_1 \cdots e_n : \varepsilon(\bar{\alpha})}$$

The use of a variable is type-checked in the same manner, the variable is looked up in the environment Γ. The following rule can be derived in HMG(X) and is a bit more clear than the original:

$$\frac{\Gamma(x) = \forall \bar{\alpha}[D].\tau \quad C \Vdash \exists \bar{\alpha}.D}{C, \Gamma \vdash x : \tau} \tag{2.2}$$

The rule for recursive definition is initially given in the Milner-Mycroft style to simplify proofs related to semantics rather than type inference:

$$\frac{C, \Gamma\{x \mapsto \sigma\} \vdash v : \sigma}{C, \Gamma \vdash \mu x.v : \sigma}$$

A v instead of an e is used to prevent diverging definitions. The final form of HMG(X), as employed in proofs of equivalence with the constraint derivation-based presentation, enforces the type annotation: $e := \ldots \mid \mu\,(x : \exists \bar{\beta}.\sigma).v$, where $\mathrm{ftv}(\sigma) \subseteq \bar{\beta}$.

A closed (i.e. without unbound variables) expression e is *well-typed* if and only if C, $\varnothing \vdash e : \sigma$ holds for some satisfiable constraint C. We have the following type soundness results:

THEOREM 2.1. (Subject reduction). *$C, \varnothing \vdash e : \sigma$ and $e \to e'$ imply $C, \varnothing \vdash e' : \sigma$.*

THEOREM 2.2. (Progress). *If e is well-typed and contains exhaustive case analyses only, then it is either reducible or a value.*

THEOREM 2.3. (Type soundness). *If e is well-typed and contains exhaustive case analyses only, then it does not reduce to a stuck expression.*

2.2.3. Constraint Derivation

The type inference for HMG(X), as well as for INVARGENT, starts by generating a constraint for an expression, whose satisfiability determines whether the expression is well-typed. The rules, or equations, for deriving the constraints form an alternative specification of the type system. Both specifications are readable, the specification by constraint derivation is actually more concise. As with the type system rules, we have constraint derivation equations for patterns, expressions, and pattern matching clauses.

The constraint $[\![p{\downarrow}\tau]\!]$ asserts that it is legal to match a value of type τ against p, while the environment fragment $[\![p{\uparrow}\tau]\!]$ represents knowledge about the bindings that arise when such a test succeeds. (Note that our use of $\downarrow$ and $\uparrow$ has nothing to do with bidirectional type inference.) The rules for $[\![p{\downarrow}\tau]\!]$ and $[\![p{\uparrow}\tau]\!]$ are the same in INVARGENT as in HMG(X). They are described in [47] on pages 32-33.

$$
\begin{array}{rcl}
[\![\Gamma \vdash x\colon \tau]\!] & = & \Gamma(x) \preccurlyeq \tau \\
[\![\Gamma \vdash \lambda \bar{c}\colon \tau]\!] & = & \exists \alpha_1 \alpha_2.\, \wedge_c [\![\Gamma \vdash c\colon \alpha_1 \to \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 \preccurlyeq \tau \\
[\![\Gamma \vdash e_1\, e_2\colon \tau]\!] & = & \exists \alpha.[\![\Gamma \vdash e_1\colon \alpha \to \tau]\!] \wedge [\![e_2\colon \alpha]\!] \\
[\![K e_1 \cdots e_n\colon \tau]\!] & = & \exists \bar{\alpha}\bar{\beta}.\, \wedge_i [\![\Gamma \vdash e_i\colon \tau_i]\!] \wedge D \wedge \varepsilon(\bar{\alpha}) \preccurlyeq \tau \\
 & & \text{where } K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha}) \\
[\![\Gamma \vdash \mu(x\colon \exists \bar{\beta}.\sigma).e\colon \tau]\!] & = & \exists \bar{\beta}.[\![\Gamma\{x \mapsto \sigma\} \vdash e\colon \sigma]\!] \wedge \sigma \preccurlyeq \tau \\
[\![\Gamma \vdash e\colon \forall \bar{\gamma}[C].\tau]\!] & = & \forall \bar{\gamma}.C \Rightarrow [\![\Gamma \vdash e\colon \tau]\!] \\
[\![\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2\colon \tau]\!] & = & [\![\Gamma\{x \mapsto \forall \alpha[C].\alpha\} \vdash e_2\colon \tau]\!] \wedge \exists \alpha.C \\
 & & \text{where } C \text{ is } [\![\Gamma \vdash e_1\colon \alpha]\!] \\
[\![\Gamma \vdash p.e\colon \tau_1 \to \tau_2]\!] & = & [\![p{\downarrow}\tau_1]\!] \wedge \forall \bar{\beta}.D \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!] \\
 & & \text{where } \exists \bar{\beta}[D]\Gamma' \text{ is } [\![p{\uparrow}\tau_1]\!]
\end{array}
$$

Table 2.4. Constraint derivation for HMG(X): expressions and clauses

We provide the HMG(X) constraint derivation rules for expressions and clauses in full in Table 2.4, because they are analogous to those in InvarGenT, but more concise. The novelty of HMG(X) compared to HM(X) resides in the last rule, which deals with clauses. The following description comes from [47] page 35. First, the function's domain type is required to match the pattern's type, via the constraint $[\![p{\downarrow}\tau_1]\!]$. Then, the clause's right-hand side e is required to have type τ_2 under a context extended with new abstract types $\bar{\beta}$ and a new typing hypothesis D and under an extended environment Γ, all three of which are obtained by evaluating $[\![p{\uparrow}\tau_1]\!]$.

2.2.4. Relevance

The specification of the type system in terms of constraints can be seen as even more declarative than the specification by natural deduction rules. Instead of requiring that the reader builds understanding by analysing the possible derivations, the constraint derivation rules reduce the meaning of type judgments to that of formulas, whose semantics is already known.

There are three major differences between HMG(X) and InvarGenT. To simplify the already daunting task, InvarGenT does not support subtyping. Recursive definitions do not carry type annotations – we perform type inference for polymorphic recursion. The third difference is the requirement in InvarGenT that the guards, or invariants, in type schemes and in value constructor definitions are existentially quantified conjunctions of atoms rather than arbitrary formulas. This is necessary to limit the space of candidate solutions to predicate variables in the task of synthesizing types and invariants of recursive definitions. But the restriction is also motivated by the need that the invariants be readily understandable to the programmer. Logically complex formulas, especially involving implications, are likely to not be sufficiently self-explanatory.

As a consequence of restricting type scheme invariants to conjunctive formulas, we cannot use the approach from Table 2.4 to let-polymorphism. Instead, we perform "division of labor": we have monomorphic bindings with a pattern on the left-hand-side $\mathbf{let}\, p = e\, \mathbf{in}\, e$, and polymorphic bindings, possibly recursive $\mathbf{let\ rec}\, x = e\, \mathbf{in}\, e$.

2.3. THE OutsideIn(X) SYSTEM

The design of the Schrijvers, Peyton Jones, Sulzmann and Vytiniotis OutsideIn(X) [53] type system is a follow-up to the OUTSIDEIN algorithm from [45] in a broader context parameterized by a constraint logic. The type judgments in OutsideIn(X) are similar to HMG(X). $C, \Gamma \vdash e\colon \tau$ means that in a context where the constraint C is available, and in type environment Γ, the term e has type τ. The important difference is that C is a conjunction of atoms rather than an arbitrary formula. Therefore, we cannot formulate an alternative, constraint derivation based specification of the type system, with the elegant connection expressed by results like Theorem 3.1. Another difference between OutsideIn(X) and HMG(X) is that OutsideIn(X) is defined against a background of a proof system for checking satisfiability of formulas rather than against an abstract model. The relevance is that having a model gives more flexibility for the implementer of type inference, while having a proof system (i.e. a logic) gives more flexibility for the designer of the type system. The third, related difference is that OutsideIn(X) uses quantifiers sparingly. Instead, it introduces a distinction into *skolem* type variables and *unification* type variables.

OutsideIn(X) has rules for type checking programs as well as single expressions. Notably, the rule for un-annotated bindings makes use of abduction:

$$\frac{C_1, \Gamma \vdash e\colon \tau \quad \bar{\alpha} = \mathrm{FV}(C, \tau) \quad \mathcal{C} \wedge C \Vdash C_1 \quad \mathcal{C}, \Gamma\{f \mapsto \forall \bar{\alpha}[C].\tau\} \vdash \mathrm{prog}}{\mathcal{C}, \Gamma \vdash f = e, \mathrm{prog}}$$

We determine the constraint C_1 which is required to make e typeable with type τ in Γ. Next, we allow the invariant of f be a simplified version of C_1, namely C. Intuitively, C is the "extra information", not deducible from $\mathcal{C}$, that is needed to show the required constraint C_1 (see [53] page 15). In our terms, C is an answer to an abduction problem $\mathcal{C} \Rightarrow C_1$.

Constraint generation in OutsideIn(X) is similar to that in HMG(X), but the formulation is less concise, and the generated constraint does not have alternating quantifiers. In our opinion, it makes the semantics of the constraints less clear. The constraints are also more restrictive than if quantifier scope was used to determine which solutions are consistent, even for programs without type families or GADTs.

2.3.1. Type Inference

[53] page 20 defines a *sound solution*, which is equivalent to our notion of an answer to a simple abduction problem, and a *guess-free solution*, which is equivalent to our notion of a fully maximal answer to a simple abduction problem. Type inference in OutsideIn(X) uses a solver of simple abduction problems, but the abduction answers are not allowed to participate in the inferred types. Instead, for type inference to succeed, the abduction answers to implications have to be limited to substitutions of local variables, which are subsequently ignored.

To make the exposition more concrete, we rewrite the solver infrastructure specification from [53] page 41. Let A, D_i, C_i be conjunctions of atoms. Let $A \in \mathrm{Abd}_{\mathcal{C}}(D_i, C_i)$ mean that A is an answer to the simple abduction problem $D_i \Rightarrow C_i$ under theory $\mathcal{C}$, i.e. $\mathcal{C} \wedge D_i \wedge A \Vdash C_i$. Let $C \equiv \mathrm{Simple}(C) \wedge \mathrm{Implic}(C)$ be a decomposition of a constraint C into a conjunction of atoms $\mathrm{Simple}(C)$ and a conjunction of existentially quantified implications $\mathrm{Implic}(C)$. For a substitution $R = [\bar{\alpha} := \bar{\tau}] = [\alpha_1 := \tau_1, ..., \alpha_n := \tau_n]$, let $\dot{R} = \bar{\alpha} \doteq \bar{\tau} = \alpha_1 \doteq \tau_1 \wedge ... \wedge \alpha_n \doteq \tau_n$. We define the solver recursively:

$$\frac{\begin{array}{l} C_r \wedge \dot{R} \in \mathrm{Abd}_{\mathcal{C}}(C_g, \mathrm{Simple}(C)) \quad \mathrm{Dom}(R) \subseteq \bar{\alpha} \\ \forall (\exists \bar{\alpha}_i . D_i \Rightarrow C_i \in \mathrm{Implic}(C)) . \mathrm{Solve}(\mathcal{C}; C_g \wedge C_r \wedge D_i; \bar{\alpha}_i; C_i) \rightsquigarrow (\varnothing, R_i) \end{array}}{\mathrm{Solve}(\mathcal{C}; C_g; \bar{\alpha}_i; C_w) \rightsquigarrow (C_r, R)}$$

While it appears that the type inference solution is built out of abduction answers (C_r, R), the interesting simple abduction problems $D_i \Rightarrow C_i$ are required to have abduction answer $(\dot{R}_i)$ limited to variables $\bar{\alpha}_i$, which can be subsequently ignored.

Motivated by considerations of efficiency and avoiding ambiguity, [53] restricts the abduction algorithms to those only searching for fully maximal answers. [53] conjectures that the algorithm they provide is complete wrt. fully maximal answers to simple abduction problems. OutsideIn(X) might not be able to use the abduction algorithm complete wrt. fully maximal answers to simple *constraint* abduction problems from [29], even when limited to GADTs, because the OutsideIn(X) type system is based on a logic instead of on fixed models like Herbrand structures.

2.3.2. Relevance

The OutsideIn(X) project exposition [53] lists multiple challenges as being in its focus. Of these, InvarGenT addresses GADTs, and is open to address *units of measure*, although the corresponding sort has not been implemented. The other challenges fall outside of the scope of InvarGenT: type-class constraints, multi-parameter type classes with functional dependencies, and type families with type family axioms. Some of these type system features, when implemented directly, violate the requirement on the interpretation of types that we preserve from HMG(X): "Every constraint of the form $\tau_1 \to \tau_2 \preccurlyeq \varepsilon(\bar{\tau})$ or $\varepsilon(\bar{\tau}) \preccurlyeq \tau_1 \to \tau_2$ or $\varepsilon(\bar{\tau}) \preccurlyeq \varepsilon'(\bar{\tau}')$ for $\varepsilon \neq \varepsilon'$, is unsatisfiable." (In InvarGenT, we have equality $\doteq$ instead of subtyping $\preccurlyeq$.)

[53] argues strongly against selecting an arbitrary correct type when a definition does not have a most general type. InvarGenT does select an arbitrary type without guarantees, although the implementation is designed to pick useful types; e.g. if possible, with the return type of a function sharing a parameter with an argument type. InvarGenT is intended to provide type signatures for toplevel definitions to the programmer. The programmer would then either accept the signature, modify the program and re-generate the signature, or modify the signature directly.

OutsideIn(X) shares with InvarGenT the restriction of type scheme invariants, and guards of data constructors, to conjunctions of atoms. On other points of difference between OutsideIn(X) and HMG(X), InvarGenT follows HMG(X). Both OutsideIn(X) and InvarGenT infer types for toplevel definitions one at a time.

Finally, the type inference algorithm of OutsideIn(X) is much weaker than that of InvarGenT, as illustrated in our presentation of the central step of the algorithm in terms of abduction. While our algorithm uses abduction to search for the solution of the type inference problem, OutsideIn(X) only uses abduction to verify a partial solution found by unification.

2.4. Pointwise GADTs

Chuan-kai Lin in [22], see also Lin and Sheard [23], presents a type inference algorithm for GADTs. Just as InvarGenT, the type system Pointwise GADTs and the algorithm $\mathcal{P}$ from [22] does not require type annotations. The Pointwise GADTs is not a constraint-based type system. Also, it does not support deep patterns. Let us look at the two most important type system rules. The recursive definition rule captures polymorphic recursion:

$$\frac{\Gamma\{x \mapsto \forall\bar{\alpha}.\tau'\} \vdash e_1 \colon \tau' \quad \bar{\alpha}\#\mathrm{FV}(\Gamma) \quad \Gamma\{x \mapsto \sigma\} \vdash e_2 \colon \tau}{\Gamma \vdash \mathbf{let\ rec}\, x = e_1 \,\mathbf{in}\, e_2 \colon \tau}$$

The rule for pattern matching expressions is straightforward, nearly the same as the corresponding derived rule in HMG(X). We turn to the pattern matching branch rule:

$$\frac{\begin{array}{ll} K :: \forall\bar{\alpha}.\bar{r} \longrightarrow \varepsilon\, \bar{s} & \bar{\alpha}\#\mathrm{FV}(\Gamma, \bar{u}, \tau) \\ S = \boldsymbol{PU}(\varepsilon\, \bar{u} \doteq \varepsilon\, \bar{s}) & S(\Gamma\{\bar{x} := \bar{r}\}) \vdash e \colon S(\tau) \end{array}}{\Gamma \vdash_p K\, \bar{x}.e \colon \varepsilon\, \bar{u} \to \tau}$$

PU stands for *pointwise unification*. It is a limited form of unification. If $U = \boldsymbol{PU}(s \doteq t)$, then $U(s) = U(t)$, but also: for $\alpha \in \mathrm{Dom}(U)$, if $s \downarrow_p = \alpha$, i.e. s has variable α at position p, then $t \downarrow_p = U(\alpha)$; similarly, if $t \downarrow_p = \alpha$, then $s \downarrow_p = U(\alpha)$.

The type inference algorithm $\mathcal{P}$ is deliberately more restrictive than the type system Pointwise GADTs. In particular, it uses *anti-unification* to infer a tight type for the pattern-matched expression, so that the pattern matching has a better chance of being exhaustive given the type.

2.4.1. Type Inference

Algorithm $\mathcal{P}$ is based on Robin Milner's algorithm $\mathcal{W}$ as in [32], and its modification by Alan Mycroft to handle polymorphic recursion as in [33]. See [22] pages 152-184 for detailed presentation. We reproduce details of the algorithm $\mathcal{P}$, because algorithm $\mathcal{P}$ provides a baseline wrt. which type inference for GADTs not relying on type annotations can be compared. We defer the reader who finds the current section cryptic to Chuan-kai Lin [22].

It might be worthwhile to collect the major pieces of algorithm $\mathcal{P}$ together:

$$\begin{array}{rcl} \mathrm{infer}(\Gamma, e_1\, e_2) & = & \\ (S_1, \tau_1) & = & \mathrm{infer}(\Gamma, e_1) \\ (S_2, \tau_2) & = & \mathrm{infer}(\Gamma, e_2) \\ S_3 & = & \boldsymbol{U}(\tau_1 \doteq \tau_2 \to \beta),\ \beta \text{ fresh} \\ S & = & \boldsymbol{C}(S_1, S_2, S_3) \\ & & (S, S(\beta)) \end{array}$$

$$
\begin{array}{rcl}
\mathrm{infer}(\Gamma, \textbf{let rec}\, x = e) & = & \mathrm{polyrec}(\{\}, \forall\alpha.\alpha) \\
\mathrm{polyrec}(S_1, \sigma) & = & \\
(S_2, \tau) & = & \mathrm{infer}(\Gamma\{u \mapsto \sigma\}, e) \\
\bar{\beta} & = & \mathrm{FV}(\tau)\backslash\mathrm{FV}(\Gamma\{u \mapsto \sigma\}) \\
\sigma' & = & \forall\bar{\beta}.\tau \\
\text{if} & & S_2(\sigma) = \sigma' \\
\text{then} & & (S_2\, S_1, \sigma') \\
\text{else} & & \mathrm{polyrec}(S_2\, S_1, \sigma')
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{infer}(\Gamma, \textbf{case}\, e\, \textbf{of}\, \overline{p_i.e_i}) & = & \\
(S_0, u_0) & = & \mathrm{infer}(\Gamma, e) \\
(S_i, u_i, \tau_i) & = & \mathrm{infer}_{\mathrm{ALT}}(\Gamma, \beta, p_i.e_i),\ \beta \text{ fresh} \\
u & = & \mathrm{LUB}(\tau_1, ..., \tau_n) \\
S & = & \boldsymbol{U}((u, ..., u) \doteq (u_0, u_1, ..., u_n)) \\
s & = & S(u) \\
R_i & = & \boldsymbol{C}(S, S_0, S_i, \boldsymbol{U}(u_i \doteq \tau_i)) \\
\bar{\alpha} & = & \mathrm{FV}(s) \cap (\cup_i \mathrm{Dom}(R_i)) \\
\bar{\beta} & = & \mathrm{FV}(s)\backslash\bar{\alpha} \\
\mathrm{trt} & = & \mathrm{tabulate}(\bar{\alpha}, \bar{R_i}) \\
\mathrm{btt} & = & \mathrm{tabulate}(\beta\mathrm{FV}(\Gamma), \bar{R_i}) \\
R & = & \mathrm{reconcile}(\bar{\beta}, \mathrm{trt}, \mathrm{btt}) \\
 & & (R, R(\beta))
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{infer}_{\mathrm{ALT}}(\Gamma, \beta, K\,\bar{x}.e) & = & \\
\forall\bar{\alpha}.\bar{r} \longrightarrow \varepsilon\,\bar{s} & = & \mathrm{lookup}(K) \ \text{ where } \bar{\alpha} \text{ fresh} \\
(S, \tau) & = & \mathrm{infer}(\Gamma\{\overline{x \mapsto r}\}, e) \\
\bar{\alpha}' & = & \bar{\alpha} \cap (\mathrm{Dom}(S) \cup \mathrm{FV}(\mathrm{Rng}(S)) \cup \mathrm{FV}(\tau)) \\
\bar{\gamma} & = & \{\gamma | \gamma \in \bar{\alpha} \wedge \mathrm{retain}(\bar{\alpha}, S, \gamma)\} \\
\text{if} & & \bar{\alpha}' \not\subseteq \mathrm{FV}(\bar{s}) \\
\text{then} & & \bot \\
\text{else if} & & \bar{\gamma} = \varnothing \\
\text{then} & & ([\beta := \tau]\, S, \varepsilon\,\bar{\gamma}', \varepsilon\,\bar{s}) \text{ where } \bar{\gamma}' \text{ fresh} \\
\text{else} & & ([\beta := \tau]\, S, S(\mathrm{transcb}(\bar{\gamma}, \varepsilon\,\bar{s})), \varepsilon\,\bar{s})
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{retain}(\bar{\alpha}, S, \gamma) & = & \\
S(\gamma) = \varepsilon'\,\bar{r} & \rightarrow & \boldsymbol{T}
\end{array}
$$

$$S(\gamma)=\alpha \;\rightarrow\; \exists\beta\in\bar{\alpha}.\beta\neq\gamma\wedge\alpha\in\mathrm{FV}(S(\beta))$$

$$\begin{array}{rcl}
\mathrm{transcb}(\bar{\gamma},\tau) & = & \\
\text{if} & & \bar{\gamma}\#\mathrm{FV}(\tau) \\
\text{then} & & \beta \text{ where } \beta \text{ fresh} \\
\text{else} & & \\
\tau=\varepsilon'\,\bar{r} & \rightarrow & \varepsilon'\,\overline{\mathrm{transcb}(\bar{\gamma},r)} \\
\tau=\alpha & \rightarrow & \alpha
\end{array}$$

The operation $\boldsymbol{C}$ is a combination of substitutions having the effect of unification of the corresponding equations:

$$\boldsymbol{C}(S_1,...,S_n)=\boldsymbol{U}\left(\wedge_i\dot{S}_i\right)=\boldsymbol{U}((\bar{a}_1,...,\overline{\alpha_n})\doteq(S_1(\bar{\alpha_1}),...,S_n(\overline{\alpha_n}))) \text{ where } \bar{\alpha_i}=\mathrm{Dom}(S_i)$$

The algorithm is designed after algorithm $\mathcal{W}$. However, it is made more modular by the use of $\boldsymbol{C}$. This can be seen as a "concealed" form of constraint derivation based type inference, where the partial constraints are solved during collection. The case $\mathrm{infer}(\Gamma, \textbf{let rec}\, x=e)$ is the Mycroft's modification of algorithm $\mathcal{W}$ to handle polymorphic recursion. It introduces iteration of type inference, for each recursive definition separately, till the definition's type converges. The $\mathrm{LUB}(\tau_1,...,\tau_n)$ subroutine used in $\mathrm{infer}(\Gamma, \mathrm{case}\, e \,\mathrm{of}\, \overline{p_i.e_i})$ is *anti-unification*, a special case of what we will call *constraint generalization*. It computes the type u for the expression e as specific as possible while agreeing with all types imposed by the pattern matching branches. The complications in handling variables in $\mathrm{infer}_{\mathrm{ALT}}(\Gamma,\ \beta,\ K\ \bar{x}.e)$, in particular the failure condition $\bar{\alpha}'\nsubseteq\mathrm{FV}(\bar{s})$, are analogous to the fact that the variables $\bar{\alpha}$ would be universally quantified by $\mathrm{HMG}(X)$, and thus cannot be part of, for example, the domain of the resulting substitution.

The aspect of algorithm $\mathcal{P}$ that is closest to the intricacies of term constraint abduction is the joint use of the tabulate and reconcile subroutines (not presented above). They decompose respectively the pattern-matched expression type, and the pattern-matching branch body type, when the types start with the same type constructor ε across branches. When the types across branches do not agree, but a btt column corresponding to a subterm of the result type, i.e. branch bodies type, is unifiable (branch-wise) with exactly one trt column, then the subterm of the result type is replaced by the corresponding subterm of the pattern-matched expression type.

2.4.2. Relevance

Algorithm $\mathcal{P}$ is designed to infer types for GADT programs without relying on type annotations. This is the major goal of InvarGenT, alongside inference of existential types and ease of extension with novel sorts. The type system behind algorithm $\mathcal{P}$ is not constraint based, and in particular, it has no mechanism for extension with, for example, numerical invariants. The Pointwise GADTs type system is much more restrictive than the plain GADTs type system around which InvarGenT is designed. Algorithm $\mathcal{P}$ is very efficient compared to InvarGenT, but still quite capable. Some inspirations for InvarGenT can be drawn from algorithm $\mathcal{P}$.

The examples from [22] motivated the extension of simple constraint abduction algorithm for terms in InvarGenT to non-fully-maximal answers. The extension allows guessing equations between invariant parameters. The use of anti-unification in algorithm $\mathcal{P}$ is interesting and the prospect of using constraint generalization in InvarGenT to infer a tighter type when otherwise a pattern matching expression would not be exhaustive, is intriguing, but we leave it as future work. Finally, table-based approach to *GADT type refinements* in algorithm $\mathcal{P}$, via the tabulate and reconcile subroutines, might inspire future work on augmenting the sequential approaches to joint constraint abduction, by "simultaneous" stages, operating jointly on all branches.

2.5. Liquid Types

Tim Freeman and Frank Pfenning's *Refinement Types for ML* [15] predates even the Odersky, Sulzmann and Wehr [34]. Refinement types are so called because they refine the ML type system, rather than extending it as GADTs do – they do not make new programs typeable. Refinement types became the universally and existentially quantified types of DML. More recently, they were independently developed in an inference-friendly way by Cormac Flanagan [13] and Knowles and Flanagan [20]. The work was continued by Ranjit Jhala, Patrick Rondon and collaborators, and their systems DSolve [41] and HSolve [52] have become more popular. They are fully focused on automatic invariant inference. The work behind DSolve ([41], expanded in [42] and [40]) is the topic of this section. [41] cites the work on predicate abstraction [1], [17] as precursory for this kind of invariant inference; and Jeffrey Scott Foster on Type Qualifiers [14], as inspiring some aspects of inference.

2.5.1. The Type System

Type refinements can be seen as preconditions (or postconditions) when they refine the argument type (or the result type) of a function. The term *Liquid Types* introduced by [41] comes from "logically qualified types", a restriction of an otherwise more depent-types-like type system, limiting type refinements to conjuctions of atoms from a sort of linear inequalities and uninterpreted functions. The refinements are over base (i.e. non-function) types, and are written: $\{\nu{:}\,B\,|e\}$, where B is a base type like `int`, e is a conjunction of atoms, and ν is a singled-out variable constrained by e. (In the formalism used by InvarGenT, the letter δ will play the role of ν here.) Recall the typing judgment form $C, \Gamma \vdash e{:}\,\tau$ we encountered in systems described earlier: assuming constraint C holds, in environment Γ, expression e has type τ. Although the Liquid Types typing judgment $\Gamma \vdash e{:}\,T$ (where T stands for a liquid type or type scheme) seems simpler, here C is a part of Γ. The assumed constraint is:

$$[\![\Gamma]\!] \equiv \wedge_{e\in\Gamma} e \wedge_{x:\{\nu:B|e\}\in\Gamma} e[\nu := x]$$

The Liquid Types type system is based on subtyping rather than type equality. Here, subtyping implicitly captures the requirements on the assumed constraint. The subtyping judgment has the form $\Gamma \vdash T_1 <: T_2$, which means: $[\![\Gamma]\!]$ implies that T_1 is a subtype of T_2.

The function type arguments are indexed by the variable corresponding to the λ-abstraction introducing the type: $\Gamma \vdash \lambda x.e\colon (x\colon T_x \to T)$. Together with the following rules:

$$\frac{\Gamma(x) = \{\nu\colon B \,|\, e\}}{\Gamma \vdash x\colon \{\nu\colon B \,|\, \nu \doteq x\}}$$

$$\frac{\Gamma \vdash e_1\colon S \quad \Gamma; x\colon S \vdash e_2\colon T}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2}$$

this avoids the use of existential quatification in inference constraints. The crucial rules relevant for invariants are:

$$\frac{\Gamma \vdash e_1\colon \mathrm{Bool} \quad \Gamma; e_1 \vdash e_2\colon T \quad \Gamma; \neg e_1 \vdash e_3\colon T}{\Gamma \vdash \mathbf{if}\, e_1\, \mathbf{then}\, e_2\, \mathbf{else}\, e_3\colon T}$$

$$\frac{\mathrm{Valid}([\![\Gamma]\!] \wedge e_1 \Rightarrow e_2)}{\Gamma \vdash \{\nu\colon B \,|\, e_1\} <\colon \{\nu\colon B \,|\, e_2\}}$$

In the first rule, the expressions e_1 are limited to well-formed formulas of the constraint domain. Note how the former rule resembles the HMG(X) Rule 2.1 in that the local constraint is enhanced by the information pertaining to the conditional branch. The latter rule resembles the Rule 2.2 in that the local constraint is used as a premise to ensure the validity of the use of a value.

2.5.2. Liquid Types Inference

Similarly to INVARGENT, the DSOLVE system first generates the inference constraint for the whole toplevel expression, and then proceeds to solve the constraint. DSOLVE uses an oracle to solve the Hindley-Milner polymorphic type inference subproblems, and the resulting type shapes guide the construction of the constraint. In this way, the constraint describes purely the subtyping issues, rather than both the typing and subtyping issues. Each subtyping requirement contributes what would be an implication in a normalized INVARGENT constraint. The Liquid Type Inference Algorithm, including constraint generation, can be found in [41], page 9, Figure 4. Liquid Types inference introduces *liquid type variables* κ, standing for unknown invariants. (In INVARGENT, we call them *predicate variables* χ.) The inference task is to find a substitution for the liquid type variables such that all the implications corresponding to subtyping constraints are valid. We present the liquid type variable solving part of the inference algorithm:

$$\begin{aligned}
\mathrm{Weaken}\,(\Gamma \vdash \{\nu\colon B \,|\, \theta \cdot \kappa\})\,A &= A \circ [\kappa := \\
&\qquad \{q \in A(\kappa) \,|\, \Gamma; \nu\colon B \vdash \theta(q)\colon \mathrm{Bool}\}] && (2.3)\\
\mathrm{Weaken}\,(\Gamma \vdash \{\nu\colon B \,|\, \rho\} <\colon \{\nu\colon B \,|\, \theta \cdot \kappa\})\,A &= A \circ [\kappa := \\
&\qquad \{q \in A(\kappa) \,|\, [\![A(\Gamma)]\!] \wedge A(\rho) \Rightarrow \theta(q)\}] && (2.4)\\
\mathrm{Weaken}\ _\ A &= \bot && (2.5)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Solve}\, C\, A \text{ when } \exists c \in C \\
&\text{where } A(c) \text{ is not valid} &&= \mathrm{Solve}\, C\, (\mathrm{Weaken}\, c\, A)\\
&\mathrm{Solve}\ _\ A &&= A
\end{aligned}$$

The pair $\theta \cdot \kappa$ of a substitution θ and liquid type variable κ is a *delayed substitution*, see Knowles and Flanagan [20]. Entry 2.3 above ensures that the solution formulas are well-formed. Entry 2.4 filters out the atoms which are not implied by the premise of the implication considered. Entry 2.5 is the failure case: the offending implication cannot be made valid. It cannot be weakened, as it does not contain a liquid type variable in the conclusion, and the premise is already as strong as possible. The initial call to Solve passes as the initial solution A, for each liquid type variable κ, a conjunction of *all* atoms over the potential parameters of invariants appearing in constraints.

2.5.3. Relevance

It will be illuminating for Chapter 4 to compare the solver of DSolve with that of InvarGenT. In a single iteration of the main algorithm of: DSolve, a single invariant (i.e. κ substitution) is updated; InvarGenT, all invariants (i.e. χ substitutions) are updated. This makes: in DSolve, a single implication valid, given before-update substitution of liquid type variables in premises; in InvarGenT, all implications valid, given before-update substitution of "liquid type", i.e. predicate variables in conclusions. DSolve focuses on the role liquid type variables play in conclusions, and ignores their role in premises, other than to verify validity of implications. InvarGenT focuses on the role all predicate variables play in premises, but also, using a different mechanism, on postcondition predicate variables in conclusions. DSolve starts with a full initial solution (a conjuction of all atoms, trivially contradictory); InvarGenT starts with an empty initial solution (an empty conjunction, trivially satisfiable). DSolve finds the most specific (least general) solution for all invariants. InvarGenT finds the least specific (most general) solution for preconditions, and the most specific solution for postconditions given these preconditions.

Here are three arguments in favor of InvarGenT over DSolve. The major limitation of DSolve is the need to generate all the atoms as the initial solution; in InvarGenT, the addition of atoms to the solution is driven by the conclusions that need to be explained. Least general preconditions are less useful than most general preconditions. Furthermore, in principle, there can be solutions that are overlooked by solving one invariant at a time.

There are two shortcomings of InvarGenT relative to DSolve that should inspire future work; we start with the major one. InvarGenT prohibits passing values of an existentially quantified type as arguments, and it does not allow type schemes as argument types. Thus functions like `ffor` from the `fft` example, see Appendix C Subsection C.8.13, require manual packing and unpacking for the argument of the function passed to the higher-order function. DSolve manages to infer types for such higher-order functions, and this ability is further improved in Abstract Refinement Types: Vazou, Rondon and Jhala [52].

The other shortcoming is that, while InvarGenT is faster with simple input programs, DSolve is faster with more complex programs for which it is able to find a solution. There are two reasons. One is that DSolve entirely eliminates variables which cannot be invariant parameters before starting the solution process, i.e. intermediate variables of the Hindley-Milner inference process. InvarGenT deploys the general mechanism of abduction to solve for all variables. The other reason is that DSolve only checks all constraints for contradiction once every update of an invariant κ. InvarGenT checks all constraints for contradiction for every candidate atom to be added during an update of invariants.

2.6. HERBRAND CONSTRAINT ABDUCTION

We have invoked "abduction" multiple times already, it is time to define the term. *Abduction* is a form of inference where we find an explanation of a target formula (often referred to as an observation) given background knowledge. Given a signature Σ and a set of variables Vars, a *constraint domain* is a pair $(\mathcal{D}, \mathcal{L})$ where $\mathcal{D}$ is a Σ-structure and $\mathcal{L}$ (the language of constraints) is a set of Σ-formulas closed under conjunction and renaming of free variables. *Constraint abduction* is a formalization of abduction in the context of a constraint domain. *Simple Constraint Abduction* is the task of solving an implication $B \Rightarrow C$, where $B, C \in \mathcal{L}$ are conjunctions of atoms. The constraint abduction answer $A \in \mathcal{L}$ is a solution to the implication $B \Rightarrow C$ if and only if $\mathcal{D} \vDash (A \wedge B) \Rightarrow C$ and $\mathcal{D} \vDash \exists \mathrm{FV}(A, B).A \wedge B$, where $\mathrm{FV}(\cdot)$ finds the free variables of an expression. $\mathcal{D}$ and B play the role of background knowledge – $\mathcal{D}$ general, and B context specific. C plays the role of observation, and A of explanation. If $B \Rightarrow C$ is a conjunct in a type inference constraint, then B contains the background information about a particular location in a program's source code, coming from the pattern matching patterns that "are the case" – the location is in their scope. C contains the requirements imposed by the source at that location, for example the preconditions of the functions that are called. The answer A explains what needs to be the case for the requirements to be met. *Joint Constraint Abduction* is the task of solving implications $\wedge_i(B_i \Rightarrow C_i)$, where B_i, $C_i \in \mathcal{L}$ are conjunctions of atoms and $\wedge_i \varphi_i$ stands for $\varphi_1 \wedge \ldots \wedge \varphi_n$. The answer $A \in \mathcal{L}$ to this Joint Constraint Abduction problem has to meet the conditions: $\mathcal{D} \vDash (A \wedge B_i) \Rightarrow C_i$ and $\mathcal{D} \vDash \exists \mathrm{FV}(A, B_i).A \wedge B_i$, for all i. The suitability of joint constraint abduction for GADTs type inference was probably first observed by Martin Sulzmann, Tom Schrijvers and Peter J. Stuckey, see [50] (originally [48]) and [49].

Michael Maher has studied constraint abduction for terms ([27], [29]) and linear arithmetic ([26]). Michael Maher and Ge Huang [29] provided the basis for our Simple Constraint Abduction algorithm for terms, which we describe in this section. An abduction answer A is *maximally general*, when for every other abduction answer A', if $\mathcal{D} \vDash A \Rightarrow A'$, then $\mathcal{D} \vDash A' \Rightarrow A$. An abduction answer A to $B \Rightarrow C$ is *fully maximal*, when it is maximally general and $\mathcal{D} \vDash (A \wedge B) \Leftrightarrow (B \wedge C)$. A fully maximal answer does not "guess" any fact not entailed by the formulas considered.

Let $\mathcal{D} = \mathcal{T} = T(\Sigma, \mathrm{Vars})$ be the free algebra of terms over signature Σ with variables Vars and $\mathcal{L} = \mathcal{FT}_\exists$ be existentially quantified conjunctions of equations. In Table 2.5 we quote the fully maximal abduction algorithm from [29], Figure 4, page 13. The following theorem is Theorem 6 from [29], page 14.

THEOREM 2.4. *Algorithm* FMA *outputs all fully maximal answers to the SCA problem over* $\mathcal{FT}_\exists$ *and terminates.*

Now let us turn to the joint problem. The following proposition is Proposition 8 from [27], page 9. In Table 2.5 we quote the corresponding algorithm **JCA-Solve**.

PROPOSITION 2.5. *Consider a joint constraint abduction problem composed of n SCA component problems. Let* $\mathcal{A} = \{\wedge_{i=1}^n A_i | A_i$ *is a maximally general answer of the i'th SCA problem*$\}$. *If A is a maximally general answer to the JCA problem then* $A \in \mathcal{A}$. *The JCA problem has no answers iff no constraint in* $\mathcal{A}$ *is an answer.*

$\text{pos}(t, A)$ returns the set of positions of the term t in the right-hand-side of A
$\text{repl}(S, A)$ replaces all terms in the right-hand-side of A occurring at a position in S by a new variable (that is existentially quantified)
$\text{next}(A)$ is the set $\{\text{repl}(S, A) | \varnothing \subsetneq S \subset \text{pos}(t, A), t \notin \text{Vars}\}$

algorithm FMA(B, C)
  if $\vDash B \Rightarrow C$ **then return** $\top$
  let A be the standard form of $B \wedge C$
  do
    let $\mathcal{A}$ be $\text{next}(A)$
    if $(\forall A' \in \mathcal{A}. \nvDash A' \wedge B \Rightarrow C)$ **then return** A
    choose $A \in \mathcal{A}$ such that $\vDash A \wedge B \Rightarrow C$

algorithm JCA-Solve
  $\mathcal{M} := \{\wedge_{i=1}^{n} A_i | A_i$ is a maximally general answer of the i'th SCA problem$\}$
  while exist $A \in \mathcal{M}$ and i s.t. $A \wedge B_i$ is unsatisfiable in $\mathcal{D}$, $\mathcal{M} := \mathcal{M} \setminus \{A\}$
  if $\mathcal{M} = \varnothing$ **then return** $\bot$
  while exist $A, A' \in \mathcal{M}$ s.t. $\mathcal{D} \vDash A \Rightarrow A'$ and $A \neq A'$, $\mathcal{M} := \mathcal{M} \setminus \{A\}$
  return $\mathcal{M}$

Table 2.5. SCA algorithm for computing fully maximal answers, JCA algorithm

2.6.1. Relevance

Sulzmann, Schrijvers and Stuckey [48] and [49] inspired the approach taken by the InvarGenT project. Maher and Huang [29] forms a basis of our simple constraint abduction solver for the Herbrand domain. It turns out however, that fully maximal answers are insufficient for type and invariant inference. We go beyond the algorithms from Table 2.5 as follows.

We extend the FMA(B, C) algorithm by considering atoms $x \doteq y$, where $x, y \in \text{Vars}$, $x \doteq t_x$ and $y \doteq t_y$ belong to the solved form of $B \wedge C$, $t_x \notin \text{Vars}$ and $t_y \notin \text{Vars}$, and $t_x \doteq t_y$ is satisfiable. These conditions limit the number of pairs of variables to consider. There are many potential SCA problems where we still will not be able to find an answer, but they appear not to be relevant: we have not encountered a practical example where the algorithm would need further extension.

We extend the **JCA-Solve** algorithm by using the partial solution, i.e. starting from $B \wedge C \wedge_{i=1}^{k-1} A_i$ instead of $B \wedge C$, when solving the kth component SCA problem.

We propose a novel algorithm (not based on [26]) for constraint abduction for the linear arithmetic domain.

Chapter 3
The Type System

The main idea of this work is that useful properties of functions can be generated by solving ordinary-looking constraints under a quantifier prefix (without resorting to Herbrandization), derived via a GADTs-based type system. The content of Sections 3.2 and 3.3 is based on Simonet and Pottier [47]. To our knowledge, the idea behind the NegClause rule is novel. The content of Sections 3.4 and 3.5 is novel.

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 mean disjointedness. With a free index i, $\bar{e_i}$ means $(e_1, ..., e_n)$ for some n associated with the index i; i.e. $\bar{e_i}$ or $\bar{e}$ is a sequence $(e_1, ..., e_n)$ and e_i is an ith element of the sequence. 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 $\{c_1, ..., c_n\}$. Sometimes we write $\bar{v}$ for a sequence of variables related to but distinct from a variable v.

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). Equality with a dot $\doteq$ is an object-level equality, i.e. a relation in the language $\mathcal{L}$ introduced below. Equality $=$ is a meta-level equality, usually syntactic equality on terms or formulas. 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. We use letters R, S, U to denote substitutions. For a substitution $S = [\bar{\alpha} := \bar{t}]$, 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 $\alpha_2 \leqslant_{\mathcal{Q}} \alpha_1$. Syntactically, this means that α_i is not to the right of variables in t_i: $\forall \beta \in \mathrm{FV}(t_i),\ \alpha_i \not<_{\mathcal{Q}} \beta$. We use letters τ, r, s, t, u to denote terms and letters $\alpha, \beta, v, x, y, z$ to denote variables.

3.1. The Language of Constraints

We are interested in a multi-sorted first order language $\mathcal{L}_1$ with equality, interpreted in a given, fixed model $\mathcal{M}$. Even when we write $\vDash \Phi$, it is usually a shortcut notation for validity of a formula Φ in the model $\mathcal{M}$: $\mathcal{M} \vDash \Phi$. The sort of terms or "types proper", denoted s_{type} and `type` in the concrete syntax of InvarGenT, plays a special role. In the current presentation, we abstract from details of the language, posing the necessary properties as assumptions.

In Appendix A, we introduce a Henkin semantics for existential second order logic $\mathcal{L}$ extending $\mathcal{L}_1$ by predicate symbols $\chi(\cdot)$ that we call *predicate variables*. $\mathcal{L}$ is tailored to our needs of invariant and postcondition inference. Let $\mathrm{PV}(\cdot)$ be the set of predicate variables in

$$
\begin{array}{lcl}
\tau_{\text{type}} & := & v_{\text{type}} \,|\, \tau_{\text{type}} \to \tau_{\text{type}} \,|\, \varepsilon(\bar{\tau}) \,|\, \text{Num}(\tau_{\text{num}}) \\
\tau_{\text{num}} & := & v_{\text{num}} \,|\, k\,\tau_{\text{num}} \,|\, \tau_{\text{num}} + \tau_{\text{num}} \,|\, k \\
k & := & 0 \,|\, 1 \,|\, -1 \,|\, 2 \,|\, -2 \,|\, \ldots \\
\tau & := & \tau_{\text{type}} \,|\, \tau_{\text{num}} \\
a_{\text{type}} & := & \tau_{\text{type}} \doteq \tau_{\text{type}} \\
a_{\text{num}} & := & \tau_{\text{num}} \doteq \tau_{\text{num}} \,|\, \tau_{\text{num}} \leq \tau_{\text{num}} \\
 & | & v_{\text{num}} \doteq \max(\tau_{\text{num}}, \tau_{\text{num}}) \,|\, v_{\text{num}} \doteq \min(\tau_{\text{num}}, \tau_{\text{num}}) \\
 & | & v_{\text{num}} \leq \max(\tau_{\text{num}}, \tau_{\text{num}}) \,|\, \min(\tau_{\text{num}}, \tau_{\text{num}}) \leq v_{\text{num}} \\
a & := & a_{\text{type}} \,|\, a_{\text{num}} | \chi(\tau_{\text{type}}) | \chi_K(\tau_{\text{type}}, \tau_{\text{type}}) \\
v & := & v_{\text{type}} \,|\, v_{\text{num}} = \alpha_i \,|\, \beta_i \,|\, \ldots \\
\sigma & := & \forall v^1_{\text{type}}[\exists \bar{v}.\bar{a}].v^1_{\text{type}} \,|\, \forall \bar{v}[\bar{a}].\tau_{\text{type}}
\end{array}
$$

Table 3.1. Abstract syntax of types and type schemes

any expression. We define *solved form formulas* to be existentially quantified conjunctions of atoms $\exists\bar{\alpha}.A$ without predicate variables. An interpretation of predicate variables $\mathcal{I}$ substitutes predicate variables by solved form formulas.

In Table 3.1, we present a particular instance of $\mathcal{L}_1$, introducing a numerical constraint domain. The terms of $\mathcal{L}_1$ are τ, and a are the atomic formulas, as currently implemented in INVARGENT. There are two sorts: s_{type} with terms τ_{type} and relations a_{type}, and s_{num} with terms τ_{num} and relations a_{num}. Variables of distinct sorts are disjoint; v_s stands for variables of sort s. The remaining entry σ is built on top of $\mathcal{L}_1$ rather than part of it. The notation v^1_{type} stands for the same variable of sort s_{type} in both occurrences. Besides τ, we also use the letter t for terms of $\mathcal{L}_1$. Rarely, in interests of readability we use the letters x, y, z besides α, β to denote variables of $\mathcal{L}_1$.

3.2. THE TYPE SYSTEM WITH GADTS

By *types* τ we mean terms of sort s_{type}. 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_{type}. 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 $\beta \notin \text{FV}(\tau)$. 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 $\chi(\beta)$. $\forall\beta\bar{\alpha}[b \doteq \tau \wedge E].\tau$ is a notational variant of $\forall\beta[\exists\bar{\alpha}.E].\beta$. 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.

Table 3.2 presents expressions currently in INVARGENT, more domain-specific assert and when clauses can be added. The table defines several expression languages, underlying the corresponding type systems: MMG(X), its supersets – in terms of expressions e and system rules – MMG(num) and MMG$_\exists(X)$, and MMG$_\exists$(num) which is a union of MMG(num) and MMG$_\exists(X)$. The domain-independent languages MMG(X) and MMG$_\exists(X)$ are extended

$$
\begin{array}{l}
p := x \mid 0 \mid 1 \mid p \wedge p \mid K\,\bar{p} \\
c := \\
\quad \mathrm{MMG}(X): \\
\qquad p.e \\
\quad \mathrm{MMG}(\mathrm{num}): \\
\qquad \mid p\ \mathbf{when}\ \overline{e \leqslant e}.e \\
e := \\
\quad \mathrm{MMG}(X): \\
\qquad x \mid K\,\bar{e} \mid \mathbf{let}\ p = e\ \mathbf{in}\ e \mid e\,e \mid \lambda(\bar{c}) \mid \mathbf{let\ rec}\ x = e\ \mathbf{in}\ e \\
\qquad \mid \mathbf{assert\ false} \\
\qquad \mid \mathbf{assert\ type}\ e = e; e \\
\qquad \mid \mathbf{runtime\ failure}\ e \\
\quad \mathrm{MMG}(\mathrm{num}): \\
\qquad \mid \mathbf{assert\ num}\ e \leqslant e; e \\
\quad \mathrm{MMG}_{\exists}(X): \\
\qquad \mid \lambda[K]\bar{c} \\
\qquad \mid \lambda_K \bar{c}
\end{array}
$$

Table 3.2. Abstract syntax of expressions

Syntax-directed:

$$
\begin{array}{lll}
\textsc{p-Empty} & \textsc{p-Wild} & \textsc{p-Var} \\
C \vdash 0\colon \tau \longrightarrow \exists \varnothing[\boldsymbol{F}]\{\} & C \vdash 1\colon \tau \longrightarrow \exists \varnothing[\boldsymbol{T}]\{\} & C \vdash x\colon \tau \longrightarrow \exists \varnothing[\boldsymbol{T}]\{x \mapsto \tau\}
\end{array}
$$

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

$$
\textsc{p-And}\quad \frac{\forall i \quad C \vdash p_i\colon \tau \longrightarrow \Delta_i}{C \vdash p_1 \wedge p_2\colon \tau \longrightarrow \Delta_1 \times \Delta_2}
$$

Non-syntax-directed:

$$
\textsc{p-EqIn}\quad \frac{C \vdash p\colon \tau' \longrightarrow \Delta \quad C \vDash \tau \doteq \tau'}{C \vdash p\colon \tau \longrightarrow \Delta}
\qquad
\textsc{p-SubOut}\quad \frac{C \vdash p\colon \tau \longrightarrow \Delta' \quad C \vDash \Delta' \leqslant \Delta}{C \vdash p\colon \tau \longrightarrow \Delta}
\qquad
\textsc{p-Hide}\quad \frac{C \vdash p\colon \tau \longrightarrow \Delta \quad \bar{\alpha} \# \mathrm{FV}(\tau, \Delta)}{\exists \bar{\alpha}.C \vdash p\colon \tau \longrightarrow \Delta}
$$

Table 3.3. Typing rules for $\mathrm{MMG}(X)$: patterns

with some expressions specific to the numerical sort in MMG(num) and $\mathrm{MMG}_{\exists}$(num). The language $\mathcal{L}_1$ and model $\mathcal{M}$ for $\mathrm{MMG}(X)$ and $\mathrm{MMG}_{\exists}(X)$ are arbitrary. We fix the language $\mathcal{L}_1$ for MMG(num) and $\mathrm{MMG}_{\exists}$(num) as having besides s_{type} only a linear arithmetic sort. We discuss the type system MMG(num) shortly, and extend it to $\mathrm{MMG}_{\exists}$(num) in Section 3.4. Disjunctive patterns are not yet implemented in InvarGenT, we omit them in the presentation for brevity.

First, we present the type system in the standard, natural deduction style. The *type judgment* $C, \Gamma \vdash e\colon \tau$ or $C, \Gamma \vdash e\colon \sigma$ 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

Syntax-directed:

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

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

APP
$$\frac{C, \Gamma \vdash e_1 : \tau' \to \tau \qquad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

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

LETIN
$$\frac{C, \Gamma \vdash \lambda(p.e_2)\, e_1 : \tau}{C, \Gamma \vdash \textbf{let}\ p = e_1\ \textbf{in}\ e_2 : \tau}$$

LETREC
$$\frac{C, \Gamma' \vdash e_1 : \sigma \qquad C, \Gamma' \vdash e_2 : \tau \qquad \sigma = \forall\beta[\exists\bar{\alpha}.D].\beta \quad \Gamma' = \Gamma\{x \mapsto \sigma\}}{C, \Gamma \vdash \textbf{let rec}\ x = e_1\ \textbf{in}\ e_2 : \tau}$$

ASSERTLEQ
$$\frac{C, \Gamma \vdash e_1 : \mathrm{Num}(\tau_1) \quad C \vDash \tau_1 \le \tau_2 \qquad C, \Gamma \vdash e_2 : \mathrm{Num}(\tau_2) \quad C, \Gamma \vdash e_3 : \tau}{C, \Gamma \vdash \textbf{assert num}\ e_1 \leqslant e_2; e_3 : \tau}$$

ASSERTEQTY
$$\frac{C, \Gamma \vdash e_1 : \tau_1 \quad C \vDash \tau_1 \doteq \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_2 \quad C, \Gamma \vdash e_3 : \tau}{C, \Gamma \vdash \textbf{assert type}\ e_1 = e_2; e_3 : \tau}$$

ASSERTFALSE
$$\frac{C \vDash \boldsymbol{F}}{C, \Gamma \vdash \textbf{assert false} : \tau}$$

RUNTIMEFAILURE
$$\frac{C, \Gamma \vdash s : \mathrm{String}}{C, \Gamma \vdash \textbf{runtime failure}\ s : \tau}$$

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

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

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

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

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

Table 3.4. Typing rules for MMG(num): expressions

constructors are considered, we make them explicit by writing $C, \Gamma, \Sigma \vdash e : \tau$. By $\mathcal{I}$ we denote substitutions of predicate variables χ. By interpretations we mean substitutions leading to ground formulas, which have truth value in the model. The intended meaning of the type judgment $C, \Gamma, \Sigma \vdash e : \tau$ is: for every interpretation $\mathcal{I}, R$, if $\mathcal{I}, R \vDash C$, then the expression e has a ground type $R(\tau)$ in a ground environment $R(\mathcal{I}(\Gamma))$; and with constructors $\mathcal{I}(\Sigma)$ but this only becomes relevant starting from Section 3.4. We define derivability of type judgments in Tables 3.4 and 3.5, where we use pattern-related derivations from Table 3.3. For example, the rule VAR: $\frac{\Gamma(x) = \forall\beta[\exists\bar{\alpha}.D].\beta \quad C \vDash D}{C, \Gamma \vdash x : \beta}$ means that we can derive a type β for x, with properties

Clauses

Clause

$$\frac{\begin{array}{ll} C\wedge D,\Gamma\Gamma'\vdash m_i\colon \mathrm{Num}(\tau^{m_i}) & e\neq \text{runtime failure}\wedge e\neq \textbf{assert false}\wedge\ldots\wedge \\ C\wedge D,\Gamma\Gamma'\vdash n_i\colon \mathrm{Num}(\tau^{n_i}) & e\neq\lambda(p'\ldots\lambda(p''.\textbf{assert false})) \\ C\vdash p\colon\tau_1\longrightarrow\exists\bar\beta[D]\Gamma'\ \bar\beta\#\mathrm{FV}(C,\Gamma,\tau_2) & C\wedge D\wedge_i\tau^{m_i}\leq\tau^{n_i},\Gamma\Gamma'\vdash e\colon\tau_2\end{array}}{C,\Gamma\vdash p\,\textbf{when}\wedge_i m_i\leqslant n_i.e\colon\tau_1\to\tau_2}$$

NegClause

$$\frac{\begin{array}{ll} C\wedge D,\Gamma\Gamma'\vdash m_i\colon \mathrm{Num}(\tau^{m_i}) & e=\textbf{assert false}\vee\ldots\vee \\ C\wedge D,\Gamma\Gamma'\vdash n_i\colon \mathrm{Num}(\tau^{n_i}) & e=\lambda(p'\ldots\lambda(p''.\textbf{assert false})) \\ C\vdash p\colon\tau_3\longrightarrow\exists\bar\beta[D]\Gamma'\ \bar\beta\#\mathrm{FV}(C,\Gamma,\tau_2) & C\wedge D\wedge\tau_1\doteq\tau_3\wedge_i\tau^{m_i}\leq\tau^{n_i},\Gamma\Gamma'\vdash e\colon\tau_2\end{array}}{C,\Gamma\vdash p\,\textbf{when}\wedge_i m_i\leqslant n_i.e\colon\tau_1\to\tau_2}$$

FailClause

$$\frac{\begin{array}{ll} C\wedge D,\Gamma\Gamma'\vdash m_i\colon \mathrm{Num}(\tau^{m_i}) & C\vdash p\colon\tau_3\longrightarrow\exists\bar\beta[D]\Gamma'\ \bar\beta\#\mathrm{FV}(C,\Gamma) \\ C\wedge D,\Gamma\Gamma'\vdash n_i\colon \mathrm{Num}(\tau^{n_i}) & C\wedge D\wedge\tau_1\doteq\tau_3\wedge_i\tau^{m_i}\leq\tau^{n_i},\Gamma\Gamma'\vdash s\colon\mathrm{String}\end{array}}{C,\Gamma\vdash p\,\textbf{when}\wedge_i m_i\leqslant n_i.\textbf{runtime failure}\,s\colon\tau_1\to\tau_2}$$

Table 3.5. Typing rules for MMG(num): clauses

D, if the properties D of β are implied by the judgement constraint C. And the standard rule App: $\frac{C,\Gamma\vdash e_1\colon\tau'\to\tau\ \ C,\Gamma\vdash e_2\colon\tau'}{C,\Gamma\vdash e_1\,e_2\colon\tau}$ means that a type for an application $e_1\,e_2$ can be derived if a function type can be derived for e_1 and the corresponding argument type can be derived for e_2. The type String is a datatype; we denote datatypes in general by the label ε.

Top-level definitions introduce new entries into a global value constructor environment Σ or a global variable environment Γ. Here we present the most interesting case of a recursive definition with a test clause. The type information flow from the test e_2 to the definition e_1 is through the shared type scheme σ in Γ'. Other cases are analogous. In particular, toplevel `let` definitions, unlike `let`...`in` expressions, are polymorphic, and, like `let`...`in` expressions, allow binding to a pattern.

$$\frac{\begin{array}{ll} C,\Gamma'\vdash e_1\colon\sigma & C,\Gamma'\vdash e_2\colon\mathrm{Bool} \\ \sigma=\forall\beta[\exists\bar\alpha.D].\beta & \Gamma'=\Gamma\{x\mapsto\sigma\}\quad\mathcal{M}\vDash C\end{array}}{C,\Gamma\vdash\textbf{let rec}\,x=e_1\,\textbf{test}\,e_2\longrightarrow\Gamma'}$$

A *data constructor* K for a *datatype* ε (recall that the sort s_{type} holds two categories of elements: datatypes and function types) has definition $K::\forall\bar\alpha\bar\beta[D].\tau_1\times\ldots\times\tau_n\to\varepsilon(\bar\alpha)$ where $\mathrm{FV}(D,\tau_1,\ldots,\tau_n)\subseteq\bar\alpha\bar\beta$. D is a conjunction of atoms.

At this point the construction LetIn is a syntactic sugar for single branch patterns – if polymorphic `let` is needed, use LetRec. We identify clauses $p.e$ of MMG(X) with $p\,\textbf{when}\,.e$ where the type system rules are like the rules for $p\,\textbf{when}\wedge_i m_i\leqslant n_i$, only with the subformula $\wedge_i\tau^{m_i}\leq\tau^{n_i}$ removed (i.e. replaced by $\boldsymbol{T}$). Note that Gen and DisjElim are unrelated to Constraint Generalization we introduce in the next chapter. Most of the rules in Tables 3.3 and 3.4 are close to HMG(X) rules well described in Simonet and Pottier

[47]. A salient difference is the rule NEGCLAUSE for clauses whose right-hand-side is an `assert false` expression. Rather than unifying the type of the function argument with the type of the pattern, we want the discrepancy between these types be a possible reason of unreachability of the pattern matching clause. Without the distinct treatment of clauses with `assert false` as right-hand-side, the `assert`-based variant of the `equal` example would fail. The notation using ... in the rules CLAUSE and NEGCLAUSE represents a check whether an expression is a cascade of function abstractions with a single case (i.e. a single pattern matching branch), the last function abstraction having a function body `assert false`. The similar rule FAILCLAUSE is an "escape hatch" for cases which cannot be ruled out by the type system.

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.

3.3. TYPE INFERENCE CONSTRAINTS

In Tables 3.6, 3.7 and 3.8, we present type judgments declaratively by reducing them to constraints. The presentation is a little bit heavy due to explicit capture-avoidance conditions. For example, $[\![\Gamma \vdash x : \tau]\!] = \exists \beta' \bar{\alpha}'. D[\beta\bar{\alpha} := \beta'\bar{\alpha}'] \wedge \beta' \doteq \tau$ where $\Gamma(x) = \forall \beta[\exists \bar{\alpha}. D].\beta$, $\beta'\bar{\alpha}' \# \mathrm{FV}(\Gamma, \tau)$ introduces a constraint over the type τ required by the properties of x stored in the environment Γ. The constraint $[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \exists \alpha. [\![\Gamma \vdash e_1 : \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!],\ \alpha \# \mathrm{FV}(\Gamma, \tau)$ for typing the result of application of e_1 to e_2, imposes a function type on e_1, and a type on e_2 – represented by the fresh variable α – that matches the argument type of e_1. The constraint for a pattern matching clause without the `when` guard becomes more readable: $[\![\Gamma \vdash p.e : \tau_1 \to \tau_2]\!] = [\![\vdash p \downarrow \tau_1]\!] \wedge \forall \bar{\beta}. D \Rightarrow [\![\Gamma\Gamma' \vdash e : \tau_2]\!]$ where $\exists \bar{\beta}[D]\Gamma' = [\![\vdash p \uparrow \alpha_3]\!]$. The premise D derived from the pattern p provides context-specific information for typing the expression e.

The two presentations are equivalent, in the sense of the *correctness* and *completeness* theorems. The proofs are in Section A.1.2.

THEOREM 3.1. Correctness *(expressions). For all environments* Γ*, expressions* e *and types* τ*,* $[\![\Gamma \vdash e : \tau]\!], \Gamma \vdash e : \tau$ *is derivable.*

THEOREM 3.2. Completeness *(expressions). Let* $\mathcal{L}_1$ *be a first order language with equality* $\doteq$ *and a model* $\mathcal{M}$*. Let* $\mathcal{L}$ *be an extension of* $\mathcal{L}_1$ *with predicate variables* $\chi(\cdot)$*. Let* Γ *be an environment,* C *a formula in* $\mathcal{L}$*,* e *an expression and* τ *a type. If* $\mathrm{PV}(C, \Gamma) = \varnothing$ *and* $C, \Gamma \vdash e : \tau$ *is derivable, then there exists an interpretation of predicate variables* $\mathcal{I}$ *such that* $\mathcal{M}, \mathcal{I} \vDash C \Rightarrow [\![\Gamma \vdash e : \tau]\!]$*.*

In Theorem 3.2, we explicitly introduce all symbols used. We often state propositions

Constraint generation:

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

Environment fragment generation:

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

Table 3.6. Type inference for patterns

and theorems in a more compact manner, with symbols introduced implicitly.

Corollary 3.3. *If* $C, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau$ *and* $\bar{\alpha} \# \mathrm{FV}(\Gamma)$*, then there is an interpretation* $\mathcal{I}$ *such that* $\mathcal{M}, \mathcal{I} \vDash C \Rightarrow (\forall \bar{\alpha}.D \Rightarrow [\![\Gamma \vdash e : \tau]\!])$.

$$
\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 \quad \text{where } \Gamma(x) = \forall \beta[\exists \bar{\alpha}.D].\beta, \\
 & & \beta'\bar{\alpha}' \# \mathrm{FV}(\Gamma, \tau) \\
[\![\Gamma \vdash \textbf{assert false}\colon \tau]\!] & = & \boldsymbol{F} \\
[\![\Gamma \vdash \textbf{runtime failure}\, s\colon \tau]\!] & = & [\![\Gamma \vdash s\colon \mathrm{String}]\!] \\
[\![\Gamma \vdash \textbf{assert num}\, e_1 \leqslant e_2; e_3\colon \tau]\!] & = & \exists \alpha^1 \alpha^2.[\![\Gamma \vdash e_1\colon \mathrm{Num}(\alpha^1)]\!] \wedge [\![\Gamma \vdash e_2\colon \mathrm{Num}(\alpha^2)]\!] \wedge \alpha^1 \leq \alpha^2 \wedge \\
 & & [\![\Gamma \vdash e_3\colon \tau]\!], \alpha_1\alpha_2 \# \mathrm{FV}(\Gamma, \tau) \\
[\![\Gamma \vdash \textbf{assert type}\, e_1 = e_2; e_3\colon \tau]\!] & = & \exists \alpha^1 \alpha^2.[\![\Gamma \vdash e_1\colon \alpha^1]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha^2]\!] \wedge \alpha^1 \doteq \alpha^2 \wedge [\![\Gamma \vdash e_3\colon \tau]\!], \\
 & & \alpha_1\alpha_2 \# \mathrm{FV}(\Gamma, \tau) \\
[\![\Gamma \vdash \lambda \bar{c}\colon \tau]\!] & = & \exists \alpha_1\alpha_2.[\![\Gamma \vdash \bar{c}\colon \alpha_1 \to \alpha_2]\!] \wedge \alpha_1 \to \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 \to \tau]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha]\!], \quad \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 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 \to \varepsilon(\bar{\alpha}), \quad \bar{\alpha}'\bar{\beta}' \# \mathrm{FV}(\Gamma, \tau) \\
[\![\Gamma \vdash \textbf{let rec}\, x = e_1 \textbf{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 \bar{c_i}\colon \tau_1 \to \tau_2]\!] & = & \wedge_i [\![\Gamma \vdash c_i\colon \tau_1 \to \tau_2]\!] \\
[\![\Gamma \vdash e\colon \forall \bar{\alpha}[D].\tau]\!] & = & \forall \bar{\alpha}'.D[\bar{\alpha} := \bar{\alpha}'] \Rightarrow [\![\Gamma \vdash e\colon \tau[\bar{\alpha} := \bar{\alpha}']]\!], \quad \bar{\alpha}' \# \mathrm{FV}(\Gamma)
\end{array}
$$

Table 3.7. Type inference for expressions

$$
\begin{array}{rcl}
[\![\Gamma \vdash p\,\mathbf{when}\,\wedge_i m_i \leqslant n_i.e\colon \tau_1 \to \tau_2]\!] & = & [\![\vdash p{\downarrow}\tau_1]\!] \wedge \forall \bar{\beta}.\exists \overline{\alpha_i^1 \alpha_i^2}.D \Rightarrow \\
& & \quad \wedge_i [\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!] \\
\text{when } e \neq \text{runtime failure} \wedge & & \quad \wedge(\wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!]) \\
e \neq \mathbf{assert\ false} \wedge \ldots \wedge & & \\
e \neq \lambda(p' \ldots \lambda(p''.\mathbf{assert\ false})) & & \\
& & \text{where } \exists \bar{\beta}[D]\Gamma' \text{ is } [\![\vdash p{\uparrow}\tau_1]\!], \bar{\beta}\overline{\alpha_i^1 \alpha_i^2} \# \mathrm{FV}(\Gamma, \tau_1, \tau_2) \\
[\![\Gamma \vdash p\,\mathbf{when}\,\wedge_i m_i \leqslant n_i.e\colon \tau_1 \to \tau_2]\!] & = & \exists \alpha_3.[\![\vdash p{\downarrow}\alpha_3]\!] \wedge \forall \bar{\beta}.\exists \overline{\alpha_i^1 \alpha_i^2}.D \Rightarrow \\
& & \quad \wedge_i [\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!] \\
\text{when } e = \mathbf{assert\ false} \vee \ldots \vee & & \quad \wedge(\alpha_3 \doteq \tau_1 \wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!]) \\
e = \lambda(p' \ldots \lambda(p''.\mathbf{assert\ false})) & & \\
& & \text{where } \exists \bar{\beta}[D]\Gamma' \text{ is } [\![\vdash p{\uparrow}\alpha_3]\!], \bar{\beta}\alpha_3\overline{\alpha_i^1 \alpha_i^2} \# \mathrm{FV}(\Gamma, \tau_1, \tau_2) \\
[\![\Gamma \vdash p\,\mathbf{when}\,\wedge_i m_i \leqslant n_i.e\colon \tau_1 \to \tau_2]\!] & = & \exists \alpha_3.[\![\vdash p{\downarrow}\alpha_3]\!] \wedge \forall \bar{\beta}.\exists \overline{\alpha_i^1 \alpha_i^2}.D \Rightarrow \\
& & \quad \wedge_i [\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!] \\
\text{when } e = \mathbf{runtime\ failure}\, s & & \quad \wedge(\alpha_3 \doteq \tau_1 \wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma' \vdash s\colon \mathrm{String}]\!]) \\
& & \text{where } \exists \bar{\beta}[D]\Gamma' \text{ is } [\![\vdash p{\uparrow}\alpha_3]\!], \bar{\beta}\alpha_3\overline{\alpha_i^1 \alpha_i^2} \# \mathrm{FV}(\Gamma, \tau_1, \tau_2)
\end{array}
$$

Table 3.8. Type inference for clauses

$$
\begin{array}{rcl}
l(x) & = & \boldsymbol{F} \\
l(\lambda \bar{c}) & = & \boldsymbol{F} \\
l(e_1\, e_2) & = & l(e_1) \\
l(K e_1 \ldots e_n) & = & \boldsymbol{F} \\
l(\mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2) & = & l(e_2) \\
l(\lambda[K]\overline{p_i.e_i}) & = & \boldsymbol{T} \\
l(\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2) & = & \boldsymbol{T} \\
l(\mathbf{assert\ num}\, e_1 \leqslant e_2; e_3) & = & l(e_3) \\
l(\mathbf{assert\ type}\, e_1 = e_2; e_3) & = & l(e_3) \\
l(\mathbf{runtime\ failure}\, s) & = & \boldsymbol{T} \\
l(\mathbf{assert\ false}) & = & \boldsymbol{T}
\end{array}
$$

Table 3.9. Expressions that directly handle an existential type

$$
\begin{array}{rcl}
\mathcal{E}(x, v) & = & \varnothing \\
\mathcal{E}(\lambda \bar{c}, v) & = & \cup \overline{\mathcal{E}(c, \boldsymbol{F})} \\
\mathcal{E}(e_1\, e_2, v) & = & \mathcal{E}(e_1, v) \cup \mathcal{E}(e_2, \boldsymbol{F}) \\
\mathcal{E}(K e_1 \ldots e_n, v) & = & \cup_i \mathcal{E}(e_i, \boldsymbol{F}) \\
\mathcal{E}(\textbf{let rec}\, x = e_1\, \textbf{in}\, e_2, v) & = & \mathcal{E}(e_1, \boldsymbol{F}) \cup \mathcal{E}(e_2, v) \\
\mathcal{E}(p\, \textbf{when}\, \overline{m_i \leqslant n_i}.e, v) & = & \cup_i \mathcal{E}(m_i, \boldsymbol{F}) \cup_i \mathcal{E}(n_i, \boldsymbol{F}) \\
 & & \cup \mathcal{E}(e, v) \\
\mathcal{E}(\lambda [K] \bar{c}, \boldsymbol{F}) & = & \{K\} \cup \overline{\mathcal{E}(c, \boldsymbol{T})} \\
\mathcal{E}(\lambda [K] \bar{c}, \boldsymbol{T}) & = & \cup \overline{\mathcal{E}(c, \boldsymbol{T})} \\
\mathcal{E}(\textbf{let}\, p = e_1\, \textbf{in}\, e_2, v) & = & \mathcal{E}(e_1, \boldsymbol{F}) \cup \mathcal{E}(e_2, v) \\
\mathcal{E}(\textbf{assert num}\, e_1 \leqslant e_2; e_3, K') & = & \mathcal{E}(e_1, \boldsymbol{F}) \cup \mathcal{E}(e_2, \boldsymbol{F}) \cup \mathcal{E}(e_3, v) \\
\mathcal{E}(\textbf{assert type}\, e_1 = e_2; e_3, K') & = & \mathcal{E}(e_1, \boldsymbol{F}) \cup \mathcal{E}(e_2, \boldsymbol{F}) \cup \mathcal{E}(e_3, v)
\end{array}
$$

Table 3.10. Collect introduced value constructors

In Definition 3.5, we describe a class of type judgments that is a better target for type inference. First, we introduce a normal form of constraints that simplifies formal considerations.

PROPOSITION 3.4. *For any Γ, e, τ, the formula $[\![\Gamma \vdash e\colon \tau]\!]$ is equivalent to a prenex-normal, normalized form: there is a quantifier prefix $\mathcal{Q}$ and pairs of conjunctions of atoms D_i, C_i, such that*

$$\mathcal{M} \vDash [\![\Gamma \vdash e\colon \tau]\!] \Leftrightarrow \mathcal{Q}.\, \wedge_i (D_i \Rightarrow C_i)$$

We put $\mathrm{NF}([\![\Gamma \vdash e\colon \tau]\!]) = \mathcal{Q}.\, \wedge_i (D_i \Rightarrow C_i)$.

DEFINITION 3.5. *We call the formula $[\![\Gamma \vdash e\colon \tau]\!]$ the* type inference problem *for environment Γ, expression e and type τ. Let $\mathcal{M} \vDash [\![\Gamma \vdash e\colon \tau]\!] \Leftrightarrow \mathcal{Q}.\Phi_N$ with $\Phi_N = \wedge_i (D_i \Rightarrow C_i)$ as in Proposition 3.4. Let F_{res} be a conjunction of atoms without predicate variables. We call $\exists \bar{\alpha}_{\mathrm{res}}.F_{\mathrm{res}}, \mathcal{I}$ a* solution to the type inference problem $[\![\Gamma \vdash e\colon \tau]\!]$ *when $\mathcal{I}, \mathcal{M} \vDash F_{\mathrm{res}} \Rightarrow \Phi_N$ (i.e. $\mathcal{M} \vDash F_{\mathrm{res}} \Rightarrow \mathcal{I}(\Phi_N)$), $\mathcal{M} \vDash \mathcal{Q}.F_{\mathrm{res}}[\bar{\alpha}_{\mathrm{res}} := \bar{t}]$ for some $\bar{t}$, and for every implication in Φ_N, if $\mathcal{M}$, $\mathcal{I} \vDash \exists \mathrm{FV}(D_i).D_i$ then $\mathcal{M}, \mathcal{I} \vDash \exists \mathrm{FV}(D_i, F_{\mathrm{res}}).D_i \wedge F_{\mathrm{res}}$.*

We are more interested in finding an unambiguous, directly given solution to a type inference problem, as introduced in Definition 3.5, than in derivability of a judgement C, $\Gamma \vdash e\colon \tau$ for some satisfiable C. By Theorems 3.1 and 3.2, these notions are close. Therefore, unlike in $\mathrm{HMG}(X)$, we say that an expression e is *well-typed* in $\mathrm{MMG}(X)$ under environment Γ with type τ, when the type inference problem $[\![\Gamma \vdash e\colon \tau]\!]$ has a solution. The class of programs admitting a solution to the type inference problem can sometimes be expanded by enriching the languages of constraint domains.

Note that both the derivability of $C, \Gamma \vdash e\colon \tau$ and the existence of a solution to the type inference problem $[\![\Gamma \vdash e\colon \tau]\!]$ allows for dead code in e, i.e. unreachable cases of pattern matching. We offer an option to reject programs with dead code, according to the following

definition: We call $\exists\bar{\alpha}_{\text{res}}.F_{\text{res}}, \mathcal{I}$ a *solution without dead code to the type inference problem* $[\![\Gamma \vdash e\colon \tau]\!]$ when $\mathcal{I}, \mathcal{M} \vDash F_{\text{res}} \Rightarrow \Phi_N$ (i.e. $\mathcal{M} \vDash F_{\text{res}} \Rightarrow \mathcal{I}(\Phi_N)$), $\mathcal{M} \vDash \mathcal{Q}.F_{\text{res}}[\bar{\alpha}_{\text{res}} := \bar{t}]$ for some $\bar{t}$, and for every implication in Φ_N, if $C_i \neq \boldsymbol{F}$ then $\mathcal{M}, \mathcal{I} \vDash \exists \text{FV}(D_i, F_{\text{res}}).D_i \wedge F_{\text{res}}$. However, some datatype-related dead code cases are rejected regardless of this option, due to constraints introduced by $[\![\vdash p{\downarrow}\tau]\!]$ for patterns p.

3.4. 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 expressiveness 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 (contravariant) argument positions of function types are redundant: they can be lifted to be traditional, polymorphic variables constrained by the invariant of the function. However, they may sometimes be needed in nested positions of higher-order function types. We prohibit the use of inferred existential types in argument positions in the current version of the type system.

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` $\bar{c}$. We also introduce a corresponding construct $\lambda_K\bar{c}$. $\lambda[K]\bar{c}$ is either eliminated or replaced by $\lambda_K\bar{c}$ in a normalization step. When $K :: \forall\bar{\alpha}\bar{\beta}\gamma[E].\gamma \to \varepsilon_K(\bar{\alpha}) \in \Sigma$ 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 E implies $\gamma \doteq \tau_e$. We also parse $\exists\bar{\beta}[E].\tau_e$ generating a fresh $K :: \forall\bar{\alpha}\bar{\beta}[E].\tau_e \to \varepsilon_K(\bar{\alpha}) \in \Sigma$ in the toplevel Σ, where $\bar{\alpha} = \text{FV}(\tau_e) \setminus \bar{\beta}$.

Let $l(e)$ defined in Table 3.9 determine whether an expression introduces or eliminates an existential type. It is used in the normalization process described below.

Let all occurrences of $\lambda[K]$ in e use distinct K. Let $n(e) := n(e, \bot)$, defined in Table 3.11, flatten nested introductions of existential types. Let $\mathcal{E}(e) := \mathcal{E}(e, \boldsymbol{F})$, defined in Table 3.10, collect value constructors introduced for existential types. The normalization-related functions $n(\cdot, v)$ and $\mathcal{E}(\cdot, v)$ are defined for both expressions and clauses.

The $\text{MMG}_\exists(X)$ typing rules that differ from $\text{MMG}(X)$ are provided in Table 3.12. We put the normalization step into the type system as a rule ExIntro. W.l.o.g. ExIntro can be used once at the beginning of a derivation. ExIntro performs a normalization of

$$
\begin{array}{rcl}
n(e, K') & = & \mathbf{let}\, x = n(e, \bot)\, \mathbf{in} K'\, x \\
\text{when } K' \neq \bot \wedge l(e) = \boldsymbol{F} & & \\
n(x, \bot) & = & x \\
n(\lambda \bar{c}, \bot) & = & \lambda(\overline{n(c, \bot)}) \\
n(e_1\, e_2, K') & = & n(e_1, K')\, n(e_2, \bot) \\
n(K e_1 ... e_n, \bot) & = & K n(e_1, \bot) ... n(e_n, \bot) \\
n(\mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2, K') & = & \mathbf{let\ rec}\, x = n(e_1, \bot)\, \mathbf{in}\, n(e_2, K') \\
n(p\, \mathbf{when}\, \overline{m_i \leqslant n_i}.e, K') & = & p\, \mathbf{when}\, \overline{n(m_i, \bot) \leqslant n(n_i, \bot)}.\mathbf{runtime\ failure}\, n(s, \bot) \\
\text{when } e = \mathbf{runtime\ failure}\, s & & \\
n(p\, \mathbf{when}\, \overline{m_i \leqslant n_i}.e, K') & = & p\, \mathbf{when}\, \overline{n(m_i, \bot) \leqslant n(n_i, \bot)}.n(e, K') \\
\text{when } e \neq \mathbf{runtime\ failure}\, s & & \\
n(\lambda[K]\bar{c}, \bot) & = & \lambda_K(\overline{n(c, K)}) \\
n(\lambda[K]\bar{c}, K') & = & \lambda(\overline{n(c, K')}) \\
\text{when } K' \neq \bot & & \\
n(\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2, K') & = & \mathbf{let}\, p = n(e_1, \bot)\, \mathbf{in}\, n(e_2, K') \\
n(\mathbf{assert\ num}\, e_1 \leqslant e_2; e_3, K') & = & \mathbf{assert\ num}\, n(e_1, \bot) \leqslant n(e_2, \bot); n(e_3, K') \\
n(\mathbf{assert\ type}\, e_1 = e_2; e_3, K') & = & \mathbf{assert\ type}\, n(e_1, \bot) = n(e_2, \bot); n(e_3, K')
\end{array}
$$

Table 3.11. Flatten nested introductions of existential types

the expression and shares the job of introducing existential types with the rule ExAbs. The RetType relation can be introduced into the model $\mathcal{M}$ by induction on the structure of the first argument: $\text{RetType}(\tau_1 \to \tau_2,\ \tau_3) \Leftrightarrow \text{RetType}(\tau_2,\ \tau_3)$ and $\text{RetType}(\varepsilon(\bar{\tau}),\ \tau') \Leftrightarrow \varepsilon(\bar{\tau}) \doteq \tau'$. ExLetIn is the elimination rule for existential types. Although LetIn and ExLetIn resemble "syntactic sugar", their application is non-deterministic. We include value constructor environment in judgments to facilitate the completeness proof. We modify the rule App to exclude function arguments that have existential types. We achieve that by introducing a new atomic predicate $\nexists$ to the sort s_{type}: $\nexists(\tau)$ iff $\wedge_K \neg \exists \bar{\alpha}.\tau \doteq \varepsilon_K(\bar{\alpha})$, i.e. τ is not an existential type.

Definition 3.6. *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:*

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

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

Note that we do not lose generality by using single-argument datatypes $\varepsilon_K(\alpha)$ rather than the general form $\varepsilon_K(\bar{\alpha})$. Multiple parameters can be captured by providing a constraint $\exists \alpha_1 ... \alpha_n.\alpha \doteq \alpha_1 \to ... \to \alpha_n$, with existential variables $\alpha_1 ... \alpha_n$, as part of the constraints of constructors of $\varepsilon_K(\alpha)$. The single parameter τ' in condition $C \vDash \text{RetType}(\tau, \varepsilon_K(\tau'))$ of the rule ExAbs is not consequential. In the implementation of InvarGenT, we recover the multiple parameter representation $\varepsilon_K(\bar{\alpha})$ of existential types, at the end of inference.

$$\textsc{App}\quad \frac{C,\Gamma,\Sigma\vdash e_1\colon \tau'\to\tau \quad C,\Gamma,\Sigma\vdash e_2\colon \tau' \quad C\vDash \not{E}(\tau')}{C,\Gamma,\Sigma\vdash e_1\,e_2\colon\tau} \qquad \textsc{ExLetIn}\quad \frac{\varepsilon_K(\bar\alpha)\text{ in }\Sigma \quad C,\Gamma,\Sigma\vdash e_1\colon\tau' \quad C,\Gamma,\Sigma\vdash Kp.e_2\colon\tau'\to\tau}{C,\Gamma,\Sigma\vdash \mathbf{let}\,p=e_1\,\mathbf{in}\,e_2\colon\tau}$$

$$\textsc{ExIntro}\quad \frac{\mathrm{Dom}(\Sigma')\backslash\mathrm{Dom}(\Sigma)=\mathcal{E}(e) \quad C,\Gamma,\Sigma'\vdash n(e)\colon\tau}{C,\Gamma,\Sigma\vdash e\colon\tau} \qquad \textsc{ExAbs}\quad \frac{C\vDash \mathrm{RetType}(\tau,\varepsilon_K(\tau')) \quad C,\Gamma\vdash\lambda\bar c\colon\tau}{C,\Gamma\vdash\lambda_K\bar c\colon\tau}$$

Table 3.12. Typing rules added by $\mathrm{MMG}_\exists(X)$

$$\begin{aligned}
[\![\Gamma\vdash e_1\,e_2\colon\tau]\!] &= \exists\alpha.[\![\Gamma\vdash e_1\colon\alpha\to\tau]\!]\wedge[\![\Gamma\vdash e_2\colon\alpha]\!]\wedge\not{E}(\alpha),\alpha\#\mathrm{FV}(\Gamma,\tau)\\
[\![\Gamma,\Sigma_0\vdash e\colon\tau]\!] &= [\![\Gamma,\Sigma\vdash n(e)\colon\tau]\!]\\
\text{when } \mathcal{E}(e)\neq\varnothing &\quad \text{where } \Sigma=\Sigma_0\overline{K::\forall\alpha_K\gamma_K[\chi_K(\gamma_K,\alpha_K)].\gamma_K\to\varepsilon_K(\alpha_K)}_{K\in\mathcal{E}(e)}\\
[\![\Gamma\vdash\mathbf{let}\,p=e_1\,\mathbf{in}\,e_2\colon\tau]\!] &= \exists\alpha_0.[\![\Gamma\vdash e_1\colon\alpha_0]\!]\wedge([\![\Gamma\vdash p.e_2\colon\alpha_0\to\tau]\!]\wedge\not{E}(\alpha_0)\\
&\qquad \vee_{\mathcal{E}}[\![\Gamma\vdash Kp.e_2\colon\alpha_0\to\tau]\!])\\
&\quad \text{where } \mathcal{E}=\{K\,|\,K::\forall\bar\alpha\bar\beta[E].\tau\to\varepsilon_K(\bar\alpha)\in\Sigma\}\\
[\![\Gamma\vdash\lambda_K\bar c\colon\tau]\!] &= \exists\alpha_0.\mathrm{RetType}(\tau,\,\alpha_0)\wedge(\exists\alpha_1.\alpha_0\doteq\varepsilon_K(\alpha_1)\wedge\chi_K(\alpha_1))\wedge[\![\Gamma\vdash\lambda\bar c\colon\tau]\!],\\
&\quad \alpha_0\alpha_1\#\mathrm{FV}(\Gamma,\tau)
\end{aligned}$$

Table 3.13. Type inference for the added expressions

Normalization defined in Table 3.11 is responsible for introduction of existential types, but it also ensures that existential types never directly wrap around other existential types. This flattening 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 $K'\neq\bot\wedge l(e)=\boldsymbol{F}$. Note the frequent need to use `efunction` (and its syntax sugar forms `ematch` and `eif`) for all branches of a definition, as for example in function `bsearch2` in Section C.8.4. The branches bound by `match` or `if` would be required to have the exact same type, e.g. the same number `Num`, height of a tree or length of a list.

3.5. Type Inference Constraints for Existential Types

The type inference uses predicate variables to determine the information bundled inside existential types. The constraint derivation rules related to existential types are provided in Table 3.13. For the initial call to $[\![\cdot]\!]$ (i.e. case $\mathcal{E}(e)\neq\varnothing$), we normalize the expression. We shorten $[\![\Gamma,\Sigma\vdash\cdot\colon\tau]\!]$ to $[\![\Gamma\vdash\cdot\colon\tau]\!]$.

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 $[\![\Gamma \vdash p.e_2 \colon \alpha_0 \to \tau]\!]$ 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 the premise $C, \Gamma, \Sigma \nvdash \lambda(p.e_2)\, e_1 \colon \tau$ to the EXLETIN rule, but it would make the formalism more complex. The proof of the following theorem is in Section A.1.3.

THEOREM 3.7. *Theorems 3.1 (Correctness) and 3.2 (Completeness) hold for the type system extended with* EXINTRO *and* EXLETIN *in the following sense. Recall the notation from Theorem 3.2.*

Correctness: For all Γ, Σ, e *and* τ, $[\![\Gamma, \Sigma \vdash e \colon \tau]\!], \Gamma, \Sigma \vdash e \colon \tau$ *is derivable.*

Completeness: For all Γ, Σ, e *and* τ, *if* $\mathrm{PV}(C, \Gamma, \Sigma) = \varnothing$ *and* $C, \Gamma, \Sigma \vdash e \colon \tau$, *then there exist interpretations of predicate variables* $\mathcal{I}_u, \mathcal{I}_e$ *such that* $\mathrm{Dom}(\mathcal{I}_u)$ *are unary,* $\mathrm{Dom}(\mathcal{I}_e) = \{\chi_K | K \in \mathcal{E}(e)\}$, *and* $\mathcal{M}, \mathcal{I}_u \vDash C \Rightarrow \mathcal{I}_e([\![\Gamma, \Sigma \vdash e \colon \tau]\!])[\overline{\varepsilon_K(\vec{\tau})} := \overline{\varepsilon_K(\bar{\tau})}]$.

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 3.6.

Example 3.8. Consider the function `filter` for lists with length. The `eif...then...else` syntax is a syntactic sugar for the `ematch...with True ->... | False ->...` syntax, which in turn is a syntactic sugar for `(efunction True ->... | False ->...)...` expressions.

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons LCons :
  ∀n, a [0⩽n]. a * List(a, n) ⟶ List(a, n+1)

let rec filter = fun f ->
  efunction LNil -> LNil
  | LCons (x, xs) ->
    eif f x
    then let ys = filter f xs in LCons (x,ys)
    else filter f xs
```

As an aside, an example interaction with INVARGENT might look as follows:

```
$ ./invargent -inform examples/filter.gadt
val filter :
  ∀n, a.
  (a → Bool) → List (a, n) → ∃k[0 ⩽ k ∧ k ⩽ n].List (a, k)
InvarGenT: Generated file examples/filter.gadti
InvarGenT: Generated file examples/filter.ml
InvarGenT: Command "ocamlc -w -25 -c examples/filter.ml" exited with code 0
```

Below we show the corresponding type inference constraint, slightly simplified by hand for readability. Constructors such as $\varepsilon_{K_2}(\alpha_6)$ identify an occurrence of an existential type, here $\exists\delta.\chi_1(\delta,\alpha_6)$ in a constructor environment with $K_2 :: \forall\delta\alpha[\chi_1(\delta,\alpha)].\delta \to \varepsilon_{K_2}(\alpha)$.

$$\exists\alpha_0.\forall\beta_0.\exists n_1\alpha_1\alpha_2\alpha_3.\forall k_1\beta_1\beta_2.\exists\alpha_5\alpha_6\alpha_7\alpha_8\alpha_9.\forall\beta_3\beta_4\beta_5\beta_6.\exists n_2\alpha_{10}\alpha_{11}\alpha_{12}\alpha_{13}.$$

$$\chi_2(\alpha_0) \tag{3.1}$$

$$\wedge$$

$$\chi_2(\beta) \implies \alpha_2 \doteq \mathrm{List}(\alpha_4, n_1) \wedge \beta \doteq \alpha_1 \to \mathrm{List}(\alpha_4, n_1) \to \alpha_3 \tag{3.2}$$

$$\wedge$$

$$\mathrm{List}(\beta_1, 0) \doteq \mathrm{List}(\alpha_4, n_1) \wedge \chi_2(\beta) \implies \alpha_3 \doteq \varepsilon_{K_2}(\alpha_6) \wedge \chi_1(\mathrm{List}(\alpha_5, 0), \alpha_6) \tag{3.3}$$

$$\wedge$$

$$\begin{aligned} \mathrm{List}(\beta_2, k_1+1) \doteq \mathrm{List}(\alpha_4, n_1) \wedge \quad &\implies \quad \alpha_1 \doteq \beta_2 \to \mathrm{Bool} \wedge \\ 0 \le k_1 \wedge \chi_2(\beta) \quad &\quad\quad \chi_2(\alpha_1 \to \mathrm{List}(\beta_2, k_1) \to \varepsilon_{K_2}(\alpha_8)) \wedge \\ &\quad\quad \chi_2(\alpha_1 \to \mathrm{List}(\beta_2, k_1) \to \varepsilon_{K_2}(\alpha_9)) \end{aligned} \tag{3.4}$$

$$\wedge$$

$$\begin{aligned} \varepsilon_{K_2}(\beta_4) \doteq \varepsilon_{K_2}(\alpha_8) \wedge \chi_1(\beta_3, \beta_4) \wedge \quad &\implies \quad \alpha_3 \doteq \varepsilon_{K_2}(\alpha_8) \wedge \chi_1(\alpha_{10}, \alpha_{11}) \wedge \beta_3 \doteq \mathrm{List}(\beta_2, n_2) \wedge \\ \mathrm{List}(\beta_2, k_1+1) \doteq \mathrm{List}(\alpha_4, n_1) \wedge \quad &\quad\quad \alpha_{10} \doteq \mathrm{List}(\beta_2, n_2+1) \wedge 0 \le n_2 \\ 0 \le k_1 \wedge \chi_2(\beta) \quad & \end{aligned} \tag{3.5}$$

$$\wedge$$

$$\begin{aligned} \varepsilon_{K_2}(\beta_6) \doteq \varepsilon_{K_2}(\alpha_9) \wedge \chi_1(\beta_5, \beta_6) \wedge \quad &\implies \quad \alpha_{12} \doteq \beta_5 \wedge \alpha_3 \doteq \varepsilon_{K_2}(\alpha_{13}) \wedge \chi_1(\alpha_{12}, \alpha_{13}) \\ \mathrm{List}(\beta_2, k_1+1) \doteq \mathrm{List}(\alpha_4, n_1) \wedge \quad & \\ 0 \le k_1 \wedge \chi_2(\beta) \quad & \end{aligned} \tag{3.6}$$

Branch 3.1 ensures that the invariant is satisfiable. Branch 3.2 decomposes the type of the recursive definition and ensures that the second argument is a list. Branch 3.3 is the case of empty list. Branch 3.4 handles the two recursive calls. Branch 3.5 is the case of kept element: the recursive call result β_3 has length one-shorter than the result α_{10}. Branch 3.6 is the case of a dropped element: the recursive call result β_5 and the function result α_{12} are the same.

3.6. Semantics by Reduction to HMG(X)

The typing rules that importantly differ between HMG(X) and MMG$_\exists$(X) are: NegClause, FailClause, App, LetIn, ExLetIn, ExIntro, ExAbs. The remaining, crucial difference lies in having predicate variables among constraints. However:

- ExIntro, LetIn and ExLetIn can be seen as "desugaring" a program to a core language,
- pattern matching branches that invoke NegClause are unreachable and so can be dropped,
- pattern matching branches that invoke FailClause can be dropped, because they lead to runtime failure, computationally equivalent to a match failure,
- App in MMG$_\exists$(X) is a restricted form of its variant from HMG(X).

Therefore, if an inference problem constraint $[\![\Gamma, \Sigma \vdash e : \tau]\!]$ holds, then we can build environment Γ', constructor environment Σ' and an expression e' such that $C', \Gamma' \vdash e' : \tau$ is derivable in HMG(X) for a satisfiable constraint C'. For the reduction to be a convincing argument for soundness of the type system, we also need to show that the "desugared" expression e' preserves the intended meaning of e. For the details of the HMG(X) type system, we refer the reader to Simonet and Pottier [47].

DEFINITION 3.9. *Consider a constructor environment* Σ *and an* HMG(X) *expression* e. *A* tag erasure *of* e *w.r.t.* Σ *is an expression achieved by iteratively replacing each subpattern* $K p_1$ *of* e *with* $K \notin \mathrm{Dom}(\Sigma)$ *by* p_1 *and each subexpression* $K e_1$ *of* e *with* $K \notin \mathrm{Dom}(\Sigma)$ *by* e_1.

Consider an MMG$_\exists(X)$ *expression* e. *An* HMG-form *of* e *is an* HMG(X) *expression achieved by iteratively replacing each subexpression* $\lambda[K]\bar{c}$ *and* $\lambda_K \bar{c}$ *of* e *by* $\lambda\bar{c}$, *each subexpression* **assert type** $e_1 = e_2; e_3$ *of* e *by* e_3, *each subexpression* **runtime failure** s *by a function call* failwith s, *and each subexpression* **let rec** $x = e_1$ **in** e_2 *in* e *by* **let** $x = \mu x.e_1$ **in** e_2.

The HMG-form from Definition 3.9 captures the intended meaning of programs. We provide semantics for the λ-calculus underlying MMG$_\exists(X)$, by reducing each expression to its HMG-form and referring to the semantics for HMG(X) provided in [47]. In terms of concrete syntax, the HMG-form of a program is the program with `assert type` clauses dropped, keywords `efunction` replaced by `function`, and `ematch` replaced by `match`.

DEFINITION 3.10. *For our purposes, let* computational equivalence *be the smallest congruence relation* $\sim_C$ *such that* **let** $x = e_1$ **in** $e_2 \sim_C (\lambda x.e_2)\, e_1$ *for any expressions* e_1, e_2, *and* $\lambda(\overline{p_i.e_i}) \sim_C \lambda\big(\overline{p_i.e_i}_{i:\neg\mathrm{unreach}(e_i)\wedge\neg\mathrm{failure}(e_i)}\big)$ *for any sequences of clauses* $\overline{p_i.e_i}$. *The condition is defined by:* $\mathrm{unreach}(e) := e =$ **assert false** $\vee (e = \lambda(p_1.e_1) \wedge \mathrm{unreach}(e_1))$ *and* $\mathrm{failure}(e) := e =$ **runtime failure** $s \vee (e = \lambda(p_1.e_1) \wedge \mathrm{failure}(e_1))$.

Expressions **let** $x = e_1$ **in** e_2 and $(\lambda x.e_2)\, e_1$ have the same result of reduction at the outermost context: $e_2[x := e_1]$. Subexpressions **assert false** in a well-typed program are guaranteed to be unreachable in both MMG$_\exists(X)$ and HMG(X) type systems. Subexpressions **runtime failure** s are intended to halt a program, so we assume they are computationally equivalent to match failures (matching against a value not covered by any pattern matching case). The proof of the follwing theorem is in Section A.1.4.

THEOREM 3.11. *Let* $\Gamma, \Sigma \vdash e : \tau$ *be a typing judgment in* MMG$_\exists(X)$. *If* $[\![\Gamma, \Sigma \vdash e : \tau]\!]$ *is satisfiable, we can construct a satisfiable constraint* C', *an* HMG(X) *environment* Γ', *constructor environment* Σ' *and an expression* e' *such that* $C', \Gamma' \vdash e' : \tau$ *is derivable in* HMG(X), *and the tag erasure of* e' *w.r.t.* Σ *is computationally equivalent to the HMG-form of* e *(see Definitions 3.9, 3.10).*

An expression is *closed* when it does not contain variables not bound by a **let**- or **let rec**-expression, or a pattern. A closed expression is *stuck* if it is neither reducible nor a value, as defined in [47].

COROLLARY 3.12. Type soundness. *If a closed expression* e *is well-typed under some constructor environment, then its HMG-form does not reduce to a stuck expression.*

Chapter 4

Solving Second Order Constraints

Least Upper Bounds and Greatest Lower Bounds computations are the standard tools for finding unknowns involved in an order structure, see e.g. Knowles and Flanagan [20] Section 6.3. In case of implicational constraints, constraint abduction (see Maher and Huang [29], Sulzmann, Schrijvers and Stuckey [49]) and constraint generalization belong to this tool-set. *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 $\mathcal{M} \vDash (\exists\bar{\alpha}.A) \Rightarrow (D \Rightarrow C)$, equivalently $\mathcal{M} \vDash (\exists\bar{\alpha}.A) \wedge D \Rightarrow C$, $\mathcal{M} \vDash \exists \mathrm{FV}(A, D, C).A \wedge D$ and $\mathcal{M} \vDash \mathcal{Q}.A[\bar{\alpha} := \bar{t}]$ for some $\bar{t}$. *Joint Constraint Abduction* under Quantifier Prefix handles several implications, i.e. $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$, simultaneously: each condition holds for each D_i, C_i pair. It is used to solve for a predicate variable appearing in multiple premises – where we are interested in the GLB of the corresponding conclusions, "modulo" the corresponding premises. *Constraint Generalization* 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, $\mathcal{M} \vDash D_i \Rightarrow \exists\bar{\alpha}.A$. It is used to solve for a predicate variable appearing in multiple conclusions – where we are interested in the LUB of the corresponding premises.

4.1. Overview of Solving for Predicate Variables

Equipped with these tools, consider first solving an un-normalized constraint Φ for invariants – unary predicates $\chi(\cdot)$. We want the invariants to be as weak as possible, to make the use of the corresponding MMG(X) definitions (toplevel expressions) as easy as possible: the weaker the invariant, the more general the type of the definition. We solve for the predicate variables in multiple steps, maintaining partial solutions $\overline{\exists\bar{\beta}^k_\chi.F^k_\chi}$ and iterating till the solutions converge (see e.g. Cousot and Cousot [10]). We start with $k = 0$, $\bar{\beta}^0_\chi = \varnothing$, $F^0_\chi = \boldsymbol{T}$ for $\chi \in \mathrm{PV}(\Phi)$. In each step, we solve the first order constraint $\mathcal{Q}^k.\wedge_i (D^k_i \Rightarrow C^k_i)$ achieved by reducing $\Phi\Big[\ \bar{\chi} := \overline{\exists\bar{\beta}^k_\chi.F^k_\chi}\Big]$ to prenex-normal form and duplicating shared premises. Let $\bar{\beta}^k = \overline{\bar{\beta}^k_\chi}$. We perform joint constraint abduction under prefix $\mathcal{Q}^k$ (with parameters $\bar{\beta}^k$), let $\exists\bar{\alpha}^k.A^k$ be one of the abduction answers. We divide the atoms of the answer $\exists\bar{\alpha}^k.A^k$ into solutions to predicate variables A^k_χ and a remainder, or residuum, $A^k_{\mathrm{res}} = A^k \setminus \cup_\chi A^k_\chi$, depending on the variables in the atoms and so that the residuum holds under the quantifiers: $\vDash \mathcal{Q}^k.A^k_{\mathrm{res}}$. Let $\exists\bar{\beta}^{k+1}_\chi.F^{k+1}_\chi = \exists\bar{\beta}^k_\chi\bar{\alpha}^k.F^k_\chi \wedge A^k_\chi[\beta_\chi := \delta]$.

If in kth iteration, $A^k_{\mathrm{res}} = A^k$, we are done. From $\vDash \mathcal{Q}^k.A^k$ and $\vDash A^k \Rightarrow \wedge_i (D^k_i \Rightarrow C^k_i)$, it follows that the constraint holds: $\mathcal{I} \vDash \Phi$ for $\mathcal{I} = \Big[\ \bar{\chi} := \overline{\exists\bar{\beta}^k.F^k_\chi}\Big]$.

When $A^k_\chi \neq \boldsymbol{T}$ for some χ, we need to perform another iteration. It might be that the constraints added by a positive occurrence of χ – a recursive call – cannot all fit in next iteration's $\vDash \mathcal{Q}.A^{k+1}_{\text{res}}$ and have to be a part of next iteration's A^{k+1}_χ.

For postconditions, we want the strongest possible solutions, because a stronger postcondition provides more information at use sites of a definition. Therefore, in each iteration, we use constraint generalization to solve for binary predicate variables $\chi_K(\cdot,\cdot)$ without "hurting" the constraint. The generalization results $\exists\bar{\alpha}^k_{\chi_K}.G^k_{\chi_K}$ are used to get the first order constraints for abduction $\left[\bar{\chi} := \overline{\exists\bar{\beta}^k_\chi.F^k_\chi};\ \overline{\chi_K} := \overline{\exists\bar{\alpha}^k_{\chi_K}.G^k_{\chi_K}}\right]$. However, we have more work when due to recursive calls of functions returning values of existential types, $A^k_{\chi_K} \neq \boldsymbol{T}$. Note that postconditions of each iteration are constructed from scratch, they do not accumulate. The way we utilize answers $A^k_{\chi_K}$ is by performing second-stage abduction. Let $\vee_i D^{k,\chi_K}_i$ be the constraint generalization problem resulting in the postcondition $G^k_{\chi_K}$. We form the joint constraint abduction problem $\mathcal{Q}^k.\wedge_i (D^{k,\chi_K}_i \Rightarrow A^k_{\chi_K})$, and split its answer A^{k,χ_K} in the same way as for the first-stage abduction described above. We include the result in the partial solutions: $\exists\bar{\beta}^{k+1}_\chi.F^{k+1}_\chi = \exists\bar{\beta}^k_\chi\bar{\alpha}^k.F^k_\chi \wedge (A^k_\chi \wedge_K A^{k,\chi_K}_\chi)[\beta_\chi := \delta]$. This way, D^{k+1,χ_K}_i may be strong enough to imply $A^k_{\chi_K}$, and hopefully $A^{k+1}_{\chi_K} = \boldsymbol{T}$.

But we are still in trouble. Without the postcondition already in place, branches with recursive calls have too little information to imply what the postcondition should be. Therefore, during initial iterations ($k = 0$ and $k = 1$), we only include non-recursive branches in the constraint generalization process. This leads to "overshooting": the initial postconditions may reflect properties specific to base cases. These properties get relaxed during successive iterations, and we only end with success after reaching a fix-point.

The termination condition of the iteration might not be met. Actually, InvarGenT reports type inference failure if the solutions F^k_χ not converge after a fixed number of iterations. We have only observed diverging solutions due to numerical constraints.

4.2. Constraint Abduction

Abduction is a reasoning technique concerned with the search for explanations. An abduction problem is usually given by a background theory Θ and a formula C, and the answer is a formula A such that $\Theta \cup \{A\} \vDash C$ (relevance), $\Theta \nvDash \neg A$ (consistency), and A has some restricted syntactical form, see Marta Mayer and Fiora Pirri [31]. We are however interested in *constraint abduction* problems, with constraints expressed over a fixed model $\mathcal{M}$. A constraint abduction problem is then given by a pair of formulas D, C, and A is its answer when (at least) $\mathcal{M} \vDash (D \wedge A) \Rightarrow C$ (relevance) and $\mathcal{M} \vDash \exists \text{FV}(D, A).D \wedge A$ (consistency), see Michael Maher *Herbrand constraint abduction* [27]. We extend the formulation of joint constraint abduction (see [27]) to constraints with quantifiers. In general abduction, where a model is built as a solution, the quantifiers could be eliminated by Herbrandization. However, Herbrandization (and Skolemization) are operations on the level of a logic, not within a model – they do not return equivalent formulas and do not work in a fixed model. The introduced functions are uninterpreted. We opt to keep the model $\mathcal{M}$ and handle quantification explicitly. We develop a combination procedure for abduction algorithms when their domains of constraints are combined in a very simple way (yet sufficing for type-inference-driven invariant generation).

4.2.1. Formulating the Joint Constraint Abduction Problem

In joint, also called simultaneous, problems, we expect a single answer to solve several problems, here: to solve a conjunction of implications.

A *solved form* is a syntactically specified class of formulas, associated with a class of problems, for which satisfiability is trivial to check. We restrict solved forms to existentially quantified conjunctions of atoms, $\exists\bar{\alpha}.A$. The variables $\bar{\alpha}$ of a solved form $\exists\bar{\alpha}.A$ that is an abduction problem answer, are "free parameters" of the answer, they are required to be "unconstrained".

Due to the iterative nature of the main algorithm, we need to account for prior parameters $\bar{\beta}$ similar to the new parameters $\bar{\alpha}$.

The constraints that we need to solve form a *Joint Constraint Abduction under a Quantifier Prefix problem* (*JCAQP* problem for short) of the form $\mathcal{Q}.\wedge_i(D_i \Rightarrow C_i)$, where D_i and C_i are conjunctions of atomic formulas, and $\mathcal{Q}$ is an arbitrary quantifier prefix. We assume that $\mathrm{FV}(\wedge_i(D_i \Rightarrow C_i)) \subseteq \mathcal{Q}$. We also have a set $\bar{\beta}$ of *parameters* of the invariants. The conditions on abduction answers $\exists\bar{\alpha}.A$ are as follows:

1. *relevance condition*: $\mathcal{M} \models \wedge_i(D_i \wedge A \Rightarrow C_i)$,
2. *validity condition*: $\mathcal{M} \models \mathcal{Q}.A[\bar{\alpha}\bar{\beta} := \bar{t}]$ for some $\bar{t}$,
3. *consistency condition*: $\mathcal{M} \models \wedge_i \exists \mathrm{FV}(D_i \wedge A).D_i \wedge A$,
4. *no escaping variables* condition: for all atoms $c \in A$ such that $\mathcal{M} \not\models \mathcal{Q}.c$ and $\mathrm{FV}(c) \cap \bar{\beta} \neq \varnothing$, and for all $\beta_1 \in \mathrm{FV}(c)$ such that $(\forall\beta_1) \in \mathcal{Q}$, there exists $\beta_2 \in \mathrm{FV}(c) \cap \bar{\beta}$ such that $\beta_1 \leqslant_{\mathcal{Q}} \beta_2$.
    a. *Strong no escaping variables condition* is a stronger variant we use for sorts whose solved forms are substitutions. For all atoms $\beta_2 \doteq t \in A$ such that $\beta_2 \in \bar{\beta}$, if $\beta_1 \in \mathrm{FV}(c)$ such that $(\forall\beta_1) \in \mathcal{Q}$, then $\beta_1 \leqslant_{\mathcal{Q}} \beta_2$.

Definition 4.1. *$\exists\bar{\alpha}.A$ is an answer to a JCAQP problem $\mathcal{Q}.\wedge_i(D_i \Rightarrow C_i)$ with parameters $\bar{\beta}$ for model $\mathcal{M}$, written $\exists\bar{\alpha}.A \in \mathrm{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$, when A is a conjunction of atoms, $\bar{\alpha}\#\mathrm{FV}(\wedge_i(D_i \Rightarrow C_i), \bar{\beta})$, meeting the relevance condition, validity condition, consistency condition and* no escaping variables *condition.*

We can also consider Joint Abduction under a Quantifier Prefix problems for a logic, checking relevance condition: $\models \wedge_i(D_i \wedge A \Rightarrow C_i)$ and validity condition: $\models \mathcal{Q}.A[\bar{\alpha}\bar{\beta} := \bar{t}]$. The consistency conditions: for all i, $D_i \not\models \neg A$, are always met.

The natural setting for constraint abduction problems is with a fixed model. Above we extend the definition to the case where instead of a model just a logic is given, just to shed light on relations between constraint abduction, general abduction, and decision (i.e. validity or satisfiability) problems.

We call a JCAQP problem $\mathcal{Q}.D \Rightarrow C$ (i.e. a non-simultaneous problem) a *simple constraint abduction under a quantifier prefix* problem SCAQP. We write $\mathrm{JCAQP}_{\mathcal{M}}$ to indicate the model.

Proposition 4.2. *If the $JCAQP_{\mathcal{M}}$ problem $\mathcal{Q}.\wedge_i(D_i \Rightarrow C_i)$ without parameters has an answer, then $\mathcal{M} \models \mathcal{Q}.\wedge_i(D_i \Rightarrow C_i)$.*

We say that JCAQP$_{\mathcal{M}}$ answer $\exists\bar{\alpha}.A$ is *more general* than $\exists\bar{\beta}.B$, when there exist terms $\bar{t}$ such that $\mathcal{M}\vDash B\Rightarrow A[\bar{\alpha}:=\bar{t}]$.

Example 4.3. For a free term algebra $T(F)$ over signature F containing binary functors f, g and constants a, b, and JCAQP$_{T(F)}$ problem $\exists x, y, z.(y\doteq f(a,x)\Rightarrow z\doteq a)\wedge(y\doteq f(b,x)\Rightarrow z\doteq b)$, the most general answer is $\exists\alpha.y\doteq f(z,\alpha)$. Indeed, $\forall\alpha\exists x,\ y,\ z.y\doteq f(z,\alpha)\wedge y\doteq f(a,x)$ holds with $x=\alpha, y=f(a,\alpha), z=a$ and $\forall\alpha\exists x, y, z.y\doteq f(z,\alpha)\wedge y\doteq f(b,x)$ holds with $x=\alpha, y=f(b,\alpha), z=b$. For the SCAQP$_{T(F)}$ problem $\exists x,\ y,\ z.(y\doteq f(a,\ x)\Rightarrow z\doteq a)$, the set of maximally general answers is $\{\exists\alpha.y\doteq f(z,\alpha), z\doteq a\}$.

Consider a conjunction of atoms C interpreted in any free term algebra $T(F)$ over a signature F. If C is satisfiable, let $\boldsymbol{U}(C)$ be a conjunction of equations whose left-hand-sides are variables not occurring in any of the right-hand-sides, such that $T(F)\vDash C\Leftrightarrow\boldsymbol{U}(C)$, otherwise let $\boldsymbol{U}(C)=\bot$. Let $\boldsymbol{U}(\mathcal{Q}.C)$ in case $T(F)\nvDash\mathcal{Q}.C$ be $\bot$, and otherwise be as before but with equations directed so that variables later in the prefix are on the left. $\boldsymbol{U}(\mathcal{Q}.C)$ can be computed by unification with linear constant restrictions, see Baader and Schulz [3]. In effect: if for some $x\doteq t\in\boldsymbol{U}(C)$ such that $(\exists t)\notin\mathcal{Q}$ (e.g. t is not a variable) and $(\forall x)\in\mathcal{Q}$, then $\boldsymbol{U}(\mathcal{Q}.C)=\bot$; if for some $x\doteq t\in\boldsymbol{U}(C)$ such that $(\exists x)\in\mathcal{Q}$, there is a $y\in\mathrm{FV}(t)$, $(\forall y)\in\mathcal{Q}$ and $x\leqslant_{\mathcal{Q}}y$, then $\boldsymbol{U}(\mathcal{Q}.C)=\bot$; otherwise $\boldsymbol{U}(\mathcal{Q}.C)=\boldsymbol{U}(C)$. Let $\boldsymbol{U}_{\bar{\alpha}}(\mathcal{Q}.C)$ be $\boldsymbol{U}_{\bar{\alpha}}((\mathcal{Q}\backslash\{\forall\bar{\alpha}\})\exists\bar{\alpha}.C)$, i.e. $\boldsymbol{U}_{\bar{\alpha}}(\mathcal{Q}.C)=\boldsymbol{U}(\mathcal{Q}'.C)$ where variables $\bar{\alpha}$ are existentially quantified in $\mathcal{Q}'$ and otherwise $\mathcal{Q}'$ is like $\mathcal{Q}$. Computing $\boldsymbol{U}_{\bar{\alpha}\bar{\beta}}(\mathcal{Q}.A)$ decides the validity condition for $\mathcal{M}=T(F)$.

DEFINITION 4.4. *An abduction algorithm for JCAQP$_{\mathcal{M}}$ with parameters $\bar{\beta}$,* Abd$(\mathcal{Q},\ \bar{\beta},\ \overline{D_i, C_i})$*, generates a sequence of quantified conjunctions of atoms $\overline{\exists\bar{\alpha}_j.A_j}$, possibly infinite. The algorithm is* correct *if for every j, $\exists\bar{\alpha}_j.A_j$ is an answer to the JCAQP$_{\mathcal{M}}$ problem $\mathcal{Q}.\wedge_i(D_i\Rightarrow C_i)$ with parameters $\bar{\beta}$. It is* complete *if for every JCAQP$_{\mathcal{M}}$ answer $\exists\bar{\alpha}.A$, there is a j and some $\bar{t}$ such that $\mathcal{M}\vDash A\Rightarrow A_j[\bar{\alpha}_j:=\bar{t}]$ (with variables renamed so that $\bar{\alpha}\#\mathrm{FV}(A_j)$). If the sequence is empty, we write* Abd$(\mathcal{Q},\bar{\beta},\overline{D_i, C_i})=\bot$.

Note that we do not require that only maximally general answers are returned. Although this would be preferrable, it is costly to guarantee in many constraint domains even with incomplete algorithms.

4.2.2. Abduction Algorithm for The Combination of Domains

We provide a plug-in architecture where to add a new sort to the logic it is enough to give an algorithm solving the JCAQP problem. Define an *alien subterm* (cf. [3]) of a term τ of sort s_{type} to be a maximally large subterm t of τ of sort $s\neq s_{\text{type}}$: a subterm t that is not a subterm of a subterm of sort $s'\neq s_{\text{type}}$. Let usorts$=$sorts$\backslash\{s_{\text{type}}\}$.

Let $\mathcal{L}_{\text{ty}}=T(F,\cup_{s\in\text{sorts}}X_s)$ be a language interpreted in a multi-sorted free term algebra $T=T(F\cup_{s\in\text{usorts}}D_s)$ which, besides the term variables $X_{s_{\text{type}}}$, has *alien subterm variables* $\cup_{s\in\text{usorts}}X_s$.

Let $\mathcal{Q}$ be a quantifier prefix and D_i, C_i be atomic conjunctions in $\mathcal{L}$ that form a joint abduction problem $\mathcal{Q}.\wedge_i(D_i\Rightarrow C_i)$. For the purpose of Theorem 4.5, let Abd$_s$ be complete JCAQP algorithms for $s\in$ usorts and Abd$_T$ be a complete JCAQP algorithm for the free term algebra T. We start the multi-sorted abduction procedure Abd$(\mathcal{Q},\bar{\beta},\overline{D_i, C_i})$ by performing

Abd_T on the s_{type} part of constraints with alien subterms replaced by variables. For each JCAQP solution $\exists\bar{\alpha}_j.A_j$, we replace the s_{type} part of the ith premise by the "residual" formula $A^i_{p_j}$ and of ith conclusion by "residual" formula $A^i_{c_j}$, defined in Table 4.1. We split the resulting JCAQP problem into single-sort problems for each sort $s \in \text{usorts}$, and we solve them using Abd_s algorithms. Finally, we build answers as conjunctions of answers for each sort.

Let $\mathcal{Q}$ be a quantifier prefix and D_i, C_i be atomic conjunctions in $\mathcal{L}$ that form a joint abduction problem $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$ with parameters $\bar{\beta}$, and $\overline{\forall\beta} \subset \mathcal{Q}$. Let Abd_s be correct and complete JCAQP algorithms for $s \in \text{usorts}$ and Abd_T be a correct and complete JCAQP algorithm for language $\mathcal{L}_{\text{ty}} = T(F, \cup_{s\in\text{sorts}} X_s)$ and model $T(F \cup_{s\in\text{usorts}} D_s)$. We define the algorithm $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$ in Table 4.1. In this algorithm, we separate formulas into single-sorted formulas, we perform abduction for terms, and we perform abduction for other sorts with additional equations due to abduction for terms. The proof of the following theorem is in Section A.2.2.

Theorem 4.5. *Assume the conditions listed above. Then* Abd *meets Definition 4.4* correct *and* complete *algorithm conditions.*

Introducing new variables $\bar{\alpha}^j_r$ puts additional burden on abduction in other sorts. In the current implementation of InvarGenT, we only replace A^j_T with $A^{j\,\prime}_T$ in the first iteration, when abduction in other sorts is not performed.

For the details of the implementation, see Section B.2.1.

4.2.3. Joint Constraint Abduction

In Table 4.1, we assumed abduction algorithms are returning sequences of answers, from which the answers of interest are extracted. We implement abduction algorithms differently. We maintain a *discard list* – a list of answers to avoid, and the algorithms return the first answer they find which does not imply any answer in the discard list.

Besides the discard list, the simple abduction algorithms take another argument: the partial answer. By starting the search for an answer to an implication from the joint solution to already solved implications, we ensure by construction that the final answer to the joint constraint abduction problem meets validity and consistency conditions. The relevance and no escaping variables conditions would be met regardless of starting from the partial answer. However, the search is greatly facilitated, because simple constraint abduction does not need to rediscover relevant parts of the answers to already solved implications.

Sometimes such rediscovery is not possible. Abduction answer $\exists\bar{\alpha}.A$ to $D \Rightarrow C$ is *fully maximal* when $\mathcal{M} \vDash (\exists\bar{\alpha}.D \wedge A) \Leftrightarrow D \wedge C$. The notion was introduced by Michael Maher, see e.g. [27], to curb the difficulty of finding all abduction answers. Even equipped only with fully maximal abduction algorithms, by accumulating answers across implications we can still solve problems where some implications do not have fully maximal answers. To this effect, we vary the order in which implications are solved, so that when an implication $D \Rightarrow C$ without fully maximal answers is encountered, the answer $\exists\bar{\alpha}_p.A_p$ to previous implications is such that $D \wedge A_p \Rightarrow C$ has a fully maximal answer. In fact, our simple constraint abduction algorithms go beyond fully maximal answers.

let $D_i \equiv \wedge_s D_i^s$ and $C_i \equiv \wedge_s C_i^s$, where, for $s \in$ sorts, D_i^s, C_i^s are atomic conjunctions in $\mathcal{L}_s$.
let D_i^t, C_i^t be formulas $D_i^{s_{\text{type}}}, C_i^{s_{\text{type}}}$ with subterms $\bar{r}_i$ replaced with fresh variables $\overline{\alpha_{r_i}}$
(such that $\alpha_r \in X_s$ for r of sort s)
where $\bar{r}_i$ are all alien subterms in $D_i^{s_{\text{type}}}, C_i^{s_{\text{type}}}$
let $\bar{\bar{r}} = \bar{r_1}...\bar{r_n}$, $\bar{\bar{\alpha}}_r = \overline{\alpha_{r_1}}...\overline{\alpha_{r_n}}$
if D_i^t is not satisfiable for some i, **then** $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i}) := \text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_j, C_j}_{j \neq i})$
else if C_i^t is not satisfiable for some i, **then return** $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i}) := \bot$
else let $\wedge_s D_{i,s}^t = \boldsymbol{U}(D_i^t)$, $\wedge_s C_{i,s}^t = \boldsymbol{U}(C_i^t)$ be solved forms of D_i^t, C_i^t
Similarly, discard branches i for which $D_{i,s}^t \wedge D_i^s$ is not satisfiable for some $s \in$ usorts.
if $\text{Abd}_T(\mathcal{Q}, \bar{\bar{\alpha}}_r \bar{\beta}, \overline{D_{i,s_{\text{type}}}^t, C_{i,s_{\text{type}}}^t}) = \bot$, **then** $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i}) := \bot$
else let $\overline{\exists \overline{\alpha_j^T}.A_T^j} = \text{Abd}_T(\mathcal{Q}, \bar{\bar{\alpha}}_r \bar{\beta}, \overline{D_{i,s_{\text{type}}}^t, C_{i,s_{\text{type}}}^t})$
let $A_T^{j\prime}$ be A_T^j with alien subterms replaced by distinct fresh variables $\bar{\alpha}_r^j$, $\overline{\alpha_j^T}' = \overline{\alpha_j^T} \bar{\alpha}_r^j$
let $A_{p_j}^i = \{x \doteq t \in \boldsymbol{U}(D_{i,s_{\text{type}}}^t \wedge A_T^{j\prime}) | x \in X_s, s \neq s_{\text{type}}\}$
let $A_{c_j}^i = \{x \doteq t \in \boldsymbol{U}(D_{i,s_{\text{type}}}^t \wedge C_{i,s_{\text{type}}}^t \wedge A_T^{j\prime}) | x \in X_s, s \neq s_{\text{type}}\}$
let $A_{p/c,j}^i = \wedge_s A_{p/c,j,s}^i$, $A_{p/c,j,s}^i \in \mathcal{L}_s$
let $J_s = \left\{ j \middle| \text{Abd}_s(\mathcal{Q}, \bar{\alpha}_r^j \bar{\beta}, \overline{D_i^s \wedge (D_{i,s}^t \wedge A_{p,j,s}^i)[\bar{\bar{\alpha}}_r := \bar{\bar{r}}], C_i^s \wedge (C_{i,s}^t \wedge A_{c,j,s}^i)[\bar{\bar{\alpha}}_r := \bar{\bar{r}}]}) \neq \bot \right\}$
let $\overline{\exists \overline{\alpha_s^{k_j^s}}.A_s^{k_j^s}} = \text{Abd}_s(\mathcal{Q}, \bar{\alpha}_r^j \bar{\beta}, \overline{D_i^s \wedge (D_{i,s}^t \wedge A_{p,j,s}^i)[\bar{\bar{\alpha}}_r := \bar{\bar{r}}], C_i^s \wedge (C_{i,s}^t \wedge A_{c,j,s}^i)[\bar{\bar{\alpha}}_r := \bar{\bar{r}}]})$,
$j \in \cap_{s \in \text{usorts}} J_s$
if for some $s \in$ usorts, $J_s = \varnothing$, **then** $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i}) := \bot$
return $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i}) := \overline{\exists \overline{\alpha_{\overline{k_j^s}}}.A_T^{j\prime} \wedge_s A_s^{k_j^s}}_{\overline{k_j^s}: j \in \cap_s J_s}$,
where $\overline{\alpha_{\overline{k_j^s}}}$ are $\overline{\alpha_j^T}' \overline{\overline{\alpha_s^{k_j^s}}} \cap \text{FV}\left(A_T^{j\prime} \wedge_s A_s^{k_j^s}\right)$,
$\overline{\overline{\alpha_s^{k_j^s}}}$ is a concatenation of $\overline{\alpha_s^{k_j^s}}$ for $s \in$ usorts,
for $j \in \cap_s J_s$, $\bar{k}_j^s$ span the cartesian product $\times_{s \in \text{usorts}} \text{Abd}_s$ of solutions for fixed j

Table 4.1. Complete multi-sorted abduction $\text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$

Rather than testing permutations blindly, we use a search scheme which might not capture all opportunities to solve a JCA problem, but detects unsolvable JCA problems earlier. We set aside branches that do not have any answer extending the partial answer so far. After all branches have been tried and the partial answer is not an empty conjunction (i.e. not $\boldsymbol{T}$), we retry the set-aside branches. If during the retry, any of the set-aside branches fails, we add the partial answer to discarded answers – which are avoided during simple abduction – and restart. Restart puts the set-aside branches to be tried first. If, when left with set-aside branches only, the partial answer is an empty conjunction, i.e. all the answer-contributing branches have been set aside, we fail – return $\bot$ from the joint abduction.

Before starting joint constraint abduction, we separate out negative constraints, as explained in Section 4.3. To ensure the overall consistency and validity conditions without needless backtracking, we pass a validation suite to SCA algorithms. The validation ensures that the partial answers are consistent with all implications of the constraint. That is, we reject the partial answers A such that $\mathcal{M} \not\models \wedge_i \exists \text{FV}(D_i \wedge C_i \wedge A).D_i \wedge C_i \wedge A$.

For the details of the algorithm, see Section B.2.2.

4.2.4. Simple Constraint Abduction

JCA problems for most interesting domains are unknown to be decidable, because the corresponding SCA problems are not known to have effective characterizations of the sets of answers. Maher [27] introduces subsets of answers which are more amenable to search: abduction answer $\exists\bar{\alpha}.A$ to SCA problem $D \Rightarrow C$ is fully maximal when $\mathcal{M} \models (\exists\bar{\alpha}.D \wedge A) \Leftrightarrow D \wedge C$. We can search for fully maximal answers using various forms of a brute-force approach: starting from an *initial candidate* $D \wedge C$ and generalizing until we find a formula, implied by $D \wedge C$, which meets all conditions for a correct SCA answer and all its further generalizations do not imply C. It turns out that fully maximal answers are insufficient. We have come up with two additional ways to introduce initial candidates. One is to add – guess – atoms constraining variables which are already significantly constrained by the premise D. The "significant constraint" condition limits the number of guesses to try. The other way is to decompose the SCA problem $D \Rightarrow C$ into subproblems $d \Rightarrow c$ for atoms $d \in D, c \in C$ and add their answers to initial candidates. In many domains, all maximally general answers to $d \Rightarrow c$ can be easily enumerated. Even considering all initial candidates described in this paragraph does not ensure finding all maximally general answers.

We preprocess the initial SCA answer candidate C_a by trying to eliminate universally quantified variables, using the premise of the SCA problem. We define this preprocessing, separately for the different sorts, as $\mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D, C_a)$. It makes the valididty condition of the resulting answer, $\mathcal{M} \models \mathcal{Q}.A$, easier to meet. See Dillig, Dillig, Li and McMillan [12] for a similar approach.

4.2.4.1. Abduction for Terms

The JAQP problem for first order logic with function symbols and equality is undecidable, because it is equivalent, by Herbrandization, to simultaneous rigid E-unification (see Degtyarev and Voronkov [11]): the substitution that is a solution to simultaneous rigid E-unification when expressed as a conjunction of equations has the same properties as a JCAQP answer (therefore the existence of JCAQP answers coincides with intuitionistic satisfiability). Remember that outside of Definition 4.1, we use $\models\Phi$ as a shorthand for $\mathcal{M} \models \Phi$.

The decision problem $T(F) \models \mathcal{Q}.\ \wedge_i (D_i \Rightarrow C_i)$ is decidable, see Comon [9]. Actually, [9] provides a disjunction as a solution, each disjunct meeting the relevance and validity conditions of the $\mathrm{JCAQP}_{T(F)}$ problem. It is often the case though that each disjunct does not meet the consistency condition, despite the $\mathrm{JCAQP}_{T(F)}$ problem considered having answers.

The JCAQP problem for free term algebra $T(F)$ is unknown to be decidable. A limited form of JCA is to find the fully maximal answers, introduced in [27], using non-simultaneous abduction algorithm from Maher and Huang [29]. [27] refers to [25] as establishing that there are finitely many fully maximal answers. [29] gives an algorithm finding fully maximal answers for simple (i.e. non-simultaneous) constraint abduction problems.

Let us recall why Herbrandization is insufficient and therefore we cannot apply the algorithm from [29] without some adjustments. Take a formula $\exists x.(a \doteq b(x) \Rightarrow x \doteq b(x)) \wedge \varphi(x)$. In the original model $T(F)$, the formula is equivalent to $\exists x.\varphi(x)$, because $a \doteq b(x)$ is equivalent to falsehood. However, it is a Herbrandization of $\exists x.(\forall b.a \doteq b \Rightarrow x \doteq b) \wedge \varphi(x)$, which is equivalent to $\varphi(a)$.

Fully maximal answers are not sufficient for the practical application of joint constraint abduction. Consider a constructor `Pair`: $\forall\alpha\beta.\text{Term}(\alpha) \times \text{Term}(\beta) \longrightarrow \text{Term}((\alpha, \beta))$ and a pattern matching branch that we could add to our `eval` function: `| Pair (y1, y2) -> (eval y1, eval y2)`. It leads to an implication:

$$\alpha \doteq \text{Term}((\alpha', \beta')) \Rightarrow \beta \doteq (\alpha'', \beta'')$$

where α', β' are universally quantified and α'', β'' are existentially quantified. The expected abduction answer $\alpha \doteq \text{Term}(\beta) \wedge \alpha'' \doteq \alpha' \wedge \beta'' \doteq \beta'$ is not fully maximal because:

$$\alpha \doteq \text{Term}((\alpha', \beta')) \wedge \beta \doteq (\alpha'', \beta'') \not\Rightarrow \alpha' \doteq \alpha'' \wedge \beta' \doteq \beta''$$

In the case of `eval`, the inference problem is solved by our JCA scheme even based on fully maximal abduction. But examples from Chuan-kai Lin [22] (for example the `zip2` and `zip1` functions) motivated a development that goes beyond fully maximal answers: guessing equations between variables.

To show how we eliminate universally quantified variables, we slightly abuse notation:

$$\begin{aligned} S &= [\bar{t_u} := \bar{t'_u}] \text{ for } \text{FV}(t_u) \cap \bar{\beta_u} \neq \varnothing, \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \Rightarrow \dot{S}, \\ S' &= [\bar{u} := \bar{t'_u}] \text{ for } \bar{u} \subset \bar{\beta_u}, \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \wedge C \Rightarrow \dot{S}', \\ \text{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D, C) &= \{c' | c = x \doteq t \in C, \text{if } x = S'(t) \text{ then } c' = S(c) \text{ else } c' = S\,S'(c)\} \end{aligned}$$

S is a substitution of subterms rather than a regular substitution of variables.

To solve $D \Rightarrow C$ the algorithm from [29] page 13, reproduced in Table 2.5, starts with $\boldsymbol{U}(D \wedge C)$ and iteratively replaces subterms by fresh variables $\alpha \in \bar{\alpha}$ for a final solution $\exists\bar{\alpha}.A$. If the same subterm occurs at multiple positions, we try replacing by the same variable at subsets of these positions. We start from $\text{Rev}_\forall(\mathcal{Q}, \bar{\beta}, \boldsymbol{U}(D \wedge A_p), \boldsymbol{U}(A_p \wedge D \wedge C))$, where $\exists\bar{\alpha}_p.A_p$ is the solution to previous problems solved by the joint abduction algorithm. Optionally, we also substitute-out in the initial candidates, constants $\tau := \alpha$ when $\alpha \doteq \tau \in \boldsymbol{U}(D \wedge A_p)$. The modification going beyond fully maximal answers, is to consider candidate atoms not implied by $D \wedge C$, even in the case without an initial partial answer: $A_p = \boldsymbol{T}$. A natural choice is to consider equations $\beta_1 \doteq \beta_2$ for parameters $\beta_1\beta_2 \subseteq \bar{\beta}$. To curtail the search space, we limit the choices of pairs β_1, β_2 to cases $A_p \wedge D \wedge C \Rightarrow \beta_1 \doteq \tau_1 \wedge \beta_2 \doteq \tau_2$ where τ_1 and τ_2 are not variables but are unifiable.

For the details of the algorithm, see Section B.2.3.

4.2.4.2. Abduction for Linear Arithmetic

We start with some insights into abduction for linear arithmetic, and then discuss our algorithm. Unlike in term abduction, there is no need to introduce variables.

PROPOSITION 4.6. *The domain of linear arithmetic has the following* quantifier elimination *property: for every constraint (i.e. conjunction of atoms) A and variables $\bar{\alpha}$ there exists a constraint A' such that $\mathcal{M} \vDash (\exists\bar{\alpha}.A) \Leftrightarrow A'$.*

With inequalities, it can happen that there are no maximally general answers, there can also be infinitely many maximally general answers outside of fully maximal answers.

The *implicit equalities* E of a conjunction C is a conjunction of equations of biggest rank such that $\mathcal{M} \vDash C \Rightarrow E$ and E are linearly independent of equations in C.

Proposition 4.7. ([26] p. 14 Lemma 10) *Suppose $D \wedge C$ has implicit equalities E. Then $\vDash A \wedge D \Rightarrow C$ iff $\vDash A \wedge D \Rightarrow E$ and $\vDash \tilde{E}(A \wedge D) \Rightarrow \tilde{E}(C)$.*

Consider the SCA problem $x \doteq 2r \wedge y \doteq 2s \Rightarrow z \doteq 2t$ and a maximally general answer $A = x + y \doteq z \wedge r + s \doteq t$. It is not fully maximal, because $C \wedge D$ does not imply any relation between x, y and z.

To simplify the search in presence of a quantifier prefix, we preprocess the initial candidates by trying to eliminate universally quantified variables:

$$\begin{aligned} S &= [\bar{\beta_u} := \bar{t_u}] \text{ for } \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \Rightarrow \dot{S}, \\ \mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D, C) &= \{c' | c \in C, \text{if } \mathcal{M} \vDash \mathcal{Q}.c[\bar{\beta} := \bar{t}] \text{ for some } \bar{t} \text{ then } c' = c \text{ else } c' = S(c)\} \end{aligned}$$

We approach solving for equations $c = t_1 \doteq t_2 \in C$ and inequalities $c = t_1 \leqslant t_2 \in C$ differently. For equations, we construct initial candidates as linear combinations of c and selected equations from D. For inequalities, we construct the initial candidates by finding the abudction answers to $d \Rightarrow c$ for each inequality $d \in D$.

To find the abduction answers to $d \Rightarrow c$, pick a common variable $\alpha \in \mathrm{FV}(d) \cap \mathrm{FV}(c)$ or the constant $\alpha = 1$. We have four possibilities:

1. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answers are c and $d_\alpha \leqslant c_\alpha$,
2. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answer is only c,
3. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answer is only c,
4. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answers are c and $c_\alpha \leqslant d_\alpha$.

Thanks to cases (1) and (4) above, the abduction algorithm can find some answers which are not fully maximal.

To check whether $A \Rightarrow B$, we check for each $b \in B$:

- if $b = x \doteq y$, that $A(x) = A(y)$, where $A(\cdot)$ is the substitution corresponding to equations and implicit equalities in A;
- if $b = x \dot{\leqslant} y$, that $A \wedge y \dot{<} x$ is not satisfiable.

For more details of the algorithm, see Section B.2.4.

4.3. Constraint Generalization

We define *constraint generalization* as the task of finding common consequences, as specific as possible, of conjunctions of atomic formulas. Notice that there is at most one maximally specific constraint generalization answer. Therefore the completeness condition for constraint generalization reduces to requiring most specific answers.

DEFINITION 4.8. *A quantified conjunction of atoms $\exists\bar{\alpha}.A \in \mathcal{F}$ is a* constraint generalization answer *to $\vee_i D_i$, where D_i are conjunctions of atoms in $\mathcal{L}$, when $\mathcal{M} \vDash \wedge_i(D_i \Rightarrow \exists\bar{\alpha}.A)$. A constraint generalization answer is* most specific *when: for every solution $\exists\bar{\alpha}_s.A_s$ with $\mathcal{M} \vDash \wedge_i(D_i \Rightarrow \exists\bar{\alpha}_s.A_s)$, $\mathcal{M} \vDash A \Rightarrow \exists\bar{\alpha}_s.A_s$ (with variables renamed so that $\bar{\alpha}_s \# \mathrm{FV}(A)$). By $\mathrm{LUB}(\overline{D_i})$ we denote the most general constraint generalization answer to $\vee_i D_i$, by $\mathrm{LUB}(\cdot)$ an algorithm that computes it.*

Consider an `eval`-like example, where the disjuncts include the conjunctions of premises and conclusions of a type inference constraint. We are interested in finding

$$\begin{aligned}\mathrm{LUB}(\overline{D_i \wedge C_i}) \;= \mathrm{LUB}\;\; & (\beta \doteq \alpha \to \beta_1 \wedge \alpha \doteq \mathrm{Term}(\mathrm{Int}) \wedge \beta_1 \doteq \mathrm{Int},\\ & \beta \doteq \alpha \to \beta_1 \wedge \alpha \doteq \mathrm{Term}(\mathrm{Bool}) \wedge \beta_1 \doteq \mathrm{Bool},\\ & \beta \doteq \alpha \to \beta_1 \wedge \alpha \doteq \mathrm{Term}(\gamma) \wedge \beta_1 \doteq \gamma)\end{aligned}$$

for which the solution is $\exists\alpha_1.\beta \doteq \alpha \to \beta_1 \wedge \alpha \doteq \mathrm{Term}(\alpha_1) \wedge \beta_1 \doteq \alpha_1$.

We discuss issues specific to postcondition inference in Section B.3.

4.3.1. Algorithm for Combining Domains

First order most specific anti-unifiers are closely related to constraint generalization.

DEFINITION 4.9. *A sequence of substitutions $\bar{S_i}$ is an* anti-unifier *of a same-length sequence of terms $\bar{t_i}$ when there is a term t_G such that $\forall i, S_i(t_G) = t_i$. The term t_G is called a* common generalization *of $\bar{t_i}$.*

An anti-unifier $\bar{S_i}$ is a most general anti-unifier *of $\bar{t_i}$ when given any other anti-unifier $\bar{\eta_i}$ there exists a substitution ρ such that $\eta_i = S_i\rho$. We call t_G such that $S_i(t_G) = t_i$ the* most specific generalization *(MSG) of $\bar{t_i}$. We require, without loss of generality, that variables used by anti-unification are fresh: $\{x \in X \,|\, S_i(x) \neq x\} \# \mathrm{FV}(\bar{t_i})$.*

We define an *alien subterm* (cf. Baader and Schulz [3]) of a term τ of sort s_{type} to be a maximally large subterm t of τ of sort $s \neq s_{\mathrm{type}}$. Let $\mathrm{LUB}_s(\overline{D_i^s})$ give most specific constraint generalization answers to $\vee_i D_i^s$ for conjunctions of atoms of sort s for $s \neq s_{\mathrm{type}}$. We define $\mathrm{LUB}(\overline{D_i}) = \exists\bar{\alpha}.A$ in Table 4.2. The algorithm separates formulas into single-sorted formulas, computes anti-unification of terms equal to the same variable in each disjunct, and calls constraint generalization for other sorts. It adds to the disjuncts passed to generalization for other sorts, equations binding the generalization variables (introduced by anti-unification) of the particular sort. To find the anti-unifiers, we adapt the anti-unification algorithm provided in Østvold [61], fig. 2. The proof of the following theorem is in Section A.3.

THEOREM 4.10. *$\mathrm{LUB}(\overline{D_i})$ from Table 4.2 meets the Definition 4.8 conditions for most specific constraint generalization answer to $\vee_i D_i$.*

In our actual implementation, we do not separate alien subterms, i.e. we do not compute D_i^t and D_i^a. Rather, we rely on our implementation of unification $\boldsymbol{U}$ to separate out equations in sorts other than s_{type}.

let $\wedge_s D_i^s \equiv D_i$ where D_i^s is of sort s
let D_i^t be $D_i^{s_{\text{type}}}$ with all alien subterms $\bar{a_i^j}$ replaced by fresh variables $\bar{\alpha_i^j}$ of corresp. sorts
let $D_i^a = \bar{\alpha_i^j} \doteq \bar{a_i^j}$ and $\wedge_s D_i^{t,s} \equiv D_i^a$ where $D_i^{t,s}$ is in sort s
let $\wedge_s D_{i,s}^t \equiv \boldsymbol{U}(D_i^t)$ where $D_{i,s}^t$ is of sort s
For the sort s_{type}:
  let $V = \{x_j, \overline{t_{i,j}} | \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{type}}}^t\}$
  let $G = \{\bar{\alpha}_j, g_j, \overline{S_{i,j}} | S_{i,j} = [\bar{\alpha}_j := \bar{g}_j^i], S_{i,j}(g_j) = t_{i,j}\}$
    be the most specific anti-unifiers of $\{\overline{t_{i,j}} | (x_j, \overline{t_{i,j}}) \in V\}$ for each j
  let $D_i^u = \wedge_j \bar{\alpha}_j \doteq \bar{g}_j^i$ and $D_i^g = D_{i,s_{\text{type}}}^t \wedge D_i^u$
  let $D_i^v = \{x \doteq y | x \doteq t_1 \in D_i^g, y \doteq t_2 \in D_i^g, \mathcal{M} \vDash D_i^g \Rightarrow t_1 \doteq t_2\}$
  let $A_{s_{\text{type}}} = \wedge_j x_j \doteq g_j \wedge \bigcap_i (D_i^g \wedge D_i^v)$
    where equations are ordered so that only one of $a \doteq b, b \doteq a$ appears anywhere
  let $\bar{\alpha}_{s_{\text{type}}} = \bar{\bar{\alpha}_j}$
let $\wedge_s D_{i,s}^u \equiv D_i^u$ for $D_{i,s}^u$ of sort s
For sorts $s \neq s_{\text{type}}$:
  let $\exists \bar{\alpha}_s. A_s = \text{LUB}_s\big(\overline{D_i^s \wedge \widetilde{D_i^a}(D_{i,s}^t \wedge D_{i,s}^u)}\big)$
return $\exists \bar{\bar{\alpha_i^j}} \bar{\alpha}_s. \wedge_s A_s$

Table 4.2. LUB algorithm $\text{LUB}(\overline{D_i}) = \exists \bar{\alpha}. A$

4.3.2. Linear Arithmetic

Equality under premises within linear arithmetic: $D \vDash \alpha \doteq \beta$ can be decided by checking whether the polytopes $D \wedge \alpha < \beta$ and $D \wedge \beta < \alpha$ are empty, which can be done by Fourier-Motzkin elimination.

In contrast to the free term algebra, for the logic of linear inequalities over real or rational numbers we have the following quantifier elimination property. For all $\exists \bar{\alpha}. A \in \mathcal{F}$, there is a conjunction of atoms A' such that $\mathcal{M} \vDash A' \Leftrightarrow \exists \bar{\alpha}. A$. Therefore, we need not introduce new variables.

A system of linear inequalities is *full-dimensional* when the set of solutions is not contained in an affine subspace of dimension smaller than the number of variables. In other words, a full-dimensional system of inequalities does not have implicit equalities. The task of constraint generalization $\text{LUB}(\overline{D_i})$ for full-dimensional inequalities is equivalent to finding the convex hull of the half-space represented, possibly unbounded polytopes D_i. Fukuda, Liebling and Lütolf [16] provide a polynomial-time algorithm to find the half-space represented convex hull of closed polytopes. The algorithm can be generalized to unbounded polytopes.

When all variables of an equation $a \doteq b$ appear in all branches D_i, we can turn the equation $a \doteq b$ into pair of inequalities $a \leqslant b \wedge b \leqslant a$. We eliminate all equations and implicit equalities which contain a variable not shared by all D_i, by substituting out such variables. We conjecture that the resulting algorithm meets the correctness and completeness conditions.

More details of the algorithm implemented in InvarGenT can be found in Section B.3.1.

4.3.3. Abductive Constraint Generalization

It turns out that constraint generalization as we defined it is insufficient. Consider replacing `function` by `efunction` in a function of type $\tau_1 \to \tau_2$ for which type inference works fine. We expect the resulting type to have the form $\tau_1 \to \exists.\tau_2$, i.e. without existential type variables. But while unification reconstructs τ_2 from the result types of the branches, constraint generalization will fail to generate τ_2 when a branch under-constrains the result, does not "care" about the specific type. To mitigate this problem we introduce *abductive constraint generalization* and modify our anti-unification algorithm. Constraint generalization in the numerical domain does not need such modification.

We extend the notion of constraint generalization:

DEFINITION 4.11. *Substitution U and solved form formula $\exists\bar{\alpha}.A$ are an answer to* abductive constraint generalization *problem $\overline{D_i}$ given a quantifier prefix $\mathcal{Q}$ when:*

1. $(\forall i)\mathcal{M} \vDash U(D_i) \Rightarrow \exists\bar{\alpha}\backslash\mathrm{FV}(U).A$;
2. *If $\alpha \in \mathrm{Dom}(U)$, then $(\exists\alpha) \in \mathcal{Q}$ – variables substituted by U are existentially quantified;*
3. $(\forall i)\mathcal{M} \vDash \forall(\mathrm{Dom}(U))\exists(\mathrm{FV}(D_i)\backslash\mathrm{Dom}(U)).D_i$.

The sort-integrating algorithm changes only minimally. We adapt the anti-unification algorithm to compute the substitution U from Definition 4.11. For details of the algorithms, see Section B.3.3.

4.4. Negative Constraints

We collect implication branches with conclusion $C_i = \boldsymbol{F}$ and do not pass them to abduction or constraint generalization.

For the numerical sort, optionally but by default, we try to turn the negative constraint $D_i^k \Rightarrow \boldsymbol{F}$ into a positive contribution to constraints under assumption that the numerical domain is integer numbers rather than rationals. We convert $\neg D_i^k$ into disjunctive normal form, and replace strict inequalities $w > 0$ with weak inequalities $-w \leq -1$, assuming w.l.o.g. integer coefficients in w. We eliminate each disjunct inconsistent with some branch $D_j^k \wedge C_j^k$. If exactly one disjunct remains, it is the solution to the negative constraint $D_i^k \Rightarrow \boldsymbol{F}$.

After a joint constraint abduction answer has been found, we check whether all negated constraints, i.e. D_i^k for $C_i^k = \boldsymbol{F}$, are inconsistent. If not, we backtrack: redo abduction with A^k put into the discard list. Moreover, we include in simple constraint abduction a heuristic to prefer partial answers which make more negated constraints inconsistent.

For details of the algorithm, see Section B.4.

4.5. Details of Solving for Predicate Variables

Faced with an inference problem Φ, we solve it using iterated abduction. Let

$$\exists\delta\Phi \equiv \exists\delta\mathcal{Q}.\, \wedge_i (D_i \Rightarrow C_i) = \exists\delta\mathrm{NF}(\Phi)$$

where D_i, C_i are conjunctions of atoms, be quantifier alternation-minimizing normalization of Φ (the variable δ is used just to bring existentially quantified variables to the front, if possible). Using the prenex-normal form simplifies formal presentation.

Definition 4.12. *We will say that a quantifier prefix and a solved form formula $\exists\bar{\alpha}.A$ are in* atomized form with respect to parameters $\bar{\beta}$*, when: variables $\bar{\beta}$ are in the same quantifier alternation in $\mathcal{Q}$, $\mathcal{M}\models\mathcal{Q}.A[\bar{\alpha}\bar{\beta}:=\bar{t}]$ for some $\bar{t}$, and the following two conditions hold:*

1. *there are no properties expressed in A in a way constraining parameters $\bar{\alpha}\bar{\beta}$ that can also be expressed without constraining them: if there are conjunctions of atoms C, D such that $\mathcal{M}\models D\wedge C\Rightarrow A$ and $\mathcal{M}\models\mathcal{Q}.D$, then there exists a conjunction of atoms B such that $B\subset A$, $\mathcal{M}\models\mathcal{Q}.B$ and $\mathcal{M}\models C\Rightarrow(A\backslash B)$, where $A\backslash B$ is a conjunction of atoms in A that are not in B;*
2. *all properties expressed in A in a way constraining parameters $\bar{\alpha}\bar{\beta}$ are also expressed using only variables $\mathcal{Q}_{<\beta}$ (for any $\beta\in\bar{\beta}$) and $\bar{\alpha}\bar{\beta}$: if there is a conjunction of atoms $C\subset A$ such that $\mathcal{M}\not\models\mathcal{Q}[(\forall\bar{\beta}):=(\forall\bar{\alpha}\bar{\beta})].C$, then there is a conjunction of atoms $D\subset A$ such that $\mathrm{FV}(D)\subset\mathcal{Q}_{<\beta}\bar{\alpha}\bar{\beta}$ and $\mathcal{M}\models\mathcal{Q}[(\forall\bar{\beta}):=(\forall\bar{\alpha}\bar{\beta})].D\Rightarrow C$.*

The atomized form is with respect to parameters $\overline{\bar{\beta}^{\chi}}$ *when it is an atomized form with respect to parameters $\bar{\beta}^{\chi}$ for all $\bar{\beta}^{\chi}\in\overline{\bar{\beta}^{\chi}}$.*

Proposition 4.13. *Let $\mathcal{Q}.\bar{\alpha}_1{\doteq}\bar{t}_1\wedge\bar{\alpha}_2{\doteq}\bar{t}_2$ be a formula in $\mathcal{L}$ with sorts s_{type} and the linear arithmetic sort. If $[\bar{\alpha}_1:=\bar{t}_1;\bar{\alpha}_2:=\bar{t}_2]$ is a substitution, $[\bar{\alpha}_1:=\bar{t}_1]$ agrees with quantifier prefix $\mathcal{Q}$ and $\bar{\alpha}_2\subseteq\bar{\alpha}\bar{\beta}$, then $\exists\bar{\alpha}.\bar{\alpha}_1{\doteq}\bar{t}_1\wedge\bar{\alpha}_2{\doteq}\bar{t}_2$ is in atomized form w.r.t. parameters $\bar{\beta}$.*

Symbolically we denote an atomized form that is equivalent to A by $\mathrm{Atomized}(A)$. For inequalities, $\mathrm{Atomized}(A)$ can be recovered during the Fourier-Motzkin procedure by keeping some of the inequalities implied by A. Atomized forms are required so that we can distribute the constraints among the predicate variables – implicit constraints are made explicit.

Take a conjunction of atoms $A\in\mathcal{L}$, a quantifier prefix $\mathcal{Q}$, and variables $\bar{\alpha}$, $\overline{\bar{\beta}^{\chi}}$, where χ varies over $\mathrm{PV}(\mathcal{Q})$. Let us discuss the routine $\mathrm{Split}\big(\mathcal{Q},\bar{\alpha},A,\overline{\bar{\beta}^{\chi}},\overline{A^0_{\chi}}\big)$ defined in Table 4.3. $\mathrm{PrimCV}(c)$ stands for *primary constrained variables* of an atom c. These are the variables that can be separated to one side of the atom c. For the sort of terms and the numerical sort, $\alpha\in\mathrm{PrimCV}(c)$ if and only if: $c\Leftrightarrow\alpha{\doteq}t$, or $c\Leftrightarrow\alpha\leqslant t$, or $c\Leftrightarrow t\leqslant\alpha$, for some t including $t=\min(t_1,t_2)$ and $t=\max(t_1,t_2)$ for some t_1,t_2. For other sorts, the definition of $\mathrm{PrimCV}(\cdot)$ needs to be extended analogically. The selections of $\overline{A^+_{\chi}}$ we try are minimal with respect to pointwise set inclusion, i.e. wrt. $\leqslant$ defined as $\overline{A^1_{\chi}}\leqslant\overline{A^2_{\chi}}\Leftrightarrow\forall\chi.A^1_{\chi}\subseteq A^2_{\chi}$. The routine Split separates an answer formula into components that need to become (parts of) the predicate variable solutions, and the residual answer A_{res}. The residual answer is a satisfiable formula, it contains solutions to the existentially quantified variables. In particular, the types of expressions of interest can be extracted from the final residual answer. Note that due to existential types predicates, we actually compute $\mathrm{Split}\big(\mathcal{Q},\bar{\alpha},A,\overline{\bar{\beta}^{\beta_{\chi}}},\overline{A^0_{\beta_{\chi}}}\big)$, i.e. we index by β_{χ} (which can be multiple for a single χ) rather than χ. We retain the notation indexing by χ as it better conveys the intent. Let $\mathrm{Split}\big(\mathcal{Q},\bar{\alpha},A,\overline{\bar{\beta}^{\chi}}\big)$ be $\mathrm{Split}\big(\mathcal{Q},\bar{\alpha},A,\overline{\bar{\beta}^{\chi}},\bar{\boldsymbol{T}}\big)$.

let

$$
\begin{array}{rcl}
\alpha \prec \beta & = & \alpha <_{\mathcal{Q}} \beta \vee \big(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not<_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar{\beta}^{\chi}} \wedge \beta \notin \overline{\bar{\beta}^{\chi}}\big) \\
A_{\alpha\beta} & = & \big\{\beta \doteq \alpha \in A \big| \, \beta \in \overline{\bar{\beta}^{\chi}} \wedge (\exists \alpha) \in \mathcal{Q} \wedge \beta \prec \alpha\big\} \\
A_0 & = & A \backslash A_{\alpha\beta} \\
A^1_{\chi} & = & \big\{c \in A_0 \big| \forall \alpha \in \mathrm{FV}(c).(\exists \alpha) \in \mathcal{Q} \vee \\
 & & \quad \alpha <_{\mathcal{Q}} \beta_{\chi} \wedge \alpha \notin \mathrm{PrimCV}(c) \vee \alpha \in \overline{\bar{\beta}^{\chi}} \wedge \alpha \in \mathrm{PrimCV}(c)\big\} \\
A^2_{\chi} & = & \mathrm{Atomized}\big(\overline{\bar{\beta}^{\chi}}, A^1_{\chi}\big) \\
A^3_{\chi} & = & A^2_{\chi} \backslash \cup_{\chi'} A^1_{\chi'}
\end{array}
$$

if $\mathcal{M} \not\models \mathcal{Q}.(A \setminus \cup_{\chi} A^2_{\chi})[\bar{\alpha} := \bar{t}]$ for all $\bar{t}$

then return $\bot$

for all $\overline{A^+_{\chi}}$ min. w.r.t. $\subset$ s.t. $\wedge_{\chi}(A^+_{\chi} \subset A^2_{\chi})$ and $\mathcal{M} \models \mathcal{Q}.(\cup_{\chi} A^+_{\chi} \Rightarrow A)[\bar{\alpha} := \bar{t}]$ for some $\bar{t}$:

if $\mathrm{Strat}(A^+_{\chi}, \bar{\beta}^{\chi})$ returns $\bot$ for some χ

then return $\bot$

else let

$$
\begin{array}{rcl}
\bar{\alpha}^{\chi}_{+}, A^L_{\chi}, A^R_{\chi} & = & \mathrm{Strat}(A^+_{\chi}, \bar{\beta}^{\chi}) \\
A_{\chi} & = & A^0_{\chi} \cup A^L_{\chi} \\
\bar{\alpha}^{\chi}_0 & = & \bar{\alpha} \cap \mathrm{FV}(A_{\chi}) \\
\bar{\alpha}^{\chi} & = & \left(\bar{\alpha}^{\chi}_0 \setminus \bigcup_{\chi' <_{\mathcal{Q}} \chi} \bar{\alpha}^{\chi'}_0\right) \bar{\alpha}^{\chi}_{+} \\
A_{+} & = & \cup_{\chi} A^R_{\chi} \\
A_{\mathrm{res}} & = & A_{+} \cup \widetilde{A_{+}}(A \setminus \cup_{\chi} A^+_{\chi})
\end{array}
$$

if $\cup_{\chi} \bar{\alpha}^{\chi} \neq \varnothing \vee \cup_{\chi} A^3_{\chi} \neq \varnothing$

then

$\mathcal{Q}', A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}'^{\chi}.A'_{\chi}} \in$

$\mathrm{Split}\big(\mathcal{Q}\big[\overline{\forall \bar{\beta}^{\chi}} := \overline{\forall(\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi})}\big], \bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}, A_{\mathrm{res}} \wedge_{\chi} A^3_{\chi}, \overline{\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi}}, \overline{A_{\chi}}\big)$

return $\mathcal{Q}', A_{\alpha\beta} \wedge A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\chi} \bar{\alpha}'^{\chi}.A'_{\chi}}$

else return $\mathcal{Q} \exists(\bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}), A_{\alpha\beta} \wedge A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\chi}.A_{\chi}}$

where $\mathrm{Strat}(A, \bar{\beta}^{\chi})$ is computed as follows:

1. for every $c \in A$, and for every $\beta_2 \in \mathrm{FV}(c)$ such that $\beta_1 <_{\mathcal{Q}} \beta_2$ for $\beta_1 \in \bar{\beta}^{\chi}$,
2. if β_2 is universally quantified in $\mathcal{Q}$, then return $\bot$;
3. otherwise, introduce a fresh variable α_f, replace $c := c[\beta_2 := \alpha_f]$,
4. add $\beta_2 \doteq \alpha_f$ to A^R_{χ} and α_f to $\bar{\alpha}^{\chi}_{+}$,
5. after replacing all such β_2 add the resulting c to A^L_{χ}.

Table 4.3. Split routine $\mathrm{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^{\chi}}, \overline{A_{\chi}}\big)$

In Table 4.4, we describe a single step of solving for predicate variables. The iteration of the algorithm starts by substituting the partial solutions to predicate variables from the previous iteration (Table 4.4, Eq. 4.1). Next, it performs constraint generalization, to find the current solutions for postcondition-related predicate variables (Eq. 4.2). The found postconditions are then substituted for predicate variables in premises, i.e. at places where a definition returning an existential type is used recursively (Eq. 4.3). Having stronger premises simplifies, and sometimes enables, the subsequent abduction (Eq. 4.4). The abduction answer is split into parts (Eq. 4.5), that are added to the partial solutions to predicate variables

(Eqs. 4.6, 4.8). When abduction shows we need richer postconditions, an auxiliary round of abduction is performed (Eq. 4.7). It infers additional preconditions that enable the required postconditions. We add these preconditions to the partial solutions (Eq. 4.8). If the resulting partial solutions differ from the initial partial solutions, we continue with another iteration of the main algorithm.

Recall Definition 3.5 of solutions to a type inference problem Φ.

Proposition 4.14. *If a solution to the inference problem Φ exists, then there is an interpretation of predicate variables $\mathcal{I}$ such that $\mathcal{M},\mathcal{I}\vDash\Phi$.*

Define

$$\Psi_0=\{(\boldsymbol{T},\bar{\boldsymbol{T}},\bar{\boldsymbol{T}})\},\quad \Psi_{k+1}=\bigcup_{\left(F^k_{\text{res}},\overline{\exists\bar\alpha^{\chi,k}.F^k_\chi},\overline{\exists\bar\alpha^{\chi_K,k}.F^k_{\chi_K}}\right)\in\Psi_k}\Psi\left(k,\Phi,\overline{\exists\bar\alpha^{\chi,k}.F^k_\chi},\overline{\exists\bar\alpha^{\chi_K,k}.F^k_{\chi_K}}\right)$$

where the operator Ψ such that $(\exists\bar\alpha_{\text{res}}.A_{\text{res}},S_{k+1},R_{k+1})\in\Psi(k,\Phi,S_k,R_k)$ for $S_k=\overline{\exists\bar\alpha^{\chi,k}.F^k_\chi}$, $R_k=\overline{\exists\bar\alpha^{\chi_K,k}.F^k_{\chi_K}}$, is defined in Table 4.4. The convergence condition is:

$$\begin{aligned}&(\forall\chi)S_{k+1}(\chi)\subseteq S_k(\chi),\\ \wedge\ &(\forall\chi_K)R_{k+1}(\chi_K)=R_k(\chi_K),\\ \wedge\ &(\forall\beta_{\chi_K})A_{\beta_{\chi_K}}=\boldsymbol{T},\\ \wedge\ &k>1\end{aligned}$$

We argue for the truth of the following theorem in Section A.4.

Theorem 4.15. Correctness. *Let $\mathcal{I}=[\bar\chi:=\overline{\exists\bar\alpha_\chi.F_\chi};\overline{\chi_K}:=\overline{\exists\bar\alpha_K.F_K}]$ be an interpretation of predicate variables for an inference problem Φ. If $(\exists\bar\alpha'_{\text{res}}.F'_{\text{res}},\overline{\exists\bar\alpha^\chi.F_\chi},\overline{\exists\bar\alpha^{\chi_K}.F_{\chi_K}})\in\Psi(\Phi,\overline{\exists\bar\alpha^\chi.F_\chi},\overline{\exists\bar\alpha^{\chi_K}.F_{\chi_K}})$, then $\mathcal{M},\mathcal{I}\vDash\Phi$.*

To prove Theorem 4.16, we have to assume that there are no existential type predicate variables, i.e. for an inference problem Φ, $\text{PV}(\Phi)=\text{PV}^1(\Phi)$; and also that Abd in the definition of Ψ is a complete abduction algorithm returning an atomized form formula. Since the completeness of abduction assumption is not met, Theorem 4.16 does not describe an actual implementation. We argue for the truth of Theorem 4.16 in Section A.4.

Theorem 4.16. *Let $(\exists\bar\alpha^{\text{res}}_s.F^s_{\text{res}},\overline{\exists\bar\alpha^\chi_s.F^s_\chi})$ be any solution to an inference problem Φ with $\bar\chi=\text{PV}(\Phi)=\text{PV}^1(\Phi)$. Assume that* Abd *in the definition of Ψ is a complete abduction algorithm returning an atomized form formula. Then there is a chain $\left(\exists\bar\alpha^{\text{res}}_k.F^k_{\text{res}},\overline{\exists\bar\alpha^{\chi,k}.F^k_\chi}\right)\in\Psi_k$, with $\left(\overline{\exists\bar\alpha^{\chi,k+1}.F^{k+1}_\chi}\right)\in\Psi(k,\Phi,\overline{\exists\bar\alpha^{\chi,k}.F^k_\chi})$, such that for all $k\geqslant 1$, $\mathcal{M}\vDash\wedge_\chi F^s_\chi\Rightarrow(\wedge_\chi F^k_\chi)\left[\overline{\bar\alpha^{\chi,k}}:=\bar t\right]$ for some $\bar t$.*

For more discussion on the implementation, see Section B.6.

$$
\begin{array}{rcll}
\overline{\exists\bar{\beta}^{\chi,k}.F_\chi} & = & S_k & \\
 & & \text{In iteration 2, remove non-term-sort atoms} & \\
 & & \text{containing outer-scope parameters from } S_k. & \\
D_K^\alpha \Rightarrow C_K^\alpha \in R_k^- S_k(\Phi) & = & \text{all such that } \chi_K(\alpha,\alpha_\alpha^K)\in C_K^\alpha, & (4.1)\\
 & & \overline{C_j^\alpha} = \{C \,|\, D\Rightarrow C\in S_k(\Phi)\wedge D\subseteq D_K^\alpha\} & \\
\underline{\alpha_K} & : & \chi_K(\alpha_K) \text{ is a positive atom in } \Phi & \\
U_{\chi_K}, \exists\bar{\alpha}_g^{\chi_K}.G_{\chi_K}, \overline{B_d^K} & = & \mathrm{LUB}\big(\alpha_K, \overline{\delta\doteq\alpha\wedge D_K^\alpha\wedge_j C_j^\alpha}_{\alpha\in\overline{\alpha_3^{i,K}}}\big) & (4.2)\\
\vec{\tau}_{\varepsilon_K} & = & \overrightarrow{\big\{\alpha\in\mathrm{FV}(G_{\chi_K})\backslash\delta\delta'\bar{\alpha}_g^{\chi_K}\big|(\exists\alpha)\in\mathcal{Q}\vee\alpha\in\overline{\bar{\beta}^\chi}\backslash\bar{\beta}^{\chi_K}\big\}} & \\
\Xi(\exists\bar{\alpha}_g^{\chi_K}.G_{\chi_K}) & = & \exists\mathrm{FV}(\vec{\tau}_{\varepsilon_K}, G_{\chi_K})\backslash\delta\delta'.\delta'\doteq\vec{\tau}_{\varepsilon_K}\wedge G_{\chi_K} & \\
(\exists\bar{\alpha}^{\chi_K}.F_{\chi_K}), \rho & = & H(R_k(\chi_K), \Xi(\exists\bar{\alpha}_g^{\chi_K}.G_{\chi_K})) & \\
R_g(\chi_K) & = & \exists\bar{\alpha}^{\chi_K}.F_{\chi_K} & \\
P_g(\chi_K(\delta,\delta')) = P_g(\chi_K(\delta')) & = & \delta'\doteq\vec{\tau}_{\varepsilon_K} & \\
F'_\chi & = & \underline{F_\chi[\varepsilon_K(\vec{\tau}_{\mathrm{old}}) := \varepsilon_K(\vec{\tau}_{\varepsilon_K})[\mathrm{recover}(\vec{\tau}_{\varepsilon_K}, \vec{\tau}_{\mathrm{old}})]]} & \\
S'_k & = & \exists\bar{\beta}^{\chi,k}\{\alpha\in\mathrm{FV}(F'_\chi)\,|\,\beta_\chi <_{\mathcal{Q}}\alpha\}.F'_\chi & \\
\mathcal{Q}'.\wedge_i(D_i\Rightarrow C_i)\wedge_j(D_j^-\Rightarrow \boldsymbol{F}) & = & R_g^- P_g^+ S'_k(\Phi\wedge_{\chi_K} U_{\chi_K}) & (4.3)\\
\exists\bar{\alpha}.A_0 & = & \mathrm{Abd}\big(\mathcal{Q}', \bar{\beta}=\beta_\chi\bar{\beta}^\chi, \overline{D_i, C_i}\big) & (4.4)\\
A & = & A_0\wedge_j \mathrm{NegElim}\big(\neg\mathrm{Simpl}\big(\mathrm{FV}(D_j^-)\backslash\overline{\bar{\beta}^\chi}.D_j^-\big), A_0, \overline{D_i, C_i}\big) & \\
 & & \text{In later iterations, check negative constraints.} & \\
\big(\mathcal{Q}^{k+1}, A_{\mathrm{res}}, \overline{\exists\bar{\alpha}^{\beta_\chi}.A_{\beta_\chi}}\big) & = & \mathrm{Split}\big(\mathcal{Q}', \bar{\alpha}, A, \overline{\beta_\chi\bar{\beta}^\chi}\big) & (4.5)\\
\vec{\tau}'_{\varepsilon_K} & = & \overrightarrow{\mathrm{FV}(\widetilde{A_{\mathrm{res}}}(\vec{\tau}_{\varepsilon_K}))} & \\
R_{k+1}(\chi_K) & = & \exists\bar{\beta}^{\chi_K,k}\bar{\alpha}^{\beta_{\chi_K}}\bar{\alpha}^{\chi_K}\backslash\mathrm{FV}(\vec{\tau}'_{\varepsilon_K}).\delta'\doteq\vec{\tau}'_{\varepsilon_K}\wedge\widetilde{A_{\mathrm{res}}}(F_{\chi_K}\backslash\delta'\doteq\ldots) & \\
 & & \quad\wedge_{\chi_K} A_{\beta_{\chi_K}}\Big[\overline{\beta_{\chi_K}\bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta\bar{\beta}^{\chi_K,k}}\Big] & (4.6)\\
A_K^d & = & \Big\{c\in A_{\beta_{\chi_K}}\Big[\overline{\beta_{\chi_K}\bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta\bar{\beta}^{\chi_K,k}}\Big]\Big| & \\
 & & \quad\mathrm{FV}(c)\subseteq\mathrm{FV}(\rho(B_d^K))\Big\} & \\
\exists\bar{\alpha}'_K.A'_K & = & \mathrm{Abd}\big(\mathcal{Q}', \overline{\beta_\chi\bar{\beta}^\chi}\backslash\overline{\beta_{\chi_K}\bar{\beta}^{\chi_K}}, \overline{\rho(B_d^K), A_K^d}\big) & (4.7)\\
\Big(\mathcal{Q}^{k+1}, \varnothing, \overline{\exists\bar{\alpha}^{\beta_{\chi'}}.A'_{\beta_\chi}}\Big) & = & \mathrm{Split}\big(\mathcal{Q}^{k+1}, \overline{\bar{\alpha}'_K}, \wedge_K A'_K, \overline{\beta_\chi\bar{\beta}^\chi}\backslash\overline{\beta_{\chi_K}\bar{\beta}^{\chi_K}}\big) & \\
S_{k+1}(\chi) & = & \exists\bar{\beta}^{\chi,k}.\mathrm{Simpl}\big(\exists\overline{\bar{\alpha}^{\beta_\chi}}.F'_\chi & \\
 & & \quad\wedge A_{\beta_\chi}\big[\overline{\beta_\chi\bar{\beta}^\chi} := \overline{\delta\bar{\beta}^{\chi,k}}\big]\wedge A'_{\beta_\chi}\big) & (4.8)
\end{array}
$$

Table 4.4. Iteration k of solving for predicate variables

Chapter 5
InvarGenT: Tests and Limitations

In this chapter we present types and inference times of tested examples, measured on a laptop with *Intel Core i3* processor at 2.30GHz, 3MB cache. The implementation does not use parallelism.

5.1. Benchmarks

Table 5.1 contains examples using the plain GADTs type system, without numerical constraints and existential types. Consider the examples from Chuan-kai Lin's [22]. We tested 7 longer examples where algorithm $\mathcal{P}$ finds the expected type. InvarGenT does not have problems with these cases either, except the `head` function. We also reproduce all 8 examples from [22] where algorithm $\mathcal{P}$ fails, adapting them only to accomodate to the type system MMG(X) behind InvarGenT. InvarGenT fails on two examples. One, the function `vary`, was contrived to not have a clear intended type. The other, the function `fd_comp`, can be slightly modified to work with InvarGenT's type inference. There is also one example, function `leq`, which requires non-default parameter setting (passing a parameter `-prefer_guess` to InvarGenT). In a future version, InvarGenT will attempt multiple parameter settings before giving up. Overall, we consider the behavior of InvarGenT on this test suite to be a success.

Table 5.2 contains examples using numerical constraints and existential types. The examples are organized around several data structures: lists and arrays with length, binary numbers, AVL balanced search trees. We provide the inferred types in the hope that they are self-explanatory.

Table 5.3 contains examples of static array (and matrix) bounds checking. These tests originally date back to the Hongwei Xi's DML system. They were selected, by [41], to demonstrate the abilities of the Liquid Types system inference implementation DSolve. The reported DSolve times come from [41] and [40]. The example `isort` lists two times: the shorter time is inferred from a program without an unused nested definition of `vecswap`. We extracted the corresponding computation as the example `swap_interval` in Table 5.2. The shorter time for the example `simplex` corresponds to a variant where some helper functions are separated out as toplevel definitions, while the longer time is for a variant with a single toplevel definition. For the example program `gauss` without assertions we again have two variants, but they both simplify the original example. They remove a redundant level of nesting in a helper, nested definition `rowMax`. One of the variants requires passing a non-default option `-prefer_bound_to_local` to guide inference, the other slightly generalizes the algorithm by passing a different matrix dimension in two places. Since our type system does not allow existential types as function arguments, for the FFT examples we introduce an explicit datatype `Bounded`. The FFT example with assertion requires non-default option `-same_with_assertions`. On the whole, we believe that the approach taken by InvarGenT shows promise in this comparison. It is clear that InvarGenT faces increasing difficulties

Class of examples	*Function name: inferred type*	*Inf. time*
incompl. ex. [53]	`rx: ∀a.R a→Int`	<0.01s
examples from [23] within the scope of algorithm $\mathcal{P}$	`rotate: ∀a.Dir→Int→RoB (Black, a)→Dir→Int→ RoB (Black, a)→RoB (Red, a)→RoB (Black, S a)`	0.02s
	`zip2: ∀a, b.Zip2 (B, a)→b→a`	0.16s
	`rotl: ∀a.AVL a→Int→AVL (S (S a))→ Choice (AVL (S (S a)), AVL (S (S (S a))))`	0.03s
	`ins: ∀a.Int→AVL a→Choice (AVL a, AVL (S a))`	0.41s
	`extract: ∀a, b.Path b→Tree (b, a)→a`	0.06s
	`run_state: ∀a, b.b→State (b, a)→(b, a)`	0.01s
	`head: ∀a, b.List (a, S b)→a`	∞
examples from [23] outside the scope of algorithm $\mathcal{P}$	`joint: ∀a.Split (a, a)→a`	<0.01s
	`rotr: ∀a.Int→AVL a→AVL (S (S a))→ Choice (AVL (S (S a)), AVL (S (S (S a))))`	0.09s
	`delmin: ∀a.AVL (S a)→ (Int, Choice (AVL a, AVL (S a)))`	0.31s
	`fd_comp: ∀a, b, c.FunDesc (c, b)→FunDesc (b, a)→ FunDesc (c, a)`	0.2s*, 0.1s*
	`zip1: ∀a, b.Zip1 (List b, a)→b→a`	0.08s
	`leq: ∀a.Nat a→NatLeq (a, a)`	<0.01s**
	`run_state: ∀a, b.b→State (b, a)→(b, a)`	0.03s
run-time type representations	`eval: ∀a.Term a→a`	0.06s
	`equal: ∀a,b.(Ty a, Ty b)→a→b→Bool`	0.3s, 2.1s

* Slight meaning-preserving modification of test program
** Needs a non-default option `-prefer_guess`

Table 5.1. Examples of types and inference times, plain GADTs

Class of examples	*Function name: inferred type*	*Inf. time*
lists with length and arrays with static bound checks	`head: ∀n,a[1⩽n].List (a, n)→a`	<0.01s
	`append: ∀a,n,k.List (a, n)→List (a, k)→List(a, n+k)`	0.02s
	`flatten_pairs: ∀n, a. List ((a, a), n) → List (a, 2 n)`	0.01s
	`flatten_quadrs: ∀n, a. List ((a, a, a, a), n) → List (a, 4 n)`	0.06s
	`filter: ∀n, a. (a → Bool) → List (a, n) → ∃k[0 ⩽ k ∧ k ⩽ n].List (a, k)`	0.18s
	`zip: ∀a,b,n,k.(List (a,n), List (b,k))→ ∃i[i=min(n, k)].List ((a,b),i)`	0.38s
	`bsearch2: ∀n, a[0 ⩽ n]. a → Array (a, n) → ∃k[0 ⩽ k + 1 ∧ k + 1 ⩽ n].Num k` (uses assertion)	1.39s
	`swap_interval: ∀i, j, k, n, a[1 ⩽ j ∧ 0 ⩽ n ∧ 0 ⩽ k ∧ i + k ⩽ j ∧ i + n ⩽ j]. Array (a, j) → Num n → Num k → Num i → ()`	0.27s
	`matmul: ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ j ∧ j ⩽ i]. Matrix (n,j) → Matrix (i,k) → Matrix (n,k)`	0.34s
binary numbers	`plus: ∀n,k,i.Carry i→Binary k→Binary n→Binary (n+k+i)`	0.66s
	`increment: ∀n.Binary n→Binary (n + 1)`	0.01s
	`bitwise_or: ∀k, n. Binary k → Binary n → ∃i[k ⩽ i ∧ n ⩽ i ∧ i ⩽ n + k].Binary i`	1.21s
AVL trees, imbalance of 2	`create: ∀k, n, a[0 ⩽ n ∧ 0 ⩽ k ∧ n ⩽ k + 2 ∧ k ⩽ n + 2]. Avl (a, k) → a → Avl (a, n) → ∃i[i=max (k + 1, n + 1)].Avl (a, i)`	0.09s
	`rotr: ∀k, n, a[0 ⩽ n ∧ n + 2 ⩽ k ∧ k ⩽ n + 3]. Avl (a, k) → a → Avl (a, n) → ∃n[k ⩽ n ∧ n ⩽ k + 1].Avl (a, n)`	1.07s
	`add: ∀n, a. a → Avl (a, n) → ∃k[1 ⩽ k ∧ n ⩽ k ∧ k ⩽ n + 1].Avl (a, k)`	0.69s
	`remove_min_binding: ∀n, a[1 ⩽ n]. Avl (a, n) → ∃k[n ⩽ k + 1 ∧ k ⩽ n ∧ k + 2 ⩽ 2 n].Avl (a, k)`	0.59s
	`merge: ∀k, n, a[n ⩽ k + 2 ∧ k ⩽ n + 2]. (Avl (a, n), Avl (a, k)) → ∃i[n ⩽ i ∧ k ⩽ i ∧ i ⩽ n + k ∧ i⩽max (k + 1, n + 1)].Avl (a, i)`	0.93s
	`remove: ∀n, a. a → Avl (a, n) → ∃k[n ⩽ k + 1 ∧ 0 ⩽ k ∧ k ⩽ n].Avl (a, k)`	0.38s
	Total time for `add`, `remove` and helper functions:	4.92s

Table 5.2. Examples of types and inference times, with numerical constraints and existentials

when analysing more complex programs. We propose remedies to the issues encountered as future work.

Program	*Inf. time*	*Time from [41]*
`dotprod`	0.05s	0.31s
`bcopy`	0.03s	0.15s
`bsearch`	0.07s	0.46s
`queen`	0.42s	0.7s
`isort`	0.3s, 0.37s	0.88s
`tower` no assertions	0.84s	∞
`tower` with assertion	3.93s	3.33s
`matmult`	0.34s	1.79s
`heapsort`	2.34s	0.53s
`fft` no assertions	36.4s*	?
`fft` with assertion	37.5s*,**	9.13s
`simplex`	8.1s*, 31.4s	7.73s
`gauss` no assertions	2.66s*, 1.02s*,***	?
`gauss` with assertion	2.72s	3.17s

* Slight meaning-preserving modification of test program
** Needs a non-default option `-same_with_assertions`
*** Needs a non-default option `-prefer_bound_to_local`

Table 5.3. Examples of inference times, bound checking

The code of examples from Tables 5.1, 5.2 and 5.3 can be found in Appendix C.

5.2. InvarGenT Failure Cases

Type inference for the type system underlying InvarGenT is undecidable. Constraint abduction itself is not known to be decidable. Rather than trying to enumerate all possible answers, we devised constraint abduction algorithms that only consider answers of restricted forms. The `vary` example from [22] fails due to such limitation. However, the practical problems we encountered are not of such nature. Each subsection below describes a class of problematic cases, exhausting the types of inference failures we encountered. Near discuss the prospects of improving the situation.

5.2.1. The Need to Expand Pattern Variables

The one function from [22] that needed modification to be type-inferred is `fd_comp`. Take a look at the fragment of original definition that needed modification:

```
let fd_comp = fun fd1 fd2 ->
  let o = fun f g x -> f (g x) in
  match fd1 with
    | FDI -> fd2
    | FDC b -> ...
```

The modified form:

```
let fd_comp = fun fd1 fd2 ->
  let o = fun f g x -> f (g x) in
  match fd1 with
    | FDI ->
      (match fd2 with | FDI -> fd2 | FDC _ -> fd2 | FDG _ -> fd2)
    | FDC b -> ...
```

Type inference needs to know the structure of both arguments to be able to infer the resulting type. If this problem proves cumbersome, we can devise heuristics for automatic expansion of pattern variables.

5.2.2. Constraints Shared by Constructors of a Datatype

The above problem is more pronounced in the numerical domain. Fortunately, here it can have a principled solution. Consider the following list appending example that does not type-check:

```
let rec append =
  function LNil -> (fun l -> l)
    | LCons (x, xs) -> (fun l -> LCons (x, append xs l))
```

We can mitigate the problem by expanding the pattern, as in Example 5.1:

```
let rec append =
  function LNil -> (function LNil -> LNil | LCons (_,_) as l -> l)
    | LCons (x, xs) ->
      (function LNil -> LCons (x, append xs LNil)
        | LCons (_,_) as l -> LCons (x, append xs l))
```

But we can also provide the "hidden" information explicitly, by either a positive assertion, or a negative assertion as below:

```
let rec append =
  function
    | LNil ->
      (function l when (length l + 1) <= 0 -> assert false | l -> l)
    | LCons (x, xs) ->
      (function l when (length l + 1) <= 0 -> assert false
      | l -> LCons (x, append xs l))
```

One can investigate a modification to the type system so that each datatype is associated with an invariant common to all constructors of the datatype. In case of lists, the shared invariant is that the length of a list is non-negative. In the modified type system, when it becomes determined that a pattern variable's type is a datatype, the corresponding invariant is made available for inference.

5.2.3. Negative Constraints in the Sort of Terms

Example 5.1. Consider the following variant of the `head` example from [22]:

```
datatype Z
datatype S : type
datatype List : type * num
datacons LNil : ∀a. List(a, Z)
datacons LCons : ∀a, b. a * List(a, b) ⟶ List(a, S b)

let head = function
  | LCons (x, _) -> x
  | LNil -> assert false
```

We introduced the assertion to disallow the overly general type `∀a, b. List (a, b) → a`, which is the correct most general type for `function LCons (x, _) -> x`, because the type system does not enforce exhaustiveness of pattern matching. The above function has type `∀a, b. List (a, S b) → a`, but type inference fails to find it. The problem stems from the inability to express disequations $t_1 \neq t_2$ in the sort of terms as conjunctions of atoms. We already implemented a workaround in InvarGenT for the case of datatypes with only constant constructors.

5.2.4. Insufficient Context to Infer Postconditions

Example 5.2. In the following variant of the `bsearch2` example it turns out to be too hard to infer the full postcondition.

```
datatype Array : type * num
external let array_make :
  ∀n, a [0⩽n]. Num n → a → Array (a, n) = "fun a b -> Array.make a b"
external let array_get :
∀n, k, a [0⩽k ∧ k+1⩽n]. Array (a, n) → Num k → a =
  "fun a b -> Array.get a b"
external let array_length :
  ∀n, a [0⩽n]. Array (a, n) → Num n = "fun a -> Array.length a"
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder
external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"
external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"
```

```
let bsearch key vec =
  let rec look key vec lo hi =
    eif lo <= hi then
        let m = div2 (hi + lo) in
        let x = array_get vec m in
        ematch compare key x with
          | LE -> look key vec lo (m + (-1))
          | GT -> look key vec (m + 1) hi
          | EQ -> eif equal key x then m else -1
    else -1 in
  look key vec 0 (array_length vec + (-1))
```

We get the result type $\exists$`n[0 ⩽ n + 1].Num n` instead of $\exists$`k[k ⩽ n ∧ 0 ⩽ k + 1].Num k`. The inference of the intended type succeeds after we introduce an appropriate assertion, e.g. `assert num -1 <= hi`. Alternatively, we can include a use case for `bsearch` where the full postcondition is required. It would be challenging to guess the assertion or use-case automatically.

5.2.5. Insufficient Search

The need to pass options, in examples like the function `leq` and the function `fft` with assertions, stems from the failure of our search strategy for partial solutions across iterations of the main algorithm (rather than within a single call to abduction). Our search can recover from choices leading to contradictions in the following iteration, but not from choices leading down blind alleys. The problem might be more serious than it appears, because the heuristics (i.e. the default options) have been tuned in favor of our test suite. The solution to the problem is *beam search*: processing a fixed number of partial solutions, those with highest scores. A score measures heuristically the quality of a partial solution: penalizing complexity, rewarding locality of scope of parameters in an atom, etc.

5.2.6. Nested Definitions with Tied Postconditions

The variant of the `gauss` program without assertions and with more nesting illustrates a problem with nested definitions introducing existential types. To arrive at the postcondition of the inner definition, we need to propagate information from the use-site of the outer definition. The postconditions are tied in the sense that the postcondition of the outer definition depends on the postcondition of the inner definition. We need to further investigate the issue to see what can be done.

Chapter 6

Conclusions

In this chapter, we summarize our work in its broader context.

6.1. Related Work

Perhaps Xi and Pfenning [57] can be considered the earliest work on GADTs with invariants over a constraint domain; see Section 2.1. Its numerical constraint language contains *min* and *max* operations enabling e.g. a concise definition of AVL trees datatype as in Chapter 1. Its presentation of existential types is similar to ours. [57] opted to automate existential type elimination by *A-translation*: $A_{\mathrm{tr}}(e_1\, e_2) = (\mathbf{let}\, x = A_{\mathrm{tr}}(e_1)\, \mathbf{in}\, \mathbf{let} y = A_{\mathrm{tr}}(e_2)\, \mathbf{in}\, x\, y)$. Thanks to the modified App rule, performing A-translation during the normalization step ExIntro would make strictly more programs typeable in $\mathrm{MMG}_{\exists}(X)$. The DML language from [57] and its successor the ATS language do not perform type inference for recursive functions.

We base our type system on the $\mathrm{HMG}(X)$ type system from Simonet and Pottier [47]; see Section 2.2. In the tradition of the Milner-Mycroft type system (see Henglein [18]), we drop the type specifications on recursive definitions from program terms. We also naturally restrict $\mathrm{HMG}(X)$ by limiting the user-specified and inferred invariant constraints to use conjunction as the only logical connective. We call the resulting type system $\mathrm{MMG}(X)$.

The traditional framework for loop invariant generation of Cousot and Cousot [10] inspired the iterative aspect of our solver. Mycroft's modification in [33] of algorithm $\mathcal{W}$ to polymorphic recursion is also iterative, but it solves each recursive definition separately. We solve all nested definitions jointly in a single iteration.

Initially we were only aware of the work in Knowles and Flanagan [20] in the framework of *refinement types*, which applies Dijkstra's weakest precondition calculus to general refinement types. An interesting question is whether work similar to ours could be done by application of the weakest precondition calculus to the Hoare logic of Regis-Gianas and Pottier [43], with the conditions inserted by type inference. Work on *Liquid Types* by Rondon, Kawaguchi and Jhala [41] is a continuation of this approach closer to InvarGenT in that it does not use weakest precondition calculus explicitly; see Section 2.5. *Abstract Refinement Types* by Vazou, Rondon and Jhala [52] is a continuation of Liquid Types in an interesting direction beyond the scope of current InvarGenT, opening an area for future work.

Early work on applying abduction to type inference was an inspiration for us. The work by Sulzmann, Schrijvers and Stuckey [49] and [48] closely relates to our work in their application of fully-maximal constraint abduction to GADTs type inference. But without annotations, this would "throw the baby out with the bathwater" by rejecting, as not having an intuitive type, for example any variant of `eval` that computes for pairs: `datacons Pair : ∀a, b. Term a * Term b ⟶ Term (a, b)`, `datacons Fst : ∀a, b. Term (a, b)`

$\longrightarrow$ `Term a` and `datacons Snd : ∀a, b. Term (a, b) ⟶ Term b`. The corresponding implications do not have fully maximal answers. [49] and [48] use Herbrandization but solve the resulting constraints in the free algebra of terms, which we find problematic (see Section 4.2.1).

Maher [27] and Maher and Huang [29] introduce fully maximal constraint abduction; see Section 2.6. Our algorithm for injective terms with multi-sorted *alien subterms* is extensible to any constraint domain, given algorithms for the new domain, as long as the domains do not interact. Allowing the domains to interact would require considerable work, Baader and Schulz [3] might be relevant. Abduction algorithm for the term algebra is provided in [29], although further work driven by practical issues was needed. The linear arithmetic constraint abduction in Maher [26] turned out not to be useful as a basis for an algorithm, our algorithm is novel. Maher [28] studies constraint domains where a (single) most general Simple Constraint Abduction answer exists, and gives a closed formula for SCA answer for the case of Boolean lattices.

The work in Unno and Kobayashi [51] can be seen as extending Knowles and Flanagan [20] 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.

The recent *counterexample-guided abstraction refinement* work of Zhu and Jagannathan [59] (building on Kobayashi, Sato and Unno [21]) pushes the envelope on both expressiveness and efficiency of inference of a refinement types based approach. It is further developed in Zhu, Nori and Jagannathan [60], the implementation is called SpecLearn. The work includes features beyond the scope of InvarGenT, for example effect tracking. Wrt. invariant inference, SpecLearn is similar to InvarGenT in that it starts with coarse invariants and refines them. However, rather than deriving the invariants mostly from the definitions they describe, SpecLearn generates test cases to refine the invariants. It still cannot solve, for example, the AVL trees inference task, without assertions in the source code. SpecLearn is slower than InvarGenT, at least for those of the tests reported in [60] which belong to our test suite (Table 5.3).

Schrijvers and Bruynooghe [44] shares the objective with our work: softening the learning curve and facilitating rapid prototyping. It reconstructs algebraic data types. An interesting direction of future work would be to reconstruct GADTs using the mechanisms already present in InvarGenT. In fact, InvarGenT reconstructs GADT constructors that serve as existential types.

There is a surge of work on type inference for GADTs over the last decade, not contributing to our approach. Works such as Pottier and Régis-Gianas [36] (older), Schrijvers, Peyton Jones, Sulzmann and Vytiniotis [45], Lin and Sheard [23] (see also [22]) modify the GADTs type system to make it more amenable to type inference (rejecting some reasonable programs as untypeable), and develop less declarative inference algorithms. These works also do not allow other domains (than the free term algebra) to express invariants, but Schrijvers, Peyton Jones, Sulzmann and Vytiniotis [53] extends the OutsideIn system to arbitrary domains, see Section 2.3. [22] and [44] stand out from our point of view as they handle type inference for polymorphic recursion: [22] by iteration, see Section 2.4, and [44] using an algorithm by Henglein.

Inductive loop invariants from the recent work Dillig, Dillig, Li and McMillan [12] are closely related to fully maximal joint constraint abduction answers. However, [12] employs a richer language of answers, including implications. We decided to limit solved forms, thus preconditions and postconditions, to conjunctions of atoms, to not tax with complexity both algorithms and users. Exploring [12] in context of InvarGenT may be fruitful, by improving on InvarGenT's abduction algorithm for linear arithmetic. One improvement is to perform quantifier elimination of universal variables (as in [12]) when they cannot be substituted out. An improvement would be constraint abduction that generates *min* and *max* relations, which in current InvarGenT are only introduced by constraint generalization, but here [12] does not help directly. [12] would gain by incorporating ideas from our abduction algorithm.

A related work by Deepak Kapur [19] is part of an interesting research program of Deepak Kapur and Enric Rodríguez Carbonell, for example [19], [38], [39]. [19] generates loop invariants for imperative programs, by quantifier elimination. The variables that are not eliminated are the invariant parameters. In fact, the generated invariant is an abduction answer to the corresponding invariant inference constraint; compare how InvarGenT uses abduction to solve for preconditions. On the other hand, [38] uses constraint generalization, iterating it until convergence. This is in effect how InvarGenT solves for postconditions. [38] and [39] use polynomial equations as the language of invariants. This suggests, as future work, that including in InvarGenT a domain of polynomial equations.

In case of the free algebra of terms, constraint generalization reduces to anti-unification. Anti-unification was first introduced by Plotkin [35] and Reynolds [37]. Bulychev, Kostylev and Zakharov [7] is a recent work on anti-unification, with an example application to invariant inference. Our constraint generalization algorithm for linear inequalities has inspirations from Fukuda, Liebling and Lütolf [16].

6.2. Future Work

Let us collect together proposed directions for future work.

- Implement beam search, where several answers are maintained and inferior answers (less general precondition, less specific postcondition), or answers with worse heuristic score (more complex, etc.), are dropped.
- Address other issues discussed in Section 5.2: handle constraints shared by constructors of a datatype, handle use-site constraint propagation for nested definitions with tied postconditions.
- Explore extending the $\mathrm{MMG}_{\exists}(X)$ type system to handle the use of invariants in arguments of higher-order functions, as in Rondon, Kawaguchi and Jhala [41] and especially in abstract refinement types of Vazou, Rondon and Jhala [52].
- Work on error reporting for InvarGenT. Trace use-site location and program path location associated with implications, separately; see Zhu and Jagannathan [59].
- Integrate InvarGenT with an IDE.
- Add more constraint domains to InvarGenT.

6.3. SUMMARY

We presented 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 provided a minimal extension MMG$_\exists$(X) of this type system that enables inference and easy use of existential types. Although insertions of introduction and elimination of existential types are not automated by the inference process, they are seamlessly integrated into expressions. We demonstrated several use cases using the INVARGENT system.

Our Joint Constraint Abduction under Quantifier Prefix algorithm builds on the fully maximal Simple Constraint Abduction answers algorithm from Maher and Huang [29], extended to guess equalities of parameters. Thanks to aggregating answers during JCA, it can find answers to some cases of joint problems where it would fail to solve some of the implications independently. Our SCA algorithm for linear arithmetic is novel.

Consider programs not using the numerical sort. Undecidability of type inference for polymorphic recursion does not enforce an unbounded number of iteration steps of the main algorithm, because there can be infinitely many (maximally general but not fully maximal) term abduction answers. We encountered examples that need two iteration steps: one to generate a solution, and the following step to discard a wrong solution and accept the correct one. However, with numerical sort constraints, a series of programs can be written needing unbounded number of iteration steps.

We define the Constraint Generalization 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. In the former case, we extend the notion to Abductive Constraint Generalization, providing more specific answers. As we do for abduction, we provide a combination-of-domains algorithm for constraint generalization.

We implemented an algorithm solving for predicate variables of the existential second order constraints generated for our type system. It uses abduction to find requirements on invariants, augmented by constraint generalization to find postconditions.

The inference times in INVARGENT are sufficient for interactive use with toplevel definitions of small-to-moderate size.

Bibliography

[1] Tilak Agerwala and Jayadev Misra. Assertion graphs for verifying and synthesizing programs. Technical Report, University of Texas Technical Report 83, 1978.

[2] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250. New York, NY, USA, 1998. ACM.

[3] Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: combining decision procedures. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 50–65. London, UK, 1992. Springer-Verlag.

[4] Sergey Berezin, Vijay Ganesh and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 521–536. Berlin, Heidelberg, 2003. Springer-Verlag.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.

[6] Edwin Brady, Christoph A. Herrmann and Kevin Hammond. Lightweight invariants with full dependent types. In *Proceedings of the Nineth Symposium on Trends in Functional Programming, TFP 2008, Nijmegen, The Netherlands, May 26-28, 2008.*, pages 161–177. 2008.

[7] 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.

[8] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 90–104. New York, NY, USA, 2002. ACM.

[9] Hubert Comon. Disunification: a survey. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.

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

[11] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid e-unification is undecidable. In Hans Kleine Büning, editor, *Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 178–190. Springer Berlin Heidelberg, 1996.

[12] Isil Dillig, Thomas Dillig, Boyang Li and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 443–456. New York, NY, USA, 2013. ACM.

[13] Cormac Flanagan. Hybrid type checking. *SIGPLAN Not.*, 41(1):245–256, jan 2006.

[14] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD dissertation, University of California, Berkeley, Department of Computer Science, December 2002.

[15] Tim Freeman and Frank Pfenning. Refinement types for ml (extended abstract). In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[16] Komei Fukuda, Thomas M. Liebling and Christine Lütolf. Extended convex hull. In *Proceedings of the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, Canada, August 16-19, 2000*, volume 20, pages 13–23. 2001.

[17] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83. London, UK, UK, 1997. Springer-Verlag.

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

[19] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In Franz Baader, Peter Baumgartner, Robert Nieuwenhuis and Andrei Voronkov, editors, *Deduction and Applications*, number 05431 in Dagstuhl Seminar Proceedings, pages 10–10. Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[20] 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.

[21] Naoki Kobayashi, Ryosuke Sato and Hiroshi Unno. Predicate abstraction and cegar for higher-order model checking. In *ACM SIGPLAN Notices*, volume 46, pages 222–233. ACM, 2011.

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

[23] 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.

[24] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, sep 1994.

[25] Michael Maher. On parameterized substitutions. Technical Report, IBM Research Report RC 16042, 1990.

[26] Michael 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.

[27] 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.

[28] Michael Maher. Heyting domains for constraint abduction. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI'06, pages 9–18. Berlin, Heidelberg, 2006. Springer-Verlag.

[29] 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.

[30] Per Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184. Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[31] Marta Cialdea Mayer and Fiora Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of IGPL*, 1(1):99–117, 1993.

[32] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[33] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228. London, UK, UK, 1984. Springer-Verlag.

[34] Martin Odersky, Martin Sulzmann and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*. 1997.

[35] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[36] 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.

[37] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 5, pages 135–151. Edin-

burgh, Scotland, 1969. Edinburgh University Press.

[38] Enric Rodrguez-Carbonell and Deepak Kapur. Program verification using automatic generation of invariants. In *1st International Colloquium on Theoretical Aspects of Computing (ICTAC'04)*, volume 3407 of *Lecture Notes in Computer Science*, pages 325–340. Springer-Verlag, 2005.

[39] Enric Rodriguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):0, 2007.

[40] Patrick Rondon. *Liquid Types*. PhD dissertation, University of California, San Diego, Department of Computer Science, 2012.

[41] Patrick Maxim Rondon, Ming Kawaguchi and Ranjit Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. 2008.

[42] Patrick Maxim Rondon, Ming Kawaguchi and Ranjit Jhala. Liquid types. Technical Report, 2008.

[43] Yann Régis-Gianas and François 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.

[44] Tom Schrijvers and Maurice Bruynooghe. Polymorphic algebraic data type reconstruction. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 85–96. 2006.

[45] 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.

[46] Tim Sheard. Languages of the future. In *In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press, 2004.

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

[48] Martin Sulzmann, Tom Schrijvers and Peter J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Manuscript, July 2006.

[49] Martin Sulzmann, Tom Schrijvers and Peter J. Stuckey. Type inference via constraint abduction for EADTs. Manuscript, April 2006.

[50] Martin Sulzmann, Tom Schrijvers and Peter J. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Technical Report, K. U. Leuven Dept. of Computer Science, 2008. Report CW 507.

[51] 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.

[52] Niki Vazou, Patrick Maxim Rondon and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228. 2013.

[53] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers and Martin Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

[54] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[55] Hongwei Xi, Chiyan Chen and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235. New York, NY, USA, 2003. ACM.

[56] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 249–257. New York, NY, USA, 1998. ACM.

[57] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. San Antonio, January 1999.

[58] Christoph Zenger. Indexed types. *Theor. Comput. Sci.*, 187(1-2):147–165, nov 1997.

[59] He Zhu and Suresh Jagannathan. Compositional and lightweight dependent type inference for ml. In Roberto Giacobazzi, Josh Berdine and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 295–314. Springer, 2013.

[60] He Zhu, Aditya V. Nori and Suresh Jagannathan. Dependent array type inference from tests. In Deepak D'Souza, Akash Lal and Kim Guldstrand Larsen, editors, *VMCAI '15: Verification, Model Checking and Abstract Interpretation*, volume 8931 of *Lecture Notes in Computer Science*, pages 412–430. Springer, January 2015.

[61] Bjarte M. Østvold. A functional reconstruction of anti-unification. Technical Report, Norwegian Computing Center, Oslo, Norway, 2004.

Appendix A

Proofs

A.1. The Type System

A.1.1. The Logic of Constraints

We work with a first order language $\mathcal{L}_1$ interpreted in a model $\mathcal{M}$, the language of constraints for our type inference problem. We assume that the logic is multi-sorted, but the sorts are combined only via a sort of finite trees, whose leaves can belong to other sorts. I.e., there is a single sort s_{type} whose terms can contain subterms from other sorts and for any its terms $C \Rightarrow f(\bar{s_i}) \doteq g(\bar{t_i})$ implies $f = g$ and $C \Rightarrow \wedge_i s_i \doteq t_i$. The sorts $\mathcal{L}_s, \mathcal{M}_s$ for $s \neq s_{\text{type}}$ are single-sorted logics and $\mathcal{L}_1 = \dot{\cup}_s \mathcal{L}_s$.

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. In the main text, we used R rather than ρ to avoid unnecessary formality. For $\Phi \in \mathcal{L}_1$, 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 $\mathcal{L}_1$ corresponds to predicate π in $\mathcal{M}$, etc.

Of the model $\mathcal{M}$ of $\mathcal{L}_1$ we require the following. For the sort of types s_{type}:

1. *Type conservation.* For any function symbols f, g such that $f \neq g$, and arbitrary $\bar{s}$, $\bar{t}$, there is no interpretation ρ such that $\mathcal{M}, \rho \vDash f(\bar{s}) \doteq g(\bar{t})$.
2. *Free generation.* For any $f \in \{\rightarrow\} \cup \bar{\varepsilon}$ and arbitrary $\bar{s_i}_{1 \leqslant i \leqslant \text{ar}(f)}$, $\bar{t_i}_{1 \leqslant i \leqslant \text{ar}(f)}$, for any interpretation ρ, if $\mathcal{M}, \rho \vDash f(\bar{s}) \doteq f(\bar{t})$, then $\mathcal{M}, \rho \vDash \wedge_{1 \leqslant i \leqslant \text{ar}(f)} s_i \doteq t_i$.

and every sort is nonempty.

Let $\mathcal{L}$ be $\mathcal{L}_1$ with: a set of unary predicates $\chi(\cdot)$, which stand for invariants of recursive definitions in the constraints we will derive for type inference problems. And 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 $\text{PV}^1(\cdot)$, resp. $\text{PV}^2(\cdot)$ be the set of unary, resp. binary predicate variables in any expression, and $\text{PV}(\Phi) = \text{PV}^1(\Phi) \cup \text{PV}^2(\Phi)$. We define *solved form formulas* to be existentially quantified conjunctions of atoms $\exists \bar{\alpha}.A$ without predicate variables. Let δ, δ' be fixed variables of sort s_{type}.

Definition A.1. *For a formula Φ, let $\bar{\chi} = \text{PV}^1(\Phi)$, resp. $\overline{\chi_K} = \text{PV}^2(\Phi)$, 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. *$\overline{\exists \bar{\alpha}_i.F_i \exists \bar{\alpha}_j.F_j}$ 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. $\mathrm{FV}(F_K) \subseteq \delta\delta'\bar{\alpha}_K$.

For an interpretation of predicate variables $\mathcal{I} = [\bar{\chi} := \overline{\exists \bar{\alpha}_\chi . F_\chi}; \overline{\chi_K} := \overline{\exists \bar{\alpha}_K . F_K}]$, define the corresponding substitution of predicate variables in a formula Φ by:

$$\begin{aligned}
\mathcal{I}(\chi(\tau_\chi)) &= \exists \bar{\alpha}_\chi . F_\chi[\delta := \tau_\chi] \\
\mathcal{I}(\chi_K(\tau_\chi, \tau'_\chi)) &= \exists \bar{\alpha}_\chi . F_\chi[\delta := \tau_\chi; \delta' := \tau'_\chi] \\
\mathcal{I}(a) &= a \quad \text{for atom } a \in \mathcal{L}_1 \\
\mathcal{I}(\mathcal{Q}.\Phi) &= \mathcal{Q}.\mathcal{I}(\Phi) \quad \text{for any quantifier prefix } \mathcal{Q} \\
\mathcal{I}(\Phi_1 \odot \Phi_2) &= \mathcal{I}(\Phi_1) \odot \mathcal{I}(\Phi_2) \quad \text{for any logical connective } \odot
\end{aligned}$$

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, $\mathcal{M}, \mathcal{I} \vDash \Phi$. Sometimes 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.

A.1.2. The GADT Type System

Let $D \vDash C$ mean here $\mathcal{M} \vDash D \Rightarrow C$. 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 $\Delta' \leqslant \Delta$ denote $D' \Rightarrow \exists \bar{\beta}.(D \wedge_{x \in \mathrm{Dom}(\Gamma)} \Gamma(x) \doteq \Gamma'(x))$ when $\mathrm{Dom}(\Gamma) = \mathrm{Dom}(\Gamma')$, and otherwise a falsehood (compare lemma 3.5 of [47]). 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 A.2. *Properties of environment fragments (see [47] 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 A.3. *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 A.4. *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 A.5. Weakening *(patterns and expressions). Assume* $C_1 \vDash C_2$*. If* $C_2 \vdash p : \tau \longrightarrow \Delta$ *(resp.* $C_2, \Gamma \vdash e : \tau$*,* $C_2, \Gamma \vdash e : \sigma$*) is derivable, then there exists a derivation of* $C_1 \vdash p : \tau \longrightarrow \Delta$ *(resp.* $C_1, \Gamma \vdash e : \tau$*,* $C_1, \Gamma \vdash e : \sigma$*) 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 A.6. *If* $\Sigma \subset \Sigma'$ *and* $C \vdash p : \tau \longrightarrow \Delta$ *(resp.* $C, \Gamma \vdash e : \tau$*,* $C, \Gamma \vdash e : \sigma$*) is derivable with constructors* Σ*, then the same derivation works with constructors* Σ'*.*

Lemma A.7. Correctness *(patterns).* $[\![\vdash p{\downarrow}\tau]\!] \vdash p : \tau \longrightarrow [\![\vdash p{\uparrow}\tau]\!]$.

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, $[\![\vdash p_i{\downarrow}\tau]\!] \vdash p_i : \tau \longrightarrow [\![\vdash p_i{\uparrow}\tau]\!]$ for $i = 1, 2$.
    2. By weakening and p-And we have the goal.
- Case $K p_1 \ldots p_n$.
    1. Let $\Sigma \ni K :: \forall \bar{\alpha} \bar{\beta}[D].\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha})$.
    2. By the induction hypothesis, $[\![\vdash p_i{\downarrow}\tau_i]\!] \vdash p_i : \tau_i \longrightarrow [\![\vdash p_i{\uparrow}\tau_i]\!]$ for $i = 1, \ldots, n$.
    3. The p-Cstr rule says $\forall i \ (C \wedge D \vdash p_i : \tau_i \longrightarrow \Delta_i) \,/_{\text{p-Cstr}}\, C \vdash p : \varepsilon(\bar{\alpha}) \longrightarrow \exists \bar{\beta}[D](\Delta_1 \times \ldots \times \Delta_n)$, where $\Delta_i := [\![\vdash p_i{\uparrow}\tau_i]\!]$. Applying it to (2) we get $C \vdash p : \varepsilon(\bar{\alpha}) \longrightarrow \exists \bar{\beta}[D](\Delta_1 \times \ldots \times \Delta_n)$ 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 $\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}'\bar{\beta}' \wedge \varepsilon(\bar{\alpha}') \doteq \tau \wedge C \vdash p : \tau \longrightarrow \exists \bar{\beta}[D](\Delta_1 \times \ldots \times \Delta_n)$. The conjunction of equations $\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}'\bar{\beta}'$ allows for renaming of the "old" variables $\bar{\alpha}\bar{\beta}$ by the "fresh" variables $\bar{\alpha}'\bar{\beta}'$.
    6. By proposition A.2, transitivity of $\leqslant$, and p-SubOut, we get $\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}'\bar{\beta}' \wedge \varepsilon(\bar{\alpha}') \doteq \tau \wedge C \vdash p : \tau \longrightarrow \exists \bar{\alpha}'\bar{\beta}'[D'](\Delta_1' \times \ldots \times \Delta_n')$.
    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 $\exists \bar{\alpha}'.\varepsilon(\bar{\alpha}') \doteq \tau \wedge \forall \bar{\beta}'.D' \Rightarrow \wedge_i [\![\vdash p_i{\downarrow}\tau_i']\!] \vdash p : \tau \longrightarrow \exists \bar{\alpha}'\bar{\beta}'[D'](\Delta_1' \times \ldots \times \Delta_n')$. □

Proof of theorem 3.1.

Proof. By induction on the structure of e.

- Case e 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 e is **assert false**. The goal follows by FELIM.
- Case e is **assert num** $m \leq n; e$.
  1. Let $\alpha^1\alpha^2\#\mathrm{FV}(\Gamma, \tau)$.
  2. By the induction hypothesis, we have $[\![\Gamma \vdash e_1\colon \mathrm{Num}(\alpha^1)]\!]$, $\Gamma \vdash e_1\colon \mathrm{Num}(\alpha^1)$, $[\![\Gamma \vdash e_2\colon \mathrm{Num}(\alpha^2)]\!], \Gamma \vdash e_2\colon \mathrm{Num}(\alpha^2)$ and $[\![\Gamma \vdash e_3\colon \tau]\!], \Gamma \vdash e_3\colon \tau$.
  3. By weakening and ASSERTLEQ, this yields $[\![\Gamma \vdash e_1\colon \mathrm{Num}(\alpha^1)]\!] \wedge [\![\Gamma \vdash e_2\colon \mathrm{Num}(\alpha^2)]\!] \wedge \alpha^1 \leq \alpha^2 \wedge [\![\Gamma \vdash e_3\colon \tau]\!], \Gamma \vdash$ **assert num** $e_1 \leq e_2; e_3\colon \tau$.
  4. By HIDE using (1), this gives the goal.
- Case e is **assert type** $e_1 = e_2; e_3$.
  1. Let $\alpha^1\alpha^2\#\mathrm{FV}(\Gamma, \tau)$.
  2. By the induction hypothesis, we have $[\![\Gamma \vdash e_1\colon \alpha^1]\!], \Gamma \vdash e_1\colon \alpha^1$, $[\![\Gamma \vdash e_2\colon \alpha^2]\!], \Gamma \vdash e_2\colon \alpha^2$ and $[\![\Gamma \vdash e_3\colon \tau]\!], \Gamma \vdash e_3\colon \tau$.
  3. By weakening and ASSERTEQTY, this yields $[\![\Gamma \vdash e_1\colon \alpha^1]\!] \wedge [\![\Gamma \vdash e_2\colon \alpha^2]\!] \wedge \alpha^1 \doteq \alpha^2 \wedge [\![\Gamma \vdash e_3\colon \tau]\!], \Gamma \vdash$ **assert type** $e_1 = e_2; e_3\colon \tau$.
  4. By HIDE using (1), this gives the goal.
- Case e is **runtime failure** s.
  1. By the induction hypothesis, we have $[\![\Gamma \vdash s\colon \mathrm{String}]\!], \Gamma \vdash s\colon \mathrm{String}$.
  2. The goal follows from RUNTIMEFAILURE, since $[\![\Gamma \vdash$ **runtime failure** $s\colon \tau]\!] = [\![\Gamma \vdash s\colon \mathrm{String}]\!]$.
- Case e 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 $[\![\Gamma \vdash c_i\colon \alpha_1 \to \alpha_2]\!], \Gamma \vdash c_i\colon \alpha_1 \to \alpha_2$.
  3. By (2), weakening and ABS, $[\![\Gamma \vdash \bar{c}\colon \alpha_1 \to \alpha_2]\!], \Gamma \vdash \lambda\bar{c}\colon \alpha_1 \to \alpha_2$.
  4. By weakening and EQU, (3) implies $[\![\Gamma \vdash \bar{c}\colon \alpha_1 \to \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 \doteq \tau, \Gamma \vdash \lambda\bar{c}\colon \tau$.
  5. By (1) and HIDE, this implies $[\![\Gamma \vdash \lambda\bar{c}\colon \tau]\!], \Gamma \vdash \lambda\bar{c}\colon \tau$.
- Case e 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 e is $K\, e_1 \ldots e_n$.
  1. Let $\Sigma \ni K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \ldots \times \tau_n \to \varepsilon(\bar{\alpha})$.
  2. By induction hypothesis and weakening for each $i = 1, \ldots, n$
  $$\wedge_j [\![\Gamma \vdash e_j\colon \tau_j]\!] \wedge D \wedge \varepsilon(\bar{\alpha}) \doteq \tau, \Gamma \vdash e_i\colon \tau_i$$
  3. Applying Cstr to (1) and (3) we obtain
  $$\wedge_i [\![\Gamma \vdash e_i\colon \tau_i]\!] \wedge D \wedge \varepsilon(\bar{\alpha}) \doteq \tau, \Gamma \vdash K\, e_1 \ldots e_n\colon \varepsilon(\bar{\alpha})$$
  4. Let $\bar{\alpha}'\bar{\beta}' \# \mathrm{FV}(\Gamma, \tau)$ and $\tau_i' := \tau_i[\bar{\alpha}\bar{\beta} := \bar{\alpha}'\bar{\beta}']$, $D' := D[\beta\bar{\alpha} := \beta'\bar{\alpha}']$.
  $$\bar{\alpha}\bar{\beta} \doteq \bar{\alpha}'\bar{\beta}' \wedge_i [\![\Gamma \vdash e_i\colon \tau_i']\!] \wedge D \wedge \varepsilon(\bar{\alpha}') \doteq \tau, \Gamma \vdash K\, e_1 \ldots e_n\colon \varepsilon(\bar{\alpha}')$$
  5. By Equ, (1) Hide and weakening, since w.l.o.g. $\bar{\alpha}\bar{\beta}$ do not appear unbounded in the goal, $[\![\Gamma \vdash K\, e_1 \ldots e_n\colon \tau]\!], \Gamma \vdash K\, e_1 \ldots e_n\colon \tau$.
- Case e is $\mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2$.
  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 $[\![\Gamma \vdash \mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2\colon \tau]\!], \Gamma \vdash \mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2\colon \tau$.
- Case e 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 A.7 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 $[\![\Gamma \vdash p.e\colon \tau]\!], \Gamma \vdash p.e\colon \tau$.

- Case e is p **when** $\wedge_i m_i \leqslant n_i.e$ and $e \neq$ **assert false** $\wedge \ldots \wedge e \neq \lambda(p' \ldots \lambda(p''.$**assert false**$))$.

1. Let $\bar{\beta}\overline{\alpha_i^1\alpha_i^2}\#\mathrm{FV}(\Gamma, \tau_1, \tau_2)$. Recall that $[\![\Gamma \vdash p$ **when** $\wedge_i m_i \leqslant n_i.e\colon \tau_1 \to \tau_2]\!] = \exists\overline{\alpha_i^1\alpha_i^2}.\Phi$, for $\Phi =$
$[\![\vdash p{\downarrow}\tau_1]\!] \wedge \forall\bar{\beta}.D \Rightarrow \wedge_i [\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!] \wedge_i$
$[\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!] \wedge (\wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!])$.
2. τ is of the form $\tau_1 \to \tau_2$. Write $[\![\vdash p{\uparrow}\tau_1]\!]$ as $\exists\bar{\beta}[D]\Gamma'$.
3. By induction hypothesis, $[\![\Gamma\Gamma' \vdash e\colon \tau_2]\!], \Gamma\Gamma' \vdash e\colon \tau_2$, and also $[\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!]$, $\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)$ and $[\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!], \Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)$.
4. By lemma A.7 and (1), we have $[\![\vdash p{\downarrow}\tau_1]\!] \vdash p\colon \tau_1 \longrightarrow \exists\bar{\beta}[D]\Gamma'$.
5. By instantiation of $\bar{\beta}$ and weakening, (3) implies
$$\begin{aligned} \Phi \wedge D \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma' &\vdash e\colon \tau_2 \\ \Phi \wedge D, \Gamma\Gamma' &\vdash m_i\colon \mathrm{Num}(\alpha_i^1) \\ \Phi \wedge D, \Gamma\Gamma' &\vdash n_i\colon \mathrm{Num}(\alpha_i^2) \end{aligned}$$
6. By weakening, (4) implies $\Phi \vdash p\colon \tau_1 \longrightarrow \exists\bar{\beta}[D]\Gamma'$.
7. By (5), (6), (1), and CLAUSE, we obtain $\Phi, \Gamma \vdash p$ **when** $\wedge_i m_i \leqslant n_i.e\colon \tau$.
8. By (7) and HIDE, we get the goal.

- Case e is p **when** $\wedge_i m_i \leqslant n_i.e$ and $e =$ **assert false** $\vee \ldots \vee e = \lambda(p' \ldots \lambda(p''.$**assert false**$))$. The proof is nearly identical to above.

1. Let $\alpha_3\bar{\beta}\overline{\alpha_i^1\alpha_i^2}\#\mathrm{FV}(\Gamma, \tau_1, \tau_2)$. Recall that $[\![\Gamma \vdash p$ **when** $\wedge_i m_i \leqslant n_i.e\colon \tau_1 \to \tau_2]\!] = \exists\alpha_3\overline{\alpha_i^1\alpha_i^2}.\Phi$, for $\Phi =$
$[\![\vdash p{\downarrow}\alpha_3]\!] \wedge \forall\bar{\beta}.D \Rightarrow \wedge_i [\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!]$
$\wedge (\alpha_3 \doteq \tau_1 \wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma' \vdash e\colon \tau_2]\!])$.
2. τ is of the form $\tau_1 \to \tau_2$. Write $[\![\vdash p{\uparrow}\alpha_3]\!]$ as $\exists\bar{\beta}[D]\Gamma'$.
3. By induction hypothesis, $[\![\Gamma\Gamma' \vdash e\colon \tau_2]\!], \Gamma\Gamma' \vdash e\colon \tau_2$, and also $[\![\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)]\!]$, $\Gamma\Gamma' \vdash m_i\colon \mathrm{Num}(\alpha_i^1)$ and $[\![\Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)]\!], \Gamma\Gamma' \vdash n_i\colon \mathrm{Num}(\alpha_i^2)$.
4. By lemma A.7 and (1), we have $[\![\vdash p{\downarrow}\alpha_3]\!] \vdash p\colon \alpha_3 \longrightarrow \exists\bar{\beta}[D]\Gamma'$.
5. By instantiation of $\bar{\beta}$ and weakening, (3) implies
$$\begin{aligned} \Phi \wedge D \wedge \alpha_3 \doteq \tau_1 \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma' &\vdash e\colon \tau_2 \\ \Phi \wedge D, \Gamma\Gamma' &\vdash m_i\colon \mathrm{Num}(\alpha_i^1) \\ \Phi \wedge D, \Gamma\Gamma' &\vdash n_i\colon \mathrm{Num}(\alpha_i^2) \end{aligned}$$
6. By weakening, (4) implies $\Phi \vdash p\colon \alpha_3 \longrightarrow \exists\bar{\beta}[D]\Gamma'$.
7. By (5), (6), (1), and NEGCLAUSE, we obtain $\Phi, \Gamma \vdash p$ **when** $\wedge_i m_i \leqslant n_i.e\colon \tau$.
8. By (7) and HIDE, we get the goal. □

$\Gamma'{\doteq}\Gamma''$ 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 A.8. Completeness *(patterns). Let* $\Delta = \exists\bar\beta'[D']\Gamma'$ *and* $[\![\vdash p{\uparrow}\tau]\!] = \exists\bar\beta''[D'']\Gamma'' = \Delta'$. $C \vdash p{:}\,\tau \longrightarrow \Delta$ *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 $C \vdash p{:}\,\tau \longrightarrow \Delta$. 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 $C \vdash p_i{:}\,\tau \longrightarrow \Delta_i$, 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 $C \wedge D_0 \vdash p_i{:}\,\tau_i[\bar\alpha\bar\beta := \bar\alpha_0\bar\beta_0] \longrightarrow \Delta_i$.
  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 $C \wedge D_0 \wedge \bar\alpha_0\bar\beta_0{\doteq}\bar\alpha'_0\bar\beta'_0 \vdash p_i{:}\,\tau'_i \longrightarrow \Delta_i$.
  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), $C \wedge \bar\alpha_0\bar\beta_0{\doteq}\bar\alpha'_0\bar\beta'_0 \vDash D'_0 \Rightarrow \forall\bar\beta''_1 \ldots \bar\beta''_n.\, \wedge_i D''_i \Rightarrow \exists\bar\beta'_1 \ldots \bar\beta'_n.\, \wedge_i (D'_i \wedge \Gamma''_i{\doteq}\Gamma'_i)$.

10. From (9) by (2) and (6), $C \vDash \forall \bar{\alpha}'_0 \bar{\beta}'_0. \bar{\alpha}_0 \bar{\beta}_0 \doteq \bar{\alpha}'_0 \bar{\beta}'_0 \wedge D'_0 \Rightarrow \forall \bar{\beta}''_1 \ldots \bar{\beta}''_n. \wedge_i D''_i \Rightarrow \exists \bar{\beta}'_1 \ldots \bar{\beta}'_n. \wedge_i (D'_i \wedge \Gamma''_i \doteq \Gamma'_i)$, which is equivalent to
$$C \vDash \forall \bar{\alpha}'_0 \bar{\beta}'_0 \bar{\beta}''_1 \ldots \bar{\beta}''_n. \bar{\alpha}_0 \bar{\beta}_0 \doteq \bar{\alpha}'_0 \bar{\beta}'_0 \wedge D'_0 \wedge_i D''_i \Rightarrow \exists \bar{\beta}'_1 \ldots \bar{\beta}'_n. \wedge_i (D'_i \wedge \Gamma''_i \doteq \Gamma'_i)$$
11. Observe, that w.l.o.g. $\bar{\beta}'' := \bar{\alpha}'_0 \bar{\beta}'_0 \bar{\beta}''_1 \ldots \bar{\beta}''_n$. 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
$$C \vDash \forall \bar{\beta}''. \bar{\beta}_0 \doteq \bar{\beta}'_0 \wedge D'' \Rightarrow \exists \bar{\beta}'_1 \ldots \bar{\beta}'_n. D' \wedge \Gamma'' \doteq \Gamma'$$
13. Also, $\bar{\beta}' = \bar{\beta}_0 \bar{\beta}'_1 \ldots \bar{\beta}'_n$. 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 A.9. *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 judgments 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 3.2.

Proof. We proceed by induction on the derivation of $C, \Gamma \vdash e\colon \tau$. To slightly simplify the proof, the induction is actually on the lexicographic ordering: (# of structural rule applications Var, Cstr, Abs, App, LetRec, Clause; # of nonstructural 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 AssertFalse. AssertFalse's premise is $C \vDash \boldsymbol{F}$, we have the goal from $\vDash \boldsymbol{F} \Rightarrow \Phi$ holding for any Φ, and transitivity of $\vDash$.
- Case AssertLeq.
  1. AssertLeq's premises are $C, \Gamma \vdash e_1\colon \mathrm{Num}(\tau_1)$, $C, \Gamma \vdash e_2\colon \mathrm{Num}(\tau_2)$, $C \vDash \tau_1 \leq \tau_2$ and $C, \Gamma \vdash e_3\colon \tau$.
  2. Pick w.l.o.g. $\alpha^1\alpha^2 \notin \mathrm{FV}(C, \tau_1, \tau_2, \Gamma, \tau)$. By rule Equ, (1) implies $C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2, \Gamma \vdash e_1\colon \mathrm{Num}(\alpha^1)$, $C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2, \Gamma \vdash e_2\colon \mathrm{Num}(\alpha^2)$ and $C, \Gamma \vdash e_3\colon \tau$.
  3. By induction hypothesis and weakening, (2) implies $\mathcal{I}_i, C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \leq \alpha^2 \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash e_i\colon \mathrm{Num}(\alpha^i)]\!]$, $i = 1, 2$, $\Phi_3 = [\![\Gamma \vdash e_3\colon \tau]\!]$.
  4. By (2) and nonemptiness of sorts, we have $C \vDash \exists\alpha^1\alpha^2.(C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \leq \alpha^2)$.
  5. By (3), the premise and because $C \vDash D$ implies $\exists\alpha.C \vDash \exists\alpha.D$, we have $\mathcal{I}_1\mathcal{I}_2\mathcal{I}_3$, $\exists\alpha^1\alpha^2.(C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \leq \alpha^2) \vDash \exists\alpha.(\Phi_1 \wedge \Phi_2 \wedge \Phi_3)$.
  6. By (4) and (5), we have the goal $\mathcal{I}, C \vDash [\![\Gamma \vdash \textbf{assert num}\, e_1 \leq e_2; e_3\colon \tau]\!]$ with $\mathcal{I} = \mathcal{I}_1\mathcal{I}_2\mathcal{I}_3$.
- Case AssertEqty.
  1. AssertEqty's premises are $C, \Gamma \vdash e_i\colon \tau_i$ for $i = 1, 2, 3, \tau_3 = \tau$ and $C \vDash \tau_1 \doteq \tau_2$.
  2. Pick w.l.o.g. $\alpha^1\alpha^2 \notin \mathrm{FV}(C, \tau_1, \tau_2, \Gamma, \tau)$. By rule Equ, (1) implies $C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2, \Gamma \vdash e_i\colon \alpha^i$ for $i = 1, 2$.
  3. By induction hypothesis and weakening, (1) and (2) imply $\mathcal{I}_i, C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \doteq \alpha^2 \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash e_i\colon \alpha^i]\!]$, $i = 1, 2$, $\Phi_3 = [\![\Gamma \vdash e_3\colon \tau]\!]$.
  4. By (2) and nonemptiness of sorts, we have $C \vDash \exists\alpha^1\alpha^2.(C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \doteq \alpha^2)$.

5. By (3), the premise and because $C \vDash D$ implies $\exists\alpha.C \vDash \exists\alpha.D$, we have $\mathcal{I}_1\mathcal{I}_2\mathcal{I}_3$, $\exists\alpha^1\alpha^2.(C \wedge \alpha^1 \doteq \tau_1 \wedge \alpha^2 \doteq \tau_2 \wedge \alpha^1 \doteq \alpha^2) \vDash \exists\alpha.(\Phi_1 \wedge \Phi_2 \wedge \Phi_3)$.
6. By (4) and (5), we have the goal $\mathcal{I}, C \vDash [\![\Gamma \vdash \textbf{assert type}\ e_1 = e_2;\ e_3 \colon \tau]\!]$ with $\mathcal{I} = \mathcal{I}_1\mathcal{I}_2\mathcal{I}_3$.

- Case RUNTIMEFAILURE.
  1. RUNTIMEFAILURE's premise is $C, \Gamma \vdash s \colon \mathrm{String}$.
  2. By induction hypothesis, (1) implies $\mathcal{I}, C \vDash [\![\Gamma \vdash s \colon \mathrm{String}]\!]$ and thus the goal.
- Case CSTR.
  1. CST'R's premises are $C, \Gamma \vdash e_i \colon \tau_i$, $i = 1, ..., n$, $C \vDash D$ and $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 ... \tau_n \to \varepsilon(\bar{\alpha})$. $\tau = \varepsilon(\bar{\alpha})$.
  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, $\mathcal{I}_i, C \wedge \bar{\alpha}'\bar{\beta}' \doteq \bar{\alpha}\bar{\beta} \vDash \Phi_i$, $i = 1, ..., n$.
  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), $\mathcal{I}, C \vDash [\![\Gamma \vdash K\, e_1 ... e_n \colon \tau]\!]$ for $\mathcal{I} = \mathcal{I}_1 ... \mathcal{I}_n$.
- Case ABS. In this case, $\tau := \tau_1 \to \tau_2$.
  1. ABS' premise is $C, \Gamma \vdash \bar{c} \colon \tau_1 \to \tau_2$, which by induction hypothesis implies $\mathcal{I}_i$, $C \vDash \Phi_i$ for $\Phi_i = [\![\Gamma \vdash p_i.e_i \colon \tau_1 \to \tau_2]\!]$, $i = 1, ..., n$.
  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 $\mathcal{I}_1\mathcal{I}_2, C \wedge \alpha_1 \doteq \tau_1 \wedge \alpha_2 \doteq \tau_2 \vDash \wedge_i \Phi_i \wedge \alpha_1 \to \alpha_2 \doteq \tau_1 \to \tau_2$.
  4. Combining (2) and (3), $\mathcal{I}_1\mathcal{I}_2, C \vDash \exists\alpha_1\alpha_2.(\wedge_i \Phi_i \wedge \alpha_1 \to \alpha_2 \doteq \tau_1 \to \tau_2)$.
  5. By (1) and (4), $\mathcal{I}_1\mathcal{I}_2, C \vDash [\![\Gamma \vdash \lambda\bar{c} \colon \tau]\!]$.
- Case APP.
  1. APP's premises are $C, \Gamma \vdash e_1 \colon \tau' \to \tau$ and $C, \Gamma \vdash e_2 \colon \tau'$.
  2. Pick w.l.o.g. $\alpha \notin \mathrm{FV}(C, \tau', \Gamma, \tau)$. By rule EQU, (1) implies $C \wedge \alpha \doteq \tau', \Gamma \vdash e_1 \colon \alpha \to \tau$ 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 (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 $\mathcal{I}, C \vDash [\![\Gamma \vdash \mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2 : \tau]\!]$.

- Case Clause.
  1. Clause's premises are: $C \vdash p : \tau_1 \longrightarrow \exists\bar{\beta}[D]\Gamma'$,
  2. $C \wedge D \wedge_i \tau^{m_i} \leq \tau^{n_i}, \Gamma\Gamma' \vdash e : \tau_2$,
  3. $C \wedge D, \Gamma\Gamma' \vdash m_i : \mathrm{Num}(\tau^{m_i})$ and $C \wedge D, \Gamma\Gamma' \vdash n_i : \mathrm{Num}(\tau^{n_i})$,
  4. and $\bar{\beta}\#\mathrm{FV}(C, \Gamma, \tau_2)$.
  5. Assume w.l.o.g. that $\bar{\beta}\#\mathrm{FV}(\tau_1)$.
  6. Let $\alpha_i^1\alpha_i^2\#\mathrm{FV}(C, \tau_1, \tau_2, \overline{\tau^{m_i}, \tau^{n_i}})$.
  7. Let $[\![\vdash p\uparrow\tau_1]\!] = \exists\bar{\beta}'[D']\Gamma''$, where $\bar{\beta}'\#\mathrm{FV}(\Gamma, C, \tau_1, \tau_2, \bar{\beta})$.
  8. By lemma A.8, (1) and (7) gives $C \vDash [\![\vdash p\downarrow\tau_1]\!]$
  9. and $C \vDash \forall\bar{\beta}'.D' \Rightarrow \exists\bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$, which is equivalent to $C \wedge D' \vDash \exists\bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$.
  10. Recall that $[\![\Gamma \vdash p\ \mathbf{when} \wedge_i m_i \leqslant n_i.e : \tau_1 \to \tau_2]\!] = \exists\overline{\alpha_i^1\alpha_i^2}.\Phi$, for $\Phi = [\![\vdash p\downarrow\tau_1]\!] \wedge \forall\bar{\beta}'.D' \Rightarrow \wedge_i [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!] \wedge (\wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma'' \vdash e : \tau_2]\!])$.
  11. By lemma A.9, weakening and Equ, (3) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$, $\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.
  12. From (9) we have $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash \exists\bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash \exists\bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}$ for some $\tau^{m_i\prime}, \tau^{n_i\prime}$.
  13. By (11), Hide and (12), we get $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime}, \Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.

14. By induction hypothesis applied to (13) we have $\mathcal{I}_i^1$, $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!]$ and $\mathcal{I}_i^2, C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!]$.
15. By lemma A.9 and weakening, (2) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge_i \alpha_i^1 \doteq \tau^{m_i} \wedge_i \alpha_i^2 \doteq \tau^{n_i} \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash e : \tau_2$.
16. By (15), HIDE and (12), we get $C \wedge D' \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash e : \tau_2$.
17. By induction hypothesis, (16) gives $\mathcal{I}^3, C \wedge D' \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge_i \alpha_i^1 \leq \alpha_i^2 \vDash [\![\Gamma\Gamma'' \vdash e : \tau_2]\!]$.
18. By (8), (10), (14) and (17), we get $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3, C \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \vDash \Phi$.
19. By nonemptiness of the domain, from (18) we get the goal $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3, C \vDash \exists \overline{\alpha_i^1 \alpha_i^2}.\Phi$.

- Case NEGCLAUSE. The proof is nearly identical as above.
  1. NEGCLAUSE's premises are: $C \vdash p : \tau_3 \longrightarrow \exists \bar{\beta}[D]\Gamma'$,
  2. $C \wedge D \wedge \tau_1 \doteq \tau_3 \wedge_i \tau^{m_i} \leq \tau^{n_i}, \Gamma\Gamma' \vdash e : \tau_2$,
  3. $C \wedge D, \Gamma\Gamma' \vdash m_i : \mathrm{Num}(\tau^{m_i})$ and $C \wedge D, \Gamma\Gamma' \vdash n_i : \mathrm{Num}(\tau^{n_i})$,
  4. and $\bar{\beta} \# \mathrm{FV}(C, \Gamma, \tau_2)$.
  5. Assume w.l.o.g. that $\bar{\beta} \# \mathrm{FV}(\tau_3)$.
  6. Let $\alpha_3 \alpha_i^1 \alpha_i^2 \# \mathrm{FV}(C, \tau_1, \tau_2, \tau_3, \overline{\tau^{m_i}, \tau^{n_i}})$.
  7. Let $[\![\vdash p \uparrow \tau_3]\!] = \exists \bar{\beta}'[D']\Gamma''$, where $\bar{\beta}' \# \mathrm{FV}(\Gamma, C, \tau_1, \tau_2, \tau_3, \bar{\beta})$.
  8. By lemma A.8, (1) and (7) gives $C \vDash [\![\vdash p \downarrow \tau_3]\!]$
  9. and $C \vDash \forall \bar{\beta}'.D' \Rightarrow \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$, which is equivalent to $C \wedge D' \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$.
  10. Recall that $[\![\Gamma \vdash p \,\mathbf{when} \wedge_i m_i \leqslant n_i.e : \tau_1 \to \tau_2]\!] = \exists \alpha_3 \overline{\alpha_i^1 \alpha_i^2}.\Phi$, for $\Phi = [\![\vdash p \downarrow \alpha_3]\!] \wedge \forall \bar{\beta}'.D' \Rightarrow \wedge_i [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!] \wedge (\tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma'' \vdash e : \tau_2]\!])$.
  11. By lemma A.9, weakening and EQU, (3) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$, $\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.
  12. From (9) we have $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}$ for some $\tau^{m_i\prime}, \tau^{n_i\prime}$.
  13. By (11), HIDE and (12), we get $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime}, \Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.
  14. By induction hypothesis applied to (13) we have $\mathcal{I}_i^1$, $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!]$ and $\mathcal{I}_i^2, C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!]$.
  15. By lemma A.9 and weakening, (2) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i} \wedge_i \alpha_i^2 \doteq \tau^{n_i} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash e : \tau_2$.
  16. By (15), HIDE and (12), we get $C \wedge D' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash e : \tau_2$.

17. By induction hypothesis, (16) gives $\mathcal{I}^3, C \wedge D' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2 \vDash [\![\Gamma\Gamma'' \vdash e : \tau_2]\!]$.
18. By (8), (10), (14) and (17), we get $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3, C \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau'_{n_i} \vDash \Phi$.
19. By nonemptiness of the domains, from (18) we get the goal $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3$, $C \vDash \exists \alpha_3 \overline{\alpha_i^1 \alpha_i^2}.\Phi$.

- Case FailClause. The proof is nearly identical as above.
    1. FailClause's premises are: $C \vdash p : \tau_3 \longrightarrow \exists \bar{\beta}[D]\Gamma'$,
    2. $C \wedge D \wedge \tau_1 \doteq \tau_3 \wedge_i \tau^{m_i} \leq \tau^{n_i}, \Gamma\Gamma' \vdash s : \mathrm{String}$,
    3. $C \wedge D, \Gamma\Gamma' \vdash m_i : \mathrm{Num}(\tau^{m_i})$ and $C \wedge D, \Gamma\Gamma' \vdash n_i : \mathrm{Num}(\tau^{n_i})$,
    4. and $\bar{\beta} \# \mathrm{FV}(C, \Gamma)$.
    5. Assume w.l.o.g. that $\bar{\beta} \# \mathrm{FV}(\tau_3)$.
    6. Let $\alpha_3 \alpha_i^1 \alpha_i^2 \# \mathrm{FV}(C, \tau_1, \tau_3, \overline{\tau^{m_i}, \tau^{n_i}})$.
    7. Let $[\![\vdash p \uparrow \tau_3]\!] = \exists \bar{\beta}'[D']\Gamma''$, where $\bar{\beta}' \# \mathrm{FV}(\Gamma, C, \tau_1, \tau_3, \bar{\beta})$.
    8. By lemma A.8, (1) and (7) gives $C \vDash [\![\vdash p \downarrow \tau_3]\!]$
    9. and $C \vDash \forall \bar{\beta}'.D' \Rightarrow \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$, which is equivalent to $C \wedge D' \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma'$.
    10. Recall that $[\![\Gamma \vdash p \,\mathbf{when}\, \wedge_i m_i \leqslant n_i.e : \tau_1 \to \tau_2]\!] = \exists \alpha_3 \overline{\alpha_i^1 \alpha_i^2}.\Phi$, for $\Phi = [\![\vdash p \downarrow \alpha_3]\!] \wedge \forall \bar{\beta}'.D' \Rightarrow \wedge_i [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!] \wedge_i [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!] \wedge (\tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2 \Rightarrow [\![\Gamma\Gamma'' \vdash e : \tau_2]\!])$.
    11. By lemma A.9, weakening and Equ, (3) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$, $\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.
    12. From (9) we have $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^1 \doteq \tau^{m_i}$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash \exists \bar{\beta}.D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_i^2 \doteq \tau^{n_i}$ for some $\tau^{m_i\prime}, \tau^{n_i\prime}$.
    13. By (11), Hide and (12), we get $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime}, \Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)$ and $C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime}, \Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)$.
    14. By induction hypothesis applied to (13) we have $\mathcal{I}_i^1$, $C \wedge D' \wedge \alpha_i^1 \doteq \tau^{m_i\prime} \vDash [\![\Gamma\Gamma'' \vdash m_i : \mathrm{Num}(\alpha_i^1)]\!]$ and $\mathcal{I}_i^2, C \wedge D' \wedge \alpha_i^2 \doteq \tau^{n_i\prime} \vDash [\![\Gamma\Gamma'' \vdash n_i : \mathrm{Num}(\alpha_i^2)]\!]$.
    15. By lemma A.9 and weakening, (2) implies $C \wedge D \wedge \Gamma'' \doteq \Gamma' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i} \wedge_i \alpha_i^2 \doteq \tau^{n_i} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash s : \mathrm{String}$.
    16. By (15), Hide and (12), we get $C \wedge D' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2, \Gamma\Gamma'' \vdash s : \mathrm{String}$.
    17. By induction hypothesis, (16) gives $\mathcal{I}^3, C \wedge D' \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \wedge \tau_1 \doteq \alpha_3 \wedge_i \alpha_i^1 \leq \alpha_i^2 \vDash [\![\Gamma\Gamma'' \vdash s : \mathrm{String}]\!]$.
    18. By (8), (10), (14) and (17), we get $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3, C \wedge \alpha_3 \doteq \tau_3 \wedge_i \alpha_i^1 \doteq \tau^{m_i\prime} \wedge_i \alpha_i^2 \doteq \tau^{n_i\prime} \vDash \Phi$.
    19. By nonemptiness of the domains, from (18) we get the goal $\bar{\mathcal{I}_i^1}\bar{\mathcal{I}_i^2}\mathcal{I}^3$, $C \vDash \exists \alpha_3 \overline{\alpha_i^1 \alpha_i^2}.\Phi$.

- Case EQU.
    1. EQU's premises are $C,\ \Gamma \vdash e\colon \tau'$, 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 $\mathcal{I}_1, C \vDash [\![\Gamma \vdash e\colon \tau]\!]$ and $\mathcal{I}_2, D \vDash [\![\Gamma \vdash e\colon \tau]\!]$ for some interpretations of predicate variables $\mathcal{I}_1, \mathcal{I}_2$.
    2. Therefore, we have $\mathcal{I}, C \vee D \vDash [\![\Gamma \vdash e\colon \tau]\!]$, for both $\mathcal{I} = \mathcal{I}_1$ and $\mathcal{I} = \mathcal{I}_2$. □

Proof of corollary 3.3.

Proof. $C, \Gamma \vdash e\colon \forall\bar{\alpha}[D].\tau$ 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 3.2, 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 $\mathcal{I}, C \vDash \forall\bar{\alpha}.D \Rightarrow [\![\Gamma \vdash e\colon \tau]\!]$. □

A.1.3. Existential Types

Proof of theorem 3.7.

Proof. By inspecting Table 3.11, 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 3.1 (Correctness) $[\![\Gamma, \Sigma_0 \vdash e\colon \tau]\!], \Gamma, \Sigma_0 \vdash e\colon \tau$. Case: $\mathcal{E}(e) \neq \varnothing$.
    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 3.1 (Correctness) Case: e 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 $[\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2 \colon \alpha_0 \to \tau]\!] \wedge \not\in(\alpha_0), \Gamma \vdash \lambda(p.e_2)e_1 \colon \tau$.
  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: $[\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2 \colon \alpha_0 \to \tau]\!] \wedge \not\in(\alpha_0), \Gamma \vdash \mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2 \colon \tau$.
  4. By (3) and DisjElim we get $([\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge [\![\Gamma \vdash p.e_2 \colon \alpha_0 \to \tau]\!] \wedge \not\in(\alpha_0)) \vee_{\mathcal{E}} ([\![\Gamma \vdash e_1 \colon \alpha_0]\!] \wedge [\![\Gamma \vdash Kp.e_2 \colon \alpha_0 \to \tau]\!]), \Gamma \vdash \mathbf{let}\, p = e_1 \,\mathbf{in}\, e_2 \colon \tau$ 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 3.1 (Correctness) Case: e 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 $[\![\Gamma \vdash e_1 \colon \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 \colon \alpha]\!] \wedge \not\in(\alpha)$, $\Gamma \vdash e_1\, e_2 \colon \tau$.
  4. By Hide using (1), $[\![\Gamma \vdash e_1\, e_2 \colon \tau]\!], \Gamma \vdash e_1\, e_2 \colon \tau$.
- Theorem 3.1 (Correctness) Case: e is $\lambda_K \bar{c}$.
  1. Let $\alpha_0, \alpha_1 \# \mathrm{FV}(\Gamma, \tau)$.
  2. By the induction hypothesis, we have $[\![\Gamma \vdash \lambda\bar{c} \colon \tau]\!], \Gamma \vdash \lambda\bar{c} \colon \tau$.
  3. By weakening, Equ, Hide and ExAbs, we have $[\![\Gamma \vdash \lambda\bar{c} \colon \tau]\!] \wedge \mathrm{RetType}(\tau, \alpha_0) \wedge \exists \alpha_1 . \alpha_0 \doteq \varepsilon_K(\alpha_1), \Gamma \vdash \lambda_K \bar{c} \colon \tau$.
  4. By weakening, this yields the goal.
- Theorem 3.2 (Completeness) Case ExIntro: premise C, Γ, $\Sigma' \vdash n(e) \colon \tau$ 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 3.2 (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 $C \vDash \not\in(\tau')$.
  2. Pick w.l.o.g. $\alpha \notin \mathrm{FV}(C, \tau', \Gamma, \tau)$. (1) implies $C \wedge \alpha \doteq \tau' \vDash \not\in(\alpha)$. 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 $\mathcal{I}_1\mathcal{I}_2, \exists\alpha.(C \wedge \alpha \doteq \tau') \vDash \exists\alpha.(\Phi_1 \wedge \Phi_2 \wedge \nexists(\alpha))$.
6. By (4) and (5), we have the goal $\mathcal{I}, C \vDash [\![\Gamma \vdash e_1\, e_2 : \tau]\!]$ with $\mathcal{I} = \mathcal{I}_1\mathcal{I}_2$.

- Theorem 3.2 (Completeness) Case LETIN: premise $C, \Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 : \tau$.
  1. LETIN's premise is: $C, \Gamma \vdash \lambda(p.e_2)\, e_1 : \tau$,
  2. derived by APP and ABS from $C, \Gamma \vdash p.e_2 : \tau' \to \tau$, $C, \Gamma \vdash e_1 : \tau'$ and $C \vDash \nexists(\tau')$.
  3. Inductive hypothesis gives $\mathcal{I}_1, C \vDash [\![\Gamma \vdash p.e_2 : \tau' \to \tau]\!]$ and $\mathcal{I}_2, C \vDash [\![\Gamma \vdash e_1 : \tau']\!]$.
  4. (1) and (3) imply $\mathcal{I}_1, C \vDash [\![\Gamma \vdash p.e_2 : \tau' \to \tau]\!] \wedge \nexists(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash Kp.e_2 : \alpha_0 \to \tau]\!]$ as the first disjunct holds.
  5. As the premise $\mathrm{PV}(C, \Gamma) = \varnothing$ gives disjoint domains for the $\mathcal{I}_i$, we have $\mathcal{I}$, $C \vDash [\![\Gamma \vdash e_1 : \tau']\!] \wedge ([\![\Gamma \vdash p.e_2 : \tau' \to \tau]\!] \wedge \nexists(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash Kp.e_2 : \tau' \to \tau]\!])$ for $\mathcal{I} = \mathcal{I}_1\mathcal{I}_2$.
  6. $\mathcal{I}, C \vDash \exists\alpha_0.[\![\Gamma \vdash e_1 : \alpha_0]\!] \wedge ([\![\Gamma \vdash p.e_2 : \alpha_0 \to \tau]\!] \wedge \nexists(\alpha_0) \vee_{\mathcal{E}} [\![\Gamma \vdash Kp.e_2 : \alpha_0 \to \tau]\!])$ by abstracting $\alpha_0 = \tau'$.
- Theorem 3.2 (Completeness) Case EXLETIN:
  premise $C, \Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 : \tau$.
  1. EXLETIN's premises are: $C, \Gamma \vdash Kp.e_2 : \tau' \to \tau$ and $C, \Gamma \vdash e_1 : \tau'$,
  2. Inductive hypothesis gives $\mathcal{I}_1, C \vDash [\![\Gamma \vdash Kp.e_2 : \tau' \to \tau]\!]$ and $\mathcal{I}_2, C \vDash [\![\Gamma \vdash e_1 : \tau']\!]$.
  3. (3) implies $\mathcal{I}_1, C \vDash [\![\Gamma \vdash p.e_2 : \tau' \to \tau]\!] \wedge \nexists(\tau') \vee_{\mathcal{E}} [\![\Gamma \vdash Kp.e_2 : \tau' \to \tau]\!]$ as one of the $\vee_{\mathcal{E}}$ disjuncts holds. The proof concludes as in the LETIN case.
- Theorem 3.2 (Completeness) Case EXABS: premise $C, \Gamma \vdash \lambda_K \bar{c} : \tau$.
  1. EXABS's premises are: *(a)* $C, \Gamma \vdash \lambda\bar{c} : \tau$ and *(b)* $C \vDash \mathrm{RetType}(\tau, \varepsilon_K(\bar{\tau}'))$.
  2. Inductive hypothesis gives $\mathcal{I}_1, C \vDash [\![\Gamma \vdash \lambda\bar{c} : \tau]\!]$.
  3. Let α_0, $\alpha_1 \# \mathrm{FV}(\Gamma, \tau)$. (1b) and (2) give $\mathcal{I}_1$, $C \vDash \exists\alpha_0.[\![\Gamma \vdash \lambda\bar{c} : \tau]\!] \wedge \mathrm{RetType}(\tau, \alpha_0) \wedge \exists\alpha_1.\alpha_0 \doteq \varepsilon_K(\alpha_1) \wedge \alpha_1 \doteq \bar{\tau}'$.
  4. Let $\mathcal{I}_0 = [\chi_K(\alpha) := \alpha \doteq \bar{\tau}']$. By (3), $\mathcal{I}_0\mathcal{I}_1$, $C \vDash \exists\alpha_0.\mathrm{RetType}(\tau, \alpha_0) \wedge (\exists\alpha_1.\alpha_0 \doteq \varepsilon_K(\alpha_1) \wedge \chi_K(\alpha_1)) \wedge [\![\Gamma \vdash \lambda\bar{c} : \tau]\!]$ which is the goal. □

A.1.4. Semantics by Reduction to HMG(X)

By induction on the structure of the derivation, we can show the following:

PROPOSITION A.10. *Let $\mathcal{I}$ be an interpretation of predicate variables for formula C, as in Definition A.1. If a typing judgment $C, \Gamma, \Sigma \vdash e : \tau$ is derivable in the type system* $\mathrm{MMG}_{\exists}(X)$*, then $\mathcal{I}(C), \mathcal{I}(\Gamma), \mathcal{I}(\Sigma) \vdash e : \tau$ is derivable in* $\mathrm{MMG}_{\exists}(X)$.

Proof of Theorem 3.11.

Proof. Let $\mathcal{I}$ be a subsitution of predicate variables such that $\mathcal{M}, \mathcal{I} \vDash \exists \mathrm{FV}(\tau).[\![\Gamma, \Sigma \vdash e : \tau]\!]$. By Theorems 3.1 and 3.7, there exists a derivation of $[\![\Gamma, \Sigma \vdash e : \tau]\!], \Gamma, \Sigma \vdash e : \tau$. Without loss of generality, let ExIntro be the rule applied at the root of the derivation, and let Δ be the derivation of the premise $[\![\Gamma, \Sigma \vdash e : \tau]\!], \Gamma, \Sigma'' \vdash n(e) : \tau$. Let $\Gamma' = \mathcal{I}(\Gamma)$, $C = \mathcal{I}([\![\Gamma, \Sigma \vdash e : \tau]\!])$ and $\Sigma' = \mathcal{I}(\Sigma'')$. By Proposition A.10, there exists a derivation Δ of $C, \Gamma', \Sigma' \vdash e : \tau$. We need to construct a derivation Δ' of $C', \Gamma' \vdash e' : \tau$ under constructor environment Σ' in the HMG(X) type system. The tag erasure of e' w.r.t. Σ has to be computationally equivalent to the HMG-form of e and C' has to be satisfiable. We will transform Δ into a derivation in HMG(X) while preserving these conditions. We transform the resulting constraint, the resulting expression and the derivation simultaneously, as follows:

- For a derivation node applying rule App, we erase the $C \vDash \not\Subset(\tau')$ condition in the premise of the node, and we erase $\not\Subset(\alpha)$ in the corresponding $[\![\Gamma \vdash e_1\, e_2 : \tau]\!]$ part of the resulting constraint, where C, τ', α etc. name the actual pieces of the derivation or constraint.
- For a derivation node Abs, with $\lambda(\overline{p_i.e_i})$ in conclusion, we erase derivation subtrees with NegClause or FailClause nodes at the root. Correspondingly, we replace the expression in the conclusion by $\lambda(\overline{(p_i.e_i')}_{i:\neg \mathrm{unreach}(e_i') \vee \mathrm{failure}(e_i)})$, which preserves computational equivalence; and we erase the $[\![\Gamma \vdash p_i.e_i : \tau_1 \to \tau_2]\!]$ conjuncts of the resulting constraint for i: $\mathrm{unreach}(e_i) \vee \mathrm{failure}(e_i)$, which leads to a weaker formula and thus preserves satisfiability.
- We replace derivation nodes ExLetIn by applications of the App rule to the premises $C, \Gamma \vdash (Kp.e_2)\, e_1 : \tau' \to \tau$ and $C, \Gamma \vdash e_1 : \tau'$, and replace the corresponding subexpressions $\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2$ of the resulting expression by $(Kp.e_2)\, e_1$. We replace the $[\![\Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 : \tau]\!]$ part of the resulting constraint by $\exists \alpha_0.[\![\Gamma \vdash e_1 : \alpha_0]\!] \wedge [\![\Gamma \vdash Kp.e_2 : \alpha_0 \to \tau]\!]$. To see that the satisfiability of the constraint is preserved, recall that $[\![\Gamma \vdash \mathbf{let}\, p = e_1\, \mathbf{in}\, e_2 : \tau]\!]$ is a disjunction where only one disjunct is consistent with $\exists \bar{\alpha}.\tau' \doteq \varepsilon_K(\bar{\alpha})$. The tag erasure of $(Kp.e_2)\, e_1$ w.r.t. the original constructor environment Σ is computationally equivalent to $\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2$.
- We excise derivation nodes ExAbs, and remove the corresponding RetType atom from C, preserving computational equivalence and satisfiability.
- For remaining nodes are easy to rearrange into applications of the corresponding HMG(X) rules. □

Proof of Corollary 3.12.

Proof. The semantics of expression e in MMG$_\exists(X)$ is given by the semantics of its HMG-form in HMG(X). By well-typedness and closedness, we have $C, \varnothing, \Sigma \vdash e : \sigma$ is derivable for some type scheme σ, and $\mathcal{M} \vDash C$. By Theorems 3.2 and 3.7, there exists an interpretation of predicate variables $\mathcal{I}$ such that $\mathcal{M}, \mathcal{I} \vDash C \Rightarrow [\![\varnothing, \Sigma \vdash e : \sigma]\!]$, thus $\mathcal{M}, \mathcal{I} \vDash [\![\varnothing, \Sigma \vdash e : \sigma]\!]$.

By Theorem 3.11, we have an HMG(X) environment Γ', constructor environment Σ' and an expression e' such that $C, \Gamma' \vdash e' : \tau$ is derivable in HMG(X) for a valid C, and the tag erasure of e' w.r.t. Σ is computationally equivalent to the HMG-form of e. By [47] Theorem 3.31, see also Theorem 2.3, e' does not go wrong. Given the call-by-value semantics presented in [47], it is easy to see that the tag erasure of e' does not go wrong. By computational equivalence, HMG-form of e does not go wrong. □

A.2. Constraint Abduction

A.2.1. Formulating the Joint Constraint Abduction Problem

PROPOSITION A.11. Solved form property for terms. *Let $\bar{\beta} = \mathrm{Dom}(\boldsymbol{U}(C))$ be all variables occurring on the left-hand sides of solved form equations $\boldsymbol{U}(C)$, and $\alpha \in \mathrm{FV}(\mathrm{Image}(\boldsymbol{U}(C))$ be a variable occurring on the right-hand side. If $T \vDash \mathcal{Q}.C$ (equivalently $\mathcal{Q}.\boldsymbol{U}(C)$) holds, then $T \vDash \mathcal{Q}.C[\alpha := \tau]$ (equivalently $T \vDash \mathcal{Q}.\boldsymbol{U}(C)[\alpha := \tau]$), for any τ such that for all $\alpha' \in \mathrm{FV}(\tau)$, $\alpha' \leqslant_{\mathcal{Q}} \alpha$.*

A.2.2. Abduction Algorithm for The Combination of Domains

LEMMA A.12. *For any conjunction of atoms $D \in \mathcal{L}_{s_{\mathrm{type}}}$, let D^t be D with all alien subterms $\bar{r}$ replaced with fresh variables $\bar{\alpha_r}$, $D^t \in \mathcal{L}_{\mathrm{ty}}$. Let $\wedge_s D^t_s = \boldsymbol{U}(D^t)$ be a solved form with $D^t_s \in \mathcal{L}_s$. Observe that for any Ψ, for any $C \in \mathcal{L}_s$, if $\mathcal{M} \vDash D \wedge \Psi \Rightarrow C$, then*

1. *$\mathcal{M} \vDash D^t_s[\bar{\alpha_r} := \bar{r}] \wedge \Psi \Rightarrow C$ for $s \neq s_{\mathrm{type}}$.*
2. *For $s = s_{\mathrm{type}}$, let $\wedge_s C^t_s = \boldsymbol{U}(C^t)$, where C^t is C with all alien subterms $\bar{r}'$ replaced with fresh variables $\bar{\alpha_r}'$. Then there exist a conjunction of equations E over $\bar{\alpha_r}\bar{\alpha_r}'$ such that $T \vDash D^t_{s_{\mathrm{type}}} \wedge E \wedge \Psi \Rightarrow C^t_{s_{\mathrm{type}}}$ and (for $s \neq s_{\mathrm{type}}$) $\mathcal{M} \vDash D^t_s[\bar{\alpha_r} := \bar{r}] \wedge \Psi \wedge \bar{\alpha_r^s} \doteq \bar{r^s} \Rightarrow E_s$ where E_s are $E \cap \mathcal{L}_s$.*

The proof uses the *type conservation* and *free generation* (s_{type} properties) and interpolation techniques. Below follows a sketch of a proof of (2).

Proof. Observe a proof of $D^t[\bar{\alpha_r} := \bar{r}] \wedge \Psi \Rightarrow C^t[\bar{\alpha_r}' := \bar{r}']$ (assuming a complete proof system for $\mathcal{M} \vDash$). It can be transformed into a proof of $D^t \wedge \Psi \wedge \bar{\alpha_r} \doteq \bar{r} \wedge \bar{\alpha_r}' \doteq \bar{r}' \Rightarrow C^t$. By s_{type} properties we get $D^t_{s_{\mathrm{type}}} \wedge \Psi \wedge E_0 \wedge \bar{\alpha_r} \doteq \bar{r} \wedge \bar{\alpha_r}' \doteq \bar{r}' \Rightarrow C^t_{s_{\mathrm{type}}}$ where E_0 are equations among $\bar{\alpha_r}$. By interpolation we get $D^t_{s_{\mathrm{type}}} \wedge E \wedge \Psi \Rightarrow C^t_{s_{\mathrm{type}}}$ and $D^t \wedge \Psi \wedge \bar{\alpha_r} \doteq \bar{r} \wedge \bar{\alpha_r}' \doteq \bar{r}' \Rightarrow E$ for $E_0 \subseteq E$ and E as required by the theorem, since $\bar{\alpha_r}$ are the only non-s_{type} subterms of $C^t_{s_{\mathrm{type}}}$. □

Proof of Theorem 4.5.

Proof. We use the same notation as in Table 4.1. Note, that the JCAQP with a branch with unsatisfiable conclusion does not have an answer, so we do not consider unsatisfiable conclusions.

Observe that validity of $\mathcal{L}_{\mathrm{ty}}$ formula in T is equivalent to its validity in $\mathcal{M}$. We will therefore sometimes drop prefixes $\mathcal{M} \vDash$ and $T \vDash$ for readability.

Let us check correctness, i.e. that $\exists \overline{\alpha_{\mathrm{ans}}}.A_{\mathrm{ans}} \in \mathrm{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$ are $\mathrm{JCAQP}_{\mathcal{M}}$ answers.

1. We need to show: $\mathcal{M} \vDash \wedge_i (D_i \wedge A_{\mathrm{ans}} \Rightarrow C_i)$,
2. $\mathcal{M} \vDash \mathcal{Q}.A_{\mathrm{ans}}[\bar{\alpha}_{\mathrm{ans}}\bar{\beta} := \bar{t}]$ for some $\bar{t}$,
3. $\mathcal{M} \vDash \wedge_i \exists \mathrm{FV}(D_i \wedge A_{\mathrm{ans}}).D_i \wedge A_{\mathrm{ans}}$,

4. for all atoms $c \in A_{\text{ans}}$ such that $\mathcal{M} \not\models \mathcal{Q}.c$ and $\text{FV}(c) \cap \bar{\beta} \neq \varnothing$, then for all $\beta_1 \in \text{FV}(c)$ such that $(\forall \beta_1) \in \mathcal{Q}$, there exists $\beta_2 \in \text{FV}(c) \cap \bar{\beta}$ such that $\beta_1 \leqslant_{\mathcal{Q}} \beta_2$.
5. We have: $\mathcal{M} \models \wedge_i (D^t_{i,s_{\text{type}}} \wedge A^j_T \Rightarrow C^t_{i,s_{\text{type}}})$,
6. $\mathcal{M} \models \mathcal{Q}.A^j_T[\bar{\alpha}^T_j \bar{\alpha}_r \bar{\beta} := \bar{t}^T_j]$ for some $\bar{t}^T_j$,
7. $\mathcal{M} \models \wedge_i \exists \text{FV}(D^t_{i,s_{\text{type}}} \wedge A^j_T).D^t_{i,s_{\text{type}}} \wedge A^j_T$,
8. for all atoms $\beta_2 \doteq t \in A^j_T$ such that $\beta_2 \in \bar{\alpha}_r \bar{\beta}$, if $\beta_1 \in \text{FV}(c)$ such that $(\forall \beta_1) \in \mathcal{Q}$, then $\beta_1 \leqslant_{\mathcal{Q}} \beta_2$.
9. We have: $\mathcal{M} \models \wedge_i (D^s_i \wedge (D^t_{i,s} \wedge A^i_{p,j,s})[\bar{\alpha}_r := \bar{r}] \wedge A^{k^s_j}_s \Rightarrow C^s_i \wedge (C^t_{i,s} \wedge A^i_{c,j,s})[\bar{\alpha}_r := \bar{r}])$,
10. $\mathcal{M} \models \mathcal{Q}.A^{k^s_j}_s \Big[\overline{\alpha^{k^s_j}_s} \bar{\alpha}^j_r \bar{\beta} := \bar{t}_{k^s_j} \Big]$ for some $\bar{t}_{k^s_j}$,
11. $\mathcal{M} \models \wedge_i \exists \text{FV}(D^s_i \wedge (D^t_{i,s} \wedge A^i_{p,j,s})[\bar{\alpha}_r := \bar{r}] \wedge A^{k^s_j}_s).D^s_i \wedge (D^t_{i,s} \wedge A^i_{p,j,s})[\bar{\alpha}_r := \bar{r}] \wedge A^{k^s_j}_s$,
12. for all atoms $c \in A^{k^s_j}_s$ such that $\mathcal{M} \not\models \mathcal{Q}.c$ and $\text{FV}(c) \cap \bar{\beta} \neq \varnothing$, then for all $\beta_1 \in \text{FV}(c)$ such that $(\forall \beta_1) \in \mathcal{Q}$, there exists $\beta_2 \in \text{FV}(c) \cap \bar{\beta}$ such that $\beta_1 \leqslant_{\mathcal{Q}} \beta_2$.
13. We have: $D_i \equiv \wedge_s D^s_i$ and $C_i \equiv \wedge_s C^s_i$; $D^t_i[\bar{\alpha}_r := \bar{r}] = D^{s_{\text{type}}}_i$, $C^t_i[\bar{\alpha}_r := \bar{r}] = C^{s_{\text{type}}}_i$; $A^i_{p_j} = \{x \doteq t \in \boldsymbol{U}(D^t_{i,s_{\text{type}}} \wedge A^{j\prime}_T) | x \in X_s, s \neq s_{\text{type}}\}$ and $A^i_{cj} = \{x \doteq t \in \boldsymbol{U}(D^t_{i,s_{\text{type}}} \wedge C^t_{i,s_{\text{type}}} \wedge A^{j\prime}_T) | x \in X_s, s \neq s_{\text{type}}\}$, $A^i_{p/c,j} = \wedge_s A^i_{p/c,j,s}$, where $A^i_{p/c,j,s} \in \mathcal{L}_s$.
14. $D_i \wedge \bar{r_i} \doteq \overline{\alpha_{r_i}} \Rightarrow \wedge_s D^s_i \wedge D^t_{i,s}$, $\wedge_s C^s_i \wedge C^t_{i,s} \wedge \bar{r_i} \doteq \overline{\alpha_{r_i}} \Rightarrow C_i$, by (13).
15. We have: $A_{\text{ans}} = A^{j\prime}_T \wedge_{s \in \text{usorts}} A^{k^s_j}_s$ for some $j, \bar{k^s_j}$ and $\bar{\alpha}_{\text{ans}} = \overline{\alpha^T_j}' \overline{\overline{\alpha^{k^s_j}_s}} \cap \text{FV}\big(A^{j\prime}_T \wedge_s A^{k^s_j}_s\big)$, $\overline{\overline{\alpha^{k^s_j}_s}}$ is a concatenation of $\overline{\alpha^{k^s_j}_s}$ for $s \in$ usorts.
16. By type preservation and free generation, (5) we can have $D^t_{i,s_{\text{type}}} \wedge A^{j\prime}_T \Rightarrow C^t_{i,s_{\text{type}}}{}'$, where $C^t_{i,s_{\text{type}}}{}'$ differs from $C^t_{i,s_{\text{type}}}$ only at alien subterm positions. Pick $C^t_{i,s_{\text{type}}}{}'$ which differs from $C^t_{i,s_{\text{type}}}$ on the least number of alien subterm positions.
17. Consider $S = \{x \doteq t \in \boldsymbol{U}(C^t_{i,s_{\text{type}}} \wedge C^t_{i,s_{\text{type}}}{}') | x \in X_s, s \neq s_{\text{type}}\}$. Note that $S \wedge C^t_{i,s_{\text{type}}}{}' \Rightarrow C^t_{i,s_{\text{type}}}$.
18. By (16) and separation of sorts, we have $\boldsymbol{U}(D^t_{i,s_{\text{type}}} \wedge C^t_{i,s_{\text{type}}} \wedge A^{j\prime}_T) \backslash \mathcal{L}_{s_{\text{type}}} \Rightarrow \boldsymbol{U}(C^t_{i,s_{\text{type}}} \wedge C^t_{i,s_{\text{type}}}{}') \backslash \mathcal{L}_{s_{\text{type}}}$, i.e. $A^i_{cj} \Rightarrow S$.
19. $D_i \wedge A_{\text{ans}} \wedge \bar{\bar{r}} \doteq \bar{\bar{\alpha}}_r \overset{(13,14)}{\Longrightarrow} D^{s_{\text{type}}}_i \wedge_s D^s_i \wedge D^t_{i,s}[\bar{\alpha}_r := \bar{r}] \wedge A^{j\prime}_T \wedge_{s \neq s_{\text{type}}} A^{k^s_j}_s \wedge \bar{\bar{r}} \doteq \bar{\bar{\alpha}}_r$
20. $\ldots \overset{(9,13)}{\Longrightarrow} D^t_{i,s_{\text{type}}} \wedge A^{j\prime}_T \wedge_{s \neq s_{\text{type}}} C^s_i \wedge C^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge A^i_{c,j,s}[\bar{\bar{\alpha}}_r := \bar{\bar{r}}] \wedge \bar{\bar{r}} \doteq \bar{\bar{\alpha}}_r$
21. $\ldots \overset{(17-18)}{\Longrightarrow} C^t_{i,s_{\text{type}}} \wedge_{s \neq s_{\text{type}}} C^s_i \wedge C^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge \bar{\bar{r}} \doteq \bar{\bar{\alpha}}_r \overset{(13)}{\Longrightarrow} C_i$.
22. (19)-(21) and interpolation give the goal (1).
23. From type preservation and free generation properties (as in Proposition A.11), by (6) we get $\mathcal{M} \models \mathcal{Q}.A^j_T[\bar{\alpha}^T_j(\bar{\beta} \cap X_{s_{\text{type}}}) := \bar{t}^T_j; (\bar{\beta} \backslash X_{s_{\text{type}}}) := \bar{t}^b_j; \bar{\alpha}_r := \bar{t}^r_j]$ for some $\bar{t}^T_j$, for all $\bar{t}^r_j, \bar{t}^b_j$.

24. Since $\mathcal{L}_s$ are single-sorted for $s \neq s_{\text{type}}$, by (15), (23) and (10) we get the goal (2).
25. Similarly, from (3), (7), type preservation and free generation properties, and because $\mathcal{L}_s$ are single-sorted for $s \neq s_{\text{type}}$, we get the goal (3).
26. Consider $c \in A_{\text{ans}}$ such that $\mathcal{M} \not\models \mathcal{Q}.c$ and $\text{FV}(c) \cap \bar{\beta} \neq \varnothing$, and a $\beta_1 \in \text{FV}(c)$ such that $(\forall \beta_1) \in \mathcal{Q}$.
27. If $\beta_1 \in \bar{\beta}$, then $\beta_2 = \beta_1$ satisfies the goal (4). Consider the cases where $\beta_1 \notin \bar{\beta}$.
28. Consider $\beta_1 \in X_{s_{\text{type}}}$. By (26) $\mathcal{M} \not\models \mathcal{Q}.c$.
29. By (6) – or (24) – $\mathcal{M} \models \mathcal{Q}.c[\bar{\alpha}_j^T \bar{\beta} := \bar{t}_j^T]$ for some $\bar{t}_j^T$.
30. By (15), c is in solved form $\beta_3 \doteq t$.
31. By (28)-(30), $\beta_3 \in \bar{\beta}$. By (8) we have the goal (4) for case $\beta_1 \in X_{s_{\text{type}}}$.
32. Consider $\beta_1 \in X_s$ for $s \neq s_{\text{type}}$. Because $\mathcal{L}_s$ are single-sorted for $s \neq s_{\text{type}}$, by (15) we have $c \in A_s^{k_j^s}$.
33. From (12) we have the goal (4).

Now we sketch a proof of completeness, i.e. that no JCAQP answer "falls through".

1. We need to show that for any JCAQP $\mathcal{Q}$. $\wedge_i (D_i \Rightarrow C_i)$ with parameters $\bar{\beta}$ answer $\exists \bar{\alpha}.A$, there is an $\exists \overline{\alpha_{\text{ans}}}.A_{\text{ans}} \in \text{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$ and $\bar{t}$ such that $A \Rightarrow A_{\text{ans}}[\overline{\alpha_{\text{ans}}} := \bar{t}]$.
2. Let $A \equiv \wedge_s A_s$ where A_s has atoms of sort s only. Let A' be a formula obtained from $A_{s_{\text{type}}}$ by substituting all alien subterms $\bar{r}'$ of sorts other than s_{type} by fresh variables $\bar{\alpha_r}'$. Let $\wedge_s A'_s = \boldsymbol{U}(A')$ be solved form of A' with $A'_s \in \mathcal{L}_s$.
3. W.l.o.g., all C_i, D_i are satisfiable. Observe, that $\models D_i \wedge A \Rightarrow C_i$ implies, by lemma A.12, that there is a conjunction of equations over variables $\overline{\alpha_{r_i}} \bar{\alpha_r}'$: $A_i'' := \wedge_{s \neq s_{\text{type}}} A''_{i,s}$, such that:
   a. $D_i \wedge A \wedge \overline{\alpha_{r_i}} \bar{\alpha_r}' \doteq \bar{r_i} \bar{r}' \Rightarrow A_i''$,
   b. $D^t_{i,s_{\text{type}}} \wedge A'_{s_{\text{type}}} \wedge A_i'' \Rightarrow C^t_{i,s_{\text{type}}}$,
   c. $D_i^s \wedge D^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge A_s \wedge A'_s[\bar{\alpha_r}' := \bar{r}'] \Rightarrow C_i^s \wedge C^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}]$ for $s \neq s_{\text{type}}$.
4. (3b) and the assumption about Abd_T give that for some $\big(\exists \overline{\alpha_j^T}.A_T^j\big) \in \text{Abd}_T\big(\mathcal{Q}, \bar{\beta}, \overline{D^t_{i,s_{\text{type}}}, C^t_{i,s_{\text{type}}}}\big)$, $A'_{s_{\text{type}}} \wedge A_i'' \Rightarrow A_T^j\big[\overline{\alpha_j^T} := \bar{t_T}\big]$ for some $\bar{t_T}$.
5. (3a), (3c) and lemma A.12 imply, by substituting free variables, $D_i^s \wedge D^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge A_s \wedge A'_s[\bar{\alpha_r}' := \bar{r}'] \Rightarrow C_i^s \wedge C^t_{i,s}[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge A''_{i,s}[\bar{\bar{\alpha}}_r \bar{\alpha_r}' := \bar{\bar{r}} \bar{r}']$.
6. To use the assumption about Abd_s, we weaken (5). With a stronger premise, we also get a stronger conclusion: $D_i^s \wedge (D^t_{i,s} \wedge A^i_{p,j,s} \wedge A^i_{c,j,s})[\bar{\bar{\alpha}}_r := \bar{\bar{r}}] \wedge A_s \wedge A'_s[\bar{\alpha_r}' := \bar{r}'] \Rightarrow C_i^s \wedge (C^t_{i,s} \wedge A^i_{c,j,s})[\overline{\alpha_{r_i}} := \bar{r_i}] \wedge A''_{i,s}[\bar{\bar{\alpha}}_r \bar{\alpha_r}' := \bar{\bar{r}} \bar{r}']$.
7. (6) gives by the assumption about Abd_s: $(A^i_{c,j,s} \backslash A^i_{p,j,s})[\bar{\bar{\alpha}}_r := \bar{\bar{r}}] \wedge A_s \wedge A'_s[\bar{\alpha_r}' := \bar{r}'] \Rightarrow A_s^{k_j^s}\Big[\overline{\alpha_s^{k_j^{\bar{s}}}} := \overline{t_s^{k_j^{\bar{s}}}}\Big]$ for some k_j^s and $\overline{t_s^{k_j^{\bar{s}}}}$.

8. Check using (3b) and (4) that $A'_{s_{\text{type}}} \wedge A''_i \wedge \overline{\alpha_{r_i}}\bar{\alpha_r}' \doteq \bar{r_i}\bar{r}' \Rightarrow \exists \bar{\alpha}^j_r.A^{j\,\prime}_T\big[\overline{\alpha^T_j} := \bar{t^-_T}\big] \wedge (A^i_{c,j,s} \backslash A^i_{p,j,s})[\bar{\alpha_r} := \bar{\bar{r}}]$. The subtraction in $(A^i_{c,j,s} \backslash A^i_{p,j,s})$ allows us to drop $D^t_{i,s_{\text{type}}}$ from the premise in (3b).
9. Select $\exists\overline{\alpha_{\text{ans}}}.A_{\text{ans}} := \exists\overline{\alpha_{\overline{k^s_j}}}.A^{j\,\prime}_T[\bar{\alpha_r} := \bar{\bar{r}}] \wedge_s A^{k^s_j}_s$ for $\overline{\alpha_{\overline{k^s_j}}} := \overline{\alpha^T_j}\bar{\alpha}^j_r\overline{\alpha^{k^s_j}_s} \cap \text{FV}\big(A^j_T[\bar{\alpha_r} := \bar{r}] \wedge_s A^{k^s_j}_s\big)$. By (7) and (8), we have $A \wedge [\bar{\alpha_r}\bar{\alpha_r}' \doteq \bar{\bar{r}}\bar{r}'] \Rightarrow A_{\text{ans}}\Big[\overline{\alpha^T_j}\bar{\alpha}^j_r\overline{\alpha^{k^s_j}_s} := \bar{t^-_T}\bar{t}^j_r\overline{t^{k^s_j}_s}\Big]$ for some $\bar{t}^j_r$, which implies (1). □

A.3. Constraint Generalization

Proposition A.13. *Let $D_s \in \mathcal{L}_s$ for all sorts s such that $D_{s_{\text{type}}} = \boldsymbol{U}(D_{s_{\text{type}}})$, and $C_{s'} \in \mathcal{L}_{s'}$ for $s' \neq s_{\text{type}}$. Then $\mathcal{M} \vDash (\wedge_s D_s) \Rightarrow C_{s'}$ if and only if $\mathcal{M} \vDash D_{s'} \Rightarrow C_{s'}$.*

Proposition A.14. (MGU property.) *MGU: $T(F, X) \vDash C \Leftrightarrow \boldsymbol{U}(C)$.*

Proposition A.15. *Let D be a conjunction of equations with $D = \boldsymbol{U}(D)$, let A be a conjunction of equations and $\bar{\alpha}$ variables such that $\text{FV}(A) \subset \bar{\alpha} \cup \text{FV}(D)$. $T(F, X) \vDash D \Rightarrow \exists\bar{\alpha}.A$ if and only if there exists a substitution $S = [\bar{\alpha} := \bar{g_\alpha}]$ and a solved form $A' \Leftrightarrow A$ with $A' = \boldsymbol{U}(A')$ such that for every $x \doteq t \in A'$, either $S(x) = S(t)$, or $x \doteq S(t) \in D$.*

Proof of Theorem 4.10.

Proof. First, we show $\mathcal{M} \vDash \wedge_i(D_i \Rightarrow \exists\bar{\alpha}.A)$.

1. $\mathcal{M} \vDash D_i \wedge D^a_i \wedge D^u_i \Rightarrow c$, for each conjunct $c \in A_{s_{\text{type}}}$.
   a. Case $c = x_j \doteq g_j$. By properties of MGU.
   b. Case $c \in D^g_i$ follows from $D_i \wedge D^a_i \Rightarrow D^t_{i,s_{\text{type}}}$.
   c. Case $c \in D^v_i$. The premises are: $x \doteq t_1 \in D^g_i$, $y \doteq t_2 \in D^g_i$, $\mathcal{M} \vDash D^g_i \Rightarrow t_1 \doteq t_2$. The goal follows from $D_i \wedge D^a_i \wedge D^u_i \Rightarrow D^g_i$, by transitivity.
2. By properties of LUB_s, we have $\vDash D^s_i \wedge D^{t,s}_i \wedge D_{i,s} \Rightarrow \exists\bar{\alpha}_s.A_s$ for $s \neq s_{\text{type}}$, and therefore $\vDash D_i \wedge D^a_i \wedge D^u_i \Rightarrow \exists\bar{\alpha}_s.A_s$.
3. We have $\vDash D_i \Rightarrow \exists\bar{\alpha^j_i}.D_i \wedge D^a_i$, and by properties of most specific anti-unification, $\vDash D_i \Rightarrow \exists\bar{\bar{\alpha}_j}.D_i \wedge D^u_i$.
4. Collecting the above points, we get $\vDash D_i \Rightarrow \exists\bar{\alpha^j_i}\bar{\alpha_s}.\wedge_s A_s$.

Now, we sketch a proof of: For every $\exists\bar{\alpha}_r.A_r$ such that $\mathcal{M} \vDash \wedge_i(D_i \Rightarrow \exists\bar{\alpha}_r.A_r)$, with variables renamed so that $\bar{\alpha}\#\text{FV}(A_r)$, $\mathcal{M} \vDash A \Rightarrow \exists\bar{\alpha}_r.A_r$.

1. The assumption is $\mathcal{M} \vDash \wedge_i(D_i \Rightarrow \exists\bar{\alpha}_r.A_r)$.
2. By definition, $\mathcal{M} \vDash D_i \wedge D^a_i \Leftrightarrow \wedge_{s \neq s_{\text{type}}} D^s_i \wedge_s D^{t,s}_i \wedge_s D^t_{i,s}$.
3. Let $\exists\bar{\alpha}_r.A_r \Leftrightarrow \exists\overline{\bar{\alpha}^r_{t,s}}.\wedge_s \exists\bar{\alpha}_{r,s}.D^a_{r,s} \wedge A_{r,s}$ where $\exists\bar{\alpha}_{r,s}.A_{r,s} \in \mathcal{L}_s$, $\bar{a}^r_{t,s}$ are all alien subterms of sort s in A_r, $\bar{\alpha}^r_{t,s}$ are fresh variables $\bar{\alpha}^r_{t,s}\#\text{FV}(A_r, \overline{D_i})$ and also all alien subterms in $A_{r,s_{\text{type}}}$, $A_{r,s_{\text{type}}} = \boldsymbol{U}(A_{r,s_{\text{type}}})$, and $D^a_{r,s} = \bar{\alpha}^r_{t,s} \doteq \bar{a}^r_{t,s}$ for $s \neq s_{\text{type}}$, $D^a_{r,s_{\text{type}}} = \boldsymbol{T}$.

4. By Proposition A.15, for each branch i there is a substitution η_i of $\overline{\bar{\alpha}^r_{t,s}}\bar{\alpha}_{r,s_{\text{type}}}$ that is an anti-unifier of $\overline{t'_{i,j}}$, for all $x_j \notin \overline{\bar{\alpha}^r_{t,s}}\bar{\alpha}_{r,s_{\text{type}}}$ such that $x_j \doteq u_{r,j} \in A_{r,s_{\text{type}}}$, where x_j, $\overline{t_{i,j}} \in V$ and $\mathcal{M} \vDash D_i \Rightarrow t'_{i,j} \doteq t_{i,j}$. $t'_{i,j}$ can be made equal to $t_{i,j}$ up to a variable-variable substitution.
   - I.e. a substitution $[\bar{\alpha} := \bar{\beta}]$ that can assign the same β to several α.
5. There exists a substitution ρ which establishes $\mathcal{M} \vDash A_{s_{\text{type}}} \Rightarrow \exists\overline{\bar{\alpha}^r_{t,s}}.\exists\bar{\alpha}_{r,s_{\text{type}}}.\overline{x_j \doteq u_{r,j}}$: by variable-variable substitution, including alien subterm variables, according to $\mathcal{M} \vDash D_i \Rightarrow t'_{i,j} \doteq t_{i,j}$, and by the factoring via most specific anti-unification.
6. Let $\exists\bar{\alpha}_s.A_s = \text{LUB}_s\big(\overline{D^s_i \wedge \widetilde{D^a_i}(D^t_{i,s} \wedge D^u_{i,s})}\big)$ as in the definition of LUB$(\cdot)$.
7. $\mathcal{M} \vDash D_i \Rightarrow \exists\bar{\alpha}_r.A_r$ by (1), $\mathcal{M} \vDash D_i \wedge D^a_i \wedge D^u_{i,s} \Rightarrow \exists\bar{\alpha}_r.A_r$ by weakening on the left, $\mathcal{M} \vDash D_i \wedge D^a_i \wedge D^u_{i,s} \Rightarrow \exists\overline{\bar{\alpha}^r_{t,s}}.\wedge_s \exists\bar{\alpha}_{r,s}.D^a_{r,s} \wedge A_{r,s}$ by (3), $\mathcal{M} \vDash D_i \wedge D^a_i \wedge D^u_{i,s} \Rightarrow \exists\bar{\alpha}_{r,s}.A_{r,s}$ by weakening on the right, $\mathcal{M} \vDash D^s_i \wedge D^{t,s}_i \wedge D^t_{i,s} \wedge D^u_{i,s} \Rightarrow \exists\bar{\alpha}_{r,s}.A_{r,s}$ by (2) and Proposition A.13.
8. By (6), (7), interpolation and assumption about LUB$_s$, $\mathcal{M} \vDash A_s \Rightarrow \exists\bar{\alpha}_{r,s}.A_{r,s}$ for every $s \neq s_{\text{type}}$.
9. More specifically, we need $\mathcal{M} \vDash A_s \wedge A_{s_{\text{type}}} \Rightarrow \exists\bar{\alpha}_{r,s}.A_{r,s} \wedge \rho(\overline{\alpha^r_{t,s}}) \doteq \overline{a^r_{t,s}}$. It can be shown by extending the argument for (8) using the fact that $D^u_{i,s}$ relates $\rho(\overline{\alpha^r_{t,s}})$ introduced in (5) to the remaining constraints of branch i. □

A.4. Solving for Predicate Variables

In this section, we provide proof sketches for correctness and a limited form of completeness of the type inference implemented in InvarGenT, correctness with respect to the type system MMG$_\exists(X)$ and the limited form of completeness wrt. MMG(X).

LEMMA A.16. *Let* $(\mathcal{Q}', \bar{\alpha}_{\text{res}}, A_{\text{res}}, \overline{\exists\bar{\alpha}^\chi.A_\chi}) \in \text{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^\chi}\big)$. *Then:*

1. $\mathcal{M} \vDash A_{\text{res}} \wedge_\chi A_\chi \Rightarrow A$.
2. $\mathcal{M} \vDash A \Rightarrow \exists\overline{\bar{\alpha}^\chi_+}.A_{\text{res}} \wedge_\chi A_\chi$.
3. $\mathcal{M} \vDash \mathcal{Q}'.A_{\text{res}}[\bar{\alpha}_{\text{res}} := \bar{t}]$ *for some* $\bar{t}$.
4. *If A only restricts $\bar{\beta}^\chi$ in ways that are expressible as $\mathcal{Q}_{<\beta_\chi}\exists\bar{\beta}^\chi\bar{\beta}^\chi_1.\varphi$ for some $\bar{\beta}^\chi_1$, and $\mathcal{M} \vDash \mathcal{Q}.A\big[\bar{\alpha}\overline{\bar{\beta}^\chi} := \bar{t}\,\big]$ for some $\bar{t}$, then* $\text{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^\chi}\big) \neq \varnothing$.
5. Split *preserves atomized form: if $\exists\bar{\alpha}.A$ is in atomized form with respect to $\mathcal{Q}$ and parameters $\overline{\bar{\beta}^\chi}$, then $\exists\bar{\alpha}_{\text{res}}.A_{\text{res}}$ is in atomized form with respect to $\mathcal{Q}'$ and parameters $\overline{\bar{\alpha}^\chi\bar{\beta}^\chi}$.*

Proof. Recall the definition of Split and the notation used there.

(1) and (2) follow from the observation that $A^L_\chi \wedge A^R_\chi \Rightarrow A^+_\chi$ and $A^+_\chi \Rightarrow \exists\overline{\bar{\alpha}^\chi_+}.A^L_\chi \wedge A^R_\chi$.

(3) follows from $\mathcal{M} \vDash \mathcal{Q}.(A \setminus \cup_\chi A^+_\chi)[\bar{\alpha}_{\text{res}} := \bar{t}]$ for some $\bar{t}$ because $\cup_\chi\bar{\alpha}^\chi = \varnothing$ and $A_{\text{res}} = A \setminus \cup_\chi A_\chi$, in the last iteration.

For (4) to hold the algorithm should at each recursion level be able to find $\overline{A^+_\chi}$ for which $\mathcal{M} \vDash \mathcal{Q}.(A \setminus \cup_\chi A^+_\chi)[\bar\alpha := \bar t]$ for some $\bar t$ and $\mathrm{Strat}(A^+_\chi, \bar\beta^\chi)$ does not return $\bot$ for any χ, where $\mathcal{Q}$ and $\bar\beta^\chi$ are either the initial arguments or the recursive call arguments. The algorithm terminates because $\bar\alpha$ always decreases in the recursive call.

A^+_χ contains restrictions on the variables $\bar\beta^\chi$. If $\mathrm{Strat}(A^+_\chi, \bar\beta^\chi)$ returns $\bot$, then, because A^+_χ is in atomized form, A^+_χ restricts $\bar\beta^\chi$ in a way not expressible as $\mathcal{Q}_{<\beta_\chi}\exists\bar\beta^\chi\bar\beta^\chi_1.\varphi$.

The final remark follows from the minimality of $\overline{A^+_\chi}$ w.r.t. inclusion. □

Lemma A.17. *Let* $\Phi \in \mathcal{L}$, $S = \overline{\exists\bar\alpha^\chi.F_\chi}$ *and* $R = \overline{\exists\bar\alpha^{\chi\kappa}.F_{\chi\kappa}}$. *Let* $\mathrm{NF}(R\,S(\Phi)) = \mathcal{Q}.\wedge_i (D_i \Rightarrow C_i) = \mathcal{Q}.\Phi_{\mathrm{PN}} \in \mathcal{L}$.

Let $(\exists\bar\alpha'_{\mathrm{res}}.F'_{\mathrm{res}}, S', R') \in \Psi(k, \Phi, S, R)$ *for any* k. *Let*

$$\mathcal{Q}'.\Phi'_{\mathrm{PN}} = \mathrm{NF}(R'^{-}\,S'^{-}\,R^+\,S^+(\Phi))$$

Then $\mathcal{M} \vDash F'_{\mathrm{res}} \Rightarrow \Phi'_{\mathrm{PN}}$ *and* $\mathcal{M} \vDash \mathcal{Q}'.F'_{\mathrm{res}}[\bar\alpha_{\mathrm{res}} := \bar t]$ *for some* $\bar t$.

Proof. We use the notation from the definition of Ψ, with $S_k = S, R_k = R, \mathcal{Q}' = \mathcal{Q}^{k+1}$. We have

$$\exists\bar\alpha.A_0 \in \mathrm{Abd}\big(\mathcal{Q}^{k'}, \overline{\beta_\chi\bar\beta^\chi}, \overline{D^{k'}_i, C^{k'}_i}\big)$$

and therefore *(1)* $\mathcal{M} \vDash \wedge_i(D^{k'}_i \wedge A_0 \Rightarrow C^{k'}_i)$. Continuing the definition of Ψ, we have

$$\big(\mathcal{Q}^{k+1}, \bar\alpha_{\mathrm{res}}, A_{\mathrm{res}}, \overline{\exists\bar\alpha^{\beta_\chi}.A_{\beta_\chi}}\big) \in \mathrm{Split}\big(\mathcal{Q}^{k'}, \bar\alpha, A, \overline{\beta_\chi\bar\beta^\chi}\big)$$

and $S'(\chi) = \exists\bar\beta^{\chi,k}.\mathrm{Simpl}\big(\exists\overline{\bar\alpha^{\beta_\chi}}.F'_\chi \wedge A_{\beta_\chi}\big[\overline{\beta_\chi\bar\beta^\chi} := \overline{\delta\bar\beta^{\chi,k}}\big]\big)$, $F'_{\mathrm{res}} = A_{\mathrm{res}}$. Therefore by Lemma A.16 point 1, *(2)* $\mathcal{M} \vDash F'_{\mathrm{res}} \wedge_{\beta_\chi} A_{\beta_\chi} \Rightarrow A$, and by Lemma A.16 point 3, the goal $\mathcal{M} \vDash \mathcal{Q}'.F'_{\mathrm{res}}[\bar\alpha_{\mathrm{res}} := \bar t]$ for some $\bar t$.

It remains to show *(3)* $\mathcal{M} \vDash F'_{\mathrm{res}} \Rightarrow \Phi'_{\mathrm{PN}}$. Observe that *(4)* $\Phi'_{\mathrm{PN}} \equiv (\wedge_{\beta_\chi} A_{\beta_\chi} \Rightarrow \Phi_{\mathrm{PN}} \wedge_{\chi\kappa} U_{\chi\kappa})$. (1) is $\mathcal{M} \vDash A_0 \Rightarrow \Phi_{\mathrm{PN}} \wedge_{\chi\kappa} U_{\chi\kappa}$. But $F'_{\mathrm{res}} \wedge_\chi A_{\beta_\chi} \Rightarrow A_0$ by (2), so (1) and (4) give (3). □

Sketch of proof of Theorem 4.15.

Proof. Lemma A.17 gives the goal $\mathcal{M}, \mathcal{I} \vDash \Phi$ with $\mathcal{I}(\Phi) \equiv \mathcal{Q}'.\Phi'_{\mathrm{PN}}$, because $S'\,R' = S\,R$ implies $\wedge_{\beta_\chi} A_{\beta_\chi} = \boldsymbol{T}$. □

Axiom A.18. *(Interpolation property for* $\mathcal{M}$*.) We assume that for conjunctions of atoms* A *and any quantifier-free formula* Φ, $\mathcal{M} \vDash A \Rightarrow \Phi$ *implies that there is a conjunction of atoms* B *with* $\mathrm{FV}(B) \subset \mathrm{FV}(A) \cap \mathrm{FV}(\Phi)$, $\mathcal{M} \vDash A \Rightarrow B$ *and* $\mathcal{M} \vDash B \Rightarrow \Phi$.

Lemma A.19. *Let* $\mathrm{NF}(\Phi[\overline{\chi(\tau)} := \overline{\exists\bar\alpha^\chi.F_\chi[\delta := \tau]}]) = \mathcal{Q}.\Phi_N$ *and* $\mathrm{PV}(\Phi) = \mathrm{PV}^1(\Phi)$. *Let*

$$\mathcal{Q}^\Delta.\Phi^\Delta_N = \mathrm{NF}(\Phi[\overline{\chi^-(\beta^\chi)} := \overline{\exists\bar\alpha^\chi\bar\alpha^\chi_\Delta.(\Delta_\chi \wedge F_\chi)[\delta := \beta^\chi]}; \overline{\chi^+(\tau)} := \overline{\exists\bar\alpha^\chi.F_\chi[\delta := \tau]}])$$

for variables $\overline{\bar\alpha^\chi_\Delta}$ *and conjunctions of atoms* $\overline{\Delta_\chi} \in \mathcal{L}$ *such that the JCAQP*$_\mathcal{M}$ *problem* $\mathcal{Q}^\Delta.\Phi^\Delta_N$ *has a solution. Assume that* Abd *in the definition of* Ψ *is a complete abduction algorithm returning an atomized form formula. Then there is* $(F'_{\mathrm{res}}, \overline{\exists\bar\alpha^{\chi\prime}.F'_\chi}) \in \Psi(k, \Phi, \overline{\exists\bar\alpha^\chi.F_\chi})$ *such that*

$$\mathcal{M} \vDash \wedge_\chi \Delta^\beta_\chi \Rightarrow (\wedge_\chi(F'_\chi \setminus F_\chi))[\overline{\bar\alpha^{\chi\prime} := \bar t}]$$

for all χ, for some $\bar{t}$.

Proof. Let $\exists \bar{\alpha}^d_{\text{res}}.D_{\text{res}}$ be a solution to the JCAQP$_{\mathcal{M}}$ problem $\mathcal{Q}^{\Delta}.\Phi^{\Delta}_N$. We have $\mathcal{M} \vDash D_{\text{res}} \wedge_\chi \Delta^\beta_\chi \Rightarrow \Phi_N$. By the completeness assumption about Abd, we have $\mathcal{M} \vDash D_{\text{res}} \wedge_\chi \Delta^\beta_\chi \Rightarrow A[\bar{\alpha}\overline{\bar{\alpha}^\chi} := \bar{t}]$ for some $\bar{t}$. Since D_{res} does not participate in restricting $\overline{\bar{\beta}^\chi}$ and Δ^β_χ is limited to $\mathcal{Q}_{<\beta_\chi}\bar{\alpha}^\chi\bar{\alpha}^\chi_\Delta$ variables, by atomized form of $\exists\bar{\alpha}.A$ and interpolation, we get the expressiveness constraint needed for Lemma A.16 point 4, therefore $\text{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^\chi}\big) \neq \varnothing$. Thus by Lemma A.16 point 2, $\mathcal{M} \vDash D_{\text{res}} \wedge_\chi \Delta^\beta_\chi \wedge \overline{\bar{\alpha}^{\chi\prime} \doteq \bar{t}} \Rightarrow \exists\overline{\bar{\alpha}^\chi_+}.A_{\text{res}} \wedge_\chi A_\chi$ where by definition of Ψ, $A_\chi = F'_\chi \setminus F_\chi$. Since the atomized form is preserved by Split (Lemma A.16 point 5), we have the goal. □

Sketch of proof of Theorem 4.16.

Proof. The thesis $\mathcal{M} \vDash \wedge_\chi F^s_\chi \Rightarrow (\wedge_\chi F^k_\chi)\big[\overline{\bar{\alpha}^{\chi,k}} := \bar{t}^k\big]$ is true for $k = 0$. Assume it holds for arbitrary k. We need to show $\mathcal{M} \vDash \wedge_\chi F^s_\chi[\delta := \beta_\chi] \Rightarrow (\wedge_\chi F^{k+1}_\chi[\delta := \beta_\chi])\big[\overline{\bar{\alpha}^{\chi,k+1}} := \bar{t}^{k+1}\big]$ for some $\big(F^{k+1}_{\text{res}}, \overline{\exists\bar{\alpha}^{\chi,k+1}.F^{k+1}_\chi}\big)$ and a substitution of variables $\overline{\bar{\alpha}^{\chi,k+1}}$.

By the assumption that $\bar{\chi} = \text{PV}(\Phi) = \text{PV}^1(\Phi)$, definition of Ψ gives $F^{k+1}_\chi = F^k_\chi \wedge A_\chi$. We will apply Lemma A.19 with $\Delta_\chi = F^s_\chi$ and $F_\chi = F^k_\chi$. By the inductive assumption, JCAQP$_{\mathcal{M}}$ answer $\exists\bar{\alpha}^{\text{res}}_s.F^s_{\text{res}}$ to $\text{NF}(\Phi[\overline{\chi(\tau)} := \overline{\exists\bar{\alpha}^\chi_s.F^s_\chi[\delta := \tau]}])$ is also an answer to $\mathcal{Q}^{\Delta}.\Phi^{\Delta}_N$. By Lemma A.19, we get the goal. □

Appendix B
Algorithmic Details

B.1. Generating and Normalizing Formulas

We inject the existential type and value constructors during parsing for user-provided existential types, and during constraint generation for inferred existential types, into the list of toplevel items. It facilitates exporting inference results as OCaml source code.

Toplevel definitions are intended as boundaries for constraint solving. This way the programmer can decompose functions that could be too complex for the solver. A toplevel `let rec` only binds a single identifier, while `let` binds variables in a pattern. To preserve the flexibility of expression-level pattern matching, for `let` – unless it just binds a value as discussed above – we pack the constraints $[\![\Sigma \vdash p{\uparrow}\alpha]\!]$ which the pattern makes available, into existential types. Each pattern variable is a separate entry to the global environment, therefore the connection between them is lost.

The `let...in` syntax has two uses: binding values of existential types means "eliminating the quantification" – the programmer has control over the scope of the existential constraint. The second use is if the value is not of existential type, the constraint is replaced by one that would be generated for a pattern matching branch. This recovers the common use of the `let...in` syntax, with exception of polymorphic `let` cases, where `let rec` needs to be used.

We optimize the disjunctive constraint coming from `let` bindings as follows. Rather than inspecting K in Σ directly, let $[\![\Sigma \vdash Kx{\uparrow}\alpha_0]\!] = \exists\bar{\beta}[D]\{x \mapsto \tau'\}$. $[\![\Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau]\!]$ is equivalent to, or at least implied by:

$$[\![\Sigma \vdash Kp{\downarrow}\alpha_0]\!] \wedge \forall\bar{\beta}.D \Rightarrow [\![\Gamma \vdash p.e_2\colon \tau' \to \tau]\!]$$

Let us set $\boldsymbol{C}_0 = \not\in(\alpha_0)$, $\boldsymbol{C}_K = [\![\Sigma \vdash Kp{\downarrow}\alpha_0]\!]$, $\boldsymbol{D}_0 = \beta_0 \doteq \alpha_0$ and $\boldsymbol{D}_K = \beta_0 \doteq \tau' \wedge D$. Then,

$$\exists\alpha_0.[\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge ([\![\Gamma \vdash p.e_2\colon \alpha_0 \to \tau]\!] \wedge \not\in(\alpha_0) \vee_{\mathcal{E}} [\![\Gamma \vdash Kp.e_2\colon \alpha_0 \to \tau]\!])$$

is equivalent to:

$$\exists\alpha_0.[\![\Gamma \vdash e_1\colon \alpha_0]\!] \wedge \vee_{i \in \{0\} \cup \mathcal{E}}(\boldsymbol{C}_i \wedge \forall\bar{\beta}\beta_0.(\boldsymbol{D}_i \Rightarrow [\![\Gamma \vdash p.e_2\colon \beta_0 \to \tau]\!]))$$

We use the same variable β_0 across disjuncts of the same disjunction, so that $[\![\Gamma \vdash p.e_2\colon \beta_0 \to \tau]\!]$ can be derived and simplified only once.

The second argument of the predicate variable $\chi_K(\gamma, \alpha)$ provides an "escape route" for free variables, i.e. precondition variables used in postcondition. In the implementation, we have user-defined existential types with explicit constraints in addition to inferred existential types. We expand the inferred existential types after they are solved into the fuller format. In the inferred form, the result type has a single parameter δ', without loss of generality because the actual parameters are passed as a tuple type. In the full format we recover after inference, we extract the parameters $\delta' \doteq (\bar{\beta})$, the non-local variables of the existential type, and the partially abstract type $\delta \doteq \tau$, and store them separately, i.e. $\varepsilon_K(\bar{\beta}) = \forall\bar{\beta}\exists\bar{\alpha}[D].\tau$. The variables $\bar{\beta}$ are instantiated whenever the constructor is used. For a toplevel `let`, we form existential types after solving the generated constraint, to have less intermediate variables in them.

Both during parsing and during inference, we inject new structure items to the program, which capture the existential types. During printing existential types in concrete syntax $\exists i{:}\ \bar{\beta}[\varphi].t$ for an occurrence $\varepsilon_K((\bar{r}))$, the variables $\bar{\alpha}$ coming from $\delta' \doteq (\bar{\alpha}) \in \varphi$ are substituted-out by $[\bar{\alpha} := \bar{r}]$.

For simplicity, only toplevel definitions accept type and invariant annotations from the user. The constraints are modified according to the $[\![\Gamma,\, \Sigma \vdash \text{ce}\colon \forall\bar{\alpha}[D].\tau]\!]$ rule. Where `let rec...in` uses a fresh variable β, a toplevel `let rec` incorporates the type from the annotation. The annotation is considered partial, D becomes part of the constraint generated for the recursive function but more constraints will be added if needed. The polymorphism of $\forall\bar{\alpha}$ variables from the annotation is preserved since they are universally quantified in the generated constraint.

The constraints solver returns three components: the *residue*, which implies the constraint when the predicate variables are instantiated, and the solutions to unary and binary predicate variables. The residue and the predicate variable solutions are separated into *solved variables* part, which is a substitution, and remaining constraints (which are currently limited to linear inequalities). To get a predicate variable solution we look for the predicate variable identifier association and apply it to one or two type variable identifiers, which will instantiate the parameters of the predicate variable. We considered several ways to deal with multiple solutions:

1. report a failure to the user;
2. ask the user for decision;
3. silently pick one solution, if the wrong one is picked the subsequent program might fail;
4. perform backtracking search for the first solution that satisfies the subsequent program.

We use approach 3 as it is simplest to implement. Traditional type inference workflow rules out approach 2, approach 4 is computationally too expensive. We might use approach 1 in a future version of the system. Upon "multiple solutions" failure – or currently, when a wrong type or invariant is picked – the user can add `assert` clauses (e.g. `assert false` stating that a program branch is impossible), and `test` clauses. The `test` clauses are boolean expressions with operational semantics of run-time tests: the test clauses are executed right after the definition is executed, and run-time error is reported when a clause returns `false`. The constraints from test clauses are included in the constraint for the toplevel definition, thus propagate more efficiently than backtracking would. The `assert` clauses are: `assert type e1 = e2` which translates as equality of types of `e1` and `e2`, `assert false` which translates as `CFalse`, and `assert num e1 <= e2`, which translates as inequality $n_1 \leqslant n_2$ assuming that `e1` has type `Num n1` and `e2` has type `Num n2`.

We treat a chain of single branch functions with only `assert false` in the body of the last function specially. We put all information about the type of the functions in the premise of the generated constraint. Therefore the user can use them to exclude unintended types. See the example `equal_assert.gadt`.

B.1.1. Normalization

We reduce the constraint to alternation-minimizing prenex-normal form, as in the formalization. We return a variable comparison function. The branches we return from normalization have unified conclusions, since we need to unify for solving disjunctions anyway.

Releasing constraints from under disjunctions is done iteratively, somewhat similar to how disjunction would be treated in constraint solvers. Releasing the sub-constraints is essential for eliminating cases of further disjunction constraints. When at the end more than one disjunct remains, we assume it is the traditional LetIn rule and select its disjunct.

When one $\lambda[K]$ expression is a branch of another $\lambda[K]$ expression, the corresponding branch does not introduce a disjunction constraint – the case is settled syntactically to be the same existential type.

B.1.1.1. Implementation Details

The unsolved constraints are particularly weak with regard to variables constrained by predicate variables. We need to propagate which existential type to select for result type of recursive functions, if any. Normalization starts by flattening constraints into implications with conjunctions of atoms as premises and conclusions, and disjunctions with disjuncts and additional information. The additional information kept with a disjunct is the conjunction of atoms that hold together with the disjunction. We try to eliminate disjuncts by using unification to check for contradiction. If only one disjunct is left, or we decide to pick LetIn anyway (when no progress can be made otherwise), we return the disjunct. Otherwise we return the filtered disjunction.

To help eliminate unintended disjuncts, we collect information about existential return types of recursive definitions by:

1. solving the conclusion of a branch together with additional conclusions, to know the return types of variables,
2. registering existential return types for all variables in the substitution,
3. registering existential return types for all RetType first arguments and their substitution instances,
4. traversing the premise and conclusion to find new variables that are types of recursive definitions,
5. registering as "type of recursive definition" the return types for all variables in the substitution registered as types of recursive definitions,
6. traversing all variables known to be types of recursive definitions, and registering existential type with recursive definition (i.e. unary predicate variable) if it has been learned,
7. traversing all variables known to be types of recursive definitions again, and registering existential type of the recursive definition (if any) with the variable.

B.1.2. Simplification

During normalization, we remove from a nested premise the atoms it is conjoined with (as in "modus ponens").

After normalization, we simplify the constraints by removing redundant atoms. We remove atoms that bind variables not occurring anywhere else in the constraint, and in case of atoms not in premises, not universally quantified. The simplification step is not currently proven correct and might need refining. We merge implications with the same premise, unless one of them is non-recursive and the other is recursive. We call an implication branch recursive when an unary predicate variable χ (not a χ_K) appears in the conclusion *or* a binary predicate variable χ_K appears in the premise.

B.2. Abduction

Our formal specification of abduction provides a scheme for combining sorts that substitutes number sort subterms from type sort terms with variables, so that a single-sort term abduction algorithm can be called. Since we implement term abduction over the multisorted datatype `typ`, we keep these *alien subterms* in terms passed to term abduction.

B.2.1. Abduction for Terms with Alien Subterms

Here we expand on the overview from Section 4.2.2. The JCAQPAS problem is more complex than simply substituting alien subterms with variables and performing joint constraint abduction on resulting implications. The ability to "outsource" constraints to the alien sorts enables more general answers to the target sort, in our case the term algebra $T(F)$. Term abduction will offer answers that cannot be extended to multisort answers.

One might mitigate the problem by preserving the joint abduction for terms algorithm, and after a solution $\exists\bar{\alpha}.A$ is found, "*dissociating*" the alien subterms (including variables) in A as follows. We replace every alien subterm n_s in A (including variables, even parameters) with a fresh variable α_s, which results in A' (in particular $A'[\bar{\alpha_s} := \bar{n_s}] = A$). Subsets $A_p^i \wedge A_c^i = A^i \subset \overline{\alpha_s \doteq n_s}$ such that $\exists\bar{\alpha}\bar{\alpha_s}.A'$, $\overline{A_p^i, A_c^i}$ is a JCAQPAS answer will be recovered automatically by a residuum-finding process after abduction for terms ends. This process is needed regardless of the "dissociation" issue, to uncover the full content of numeric sort constraints.

To face efficiency of numerical abduction with many variables, we modify the approach. On the first iteration of the main algorithm, we remove (purge) alien subterms both from the branches and from the answer, but we do not perform other-sort abduction at all. On the next iteration, we do not purge alien subterms, neither from the branches nor from the answer, as we expect the dissociation in the partial solutions (to predicate variables) from the first step to be sufficient. Other-sort abduction algorithms now have less work, because only a fraction of alien subterm variables α_s remain in the partial solutions (see main algorithm in section B.6). They also have more information to work with, present in the instatiation of partial solutions. However, this optimization violates completeness guarantees of the combination of sorts algorithm. To faciliate finding term abduction solutions that hold under the quantifiers, we substitute-out other sort variables, by variables more to the left in the quantifier, using equations from the premise.

The dissociation interacts with the discard list mechanism. Since dissociation introduces fresh variables, no answers with alien subterms would be syntactically identical. When checking whether a partial answer should be discarded, in case alien subterm dissociation is *on*, we ignore alien sort subterms in the comparison.

B.2.2. Joint Constraint Abduction

Here we expand on the overview from Section 4.2.3. We further lose generality by using a heuristic search scheme instead of testing all combinations of simple abduction answers. In particular, our search scheme returns from joint abduction for types with a single answer, which eliminates deeper interaction between the sort of types and other sorts. Some amount of interaction is provided by the validation procedure, which checks for consistency of the partial answer, the premise and the conclusion of each branch, including consistency for other sorts.

We accumulate simple abduction answers into the partial abduction answer, we set aside branches that do not have any answer satisfiable with the partial answer so far. After all branches have been tried and the partial answer is not an empty conjunction (i.e. not $\top$), we retry the set-aside branches. If during the retry, any of the set-aside branches fails, we add the partial answer to discarded answers – which are avoided during simple abduction – and restart. Restart puts the set-aside branches to be tried first. If, when left with set-aside branches only, the partial answer is an empty conjunction, i.e. all the answer-contributing branches have been set aside, we fail – return $\bot$ from the joint abduction. This does not peform complete backtracking (no completeness guarantee), but is therefore quicker to report unsolvable cases and does sufficient backtracking. After an answer working for all branches has been found, we perform additional check, which encapsulates negative constraints introduced by the `assert false` construct. If the check fails, we add the answer to discarded answers and repeat the search.

If a partial answer becomes as strong as one of the discarded answers inside SCA, simple constraint abduction skips to find a different answer. The discarded answers are initialized with a discard list passed from the main algorithm.

To check validity of answers, we use a modified variant of unification under quantifiers: unification with parameters, where the parameters do not interact with the quantifiers and thus can be freely used and eliminated. Note that to compute conjunction of the candidate answer with a premise, unification does not check for validity under quantifiers.

Because it would be difficult to track other sort constraints while updating the partial answer, we discard numeric sort constraints in simple abduction algorithm, and recover them after the final answer for terms (i.e. for the type sort) is found.

Searching for an abduction answer can fail in only one way: we have set aside all the branches that could contribute to the answer. It is difficult to pin-point the culprit. We remember which branch caused the restart when the number of set-aside branches was the smallest. The conclusion of that branch can be used to construct the error report.

B.2.3. Simple Constraint Abduction for Terms

Here we expand on the overview from Section 4.2.4.1. Our initial implementation of simple constraint abduction for terms follows [29] p. 13. The mentioned algorithm only gives *fully maximal answers* which is loss of generality w.r.t. our requirements. To solve $D \Rightarrow C$ the algorithm starts with $\boldsymbol{U}(D \wedge C)$ and iteratively replaces subterms by fresh variables $\alpha \in \bar{\alpha}$ for a final solution $\exists \bar{\alpha}.A$. As our primary approach to mitigate some of the limitations of fully maximal answers, we start from $\boldsymbol{U}(\tilde{A}(D \wedge C))$, where $\exists \bar{\alpha}.A$ is the solution to previous problems solved by the joint abduction algorithm, and $\tilde{A}(\cdot)$ is the corresponding substitution. Moreover, motivated by examples from Chuan-kai Lin [22], we intruduce variable-variable equations $\beta_1 \doteq \beta_2$, for $\beta_1\beta_2 \subset \bar{\beta}$, not implied by $\tilde{A}(D \wedge C)$, as additional candidate answer atoms. During abduction $\mathrm{Abd}(\mathcal{Q}, \bar{\beta}, \overline{D_i, C_i})$, we ensure that the (partial as well as final) answer $\exists \bar{\alpha}.A$ satisfies $\vDash \mathcal{Q}.A[\bar{\alpha}\bar{\beta} := \bar{t}]$ for some $\bar{t}$. We achieve this by normalizing the answer using parameterized unification under quantifiers $\boldsymbol{U}_{\bar{\alpha}\bar{\beta}}(\mathcal{Q}.A)$. $\bar{\beta}$ are the parameters of the invariants.

In fact, when performing unification, we check more than $\boldsymbol{U}_{\bar{\alpha}\bar{\beta}}(\mathcal{Q}.A)$ requires. We also ensure that the use of parameters will not cause problems in the Split phase of the main algorithm. To this effect, we forbid substitution of a variable β_1 from $\bar{\beta}$ with a term containing a universally quantified variable that is not in $\bar{\beta}$ and to the right of β_1 in $\mathcal{Q}$. Also, we forbid substitution of a variable β_1 from $\bar{\beta}^{\chi}$ with a term containing a variable $\beta_2 \in \bar{\beta}^{\chi'}$ for $\chi \neq \chi'$.

In implementing [29] p. 13, we follow a top-down approach where bigger subterms are abstracted first – replaced by a fresh variable, together with an arbitrary selection of other occurrences of the subterm. If dropping the candidate atom maintains $T(F) \vDash A \wedge D \Rightarrow C$, we proceed to neighboring subterm or next equation. Otherwise, we try all of: replacing the subterm by the fresh variable; proceeding to subterms of the subterm; preserving the subterm; replacing the subterm by variables corresponding to earlier occurrences of the subterm. This results in a single, branching pass over all subterms considered. Finally, we clean-up the solution by eliminating fresh variables when possible (i.e. substituting-out equations $x \doteq \alpha$ for variable x and fresh variable α).

Although there could be an infinite number of abduction answers, there is always a finite number of *fully maximal* answers, or more generally, a finite number of equivalence classes of formulas strictly stronger than a given conjunction of equations in the domain $T(F)$. We use a search scheme that tests as soon as possible. The simple abduction algorithm takes a partial solution – a conjunction of candidate solutions for some other branches – and checks if the solution being generated is satisfiable together with the candidate partial solution. The algorithm also takes several indicators to let it select the expected answer:

- a number that determines how many correct solutions to skip;
- a validation procedure that checks whether the partial answer meets a condition, in joint abduction the condition is consistency with premise and conclusion of each branch;
- the parameters and candidates for parameters of the invariants, $\bar{\beta}$, updated as we add new atoms to the partial answer; existential variables that are not to the left of parameters and are connected to parameters become parameters; we process atoms containing parameters first;

- the quantifier $\mathcal{Q}$ (source `q`) so that the partial answer $\exists\bar{\alpha}.A$ (source `vs,ans`) can be checked for validity with parameters: $\vDash\mathcal{Q}.A[\bar{\alpha}:=\bar{t}]$ for some $\bar{t}$;
- a discard list of partial answers to avoid (a tabu list) – implements backtracking, with answers from abductions raising "fallback" going there.

Since an atom can be mistakenly discarded when some variable could be considered an invariant parameter but is not at the time, we process atoms incident with candidates for invariant parameters first. A variable becomes a candidate for a parameter if there is a parameter that depends on it. That is, we process atoms $x\doteq t$ such that $x\in\bar{\beta}$ first, and if equation $x\doteq t$ is added to the partial solution, we add to $\bar{\beta}$ existential variables in t. Note that $x\doteq t$ can stand for either $x:=t$, or $y:=x$ for $t=y$. For a universally quantified variable $x\notin\bar{\beta}$, $x:=t$ will not be part of the answer as it does not hold under the quantifier.

To simplify the search in presence of a quantifier prefix, we preprocess the initial candidate by eliminating universally quantified variables:

$$\begin{aligned} S &= [\bar{t_u}:=\bar{t'_u}] \text{ for } \mathrm{FV}(t_u)\cap\bar{\beta_u}\neq\varnothing, \overline{\forall\beta_u}\subset\mathcal{Q} \text{ such that } \mathcal{M}\vDash D\Rightarrow\dot{S},\\ S' &= [\bar{u}:=\bar{t'_u}] \text{ for } \bar{u}\subset\bar{\beta_u}, \overline{\forall\beta_u}\subset\mathcal{Q} \text{ such that } \mathcal{M}\vDash D\wedge C\Rightarrow\dot{S}',\\ \mathrm{Rev}_\forall(\mathcal{Q},\bar{\beta},D,C) &= \{c'|c=x\doteq t\in C, \text{if } x=S'(t) \text{ then } c'=S(c) \text{ else } c'=S\,S'(c)\}\end{aligned}$$

Note that S above is a substitution of subterms rather than of variables. To move further beyond fully maximal answers, we incorporate candidates $\beta_1\doteq\beta_2$ for which the following conditions hold: $\beta_1\beta_2\subset\bar{\beta}$, $\beta_1:=t_1\in\boldsymbol{U}(\tilde{A}(D\wedge C))$, $\beta_2:=t_2\in\boldsymbol{U}(\tilde{A}(D\wedge C))$ and $t_1\doteq t_2$ is satisfiable. We also need to include the unifier of $t_1\doteq t_2$ among the candidates, since otherwise the equation $\beta_1\doteq\beta_2$ would not suffice to imply that of the atoms $\beta_1\doteq t_1$, $\beta_2\doteq t_2$ which belongs to the conclusion C. The *full candidates* $\boldsymbol{U}(\tilde{A}(D\wedge C))$ and the *guess candidates* $\overline{\beta_1:=\beta_2;\boldsymbol{U}(t_1\doteq t_2)}$ are kept apart, the guess candidates are guessed before the full candidates. By default, we additionally limit consideration to atoms $\beta_1\doteq t_1$, $\beta_2\doteq t_2$ where t_1, t_2 are not themselves variables.

To recapitulate, the implementation is:

- If there are no more candidates to add to the partial solution: check for repeated answers, skipping, and discarded answers.
- If there are no more guessed candidates, pick the next full candidate atom $\mathrm{FV}(c)\cap\bar{\beta}\neq\varnothing$ if any, reordering the candidates until one is found. Otherwise, pick the first guess candidate atom without reordering.
- The are 6 mutually exclusive choices through which the algorithm loops.
    1. Try to drop the atom (if the partial answer plus remaining candidates can still be completed to a correct answer).
    2. Replace the current subterm of the atom with a fresh parameter, adding the subterm to replacements; if at the root of the atom, check connected and validate before proceeding to remaining candidates.
    3. Step into subterms of the current subterm, if any, and if at the sort of types.
    4. Keep the current part of the atom unchanged; if at the root of the atom, check connected and validate before proceeding to remaining candidates.

5. Replace the current subterm with a parameter introduced for an earlier occurrence; branch over all matching parameters; if at the root of the atom, check connected and validate before proceeding to remaining candidates.
6. Keep a variant of the original atom, but with constants substituted-out by variable-constant equations from the premise. Redundant, not available when the option `-more_general` is on.

- Each iteration is a backtracking point, the choices are tried in turn, depending on options selected.
- Default ordering of choices is 1, 6, 2, 4, 3, 5 – pushing 4 up minimizes the amount of branching in 5.
  - There is an option `-more_general`, which reorders the choices to: 1, 6, 4, 2, 3, 5; however the option is not exposed in the interface because the cost of this reordering is prohibitive.
  - An option `-richer_answers` reorders the choices to: 6, 1, 2, 4, 3, 5; it does not increase computational cost but sometimes leads to answers that are not most general.
  - If choice 6 would lead to more negative constraints contradicted than choice 1, we pick choice 6 first for a particular candidate atom.
  - An option `-prefer_guess` reorders choice 6 prior to choice 1, but only for guess candidates.
- Form initial candidates $\mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, \boldsymbol{U}(D \wedge A_p), \boldsymbol{U}(A_p \wedge D \wedge C))$.
- Form the substitution of subterms for choice-6 counterparts of initial candidate atoms. For $\alpha_1 \doteq \tau$, ..., $\alpha_n \doteq \tau \in \boldsymbol{U}(D \wedge A_p)$, form the substitution of subterms $\alpha_1 := \alpha_i$, ..., $\alpha_n := \alpha_i$, $\tau := \alpha_i$ (excluding $\alpha_i := \alpha_i$) where α_i is the most upstream existential variable (or parameter) and τ is a constant. Note that analogous transformation for a universally quantified variable as right-hand-sides is performed by $\mathrm{Rev}_\forall$.
  - Since for efficiency reasons we do not always remove alien subterms, we need to mitigate the problem of alien subterm variables causing violation of the quantifier prefix. To this effect, we include the premise equations from other sorts in the process generating the initial candidates and choice 6 candidates, but not as candidates. Not to lose generality of answer, we only keep a renaming substitution, in particular we try to eliminate universal variables.
- Sort the initial candidates by decreasing size, because shorter answer atoms are more valuable and dropping a candidate from participating in an answer is the first choice.
  - There is an argument in favor of sorting by increasing size: so that the replacements of step 2 are formed at a root position before they are used in step 5 – instead of forming a replacement at a subterm, and using it in step 5 at a root.
  - If ordering in increasing size turns out to be necessary, a workaround should be introduced to favor answers that, if possible, do not have parameters $\bar{\beta}$ as left-hand-sides.

The above ordering of choices ensures that more general answers are tried first. Moreover:

- choice 1 could be dropped as it is equivalent to choice 2 applied on the root term;
- choices 4 and 5 could be reordered but having choice 4 as soon as possible is important for efficiency.

We perform a two-layer iterative deepening (when without the `-more_general` option): in the first run we only try choices 1 and 6. It is an imperfect optimization since the running time gets longer whenever choices 2-5 are needed.

B.2.3.1. Heuristic for Better Answers to Invariants

We implement an optional heuristic in forming the candidates proposed by choice 6. It may lead to better invariants when multiple maximally general types are possible, but also it may lead to getting the most general type without the need for backtracking across iterations of the main algorithm, which unfortunately often takes very long.

We look at the types of substitutions for the variables that are invariant parameters, in the partial answer, and try to form the initial candidates for choice 6 so that the return type variables cover the most of argument types variables, for each term substituted for an invariant parameter. We select from the candidates equations between any variable, or only non-argument-type variable, and a FV(argument types)\FV(return type) variable – we turn the equation so that the latter is the RHS. We locate the equations among the candidates that have an invariant parameter variable or a FV(return type)\FV(argument types) variable as LHS. We apply the substitution to the RHS of these equations; if the LHS is an invariant parameter, we use the substitution based on equations where one side is a non-argument-type variable. We preserve the order of equations in the candidate list.

B.2.4. Simple Constraint Abduction for Linear Arithmetics

Here we expand on the overview from Section 4.2.4.2. For checking validity or satisfiability, we use *Fourier-Motzkin elimination*. To avoid complexities we only handle the rational number domain. To extend the algorithm to integers, *Omega-test* procedure as presented in [4] needs to be adapted. The major operations are:

- *Elimination* of a variable takes an equation and selects a variable that is not upstream, i.e. to the left in $\mathcal{Q}$ regarding alternations, of any other variable of the equation, and substitutes-out this variable from the rest of the constraint. The solved form contains an equation for this variable.
- *Projection* of a variable takes a variable x that is not upstream of any other variable in the unsolved part of the constraint, and reduces all inequalities containing x to the form $x \dot{\leqslant} a$ or $b \dot{\leqslant} x$, depending on whether the coefficient of x is positive or negative. For each such pair of inequalities: if $b = a$, we add $x \doteq a$ to implicit equalities; otherwise, we add the inequality $b \dot{\leqslant} a$ to the unsolved part of the constraint.

We use elimination to solve all equations before we proceed to inequalities. The starting point of our algorithm is [4] section *4.2 Online Fourier-Motzkin Elimination for Reals*. We add detection of implicit equalities, and more online treatment of equations, introducing known inequalities on eliminated variables to the projection process. When implicit equalities have been found, we iterate the process to normalize them as well.

Our abduction algorithm follows a familiar incrementally-generate-and-test scheme as in term abduction. There are two new ideas, which can also be applied to abduction in other domains. We build a lazy list of possible transformations with linear combinations involving equations a implied by $D \wedge C$. We pair each inequality c in C with all inequalities d implied by D which share a variable with c and try out the abduction answers to $d \Rightarrow c$ as contributions to the partial abduction answer to $D \Rightarrow C$.

To simplify the search in presence of a quantifier prefix, we preprocess the initial candidate by eliminating universally quantified variables:

$$\begin{aligned} S &= [\bar{\beta_u} := \bar{t_u}] \text{ for } \overline{\forall \beta_u} \subset \mathcal{Q} \text{ such that } \mathcal{M} \vDash D \Rightarrow \dot{S}, \\ \mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D, C) &= \{c' | c \in C, \text{if } \mathcal{M} \vDash \mathcal{Q}.c[\bar{\beta} := \bar{t}] \text{ for some } \bar{t} \text{ then } c' = c \text{ else } c' = S(c)\} \end{aligned}$$

Before accepting a new atom into the partial answer, we check that it would not violate the quantifier conditions from the *split* phase of the main algorithm, and that the partial answer is satisfiable with all implication branches of the joint abduction problem. As part of the quantifier conditions, we ensure that the escaping parameters are upward in the constraint before prenexization, i.e. are parameters of the type of a parent definition (containing a given definition in its body) rather than a parallel definition. The check of satisfiability we call *validation*. For domains other than the term domain, validation also involves instantiating use-sites of recursive definitions with parts of the partial answer, split in a simplified way.

Abduction algorithm:

1. Let $C^{=\prime} = \tilde{A}_i(C^=)$, resp. $C^{\leqslant\prime} = \tilde{A}_i(C^\leqslant)$ where $C^=$, resp. $C^\leqslant$ are the equations, resp. inequalities in C and $\tilde{A}_i$ is the substitution according to equations in A_i. Let $D' = \tilde{A}_i(D \wedge A_i)$ and $D^{=\prime}$ be the equations in D', i.e. substituted equations and implicit equalities. Let $D^{\leqslant\prime}$ be a solved form of inequalities.
    a. Let $C_0^= = \mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D^{=\prime}, C^{=\prime})$ and $C_0^\leqslant = \mathrm{Rev}_\forall(\mathcal{Q}, \bar{\beta}, D^{=\prime}, C^{\leqslant\prime})$.
2. Prepare the initial transformations from atoms $a \in D^{=\prime}$:
    a. Add combinations $k^s\, a + b$ for $k = -n...n, s = -1, 1$ to the stack of transformations to be tried for atoms b.
    b. The final transformations have the form: $b \mapsto b + \Sigma_{a \in D} k_a^{s_a} a$.
3. Modify $C^{=\prime}$ to promote answers with variables rather than constants, as in term abduction: For $\alpha_1 \doteq \tau, ..., \alpha_n \doteq \tau \in C^{=\prime}$, form the substitution of subterms $\alpha_1 := \alpha_i, ..., \alpha_n := \alpha_i, \tau := \alpha_i$ (excluding $\alpha_i := \alpha_i$) where α_i is the most upstream existential variable (or parameter) and τ is a constant.
4. Start from $\mathrm{Acc} := \{\}$ and $C_0 := C^{=\prime} \wedge C^{\leqslant\prime}$. Handle atoms a in $C_0 = a C_0'$, equations first.
5. Let $B = A_i \wedge D \wedge C_0' \wedge \mathrm{Acc}$.
6. If a is a tautology ($0 \doteq 0$ or $c \leq 0$ for $c \leqslant 0$) or $B \Rightarrow C$, repeat with $C_0 := C_0'$. Corresponds to choice 1 of term abduction.
7. If $B \not\Rightarrow C$, for a candidate a' generated for a, starting with a, which passes validation against other branches in a joint problem: $\mathrm{Acc} := \mathrm{Acc} \cup \{a'\}$, or fail if all a' fail.
    a. If a is an inequality, let a_0 be an abduction answer to $d \Rightarrow \mathrm{Rev}_\forall\big(\mathcal{Q}, \bar{\beta}, D^{=\prime}, \widetilde{\mathrm{Acc}^=}(a)\big)$, where d belongs to the solved form inequalities implied by D'. Otherwise, let $a_0 = \widetilde{\mathrm{Acc}^=}(a)$.

b. Sort the candidates a_0 in order of decreasing value, described below. Let a' be a_0 with some $D^{=\prime}$-derived transformation applied.

- We increase the value of a_0 for each variable that is bound by a_0 on the side that it is unbound in B. We decrease the value of a_0 for:
  - its size, proportionally to the sum of nominators and denominators,
  - containing a constant term,
  - containing parameters from multiple invariants,
  - introducing an implicit equality,
  - binding a variable by a constant on the side where it is already bounded by B,
  - binding a variable by a constant on the right (i.e. upper bound),
  - optionally, being less general than some other candidate, modulo B,
  - we include the value of inequalities implied by the candidate together with the partial answer (but not the partial answer alone) and not holding when parameters are universally quantified (see *atomization*).

  The score determines the order in which atoms are tried.
- Currently, we do not perform transformations when a_0 is an inequality, for simplicity and speed at cost of missing some answers. However, we eliminate the universal variables, i.e. we use $\mathrm{Rev}_\forall$ above. If it proves necessary, we will also try the transformations for the inequalities. For example, by trying all candidates a_0 before proceeding to the next transformation.

c. If $A_i \wedge (\mathrm{Acc} \cup \{a'\})$ (resp. $A_i \wedge (\mathrm{Acc} \cup \{a''\})$) does not pass validation for all a', backtrack.

d. If $A_i \wedge (\mathrm{Acc} \cup \{a'\})$ (resp. $A_i \wedge (\mathrm{Acc} \cup \{a''\})$) passes validation, repeat from step 5 with $C_0 := C_0', \mathrm{Acc} := \mathrm{Acc} \cup \{a'\}$ (resp. $\mathrm{Acc} := \mathrm{Acc} \cup \{a''\}$).

8. The answers are $A_{i+1} = A_i \wedge \mathrm{Acc}$.

We precompute the tranformation variants to try out. The parameter n is set by option `-num_abduction_rotations` and defaults to a small value (currently 3).

To check whether $B \Rightarrow C$, we check for each $c \in C$:

- if $c = x \doteq y$, that $A(x) = A(y)$, where $A(\cdot)$ is the substitution corresponding to equations and implicit equalities in A;
- if $c = x \dot{\leqslant} y$, that $B \wedge y \dot{<} x$ is not satisfiable.

We use the `nums` library for exact precision rationals.

To find the abduction answers to $d \Rightarrow c$, pick a common variable $\alpha \in \mathrm{FV}(d) \cap \mathrm{FV}(c)$ or the constant $\alpha = 1$. We have four possibilities:

1. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answers are c and $d_\alpha \leqslant c_\alpha$,
2. $d \Leftrightarrow \alpha \leqslant d_\alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answer is only c,
3. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow \alpha \leqslant c_\alpha$: the abduction answer is only c,
4. $d \Leftrightarrow d_\alpha \leqslant \alpha$ and $c \Leftrightarrow c_\alpha \leqslant \alpha$: the abduction answers are c and $c_\alpha \leqslant d_\alpha$.

Thanks to cases (1) and (4) above, the abduction algorithm can find some answers which are not fully maximal. The joint constraint abduction algorithm can help in some of the remaining cases where fully maximal abduction is insufficient for some implications, by solving simpler implications first.

We provide an optional optimization: we do not pass, in the first call to numerical abduction (the second iteration of the main algorithm), branches that contain unary predicate variables in the conclusion, i.e. we only use the "non-recursive" branches. Other optimizations that we use are: iterative deepening on the constant n used to generate k^s factors. We also constrain the algorithm by filtering out transformations that contain "too many" variables, which can lead to missing answers if the setting `-num_prune_at` – "too many" – is too low. Similarly to term abduction, we count the number of steps of the loop and fail if more than the option `-num_abduction_timeout` steps have been taken.

B.3. Constraint Generalization

Here we expand on the exposition from Section 4.3. *Constraint generalization* answers are the maximally specific conjunctions of atoms that are implied by each of a given set of conjunction of atoms. In case of term equations the constraint generalization algorithm is based on the *anti-unification* algorithm. In case of linear arithmetic inequalities, constraint generalization is exactly finding the convex hull of a set of possibly unbounded polyhedra. We employ our unification algorithm to separate sorts. Since as a result we do not introduce variables for *alien subterms*, we include the variables introduced by anti-unification in constraints sent to constraint generalization for their respective sorts.

The adjusted algorithm looks as follows:

1. Let $\wedge_s D_{i,s} \equiv \boldsymbol{U}(D_i)$ where $D_{i,s}$ is of sort s, be the result of our sort-separating unification.
2. For the sort s_{type}:
   a. Let $V = \{x_j, \overline{t_{i,j}} | \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{type}}}\}$.
   b. Let $G = \{\bar{\alpha}_j, u_j, \overline{\theta_{i,j}} | \theta_{i,j} = [\bar{\alpha}_j := \bar{g}^i_j], \theta_{i,j}(u_j) = t_{i,j}\}$ be the most specific anti-unifiers of $\overline{t_{i,j}}$ for each j.
   c. Let $D^u_i = \wedge_j \bar{\alpha}_j \doteq \bar{g}^i_j$ and $D^g_i = D_{i,s_{\text{type}}} \wedge D^u_i$.
   d. Let $D^v_i = \{x \doteq y | x \doteq t_1 \in D^g_i, y \doteq t_2 \in D^g_i, D^g_i \vDash t_1 \doteq t_2\}$.
   e. Let $A_{s_{\text{type}}} = \wedge_j x_j \doteq u_j \wedge \bigcap_i (D^g_i \wedge D^v_i)$ (where conjunctions are treated as sets of conjuncts and equations are ordered so that only one of $a \doteq b, b \doteq a$ appears anywhere), and $\bar{\alpha}_{s_{\text{type}}} = \overline{\bar{\alpha}_j}$.

f. Let $\wedge_s D^u_{i,s} \equiv D^u_i$ for $D^u_{i,s}$ of sort s.

3. For sorts $s \neq s_{\text{type}}$, let $\exists \bar{\alpha}_s . A_s = \text{LUB}_s(\overline{D^s_i \wedge D^u_{i,s}})$.

4. The answer is $\exists \overset{=}{\alpha^j_i} \bar{\alpha}_s . \wedge_s A_s$.

We simplify the result by substituting-out redundant answer variables.

We follow the anti-unification algorithm provided in [61], fig. 2.

B.3.1. Extended Convex Hull

[16] provides a polynomial-time algorithm to find the half-space represented convex hull of closed polytopes. It can be generalized to unbounded polytopes – conjunctions of linear inequalities. Our implementation is inspired by this algorithm but very much simpler, at cost of losing the optimality requirement.

First we find among the given inequalities those which are also the faces of resulting convex hull. The negation of such inequality is not satisfiable in conjunction with any of the polytopes – any of the given sets of inequalities. Next we iterate over *ridges* touching the selected faces: pairs of the selected face and another face from the same polytope. We rotate one face towards the other: we compute a convex combination of the two faces of a ridge. We add to the result those half-spaces whose complements lie outside of the convex hull (i.e. negation of the inequality is unsatisfiable in conjunction with every polytope). For a given ridge, we add at most one face, the one which is farthest away from the already selected face, i.e. the coefficient of the selected face in the convex combination is smallest. We check a small number of rotations, where the algorithm from [16] would solve a linear programming problem to find the rotation which exactly touches another one of the polytopes.

When all variables of an equation $a \doteq b$ appear in all branches D_i, we can turn the equation $a \doteq b$ into pair of inequalities $a \leqslant b \wedge b \leqslant a$. We eliminate all equations and implicit equalities which contain a variable not shared by all D_i, by substituting out such variables. We pass the resulting inequalities to the convex hull algorithm. Separately, we compute the equations common to all branches, because the convex hull algorithm is not guaranteed to recover them.

B.3.2. Issues in Inferring Postconditions

Although finding recursive function invariants – predicate variables solved by abduction – could theoretically fail to converge for both the type sort and the numerical sort constraints, neither problem was observed. Finding existential type constraints can only fail to converge for numerical sort, because solutions are expected to decrease in strength. But such diverging numerical constraints are commonplace. The main algorithm starts by performing constraint generalization only on implication branches corresponding to non-recursive cases, i.e. without binary predicate variables in premise (or unary predicate variables in conclusion). This generates a stronger constraint than the correct one. Subsequent iterations include all branches in constraint generalization, weakening the constraints, and so still weaker constraints are fed to constraint generalization in each following step. To ensure convergence of the numerical part, starting from some step of the main loop, we compare consecutive solutions and extrapolate the trend. Currently we simply intersect the sets of atoms, but first we expand equations into pairs of inequalities.

We "lift" variables escaping the scope of a postcondition by renaming them to fresh variables, which are added to the answer variables of generalization. We apply this renaming ater anti-unification, prior to performing generalization in other domains. We pass to other domains as parameters to preserve only non-escaping variables. The non-escaping variables are these which have in their scope a variable α_K, passed as argument to the generalization algorithm.

Constraint generalization limited to non-recursive branches, the initial iteration of post-condition inference, will often generate constraints that contradict other branches. For another iteration to go through, the partial solutions need to be consistent. Therefore we filter the constraints using the same validation mechanism as in abduction. We add atoms to a constraint greedily, but to favor relevant atoms, we do the filtering while computing the connected component of constraint generalization result. See the details of the main algorithm in section B.6.2.

While reading section B.6.2, you will notice that postconditions are not subjected to stratification. This is because the type system does not support nested existential types.

In the simplification step at the end of constraint generalization, we try to preserve alien variables that are parameters rather than substituting them by constants. A parameter can only equal a constant if not all branches have been considered for constraint generalization. The parameter is both as informative as the constant, and less likely to contradict other branches.

B.3.3. Abductive Constraint Generalization

Here we expand on the overview from Section 4.3.3.

Global variables here are the variables shared by all disjuncts, i.e. $\cap_i \mathrm{FV}(D_i)$, remaining variables are *non-global*. Recall that for numerical constraint generalization, we either substitute-out a non-global variable in a branch if it appears in an equation, or we drop the inequalities it appers in if it is not part of any equation. Non-global variables can also pose problems for the term sort, by forcing constraint generalization answers to be too general. When inferring the type for a function, which has a branch that does not use one of arguments of the function, the existential type inferred would hide the corresponding information in the result, even if the remaining branches assume the argument has a single concrete type. We would like the corresponding non-global variable to resolve to the concrete type suggested by other branches of the resulting constraint.

We extend the notion of constraint generalization: substitution U and solved form $\exists\bar{\alpha}.A$ is an answer to *abductive constraint generalization* problem $\overline{D_i}$ given a quantifier prefix $\mathcal{Q}$ when:

1. $(\forall i) \vDash U(D_i) \Rightarrow \exists\bar{\alpha}\backslash\mathrm{FV}(U).A$;
2. If $\alpha \in \mathrm{Dom}(U)$, then $(\exists\alpha) \in \mathcal{Q}$ – variables substituted by U are existentially quantified;
3. $(\forall i) \vDash \forall(\mathrm{Dom}(U))\exists(\mathrm{FV}(D_i)\backslash\mathrm{Dom}(U)).D_i$.

Our generalized anti-unification algorithm is in Table B.1. The notational shorthand $\ldots;\beta_i;\ldots;f(\bar{t}^j);\ldots$ represents the case where all terms are either existential variables or start with a function symbol f. Similarly, $\ldots;\beta_i;\ldots;\beta_j;\ldots$ represents the case when there is a variable β_j such that all terms are either β_j or are existential variables to the right of β_j in the quantifier.

$$
\begin{array}{rcl}
\mathrm{au}_{U,G}(t;\ldots;t) & = & \varnothing, t, U, G \\
\mathrm{au}_{U,G}(f(\bar{t}^1);\ldots;f(\bar{t}^n)) & = & \bar{\alpha}, f(\bar{g}), U', G' \\
\text{where } \bar{\alpha}, \bar{g}, U', G' & = & \mathrm{aun}_{U,G}(\bar{t}^1;\ldots;\bar{t}^n) \\
\mathrm{au}_{U,G}(t_1;\ldots;t_n) & = & \varnothing, \alpha, U, G \\
\text{when} & & ([t_1;\ldots;t_n] \mapsto \alpha) \in G \\
\mathrm{au}_{U,G}(\ldots;\beta_i;\ldots;f(\bar{t}^j);\ldots \text{ as } \bar{t}) & = & \bar{\alpha}\bar{\alpha}', g, U'', G' \\
\text{where } \bar{\alpha}', g, U'', G' & = & \mathrm{au}_{U',G}(\bar{t}[\beta_i := f(\bar{\alpha})]) \\
U' & = & U[\beta_i := f(\bar{\alpha})] \wedge \beta_i \doteq f(\bar{\alpha}) \\
\text{when} & & (\exists \beta_i) \in \mathcal{Q} \vee \beta_i \in \bar{\beta}, \text{ treat } \bar{\alpha} \text{ as quantified with } \exists \beta_i \\
\mathrm{au}_{U,G}(\ldots;\beta_i;\ldots;\beta_j;\ldots \text{ as } \bar{t}) & = & \bar{\alpha}', g, U'', G' \\
\text{where } \bar{\alpha}', g, U'', G' & = & \mathrm{au}_{U',G}(\bar{t}[\beta_i := \beta_j]) \\
U' & = & U[\beta_i := \beta_j] \wedge \beta_i \doteq \beta_j \\
\text{when} & & \exists \beta_i \in \mathcal{Q}, \beta_j \leqslant_{\mathcal{Q}} \beta_i \\
\mathrm{au}_{U,G}(t_1;\ldots;t_n) & = & \alpha, \alpha, U, ([t_1;\ldots;t_n] \mapsto \alpha)G \\
\text{otherwise,} & & \text{where } \alpha \# \mathrm{FV}(t_1;\ldots;t_n, U, G) \\
\mathrm{aun}_{U,G}(\varnothing) & = & \varnothing, \varnothing, U, G \\
\mathrm{aun}_{U,G}(\varnothing;\ldots) & = & \varnothing, \varnothing, U, G \\
\mathrm{aun}_{U,G}(t_1^1, \bar{t}^1;\ldots;t_1^n, \bar{t}^n) & = & \bar{\alpha}\bar{\alpha}', g\bar{g}', U'', G'' \\
\text{where } \bar{\alpha}, g, U', G' & = & \mathrm{au}_{U,G}(t_1^1;\ldots;t_1^n) \\
\text{and } \bar{\alpha}', \bar{g}', U'', G'' & = & \mathrm{aun}_{U',G'}(\bar{t}^1;\ldots;\bar{t}^n)
\end{array}
$$

Table B.1. Abductive anti-unification algorithm

The sort-integrating algorithm essentially does not change:

1. Let $\wedge_s D_{i,s} \equiv \boldsymbol{U}(D_i)$ where $D_{i,s}$ is of sort s, be the result of our sort-separating unification.

2. For the sort s_{type}:

   a. Let $V = \{x_j, \overline{t_{i,j}} | \forall i \exists t_{i,j}. x_j \doteq t_{i,j} \in D_{i,s_{\text{type}}}\}$.

   b. Let $G = \{\bar{\alpha}_j, g_j, u_j, \overline{\theta_{i,j}} | \theta_{i,j} = [\bar{\alpha}_j := \bar{g}_j^i], \theta_{i,j}(g_j) = (\wedge_{k \leqslant j} u_k)(t_{i,j})\}$ be the most specific anti-unifiers of $\overline{(\wedge_{k \leqslant j} u_k)(t_{i,j})}$ for each j, where $\wedge_j u_j$ is the resulting U.

   c. Let $D_i^u = \wedge_j \bar{\alpha}_j \doteq \bar{g}_j^i$ and $D_i^g = D_{i,s_{\text{type}}} \wedge D_i^u$.

   d. Let $D_i^v = \{x \doteq y | x \doteq t_1 \in D_i^g, y \doteq t_2 \in D_i^g, D_i^g \vDash t_1 \doteq t_2\}$.

   e. Let $A_{s_{\text{type}}} = \wedge_j x_j \doteq g_j \wedge \bigcap_i (D_i^g \wedge D_i^v)$ (where conjunctions are treated as sets of conjuncts and equations are ordered so that only one of $a \doteq b, b \doteq a$ appears anywhere), and $\bar{\alpha}_{s_{\text{type}}} = \bar{\bar{\alpha}}_j$.

   f. Let $\wedge_s D_{i,s}^u \equiv D_i^u$ for $D_{i,s}^u$ of sort s.

3. For sorts $s \neq s_{\text{type}}$, let $\exists \bar{\alpha}_s. A_s = \mathrm{LUB}_s(\overline{D_i^s \wedge D_{i,s}^u})$.

4. The answer is substitution $U = \wedge_j u_j$ and solved form $\exists \overset{=}{\alpha}_i^j \bar{\alpha}_s. \wedge_s A_s$.

The task of constraint generalization is to find postconditions. Usually, it is beneficial to make the postcondition, i.e. the existential type that is the return type of a function, more specific, at the expense of making the overal type of the function less general. To this effect, constraint generalization, just as abduction takes invariant parameters $\bar{\beta}$. We replace conditions $\exists\beta_i \in \mathcal{Q}$ above by $\beta_i \in \bar{\beta} \vee (\exists\beta_i) \in \mathcal{Q}$. Recall that the right-hand-side (RHS) variable β_j can in general be universally quantified: $\forall\beta_j \in \mathcal{Q}$. We exclude universal non-parameter RHS when a parameter is present: if for any β_i, $\beta_i \in \bar{\beta}$, then for all β_i including RHS, $\beta_i \in \bar{\beta} \vee \exists\beta_i \in \mathcal{Q}$. Note that having weaker postconditions also results in correct types, just not the intended ones. In rare cases a weaker postcondition but a more general invariant can be beneficial. To this effect, the option `-more_existential` turns off generating the substitution entries when the RHS is a variable, i.e. the case $\mathrm{au}_{U,G}(\ldots;\beta_i;\ldots;\beta_j;\ldots \text{ as } \bar{t})$ is skipped.

Due to greater flexibility of the numerical domain, abductive extension of numerical constraint generalization does not seem necessary and is turned off by default. It could take a similar form, we experiment with the following heuristic. If atoms specific to a disjunct (i.e. not shared by all disjuncts) do not contain a variable, do not include the disjunct when considering inclusion of inequalities containing the variable in the constraint generalization answer.

B.4. Incorporating Negative Constraints

Here we expand on the overview from Section 4.4. We call a *negative constraint* an implication $D \Rightarrow \boldsymbol{F}$ in the normalized constraint $\mathcal{Q}.\ \wedge_i (D_i \Rightarrow C_i)$, and we call D the *negated constraint*. Such constraints are generated for pattern matching branches whose right-hand-side is the expression `assert false`. A generic approach to account for negative constraints is as follows. We check whether the solution found by multisort abduction contradicts the negated constraints. If some negated constraint is satisfiable, we discard the answer and "fall back" to try finding another answer. Unfortunately, this approach is insufficient when the answer sought for is not among the maximally general answers to the remaining, *positive part* of the constraint. Therefore, we introduce *negation elimination*.

For the numerical sort, our implementation of negation elimination can produce too strong constraints when the numerical domain is intended to contain non-integer rational numbers, and can be turned off by the `-no_int_negation` option. It can also produce too weak constraints, because it produces at most one atom for a given negated constraint. If the abduction answer for terms does already contradict a negated constraint D, we are done. Otherwise, let $D = c_1 \wedge \ldots \wedge c_n$. For numerical sort atoms c_i, either drop them from consideration or convert their negation $\neg c_i$ into d_i or $d_{i_1} \vee d_{i_2}$ as follows. The conversion assumes that the numerical domain is integers. We convert an inequality $w \leqslant 0$, e.g. $\frac{1}{3}x - \frac{1}{2}y - 2 \leqslant 0$, to $-k\,w + 1 \leqslant 0$, e.g. $3y - 2x + 13 \leqslant 0$, where k is the common denominator so that $k\,w$ has only integer numbers. Note that $\neg(w \leqslant 0) \Leftrightarrow w > 0 \Leftrightarrow -w < 0 \Leftrightarrow -k\,w < 0 \Leftarrow -k\,w + 1 \leqslant 0$. Similarly, we convert an equation $w \doteq 0$ to inequalities $-k\,w + 1 \leqslant 0 \vee k\,w + 1 \leqslant 0$. Note that $\neg(w \geqslant 0) \Leftrightarrow w < 0 \Leftrightarrow k\,w < 0 \Leftarrow k\,w + 1 \leqslant 0$. In both cases the implications $\Leftarrow$ would be equivalences if the numerical domain was integers rather than rational numbers. At present, we ignore *opti* atoms. The disjunct d_i is a conjunction of inequalities if c_i is a *subopti* atom.

Assuming that each negative constraint points to a single atomic fact, we try to find one disjunct d_i, resp. d_{i_1} or d_{i_2}, corresponding to $\neg c_i$, discarding those disjuncts that contradict any implication branch. Specifically, let $\mathcal{Q}.\wedge_i (D_i \Rightarrow C_i)$ be the constraint we solve, and let $\exists\bar{\alpha}.A$ be a term abduction answer for $\mathcal{Q}.\wedge_{i:C_i\neq\boldsymbol{F}} (D_i \Rightarrow C_i)$. We search for i such that for all k with $C_k \neq \boldsymbol{F}$ and D_k satisfiable, $d_i \wedge A \wedge D_k \wedge C_k$ is satisfiable. We provide a function $\text{NegElim}(\neg D, \bar{B_i}) = d_{i_0}$, where d_{i_0} is the biggest, syntactically, such atom, and $B_i = A \wedge D_i \wedge C_i$.

Unfortunately, for the sort of terms we do not have well-defined representation of disequalities not using negations. The generic approach of relying on backtracking to find another abduction answer is more useful for terms than for the numeric sort. It falls short, however, when negation was intended to prevent the answer from being too general. Ideally, we would introduce disequation atoms $\tau \dot{\neq} \tau$ and follow the scheme we use for the numerical sort. For now, we only cover a very specific use of negation, to discriminate among type-level "enumeration". We limit negation elimination to considering atoms of the form $\beta \doteq \varepsilon_1$, and contradict them by introducing atoms $\beta \doteq \varepsilon_2$, for types $\varepsilon_1 \neq \varepsilon_2$ without parameters which we call *phantom enumerations*. The variables β are limited to the answer variables generated in the previous iteration of the main algorithm. The nullary datatype constructor ε_2 is picked so that the atom is valid, using the same validation procedure as the one passed to the abduction algorithm. The heuristic defines phantom enumerations as nullary phantom types that do not share datype parameter position (in GADT constructor definitions) with non-enumeration types. When the equations derived for different negated constraints involve a common variable as the left-hand-side, we select a common right-hand-side. In the end, we only introduce the negation elimination result to the answer when a single disjunct remains.

Since the *discard* (or taboo) list used by backtracking is based on complete answers, it is preferable to perform negation elimination prior to abduction. Otherwise, the search might fall into a loop where abduction keeps returning the same answer, since it is more general than the discarded answer incorporating negation elimination result.

B.5. *opti* AND *subopti*: *minimum* AND *maximum* RELATIONS IN NUM

We extend the numerical domain with relations *opti* and *subopti* defined below. Operations *min* and *max* can then be defined using it. Let k, v, w be any linear combinations. Note that the relations are introduced to smuggle in a limited form of disjunction into the solved forms.

$$\begin{aligned}
\text{opti}(v, w) &= v \leqslant 0 \wedge w \leqslant 0 \wedge (v \doteq 0 \vee w \doteq 0) \\
k \doteq \min(v, w) &= \text{opti}(k - v, k - w) \\
k \doteq \max(v, w) &= \text{opti}(v - k, w - k) \\
\text{subopti}(v, w) &= v \leqslant 0 \vee w \leqslant 0 \\
k \leqslant \max(v, w) &= \text{subopti}(k - v, k - w) \\
\min(v, w) \leqslant k &= \text{subopti}(v - k, w - k)
\end{aligned}$$

In particular, $\text{opti}(v, w) \equiv \max(v, w) \doteq 0$ and $\text{subopti}(v, w) \equiv \min(v, w) \leqslant 0$. We call an *opti* or *subopti* atom *directed* when there is a variable n that appears in v and w with the same sign. We do not prohibit undirected *opti* or *subopti* atoms, but we do not introduce them, to avoid bloat.

For simplicity, we do not support *min* and *max* as subterms in concrete syntax. Instead, we parse atoms of the form `k=min(...,...)`, resp. `k=max(...,...)` into the corresponding *opti* atoms, where `k` is any numerical term. Similarly for *subopti*. We also print directed *opti* and *subopti* atoms using the syntax with min and max expressions. Not to pollute the syntax with a new keyword, we use concrete syntax `min|max(...,...)` for parsing arbitrary, and printing non-directed, *opti* atoms, and `min||max(...,...)` for *subopti* respectively.

If need arises, in a future version, we can extend *opti* to a larger arity N.

B.5.1. Normalization, Validity and Implication Checking

In the function that produces solved forms of numerical constraints, we treat *opti* clauses in an efficient but incomplete manner, doing a single step of constraint solving. We include the *opti* terms in processed inequalities. After equations have been solved, we apply the substitution to the *opti* and *subopti* disjunctions. When one of the *opti* resp. *subopti* disjunct terms becomes contradictory or the disjunct terms become equal, we include the other in implicit equalities, resp. in inequalities to solve. When one of the *opti* or *subopti* terms becomes tautological, we drop the disjunction. We iterate calls to the solver function to propagate implicit equalities.

We do not perform case splitting on *opti* and *subopti* disjunctions, therefore some contradictions may be undetected. However, abduction and constraint generalization currently perform upfront case splitting on *opti* and *subopti* disjunctions, sometimes leading to splits that a smarter solver would avoid.

B.5.2. Abduction

We eliminate *opti* and *subopti* in premises by expanding the definition and converting the branch into two branches, e.g. $D \wedge (v \doteq 0 \vee w \doteq 0) \Rightarrow C$ into $(D \wedge v \doteq 0 \Rightarrow C) \wedge (D \wedge w \doteq 0 \Rightarrow C)$. Recall that an *opti* atom also implies inequalities $v \leqslant \wedge w \leqslant 0$ assumed to be in D above. This is one form of *case splitting*: we consider cases $v \doteq 0$ and $w \doteq 0$, resp. $v \leqslant 0$ and $w \leqslant 0$, separately. We do not eliminate *opti* and *subopti* in conclusions. Rather, we consider whether to keep or drop it in the answer, like with other candidate atoms. The transformations apply to an *opti* atom by applying to both its arguments.

Generating a new *opti* atom for inclusion in an answer means finding a pair of equations such that the following conditions hold. Each equation, together with remaining atoms of an answer but without the remaining equation selected, is a correct answer to a simple abduction problem. The equations selected share a variable and are oriented so that the variable appears with the same sign in them. The resulting *opti* atom passes the validation test for joint constraint abduction. We may implement generating new *opti* atoms for abduction answers in a future version, when need arises. Currently, we only generate new *opti* and *subopti* atoms for postconditions, i.e. during constraint generalization.

B.5.3. Constraint Generalization

We eliminate *opti* and *subopti* atoms prior to finding the extended convex hull of $\overline{D_i}$ by expanding the definition and converting the disjunction $\vee_i D_i$ to disjunctive normal form. This is another form of case splitting.

In addition to finding the extended convex hull, we need to discover *opti* relations that are implied by $\vee_i D_i$. We select these faces of the convex hull which also appear as an equation in some disjuncts. Out of these faces, we find all minimal covers of size 2, i.e. pairs of faces such that in each disjunct, either one or the other linear combination appears as an equation. We only keep pairs of faces that share a same-sign variable. For the purposes of detecting *opti* relations, we need to perform transitive closure of the extended convex hull equations and inequalities, because the redundant inequalities might be required to find a cover.

Finding *subopti* atoms is similar. We find all minimal covers of size 2, i.e. pairs of inequalities such that one or the other appears in each disjunct. We only keep pairs of inequalities that share a same-sign variable.

We provide a function for the numerical domain to remove *opti* atoms of the form $k \doteq \min(c, v)$, $k \doteq \min(v, c)$, $k \doteq \max(c, v)$ or $k \doteq \min(v, c)$ for a constant c, similarly for *subopti* atoms, while in initial iterations where constraint generalization is only performed for non-recursive branches.

We need to further extend the notion of (abductive) constraint generalization, to achieve the results required for postcondition inference. For constraint branches $\overline{D_i \Rightarrow C_i}$, we need not only the disjuncts $\overline{D_i \wedge C_i}$, but also the premises $\overline{D_i}$. We keep *opti* and *subopti* atoms opti(v, w), subopti(v, w) such that either both $v \leqslant w$ and $w \leqslant v$ are *satisfiable with* all *implication branches*, or neither is. An atom c is satisfiable with implication $D_i \Rightarrow C_i$ here, when either $c \wedge D_i$ is not satisfiable, or $c \wedge D_i \wedge C_i$ is satisfiable. The underlying idea is that since *opti* and *subopti* atoms express a disjunction, resp. $v \leqslant 0 \wedge w \leqslant 0 \wedge (v \doteq 0 \vee w \doteq 0)$ and $v \leqslant 0 \vee w \leqslant 0$, they are meaningful when both cases of the disjunction can obtain under some circumstances. When only one of the atoms, say $v \leqslant w$, is satisfiable with all implication branches, we make an abductive guess that $v \leqslant w$.

B.6. Solving for Predicate Variables

Here we discuss the implementation in InvarGenT of ideas in the overview from Section 4.1 and in the formal discussion from Section 4.5. As we decided to provide the first solution to abduction problems, we accordingly simplify the task of solving for predicate variables. Instead of a tree of solutions being refined, we have a single sequence which we unfold until reaching fixpoint or contradiction. Another choice point besides abduction in the original algorithm is the selection of invariants that leaves a consistent subset of atoms as residuum. Here we also select the first solution found. We introduce a form of backtracking, described in section B.6.4.

B.6.1. Solving for Predicates in Premises

The core of our inference algorithm consists of distributing atoms of an abduction answer among the predicate variables. The negative occurrences of predicate variables, when instantiated with the updated solution, enrich the premises so that the next round of abduction leads to a smaller answer (in number of atoms).

Let us discuss the algorithm for $\mathrm{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^{\chi}}, \overline{A^0_{\chi}}\big)$. Note that due to existential types predicates, we actually compute $\mathrm{Split}\big(\mathcal{Q}, \bar{\alpha}, A, \overline{\bar{\beta}^{\beta_\chi}}, \overline{A^0_{\beta_\chi}}\big)$, i.e. we index by β_χ (which can be multiple for a single χ) rather than χ. We retain the notation indexing by χ as it better conveys the intent. We do not pass quantifiers around to reflect the source code: the helper function `loop avs ans sol` of function `split` corresponds to $\mathrm{Split}\big(\bar{\alpha}, A, \overline{A^0_{\beta_\chi}}\big)$.

$$
\begin{array}{rcl}
\alpha \prec \beta & \equiv & \alpha <_{\mathcal{Q}} \beta \vee \big(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not<_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar{\beta}^{\chi}} \wedge \beta \notin \overline{\bar{\beta}^{\chi}}\big) \\
A_{\alpha\beta} & = & \big\{\beta \doteq \alpha \in A \big| \beta \in \overline{\bar{\beta}^{\chi}} \wedge (\exists \alpha) \in \mathcal{Q} \wedge \beta \prec \alpha\big\} \\
A_0 & = & A \backslash A_{\alpha\beta} \\
A^1_{\chi} & = & \big\{c \in A_0 \big| \forall \alpha \in \mathrm{FV}(c).(\exists \alpha) \in \mathcal{Q} \vee \\
 & & \quad \alpha <_{\mathcal{Q}} \beta_\chi \wedge \alpha \notin \mathrm{PrimCV}(c) \vee \alpha \in \overline{\bar{\beta}^{\chi}} \wedge \alpha \in \mathrm{PrimCV}(c)\big\} \\
A^2_{\chi} & = & \mathrm{Atomized}\big(\overline{\bar{\beta}^{\chi}}, A^1_{\chi}\big) \\
A^3_{\chi} & = & A^2_{\chi} \setminus \cup_{\chi'} A^1_{\chi'} \\
\text{if} & & \mathcal{M} \nvDash \mathcal{Q}.(A \setminus \cup_{\chi} A^2_{\chi})[\bar{\alpha} := \bar{t}] \ \text{ for all } \bar{t} \\
\text{then return} & & \bot \\
\text{for all } \overline{A^+_{\chi}} \text{ min. w.r.t. } \subset \text{ s.t.} & & \wedge_{\chi}(A^+_{\chi} \subset A^2_{\chi}) \wedge \mathcal{M} \vDash \mathcal{Q}.(\cup_{\chi} A^+_{\chi} \Rightarrow A)[\bar{\alpha} := \bar{t}] \ \text{ for some } \bar{t}\text{:} \\
\text{if} & & \mathrm{Strat}(A^+_{\chi}, \bar{\beta}^{\chi}) \ \text{ returns } \bot \text{ for some } \chi \\
\text{then return} & & \bot \\
\text{else } \bar{\alpha}^{\chi}_+, A^L_{\chi}, A^R_{\chi} & = & \mathrm{Strat}(A^+_{\chi}, \bar{\beta}^{\chi}) \\
A_{\chi} & = & A^0_{\chi} \cup A^L_{\chi} \\
\bar{\alpha}^{\chi}_0 & = & \bar{\alpha} \cap \mathrm{FV}(A_{\chi}) \\
\bar{\alpha}^{\chi} & = & \left(\bar{\alpha}^{\chi}_0 \setminus \bigcup_{\chi' <_{\mathcal{Q}} \chi} \bar{\alpha}^{\chi'}_0\right) \bar{\alpha}^{\chi}_+ \\
A_+ & = & \cup_{\chi} A^R_{\chi} \\
A_{\mathrm{res}} & = & A_+ \cup \widetilde{A_+}(A \setminus \cup_{\chi} A^+_{\chi}) \\
\text{if} & & \cup_{\chi} \bar{\alpha}^{\chi} \neq \varnothing \vee \cup_{\chi} A^3_{\chi} \neq \varnothing \ \text{ then} \\
\mathcal{Q}', A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}'^{\chi}.A'_{\chi}} & \in & \mathrm{Split}\big(\mathcal{Q}\big[\overline{\forall \bar{\beta}^{\chi}} := \overline{\forall(\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi})}\big], \bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}, A_{\mathrm{res}} \wedge_{\chi} A^3_{\chi}, \\
 & & \qquad \overline{\bar{\beta}^{\chi} \cup \bar{\alpha}^{\chi}}, \overline{A_{\chi}}\big) \\
\text{return} & & \mathcal{Q}', A_{\alpha\beta} \wedge A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\chi} \bar{\alpha}'^{\chi}.A'_{\chi}} \\
\text{else return} & & \mathcal{Q} \exists(\bar{\alpha} \setminus \cup_{\chi} \bar{\alpha}^{\chi}), A_{\alpha\beta} \wedge A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\chi}.A_{\chi}}
\end{array}
$$

where $\mathrm{Strat}(A, \bar{\beta}^{\chi})$ is computed as follows: for every $c \in A$, and for every $\beta_2 \in \mathrm{FV}(c)$ such that $\beta_1 <_{\mathcal{Q}} \beta_2$ for $\beta_1 \in \bar{\beta}^{\chi}$, if β_2 is universally quantified in $\mathcal{Q}$ and $\beta_2 \notin \overline{\bar{\beta}^{\chi}}$, then return $\bot$; otherwise, introduce a fresh variable α_f, replace $c := c[\beta_2 := \alpha_f]$, add $\beta_2 \doteq \alpha_f$ to A^R_{χ} and α_f to $\bar{\alpha}^{\chi}_+$, after

replacing all such β_2 add the resulting c to A^L_χ. PrimCV(c) stands for *primary constrained variables* of an atom c. These are defined as the substituted variables in solved forms for term constraints, and in the case of a numerical atom, the shared variable of a directed opti or subopti atom, i.e. the variable v in $v \leqslant \max(\ldots)$, $\min(\ldots) \leqslant v$, $v = \max(\ldots)$, $v = \min(\ldots)$.

Description of the algorithm in more detail:

1. $\alpha \prec \beta \;\equiv\; \alpha <_{\mathcal{Q}} \beta \vee \big(\alpha \leqslant_{\mathcal{Q}} \beta \wedge \beta \not<_{\mathcal{Q}} \alpha \wedge \alpha \in \overline{\bar\beta^\chi} \wedge \beta \notin \overline{\bar\beta^\chi}\big)$ The variables $\overline{\bar\beta^\chi}$ are the invariant parameters of the solution from the previous round. We need to keep them apart from other variables even when they're not separated by quantifier alternation.

2. $A_0 = A \backslash A_{\alpha\beta}$ where $A_{\alpha\beta} = \big\{\beta \doteq \alpha \in A \big| \beta \in \overline{\bar\beta^\chi} \wedge (\exists\alpha) \in \mathcal{Q} \wedge \beta \prec \alpha\big\}$ Discard the "scaffolding" information, in particular the A_+ equations introduced by Strat in an earlier iteration.

3. $A^1_\chi = \{c \in A_0 | \forall \alpha \in \mathrm{FV}(c) \backslash \bar\beta^\chi . (\exists\alpha) \in \mathcal{Q} \vee \alpha <_{\mathcal{Q}} \beta_\chi\}$ Gather atoms pertaining to predicate χ, which should not remain in the residuum.

4. $A^2_\chi = \mathrm{Atomized}\big(\overline{\bar\beta^\chi}, A^1_\chi\big)$ An atomized form of A^1_χ wrt. $\overline{\bar\beta^\chi}$: a conjunction of atoms equivalent to A^1_χ containing some of the implications of A^1_χ, see the formal exposition of InvarGenT. Actually, rather than computing the atomized form upfront, we generate its contribution to A^+_χ while performing Fourier-Motzkin elimination to check $\mathcal{M} \vDash \mathcal{Q}.(\cup_\chi A^+_\chi \Rightarrow A)$.

5. $A^3_\chi = A^2_\chi \backslash \cup_{\chi'} A^1_{\chi'}$ We prune atoms coming from atomization that are already present in the invariants. Actually, in the implementation we prune by redundancy with the invariants assigned so far.

6. if $\mathcal{M} \nvDash \mathcal{Q}.(A \setminus \cup_\chi A^1_\chi)[\bar\alpha := \bar t]$ for all $\bar t$ then return $\bot$: Failed solution attempt. A common example is when the use site of recursive definition, resp. the existential type introduction site, is not in scope of a defining site of recursive definition, resp. an existential type elimination site, and has too strong requirements. FIXME:

7. for all $\overline{A^+_\chi}$ min. w.r.t. $\subset$ s.t.$\wedge_\chi(A^+_\chi \subset A^2_\chi) \wedge \mathcal{M} \vDash \mathcal{Q}.(\cup_\chi A^+_\chi \Rightarrow A)[\bar\alpha := \bar t]$ for some $\bar t$: Select invariants such that the residuum $A \setminus \cup_\chi A^+_\chi$ is consistent. The final residuum A_{res} represents the global constraints, the solution for global type variables. The solutions A^+_χ represent the invariants, the solution for invariant type parameters.

8. if $\mathrm{Strat}(A^+_\chi, \bar\beta^\chi)$ returns $\bot$ for some χ then return $\bot$ In the implementation, we address stratification issues already during abduction.

9. $\bar\alpha^\chi_+, A^L_\chi, A^R_\chi = \mathrm{Strat}(A^+_\chi, \bar\beta^\chi)$ is computed as follows: for every $c \in A^+_\chi$, and for every $\beta_2 \in \mathrm{FV}(c)$ such that $\beta_1 <_{\mathcal{Q}} \beta_2$ for $\beta_1 \in \bar\beta^\chi$, if β_2 is universally quantified in $\mathcal{Q}$, then return $\bot$; otherwise, introduce a fresh variable α_f, replace $c := c[\beta_2 := \alpha_f]$, add $\beta_2 \doteq \alpha_f$ to A^R_χ and α_f to $\bar\alpha^\chi_+$, after replacing all such β_2 add the resulting c to A^L_χ.

   - We will add $\bar\alpha^\chi_+$ to $\bar\beta^\chi$.

10. $A_\chi = A^0_\chi \cup A^L_\chi$ is the updated solution formula for χ, where A^0_χ is the solution from previous round.

11. $\bar{\alpha}^\chi_0 = \bar{\alpha} \cap \mathrm{FV}(A_\chi)$ are the additional solution parameters coming from variables generated by abduction.

12. $\bar{\alpha}^\chi = (\bar{\alpha}^\chi_0 \setminus \bigcup_{\chi' <_\mathcal{Q} \chi} \bar{\alpha}^{\chi'}_0)\bar{\alpha}^\chi_+$ The final solution parameters also include the variables generated by Strat.

13. $A_+ = \cup_\chi A^R_\chi$ and $A_{\mathrm{res}} = A_+ \cup \widetilde{A_+}(A \setminus \cup_\chi A^+_\chi)$ is the resulting global constraint, where $\widetilde{A_+}$ is the substitution corresponding to A_+.

14. if $\cup_\chi \bar{\alpha}^\chi \neq \varnothing \vee \cup_\chi A^3_\chi \neq \varnothing$ then – If new parameters or new atoms have been introduced, we need to redistribute the remaining atoms to make $\mathcal{Q}'.A_{\mathrm{res}}$ valid again.

15. $\mathcal{Q}', A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}'^\chi.A'_\chi} \in \mathrm{Split}\big(\mathcal{Q}\big[\overline{\forall \bar{\beta}^\chi} := \overline{\forall(\bar{\beta}^\chi \cup \bar{\alpha}^\chi)}\big], \bar{\alpha} \setminus \cup_\chi \bar{\alpha}^\chi, A_{\mathrm{res}} \wedge_\chi A^3, \overline{\bar{\beta}^\chi \cup \bar{\alpha}^\chi}, \overline{A_\chi}\big)$ Recursive call includes $\bar{\alpha}^\chi_+$ in $\bar{\beta}^\chi$ so that, among other things, A_+ are redistributed into A_χ.

16. return $\mathcal{Q}', A'_{\mathrm{res}}, \overline{\exists \bar{\alpha}^\chi \bar{\alpha}'^\chi.A'_\chi}$ We do not add $\forall \bar{\alpha}$ in front of $\mathcal{Q}'$ because it already includes these variables. $\overline{\bar{\alpha}^\chi_+ \bar{\alpha}^{\chi\prime}_+}$ lists all variables introduced by Strat.

17. else return $\mathcal{Q}\exists(\bar{\alpha} \setminus \cup_\chi \bar{\alpha}^\chi), A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^\chi.A_\chi}$ Note that $\bar{\alpha} \setminus \cup_\chi \bar{\alpha}^\chi$ does not contain the current $\bar{\beta}^\chi$, because $\bar{\alpha}$ does not contain it initially and the recursive call maintains that: $\bar{\alpha} := \bar{\alpha} \setminus \cup_\chi \bar{\alpha}^\chi$, $\bar{\beta}^\chi := \bar{\beta}^\chi \bar{\alpha}^\chi$.

Finally we define $\mathrm{Split}(\bar{\alpha}, A) := \mathrm{Split}(\bar{\alpha}, A, \bar{\boldsymbol{T}})$. The complete algorithm for solving predicate variables is presented in the next section.

B.6.2. Solving for Existential Types Predicates and Main Algorithm

The general scheme is that we perform constraint generalization on branches with positive occurrences of existential type predicate variables on each round. Since the branches are substituted with the solution from previous round, constraint generalization will automatically preserve monotonicity. We retain existential types predicate variables in the later abduction step.

What differentiates existential type predicate variables from recursive definition predicate variables is that the former are not local to context, while the latter are treated as local to context. It means that free variables need to be bound in the former while they are retained in the latter. However, it is just a technical difference because

In the algorithm we operate on all predicate variables, and perform additional operations on existential type predicate variables; there are no operations pertaining only to recursive definition predicate variables. The equation numbers (N) below are matched by comment numbers `(* N *)` in the source code. Let $\chi_{\varepsilon K}$ be the invariant which will constrain the existential type ε_K, or $\top$. When $\chi_{\varepsilon K} = \top$, then we define $\bar{\beta}^{\chi_{\varepsilon K},k} = \varnothing$. Step k of the loop of the final algorithm:

$$
\begin{aligned}
\overline{\exists \bar{\beta}^{\chi,k}.F_\chi} &= S_k \\
&\quad \text{In iteration 2, remove non-term-sort atoms} \\
&\quad \text{containing outer-scope parameters from } S_k. \\
D_K^\alpha \Rightarrow C_K^\alpha \in R_k^- S_k(\Phi) &= \text{all such that } \chi_K(\alpha, \alpha_\alpha^K) \in C_K^\alpha, \qquad \text{(B.1)} \\
&\quad \overline{C_j^\alpha} = \{C \,|\, D \Rightarrow C \in S_k(\Phi) \wedge D \subseteq D_K^\alpha\} \\
\alpha_K &: \chi_K(\alpha_K) \text{ is a positive atom in } \Phi \\
U_{\chi_K}, \exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K}, \overline{B_d^K} &= \mathrm{LUB}\big(\alpha_K, \overline{\delta \doteq \alpha \wedge D_K^\alpha \wedge_j C_j^\alpha}_{\alpha \in \overline{\alpha_3^{i,K}}}\big) \qquad \text{(B.2)} \\
\vec{\tau}_{\varepsilon_K} &= \overrightarrow{\big\{\alpha \in \mathrm{FV}(G_{\chi_K}) \backslash \delta\delta' \bar{\alpha}_g^{\chi_K} \big| (\exists \alpha) \in \mathcal{Q} \vee \alpha \in \overline{\bar{\beta}^\chi} \backslash \bar{\beta}^{\chi_K}\big\}} \\
\Xi(\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K}) &= \exists \mathrm{FV}(\vec{\tau}_{\varepsilon_K}, G_{\chi_K}) \backslash \delta\delta'.\delta' \doteq \vec{\tau}_{\varepsilon_K} \wedge G_{\chi_K} \\
(\exists \bar{\alpha}^{\chi_K}.F_{\chi_K}), \rho &= H(R_k(\chi_K), \Xi(\exists \bar{\alpha}_g^{\chi_K}.G_{\chi_K})) \qquad \text{(B.3)} \\
R_g(\chi_K) &= \exists \bar{\alpha}^{\chi_K}.F_{\chi_K} \\
P_g(\chi_K(\delta, \delta')) = P_g(\chi_K(\delta')) &= \delta' \doteq \vec{\tau}_{\varepsilon_K} \\
F'_\chi &= F_\chi[\varepsilon_K(\vec{\tau}_{\mathrm{old}}) := \varepsilon_K(\vec{\tau}_{\varepsilon_K})[\mathrm{recover}(\vec{\tau}_{\varepsilon_K}, \vec{\tau}_{\mathrm{old}})]] \\
S'_k &= \overline{\exists \bar{\beta}^{\chi,k} \{\alpha \in \mathrm{FV}(F'_\chi) | \beta_\chi <_{\mathcal{Q}} \alpha\}.F'_\chi} \qquad \text{(B.4)} \\
\mathcal{Q}'.\wedge_i (D_i \Rightarrow C_i) \wedge_j (D_j^- \Rightarrow \boldsymbol{F}) &= R_g^- P_g^+ S'_k(\Phi \wedge_{\chi_K} U_{\chi_K}) \\
\exists \bar{\alpha}.A_0 &= \mathrm{Abd}\big(\mathcal{Q}', \bar{\beta} = \overline{\beta_\chi \bar{\beta}^\chi}, \overline{D_i, C_i}\big) \qquad \text{(B.5)} \\
A &= A_0 \wedge_j \mathrm{NegElim}\big(\neg \mathrm{Simpl}\big(\mathrm{FV}(D_j^-) \backslash \overline{\bar{\beta}^\chi}.D_j^-\big), A_0, \overline{D_i, C_i}\big) \\
&\quad \text{In later iterations, check negative constraints.} \qquad \text{(B.6)} \\
\big(\mathcal{Q}^{k+1}, A_{\mathrm{res}}, \overline{\exists \bar{\alpha}^{\beta_\chi}.A_{\beta_\chi}}\big) &= \mathrm{Split}\big(\mathcal{Q}', \bar{\alpha}, A, \overline{\beta_\chi \bar{\beta}^\chi}\big) \\
\vec{\tau}'_{\varepsilon_K} &= \overrightarrow{\mathrm{FV}(\widetilde{A_{\mathrm{res}}}(\vec{\tau}_{\varepsilon_K}))} \\
R_{k+1}(\chi_K) &= \exists \bar{\beta}^{\chi_K,k} \overline{\bar{\alpha}^{\beta_{\chi_K}}} \bar{\alpha}^{\chi_K} \backslash \mathrm{FV}(\vec{\tau}'_{\varepsilon_K}).\delta' \doteq \vec{\tau}'_{\varepsilon_K} \wedge \widetilde{A_{\mathrm{res}}}(F_{\chi_K} \backslash \delta' \doteq \ldots) \\
&\qquad \wedge_{\chi_K} A_{\beta_{\chi_K}}\Big[\overline{\beta_{\chi_K} \bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\chi_K,k}}\Big] \qquad \text{(B.7)} \\
A_K^d &= \Big\{c \in A_{\beta_{\chi_K}}\Big[\overline{\beta_{\chi_K} \bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\chi_K,k}}\Big] \Big| \\
&\qquad \mathrm{FV}(c) \subseteq \mathrm{FV}(\rho(B_d^K))\Big\} \\
\exists \bar{\alpha}'_K.A'_K &= \mathrm{Abd}\big(\mathcal{Q}', \overline{\beta_\chi \bar{\beta}^\chi} \backslash \overline{\beta_{\chi_K} \bar{\beta}^{\chi_K}}, \overline{\rho(B_d^K), A_K^d}\big) \qquad \text{(B.8)} \\
\Big(\mathcal{Q}^{k+1}, \varnothing, \overline{\exists \bar{\alpha}^{\beta_\chi}{}'.A'_{\beta_\chi}}\Big) &= \mathrm{Split}\big(\mathcal{Q}^{k+1}, \overline{\bar{\alpha}'_K}, \wedge_K A'_K, \overline{\beta_\chi \bar{\beta}^\chi} \backslash \overline{\beta_{\chi_K} \bar{\beta}^{\chi_K}}\big) \\
S_{k+1}(\chi) &= \exists \bar{\beta}^{\chi,k}.\mathrm{Simpl}\big(\exists \overline{\bar{\alpha}^{\beta_\chi}}.F'_\chi \\
&\qquad \wedge A_{\beta_\chi}\big[\overline{\beta_\chi \bar{\beta}^\chi} := \overline{\delta \bar{\beta}^{\chi,k}}\big] \wedge A'_{\beta_\chi}\big) \qquad \text{(B.9)} \\
\text{if} \quad &(\forall \chi) S_{k+1}(\chi) \subseteq S_k(\chi), \qquad \text{(B.10)} \\
&(\forall \chi_K) R_{k+1}(\chi_K) = R_k(\chi_K),
\end{aligned}
$$

$$
\begin{array}{rl}
 & (\forall \beta_{\chi_K}) A_{\beta_{\chi_K}} = \boldsymbol{T}, \\
 & k > 1 \\
\text{then return} & A_{\text{res}}, S_{k+1}, R_{k+1} \\
\text{repeat} & k := k+1
\end{array}
$$

Note that Split returns $\overline{\exists \bar{\alpha}^{\beta_\chi}.A_{\beta_\chi}}$ rather than $\overline{\exists \bar{\alpha}^{\chi}.A_{\chi}}$. This is because in case of existential type predicate variables χ_K, there can be multiple negative position occurrences $\chi_K(\beta_{\chi_K}, \cdot)$ with different β_{χ_K} when the corresponding value is used in multiple **let**...**in** expressions. The variant of the algorithm to achieve completeness would compute all answers for variants of Abd and Split algorithms that return multiple answers. Unary predicate variables $\chi(\beta_\chi)$ can also have multiple negative occurrences in the normalized form, but always with the same argument β_χ. The substitution $\left[\overline{\beta_{\chi_K} \bar{\beta}^{\beta_{\chi_K}}} := \overline{\delta \bar{\beta}^{\chi_K, k}}\right]$ replaces the instance parameters introduced in $\chi_K(\beta_{\chi_K}, \cdot)$ by the formal parameters used in $R_k(\chi_K)$.

Even for a fixed K, the same branch $D_K^\alpha \Rightarrow C_K^\alpha$ can contribute to multiple disjuncts, with different $\alpha = \alpha_3^{i,K}$. Substitution R_g^- substitutes only negative occurrences of χ_K, i.e. it affects only the premises. Substitution P_g^+ substitutes only positive occurrences of χ_K, i.e. it affects only the conclusions. P_g^+ ensures that the parameter instances of the postcondition are connected with the argument variables. As a matter of implementation, the substitution recover$(\vec{\tau}_{\varepsilon_K}, \vec{\tau}_{\text{old}})$ is actually derived from the way the variables are freshened in step B.4 above. Its effect is currently implemented as a substitution over the intermediate formula $F_\chi[\varepsilon_K(\vec{\tau}_{\text{old}}) := \varepsilon_K(\vec{\tau}_{\varepsilon_K})]$.

We start with $S_0 := \bar{\boldsymbol{T}}$ and $R_0 := \bar{\boldsymbol{T}}$. S_k grow in strength by definition. The constraint generalization parts G_{χ_K} of R_1 and R_2 are computed from non-recusrive branches only. Starting from R_2, R_k are expected to decrease in strength, but monotonicity is not guaranteed because of contributions from abduction: mostly in form of $A_{\beta_{\chi_K}}$, but also from stronger premises due to S_k. We remove non-term-sort atoms containing containing outer-scope parameters from S_2, to enable considering a different solution when new facts about outer-scope parameters propagate.

Connected(α, G) is the connected component of hypergraph G containing node α, where nodes are variables FV(G) and hyperedges are atoms $c \in G$. Connected$_0(\delta, (\bar{\alpha}, G))$ additionally removes atoms c: FV$(c) \# \delta\bar{\alpha}$. In initial iterations, when the branches $D_K^\alpha \Rightarrow C_K^\alpha$ are selected from non-recursive branches only, we include a connected atom only if it is satisfiable in all branches. $H(R_k, R_{k+1})$ is a convergence improving heuristic, with $H(R_k, R_{k+1}) = \rho(R_g)$ for early iterations and "roughly" $H(R_k,\ R_g) = R_k \cap \rho(R_g)$ later. The substitution H, also computed by the function H, is a renaming that replaces the variables introduced by constraint generalization in the current iteration, by the corresponding variables introduced by constraint generalization in the previous iteration.

The use sites of definitions with postconditions can introduce requirements $A_{\beta_{\chi_K}}$ on postconditions. The inferred postconditions can only meet those requirements which are guaranteed by all defining cases. We use abduction again to strengthen the preconditions, so that the postconditions meet the requirements. Abduction is applied directly to the constraint branches preprocessed by constraint generalization, so that the required postconditions will follow in the next iteration.

The condition $(\forall \beta_{\chi_K}) A_{\beta_{\chi_K}} = \boldsymbol{T}$ is required for correctness of returned solution. Fixpoint of postconditions $R_k(\chi_K)$ is not a sufficient stopping condition, because it does not imply $A_{\beta_{\chi_K}} = \boldsymbol{T}$, the same $A_{\beta_{\chi_K}} \neq \boldsymbol{T}$ may be introduced in consecutive iterations. This is not the case for invariants, where $S_k(\chi)$ and $S_{k+1}(\chi)$ differ by the portion A_{β_χ} of abduction answer.

We introduced the `assert false` construct into the programming language to indicate that a branch of code should not be reached. Type inference generates for it the logical connective $\boldsymbol{F}$ (falsehood). We partition the implication branches D_i, C_i into $\{D_i, C_i | \boldsymbol{F} \notin C_i\}$ which are fed to the algorithm and $\{D_j^-\} = \{D_i | \boldsymbol{F} \in C_i\}$. After the main algorithm ends we check that for each D_j^-, $S_k(D_i)$ fails. Optionally, but by default, we perform the check in each iteration, starting from the third iteration, i.e. $k > 1$. Turning this option *on* gives a way to express negative constraints for the term domain. The option should be turned *off* when a single iteration (plus fallback backtracking described below) is insufficient to solve for the invariants. Moreover, for domains with negation elimination, i.e. the numerical domain, we incorporate negative constraints as described in section B.4. The corresponding computation is $A_0 \wedge_j \mathrm{NegElim}\big(\neg\mathrm{Simpl}\big(\mathrm{FV}(D_j^-) \backslash \overline{\beta^\chi}.D_j^-\big), \overline{D_i, C_i}\big)$ in the specification of the main algorithm. The simplification reduces the number of atoms in the formula and keeps those that are most relevant to finding invariants and postconditions. For convenience, negation elimination is called from the function that computes the abduction answer.

We implement backtracking using a tabu-search-like discard list. When abduction raises an exception: for example contradiction arises in the branches $S_k(\Phi)$ passed to it, or it cannot find an answer and raises `Suspect` with information on potential problem, we fall-back to step $k-1$. Similarly, with checking for negative constraints *on*, when the check of negative branches $D_i \in \Phi_{\boldsymbol{F}}$, $\mathcal{M} \not\models \exists \mathrm{FV}(S_k(D_i)).S_k(D_i)$ fails. In step $k-1$, we maintain a discard list of different answers found not to work in this step and previous steps: initially empty, after fall-back we add there the latest partial solution. We redo step $k-1$ starting from $S_{k-1}(\Phi)$. Infinite loop is avoided because answers already attempted are discarded. When step $k-1$ cannot find a new partial solution, we fall back to step $k-2$, etc. We store discard lists for distinct sorts separately and we only add to the discard list of the sort that caused fallback. Unfortunately this means a slight loss of completeness, as an error in another sort might be due to bad choice in the sort of types. The loss is "slight" because of the dissociation step described previously. Moreover, the sort information from checking negative branches is likewise only approximate.

B.6.3. Stages of Iteration

Changes in the algorithm between iterations were mentioned above but not clearly exposed. We divide iterations into multiple stages, demarcated by parameters j_k we now describe.

1. j_0 is when inferring any postconditions starts; we use the initial iteration to infer the shape of types using abduction.

2. j_1 is when inferring numerical postconditions starts.

3. j_2 is when using all branches of the constraint in inference starts; before j_2, we only use the branches of the constraint that do not contain requirements derived from recursive calls. If we use all branches too early, we suffer from missing information.

4. j_3 is when second-phase abduction, to enrich preconditions based on postconditions, starts.

5. j_4 is when any forms of guessing in constraint generation should end; we have not yet implemented any such guessing mechanisms.

6. j_5 is when convergence of postconditions is enforced; from this step onward, a solution to postconditions in an iteration is truncated to contain only atoms matching atoms from the previous iteration.

7. j_6 is the last iteration; if the fixpoint is not reached, we signal an error *"Answers do not converge"*.

Default settings are $[j_0; j_1; j_2; j_3; j_4; j_5; j_6] = [1; 1; 2; 3; 4; 8; 11]$ where the initial iteration has index 0. In a single iteration, constraint generalization precedes abduction.

When existential types are used, the expected number of iterations is $k = 5$ (six iterations), because the last iteration needs to verify that the last-but-one iteration has found the correct solution. The minimal number of iterations is $k = 2$ (three iterations), so that all branches are considered.

B.6.4. Implementation Details

We represent $\vec{\alpha}$ as a tuple type rather than as a function type. We modify the quantifier $\mathcal{Q}$ imperatively, because it mostly grows monotonically, and when variables are dropped they do not conflict with fresh variables generated later.

The code that selects $\overline{A^+_\chi}$ by $\wedge_\chi(A^+_\chi \subset A^1_\chi)$ and $\mathcal{M} \vDash \mathcal{Q}.A \setminus \cup_\chi A^+_\chi$ is an incremental validity checker. It starts with $A \setminus \cup_\chi A^1_\chi$ and tries to add as many atoms $c \in \cup_\chi A^1_\chi$ as possible to what in effect becomes A_{res}.

We count the number of iterations of the main loop, a fallback decreases the iteration number to the previous value. The main loop decides whether multisort abduction should dissociate alien subterms – in the first iteration of the loop – or should perform abductions for other sorts – in subsequent iteration. See discussion in subsection B.2.1. In the first two iterations, we remove branches that contain unary predicate variables in the conclusion (or binary predicate variables in the premise, keeping only non-recursive branches), as discussed at the end of subsection B.2.4 and beginning of subsection B.3.2. As discussed in subsection B.3.2, starting from iteration k_3, we enforce convergence on solutions for binary predicate variables.

Computing abduction is the "axis" of the main loop. If anything fails, the previous abduction answer is the culprit. We add the previous answer to the discard list and retry, without incrementing the iteration number. If abduction and splitting succeeds, we reset the discard list and increment the iteration number. We use recursion for backtracking, instead of making `loop` tail-recursive.

B.7. Generating OCaml Source and Interface Code

We have a single basis from which to generate all generated output files: `.gadti`, `.ml` – `annot_item`. It contains a superset of information in `struct_item`: type scheme annotations on introduced names, and source code annotated with type schemes at recursive definition nodes. We use `type a.` syntax instead of `'a.` syntax because the former supports inference for GADTs in OCaml. A benefit of the nicer `type a.` syntax is that nested type schemes can have free variables, which will be correctly captured by the outer type scheme. For completeness we sometimes need to annotate all `function` nodes with types. To avoid clutter, we start by only annotating `let rec` nodes, and in case `ocamlc -c` fails on generated code, we re-annotate by putting type schemes on `let rec` nodes and types on `function` nodes. If need arises, `let-in` node annotations can also be introduced in this fallback – we provide an option to annotate the definitions on `let-in` nodes. Type annotations are optional because they introduce a slight burden on the solver – the corresponding variables cannot be removed by the initial simplification of the constraints. `let-in` node annotations are more burdensome than `function` node annotations.

In the signature declarations for existential types, we replace existential identifiers with regular indentifiers of constructors, to get informative output for printing the various result files. We print constraint formulas and alien subterms in the original InvarGenT syntax, commented out.

The types `Int`, `Num`, `Bool` and `String` should be considered built-in. `Int`, `Bool` and `String` follow the general scheme of exporting a datatype constructor with the same name, only lower-case. However, numerals `0`, `1`, ... are always type-checked as `Num 0`, `Num 1`... A parameter `-num_is` decides the type alias definition added in the generated code: `-num_is bar` adds `type num = bar` in front of an `.ml` file, by default `int`. Numerals are exported as integers passed to a `bar_of_int` function. The variant `-num_is_mod` exports numerals by passing to a `Bar.of_int` function. Special treatment for `Bool` amounts to exporting `True` and `False` as `true` and `false`, unlike other constants. In addition, pattern matching `match`... `with True ->`... `| False ->`..., i.e. the corresponding beta-redex, is exported as the `if`... `then`... `else`... expression.

In declarations which have concrete syntax starting with the word `external`, we provide names assumed to have given type scheme. The syntax has two variants, differing in the way the declaration is exported. It can be either an `external` declaration in OCaml, which is the binding mechanism of the *foreign function interface*. But in the InvarGenT form `external let`, the declaration provides an OCaml definition, which is exported as the toplevel `let` definition of OCaml. It has the benefit that the OCaml compiler will verify this definition, since InvarGenT calls `ocamlc -c` to verify the exported code.

Appendix C

Source Code of Examples

Subsection titles start with "Function" for examples from Tables 5.1 and 5.2, and "Program" for examples from Table 5.3.

C.1. Incompleteness Example for OutsideIn: Function rx

Example from [53], described on page 50. Source file *non_ outsidein_ rx.gadt*:

```
datatype R : type
datacons RBool : R Bool
external let fortytwo : Int = "42"

let rx = function RBool -> fortytwo
```

Value items from *non_ outsidein_ rx.gadti.target*:

```
val rx : ∀a. R a → Int
```

C.2. Pointwise Examples

These are the longer examples from [23] within the scope of algorithm $\mathcal{P}$.

C.2.1. Function `rotate`

Example from [22], described on pages 193-194. Source file *pointwise_ rbtree_ rotate.gadt*:

```
datatype Z
datatype S : type

datatype RoB : type * type
datatype Black
datatype Red
datacons Leaf : RoB (Black, Z)
datacons RNode : ∀a. RoB (Black, a) * Int * RoB (Black, a) ⟶ RoB (Red, a)
datacons BNode :
  ∀a, b, c. RoB (a, c) * Int * RoB (b, c) ⟶ RoB (Black, S c)
```

```
datatype Dir
datacons LeftD : Dir
datacons RightD : Dir

let rotate = fun dir1 p_e sib dir2 g_e uncle -> function
| RNode (x, e, y) ->
  (match dir1, dir2 with
  | RightD, RightD -> BNode (RNode (x, e, y), p_e, RNode (sib, g_e, uncle))
  | RightD, LeftD -> BNode (RNode (uncle, g_e, x), e, RNode (y, p_e, sib))
  | LeftD, RightD -> BNode (RNode (sib, p_e, x), e, RNode (y, g_e, uncle))
  | LeftD, LeftD -> BNode (RNode (uncle, g_e, sib), p_e, RNode (x, e, y)))
| BNode (_, _, _) -> assert false
```

Value items from *pointwise_rbtree_rotate.gadti.target*:

```
val rotate :
   ∀a.
   Dir → Int → RoB (Black, a) → Dir → Int → RoB (Black, a) →
     RoB (Red, a) → RoB (Black, S a)
```

C.2.2. Function zip2: *N*-way zip_with

Example from [22], provided on page 206. The definition of z2 is η-expanded here to fit the call-by-value semantics of InvarGenT. Source file *pointwise_zip2.gadt*:

```
datatype List : type
datacons N : ∀a. List a
datacons C : ∀a. a * List a ⟶ List a

datatype Zip2 : type * type
datacons Zero2 : ∀a. Zip2 (a, List a)
datacons Succ2 :
  ∀a, b, c. Zip2 (a, b) ⟶ Zip2 ((c → a), (List c → b))

let zip2 =
  let rec zipZ = function
    | Zero2 -> N
    | Succ2 n -> fun _ -> zipZ n in
  let rec zipS = fun f r -> function
    | Zero2 -> C (f, r)
    | Succ2 n -> function
      | N -> zipZ n
      | C (z, zs) -> zipS (f z) (r zs) n in
  let rec z2 = fun n f -> zipS f (z2 n f) n in
  z2
```

Value items from *pointwise_ zip2.gadti.target*:

```
val zip2 : ∀a, b. Zip2 (b, a) → b → a
```

C.2.3. Function `rotl`

Example from [22], described on pages 198-199. Source file *pointwise_ avl_ rotl.gadt*:

```
datatype Z
datatype S : type
datatype AVL : type
datacons Tip : AVL Z
datacons LNode : ∀a. AVL a * Int * AVL (S a) ⟶ AVL (S (S a))
datacons SNode : ∀a. AVL a * Int * AVL a ⟶ AVL (S a)
datacons MNode : ∀a. AVL (S a) * Int * AVL a ⟶ AVL (S (S a))

datatype Choice : type * type
datacons L : ∀a, b. a ⟶ Choice (a, b)
datacons R : ∀a, b. b ⟶ Choice (a, b)

let rotl = fun u v -> function
  | Tip -> assert false
  | SNode (a, x, b) -> R (MNode (LNode (u, v, a), x, b))
  | LNode (a, x, b) -> L (SNode (SNode (u, v, a), x, b))
  | MNode (k, y, c) ->
    (match k with
      | SNode (a, x, b) -> L (SNode (SNode (u, v, a), x, SNode (b, y, c)))
      | LNode (a, x, b) -> L (SNode (MNode (u, v, a), x, SNode (b, y, c)))
      | MNode (a, x, b) -> L (SNode (SNode (u, v, a), x, LNode (b, y, c))))
```

Value items from *pointwise_ avl_ rotl.gadti.target*:

```
val rotl :
   ∀a.
   AVL a → Int → AVL (S (S a)) →
     Choice (AVL (S (S a)), AVL (S (S (S a))))
```

C.2.4. Function `ins`

Example from [22], provided on page 209. Source file *pointwise_ avl_ ins.gadt*:

```
datatype Z
datatype S : type
datatype AVL : type
datacons Tip : AVL Z
datacons LNode : ∀a. AVL a * Int * AVL (S a) ⟶ AVL (S (S a))
datacons SNode : ∀a. AVL a * Int * AVL a ⟶ AVL (S a)
```

```
datacons MNode : ∀a. AVL (S a) * Int * AVL a ⟶ AVL (S (S a))

datatype Choice : type * type
datacons L : ∀a, b. a ⟶ Choice (a, b)
datacons R : ∀a, b. b ⟶ Choice (a, b)

datatype LinOrder
datacons LT : LinOrder
datacons EQ : LinOrder
datacons GT : LinOrder
external let compare : ∀a. a → a → LinOrder =
  "fun x y -> let c=Pervasives.compare x y in
              if c<0 then LT else if c=0 then EQ else GT"

external rotl :
  ∀a. AVL a → Int → AVL (S (S a)) →
    Choice (AVL (S (S a)), AVL (S (S (S a))))

external rotr :
  ∀a. AVL (S (S a)) → Int → AVL a →
    Choice (AVL (S (S a)), AVL (S (S (S a))))

let rec ins = fun i -> function
  | Tip -> R (SNode (Tip, i, Tip))
  | SNode (a, x, b) as t ->
    (match compare i x with
      | EQ -> L t
      | LT ->
        (match ins i a with
          | L a -> L (SNode (a, x, b))
          | R a -> R (MNode (a, x, b)))
      | GT ->
        (match ins i b with
          | L b -> L (SNode (a, x, b))
          | R b -> R (LNode (a, x, b))))
  | LNode (a, x, b) as t ->
    (match compare i x with
      | EQ -> L t
      | LT ->
        (match ins i a with
          | L a -> L (LNode (a, x, b))
          | R a -> L (SNode (a, x, b)))
      | GT ->
        (match ins i b with
          | L b -> L (LNode (a, x, b))
```

```
          | R b -> rotl a x b))
  | MNode (a, x, b) as t ->
    (match compare i x with
      | EQ -> L t
      | LT ->
        (match ins i a with
          | L a -> L (MNode (a, x, b))
          | R a -> rotr a x b)
      | GT ->
        (match ins i b with
          | L b -> L (MNode (a, x, b))
          | R b -> L (SNode (a, x, b))))
```

Value items from *pointwise_ avl_ ins.gadti.target*:

```
val ins : ∀a. Int → AVL a → Choice (AVL a, AVL (S a))
```

C.2.5. Function `extract`

Example from [22], provided on page 207. Source file *pointwise_ extract.gadt*:

```
datatype List : type
datacons N : ∀a. List a
datacons C : ∀a. a * List a ⟶ List a

datatype Nd
datatype Fk : type * type

datatype Tree : type * type
datacons End : ∀a. a ⟶ Tree (Nd, a)
datacons Fork : ∀a, b, c. Tree (a, c) * Tree (b, c) ⟶ Tree (Fk (a, b), c)

datatype Path : type
datacons Here : Path Nd
datacons ForkL : ∀a, b. Path a ⟶ Path (Fk (a, b))
datacons ForkR : ∀a, b. Path b ⟶ Path (Fk (a, b))

external append : ∀a. List a → List a → List a
external map : ∀a, b. (a → b) → List a → List b

let rec find f = function
  | End m -> if f m then C (Here, N) else N
  | Fork (x, y) -> append (map (fun y -> ForkL y) (find f x))
                          (map (fun y -> ForkR y) (find f y))

let rec extract = fun p t ->
```

```
  match p with
    | Here -> (match t with End m -> m | Fork (_, _) -> assert false)
    | ForkL p1 -> (match t with Fork (x, y) -> extract p1 x)
    | ForkR p1 -> (match t with Fork (x, y) -> extract p1 y)
```

Value items from *pointwise_extract.gadti.target*:

```
val find : ∀a, b. (a → Bool) → Tree (b, a) → List (Path b)

val extract : ∀a, b. Path b → Tree (b, a) → a
```

C.2.6. Function `run_state`

Example from [22], provided on page 208. Source file *pointwise_run_state.gadt*:

```
datatype State : type * type
datacons Bind :
  ∀a, b, s. State (s, a) * (a → State (s, b)) ⟶ State (s, b)
datacons Return : ∀a, s. a ⟶ State (s, a)
datacons Get : ∀s. State (s, s)
datacons Put : ∀s. s ⟶ State (s, ())

let rec run_state = fun s -> function
  | Return a -> (s, a)
  | Get -> (s, s)
  | Put u -> (u, ())
  | Bind (m, k) ->
    let s1, a1 = run_state s m in
    run_state s1 (k a1)
```

Value items from *pointwise_run_state.gadti.target*:

```
val run_state : ∀a, b. a → State (a, b) → (a, b)
```

C.3. NON-POINTWISE EXAMPLES

These are the examples from [23] falling outside the scope of algorithm $\mathcal{P}$.

C.3.1. Function `joint`

Example from [22], described on page 86. Source file *non_pointwise_split.gadt*:

```
datatype Split : type * type
datacons Whole : Split (Int, Int)
datacons Parts : ∀a, b. Split ((Int, a), (b, Bool))
external let seven : Int = "7"
```

```
external let three : Int = "3"

let joint = function
  | Whole -> seven
  | Parts -> three, True
```

Value items from *non_pointwise_split.gadti.target*:

```
val joint : ∀a. Split (a, a) → a
```

C.3.2. Function `rotr`

Example from [22], described on page 88. Source file *non_pointwise_avl.gadt*:

```
(** Normally we would use [num], but this is a stress test for [type]. *)
datatype Z
datatype S : type
datatype Balance : type * type * type
datacons Less : ∀a. Balance (a, S a, S a)
datacons Same : ∀a. Balance (a, a, a)
datacons More : ∀a. Balance (S a, a, S a)
datatype AVL : type
datacons Leaf : AVL Z
datacons Node :
  ∀a, b, c. Balance (a, b, c) * AVL a * Int * AVL b ⟶ AVL (S c)

datatype Choice : type * type
datacons Left : ∀a, b. a ⟶ Choice (a, b)
datacons Right : ∀a, b. b ⟶ Choice (a, b)

let rotr = fun z d -> function
  | Leaf -> assert false
  | Node (Less, a, x, Leaf) -> assert false
  | Node (Same, a, x, (Node (_,_,_,_) as b)) ->
    Right (Node (Less, a, x, Node (More, b, z, d)))
  | Node (More, a, x, (Node (_,_,_,_) as b)) ->
    Left (Node (Same, a, x, Node (Same, b, z, d)))
  | Node (Less, a, x, Node (Same, b, y, c)) ->
    Left (Node (Same, Node (Same, a, x, b), y, Node (Same, c, z, d)))
  | Node (Less, a, x, Node (Less, b, y, c)) ->
    Left (Node (Same, Node (More, a, x, b), y, Node (Same, c, z, d)))
  | Node (Less, a, x, Node (More, b, y, c)) ->
    Left (Node (Same, Node (Same, a, x, b), y, Node (Less, c, z, d)))
```

Value items from *non_pointwise_avl.gadti.target*:

```
val rotr :
```

```
   ∀a.
   Int → AVL a → AVL (S (S a)) →
     Choice (AVL (S (S a)), AVL (S (S (S a))))
```

C.3.3. Function `delmin`

Example from [22], described on pages 212-213. Source file *non_ pointwise_ avl_ delmin.gadt*:

```
(** Normally we would use [num], but this is a stress test for [type]. *)
datatype Z
datatype S : type
(** This datatype definition is modified to make type inference for
    rotr, rotl, ins functions pointwise. *)
datatype AVL : type
datacons Tip : AVL Z
datacons LNode : ∀a. AVL a * Int * AVL (S a) ⟶ AVL (S (S a))
datacons SNode : ∀a. AVL a * Int * AVL a ⟶ AVL (S a)
datacons MNode : ∀a. AVL (S a) * Int * AVL a ⟶ AVL (S (S a))

datatype Choice : type * type
datacons L : ∀a, b. a ⟶ Choice (a, b)
datacons R : ∀a, b. b ⟶ Choice (a, b)

datatype Zero : type
datacons IsZ : Zero Z
datacons NotZ : ∀a. Zero (S a)

external rotl :
  ∀a. AVL a → Int → AVL (S (S a)) → Choice (AVL (S (S a)),
                                            AVL (S (S (S a))))

external rotr :
  ∀a. AVL (S (S a)) → Int → AVL a → Choice (AVL (S (S a)),
                                            AVL (S (S (S a))))

let empty = function
  | Tip -> IsZ
  | LNode (_, _, _) -> NotZ
  | SNode (_, _, _) -> NotZ
  | MNode (_, _, _) -> NotZ

let rec delmin = function
  | LNode (a, x, b) ->
    (match empty a with
      | IsZ -> x, L b
      | NotZ ->
```

```
        (match delmin a with
          y, k ->
          (match k with
            | L a -> y, rotl a x b
            | R a -> y, R (LNode (a, x, b)))))
  | SNode (a, x, b) ->
    (match empty a with
      | IsZ -> x, L b
      | NotZ ->
        (match delmin a with
          y, k ->
          (match k with
            | L a -> y, R (LNode (a, x, b))
            | R a -> y, R (SNode (a, x, b)))))
  | MNode (a, x, b) ->
    (match delmin a with
      y, k ->
      (match k with
        | L a -> y, L (SNode (a, x, b))
        | R a -> y, R (MNode (a, x, b))))
```

Value items from *non_pointwise_avl_delmin.gadti.target*:

```
val empty : ∀a. AVL a → Zero a

val delmin : ∀a. AVL (S a) → (Int, Choice (AVL a, AVL (S a)))
```

C.3.4. Function `fd_comp`

Example from [22], described on pages 213-215. The only difference is that the original has `| FDI -> fd2` as the fourth line of `fd_comp` instead of our expanded `| FDI -> (match fd2 with | FDI -> fd2 | FDC _ -> fd2 | FDG _ -> fd2)`. Source file *non_pointwise_fd_comp.gadt*:

```
datatype FunDesc : type * type
datacons FDI : ∀a. FunDesc (a, a)
datacons FDC : ∀a, b. b ⟶ FunDesc (a, b)
datacons FDG : ∀a, b. (a → b) ⟶ FunDesc (a, b)

external fd_fun : ∀a, b. FunDesc (a, b) → a → b

let fd_comp = fun fd1 fd2 ->
  let o = fun f g x -> f (g x) in
  match fd1 with
    | FDI -> (match fd2 with | FDI -> fd2 | FDC _ -> fd2 | FDG _ -> fd2)
    | FDC b ->
```

```
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC (fd_fun fd2 b)
        | FDG g -> FDC (fd_fun fd2 b))
    | FDG f ->
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC c
        | FDG g -> FDG (o (fd_fun fd2) f))
```

Compare also the example *non_pointwise_fd_comp2.gadt*:

```
datatype FunDesc : type * type
datacons FDI : ∀a. FunDesc (a, a)
datacons FDC : ∀a, b. b ⟶ FunDesc (a, b)
datacons FDG : ∀a, b. (a → b) ⟶ FunDesc (a, b)

external fd_fun : ∀a, b. FunDesc (a, b) → a → b

let o = fun f g x -> f (g x)

let fd_comp =
  function
    | FDI -> (function FDI -> FDI | FDC c -> FDC c | FDG g -> FDG g)
    | FDC b as fd1 ->
      (function
        | FDI -> fd1
        | FDC c as fd2 -> FDC (fd_fun fd2 b)
        | FDG g as fd2 -> FDC (fd_fun fd2 b))
    | FDG f as fd1 ->
      (function
        | FDI -> fd1
        | FDC c -> FDC c
        | FDG g as fd2 -> FDG (o (fd_fun fd2) f))
```

Value items from *non_pointwise_fd_comp2.gadti.target*:

```
val o : ∀a, b, c. (b → a) → (c → b) → c → a

val fd_comp :
   ∀a, b, c. FunDesc (b, c) → FunDesc (c, a) → FunDesc (b, a)
```

C.3.5. Function `zip1`: *N*-way `zip_with`

Example from [22], described on pages 216-217. The local definition of `apply` is a `let rec` here because it needs to be polymorphic. Source file *non_pointwise_zip1.gadt*:

```
datatype List : type
datacons N : ∀a. List a
datacons C : ∀a. a * List a ⟶ List a

datatype Zip1 : type * type
datacons Zero1 : ∀a. Zip1 (List a, List a)
datacons Succ1 :
  ∀a, b, c. Zip1 (List a, b) ⟶ Zip1 (List (c → a), (List c → b))

external zip_with : ∀a, b, c. (a → b → c) → List a → List b → List c

external repeat : ∀a. a → List a

let zip1 =
  let rec apply = fun f x -> f x in
  let rec z1 = fun fs -> function
    | Zero1 -> fs
    | Succ1 n2 -> fun xs -> z1 (zip_with apply fs xs) n2 in
  fun n f -> z1 (repeat f) n
```

Value items from *non_pointwise_zip1.gadt*:

```
val zip1 : ∀a, b. Zip1 (List b, a) → b → a
```

C.3.6. Function `leq`

Example from [22], described on pages 217-219. Source file *non_pointwise_leq.gadt*, requires option `-prefer_guess` for inference:

```
datatype Z
datatype S : type

datatype Nat : type
datacons Zn : Nat Z
datacons Sn : ∀a. Nat a ⟶ Nat (S a)

datatype NatLeq : type * type
datacons LeZ : ∀a. NatLeq (Z, a)
datacons LeS : ∀a, b. NatLeq (a, b)  ⟶ NatLeq (S a, S b)

let rec leq = function
  | Zn -> LeZ
  | Sn n -> LeS (leq n)
```

Value items from *non_pointwise_leq.gadti.target*:

```
val leq : ∀a. Nat a → NatLeq (a, a)
```

C.3.7. Function `run_state`

Example from [22], described on pages 219-220. Source file *non_pointwise_run_state.gadt*:

```
datatype State : type * type
datacons Bind :
  ∀a, b, s. State (s, a) * (a → State (s, b)) ⟶ State (s, b)
datacons Return : ∀a, s. a ⟶ State (s, a)
datacons Get : ∀s. State (s, s)
datacons Put : ∀s. s ⟶ State (s, ())

let rec run_state = fun s -> function
  | Return a -> (s, a)
  | Get -> (s, s)
  | Put u -> (u, ())
  | Bind (m, k) ->
    match m with
      | Return a -> run_state s (k a)
      | Get -> run_state s (k s)
      | Put u -> run_state u (k ())
      | Bind (n, j) -> run_state s (Bind (n, fun x -> Bind (j x, k)))
```

Value items from *non_pointwise_run_state.gadti.target*:

```
val run_state : ∀a, b. a → State (a, b) → (a, b)
```

C.4. Run-time Type Representations

C.4.1. Function `eval`

Source file *eval.gadt*:

```
datatype Term : type

external let plus : Int → Int → Int = "(+)"
external let is_zero : Int → Bool = "(=) 0"

datacons Lit : Int ⟶ Term Int
datacons Plus : Term Int * Term Int ⟶ Term Int
datacons IsZero : Term Int ⟶ Term Bool
datacons If : ∀a. Term Bool * Term a * Term a ⟶ Term a
datacons Pair : ∀a, b. Term a * Term b ⟶ Term (a, b)
datacons Fst : ∀a, b. Term (a, b) ⟶ Term a
datacons Snd : ∀a, b. Term (a, b) ⟶ Term b

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) -> (match eval b with True -> eval t | False -> eval e)
  | Pair (x, y) -> eval x, eval y
  | Fst p -> (match eval p with x, y -> x)
  | Snd p -> (match eval p with x, y -> y)
```

Value items from *eval.gadti.target*:

```
val eval : ∀a. Term a → a
```

Exported OCaml source file *eval.ml.target*:

```
type num = int
type _ term =
  | Lit : int -> int term
  | Plus : int term * int term -> int term
  | IsZero : int term -> bool term
  | If : (*∀'a.*)bool term * 'a term * 'a term -> 'a term
  | Pair : (*∀'a, 'b.*)'a term * 'b term -> (('a * 'b)) term
  | Fst : (*∀'a, 'b.*)(('a * 'b)) term -> 'a term
  | Snd : (*∀'a, 'b.*)(('a * 'b)) term -> 'b term
let plus : (int -> int -> int) = (+)
let is_zero : (int -> bool) = (=) 0
let rec eval : type a . (a term -> a) =
  ((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 then eval t else eval e)
    | Pair (x, y) -> (eval x, eval y)
    | Fst p -> let ((x, y): (a * _)) = eval p in x
    | Snd p -> let ((x, y): (_ * a)) = eval p in y): a term -> a)
```

C.4.2. Function `equal`

Source file *equal_assert.gadt*:

```
datatype Ty : type
datatype List : type
datacons Zero : Int
datacons Nil : ∀a. List a
datacons TInt : Ty Int
datacons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
datacons TList : ∀a. Ty a ⟶ Ty (List a)
external let eq_int : Int → Int → Bool = "(=)"
external let b_and : Bool → Bool → Bool = "(&&)"
external let b_not : Bool → Bool = "not"
```

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

Source file *equal_test.gadt*:

```
datatype Ty : type
datatype List : type
datacons Nil : ∀a. List a
datacons TInt : Ty Int
datacons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
datacons TList : ∀a. Ty a ⟶ Ty (List a)
external let zero : Int = "0"
external let eq_int : Int → Int → Bool = "(=)"
external let b_and : Bool → Bool → Bool = "(&&)"
external let b_not : Bool → Bool = "not"
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
test b_not (equal (TInt, TList TInt) zero Nil)
```

Value items from *equal_test.gadti.target*:

```
val equal : ∀a, b. (Ty a, Ty b) → a → b → Bool
```

C.5. LISTS WITH LENGTH

C.5.1. Function `head`

Source file *list_head.gadt*:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons LCons : ∀n, a [0⩽n]. a * List(a, n) ⟶ List(a, n+1)

let head = function
  | LCons (x, _) -> x
  | LNil -> assert false
```

Value items from *list_ head.gadti.target*:

```
val head : ∀n, a[1 ⩽ n]. List (a, n) → a
```

C.5.2. Function append

This example is not featured in the *examples* directory. The natural solution would be to propagate one of the arguments when the other list is empty: `function LNil -> (fun l -> l) |...` This currently leads to the problem of insufficient information about `l`. We can expand all cases of both arguments:

```
datatype Elem
datatype List : num
datacons LNil : List 0
datacons LCons : ∀n [0⩽n]. Elem * List n ⟶ List (n+1)

let rec append =
  function LNil -> (function LNil -> LNil | LCons (_,_) as l -> l)
    | LCons (x, xs) ->
      (function LNil -> LCons (x, append xs LNil)
        | LCons (_,_) as l -> LCons (x, append xs l))
```

Inferred type:

```
val append : ∀n, k. List (a, k) → List (a, n) → List (n + k)
```

We can also use assertions:

```
let rec append =
  function
    | LNil ->
      (function l when (length l + 1) <= 0 -> assert false | l -> l)
    | LCons (x, xs) ->
      (function l when (length l + 1) <= 0 -> assert false
      | l -> LCons (x, append xs l))
```

Inferred type:

```
val append : ∀n, k[0⩽n ∧ 0⩽n + k]. List(a, k) → List(a, n) → List(n + k)
```

C.5.3. Function `flatten_pairs`

Source file *flatten_pairs.gadt*:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons LCons : ∀n, a [0⩽n]. a * List(a, n) ⟶ List(a, n+1)

let rec flatten_pairs =
  function LNil -> LNil
    | LCons ((x, y), l) ->
      LCons (x, LCons (y, flatten_pairs l))
```

Value items from *flatten_pairs.gadti.target*:

```
val flatten_pairs : ∀n, a. List ((a, a), n) → List (a, 2 n)
```

C.5.4. Function `filter`

Source file *filter.gadt*:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
datacons 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
```

Value items from *filter.gadti.target*:

```
val filter :
   ∀n, a.
   (a → Bool) → List (a, n) → ∃k[0 ⩽ k ∧ k ⩽ n].List (a, k)
```

C.5.5. Function `zip`

Source file *zip.gadt*:

```
datatype List : type * num
datacons LNil : ∀a. List(a, 0)
```

```
datacons LCons : ∀n, a [0⩽n]. a * List(a, n) ⟶ List(a, n+1)

let rec zip =
  efunction
    | LNil, LNil -> LNil
    | LNil, LCons (_, _) -> LNil
    | LCons (_, _), LNil -> LNil
    | LCons (x, xs), LCons (y, ys) ->
      let zs = zip (xs, ys) in
      LCons ((x, y), zs)
```

The inferred type is:

```
val ∀n, k, a, b. (List (a, n), List (b, k)) →
                  ∃i[i=min (k, n)].List ((a, b), i)
```

C.6. Binary Numbers

C.6.1. Function plus

Source file *binary_plus.gadt*:

```
datatype Binary : num
datatype Carry : num

datacons Zero : Binary 0
datacons PZero : ∀n [0⩽n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0⩽n]. Binary n ⟶ Binary(2 n + 1)

datacons CZero : Carry 0
datacons COne : Carry 1

let rec plus =
  function CZero ->
    (function
      | Zero ->
        (function Zero -> Zero
          | PZero _ as b -> b
          | POne _ as b -> b)
      | PZero a1 as a ->
        (function Zero -> a
          | PZero b1 -> PZero (plus CZero a1 b1)
          | POne b1 -> POne (plus CZero a1 b1))
      | POne a1 as a ->
        (function Zero -> a
```

```
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1)))
    | COne ->
    (function Zero ->
        (function Zero -> POne(Zero)
          | PZero b1 -> POne b1
          | POne b1 -> PZero (plus COne Zero b1))
      | PZero a1 ->
        (function Zero -> POne a1
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1))
      | POne a1 ->
        (function Zero -> PZero (plus COne a1 Zero)
          | PZero b1 -> PZero (plus COne a1 b1)
          | POne b1 -> POne (plus COne a1 b1)))
```

Value items from *binary_plus.gadti.target*:

```
val plus :
   ∀i, k, n. Carry i → Binary k → Binary n → Binary (n + k + i)
```

C.6.2. Function `increment`

This example is not featured in the *examples* directory.

```
datatype Binary : num

datacons Zero : Binary 0
datacons PZero : ∀n [0⩽n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0⩽n]. Binary n ⟶ Binary(2 n + 1)

let rec increment =
  function Zero -> POne Zero
    | PZero a1 -> POne a1
    | POne a1 -> PZero (increment a1)
```

Inferred type:

```
val increment : ∀n. Binary n → Binary (n + 1)
```

C.6.3. Function `bitwise_or`

For brevity, the function is named `ub` rather than `bitwise_or` in the code example. Source file *binary_upper_bound.gadt*:

```
datatype Binary : num
datacons Zero : Binary 0
```

```
datacons PZero : ∀n [0⩽n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0⩽n]. Binary n ⟶ Binary(2 n + 1)

let rec ub = efunction
  | Zero ->
      (efunction Zero -> Zero
        | PZero b1 as b -> b
        | POne b1 as b -> b)
  | PZero a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          PZero r
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
  | POne a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          POne r
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
```

Value items from

```
val plus :
   ∀i, k, n. Carry i → Binary k → Binary n → Binary (n + k + i)
```

C.7. AVL Trees

Source file *avl_ tree.gadt*:

```
(** We follow the AVL tree algorithm from OCaml Standard Library, where
    the branches of a node are allowed to differ in height by at most 2. *)

datatype Avl : type * num
datacons Empty : ∀a. Avl (a, 0)
datacons Node :
  ∀a,k,m,n [k=max(m,n) ∧ 0⩽m ∧ 0⩽n ∧ n⩽m+2 ∧ m⩽n+2].
     Avl (a, m) * a * Avl (a, n) * Num (k+1) ⟶ Avl (a, k+1)

datatype LinOrder
datacons LT : LinOrder
```

```
datacons EQ : LinOrder
datacons GT : LinOrder
external let compare : ∀a. a → a → LinOrder =
  "fun x y -> let c=Pervasives.compare x y in
              if c<0 then LT else if c=0 then EQ else GT"

let height = function
  | Empty -> 0
  | Node (_, _, _, k) -> k

let create = fun l x r ->
  eif height l <= height r then Node (l, x, r, height r + 1)
  else Node (l, x, r, height l + 1)

let singleton x = Node (Empty, x, Empty, 1)

let rotr = fun l x r ->
    ematch l with
    | Empty -> assert false
    | Node (ll, lx, lr, _) ->
      eif height lr <= height ll then
        let r' = create lr x r in
        create ll lx r'
      else
        ematch lr with
        | Empty -> assert false
        | Node (lrl, lrx, lrr, _) ->
          let l' = create ll lx lrl in
          let r' = create lrr x r in
          create l' lrx r'

let rotl = fun l x r ->
    ematch r with
    | Empty -> assert false
    | Node (rl, rx, rr, _) ->
      eif height rl <= height rr then
        let l' = create l x rl in
        create l' rx rr
      else
        ematch rl with
        | Empty -> assert false
        | Node (rll, rlx, rlr, _) ->
          let l' = create l x rll in
          let r' = create rlr rx rr in
          create l' rlx r'
```

```
let rec add x = efunction
  | Empty -> Node (Empty, x, Empty, 1)
  | Node (l, y, r, h) ->
    ematch compare x y, height l, height r with
    | EQ, _, _ -> Node (l, x, r, h)
    | LT, hl, hr ->
      let l' = add x l in
      eif height l' <= hr+2 then create l' y r else rotr l' y r
    | GT, hl, hr ->
      let r' = add x r in
      eif height r' <= hl+2 then create l y r' else rotl l y r'

let rec mem x = function
  | Empty -> False
  | Node (l, y, r, _) ->
    match compare x y with
    | LT -> mem x l
    | EQ -> True
    | GT -> mem x r

let rec min_binding = function
  | Empty -> assert false
  | Node (Empty, x, r, _) -> x
  | Node ((Node (_,_,_,_) as l), x, r, _) -> min_binding l

let rec remove_min_binding = efunction
  | Empty -> assert false
  | Node (Empty, x, r, _) -> r
  | Node ((Node (_,_,_,_) as l), x, r, _) ->
    let l' = remove_min_binding l in
    eif height r <= height l' + 2 then create l' x r
    else rotl l' x r

let merge = efunction
  | Empty, Empty -> Empty
  | Empty, (Node (_,_,_,_) as t) -> t
  | (Node (_,_,_,_) as t), Empty -> t
  | (Node (_,_,_,_) as t1), (Node (_,_,_,_) as t2) ->
    let x = min_binding t2 in
    let t2' = remove_min_binding t2 in
    eif height t1 <= height t2' + 2 then create t1 x t2'
    else rotr t1 x t2'

let rec remove = fun x -> efunction
```

```
  | Empty -> Empty
  | Node (l, y, r, h) ->
    ematch compare x y with
    | EQ -> merge (l, r)
    | LT ->
      let l' = remove x l in
      eif height r <= height l' + 2 then create l' y r
      else rotl l' y r
    | GT ->
      let r' = remove x r in
      eif height l <= height r' + 2 then create l y r'
      else rotr l y r'
```

Value items from *avl_tree.gadti.target*:

```
val height : ∀n, a. Avl (a, n) → Num n

val create :
   ∀k, n, a[0 ⩽ n ∧ 0 ⩽ k ∧ n ⩽ k + 2 ∧ k ⩽ n + 2].
   Avl (a, k) → a → Avl (a, n) → ∃i[i=max (k + 1, n + 1)].Avl (a, i)

val singleton : ∀a. a → Avl (a, 1)

val rotr :
   ∀k, n, a[0 ⩽ n ∧ n + 2 ⩽ k ∧ k ⩽ n + 3].
   Avl (a, k) → a → Avl (a, n) → ∃n[k ⩽ n ∧
     n ⩽ k + 1].Avl (a, n)

val rotl :
   ∀k, n, a[0 ⩽ k ∧ n ⩽ k + 3 ∧ k + 2 ⩽ n].
   Avl (a, k) → a → Avl (a, n) → ∃k[k ⩽ n + 1 ∧
     n ⩽ k].Avl (a, k)

val add :
   ∀n, a.
   a → Avl (a, n) → ∃k[1 ⩽ k ∧ n ⩽ k ∧ k ⩽ n + 1].Avl (a, k)

val mem : ∀n, a. a → Avl (a, n) → Bool

val min_binding : ∀n, a[1 ⩽ n]. Avl (a, n) → a

val remove_min_binding :
   ∀n, a[1 ⩽ n].
   Avl (a, n) → ∃k[n ⩽ k + 1 ∧ k ⩽ n ∧ k + 2 ⩽ 2 n].Avl (a, k)
```

```
val merge :
   ∀k, n, a[n ⩽ k + 2 ∧ k ⩽ n + 2].
   (Avl (a, n), Avl (a, k)) → ∃i[n ⩽ i ∧ k ⩽ i ∧ i ⩽ n + k ∧
     i⩽max (k + 1, n + 1)].Avl (a, i)

val remove :
   ∀n, a.
   a → Avl (a, n) → ∃k[n ⩽ k + 1 ∧ 0 ⩽ k ∧ k ⩽ n].Avl (a, k)
```

C.8. Arrays and Matrices

For clarity, we present the array and matrix operations prelude separately here. Arrays are polymorphic, matrices only store floating point numbers.

```
datatype Array : type * num
external let array_make :
  ∀n, a [0⩽n]. Num n → a → Array (a, n) = "fun a b -> Array.make a b"
external let array_get :
  ∀n, k, a [0⩽k ∧ k+1⩽n]. Array (a, n) → Num k → a = "fun a b ->
Array.get a b"
external let array_set :
  ∀n, k, a [0⩽k ∧ k+1⩽n]. Array (a, n) → Num k → a → () =
  "fun a b c -> Array.set a b c"
external let array_length :
  ∀n, a. Array (a, n) → Num n = "fun a -> Array.length a"

external type Matrix : num * num =
  "(int, Bigarray.int_elt, Bigarray.c_layout) Bigarray.Array2.t"
external let matrix_make :
  ∀n, k [0⩽n ∧ 0⩽k]. Num n → Num k → Matrix (n, k) =
  "fun a b -> Bigarray.Array2.create Bigarray.int Bigarray.c_layout a b"
external let matrix_get :
  ∀n, k, i, j [0⩽i ∧ i+1⩽n ∧ 0⩽j ∧ j+1⩽k].
   Matrix (n, k) → Num i → Num j → Int = "Bigarray.Array2.get"
external let matrix_set :
  ∀n, k, i, j [0⩽i ∧ i+1⩽n ∧ 0⩽j ∧ j+1⩽k].
   Matrix (n, k) → Num i → Num j → Int → () = "Bigarray.Array2.set"
external let matrix_dim1 :
  ∀n, k [0⩽n ∧ 0⩽k]. Matrix (n, k) → Num n = "Bigarray.Array2.dim1"
external let matrix_dim2 :
  ∀n, k [0⩽n ∧ 0⩽k]. Matrix (n, k) → Num k = "Bigarray.Array2.dim2"
```

C.8.1. Program `dotprod`

Source file *liquid_ dotprod.gadt*:

```
external let add : Int → Int → Int = "fun n k -> n + k"
external let prod : Int → Int → Int = "fun n k -> n * k"
external let int0 : Int = "0"

let dotprod v1 v2 =
  let rec loop n sum i =
    if n <= i then sum else
        loop n (add (prod (array_get v1 i) (array_get v2 i)) sum)
          (i + 1) in
  loop (array_length v1) int0 0
```

Value items from *liquid_ dotprod.gadti.target*:

```
val dotprod : ∀k, n[k ⩽ n]. Array (Int, k) → Array (Int, n) → Int
```

C.8.2. Program bcopy

Source file *liquid_ bcopy.gadt*:

```
let rec bcopy_aux src des = function
  | i, m when m <= i -> ()
  | i, m when i+1 <= m ->
      let n = array_get src i in
      array_set des i n;
      let j = i + 1 in
      bcopy_aux src des (j, m)

let bcopy src des =
  let sz = array_length src in
  bcopy_aux src des (0, sz)
```

Value items from *liquid_ bcopy.gadti.target*:

```
val bcopy_aux :
   ∀i, j, k, n, a[0 ⩽ n ∧ k ⩽ i ∧ k ⩽ j].
   Array (a, j) → Array (a, i) → (Num n, Num k) → ()

val bcopy : ∀k, n, a[k ⩽ n]. Array (a, k) → Array (a, n) → ()
```

C.8.3. Program bsearch

Source file *liquid_ bsearch_ harder.gadt*:

```
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder
```

```
external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"

external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"

datatype Answer : type
datacons NotFound : ∀a. Answer a
datacons Found : ∀a. a ⟶ Answer a

let bsearch key vec =
  let rec look lo hi =
    if lo <= hi then
        let m = div2 (hi + lo) in
        let x = array_get vec m in
        match compare key x with
          | LE -> look lo (m + (-1))
          | GT -> look (m + 1) hi
          | EQ -> if equal key x then Found x else NotFound
    else NotFound in
  look 0 (array_length vec + (-1))
```

Value items from *liquid_bsearch_harder.gadti.target*:

```
val bsearch : ∀n, a[0 ⩽ n]. a → Array (a, n) → Answer a
```

C.8.4. Function `bsearch2`

Source file *liquid_bsearch2_harder3.gadt*:

```
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder

external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"

external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"

let bsearch key = efunction vec ->
  let rec look key vec lo hi =
```

```
    assert num -1 <= hi;
    eif lo <= hi then
        let m = div2 (hi + lo) in
        let x = array_get vec m in
        ematch compare key x with
          | LE -> look key vec lo (m + (-1))
          | GT -> look key vec (m + 1) hi
          | EQ -> eif equal key x then m else -1
    else -1 in
  look key vec 0 (array_length vec + (-1))
```

Value items from *liquid_bsearch2_harder3.gadti.target*:

```
val bsearch :
   ∀n, a[0 ⩽ n].
   a → Array (a, n) → ∃k[0 ⩽ k + 1 ∧ k + 1 ⩽ n].Num k
```

Exported OCaml source *liquid_bsearch2_harder3.ml.target*:

```
type num = int
(** type _ array = built-in *)
let array_make : (*0 ⩽ n*) ((* n *) num -> 'a -> ('a (* n *)) array) =
  fun a b -> Array.make a b
let array_get :
  (*0 ⩽ k ∧ k + 1 ⩽ n*) (('a (* n *)) array -> (* k *) num -> 'a) =
  fun a b -> Array.get a b
let array_set :
  (*0 ⩽ k ∧ k + 1 ⩽ n*)
  (('a (* n *)) array -> (* k *) num -> 'a -> unit) =
  fun a b c -> Array.set a b c
let array_length : (*0 ⩽ n*) (('a (* n *)) array -> (* n *) num) =
  fun a -> Array.length a
type linOrder =
  | LE : linOrder
  | GT : linOrder
  | EQ : linOrder
let compare : ('a -> 'a -> linOrder) =
  fun a b -> let c = Pervasives.compare a b in
             if c < 0 then LE else if c > 0 then GT else EQ
let equal : ('a -> 'a -> bool) = fun a b -> a = b
let div2 : ((* 2 n *) num -> (* n *) num) = fun x -> x / 2
type ex4 =
  | Ex4 : (*∀'k, 'n[0 ⩽ k + 1 ∧ k + 1 ⩽ n].*)(* k *) num ->
    (* n *) ex4
type ex3 =
  | Ex3 : (*∀'k, 'n[0 ⩽ k + 1 ∧ k ⩽ n].*)(* k *) num -> (* n *) ex3
```

```
let bsearch
  : type (*n*) a (*0 ⩽ n*). (a -> (a (* n *)) array -> (* n *) ex4) =
  (fun key vec ->
    let Ex3 xcase =
    let rec look :
    type (*i*) (*j*) (*k*) b (*0 ⩽ k + 1 ∧ 0 ⩽ i ∧
    k + 1 ⩽ j*). (b -> (b (* j *)) array -> (* i *) num -> (* k *) num ->
                    (* k *) ex3)
    =
    (fun key vec lo hi ->
      (if lo <= hi then
      let m = div2 (hi + lo) in
      let x = array_get vec m in
      (((match (compare key x: linOrder) with
      LE -> let Ex3 xcase = look key vec lo (m + -1) in Ex3 xcase
        | GT -> let Ex3 xcase = look key vec (m + 1) hi in Ex3 xcase
        | EQ ->
            (if equal key x then let xcase = m in Ex3 xcase else
            let xcase = -1 in Ex3 xcase))) :
      (* k *) ex3) else let xcase = -1 in Ex3 xcase)) in
    look key vec 0 (array_length vec + -1) in Ex4 xcase)
```

C.8.5. Program queen

Source file *liquid_ queen.gadt*:

```
external let n2i : ∀n. Num n → Int = "fun i -> i"
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"

external let print : String → () = "print_string"
external let string_make : Int → String → String =
  "fun n s -> String.make n s.[0]"
external let abs : Int → Int = "fun i -> if i < 0 then ~-i else i"
external let minus : Int → Int → Int = "(-)"
external let plus : Int → Int → Int = "(+)"

let queens size =
  let board = array_make size (n2i 0) in
  let print_row pos =
    print (string_make (minus pos (n2i 1)) "."); print "Q";
    print (string_make (minus (n2i size) pos) ".") in
  let print_queens () =
    let rec aux row =
      if size <= row then ()
```

```
      else (print_row (array_get board row); aux (row + 1)) in
    aux 0 in
  let rec solved j =
    let q2j = array_get board j in
    let rec aux i =
      if i + 1 <= j then
        let q2i = array_get board i in
        let qdiff = abs (minus q2j q2i) in
        if equal q2i q2j then False
        else if equal qdiff (n2i (j - i)) then False
        else aux (i + 1)
      else True in
    aux 0 in
  let rec loop row =
    let next = plus (array_get board row) (n2i 1) in
    if leq (n2i (size + 1)) next then (
      array_set board row (n2i 0);
      if row <= 0 then () else loop (row - 1))
    else (
      array_set board row next;
      if solved row then
        if size <= row + 1 then (print_queens (); loop row)
        else loop (row + 1)
      else loop row) in
  loop 0
```

Value items from *liquid_ queen.gadti.target*:

```
val queens : ∀n[1 ⩽ n]. Num n → ()
```

C.8.6. Function `swap_interval`

Source file *liquid_ vecswap.gadt*:

```
let swap_interval arr start middle n =
  let rec item i = array_get arr i in
  let rec swap i j =
    let tmpj = item j in
    let tmpi = item i in
    array_set arr i tmpj; array_set arr j tmpi in
  let rec vecswap i j n =
    if n <= 0 then () else (
      swap i j; vecswap (i + 1) (j + 1) (n - 1)) in
  vecswap start middle n
```

Value items from *liquid_ vecswap.gadti.target*:

```
val swap_interval :
   ∀i, j, k, n, a[1 ⩽ j ∧ 0 ⩽ n ∧ 0 ⩽ k ∧ i + k ⩽ j ∧
   i + n ⩽ j]. Array (a, j) → Num n → Num k → Num i → ()
```

C.8.7. Program `isort`

Source file *liquid_ isort_ harder.gadt*:

```
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder

external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"

let rec insertSort arr start n =
  let rec item i = array_get arr i in
  let rec swap i j =
    let tmpj = item j in
    let tmpi = item i in
    array_set arr i tmpj; array_set arr j tmpi in
  let rec outer i =
    if start + n <= i then ()
    else
      let rec inner j =
        if j <= start then outer (i + 1)
        else if equal (compare (item j) (item (j - 1))) LE
        then (swap j (j - 1); inner (j - 1))
        else outer (i + 1) in
      inner i in
  outer (start + 1)
```

Value items from *liquid_ isort_ harder.gadti.target*:

```
val insertSort :
   ∀i, k, n, a[0 ⩽ k ∧ 1 ⩽ i ∧ k + n ⩽ i].
   Array (a, i) → Num k → Num n → ()
```

C.8.8. Program `tower`

Source file *liquid_ tower.gadt*:

```
external let n2i : ∀n. Num n → Int = "fun i -> i"
```

```
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"

external let print : String → () = "print_string"
external let string_make : Int → String → String =
  "fun n s -> String.make n s.[0]"
external let string_of_int : Int → String = "string_of_int"

external let abs : Int → Int = "fun i -> if i < 0 then ~-i else i"
external let minus : Int → Int → Int = "(-)"
external let plus : Int → Int → Int = "(+)"
external let int0 : Int = "0"

let play sz =
  let leftPost = array_make sz int0 in
  let middlePost = array_make sz int0 in
  let rightPost = array_make sz int0 in

  let initialize post =
    let rec init_rec i =
      if i + 1 <= sz - 1 then (
        array_set post i (n2i (i+1));
        init_rec (i+1))
      else () in
    init_rec 0 in

  let showpiece n =
    let rec r_rec i =
      if leq (plus i (n2i 2)) n then (
        print " "; r_rec (plus i (n2i 1)))
      else () in
    let rec r2_rec j =
      if leq (plus j (n2i 1)) (n2i sz)
      then (print "#"; r2_rec (plus j (n2i 1)))
      else () in
    r_rec (n2i 1);
    r2_rec (plus n (n2i 1)) in

  let showposts () =
    let rec show_rec i =
      if i + 1 <= sz - 1 then (
        showpiece (array_get leftPost i); print " ";
        showpiece (array_get middlePost i); print " ";
        showpiece (array_get rightPost i); print "\n";
        show_rec (i+1))
```

```
      else () in
    show_rec 0; print "\n" in

  initialize leftPost;
  let rec move n source s post p post' p' =
    if n <= 1 then (
      array_set post (p - 1) (array_get source s);
      array_set source s int0; showposts ())
    else (
      move (n - 1) source s post' p' post p;
      array_set post (p - 1) (array_get source (s + n - 1));
      array_set source (s + n - 1) int0;
      showposts ();
      move (n - 1) post' ((p' - n) + 1) post (p - 1) source (s + n)) in

  showposts ();
  move sz leftPost 0 rightPost sz middlePost sz
```

Value items from *liquid_ tower.gadti.target*:

```
val play : ∀n[1 ⩽ n]. Num n → ()
```

C.8.9. Program `matmult`

Source file *liquid_ matmult.gadt*:

```
external let n2i : ∀n. Num n → Int = "fun i -> i"
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"

external let abs : Int → Int = "fun i -> if i < 0 then ~-i else i"
external let minus : Int → Int → Int = "(-)"
external let plus : Int → Int → Int = "(+)"
external let mult : Int → Int → Int = "( * )"
external let int0 : Int = "0"

let fillar arr2 fill =
  let d1 = matrix_dim1 arr2 in
  let d2 = matrix_dim2 arr2 in
  let rec loop i =
    if i + 1 <= d1
    then
      let rec loopi j =
        if j + 1 <= d2 then (
          matrix_set arr2 i j (fill ());
          loopi (j + 1))
```

```
        else () in
      loopi 0
    else loop (i + 1) in
  loop 0

let matmul a b =
  let p = matrix_dim1 a in
  let q = matrix_dim2 a in
  let r = matrix_dim2 b in

  let cdata = matrix_make p r in
  let callback0 () = int0 in
  fillar cdata callback0;

  let rec loop1 i =
    if i + 1 <= p then (
      let rec loop2 j =
        if j + 1 <= r then (
          let rec loop3 k sum =
            if k + 1 <= q then (
              let sum_p =
                plus sum (mult (matrix_get a i k) (matrix_get b k j)) in
              loop3 (k + 1) sum_p)
            else sum in
          let l3 = loop3 0 int0 in
          matrix_set cdata i j l3;
          loop2 (j + 1))
        else () in
      loop2 0;
      loop1 (i + 1))
    else () in
  loop1 0;
  cdata
```

Value items from *liquid_ matmult.gadti.target*:

```
val fillar :
   ∀k, n[0 ⩽ n ∧ 0 ⩽ k]. Matrix (n, k) → (() → Int) → ()

val matmul :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ j ∧ j ⩽ i].
   Matrix (n, j) → Matrix (i, k) → Matrix (n, k)
```

C.8.10. Program `heapsort`

Source file *liquid_ heapsort.gadt*:

```
external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"

external let n2i : ∀n. Num n → Int = "fun i -> i"
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"
external let less : ∀a. a → a → Bool = "fun x y -> x < y"

external let print : String → () = "print_string"
external let string_make : Int → String → String =
  "fun n s -> String.make n s.[0]"
external let string_of_int : Int → String = "string_of_int"

external let abs : Int → Int = "fun i -> if i < 0 then ~-i else i"
external let minus : Int → Int → Int = "(-)"
external let plus : Int → Int → Int = "(+)"
external let int0 : Int = "0"

let rec heapify size data i =
  let left = 2 * i + 1 in
  let right = 2 * i + 2 in
  let largest1 =
    eif left + 1 <= size then
      eif less (array_get data i) (array_get data left)
      then left else i
    else i in
  let largest2 =
    eif right + 1 <= size then
      eif less (array_get data largest1) (array_get data right) then right
      else largest1
    else largest1 in
  if i + 1 <= largest2 then
    let temp = array_get data i in
    let temp2 = array_get data largest2 in
    array_set data i temp2;
    array_set data largest2 temp;
    heapify size data largest2
  else ()
(* We do not get the constraint [i + 1 <= size], because if [i] is larger,
   the last [else] branch is entered. *)

let buildheap size data =
  let rec loop i =
    if 0 <= i then (
      heapify size data i;
      loop (i - 1))
```

```
    else () in
  loop (div2 size - 1)

let heapsort maxx size data =
  buildheap size data;
  let rec loop i =
    if 1 <= i then
      let temp = array_get data i in
      array_set data i (array_get data 0);
      array_set data 0 temp;
      heapify i data 0;
      loop (i - 1)
    else () in
  loop (maxx - 1)

let print_array data i j =
  let rec loop k =
    if k + 1 <= j then (
      print (array_get data k);
      loop (k + 1))
    else () in
  loop i
```

Value items from *liquid_ heapsort.gadti.target*:

```
val heapify :
   ∀i, k, n, a[0 ⩽ n ∧ i ⩽ k].
   Num i → Array (a, k) → Num n → ()

val buildheap : ∀k, n, a[k ⩽ n]. Num k → Array (a, n) → ()

val heapsort :
   ∀i, k, n, a[k ⩽ n ∧ i ⩽ k].
   Num i → Num k → Array (a, n) → ()

val print_array :
   ∀i, k, n[k ⩽ i ∧ 0 ⩽ n].
   Array (String, i) → Num n → Num k → ()
```

C.8.11. Program `simplex`

Source file *liquid_ simplex.gadt*:

```
external let n2f : ∀n. Num n → Float = "float_of_int"
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"
```

```
external let less : ∀a. a → a → Bool = "fun x y -> x < y"

external let minus : Float → Float → Float = "(-.)"
external let plus : Float → Float → Float = "(+.)"
external let mult : Float → Float → Float = "( *. )"
external let div : Float → Float → Float = "( /. )"
external let fl0 : Float = "0.0"

(* step 1 *)

let rec is_neg_aux a j =
  if j + 2 <= matrix_dim2 a then
    if less (matrix_get a 0 j) fl0 then True
    else is_neg_aux a (j+1)
  else False

let is_neg a = is_neg_aux a 1

(* step 2 *)

let rec unb1 a i j =
  let rec unb2 a i j =
    if i + 1 <= matrix_dim1 a then
      if less (matrix_get a i j) fl0
      then unb2 a (i+1) j
      else unb1 a 0 (j+1)
    else True in

  if j + 2 <= matrix_dim2 a then
    if less (matrix_get a 0 j) fl0
    then unb2 a (i+1) j
    else unb1 a 0 (j+1)
  else False

(* step 6 *)

let rec norm_aux a i c j =
  if j + 1 <= matrix_dim2 a then (
    matrix_set a i j (div (matrix_get a i j) c);
    norm_aux a i c (j+1))
  else ()

let rec norm a i j =
  let c = matrix_get a i j in
  norm_aux a i c 1
```

```
let rec row_op_aux1 a i i' c j =
  if j + 1 <= matrix_dim2 a then
    matrix_set a i' j
      (minus (matrix_get a i' j)
        (mult (matrix_get a i j) c));
    row_op_aux1 a i i' c (j+1)
  else ()

let rec row_op_aux2 a i i' j =
  row_op_aux1 a i i' (matrix_get a i' j) 1

let rec row_op_aux3 a i j i' =
  if i' + 1 <= matrix_dim1 a then
    if i' <= i && i <= i' then (
      row_op_aux2 a i i' j;
      row_op_aux3 a i j (i'+1))
    else row_op_aux3 a i j (i'+1)
  else ()

let rec row_op a i j =
  norm a i j;
  row_op_aux3 a i j 0

let rec simplex a =

  (* step 3 *)

  let rec enter_var j c j' =
    eif j' + 2 <= matrix_dim2 a then
      let c' = matrix_get a 0 j' in
      eif less c' c then enter_var j' c' (j'+1)
      else enter_var j c (j'+1)
    else j in

  (* step 4 *)

  let rec depart_var j i r i' =
    eif i' + 1 <= matrix_dim1 a then
      let c' = matrix_get a i' j in
      eif less fl0 c' then
        let r' = div (matrix_get a i' (matrix_dim2 a + (-1))) c' in
        eif less r' r then depart_var j i' r' (i'+1)
        else depart_var j i r (i'+1)
      else depart_var j i r (i'+1)
```

```
    else i in

  (* step 5 *)

  let rec init_ratio j i =
    eif i + 1 <= matrix_dim1 a then
      let c = matrix_get a i j in
      eif less fl0 c then i, div (matrix_get a i (matrix_dim2 a + (-1))) c
      else init_ratio j (i+1)
    else runtime_failure "init_ratio: no variable found" in

  (* step 7 *)

  if is_neg a then
    if unb1 a 0 1 then () (* assert false *)
    else
      let j = enter_var 1 (matrix_get a 0 1) 2 in
      let i, r = init_ratio j 1 in
      let i = depart_var j i r (i+1) in
      row_op a i j;
      simplex a
  else ()
```

The file *liquid_simplex_harder.gadt* has the above steps in order as part of a single toplevel definition. Value items from *liquid_simplex.gadti.target*:

```
val is_neg_aux :
   ∀i, k, n[0 ⩽ n ∧ 1 ⩽ k ∧ 0 ⩽ i].
   Matrix (k, i) → Num n → Bool

val is_neg : ∀k, n[1 ⩽ n ∧ 0 ⩽ k]. Matrix (n, k) → Bool

val unb1 :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k + 1 ∧ 1 ⩽ i ∧ 0 ⩽ j].
   Matrix (i, j) → Num k → Num n → Bool

val norm_aux :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧ 0 ⩽ j].
   Matrix (i, j) → Num k → Float → Num n → ()

val norm :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧ n + 1 ⩽ j].
   Matrix (i, j) → Num k → Num n → ()

val row_op_aux1 :
   ∀i, j, k, m, n[0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ i ∧ i + 1 ⩽ j ∧
```

```
   k + 1 ⩽ j ∧ 0 ⩽ m].
   Matrix (j, m) → Num i → Num k → Float → Num n → ()

val row_op_aux2 :
   ∀i, j, k, m, n[0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ i ∧ i + 1 ⩽ j ∧
   k + 1 ⩽ j ∧ n + 1 ⩽ m].
   Matrix (j, m) → Num i → Num k → Num n → ()

val row_op_aux3 :
   ∀i, j, k, m, n[0 ⩽ k ∧ 0 ⩽ i ∧ 0 ⩽ j ∧ k + 1 ⩽ m].
   Matrix (j, m) → Num i → Num k → Num n → ()

val row_op :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧ n + 1 ⩽ j].
   Matrix (i, j) → Num k → Num n → ()

val simplex : ∀k, n[1 ⩽ n ∧ 2 ⩽ k]. Matrix (n, k) → ()
```

Exported OCaml source *liquid_simplex.ml.target*:

```
type num = int
type matrix =
  (float, Bigarray.float64_elt, Bigarray.c_layout) Bigarray.Array2.t
let matrix_make :
  (*0 ⩽ n ∧ 0 ⩽ k*) ((* n *) num -> (* k *) num -> (* n, k *) matrix) =
  fun a b -> Bigarray.Array2.create Bigarray.float64 Bigarray.c_layout a b
let matrix_get :
  (*0 ⩽ i ∧ i + 1 ⩽ n ∧ 0 ⩽ j ∧ j + 1 ⩽ k*)
  ((* n, k *) matrix -> (* i *) num -> (* j *) num -> float) =
  Bigarray.Array2.get
let matrix_set :
  (*0 ⩽ i ∧ i + 1 ⩽ n ∧ 0 ⩽ j ∧ j + 1 ⩽ k*)
  ((* n, k *) matrix -> (* i *) num -> (* j *) num -> float -> unit) =
  Bigarray.Array2.set
let matrix_dim1 :
  (*0 ⩽ n ∧ 0 ⩽ k*) ((* n, k *) matrix -> (* n *) num) =
  Bigarray.Array2.dim1
let matrix_dim2 :
  (*0 ⩽ n ∧ 0 ⩽ k*) ((* n, k *) matrix -> (* k *) num) =
  Bigarray.Array2.dim2
let n2f : ((* n *) num -> float) = float_of_int
let equal : ('a -> 'a -> bool) = fun x y -> x = y
let leq : ('a -> 'a -> bool) = fun x y -> x <= y
let less : ('a -> 'a -> bool) = fun x y -> x < y
let minus : (float -> float -> float) = (-.)
let plus : (float -> float -> float) = (+.)
```

```
let mult : (float -> float -> float) = ( *. )
let div : (float -> float -> float) = ( /. )
let fl0 : float = 0.0
let rec is_neg_aux :
  (*type i k n [0 ⩽ n ∧ 1 ⩽ k ∧
    0 ⩽ i].*) ((* k, i *) matrix -> (* n *) num -> bool) =
  (fun a j ->
    (if j + 2 <= matrix_dim2 a then
    (if less (matrix_get a 0 j) fl0 then true else is_neg_aux a (j + 1))
else
    false))
let is_neg
  : (*type k n [1 ⩽ n ∧ 0 ⩽ k].*) ((* n, k *) matrix -> bool) =
  (fun a -> is_neg_aux a 1)
let rec unb1 :
  (*type i j k n [0 ⩽ n ∧ 0 ⩽ k + 1 ∧ 1 ⩽ i ∧
    0 ⩽ j].*) ((* i, j *) matrix -> (* k *) num -> (* n *) num -> bool) =
  (fun a i j ->
    let rec unb2 :
    (*type k1 l m n1 [0 ⩽ m ∧ 0 ⩽ l ∧ 0 ⩽ n1 ∧
      m + 1 ⩽ k1].*) ((* n1, k1 *) matrix -> (* l *) num -> (* m *) num ->
                          bool)
    =
    (fun a i j ->
      (if i + 1 <= matrix_dim1 a then
      (if less (matrix_get a i j) fl0 then unb2 a (i + 1) j else
      unb1 a 0 (j + 1)) else true)) in
    (if j + 2 <= matrix_dim2 a then
    (if less (matrix_get a 0 j) fl0 then unb2 a (i + 1) j else
    unb1 a 0 (j + 1)) else false))
let rec norm_aux :
  (*type i j k n [0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧
    0 ⩽ j].*) ((* i, j *) matrix -> (* k *) num -> float -> (* n *) num ->
                  unit) =
  (fun a i c j ->
    (if j + 1 <= matrix_dim2 a then
    (matrix_set a i j (div (matrix_get a i j) c) ; norm_aux a i c (j + 1))
    else ()))
let rec norm :
  (*type i j k n [0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧
    n + 1 ⩽ j].*) ((* i, j *) matrix -> (* k *) num -> (* n *) num -> unit)
=
  (fun a i j -> let c = matrix_get a i j in norm_aux a i c 1)
let rec row_op_aux1 :
  (*type i j k m n [0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ i ∧ i + 1 ⩽ j ∧
```

```
    k + 1 ⩽ j ∧
    0 ⩽ m].*) ((* j, m *) matrix -> (* i *) num -> (* k *) num -> float ->
                  (* n *) num -> unit) =
  (fun a i i' c j ->
    (if j + 1 <= matrix_dim2 a then
    (matrix_set a i' j
       (minus (matrix_get a i' j) (mult (matrix_get a i j) c))
    ; row_op_aux1 a i i' c (j + 1)) else ()))
let rec row_op_aux2 :
  (*type i j k m n [0 ⩽ n ∧ 0 ⩽ k ∧ 0 ⩽ i ∧ i + 1 ⩽ j ∧
    k + 1 ⩽ j ∧
    n + 1 ⩽ m].*) ((* j, m *) matrix -> (* i *) num -> (* k *) num ->
                      (* n *) num -> unit) =
  (fun a i i' j -> row_op_aux1 a i i' (matrix_get a i' j) 1)
let rec row_op_aux3 :
  (*type i j k m n [0 ⩽ k ∧ 0 ⩽ i ∧ 0 ⩽ j ∧
    k + 1 ⩽ m].*) ((* j, m *) matrix -> (* i *) num -> (* k *) num ->
                      (* n *) num -> unit) =
  (fun a i j i' ->
    (if i' + 1 <= matrix_dim1 a then
    (if i <= i'&& i' <= i then
    (row_op_aux2 a i i' j ; row_op_aux3 a i j (i' + 1)) else
    row_op_aux3 a i j (i' + 1)) else ()))
let rec row_op :
  (*type i j k n [0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧
    n + 1 ⩽ j].*) ((* i, j *) matrix -> (* k *) num -> (* n *) num -> unit)
=
  (fun a i j -> (norm a i j ; row_op_aux3 a i j 0))
type ex7 =
  | Ex7 : (*∀'i, 'k, 'n[k ⩽ i ∧ i + 1 ⩽ n].*)((* i *) num * float) ->
    (* n, k *) ex7
type ex5 =
  | Ex5 : (*∀'i, 'k, 'n[0 ⩽ k ∧ k⩽max (i, n + -1)].*)(* k *) num ->
    (* n, i *) ex5
type ex2 =
  | Ex2 : (*∀'i, 'k, 'n[0 ⩽ k ∧ k⩽max (i, n + -1)].*)(* k *) num ->
    (* n, i *) ex2
let rec simplex :
  (*type k n [1 ⩽ n ∧ 2 ⩽ k].*) ((* n, k *) matrix -> unit) =
  (fun a ->
    let rec enter_var :
    (*type i j [0 ⩽ j ∧
      0 ⩽ i].*) ((* i *) num -> float -> (* j *) num -> (* k, i *) ex2)
    =
    (fun j c j' ->
```

```
    (if j' + 2 <= matrix_dim2 a then
    let c' = matrix_get a 0 j' in
    (if less c' c then
    let Ex2 xcase = enter_var j' c' (j' + 1) in Ex2 xcase else
    let Ex2 xcase = enter_var j c (j' + 1) in Ex2 xcase) else
    let xcase = j in Ex2 xcase)) in
  let rec depart_var :
  (*type l m n1 [0 ⩽ l ∧ 0 ⩽ n1 ∧ n1 + 1 ⩽ k ∧
    0 ⩽ m].*) ((* n1 *) num -> (* m *) num -> float -> (* l *) num ->
                  (* n, m *) ex5)
  =
  (fun j i r i' ->
    (if i' + 1 <= matrix_dim1 a then
    let c' = matrix_get a i' j in
    (if less fl0 c' then
    let r' = div (matrix_get a i' (matrix_dim2 a + -1)) c' in
    (if less r' r then
    let Ex5 xcase = depart_var j i' r' (i' + 1) in Ex5 xcase else
    let Ex5 xcase = depart_var j i r (i' + 1) in Ex5 xcase) else
    let Ex5 xcase = depart_var j i r (i' + 1) in Ex5 xcase) else
    let xcase = i in Ex5 xcase)) in
  let rec init_ratio :
  (*type i1 k1 [0 ⩽ k1 ∧ 0 ⩽ i1 ∧
    i1 + 1 ⩽ k].*) ((* i1 *) num -> (* k1 *) num -> (* n, k1 *) ex7)
  =
  (fun j i ->
    (if i + 1 <= matrix_dim1 a then
      let c = matrix_get a i j in
      (if less fl0 c then
      let xcase = (i, div (matrix_get a i (matrix_dim2 a + -1)) c) in
      Ex7 xcase else let Ex7 xcase = init_ratio j (i + 1) in Ex7 xcase)
    else
      (failwith "init_ratio: no variable found"))) in
  (if is_neg a then
  (if unb1 a 0 1 then () else
    let Ex2 j = enter_var 1 (matrix_get a 0 1) 2 in
    let Ex7 (i, r) = init_ratio j 1 in
    let Ex5 i = depart_var j i r (i + 1) in (row_op a i j ; simplex a))
  else
    ()))
```

C.8.12. Program gauss

Source file *liquid_ gauss.gadt*:

```
external let n2f : ∀n. Num n → Float = "float_of_int"
```

```
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"
external let less : ∀a. a → a → Bool = "fun x y -> x < y"

external let minus : Float → Float → Float = "(-.)"
external let plus : Float → Float → Float = "(+.)"
external let mult : Float → Float → Float = "( *. )"
external let div : Float → Float → Float = "( /. )"
external let fabs : Float → Float = "abs_float"
external let fl0 : Float = "0.0"
external let fl1 : Float = "1.0"

let getRow data i =
  let stride = matrix_dim2 data in
  let rowData = array_make stride fl0 in
  let rec extract j =
    if j + 1 <= stride then (
      array_set rowData j (matrix_get data i j);
      (* lukstafi: the call below missing in the original source? *)
      extract (j + 1))
    else () in
  extract 0;
  rowData

let putRow data i row =
  let stride = array_length row in
  let rec put j =
    if j + 1 <= stride then (
      matrix_set data i j (array_get row j);
      (* lukstafi: the call below missing in the original source? *)
      put (j + 1))
    else () in
  put 0

let rowSwap data i j =
  let temp = getRow data i in
  putRow data i (getRow data j);
  putRow data j temp

let norm r n i =
  let x = array_get r i in
  array_set r i fl1;
  let rec loop k =
    if k + 1 <= n then (
      array_set r k (div (array_get r k) x);
```

```
      loop (k+1))
    else () in
  loop (i+1)

let rowElim r s n i =
  let x = array_get s i in
  array_set s i fl0;
  let rec loop k =
    if k + 1 <= n then (
      array_set s k (minus (array_get s k) (mult x (array_get r k)));
      loop (k+1))
    else () in
  loop (i+1)

let gauss data =
  let n = matrix_dim1 data in
  let m = matrix_dim2 data in

  let rec rowMax i j x mx =
    eif j + 1 <= n then
      let y = fabs (matrix_get data j i) in
      eif (less x y) then rowMax i (j+1) y j
      else rowMax i (j+1) x mx
    else mx in

  let rec loop1 i =
    if i + 1 <= n then
      let x = fabs (matrix_get data i i) in
      let mx = rowMax i (i+1) x i in
      norm (getRow data mx) m i;
      rowSwap data i mx;
      let rec loop2 j =
        if j + 1 <= n then (
          rowElim (getRow data i) (getRow data j) m i;
          loop2 (j+1))
        else () in
      loop2 (i+1);
      loop1 (i+1)
    else () in
  loop1 0
```

Value items from *liquid_ gauss.gadti.target*:

```
val getRow :
   ∀i, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i].
   Matrix (i, n) → Num k → Array (Float, n)
```

```
val putRow :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧ n ⩽ j].
   Matrix (i, j) → Num k → Array (Float, n) → ()

val rowSwap :
   ∀i, j, k, n[0 ⩽ n ∧ 0 ⩽ k ∧ k + 1 ⩽ i ∧ n + 1 ⩽ i ∧
   0 ⩽ j]. Matrix (i, j) → Num k → Num n → ()

val norm :
   ∀i, k, n[0 ⩽ n ∧ n + 1 ⩽ i ∧ k ⩽ i].
   Array (Float, i) → Num k → Num n → ()

val rowElim :
   ∀i, j, k, n[0 ⩽ n ∧ n + 1 ⩽ i ∧ k ⩽ i ∧ k ⩽ j].
   Array (Float, j) → Array (Float, i) → Num k → Num n → ()

val gauss : ∀k, n[0 ⩽ n ∧ 1 ⩽ k ∧ n ⩽ k]. Matrix (n, k) → ()
```

The last toplevel definition from *liquid_ gauss2.gadt*, requires `-prefer_bound_to_local`:

```
let gauss data =
  let n = matrix_dim1 data in

  let rec rowMax i j x mx =
    eif j + 1 <= n then
      let y = fabs (matrix_get data j i) in
      eif (less x y) then rowMax i (j+1) y j
      else rowMax i (j+1) x mx
    else mx in

  let rec loop1 i =
    if i + 1 <= n then
      let x = fabs (matrix_get data i i) in
      let mx = rowMax i (i+1) x i in
      norm (getRow data mx) (n+1) i;
      rowSwap data i mx;
      let rec loop2 j =
        if j + 1 <= n then (
          rowElim (getRow data i) (getRow data j) (n+1) i;
          loop2 (j+1))
        else () in
      loop2 (i+1);
      loop1 (i+1)
    else () in
  loop1 0
```

The inferred types do not differ from the *liquid_gauss.gadt* case.

The last toplevel definition from *liquid_gauss_harder_asserted.gadt*:

```
let gauss data =
  let n = matrix_dim1 data in

  let rec rowMax = efunction i ->
    let x = fabs (matrix_get data i i) in
    let rec loop j x mx =
      assert num mx + 1 <= n;
      eif j + 1 <= n then
        let y = fabs (matrix_get data j i) in
        eif (less x y) then loop (j+1) y j
        else loop (j+1) x mx
      else mx in
    loop (i+1) x i in

  let rec loop1 i =
    if i + 1 <= n then
      let mx = rowMax i in
      norm (getRow data mx) (n+1) i;
      rowSwap data i mx;
      let rec loop2 j =
        if j + 1 <= n then (
          rowElim (getRow data i) (getRow data j) (n+1) i;
          loop2 (j+1))
        else () in
      loop2 (i+1);
      loop1 (i+1)
    else () in
  loop1 0
```

The inferred types do not differ from the *liquid_gauss.gadt* case.

C.8.13. Program `fft`

Source file *liquid_fft_full.gadt*:

```
external let n2i : ∀n. Num n → Int = "fun i -> i"
external let div2 : ∀n. Num (2 n) → Num n = "fun i -> i / 2"
external let div4 : ∀n. Num (4 n) → Num n = "fun i -> i / 4"
external let n2f : ∀n. Num n → Float = "float_of_int"
external let equal : ∀a. a → a → Bool = "fun x y -> x = y"
external let leq : ∀a. a → a → Bool = "fun x y -> x <= y"
external let less : ∀a. a → a → Bool = "fun x y -> x < y"
external let ignore : ∀a. a → () = "ignore"
```

```
external let abs : Float → Float = "abs_float"
external let cos : Float → Float = "cos"
external let sin : Float → Float = "sin"
external let neg : Float → Float = "(~-.)"
external let minus : Float → Float → Float = "(-.)"
external let plus : Float → Float → Float = "(+.)"
external let mult : Float → Float → Float = "( *. )"
external let div : Float → Float → Float = "( /. )"
external let fl0 : Float = "0.0"
external let fl05 : Float = "0.5"
external let fl1 : Float = "1.0"
external let fl2 : Float = "2.0"
external let fl3 : Float = "3.0"
external let fl4 : Float = "4.0"
external let pi : Float = "4.0 *. atan 1.0"
external let two_pi : Float = "8.0 *. atan 1.0"

datatype Bounded : num * num
datacons Index : ∀i, k, n[n ⩽ i ∧ i ⩽ k].Num i ⟶ Bounded (n, k)

let ffor s d body =
  let rec loop i =
    if i <= d then (body (Index i); loop (i + 1)) else () in
  loop s

external let ref : ∀a. a → Ref a = "fun a -> ref a"
external let asgn : ∀a. Ref a → a → () = "fun a b -> a := b"
external let deref : ∀a. Ref a → a = "(!)"

let fft px py = (* n must be a power of 2! *)
  let n = array_length px + (-1) in
  let rec loop n2 n4 =
    if n2 <= 2 then () else (* the case n2 = 2 is treated below *)
    let e = div two_pi (n2f n2) in
    let e3 = mult fl3 e in
    let a = ref fl0 in
    let a3 = ref fl0 in
    let rec forbod j' =
    match j' with Index j ->
    (*for j = 1 to n4 do*)
      let cc1 = cos (deref a) in
      let ss1 = sin (deref a) in
      let cc3 = cos (deref a3) in
      let ss3 = sin (deref a3) in
```

```
    asgn a (plus (deref a) e);
    asgn a3 (plus (deref a3) e3);
    let rec loop1 i0 i1 i2 i3 id =
      if n + 1 <= i3 then () else (* out_of_bounds *)
      let g_px_i0 = array_get px i0 in
      let g_px_i2 = array_get px i2 in
      let r1 = minus g_px_i0 g_px_i2 in
      let r1' = plus g_px_i0 g_px_i2 in
      array_set px i0 r1';
      let g_px_i1 = array_get px i1 in
      let g_px_i3 = array_get px i3 in
      let r2 = minus g_px_i1 g_px_i3 in
      let r2' = plus g_px_i1 g_px_i3 in
      array_set px i1 r2';
      let g_py_i0 = array_get py i0 in
      let g_py_i2 = array_get py i2 in
      let s1 = minus g_py_i0 g_py_i2 in
      let s1' = plus g_py_i0 g_py_i2 in
      array_set py i0 s1';
      let g_py_i1 = array_get py i1 in
      let g_py_i3 = array_get py i3 in
      let s2 = minus g_py_i1 g_py_i3 in
      let s2' = plus g_py_i1 g_py_i3 in
      array_set py i1 s2';
      let s3 = minus r1 s2 in
      let r1 = plus r1 s2 in
      let s2 = minus r2 s1 in
      let r2 = plus r2 s1 in
      array_set px i2 (minus (mult r1 cc1) (mult s2 ss1));
      array_set py i2 (minus (mult (neg s2) cc1) (mult r1 ss1));
      array_set px i3 (plus (mult s3 cc3) (mult r2 ss3));
      array_set py i3 (minus (mult r2 cc3) (mult s3 ss3));
      loop1 (i0 + id) (i1 + id) (i2 + id) (i3 + id) id in
    let rec loop2 is id =
      if n <= is then ()
      else (
        let i1 = is + n4 in
        let i2 = i1 + n4 in
        let i3 = i2 + n4 in
        loop1 is i1 i2 i3 id;
        loop2 (2 * id + j - n2) (4 * id)) in
    loop2 j (2 * n2) in
  ffor 1 n4 forbod;
  loop (div2 n2) (div2 n4) in
loop n (div4 n);
```

```
  let rec loop1 i0 i1 id =
    if n + 1 <= i1 then () else
    let r1 = array_get px i0 in
    array_set px i0 (plus r1 (array_get px i1));
    array_set px i1 (minus r1 (array_get px i1));
    let r1 = array_get py i0 in
    array_set py i0 (plus r1 (array_get py i1));
    array_set py i1 (minus r1 (array_get py i1));
    loop1 (i0 + id) (i1 + id) id in
  let rec loop2 is id =
    if n <= is then () else (
      loop1 is (is + 1) id;
      loop2 (2 * id + (-1)) (4 * id)) in
  loop2 1 4;

  let rec loop1 j k =
    eif j <= k then j + k
    else loop1 (j - k) (div2 k) in

  let rec loop2 i j =
    if n <= i then () else (
      if j <= i then () else (
        let xt = array_get px j in
        array_set px j (array_get px i); array_set px i (xt);
        let xt = array_get py j in
        array_set py j (array_get py i); array_set py i (xt));
      let j' = loop1 j (div2 n) in
      loop2 (i + 1) j') in
  loop2 1 1; n

let ffttest np =
  let enp = n2f np in
  let n2 = div2 np in
  let npm = n2 - 1 in
  let pxr = array_make (np+1) fl0 in
  let pxi = array_make (np+1) fl0 in
  let t = div pi enp in
  array_set pxr 1 (mult (minus enp fl1) fl05);
  array_set pxi 1 (fl0);
  array_set pxr (n2+1) (neg (mult fl1 fl05));
  array_set pxi (n2+1) fl0;
  let rec forbod i =
    if i <= npm then
      let j = np - i in
```

```
      array_set pxr (i+1) (neg (mult fl1 fl05));
      array_set pxr (j+1) (neg (mult fl1 fl05));
      let z = mult t (n2f i) in
      let y = mult (div (cos z) (sin z)) fl05 in
      array_set pxi (i+1) (neg y); array_set pxi (j+1) (y)
    else () in
  forbod 1;
  ignore (fft pxr pxi);
  (* lukstafi: kr and ki are placeholders? *)
  let rec loop i zr zi kr ki =
    if np <= i then (zr, zi) else
      let a = abs (minus (array_get pxr (i+1)) (n2f i)) in
      let b = less zr a in
      let zr = if b then a else zr in
      let kr = eif b then i else kr in
      let a = abs (array_get pxi (i+1)) in
      let b = less zi a in
      let zi = if b then a else zi in
      let ki = eif b then i else ki in
      loop (i+1) zr zi kr ki in
  let zr, zi = loop 0 fl0 fl0 0 0 in
  let zm = if less (abs zr) (abs zi) then zi else zr in
  (*in print_float zm; print_newline ()*) zm

let rec loop_np i np =
  if 17 <= i then () else
  ( ignore (ffttest np); loop_np (i + 1) (2 * np) )

let doit _ = loop_np 4 16
```

Value items from *liquid_fft_full.gadti.target*:

```
val ffor :
   ∀i, j, k, n[j ⩽ i ∧ k ⩽ n].
   Num n → Num j → (Bounded (k, i) → ()) → ()

val fft :
   ∀k, n[1 ⩽ n ∧ n + 1 ⩽ k].
   Array (Float, n + 1) → Array (Float, k) → Num n

val ffttest : ∀n[2 ⩽ n]. Num n → Float

val loop_np : ∀k, n[2 ⩽ n]. Num k → Num n → ()

val doit : ∀a. a → ()
```

Appendix D

InvarGenT: Manual

D.1. Introduction

Type systems are an 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 between types can be very simple. Existential types 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, when the dependence on the input to the program is complex. GADTs express existential types by using local type variables for the hidden parts of the type encapsulated in a GADT.

The InvarGenT type system for GADTs differs from more pragmatic approaches in mainstream functional languages in that we do not require any type annotations on expressions, even on recursive functions. The implementation also includes linear equations and inequalities over rational numbers in the language of types, with the possibility to introduce more domains in the future.

D.2. Tutorial

The concrete syntax of InvarGenT is similar to that of OCaml. However, it does not currently cover records, the module system, objects, and polymorphic variant types. It supports higher-order functions, algebraic data-types including built-in tuple types, and linear pattern matching. It supports conjunctive patterns using the `as` keyword, but it currently does not support disjunctive patterns. It currently has limited support for guarded patterns: after `when`, only inequality `<=` between values of the `Num` type are allowed.

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 called `type`, i.e. "types proper". `i`,`j`,`k`,`l`,`m`,`n`,`i1`,... are in the sort of linear arithmetics over rational numbers called `num`. Remaining letters are reserved for sorts that may be added in the future. Value constructors (like in OCaml) and type constructors (unlike in OCaml) have the same syntax: capitalized name followed by a tuple of arguments. They are introduced by `datatype` and `datacons` respectively. The `datatype` declaration

might be misleading in that it only lists the sorts of the arguments of the type, the resulting sort is always `type`. Values assumed into the environment are introduced by `external`. There is a built-in type corresponding to declaration `datatype Num : num` and definitions of numeric constants `newcons 0 : Num 0 newcons 1 : Num 1`... The programmer can use `external` declarations to give the semantics of choice to the `Num` data-type. The type with additional support as `Num` is the integers.

When solving negative constraints, arising from `assert false` clauses, we assume that the intended domain of the sort `num` is integers. This is a workaround to the lack of strict inequality in the sort `num`. We do not make the whole sort `num` an integer domain because it would complicate the algorithms.

In examples here we use Unicode characters. For ASCII equivalents, take a quick look at the tables in the following section.

We start with a simple example, a function that can compute a value from a representation of an expression – a ready to use value whether it be `Int` or `Bool`. Prior to the introduction of GADT types, we could only implement a function `eval : ∀a. Term a → Value` where, using OCaml syntax, `type value = Int of int | Bool of bool`.

```
datatype Term : type
external let plus : Int → Int → Int = "(+)"
external let is_zero : Int → Bool = "(=) 0"
datacons Lit : Int ⟶ Term Int
datacons Plus : Term Int * Term Int ⟶ Term Int
datacons IsZero : Term Int ⟶ Term Bool
datacons 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 then eval t else eval e
```

Let us look at the corresponding generated, also called *exported*, OCaml source code:

```
type _ term =
  | Lit : int -> int term
  | Plus : int term * int term -> int term
  | IsZero : int term -> bool term
  | If : (*∀'a.*)bool term * 'a term * 'a term -> 'a term

let plus : (int -> int -> int) = (+)
let is_zero : (int -> bool) = (=) 0
let rec eval : type a . (a term -> a) =
  (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 then eval t else eval e))
```

The `Int`, `Num` and `Bool` types are built-in. `Int` and `Bool` follow the general scheme of exporting a datatype constructor with the same name, only lower-case. However, numerals `0`, `1`, ... are always type-checked as `Num 0`, `Num 1`... `Num` can also be exported as a type other than `int`, and then numerals are exported via an injection function (ending with) `of_int`.

The syntax `external let` allows us to name an OCaml library function or give an OCaml definition which we opt-out from translating to InvarGenT. Such a definition will be verified against the rest of the program when InvarGenT calls `ocamlc -c` to verify the exported code. Another variant of `external` (omitting the `let` keyword) exports a value using `external` in OCaml code, which is OCaml source declaration of the foreign function interface of OCaml. When we are not interested in linking and running the exported code, we can omit the part starting with the `=` sign. The exported code will reuse the name in the FFI definition: `external f : ... = "f"`.

The type inferred for the above example is `eval : ∀a. Term a→a`. GADTs make it possible to reveal that `IsZero x` is a `Term Bool` and therefore the result of `eval` should in its case be `Bool`, `Plus (x, y)` is a `Term Num` and the result of `eval` should in its case be `Num`, etc. The `if/eif...then...else...` syntax is a syntactic sugar for `match/ematch...with True ->... | False ->...`, and any such expressions are exported using `if` expressions.

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

```
datatype Ty : type
datatype Int
datatype List : type
datacons Zero : Int
datacons Nil : ∀a. List a
datacons TInt : Ty Int
datacons TPair : ∀a, b. Ty a * Ty b ⟶ Ty (a, b)
datacons TList : ∀a. Ty a ⟶ Ty (List a)
datatype Boolean
datacons True : Boolean
datacons False : Boolean
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)→a→a→Bool`, one of four maximally general types of `equal` as defined above. The other maximally general "wrong" types are `∀a,b.(Ty a, Ty b)→b→b→Bool` and `∀a,b.(Ty a, Ty b)→b→a→Bool`. 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 three means to enrich the generated constraints: `assert false` syntax for providing negative constraints, `assert type e1 = e2;...` and `assert num e1 <= e2;...` for positive constraints, and `test` syntax for including constraints of use cases with constraint of a toplevel definition. To ensure only one maximally general type for `equal`, we use `assert false` and `test`. We can either add the `assert false` clauses:

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

The first assertion excludes independence of the first encoded type and the second argument. The second assertion excludes independence of the second encoded type and the third argument. Or we can add the `test` clause:

```
test b_not (equal (TInt, TList TInt) Zero Nil)
```

The test ensures that arguments of distinct types can be given. InvarGenT returns the expected type `equal: ∀a,b.(Ty a, Ty b)→a→b→Bool`.

Now we demonstrate numerical invariants:

```
datatype Binary : num
datatype Carry : num
datacons Zero : Binary 0
datacons PZero : ∀n[0⩽n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n[0⩽n]. Binary n ⟶ Binary(2 n + 1)
datacons CZero : Carry 0
datacons 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))
      | POne a1 as a ->
```

```
        (function Zero -> a
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1)))
    | COne ->
    (function Zero ->
        (function Zero -> POne(Zero)
          | PZero b1 -> POne b1
          | POne b1 -> PZero (plus COne Zero b1))
      | PZero a1 as a ->
        (function Zero -> POne a1
          | PZero b1 -> POne (plus CZero a1 b1)
          | POne b1 -> PZero (plus COne a1 b1))
      | POne a1 as a ->
        (function Zero -> PZero (plus COne a1 Zero)
          | PZero b1 -> PZero (plus COne a1 b1)
          | POne b1 -> POne (plus COne a1 b1)))
```

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

We can introduce existential types directly in type declarations. To have an existential type inferred, we have to use `efunction`, `ematch` or `eif` expressions, which differ from `function`, `match`, `eif` respectively 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. In the following artificial example, we abstract away the particular resulting location.

```
datatype Room
datatype Yard
datatype Village
datatype Castle : type
datatype Place : type
datacons Room : Room ⟶ Castle Room
datacons Yard : Yard ⟶ Castle Yard
datacons CastleRoom : Room ⟶ Place Room
datacons CastleYard : Yard ⟶ Place Yard
datacons 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`. Next consider a slightly less artificial, toy example of computer hardware configuration. It illustrates many aspects of existential types in InvarGenT. We introduce functions `config_mem_board` and `config_gpu` as external, i.e., ones whose definition is not type-checked by InvarGenT. Their types illustrate that existential types can be used in type annotations. Types `Slow` and `Fast`, although declared as data-types, are phantom types, i.e. are not inhabited and convey information as parameters of other types.

```
datatype Slow  datatype Fast  datatype Budget
datacons Small : Budget  datacons Medium : Budget  datacons Large : Budget
datatype Memory : type  datacons Best_mem : Memory Fast
datatype Motherboard : type  datacons Best_board : Motherboard Fast
external config_mem_board : Budget → ∃a. (Memory a, Motherboard a)
datatype CPU : type
datacons FastCPU : CPU Fast  datacons SlowCPU : CPU Slow
datatype GPU : type
datacons FastGPU : GPU Fast  datacons SlowGPU : GPU Slow
external config_gpu : Budget → ∃a. GPU a
datatype PC : type * type * type * type
datacons PC :
  ∀a,b,c,r. CPU a * GPU b * Memory c * Motherboard r ⟶ PC (a,b,c,r)
datatype Usecase  datacons Gaming : Usecase
datacons Scientific : Usecase  datacons Office : Usecase

let budget_to_cpu = efunction
  | Small -> SlowCPU | Medium -> FastCPU | Large -> FastCPU
let usecase_to_gpu budget = efunction
  | Gaming -> FastGPU | Scientific -> FastGPU
  | Office -> config_gpu budget
let rec configure = efunction
  | Small, Gaming -> configure (Small, Office)
  | Large, Gaming -> PC (FastCPU, FastGPU, Best_mem, Best_board)
  | budget, usecase ->
    let mem, board = config_mem_board budget in
    let cpu = budget_to_cpu budget in
    let gpu = usecase_to_gpu budget usecase in
    PC (cpu, gpu, mem, board)
```

InvarGenT infers the following types:

```
budget_to_cpu : Size → ∃a.CPU a
usecase_to_gpu : Usecase → ∃a.GPU a
```

```
configure : (Size, Usecase) → ∃a, b, c.PC (a, b, c, c)
```

The definition of `configure` illustrates explicit elimination of existential types by `let..in` definitions: by design, inlining of `cpu` or `gpu` definitions would make the function not typeable. The call to `config_gpu` and the recursive call to `configure` illustrate implicit elimination of existential types in return positions.

A more practical existential type example:

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

let rec filter f =
  efunction LNil -> LNil
    | LCons (x, xs) ->
      eif f x then
          let ys = filter f xs in
          LCons (x, ys)
      else filter f xs
```

We get `filter: ∀n, a.(a→Bool)→List (a, n)→ ∃k[0⩽k ∧ k⩽n].List (a, k)`. Note that we need to use both `efunction` and `eif` above, since every use of `function`, `match` or `if` will force the types of its branches to be equal. In particular, for lists with length the resulting length would have to be the same in each branch. If the constraint cannot be met, as for `filter` with either `function` or `if`, the code will not type-check.

A more complex example that computes bitwise *or* – `ub` stands for "upper bound":

```
datatype Binary : num
datacons Zero : Binary 0
datacons PZero : ∀n [0⩽n]. Binary n ⟶ Binary(2 n)
datacons POne : ∀n [0⩽n]. Binary n ⟶ Binary(2 n + 1)

let rec ub = efunction
  | Zero ->
      (efunction Zero -> Zero
        | PZero b1 as b -> b
        | POne b1 as b -> b)
  | PZero a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          PZero r
```

```
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
  | POne a1 as a ->
      (efunction Zero -> a
        | PZero b1 ->
          let r = ub a1 b1 in
          POne r
        | POne b1 ->
          let r = ub a1 b1 in
          POne r)
```

`ub:∀k,n.Binary k→Binary n→∃:i[0⩽n ∧ 0⩽k ∧ n⩽i ∧ k⩽i ∧ i⩽n+k].Binary i`.

Why cannot we shorten the above code by converting the initial cases to `Zero -> (efunction b -> b)`? Without pattern matching, we do not make the contribution of `Binary n` available. Knowing `n=i` and not knowing `0⩽n`, for the case `k=0`, we get: `ub:∀k, n.Binary k→Binary n→∃i[0⩽k∧n⩽i∧i⩽n+k].Binary i`. `n≤i` follows from `n=i`, `i⩽n+k` follows from `n=i` and `0⩽k`, but `k⩽i` cannot be inferred from `k=0` and `n=i` without knowing that `0⩽n`.

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

D.3. Syntax

Below we present, using examples, the syntax of InvarGenT: the mathematical notation, the concrete syntax in ASCII and the concrete syntax using Unicode.

type variable: types	$\alpha, \beta, \gamma, \tau$	`a,b,c,r,s,t,a1,...`	
type variable: nums	k, m, n	`i,j,k,l,m,n,i1,...`	
type var. with coef.	$\frac{1}{3}n$	`1/3 n`	
type constructor	List	`List`	
number (type)	7	`7`	
numerical sum (type)	$m+n$	`m+n`	
existential type	$\exists k, n[k \leqslant n].\tau$	`ex k, n [k<=n].t`	`∃k,n[k⩽n].t`
type sort	s_{type}	`type`	
number sort	s_R	`num`	
function type	$\tau_1 \to \tau_2$	`t1 -> t2`	`t1 → t2`
equation	$a \doteq b$	`a = b`	
inequation	$k \leqslant n$	`k <= n`	`k ⩽ n`
conjunction	$\varphi_1 \wedge \varphi_2$	`a=b && b=a`	`a=b ∧ b=a`

For the syntax of expressions, we discourage non-ASCII symbols. Below e, e_i stand for any expression, p, p_i stand for any pattern, x stands for any lower-case identifier and K for an upper-case identifier. K_T stands for `True`, K_F for `False`, and K_u for `()`.

named value	x	`x` –lower-case identifier
numeral (expr.)	7	`7`
constructor	K	`K` –upper-case identifier
application	$e_1\, e_2$	`e1 e2`
non-br. function	$\lambda(p_1.\lambda(p_2.e))$	`fun (p1,p2) p3 -> e`
branching function	$\lambda(p_1.e_1 ... p_n.e_n)$	`function p1->e1 | ... | pn->en`
pattern match	$\lambda(p_1.e_1 ... p_n.e_n)\, e$	`match e with p1->e1 | ...`
if-then-else clause	$\lambda(K_T.e_1, K_F.e_2)\, e$	`if e then e1 else e2`
if-then-else condition	$\lambda(\_\ \mathbf{when}\, m \leqslant n.e_1, ...)\, K_u$	`if m <= n then e1 else e2`
postcond. function	$\lambda[K](p_1.e_1 ... p_n.e_n)$	`efunction p1->e1 | ...`
postcond. match	$\lambda[K](p_1.e_1 ... p_n.e_n)\, e$	`ematch e with p1->e1 | ...`
eif-then-else clause	$\lambda[K](K_T.e_1, K_F.e_2)\, e$	`eif e then e1 else e2`
eif-then-else condition	$\lambda[K](\_\ \mathbf{when}\, m \leqslant n.e_1, ...)\, K_u$	`eif m <= n then e1 else e2`
rec. definition	$\mathbf{let\ rec}\, x = e_1\, \mathbf{in}\, e_2$	`let rec x = e1 in e2`
definition	$\mathbf{let}\, p = e_1\, \mathbf{in}\, e_2$	`let p1,p2 = e1 in e2`
asserting dead br.	**assert false**	`assert false`
runtime failure	runtime failure s	`runtime_failure s`
assert equal types	$\mathbf{assert\ type}\, \tau_{e_1} \doteq \tau_{e_2}; e_3$	`assert type e1 = e2; e3`
assert inequality	$\mathbf{assert\ num}\, e_1 \leqslant e_2; e_3$	`assert num e1 <= e2; e3`

A built-in fail at runtime with the given text message is only needed for introducing existential types: a user-defined equivalent of `runtime_failure` would introduce a spurious branch for generalization.

Toplevel expressions (corresponding to structure items in OCaml) introduce types, type and value constructors, global variables with given type (external names) or inferred type (definitions).

type constructor	`datatype List : type * num`
value constructor	`datacons Cons : all n a. a * List(a,n) --> List(a,n+1)`
	`datacons Cons : ∀n,a. a * List(a,n) ⟶ List(a,n+1)`
declaration	`external foo : ∀n,a. List(a,n)→∃k[k<=n].List(a,k)="c_foo"`
	`external filter : ∀n,a. List(a,n)→∃k[k<=n].List(a,k)`
let-declaration	`external let mult : ∀n,m. Num n→Num m→∃k.Num k = "( * )"`
rec. definition	`let rec f =...`
non-rec. definition	`let a, b =...`
definition with test	`let rec f =... test e1; ...; en`

Toplevel non-recursive `let` definitions are polymorphic as an exception. In expressions, `let...in` definitions are monomorphic, one should use the `let rec...in` syntax to get a polymorphic `let`-binding.

Tests list expressions of type `Bool` that at runtime have to evaluate to `True`. Type inference is affected by the constraints generated to typecheck the expressions.

There are variants of the if-then-else clause syntax supporting when conditions:

- `if m1 <= n1 && m2 <= n2 &&... then e1 else e2` is $\lambda(\_\ \mathbf{when}\ \wedge_i m_i \leqslant n_i.e_1,\ \_\ .e_2)\, K_u$,

- `if m <= n then e1 else e2` is $\lambda(\_\ \mathbf{when}\, m \leqslant n.e_1, \_\ \mathbf{when}\, n+1 \leqslant m.e_2)\, K_u$ if integer mode is on (as in default setting),
- similarly for the `eif` variants.

We add the standard syntactic sugar for function definitions:

- `let` p_1 $p_2 \ldots$ p_n `=` e_1 `in` e_2 expands to `let` p_1 `= fun` $p_2 \ldots$ p_n `->` e_1 `in` e_2
- `let rec` l_1 $p_2 \ldots$ p_n `=` e_1 `in` e_2 expands to `let rec` l_1 `= fun` $p_2 \ldots$ p_n `->` e_1 `in` e_2
- toplevel `let` and `let rec` definitions expand correspondingly.

For simplicity of theory and implementation, mutual non-nested recursion and or-patterns are not provided. For mutual recursion, nest one recursive definition inside another.

Like in OCaml, types of arguments in declarations of constructors are separated by asterisks. However, the type constructor for tuples is represented by commas, like in Haskell but unlike in OCaml.

At any place between lexemes, regular comments encapsulated in `(*...*)` can occur. They are ignored during lexing. In front of all toplevel definitions and declarations, e.g. before a `datatype`, `datacons`, `external`, `let rec` or `let`, and in front of `let rec...in` and `let...in` nodes in expressions, documentation comments `(**...*)` can be put. Documentation comments at other places are syntax errors. Documentation comments are preserved both in generated interface files and in exported source code files.

D.4. Solver Parameters and CLI

The default settings of InvarGenT parameters should be sufficient for most cases. For example, after downloading InvarGenT source code and changing current directory to `invargent`, we can enter, assuming a Unix-like shell:

```
$ make main
$ ./invargent examples/binary_upper_bound.gadt
```

To get the inferred types printed on standard output, use the `-inform` option:

```
$ ./invargent -inform examples/avl_tree.gadt
```

Below we demonstrate what happens with insufficiently high parameter setting. Consider this example:

```
$ ./invargent -inform examples/flatten_septs.gadt
File "examples/flatten_septs.gadt", line 8, characters 6-104:
No answer in type: term abduction failed

Perhaps increase the -term_abduction_timeout parameter.
Perhaps increase the -term_abduction_fail parameter.
```

```
$ ./invargent -inform -term_abduction_timeout 3000 examples/
flatten_septs.gadt
File "examples/flatten_septs.gadt", line 3, characters 26-261:
No answer in num: numerical abduction failed

Perhaps do not pass the -no_dead_code flag.
Perhaps increase the -num_abduction_timeout parameter.
Perhaps increase the -num_abduction_fail parameter.
$ ./invargent -inform -term_abduction_timeout 3000 \
  -num_abduction_rotations 4 examples/flatten_septs.gadt
val flatten_septs :
  ∀n, a. List ((a, a, a, a, a, a, a), n) → List (a, 7 n)
InvarGenT: Generated file examples/flatten_septs.gadti
InvarGenT: Generated file examples/flatten_septs.ml
InvarGenT: Command "ocamlc -w -25 -c examples/flatten_septs.ml" exited with
code 0
```

The `Perhaps increase` suggestions are generated only when the corresponding limit has actually been exceeded. Remember however that the limits will often be exceeded for erroneus programs which should not type-check. Moreover, as illustrated above, other settings might be the culprit.

To understand the intent of the solver parameters, we need a rough "birds-eye view" understanding of how InvarGenT works. The invariants and postconditions that we solve for are logical formulas and can be ordered by strength. Least Upper Bounds (LUBs) and Greatest Lower Bounds (GLBs) computations are traditional tools used for solving recursive equations over an ordered structure. In case of implicational constraints that are generated for type inference with GADTs, constraint abduction is a form of LUB computation. *Constraint generalization* is our term for computing the GLB wrt. strength for formulas that are conjunctions of atoms. We want the invariants of recursive definitions – i.e. the types of recursive functions and formulas constraining their type variables – 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. Therefore the use of LUB, constraint abduction. For postconditions – i.e. the existential types of results computed by `efunction` expressions and formulas constraining their type variables – we want the strongest possible solutions, because stronger postcondition provides more information at use sites of a definition. Therefore we use GLB, constraint generalization, but only if existential types have been introduced by `efunction` or `ematch`.

Below we discuss all of the InvarGenT options. We use the technical term *terms* to mean type shapes, types without the concern for the sort of numbers (or other sorts to come in the future).

`-inform`. Print type schemes of toplevel definitions as they are inferred.

`-time`. Print the time it took to infer type schemes of toplevel definitions.

`-no_sig`. Do not generate the `.gadti` file.

`-no_ml`. Do not generate the `.ml` file.

`-overwrite_ml`. Overwrite the `.ml` file if it already exists.

`-ml_file`. Generate the exported OCaml file under the provided name.

`-no_verif`. Do not call `ocamlc -c` on the generated `.ml` file.

`-num_is`. The exported type for which `Num` is an alias (default `int`). If `-num_is bar` for `bar` different than `int`, numerals are exported as integers passed to a `bar_of_int` function. The variant `-num_is_mod` exports numerals by passing to a `Bar.of_int` function.

`-full_annot`. Annotate the `function` and `let..in` nodes in generated OCaml code. This increases the burden on inference a bit because the variables associated with the nodes cannot be eliminated from the constraint during initial simplification.

`-keep_assert_false`. Keep `assert false` clauses in exported code. When faced with multiple maximally general types of a function, we sometimes want to prevent some interpretations by asserting that a combination of arguments is not possible. These arguments will not be compatible with the type inferred, causing exported code to fail to typecheck. Sometimes we indicate unreachable cases just for documentation. If the type is tight this will cause exported code to fail to typecheck too. This option keeps pattern matching branches with `assert false` in their bodies in exported code nevertheless.

`-allow_dead_code`. Allow more programs with dead code than would otherwise pass.

`-force_no_dead_code`. Reject all programs with dead code (may misclassify programs using *min* or *max* atoms). Unreachable pattern matching branches lead to unsatisfiable premises of the type inference constraint, which we detect. However, sometimes multiple implications in the simplified form of the constraint can correspond to the same path through the program, in particular when solving constraints with *min* and *max* clauses. Dead code due to datatype mismatch, i.e. patterns unreachable without resort to numerical constraints, is detected even without using this option.

`-term_abduction_timeout`. Limit on term simple abduction steps (default 700). Simple abduction works with a single implication branch, which roughly corresponds to a single branch – an execution path – of the program.

`-term_abduction_fail`. Limit on backtracking steps in term joint abduction (default 4). Joint abduction combines results for all branches of the constraints.

`-no_alien_prem`. Do not include alien (e.g. numerical) premise information in term abduction.

`-early_num_abduction`. Include recursive branches in numerical abduction from the start. By default, in the second iteration of solving constraints, which is the first iteration that numerical abduction is performed, we only pass non-recursive branches to numerical abduction. This makes it faster but less likely to find the correct solution.

`-convergence_step`. The iteration at which to start truncating postconditions by only keeping atoms present in the previous iteration, to force convergence (default 8).

`-early_postcond_abd`. Include postconditions from recursive calls in abduction from the start. We do not derive requirements put on postconditions by recursive calls on first iteration. The requirements may turn smaller after some derived invariants are included in the premises. This option turns off the special treatment of postconditions on first iteration.

`-num_abduction_rotations`. Numerical abduction: coefficients from ± 1 / N to $\pm N$ (default 3). Numerical abduction answers are built, roughly speaking, by adding premise equations of a branch with conclusion of a branch to get an equation or inequality that does not conflict with other branches, but is equivalent to the conclusion equation/inequality. This parameter decides what range of coefficients is tried. If the highest coefficient in correct answer is greater, abduction might fail. However, it often succeeds because of other mechanisms used by the abduction algorithm.

`-num_prune_at`. Keep less than N elements in abduction sums (default 6). By elements here we mean distinct variables – lack of constant multipliers in concrete syntax of types is just a syntactic shortcoming.

`-num_abduction_timeout`. Limit on numerical simple abduction steps (default 1000).

`-num_abduction_fail`. Limit on backtracking steps in numerical joint abduction (default 10).

`-affine_penalty`. How much to penalize an abduction candidate inequality for containing a constant term (default 4). Too small a value may lead to divergence, e.g. in some examples abduction will pick an answer $a + 1$, which in the following step will force an answer $a + 2$, then $a + 3$, etc.

`-reward_constrn`. How much to reward introducing a constraint on so-far unconstrained varialbe, or penalize if negative (default 2).

`-complexity_penalty`. How much to penalize an abduction candidate inequality for complexity of its coefficients; the coefficient of either the linear or power scaling of the coefficients (default 2.5).

`-abd_lin_thres_scaling`. Scale the complexity cost of coefficients linearly with a jump of the given height after coefficient 1 (default 2.0).

`-abd_pow_scaling`. Scale the complexity cost of coefficients according to the given power.

`-prefer_bound_to_local`. Prefer a bound coming from outer scope, to inequality between two local parameters. In numerical abduction heuristic, such bounds are usually doubly penalized: for having a constant, and non-locality of parameters.

`-prefer_bound_to_outer`. Prefer a bound coming from outer scope, to inequality between two outer scope parameters. Outer-scope constraints sometimes lead to a solution not general enough.

`-only_off_by_1`. Limit the effect of `-prefer_bound_to_local` and `-prefer_bound_to_outer` to inequalities with a constant 1. This corresponds to an upper bound of an index into a zero-indexed array/matrix/etc.

`-same_with_assertions`. Do not treat definitions with positive assertions (`assert num`, `assert type`) specially. The special treatment is currently equivalent to passing `-reward_constrn -1` and `-prefer_bound_to_local`.

`-concl_abd_penalty`. Penalize abductive guess when the supporting argument comes from the partial answer, instead of from the current premise (default 4). Guesses involving the partial answer are less secure, for example they depend on the order in which the constraint to explain is being processed.

`-more_general_num`. Filter out less general abduction candidate atoms (does not guarantee overall more general answers). The filtering is currently not performed by default to save on computational cost.

`-no_num_abduction`. Turn off numerical abduction; will not ensure correctness. Numerical abduction uses a brute-force algorithm and will fail to work in reasonable time for complex constraints. However, including the effects of `assert false` clauses, and inference of postconditions, do not rely on numerical abduction. If the numerical invariant of a typeable (i.e. correct) function follows from `assert false` facts alone, a call with `-no_num_abduction` may still find the correct invariant and postcondition.

`-if_else_no_when`. Do not add `when` clause to the `else` branch of an `if` expression with a single inequality as condition. Expressions `if`, resp. `eif`, with a single inequality as the condition are expanded into expressions `match`, resp. `ematch`, with `when` conditions on both the `True` branch and the `False` branch. I.e. `if m <= n then e1 else e2` is expanded into `match () with _ when m <= n -> e1 | _ when n+1 <= m -> e2`. Passing `-if_else_no_when` will result in expansion `match () with _ when m <= n -> e1 | _ -> e2`. The same effect can be achieved for a particular expression by artificially incresing the number of inequalities: `if m <= n && m <= n then e1 else e2`.

`-weaker_pruning`. Do not assume integers as the numerical domain when pruning redundant atoms.

`-stronger_pruning`. Prune atoms that force a numerical variable to a single value under certain conditions; exclusive with `-weaker_pruning`.

`-postcond_rotations`. In postconditions, check coefficients from $1 / N$ (default 3). Numerical constraint generalization is performed by approximately finding the convex hull of the polytopes corresponding to disjuncts. A step in an exact algorithm involves rotating a side along a ridge – an intersection with another side – until the side touches yet another side. We approximate by trying out a couple of rotations: convex combinations of the inequalities defining the sides. This parameter decides how many rotations to try.

`-postcond_opti_limit`. Limit the number of atoms $x = \min(a, b)$, $x = \max(a, b)$ in (intermediate and final) postconditions (default 4). Unfortunately, inference time is exponential in the number of atoms of this form. The final postconditions usually have few of these atoms, but a greater number is sometimes needed in the intermediate steps of the main loop.

`-postcond_subopti_limit`. Limit the number of atoms $\min(a, b) \leqslant x$, $x \leqslant \max(a, b)$ in (intermediate and final) postconditions (default 4). Unfortunately, inference time is exponential in the number of atoms of this form. The final postconditions usually have few of these atoms, but a greater number is sometimes needed in the intermediate steps of the main loop.

`-iterations_timeout`. Limit on main algorithm iterations (default 6). Answers found in an iteration of the main algorithm are propagated to use sites in the next iteration. However, for about four initial iterations, each iteration turns on additional processing which makes better sense with the results from the previous iteration propagated. At least three iterations will always be performed.

`-richer_answers`. Keep some equations in term abduction answers even if redundant. Try keeping an initial guess out of a list of candidate equations before trying to drop the equation from consideration. We use fully maximal abduction for single branches, which cannot find answers not implied by premise and conclusion of a branch. But we seed it with partial answer to branches considered so far. Sometimes an atom is required to solve another branch although it is redundant in given branch. `-richer_answers` does not increase computational cost but sometimes leads to answers that are not most general. This can always be fixed by adding a `test` clause to the definition which uses a type conflicting with the too specific type.

`-prefer_guess`. Try to guess equality-between-parameters before considering other possibilities. Implied by `-richer_answers` but less invasive.

`-more_existential`. More general invariant at expense of more existential postcondition. To avoid too abstract postconditions, constraint generalization can infer additional constraints over invariant parameters. In rare cases a weaker postcondition but a more general invariant can be beneficial.

`-show_extypes`. Show datatypes encoding existential types, and their identifiers with uses of existential types. The type system in InvarGenT encodes existential types as GADT types, but this representation is hidden from the user. Using `-show_extypes` exposes the representation as follows. The encodings are exported in `.gadti` files as regular datatypes named `exN`, and existential types are printed using syntax `∃N:...` instead of `∃...`, where `N` is the identifier of an existential type.

`-passing_ineq_trs`. Include inequalities in conclusion when solving numerical abduction. This setting leads to more inequalities being tried for addition in numeric abduction answer.

`-not_annotating_fun`. Do not keep information for annotating `function` nodes. This may allow eliminating more variables during initial constraint simplification.

`-annotating_letin`. Keep information for annotating `let..in` nodes. Will be set automatically anyway when `-full_annot` is passed.

`-let_in_fallback`. Annotate `let..in` nodes in fallback mode of `.ml` generation. When verifying the resulting `.ml` file fails, a retry is made with `function` nodes annotated. This option additionally annotates `let..in` nodes with types in the regenerated `.ml` file.

Let us have a look at tests from the `examples` direcotory that need a non-default parameter setting. The program `examples/flatten_septs.gadt` has already been shown above. It is the only example that needs the `-term_abduction_timeout` and `-num_abduction_rotations` settings. The need for `-num_abduction_rotations` comes from having an equation with large coefficient in the answer. The need for `-term_abduction_timeout` comes from having bigger type shapes to handle during term, i.e. type shape, abduction.

```
$ ./invargent -inform examples/non_pointwise_leq.gadt
File "examples/non_pointwise_leq.gadt", line 12, characters 14-60:
No answer in type: Answers do not converge

Perhaps increase the -iterations_timeout parameter or try one of the
options: -more_existential, -prefer_guess, -prefer_bound_to_local.
$ ./invargent -inform -prefer_guess examples/non_pointwise_leq.gadt
val leq : ∀a. Nat a → NatLeq (a, a)
InvarGenT: Generated file examples/non_pointwise_leq.gadti
InvarGenT: Generated file examples/non_pointwise_leq.ml
InvarGenT: Command "ocamlc -w -25 -c examples/non_pointwise_leq.ml" exited
with code 0
```

Other examples that need the `-prefer_guess` option: `non_pointwise_zip1_simpler.gadt`, `non_pointwise_zip1_simpler2.gadt`, `non_pointwise_zip1_modified.gadt`. On the other hand, `non_pointwise_zip1.gadt` is inferred with default settings.

The response from the system does not always include an option which would make the inference succeed.

```
$ ./invargent -inform examples/liquid_simplex_step_3a.gadt
File "examples/liquid_simplex_step_3a.gadt", line 7, characters 49-1651:
No answer in type: Answers do not converge

Perhaps do not pass the -no_dead_code flag.
Perhaps increase the -iterations_timeout parameter or try one of the
options: -more_existential, -prefer_guess, -prefer_bound_to_local.
Perhaps some definition is used with requirements on
its inferred postcondition not warranted by the definition.
$ ./invargent -inform -prefer_bound_to_local -only_off_by_1 \
examples/liquid_simplex_step_3a.gadt
val main_step3_test : ∀k, n[1 ⩽ n ∧ 3 ⩽ k]. Matrix (n, k) → Float
InvarGenT: Generated file examples/liquid_simplex_step_3a.gadti
InvarGenT: Generated file examples/liquid_simplex_step_3a.ml
File "examples/liquid_simplex_step_3a.ml", line 43, characters 8-9:
Warning 26: unused variable m.
InvarGenT: Command "ocamlc -w -25 -c examples/liquid_simplex_step_3a.ml"
exited with code 0
```

The other examples that need the `-prefer_bound_to_local` option, but not the `-only_off_by_1` option: `liquid_simplex_step_6a_2.gadt`, `liquid_tower_harder.gadt`, `liquid_gauss2.gadt`.

```
$ ./invargent -inform examples/pointwise_zip2_harder.gadt
val zip2 : ∀a, b. Zip2 (a, b) → a → b
InvarGenT: Generated file examples/pointwise_zip2_harder.gadti
InvarGenT: Generated file examples/pointwise_zip2_harder.ml
File "examples/pointwise_zip2_harder.ml", line 19, characters 21-32:
Error: This kind of expression is not allowed as right-hand side of ‘let
rec’
InvarGenT: Command "ocamlc -w -25 -c examples/pointwise_zip2_harder.ml"
exited with code 2
InvarGenT: Regenerated file examples/pointwise_zip2_harder.ml
File "examples/pointwise_zip2_harder.ml", line 21, characters 21-32:
Error: This kind of expression is not allowed as right-hand side of ‘let
rec’
InvarGenT: Command "ocamlc -w -25 -c examples/pointwise_zip2_harder.ml"
exited with code 2
$ ./invargent -inform -no_ml examples/pointwise_zip2_harder.gadt
val zip2 : ∀a, b. Zip2 (a, b) → a → b
InvarGenT: Generated file examples/pointwise_zip2_harder.gadti
```

The example `pointwise_zip2_harder.gadt` is not compatible with the pass-by-value semantics. We can avoid the complaint of the OCaml compiler by passing either the `-no_ml` flag or the `-no_verif` flag. More interestingly, we can notice that the file `pointwise_zip2_harder.ml` is generated twice. This happens because InvarGenT, noticing the failure, generates an OCaml source with more type information, as if the `-full_annot` option was used.

The examples `liquid_fft_simpler.gadt` and `liquid_fft_full_asserted.gadt` contain assertions, but are nearly as hard as `liquid_fft.gadt`, `liquid_fft_full.gadt` respectively. They need the option `-same_with_assertions` to not switch to settings tuned for cases where assertions capture the harder aspects of the invariants to infer.

Unfortunately, inference fails for some examples regardless of parameters setting. We discuss them in the next section.

D.5. Limitations of Current InvarGenT Inference

Type inference for the type system underlying InvarGenT is undecidable. In some cases, the failure to infer a type is not at all problematic. Consider this example due to Chuan-kai Lin:

```
datatype EquLR : type * type * type
datacons EquL : ∀a, b. EquLR (a, a, b)
datacons EquR : ∀a, b. EquLR (a, b, b)
datatype Box : type
```

```
datacons Cons : ∀a. a ⟶ Box a
external let eq : ∀a. a → a → Bool = "(=)"

let vary = fun e y ->
  match e with
  | EquL, EquL -> eq y "c"
  | EquR, EquR -> Cons (match y with True -> 5 | False -> 7)
```

Although `vary` has multiple types, it is a contrived example unlikely to have an intended type. However, not all cases of failure to infer a type for a correct program are due to contrived examples. The problems are not insurmountable theoretically. The algorithms used in the inference can incorporate heuristics for special cases, and can be modified to do a more exhaustive search.

The example `pointwise_head.gadt` fails because of the limitations of the `type` sort in representing disequalities.

```
datatype Z
datatype S : type
datatype List : type * num
datacons LNil : ∀a. List(a, Z)
datacons LCons : ∀a, b. a * List(a, b) ⟶ List(a, S b)

let head = function
  | LCons (x, _) -> x
  | LNil -> assert false
```

If we omit the `LNil` branch, we get the technically correct but inadequate type `∀a, b. List(a, b) → a`, because the type system does not guarantee exhaustiveness of the pattern matching. The intended type is `∀a, b. List(a, S b) → a`.

The example `non_pointwise_fd_comp_harder.gadt` is inferred an insufficiently general type `∀a, b. FunDesc (b, b) → FunDesc (b, a) → FunDesc (b, a)`.

```
datatype FunDesc : type * type
datacons FDI : ∀a. FunDesc (a, a)
datacons FDC : ∀a, b. b ⟶ FunDesc (a, b)
datacons FDG : ∀a, b. (a → b) ⟶ FunDesc (a, b)
external fd_fun : ∀a, b. FunDesc (a, b) → a → b

let fd_comp fd1 fd2 =
  let o f g x = f (g x) in
  match fd1 with
    | FDI -> fd2
    | FDC b ->
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC (fd_fun fd2 b)
        | FDG g -> FDC (fd_fun fd2 b))
```

```
    | FDG f ->
      (match fd2 with
        | FDI -> fd1
        | FDC c -> FDC c
        | FDG g -> FDG (o (fd_fun fd2) f))
```

This happens because the second argument `fd2` is not expanded when `fd1` is equal to `FDI`. Type inference cannot carry out the different reasoning steps leading to the more general type.

In the example `liquid_bsearch2_harder4.gadt` it turns out to be too hard to infer the full postcondition.

```
datatype Array : type * num
external let array_make :
  ∀n, a [0⩽n]. Num n → a → Array (a, n) = "fun a b -> Array.make a b"
external let array_get :
∀n, k, a [0⩽k ∧ k+1⩽n]. Array (a, n) → Num k → a =
  "fun a b -> Array.get a b"
external let array_length :
  ∀n, a [0⩽n]. Array (a, n) → Num n = "fun a -> Array.length a"
datatype LinOrder
datacons LE : LinOrder
datacons GT : LinOrder
datacons EQ : LinOrder
external let compare : ∀a. a → a → LinOrder =
  "fun a b -> let c = Pervasives.compare a b in
              if c < 0 then LE else if c > 0 then GT else EQ"
external let equal : ∀a. a → a → Bool = "fun a b -> a = b"
external let div2 : ∀n. Num (2 n) → Num n = "fun x -> x / 2"

let bsearch key vec =
  let rec look key vec lo hi =
    eif lo <= hi then
        let m = div2 (hi + lo) in
        let x = array_get vec m in
        ematch compare key x with
          | LE -> look key vec lo (m + (-1))
          | GT -> look key vec (m + 1) hi
          | EQ -> eif equal key x then m else -1
    else -1 in
  look key vec 0 (array_length vec + (-1))
```

We get the result type $\exists$`n[0 ⩽ n + 1].Num n` instead of $\exists$`k[k ⩽ n ∧ 0 ⩽ k + 1].Num k`. The inference of the intended type succeeds after we introduce an appropriate assertion, e.g. `assert num -1 <= hi`. Alternatively, we could include a use case for `bsearch` where the full postcondition is required.

Examples `liquid_simplex_step_3.gadt`, `liquid_simplex_step_4.gadt` and `liquid_gauss_rowMax.gadt` result in uninformative, empty postconditions, because to tell more would require inspecting the behavior of the respective function across recursive calls. To save space, we list just the function definition from `liquid_simplex_step_3.gadt`:

```
let rec enter_var arr2 n j c j' =
  eif j' + 2 <= n then
    let c' = matrix_get arr2 0 j' in
    eif less c' c then enter_var arr2 n j' c' (j'+1)
    else enter_var arr2 n j c (j'+1)
  else j
```

Fortunately, if the function is used in the same toplevel definition in which it is defined, use-site requirements facilitate the inference of the intended postcondition.

The example `liquid_gauss_harder.gadt` poses too big a challenge for InvarGenT. To get it pass the inference, we streamline one of the nested definitions, to not introduce another, unnecessary level of nesting. This gives the example `liquid_gauss2.gadt`, which needs to be run with the option `-prefer_bound_to_local`. Additionally, we can relax the constraint on the processed portion of the matrix, coming from the restriction on the matrix size intended in the original source of the `liquid_gauss_harder.gadt` example. In `liquid_gauss.gadt`, the whole matrix is processed and the inferred type is most general, under the default settings – no need to pass any options to InvarGenT. The reason `liquid_gauss_harder.gadt` is too difficult for InvarGenT is that the nesting interferes with the propagation of use-site constraints to the postcondition of the nested definition (the `loop` inside `rowMax`). Inference works for `liquid_gauss_harder_asserted.gadt`, because the assertion provides the required information to infer the `rowMax` invariants directly.