Constraint-Based Type Inference for Guarded Algebraic Data Types

Vincent Simonet, François Pottier

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Constraint-Based Type Inference for Guarded Algebraic Data Types

Vincent Simonet — François Pottier

N° 5462

Janvier 2005

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--5462--FR+ENG

Constraint-Based Type Inference for Guarded Algebraic Data Types

Vincent Simonet , François Pottier

Thème SYM — Systèmes symboliques
Projet Cristal

Rapport de recherche n° 5462 — Janvier 2005 — 65 pages

Abstract: *Guarded* algebraic data types subsume the concepts known in the literature as *indexed types*, *guarded recursive datatype constructors*, and *first-class phantom types*, and are closely related to *inductive types*. They have the distinguishing feature that, when typechecking a function defined by cases, every branch may be checked under different assumptions about the type variables in scope. This mechanism allows exploiting the presence of dynamic tests in the code to produce extra static type information.

We propose an extension of the constraint-based type system HM(X) with deep pattern matching, guarded algebraic data types, and polymorphic recursion. We prove that the type system is sound and that, provided recursive function definitions carry a type annotation, type inference may be reduced to constraint solving. Then, because solving arbitrary constraints is expensive, we further restrict the form of type annotations and prove that this allows producing so-called *tractable* constraints. Last, in the specific setting of equality, we explain how to solve tractable constraints.

To the best of our knowledge, this is the first *generic* and *comprehensive* account of type inference in the presence of guarded algebraic data types.

Key-words: guarded algebraic data types, typechecking, constraints, pattern matching, type inference

Inférence de types à base de contraintes pour les types de données algébriques gardés

Résumé : Les types de données algébriques *gardés* généralisent les concepts connus dans la littérature sous les noms de *types indexés*, *constructeurs de types de données récursifs gardés* et *types fantômes de première classe*, et sont intimement liés aux *types inductifs*. Ils ont pour trait caractéristique le fait que, lors du typage d'une fonction définie par cas, chaque branche peut être typée sous des hypothèses différentes à propos des variables de types connues. Ce mécanisme permet d'exploiter la présence de tests dynamiques dans le code pour produire une information de typage statique supplémentaire.

Nous proposons une extension du système de types à base de contraintes HM(X) avec filtrage profond, types de données algébriques gardés et récursivité polymorphe. Nous démontrons que ce système de types est sûr et que, pourvu que les définitions de fonctions récursives soient annotées par un schéma de types, l'inférence de types se réduit à la résolution de contraintes. Ensuite, parce que la résolution de contraintes arbitraires est coûteuse, nous restreignons la forme des annotations autorisées et démontrons que cela permet de produire des contraintes dites *gérables*. Enfin, dans le cas particulier de l'égalité, nous expliquons comment résoudre les contraintes gérables.

À notre connaissance, ceci est le premier traité *générique* et *exhaustif* de l'inférence de types en présence de types de données algébriques gardés.

Mots-clés : types de données algébriques gardés, typage, contraintes, filtrage, inférence de types

Contents

1 Introduction

Programming languages in the ML family are equipped with *algebraic data types* and with *pattern matching*, which provide high-level facilities for defining and manipulating data structures. They also have *type inference* in the style of Hindley (1969) and Milner (1978), keeping mandatory type annotations to a minimum. The purpose of the present paper is to conservatively extend these languages with *guarded* algebraic data types. Type inference should be preserved: while programs that exploit guarded algebraic data types may require some type annotations, existing ML programs must not. In fact, for much greater generality, we extend not only ML, but the generic constraint-based type system HM(X) (Odersky et al., 1999).

This introduction begins with a description of guarded algebraic data types (§1.1) and a review of some of their applications (§1.2), provides some details about our approach and a comparison with related work (§1.3), and ends with an outline of the paper (§1.4).

1.1 From algebraic data types to guarded algebraic data types

Let us first recall how algebraic data types are defined, and explain the more general notion of guarded algebraic data types.

Algebraic data types Let ε be an algebraic data type, parameterized by a vector of distinct type variables $\bar{\alpha}$. Let K be one of the data constructors associated with ε. The (closed) type scheme assigned to K, which may be derived from the declaration of ε, must be of the form

$$K :: \forall\bar{\alpha}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}), \tag{1}$$

where n is the arity of K. Then, the typing discipline for pattern matching may be summed up as follows: if the pattern $K\,x_1 \cdots x_n$ matches a value of type $\varepsilon(\bar{\alpha})$, then the variable x_i becomes bound to a value of type τ_i.

For instance, an algebraic data type $\mathit{tree}(\alpha)$, describing binary trees whose internal nodes are labeled with values of type α, might consist of the following data constructors:

$$\begin{aligned} &\mathit{Leaf} :: \forall\alpha.\mathit{tree}(\alpha),\\ &\mathit{Node} :: \forall\alpha.\mathit{tree}(\alpha) \cdot \alpha \cdot \mathit{tree}(\alpha) \to \mathit{tree}(\alpha). \end{aligned}$$

The arities of *Leaf* and *Node* are respectively 0 and 3 (we use the symbol $\cdot$ to separate the types of constructor's arguments). Matching a value of type $\mathit{tree}(\alpha)$ against the pattern *Leaf* binds no variables. Matching such a value against the pattern $\mathit{Node}(l, v, r)$ binds the variables l, v, and r to values of types $\mathit{tree}(\alpha)$, α, and $\mathit{tree}(\alpha)$, respectively.

Läufer-Odersky-style existential types It is possible to imagine extensions of ML that allow more liberal forms of algebraic data type declarations. Consider, for instance, Läufer and Odersky's extension of ML with existential types (1994). There, the type scheme associated with a data constructor may be of the form

$$K :: \forall\bar{\alpha}\bar{\beta}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}). \tag{2}$$

The novelty resides in the fact that the argument types $\tau_1 \cdots \tau_n$ may contain type variables, namely $\bar{\beta}$, that are not parameters of the algebraic data type constructor ε. Then, the typing discipline

for pattern matching becomes: if the pattern $K\,x_1 \cdots x_n$ matches a value of type $\varepsilon(\bar{\alpha})$, then *there exist* unknown types $\bar{\beta}$ such that the variable x_i becomes bound to a value of type τ_i.

For instance, an algebraic data type *key*, describing pairs of a key and a function from keys to integers, where the type of keys remains abstract, might be declared as follows:

$$Key :: \forall\beta.\beta \cdot (\beta \to int) \to key.$$

The values $Key\,(3, \lambda x.5)$ and $Key\,([1;2;3], length)$ both have type *key*. Matching either of them against the pattern $Key\,(v, f)$ binds the variables v and f to values of type β and $\beta \to int$, for a fresh β, which allows, say, evaluating $(f\,v)$, but prevents viewing v as an integer or as a list of integers—either of which would be unsafe.

Guarded algebraic data types Let us now go one step further by allowing data constructors to be assigned *constrained* type schemes:

$$K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}). \tag{3}$$

Here, D is a constraint, that is, a first-order formula built out of a fixed set of predicates on types. The value $K\,(v_1, \ldots, v_n)$, where every v_i has type τ_i, is well-typed, and has type $\varepsilon(\bar{\alpha})$, only if the type variables $\bar{\alpha}\bar{\beta}$ satisfy the constraint D. In exchange for this restricted *construction* rule, the typing discipline for *destruction*—that is, pattern matching—becomes more flexible: if the pattern $K\,x_1 \cdots x_n$ matches a value of type $\varepsilon(\bar{\alpha})$, then there exist unknown types $\bar{\beta}$ *that satisfy* D such that the variable x_i becomes bound to a value of type τ_i. Thus, the success of a *dynamic* test, namely pattern matching, now allows extra *static* type information, expressed by D, to be recovered within the scope of a branch. We defer a more concrete illustration of this mechanism to §2. We refer to this flavor of algebraic data types as *guarded*, because their data constructors have guarded (constrained) type schemes.

1.2 Applications of guarded algebraic data types

Guarded algebraic data types are a fairly general concept. This is due, in particular, to the fact that the constraint language in which D is expressed is not fixed *a priori*. On the contrary, many choices are possible; let us suggest a few.

Unification In the simplest case, D must be a unification constraint, that is, a conjunction of type equations. Then, the data constructors associated with guarded algebraic data types may in fact be assigned type schemes of the form

$$K :: \forall\bar{\beta}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\tau}). \tag{4}$$

In this form, no constraint is specified, but the type constructor ε may be applied to a vector of arbitrary types, rather than to a vector of distinct type variables $\bar{\alpha}$. It is not difficult to check that the forms (3) and (4) are equivalent. On the one hand, a declaration of the form (4) may be written

$$K :: \forall\bar{\alpha}\bar{\beta}[\bar{\alpha} = \bar{\tau}].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}),$$

an instance of (3). Conversely, because every satisfiable unification constraint admits a most general unifier, a declaration of the form (3) either has no instance (making K inapplicable, a pathological case) or may be written under the form (4); we omit the details.

We sometimes refer to data types introduced by declarations of the form (4) as *inductive types* because they are strongly reminiscent of those found in the Calculus of Inductive Constructions (Paulin-Mohring, 1992; Werner, 1994). (The only difference lies in the fact that true inductive types come with a positivity restriction, which ensures logical consistency, whereas, in a programming-language setting, no such restriction is necessary.) They are also referred to as *guarded recursive datatype constructors* by Xi et al. (2003) and as *first-class phantom types* by Cheney and Hinze (2003). An instance of this mechanism was earlier proposed by Crary, Weirich and Morrisett (2002). A basic example of their use is given in §2.1.

Arithmetic Assume that, in addition to type equations, the constraint language contains a decidable fragment of first-order arithmetic, such as Presburger arithmetic. Then, constraints contain variables of two kinds: variables of the first kind stand for types, while variables of the second kind stand for integers. In such a setting, guarded algebraic data types are of great interest.

For instance, one may declare the type of integers, *int*, as a unary guarded algebraic data type constructor, whose parameter is of integer kind, and whose data constructors are the integer constants:

$$k :: \forall\alpha[\alpha = k].\mathit{int}(\alpha) \qquad (k \in \mathbb{Z})$$

This is usually known as a *singleton type*, because only the data constructor k has type $\mathit{int}(k)$.

Remark 1.1. One should note that, although it is pleasant to view *int* as an algebraic data type, this only leads us half of the way. Indeed, because pattern matching for algebraic data types only allows matching an unknown term against a *fixed* pattern, it only allows encoding equality tests between an integer variable and a constant. Ordering comparisons, or comparisons between two integer variables, still require a special-purpose extension of the programming language. A similar remark applies to the singleton type constructor *level* in §2.3, where ordering comparisons between two level variables are needed.

In the present paper, we only deal with the usual notion of pattern matching for algebraic data types, and do not consider these additional elimination constructs for singleton types. The basic typechecking and type inference techniques that they require are the same. ◇

To go further, one may declare the type of lists, *list*, as a binary guarded algebraic data type constructor, whose parameters are respectively of type and integer kinds:

$$\begin{aligned} \mathit{Nil} &:: \forall\alpha\beta[\beta = 0].\mathit{list}(\alpha, \beta) \\ \mathit{Cons} &:: \forall\alpha\beta_1\beta_2[1 + \beta_1 = \beta_2].\alpha \cdot (\mathit{list}(\alpha, \beta_1)) \to \mathit{list}(\alpha, \beta_2) \end{aligned}$$

The idea is that, while the parameter α is the type of the elements of the list, the parameter β reflects the length of the list, so that only lists of length k have type $\mathit{list}(\tau, k)$. Type systems that allow dealing with lists in this manner include Zenger's *indexed types* (Zenger, 1997, 1998) as well as Xi and Pfenning's so-called *dependent types* (Xi, 1998; Xi and Pfenning, 1999). Both are constraint-based type systems, where typechecking relies on the decidability of constraint entailment, as opposed to true dependent type systems, where typechecking involves deciding *term* equality. The use of indexed types is further illustrated in §2.2.

Subtyping Guarded algebraic data types subsume the *guarded existential types* studied by the first author (Simonet, 2003a) with information flow analysis in mind. In this case, types contain atomic security levels, which belong to a fixed lattice, and induce a *structural subtyping* relation. More details are given in §2.3.

1.3 Extending ML with guarded algebraic data types

We have tried to convince the reader that guarded algebraic data types are a very general notion and subsume several existing type-theoretic mechanisms. Their usefulness is two-fold. On the one hand, increasing the expressiveness of the type system means *accepting* more programs, thus making new programming idioms available; see, for instance, how inductive types are put to use (Xi et al., 2003; Cheney and Hinze, 2003; Pottier and Gauthier, 2004). On the other hand, it also allows more precise type specifications to be written, thus *rejecting* some (safe, but incorrect) programs, in the spirit of *refinement types* (Freeman and Pfenning, 1991; Zenger, 1997, 1998; Xi, 1998; Xi and Pfenning, 1999). Thus, it is tempting to augment ML with guarded algebraic data types, and to do so in a *generic* fashion, so that all of the applications mentioned in §1.2 become instances of a common framework. This is the purpose of the present paper.

One must be careful, however, not to pursue contradictory goals. On the one hand, inductive types form a *strict extension* of ML, in the sense that every well-typed ML program remains well-typed in their presence, and some new programs become well-typed. On the other hand, refinement types *refine* ML, in the sense that every program that is well-typed in their presence is a well-typed ML program in the first place. These properties are mutually exclusive. Because we aim to include inductive types, we must aim for a strict extension of ML. As a result, although our type system also includes indexed types in the style of Zenger and dependent types in the style of Xi and Pfenning, it does not refine ML in the above sense.

In the next few paragraphs, we give more details about this issue and motivate our work by comparing it with related efforts.

In Zenger's work (1997; 1998), there is an important distinction between *indices*, which are taken from some abstract domain, and *types*, which are first-order terms that carry indices. The logical interpretation of constraints is defined in a careful way, so as to ensure that the type system restricts ML. More specifically, with every constraint, Zenger associates *two* interpretations (Zenger, 1998, Definition 19). The former tells whether the constraint is satisfied, as usual, while the latter tells whether it is *structurally consistent*, that is, roughly speaking, whether it would be satisfied if all indices were erased. The former depends on the latter, as follows: for an implication $C_1 \Rightarrow C_2$ to be satisfied, not only must the satisfaction of C_1 imply that of C_2, as usual, but also must the constraint C_2 be structurally consistent, *regardless of whether* C_1 *is satisfied*. As a result of this definition, a constraint of the form $C \Rightarrow \alpha_1 = \mathit{list}(\alpha_2, \beta)$ is equivalent to $\exists \alpha'_2 \beta'.(\alpha_1 = \mathit{list}(\alpha'_2, \beta') \wedge C \Rightarrow (\alpha_2 = \alpha'_2 \wedge \beta = \beta'))$—that is, α_1 must be a list, regardless of whether C_1 is satisfied.

In the present paper, we cannot follow Zenger's approach, because *inductive types*, as described above, require implication to be interpreted in a standard way. Indeed, in the setting of inductive types, there are no indices: we are dealing with equations between standard ML types. Thus, Zenger's interpretation of implication would equate $C_1 \Rightarrow C_2$ with C_2, effectively disallowing the use of implication and making guarded algebraic data types useless. For instance, in the case of `print` (§2.1), the two branches would *unconditionally* be required to produce values of the same

type, leading to a type error, exactly as in ML. For this reason, we adopt a *standard* interpretation of implication, and employ constraint solving algorithms that must differ from Zenger's. As a result, our type system is not a refinement type system—that is, it does not restrict ML: in fact, it extends it. Every program that is well-typed in Zenger's system is also well-typed in ours, *with the same type*; however, we accept more programs, because of our more liberal interpretation of implication.

Some other differences between Zenger's work and ours is that Zenger does not allow subtyping or nested patterns, while we suppress these restrictions. In spite of these differences, Zenger's work remains very close in spirit to ours: in particular, type inference is reduced to constraint solving.

The type system DML(C) (Xi, 1998; Xi and Pfenning, 1999), which was developed independently, is a close cousin of Zenger's. It entertains a similar distinction between indices and types, and interprets implication in a similar manner (Xi, 1998, Section 4.1.1), so as to refine ML. Its implementation, Dependent ML (Xi, 2001), includes a type inference process, known as *elaboration* (Xi, 1998, Section 4.2). Because Dependent ML features first-class universal and existential types, the elaboration algorithm isn't an extension of ML's type inference algorithm. Instead, it is *bidirectional*: it alternates between *type verification* and *type synthesis* phases. For this reason, some well-typed ML programs appear to require extra type annotations before they can be accepted by Dependent ML. One may view this fact as a misfeature. Like Zenger's, Xi's elaboration algorithm does, in the end, produce a constraint, whose satisfiability must then be determined.

Previous accounts of guarded algebraic data types are written with *type checking*, as opposed to *type inference*, in mind. This includes Cheney and Hinze's (2003), Xi et al.'s (2003), as well as Xi's *applied type system* (2004). The latter, which was written concurrently with the present paper, is nevertheless interesting, because of its generality: it removes the distinction between indices and types, and keeps only (multiple sorts of) types; furthermore, it is parameterized with an arbitrary (decidable) constraint domain. These are also features of the type system presented in this paper.

To sum up, we are interested in extending ML with a general form of guarded algebraic data types, so as to encompass all of the applications described in §1.2 in a single, abstract framework. Furthermore, we aim to extend ML's type inference discipline in a conservative manner, so that existing ML programs remain well-typed without requiring extra type annotations. To achieve these goals, we start with the generic type system HM(X) (Odersky et al., 1999), a constraint-based extension of ML, which in fact coincides with ML when constraints are made up of type equations only. We enrich HM(X) with guarded algebraic data types, pattern matching, and polymorphic recursion, and refer to this extension as HMG(X). Type inference for HMG(X) is reduced to constraint solving in such a way that, for programs that do not involve guarded algebraic data types, constraint generation is entirely standard.

In contrast with DML(C) (Xi, 1998; Xi and Pfenning, 1999), HMG(X) does not have first-class existential or universal types with *implicit* introduction and elimination forms. HMG(X) does have first-class existential types, since they can be encoded using guarded algebraic data types (§1.1). However, in this encoding, creating an existential package amounts to applying a data constructor, while opening it amounts to performing case analysis; both operations must be made *explicit* in the program. First-class universal types with explicit introduction and elimination forms could be added to HMG(X) in the same manner; see, for instance, Rémy (1994), Jones (1995), Ordersky and Läufer (1996), or Simonet (2003a).

1.4 Road map

The paper is laid out as follows. We first illustrate several instances of HMG(X), namely: inductive types (§2.1), indexed types (§2.2), and *dynamic security levels* (§2.3) in the context of an information flow analysis—a novel application, which involves structural subtyping. Our formal development begins with a presentation of the untyped calculus, a call-by-value λ-calculus featuring data constructors and pattern matching (§3). We then define the type system HMG(X), and establish subject reduction and progress theorems (§4). The constraint-based nature of the system makes it easy to reason about type inference: we show that, provided recursive function definitions carry a type annotation, type inference may be reduced to constraint solving (§5). Then, because solving arbitrary constraints is expensive, we further restrict the form of type annotations and prove that this allows producing so-called *tractable* constraints (§6.1). Last, in the specific setting of equality, we explain how to solve tractable constraints (§6.2).

2 Examples

We now give concrete program fragments that are well-typed in several distinct instances of HMG(X), obtained by specializing the constraint logic X. This highlights the usefulness and versatility of guarded algebraic data types. All program fragments but one (`rmap_f` in §2.2) carry sufficient type annotations to be accepted by the type inference algorithm of §6. The syntax of our examples mimics that of Objective Caml (Leroy et al., 2004). We assume that a few standard library functions, such as `print_int` and `print_string`, are available.

2.1 Inductive types

We first illustrate how *inductive types*, introduced by Xi et al. under the name *guarded recursive datatypes* (Xi et al., 2003), are dealt with in HMG(X). Here, types are interpreted as finite trees, and constraints are *type equations*, as in constraint-based presentations of ML. We assume that the type constructors for integers and (binary) pairs, *int* and $\times$, are declared as in ML (they are algebraic data type constructors, too). Then, following Crary et al. (2002) and Xi et al. (2003), we introduce a unary guarded algebraic data type constructor *ty*, and associate the following data constructors with it:

$$\begin{aligned} \mathit{Int} &:: \forall\alpha[\alpha = \mathit{int}].\mathit{ty}(\alpha) \\ \mathit{Pair} &:: \forall\alpha\beta_1\beta_2[\alpha = \beta_1 \times \beta_2].\mathit{ty}(\beta_1) \cdot \mathit{ty}(\beta_2) \to \mathit{ty}(\alpha) \end{aligned}$$

This declaration is of the form (3) (§1.1), which is the form used in our theory. In concrete syntax, one could (and should) allow programmers to use the form (4) (§1.2), that is, to write:

$$\begin{aligned} \mathit{Int} &:: \mathit{ty}(\mathit{int}) \\ \mathit{Pair} &:: \forall\beta_1\beta_2.\mathit{ty}(\beta_1) \cdot \mathit{ty}(\beta_2) \to \mathit{ty}(\beta_1 \times \beta_2) \end{aligned}$$

For the sake of brevity, we associate only two data constructors, *Int* and *Pair*, with the type constructor *ty*. In a practical application, it could have more.

The motivation for this definition is the following: *we may interpret* $\mathit{ty}(\tau)$ *as a singleton type whose only value is a runtime representation of* τ (Crary et al., 2002). The constraints carried by the above declarations capture the relationship between the structure of a value v whose type is

$ty(\tau)$ and that of the type τ. Indeed, if v is *Int*, then τ must be *int*; if v is $Pair\, v_1\, v_2$, then τ must be of the form $\tau_1 \times \tau_2$, where v_i has type $ty(\tau_i)$. Thus, by examining v, one may gain knowledge about τ. This is particularly useful when τ is a type variable, or has free type variables: the branches of a `match` construct that examines v may be typechecked under additional assumptions about these type variables. This is illustrated by the definition of `print`, a generic printing function:

```
let rec print : ∀α.ty(α) → α → unit = fun t ->
  match t with
  | Int ->
      fun x -> print_int x
  | Pair (t1, t2) ->
      fun (x1, x2) -> print t1 x1; print_string " * "; print t2 x2
```

For an arbitrary α, `print` accepts a runtime representation of the type α, that is, a value `t` of type $ty(\alpha)$, as well as a value `x` of type α, and prints out a human-readable representation of `x`. The function first examines the structure of `t`, so as to gain knowledge about α. Indeed, each branch is typechecked under additional static knowledge. For instance, in the first branch, the assumption $\alpha = int$ is available, so that `x` may be passed to the standard library function `print_int`, which has type $int \to unit$. Similarly, in the second branch, we have $\alpha = \beta_1 \times \beta_2$, where β_1 and β_2 are abstract, so that `x` must in fact be a pair `(x1, x2)`.

In the second branch, `print` recursively invokes itself in order to display `x1` and `x2`. There is a need for polymorphic recursion: the recursive calls to `print` use two different instances of its type scheme, namely $\beta_i \to ty(\beta_i) \to \beta_i \to unit$, for $i \in \{1, 2\}$. In the presence of polymorphic recursion, type inference is known to be undecidable, unless an explicit type annotation is given (Henglein, 1993). For this reason, the type scheme $\forall\alpha.ty(\alpha) \to \alpha \to unit$ must be explicitly supplied by the programmer. The constraint generation algorithm given in §6 also exploits this type annotation in order to analyze the `match` construct, which eliminates a guarded algebraic data type. Thus, in this case, a single type annotation suffices. We believe this pattern to be common.

Let us go on with a generic comparison function, which is similar to `print`, but simultaneously analyzes *two* runtime type representations:

```
let rec equal : ∀αβ.ty(α) → ty(β) → α → β → bool = fun t u ->
  match t, u with
  | Int, Int ->
      fun x y -> x = y
  | Pair (t1, t2), Pair (u1, u2) ->
      fun (x1, x2) (y1, y2) -> (equal t1 u1 x1 y1) && (equal t2 u2 x2 y2)
  | _, _ ->
      false
```

The values `x` and `y` are structurally compared if they have the same type, otherwise they are considered different. More generally, it is possible to define a *type cast* operator that relies on a *dynamic* comparison between type representations:

```
let rec cast : ∀αβ.ty(α) → ty(β) → α → β = fun t u ->
  match t, u with
  | Int, Int ->
      fun x -> x
  | Pair (t1, t2), Pair (u1, u2) ->
      fun (x1, x2) -> (cast t1 u1 x1, cast t2 u2 x2)
```

```
| _, _ ->
    raise CastException
```

If the type representations `t` and `u` match, then `cast t u x` returns the value `x`, otherwise it raises the exception `CastException`. A more elaborate version of such a cast operator has been suggested by Weirich (2000).

More examples of the use of inductive types are given by Xi et al. (2003) and by Cheney and Hinze (2003). Furthermore, some recent works suggest not individual examples, but general programming patterns that exploit inductive types. For instance, Pottier and Gauthier (2004) show that inductive types allow expressing defunctionalization. This means that some ML programs whose well-typedness crucially relies on the use of higher-order functions, such as Danvy's version of `printf` (1998), can now be modified to use algebraic data structures instead. This is important, because data structures are often easier to reason about than first-class functions, and are less opaque: a function can only be applied, while a data structure may be analyzed, traversed, modified, etc. in more than one way.

The type system described here is an instance of HMG(X): in fact, it is its simplest instance. The constraint generation and constraint solving algorithms given in this paper apply.

2.2 Indexed types

We now consider so-called *indexed* types. The constraint language is that of 2.1, extended with a decidable fragment of first-order arithmetic. We use the binary guarded algebraic data type constructor *list* introduced in §1.2. Following Objective Caml syntax, the data constructors *Nil* and *Cons* are written `[]` and `::`.

Our first example is `combine`, a function that transforms a pair of lists of matching length into a list of pairs. It is taken from Objective Caml's standard library.

```
let rec combine : ∀αβγ.list(α,γ) → list(β,γ) → list(α × β,γ) = fun l1 l2 ->
  match l1, l2 with
  | [], [] ->
      []
  | a1 :: l1, a2 :: l2 ->
      (a1, a2) :: (combine l1 l2)
```

The type scheme supplied by the programmer specifies that the two input lists must have the same length, represented by the variable γ, and that the list that is returned has the same length as well.

It is worth noting that the implementation of `combine` in Objective Caml's standard library includes an additional safety clause:

```
| _, _ ->
    invalid_arg "List.combine"
```

which is executed when `combine` is applied to lists of incompatible lengths. Here, this clause is unnecessary. Indeed, because of the type scheme that was explicitly assigned to `combine`, this clause is typechecked under inconsistent assumptions: in other words, the type system is able to prove that it is dead code, and, for this reason, allows it to be omitted. More details are given in §4.7.

Our second example is `rev_map`, a function that reverses a list and at the same time applies a transformation to each of its elements, also taken from Objective Caml's standard library.

```
let rev_map f l =
  let rec rmap_f : ∃αβ.∀γ1γ2.list(β,γ1) → list(α,γ2) → list(β,γ1+γ2) =
    fun accu -> function
    | [] ->
        accu
    | a :: l ->
        rmap_f ((f a) :: accu) l
  in
  rmap_f [] l
```

The principal type scheme of `rev_map` is $\forall\alpha\beta\gamma.(\alpha \to \beta) \to \mathit{list}(\alpha,\gamma) \to \mathit{list}(\beta,\gamma)$, which reflects that the function's input and output lists have the same length.

Here, the internal auxiliary function `rmap_f` must be explicitly annotated, because it involves polymorphic recursion, and because it involves pattern matching over a guarded algebraic data type. However, it does not have a closed type scheme: the variables α and β, which we have existentially quantified in the type annotation, occur in the type of `f`, and cannot be universally quantified.

For this reason, this code is *not* accepted by the simple type inference algorithm of §6, which essentially requires annotations to be *closed* type schemes, even though it is well-typed under the rules of HMG(X). Currently, the only workaround is to modify the code so that `f` is passed as an argument to `rmap_f`, which may incur a runtime penalty. This illustrates a situation where requiring annotations to be closed type schemes is too restrictive. So far, we believe such a situation to be uncommon, but this remains to be backed with experience.

The type system described here is an instance of HMG(X). The constraint generation algorithm given in this paper applies. We do not address constraint solving, but we believe it is possible to reduce it to solving arithmetic formulæ, as in Xi's works (1998; 2001). The reduction is not quite the same as Xi's, because our interpretations of implication differ (§1.3).

2.3 Dynamic security levels

Our last series of examples introduces a novel use of guarded algebraic data types. The type system discussed here is not exactly an instance of HMG(X): it combines the treatment of guarded algebraic data types, as found in HMG(X), with an information flow analysis for ML, as described elsewhere by the authors (Pottier and Simonet, 2003) and implemented in Flow Caml (Simonet, 2003b). Our point is that guarded algebraic data types, together with dedicated forms of pattern matching, allow modeling *dynamic security levels*.

Because information flow analysis is not the central topic of this paper, our exposition remains informal and omits many details that are relevant to the correctness of the analysis, but not the type inference mechanism. (In particular, the type constructor *level*, which is introduced below, should carry not one, but two, parameters.)

As usual in information flow analyses, types carry *security levels*, which are either atoms, taken from some fixed lattice $\mathcal{L}$, or variables. (Thus, security levels are types, of a distinguished kind.) For instance, the integer type constructor *int* is now unary and covariant: its parameter is of security level kind. (It should not be confused with the singleton type *int* of §1.2, whose parameter

is of integer kind.) The ordering on atoms gives rise to an ordering on types, which is referred to as *structural subtyping*, because any two comparable types must have the same structure and may differ only in their security levels.

The lattice of security levels may be used to model principals and sets thereof. For instance, assume we are writing software for a bank. Then, each of the bank's clients may be represented by a distinct security level (alice, bob, ...). The set of all European clients and the set of all clients worldwide may also be encoded as security levels, say europe and world. Then, the security lattice is defined so as to reflect membership in these sets: for instance, alice $\leq$ world and europe $\leq$ world hold.

Dynamic security levels (Myers, 1999) are runtime representations of security levels. They allow decisions about the dissemination of information to be made at runtime, while maintaining a strong security guarantee. In the bank example, dynamic security levels are useful because they allow building heterogeneous data structures, such as a list of client records, where every element has a different security level. In the absence of dynamic security levels, all elements of a list must have the same security level.

Zheng and Myers (2004) explain dynamic security levels in terms of dependent types, while Tse and Zdancewic (2004) study the closely related notion of *dynamic principals* using singleton types. The technique that we suggest here is also based on singleton types, which guarded algebraic data types subsume, as pointed out in §1.2. It is very close to Tse and Zdancewic's work.

Let us introduce a unary guarded algebraic data type constructor *level*, whose data constructors are the constant security levels ℓ, that is, all elements of the security lattice $\mathcal{L}$:

$$\ell :: \forall\alpha[\alpha = \ell].level(\alpha) \qquad (\ell \in \mathcal{L})$$

We require *level* to be an *invariant* type constructor: that is, $level(\alpha) \leq level(\alpha')$ entails $\alpha = \alpha'$. Then, for every $\ell \in \mathcal{L}$, only one value has type $level(\ell)$, namely the constant security level ℓ. Thus, *level* is a singleton type constructor, just like *int* in §1.2.

Back to the bank example, a client record may be represented using the guarded algebraic data type *record*, whose only data constructor, *Record*, is declared as follows:

$$Record :: \forall\beta[\beta \leq \mathsf{world}].int(\beta) \cdot level(\beta) \rightarrow record$$

The type variable β represents the security level associated with this client. It is *not* a parameter of the type constructor *record*. As a result, the type *record* is isomorphic to the bounded existential type $\exists\beta[\beta \leq \mathsf{world}].int(\beta) \times level(\beta)$. A value of type *record* contains two pieces of data, namely the account's balance, an integer value, and a dynamic security level, which allows telling, at runtime, which principals should have access to this information.

Because the type *record* isn't parameterized, it is possible to build a list of client records, whose type is $list(record)$. This list is heterogeneous, in the sense that every record may contain an integer value of a different security level. Thanks to the use of an existential type, the type system views it as a homogeneous list. A function that accepts such a list and computes the total amount of money available to the bank may be written as follows:

```
let rec total : list(record) → int(world) = function
  | [] ->
      0
  | (Record (x, id)) :: records ->
      x + total records
```

This function is well-typed, even though the security level of the integer `x` is an abstract type variable β, because β carries the bound $\beta \leq \mathsf{world}$, so `x` also has type $\mathit{int}(\mathsf{world})$, and the addition may be performed at this type. The function's return type $\mathit{int}(\mathsf{world})$ correctly reflects the fact that its result may reveal information about any client.

One may wish to compute the sum over a strict subset of the list, instead of over the entire list of records. One way to do so is to examine the dynamic security level that each record contains. We parameterize the function `total` by a dynamic security level `region`, and include in the sum only those clients whose associated security level is below `region`:

```
let rec subtotal : ∀α[α ≤ world].level(α) → list(record) → int(α) =
  fun region records ->
    match records with
    | [] ->
        0
    | (Record (x, level)) :: records' when level <= region ->
        x + subtotal region records'
    | _ :: records' ->
        subtotal region records'
```

The hypothetical syntax `when level <= region` performs a dynamic comparison between two dynamic security levels. Such a construct is common in type systems with singleton types: for instance, DML (Xi, 1998, 2001) has integer comparisons, and Tse and Zdancewic (2004) allow comparing dynamic principals. Our formal development does not deal with two-way comparisons (see Remark 1.1), but it is straightforward to imagine how they could be added. Here, `level` has type $\mathit{level}(\beta)$, for some abstract β known to satisfy $\beta \leq \mathsf{world}$, while `region` has type $\mathit{level}(\alpha)$. The success of the *dynamic* comparison between `level` and `region` allows the *static* hypothesis $\beta \leq \alpha$ to be made available in the second branch. Because the type constructor *int* is covariant, the variable `x`, which has type $\mathit{int}(\beta)$, also has type $\mathit{int}(\alpha)$, and the addition may be performed at this type. The function's return type, namely $\mathit{int}(\alpha)$, reflects the fact that the function's result may reveal information about clients whose security level is at most α. For instance, applying `subtotal` to the constant security level `europe`, whose type is $\mathit{level}(\mathsf{europe})$, yields a specialized function of type $\mathit{list}(\mathit{record}) \to \mathit{int}(\mathsf{europe})$, whose result reveals information about European clients only.

The type system described here is not an instance of HMG(X), because its typing rules are modified to keep track of information flow, and because two-way comparisons at singleton types, such as `level <= region`, are not part of the pattern language considered in this paper. Nevertheless, the machinery set up in this paper could easily be adapted to define an information flow analysis with dynamic security levels. Type inference would then be reduced to constraint solving in a fragment of the first-order theory of structural subtyping, which has been studied by the first author (Simonet, 2003c). Even though this type system is not an instance of HMG(X), this was one of our motivations for introducing subtyping in HMG(X), instead of focusing on the case of equality.

3 The untyped calculus

This section introduces a call-by-value λ-calculus featuring data constructors and pattern matching. It is entirely standard, hence we do not explain it in depth.

$$
\begin{aligned}
\mathrm{dpv}(0) &= \varnothing \\
\mathrm{dpv}(1) &= \varnothing \\
\mathrm{dpv}(x) &= \{x\} \\
\mathrm{dpv}(p_1 \wedge p_2) &= \mathrm{dpv}(p_1) \uplus \mathrm{dpv}(p_2) \\
\mathrm{dpv}(p_1 \vee p_2) &= \mathrm{dpv}(p_1) = \mathrm{dpv}(p_2) \qquad \text{if } \mathrm{dpv}(p_1) = \mathrm{dpv}(p_2) \\
\mathrm{dpv}(K\, p_1 \cdots p_n) &= \mathrm{dpv}(p_1) \uplus \cdots \uplus \mathrm{dpv}(p_n)
\end{aligned}
$$

Figure 1: The variables defined by a pattern

$$
\begin{aligned}
[1 \mapsto v] &= \varnothing \\
[p_1 \wedge p_2 \mapsto v] &= [p_1 \mapsto v] \otimes [p_2 \mapsto v] \\
[p_1 \vee p_2 \mapsto v] &= [p_1 \mapsto v] \oplus [p_2 \mapsto v] \\
[K\, p_1 \cdots p_n \mapsto K\, v_1 \cdots v_n] &= [p_1 \mapsto v_1] \otimes \cdots \otimes [p_n \mapsto v_n]
\end{aligned}
$$

Figure 2: Extended substitution

3.1 Syntax

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*. Then *patterns*, *expressions*, *clauses*, and *values* are defined as follows:

$$
\begin{aligned}
p &::= 0 \mid 1 \mid x \mid p \wedge p \mid p \vee p \mid K\, \bar{p} \\
e &::= x \mid \lambda \bar{c} \mid K\, \bar{e} \mid e\, e \mid \mu x.v \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \\
c &::= p.e \\
v &::= \lambda \bar{c} \mid K\, \bar{v}
\end{aligned}
$$

Patterns include the empty pattern 0, the wildcard pattern 1, variables, conjunction and disjunction patterns, and data constructor applications. To a pattern p, we associate a set of *defined program variables* $\mathrm{dpv}(p)$, as specified in Figure 1. (The operator $\uplus$ stands for set-theoretic union $\cup$, but is defined only if its operands are disjoint.) The pattern p is considered ill-formed if $\mathrm{dpv}(p)$ is undefined, thus ruling out nonlinear patterns. Expressions include variables, functions, data constructor or function applications, recursive definitions, and local variable definitions. Functions are defined by cases: a λ-abstraction, written $\lambda(c_1, \ldots, c_n)$, 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 data structures, that is, 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.

3.2 Semantics

Whether a pattern p *matches* a value v is defined by an *extended substitution* $[p \mapsto v]$ 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 $[x \mapsto v]$ coincides with the ordinary substitution $[x \mapsto v]$,

$$
\begin{array}{rll}
\lambda(p_1.e_1 \cdots p_n.e_n)\, v \to \bigoplus_{i=1}^{n} [p_i \mapsto v]e_i & & (\beta) \\
\mu x.v \to [x \mapsto \mu x.v]v & & (\mu) \\
\mathsf{let}\, x = v\ \mathsf{in}\ e \to [x \mapsto v]e & & (\text{let}) \\
E[e] \to E[e'] & \text{if } e \to e' & (\text{context})
\end{array}
$$

Figure 3: Operational semantics

$$
\begin{array}{rl}
K\,\bar{p}\,(p_1 \vee p_2)\,\bar{p}' \rightsquigarrow K\,\bar{p}\,p_1\,\bar{p}' \vee K\,\bar{p}\,p_2\,\bar{p}' & \\
(p_1 \vee p_2) \wedge p \rightsquigarrow (p_1 \wedge p) \vee (p_2 \wedge p) & \\
K\,p_1 \cdots p_n \wedge K\,p'_1 \cdots p'_n \rightsquigarrow K\,(p_1 \wedge p'_1) \cdots (p_n \wedge p'_n) & \\
K\,p_1 \cdots p_n \wedge K'\,p'_1 \cdots p'_{n'} \rightsquigarrow 0 & \text{if } K \neq K' \\
K\bar{p}\,0\,\bar{p}' \rightsquigarrow 0 & \\
p \vee 0 \rightsquigarrow p & \\
0 \vee p \rightsquigarrow p & \\
0 \wedge p \rightsquigarrow 0 &
\end{array}
$$

Figure 4: Normalizing patterns

which justifies our abuse of notation. Extended substitution for other pattern forms is defined in Figure 2. Let us briefly review the definition. The pattern 0 matches no value, so $[0 \mapsto v]$ is always undefined. Conversely, the pattern 1 matches every value, but binds no variables, so $[1 \mapsto v]$ 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 $K\,p_1 \cdots p_n$ matches values of the form $K\,v_1 \cdots v_n$ only; it matches such a value if and only if p_i matches v_i for every $i \in \{1, \ldots, n\}$.

The call-by-value small-step semantics, written $\to$, is defined by the rules of Figure 3. It is standard. Rule (β) governs function application and pattern-matching: $\lambda(p_1.e_1 \cdots p_n.e_n)\, v$ reduces to $[p_i \mapsto v_i]e_i$, where i is the least element of $\{1, \ldots, n\}$ 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, whose grammar is the following:

$$E ::= K\,\bar{v}\,[]\,\bar{e} \mid []\,e \mid v\,[] \mid \mathsf{let}\, x = []\ \mathsf{in}\ e$$

3.3 Properties of patterns

We define a notion of *equivalence* between patterns as follows: p_1 and p_2 are equivalent, which we write $p_1 \equiv p_2$, if and only if they give rise to the same extended substitutions, that is, if and only if the functions $[p_1 \mapsto \cdot]$ and $[p_2 \mapsto \cdot]$ coincide.

It is possible to *normalize* a pattern using the reduction rules given in Figure 4, applied modulo associativity and commutativity of $\wedge$, modulo associativity of $\vee$, and under arbitrary contexts. (Note that $\vee$ cannot be considered commutative, since $p_1 \vee p_2 \equiv p_2 \vee p_1$ may not hold when p_1 and p_2 bind variables.) This process is terminating and meaning-preserving:

LEMMA 3.1. *The relation $\rightsquigarrow$ is strongly normalizing.* ◇ Proof on page 48

LEMMA 3.2. *$p_1 \rightsquigarrow p_2$ implies $p_1 \equiv p_2$.* ◇ Proof on page 48

Normalization may be exploited to decide whether a pattern is empty:

LEMMA 3.3. *$p \equiv 0$ holds if and only if $p \rightsquigarrow^\star 0$ holds.* ◇ Proof on page 48

Thus, if a pattern is empty, then one of its normal forms is 0. (In fact, the previous lemmas imply that it then has no normal form other than 0.) The interaction between normalizing patterns and the type system is discussed in §4.7.

4 The type system

We now equip our core calculus with a constraint-based type system featuring guarded algebraic data types. As argued, for instance, by Pottier and Rémy (2005), the introduction of constraints is beneficial for at least two reasons: (i) it is pleasant to reduce type inference to constraint solving, even when working within the basic setting of ML, where constraints involve type equations only; and (ii) this allows easily moving to much more general settings, where constraints may involve type class membership predicates, as in Haskell (Wadler and Blott, 1989), Presburger arithmetic, as in DML (Xi, 1998), polynomials, as in Zenger's work (1997), subtyping, etc. We present our type system as a conservative extension of HM(X) (Odersky et al., 1999), and refer to it as HMG(X).

In §4.1, we introduce the syntactic objects involved in the definition of the type system, namely type variables, types, constraints, type schemes, environments, and environment fragments. All but the last are inherited from HM(X). Environment fragments are a new entity, used to describe the static knowledge that is gained by successfully matching a value against a pattern. In §4.2, we explain how these syntactic objects are interpreted in a logical model.

In order to guarantee type soundness, some requirements must be placed on the model: they are expressed in §4.3. Some of them concern the (guarded) algebraic data types defined by the programmer, so the syntax of algebraic data type definitions is also made precise in §4.3.

In §4.4, we introduce a few tools that allow manipulating environment fragments. Then, we review the typing judgments (§4.5) and typing rules (§4.6) of HMG(X), and prove that type soundness holds (§4.7).

4.1 Syntax

In keeping with the HM(X) tradition, the type system is parameterized by a first-order constraint logic X, whose variables, terms, and formulas are respectively called *type variables*, *types*, and *constraints*.

Type variables α, β, γ are drawn from a denumerable set. Given two sets of variables $\bar{\alpha}$ and $\bar{\beta}$, we write $\bar{\alpha} \# \bar{\beta}$ for $\bar{\alpha} \cap \bar{\beta} = \emptyset$. If o is a syntactic object, we write $\mathrm{ftv}(o)$ for the *free type variables* of o. We say that $\bar{\alpha}$ is *fresh for* o if and only if $\bar{\alpha} \# \mathrm{ftv}(o)$ holds.

In some proofs, we use *renamings* θ, which are total, bijective mappings from type variables to type variables with finite domain. The *domain* $\mathrm{dom}(\theta)$ of a renaming θ is the set of type variables α such that α and $\theta(\alpha)$ differ. We say that θ is *fresh for* an object o if and only if $\mathrm{dom}(\theta) \# \mathrm{ftv}(o)$ holds, or equivalently, if $\theta(o)$ is o. When proving a theorem T, we say that a hypothesis H may be assumed *without loss of generality* (*w.l.o.g.*) if the theorem T follows from the theorem $H \Rightarrow T$ via a renaming argument, which is usually left implicit.

We assume a fixed, arbitrary set of *algebraic data type constructors* ε, each of which is equipped with a nonnegative arity. Then, *types* τ are built out of type variables using a distinguished arrow type constructor $\rightarrow$ and algebraic data type constructors (whose arity must be obeyed).

Constraints C, D are built out of types using basic predicates π and the standard first-order connectives.

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \varepsilon(\tau, \ldots, \tau)$$
$$C, D ::= \pi\, \bar{\tau} \mid C \wedge C \mid \exists \alpha.C \mid \neg\, C$$

The set of basic predicates π is left unspecified, which allows the system to be instantiated in may ways. Every predicate is assumed to have a fixed arity. We assume that a distinguished binary predicate $\leq$ is given, and write $\tau_1 \leq \tau_2$ (read: τ_1 *is a subtype of* τ_2) for $\leq \tau_1\, \tau_2$. Via some syntactic sugar, it is possible to view equality $\tau = \tau$, truth true, falsity false, universal quantification $\forall \alpha.C$, disjunction $C \vee C$, and implication $C \Rightarrow C$ as part of the constraint language.

Remark 4.1. In many applications, it is necessary to partition types into several *sorts*. Doing so does not introduce any fundamental complication, so, for the sake of simplicity, we ignore this aspect and assume that there is only one sort. ◇

Remark 4.2. Negation $\neg\, C$ was not considered part of the constraint language by Odersky et al. (1999). In the present paper, we exploit the connectives $\forall$, $\vee$, and $\Rightarrow$. The latter contains negation, so introducing these three connectives is equivalent to introducing negation alone. The presence of full negation does not affect the type soundness proof (§4) or the reduction of type inference to constraint solving (§5). However, we do find it necessary to restrict its use in order for the reduction to produce *tractable* constraints (§6). ◇

As in HM(X), a (constrained) *type scheme* σ is a pair of a constraint C and a type τ, wrapped within a set of universal quantifiers $\bar{\alpha}$; we write $\sigma ::= \forall \bar{\alpha}[C].\tau$. By abuse of notation, a type τ may be viewed as the type scheme $\forall \varnothing[\mathsf{true}].\tau$, so types form a subset of type schemes.

An *environment* Γ is a finite mapping from variables to type schemes. An environment is *simple* if it maps variables to types. We write $\mathrm{dom}(\Gamma)$ for the domain of Γ.

An *environment fragment* Δ is a pair of a constraint D and a *simple* environment Γ, wrapped within a set of existential quantifiers $\bar{\beta}$; we write $\Delta ::= \exists \bar{\beta}[D]\Gamma$. The domain of Δ is that of Γ. This notion is new in HMG(X). Environment fragments appear in judgments about patterns (§4.5) and are meant to describe the static knowledge that is gained by successfully matching a value against a pattern. The reader may wish to peek ahead at Example 4.21.

4.2 Interpretation

The logic is interpreted in a *model* $(T, \leq)$, a nonempty, partially ordered set whose elements t are referred to as *ground types* and whose ordering is used to interpret the basic predicate $\leq$. Keep in mind that this ordering may in fact be equality: in such a case, the type system does not have subtyping.

Because syntactic objects may have free type variables, they are interpreted under a *ground assignment* ρ, a total mapping of the type variables to ground types. The interpretation of a type variable α under ρ is simply $\rho(\alpha)$. The interpretation of a type τ under ρ, written $\rho(\tau)$, is then defined in a compositional manner. For instance, $\rho(\tau_1 \to \tau_2)$ is $\rho(\tau_1) \to \rho(\tau_2)$, where the second $\to$ symbol denotes a fixed, unspecified total mapping of T^2 into T. The interpretation of every type constructor ε is defined similarly.

The interpretation of a constraint C under ρ is a truth value: we write $\rho \vdash C$ when ρ *satisfies* C. The partial ordering on T is used to interpret subtyping constraints: that is, $\rho \vdash \tau_1 \leq \tau_2$ is defined as $\rho(\tau_1) \leq \rho(\tau_2)$. The interpretation of the other basic predicates π is unspecified. The interpretation of the first-order connectives $\wedge$, $\exists$ and $\neg$ is standard. We write $C_1 \Vdash C_2$ (read: C_1 *entails* C_2) if and only if, for every ground assignment ρ, $\rho \vdash C_1$ implies $\rho \vdash C_2$. We write $C_1 \equiv C_2$ (read: C_1 *and* C_2 *are equivalent*) if and only if both $C_1 \Vdash C_2$ and $C_2 \Vdash C_1$ hold.

A constraint C *determines* a set of type variables $\bar{\alpha}$ if and only if any two ground assignments that satisfy C and that coincide outside $\bar{\alpha}$ must coincide on $\bar{\alpha}$ as well. This standard notion (Pottier and Rémy, 2005) is exploited in §6. It enjoys the following property:

LEMMA 4.3. *Assume C_1 determines $\bar{\alpha}$. Then, $\exists\bar{\alpha}.C_1 \wedge \forall\bar{\alpha}.C_1 \Rightarrow C_2$ is equivalent to $\exists\bar{\alpha}.(C_1 \wedge C_2)$.* ◇ Proof on page 49

In order to guarantee type soundness, the model must satisfy a number of requirements, which we state in §4.3. Before doing so, however, we interpret type schemes and environment fragments, and explain how this gives rise to orderings on these objects. (In fact, they are only preorders, but we stick to the word "ordering.")

As in HM(X), a type scheme is interpreted as an upward-closed set of ground types. This is standard: see, for instance, Trifonov and Smith (1996) or Sulzmann (2000).

Definition 4.4. The interpretation $\rho(\forall\bar{\alpha}[D].\tau)$ of the type scheme $\forall\bar{\alpha}[D].\tau$ under ρ is the set of ground types $\{t \;/\; \exists\bar{t} \quad \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D \wedge \rho[\bar{\alpha} \mapsto \bar{t}](\tau) \leq t\}$. It may also be written $\uparrow\{\rho[\bar{\alpha} \mapsto \bar{t}](\tau) \;/\; \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D\}$, where $\uparrow$ is the upward closure operator. ◇

This definition gives rise to an ordering on type schemes, which extends the ordering on types. It is defined as follows. Given two type schemes σ and σ', we consider $\sigma \leq \sigma'$ to be a valid constraint, which we interpret by defining $\rho \vdash \sigma \leq \sigma'$ as $\rho(\sigma) \supseteq \rho(\sigma')$.

As a sanity check, one may verify that the type scheme $\forall\alpha.\alpha$ is a least element in this ordering: indeed, its interpretation under every ground assignment is the full model T, so every constraint of the form $(\forall\alpha.\alpha) \leq \sigma$ is a tautology. One may also check that $\forall\alpha.\sigma$ is more general than σ, that is, every constraint of the form $(\forall\alpha.\sigma) \leq \sigma$ is a tautology. Thus, the ordering allows a universally quantified type variable to be instantiated (to become free).

The following property is also useful on a few occasions:

LEMMA 4.5. $D \Vdash \forall\bar{\alpha}[D].\tau \leq \tau$. ◇ Proof on page 49

It is well-known that ordering constraints on type schemes may also be viewed as syntactic sugar for constraints that involve the ordering on types. This is stated by the following lemma:

Proof on page 49

LEMMA 4.6. *Let σ and σ' stand for $\forall\bar{\alpha}[D].\tau$ and $\forall\bar{\alpha}'[D'].\tau'$, respectively. Let $\bar{\alpha}' \# \mathrm{ftv}(\sigma)$. Then, $\sigma \leq \sigma' \equiv \forall\bar{\alpha}'.D' \Rightarrow \sigma \leq \tau'$ holds. Furthermore, let $\bar{\alpha} \# \mathrm{ftv}(\tau')$. Then, $\sigma \leq \tau' \equiv \exists\bar{\alpha}.(D \wedge \tau \leq \tau')$ holds.* ◇

We write $\exists\sigma$ for $\exists\alpha.(\sigma \leq \alpha)$, where α is fresh for σ. This constraint, which requires σ to denote a nonempty set of ground types, is used in VAR (§4.6).

The ordering on type schemes may be extended pointwise to an ordering on environments. Thus, when Γ and Γ' are environments with a common domain, we consider $\Gamma' \leq \Gamma$ to be syntactic sugar for the conjunction of the constraints $\Gamma'(x) \leq \Gamma(x)$, where x ranges over the domain of Γ and Γ'.

Let us now turn to the interpretation of environment fragments. Let a *ground environment* g be a finite mapping from variables to ground types. Given a ground assignment ρ and a simple environment Γ, let $\rho(\Gamma)$ stand for the ground environment that maps every $x \in \mathrm{dom}(\Gamma)$ to $\rho(\Gamma(x))$. The ordering on ground types is extended pointwise to ground environments with a common domain. Then, an environment fragment is interpreted as a downward-closed set of ground environments, as follows:

Definition 4.7. The interpretation of the environment fragment $\exists\bar{\beta}[D]\Gamma$ under the ground assignment ρ, written $\rho(\exists\bar{\beta}[D]\Gamma)$, is the set of ground environments $\{g \;/\; \exists\bar{t} \quad \rho[\bar{\beta} \mapsto \bar{t}] \vdash D \wedge g \leq \rho[\bar{\beta} \mapsto \bar{t}](\Gamma)\}$. It may also be written $\downarrow\{\rho[\bar{\beta} \mapsto \bar{t}](\Gamma) \;/\; \rho[\bar{\beta} \mapsto \bar{t}] \vdash D\}$, where $\downarrow$ is the downward closure operator. ◇

Again, this definition gives rise to an ordering on environment fragments, which extends the ordering on simple environments. Given two environment fragments Δ and Δ' with a common domain, we consider $\Delta' \leq \Delta$ to be a valid constraint, which we interpret by defining $\rho \vdash \Delta' \leq \Delta$ as $\rho(\Delta') \subseteq \rho(\Delta)$.

As a sanity check, one may verify that the environment fragment $\exists\beta.(x : \beta)$ is a greatest element among the environment fragments of domain $\{x\}$. We also prove below (see rule F-HIDE in Figure 5) that Δ is more general than $\exists\alpha.\Delta$, that is, every constraint of the form $\Delta \leq \exists\alpha.\Delta$ is a tautology. Thus, the ordering allows a free type variable to become abstract (existentially quantified).

The interpretation of environment fragments, and the definition of their ordering, are dual to those of type schemes: $\uparrow$ and $\supseteq$ are replaced with $\downarrow$ and $\subseteq$, respectively. These changes reflect the dual nature of the $\forall$ and $\exists$ quantifiers.

As in the case of type schemes, ordering constraints on environment fragments may be viewed as syntactic sugar for constraints that involve the ordering on types. This is stated by the following lemma:

Proof on page 49

LEMMA 4.8. *Let Δ and Δ' stand for $\exists\bar{\beta}[D]\Gamma$ and $\exists\bar{\beta}'[D']\Gamma'$, respectively. Let $\bar{\beta} \# \mathrm{ftv}(\Gamma')$ and $\bar{\beta}' \# \mathrm{ftv}(\Delta)$. Then, we have $\Delta' \leq \Delta \equiv \forall\bar{\beta}'.D' \Rightarrow \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$. As a corollary, if, in addition, $\bar{\beta}' \# \mathrm{ftv}(C)$ holds, then $C \Vdash \Delta' \leq \Delta$ is equivalent to $C \wedge D' \Vdash \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$.* ◇

4.3 Requirements on the model

So far, the ordering $\leq$ on ground types, as well as the interpretation of the type constructors $\rightarrow$ and ε, have been left unspecified. This is intended to offer a great deal of flexibility when

defining instances of HMG(X). However, in order to establish type soundness, we must make a few assumptions about them.

First, subtyping assertions that involve an arrow type and an algebraic data type, or two algebraic data types with distinct head symbols, must be unsatisfiable. This is required for progress to hold (Lemma 4.40 and Theorem 4.42).

Requirement 4.9. Every constraint of the form $\tau_1 \to \tau_2 \leq \varepsilon(\bar{\tau})$ or $\varepsilon(\bar{\tau}) \leq \tau_1 \to \tau_2$ or $\varepsilon(\bar{\tau}) \leq \varepsilon'(\bar{\tau}')$, where ε and ε' are distinct, is unsatisfiable. ◇

Second, the arrow type constructor must be contravariant in its domain and covariant in its codomain. This is required for subject reduction to hold (Lemma 4.34). This requirement appears in all type systems equipped with subtyping.

Requirement 4.10. $\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$ entails $\tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2'$. ◇

Last, we must make similar *variance* requirements about every algebraic data type constructor ε. The requirements that bear on ε depend on its definition, however; so, before stating these requirements, we must recall how (guarded) algebraic data types are defined.

In keeping with the ML tradition, algebraic data types are explicitly defined, as part of the program text. As a simplifying assumption, we assume that all such definitions are placed in a prologue, so that they are available to the typechecker when it starts examining the program's body (an expression). A prologue consists of a series of *data constructor declarations*, each of which assigns a closed type scheme to a (distinct) data constructor K, as follows:

$$K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$$

This is the form (3) of §1.1. Here, n must be the arity of K, and $\bar{\alpha}$ must be a vector of distinct type variables. When K is declared in such a way, we say that it is *associated* with the algebraic data type constructor ε.

We may now state the variance requirements that bear on algebraic data type constructors. They are necessary to establish subject reduction and progress (Lemmas 4.37 and 4.40), and are standard in type systems featuring both subtyping and isorecursive types: see, for instance, Pottier and Rémy (2005) or Simonet (2003a).

Requirement 4.11. For every data constructor K, if $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ and $K :: \forall \bar{\alpha}'\bar{\beta}'[D'].\tau_1' \cdots \tau_n' \to \varepsilon(\bar{\alpha}')$ are two α-equivalent versions of K's declaration, and if $\bar{\beta}$ is fresh for every τ_i', then $D' \wedge \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}.(D \wedge_i \tau_i' \leq \tau_i)$ must hold. ◇

(Throughout the paper, we write $C \wedge_i C_i$ for $C \wedge C_1 \wedge \ldots \wedge C_n$.) Although these requirements are standard, they may conceivably seem cryptic. Here is a brief and informal attempt at explaining them. Assigning K the type scheme $\forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ amounts to declaring that the abstract data type $\varepsilon(\bar{\alpha})$ is *isomorphic* to a sum type of the form $\exists \bar{\beta}[D](\tau_1 \cdots \tau_n) + \ldots$. A similar statement can be made about the α-equivalent declaration $K :: \forall \bar{\alpha}'\bar{\beta}'[D'].\tau_1' \cdots \tau_n' \to \varepsilon(\bar{\alpha}')$. Now, for this isomorphism, which is declared as part of the prologue, to be consistent with the model, which defines the interpretation of ε and of $\leq$, the existence of a subtyping relationship between the abstract types $\varepsilon(\bar{\alpha}')$ and $\varepsilon(\bar{\alpha})$ must entail the existence of an analogous relationship between their concrete representations $\exists \bar{\beta}'[D'](\tau_1' \cdots \tau_n') + \ldots$ and $\exists \bar{\beta}[D](\tau_1 \cdots \tau_n) + \ldots$. In other words, since the sum type constructor $+$ is covariant, the law $\varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}'[D'](\tau_1' \cdots \tau_n') \leq \exists \bar{\beta}[D](\tau_1 \cdots \tau_n)$

must hold. In fact, the variance requirement could conceivably be stated in this manner. Under the hypothesis that $\bar{\beta}$ is fresh for every τ_i', however, one may prove, by exploiting Lemma 4.8, that a consequence of this law is $D' \wedge \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists\bar{\beta}.(D \wedge_i \tau_i' \leq \tau_i)$. This more basic formulation is the one adopted in the statement of Requirement 4.11.

Remark 4.12. If the model is defined in such a way that every algebraic data type constructor ε is *invariant* in every parameter, then Requirement 4.11 is met. The proof of this assertion is not difficult, and relies on the fact that, in this case, $\varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha})$ entails $\bar{\alpha}' = \bar{\alpha}$. In particular, in a type system *without* subtyping, where every type constructor is invariant, Requirement 4.11 is always met. The requirement becomes nontrivial only when subtyping is interpreted in a nontrivial way. ◇

Remark 4.13. We have assumed that the model preexists, and only then required the program's prologue to be consistent with it. In practice, it is more natural to let the prologue influence the construction of the model, so that the two are consistent by design. As far as the present paper is concerned, the distinction is irrelevant. ◇

4.4 Environment fragments

Before we attack the definition of the type system, we must introduce a few operations on environment fragments. The first operation enriches an environment fragment Δ with a constraint C, yielding a (more precise) environment fragment $[C]\Delta$. The second one abstracts a set of type variables $\bar{\alpha}$ out of an environment fragment Δ, yielding a (less precise) environment fragment $\exists\bar{\alpha}.C$. The two operations are defined at once below.

Definition 4.14. If Δ is $\exists\bar{\beta}[D]\Gamma$ and $\bar{\beta} \# \mathrm{ftv}(\bar{\alpha}, C)$ holds, then we write $\exists\bar{\alpha}[C]\Delta$ for the environment fragment $\exists\bar{\alpha}\bar{\beta}[C \wedge D]\Gamma$. We write $\exists\bar{\alpha}.\Delta$ for $\exists\bar{\alpha}[\mathsf{true}]\Delta$ and $[C]\Delta$ for $\exists\varnothing[C]\Delta$. ◇

The next lemma provides an interpretation of the composite operation. This is a low-level result, used only in the proof of more elaborate laws (see Lemma 4.20 and Figure 5).

LEMMA 4.15. *$\rho(\exists\bar{\alpha}[C]\Delta)$ is $\cup\{\rho[\bar{\alpha} \mapsto \bar{t}](\Delta) \,/\, \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\}$.* ◇ Proof on page 49

The next two operations are binary. Given two environment fragments Δ_1 and Δ_2, they produce a new environment fragment. They are intended to reflect the effect of conjunction and disjunction patterns, respectively. Although their syntactic definitions, which follow, are rather heavy, their interpretations, given by the two lemmas that follow, are simple.

Definition 4.16. Given two simple environments Γ_1 and Γ_2 with disjoint domains, their *conjunction* $\Gamma_1 \times \Gamma_2$ is their set-theoretic union. (Recall that a simple environment is a partial mapping of variables to types.) Given two environment fragments Δ_1 and Δ_2 with disjoint domains, their *conjunction* $\Delta_1 \times \Delta_2$ is the environment fragment $\exists\bar{\beta}_1\bar{\beta}_2[D_1 \wedge D_2](\Gamma_1 \times \Gamma_2)$, provided Δ_i is $\exists\bar{\beta}_i[D_i]\Gamma_i$ and provided $\bar{\beta}_1 \# \bar{\beta}_2$, $\bar{\beta}_1 \# \mathrm{ftv}(\Delta_2)$, and $\bar{\beta}_2 \# \mathrm{ftv}(\Delta_1)$ hold. ◇

Definition 4.17. Given two environment fragments Δ_1 and Δ_2 with a common domain, their *disjunction* $\Delta_1 + \Delta_2$ is the environment fragment $\exists\bar{\beta}_1\bar{\beta}_2\bar{\alpha}[(D_1 \wedge \Gamma \leq \Gamma_1) \vee (D_2 \wedge \Gamma \leq \Gamma_2)]\Gamma$, provided Δ_i is $\exists\bar{\beta}_i[D_i]\Gamma_i$, provided $\bar{\beta}_1 \# \bar{\beta}_2$, $\bar{\beta}_1 \# \mathrm{ftv}(\Delta_2)$, and $\bar{\beta}_2 \# \mathrm{ftv}(\Delta_1)$ hold, and provided the environment Γ, whose domain is that of Γ_1 and Γ_2, maps every variable to a distinct type variable in $\bar{\alpha}$, where $\bar{\alpha} \# \mathrm{ftv}(\bar{\beta}_1, \bar{\beta}_2, \Delta_1, \Delta_2)$ holds. ◇

$$
\begin{array}{rcll}
\mathsf{true} & \Vdash & \Delta \leq \exists\bar{\alpha}.\Delta & (\text{F-HIDE}) \\
C_1 \Rightarrow C_2 & \Vdash & [C_1]\Delta \leq [C_2]\Delta & (\text{F-IMPLY}) \\
C \Rightarrow \Delta_1 \leq \Delta_2 & \Vdash & [C]\Delta_1 \leq [C]\Delta_2 & (\text{F-ENRICH}) \\
\forall\bar{\alpha}.(\Delta_1 \leq \Delta_2) & \Vdash & \exists\bar{\alpha}.\Delta_1 \leq \exists\bar{\alpha}.\Delta_2 & (\text{F-EX}) \\
\Delta_1 \leq \Delta_2 & \Vdash & \Delta \times \Delta_1 \leq \Delta \times \Delta_2 & (\text{F-AND}) \\
\Delta_1 \leq \Delta_2 & \Vdash & \Delta + \Delta_1 \leq \Delta + \Delta_2 & (\text{F-OR}) \\
\Delta_1 \leq \Delta \wedge \Delta_2 \leq \Delta & \Vdash & \Delta_1 + \Delta_2 \leq \Delta & (\text{F-GLB}) \\
\mathsf{true} & \Vdash & \Delta_1 \leq \Delta_1 + \Delta_2 & (\text{F-LUB})
\end{array}
$$

Figure 5: Some properties of subsumption between environment fragments

The next two lemmas are also low-level results, used only in the proof of the laws in Figure 5. To state the first of these lemmas, we must define conjunction of *ground* environments and of sets thereof. (The disjunction of two sets of ground environments is simply their set-theoretic union.) Given two ground environments g_1 and g_2 of disjoint domains, we let $g_1 \times g_2$ stand for their set-theoretic union, that is, the ground environment g of domain $\mathrm{dom}(g_1) \cup \mathrm{dom}(g_2)$ that maps x to $g_i(x)$ if $x \in \mathrm{dom}(g_i)$ and $i \in \{1, 2\}$. If G_1 and G_2 are two sets of ground environments, we let $G_1 \times G_2$ stand for $\{g_1 \times g_2 \mathbin{/} g_1 \in G_1 \wedge g_2 \in G_2\}$.

LEMMA 4.18. *$\rho(\Delta_1 \times \Delta_2)$ is $\rho(\Delta_1) \times \rho(\Delta_2)$.* ◇ Proof on page 50

LEMMA 4.19. *$\rho(\Delta_1 + \Delta_2)$ is $\rho(\Delta_1) \cup \rho(\Delta_2)$.* ◇ Proof on page 50

The previous lemmas allow establishing a number of laws about environment fragments, which are useful when reasoning about the correctness and completeness of the constraint generation rules (§5).

LEMMA 4.20. *The entailment laws in Figure 5 are valid.* ◇ Proof on page 50

4.5 Typing judgments

The type system features three distinct judgment forms, corresponding to patterns, expressions, and clauses.

Judgments about patterns are written $C \vdash p : \tau \rightsquigarrow \exists\bar{\beta}[D]\Gamma$, where the domain of Γ is $\mathrm{dpv}(p)$. Such a judgment may 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.*

If the system only had ordinary (as opposed to guarded) algebraic data types, then there would be no need for $\bar{\beta}$ and D. In other words, it would be possible to identify environment fragments Δ with simple environments Γ. For instance, assuming $K :: \forall\bar{\alpha}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$, the familiar judgment $\mathsf{true} \vdash K\, x_1 \cdots x_n : \varepsilon(\bar{\alpha}) \rightsquigarrow (x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n)$ holds: when matching a value of type $\varepsilon(\bar{\alpha})$, the pattern $K\, x_1 \cdots x_n$ binds the variable x_i to a value of type τ_i, for every $i \in \{1, \ldots, n\}$.

If the system only had existential types in the style of Läufer and Odersky (1994), then environment fragments would be of the form $\exists\bar{\beta}.\Gamma$. For instance, imagine we have $K :: \forall\bar{\alpha}\bar{\beta}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$. Then, because the type variables $\bar{\beta}$ do not appear in the data constructor's result type, the type constructor ε behaves as an existential type: applying K amounts to creating an existential

package, while matching against K amounts to opening such a package. Thus, matching against K locally introduces $\bar{\beta}$ as a vector of abstract types. In our system, this is reflected by the judgment $\mathsf{true} \vdash K\, x_1 \cdots x_n : \varepsilon(\bar{\alpha}) \rightsquigarrow \exists\bar{\beta}.(x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n)$.

In the full system, the declaration of a data constructor K may involve a constraint D, which bears on the type variables $\bar{\alpha}$ and $\bar{\beta}$. Then, a successful match against K not only introduces the abstract types $\bar{\beta}$, but also guarantees that D holds. To keep track of this information, we allow fragments to carry a constraint. For instance, if $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ holds, then we have $\mathsf{true} \vdash K\, x_1 \cdots x_n : \varepsilon(\bar{\alpha}) \rightsquigarrow \exists\bar{\beta}[D](x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n)$.

Judgments about expressions retain the same form as in HM(X): they are written $C, \Gamma \vdash e : \sigma$, 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. Judgments about clauses, of the form $C, \Gamma \vdash c : \tau$, are to be interpreted in a similar way.

As in HM(X), all judgments are identified up to constraint equivalence: for instance, the judgments $C_1, \Gamma \vdash e : \sigma$ and $C_2, \Gamma \vdash e : \sigma$ are considered interchangeable when $C_1 \equiv C_2$ holds. In a valid judgment $C, \Gamma \vdash e : \sigma$, the constraint C may well be unsatisfiable. A closed expression e is *well-typed* if and only if $C, \varnothing \vdash e : \sigma$ holds for some *satisfiable* constraint C.

4.6 Typing rules

Whether a judgment is valid is defined by the rules in Figure 6, which we now review, beginning with the rules that concern patterns.

P-Empty and P-Wild tell that the patterns 0 and 1 may be used at any type, and bind no variables. Because matching against 0 never succeeds, the environment fragment produced in P-Empty includes the absurd constraint false. Conversely, because matching against 1 always succeeds, it provides no information; hence, the environment fragment produced in P-Wild includes the tautology true.

P-Var is similar to P-Wild, except the environment fragment has nonempty domain. The rule may be read: *if the pattern x matches a value of type τ, then the variable x becomes bound to a value of type τ.*

P-And requires both p_1 and p_2 to match values of type τ, producing two environment fragments Δ_1 and Δ_2 of disjoint domains, because the pattern $p_1 \wedge p_2$ is well-formed; thus, the conjunction $\Delta_1 \times \Delta_2$ is defined.

Similarly, P-Or requires both p_1 and p_2 to match values of type τ. Furthermore, it requires both to produce the same environment fragment Δ, so that it becomes possible to state that the pattern $p_1 \vee p_2$ gives rise to Δ, without knowing which of p_1 or p_2 leads to a successful match.

P-Cstr looks up the declaration of the data constructor K and introduces the type variables $\bar{\beta}$. These type variables are chosen fresh (indeed, the reader may check that they cannot appear free in the rule's conclusion), so as to play the role of abstract types. Every p_i is typechecked under a hypothesis *augmented* with D, a constraint that bears on $\bar{\alpha}$ and $\bar{\beta}$, and is found in the declaration of K. Thus, the type information gained by ensuring that the value at hand is indeed an application of K becomes available when checking that every subpattern is well-typed. In other words, new type information is propagated *top-down* through the pattern. The environment fragment associated with the entire pattern is obtained by fusing the environment fragments associated with its subpatterns, as in the case of conjunction, and by incorporating

Patterns (syntax-directed)

$$\textsc{p-Empty}\quad C \vdash 0 : \tau \rightsquigarrow \exists\varnothing[\mathsf{false}]\varnothing \qquad \textsc{p-Wild}\quad C \vdash 1 : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}]\varnothing \qquad \textsc{p-Var}\quad C \vdash x : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}](x \mapsto \tau)$$

$$\textsc{p-And}\ \frac{\forall i \quad C \vdash p_i : \tau \rightsquigarrow \Delta_i}{C \vdash p_1 \wedge p_2 : \tau \rightsquigarrow \Delta_1 \times \Delta_2} \qquad \textsc{p-Or}\ \frac{\forall i \quad C \vdash p_i : \tau \rightsquigarrow \Delta}{C \vdash p_1 \vee p_2 : \tau \rightsquigarrow \Delta}$$

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

Patterns (non-syntax-directed)

$$\textsc{p-EqIn}\ \frac{C \vdash p : \tau' \rightsquigarrow \Delta \qquad C \Vdash \tau = \tau'}{C \vdash p : \tau \rightsquigarrow \Delta} \qquad \textsc{p-SubOut}\ \frac{C \vdash p : \tau \rightsquigarrow \Delta' \qquad C \Vdash \Delta' \leq \Delta}{C \vdash p : \tau \rightsquigarrow \Delta} \qquad \textsc{p-Hide}\ \frac{C \vdash p : \tau \rightsquigarrow \Delta \qquad \bar\alpha \# \mathrm{ftv}(\tau, \Delta)}{\exists\bar\alpha.C \vdash p : \tau \rightsquigarrow \Delta}$$

Expressions (syntax-directed)

$$\textsc{Var}\ \frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma} \qquad \textsc{Cstr}\ \frac{\forall i \quad C, \Gamma \vdash e_i : \tau_i \qquad K :: \forall\bar\alpha\bar\beta[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar\alpha) \qquad C \Vdash D}{C, \Gamma \vdash K\, e_1 \cdots e_n : \varepsilon(\bar\alpha)} \qquad \textsc{Abs}\ \frac{\forall i \quad C, \Gamma \vdash c_i : \tau}{C, \Gamma \vdash \lambda(c_1 \cdots c_n) : \tau}$$

$$\textsc{App}\ \frac{C, \Gamma \vdash e_1 : \tau' \to \tau \qquad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1\, e_2 : \tau} \qquad \textsc{Fix}\ \frac{C, \Gamma[x \mapsto \sigma] \vdash v : \sigma}{C, \Gamma \vdash \mu x.v : \sigma} \qquad \textsc{Let}\ \frac{C, \Gamma \vdash e_1 : \sigma' \qquad C, \Gamma[x \mapsto \sigma'] \vdash e_2 : \sigma}{C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \sigma}$$

Expressions (non-syntax-directed)

$$\textsc{Gen}\ \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar\alpha \# \mathrm{ftv}(\Gamma, C)}{C \wedge \exists\bar\alpha.D, \Gamma \vdash e : \forall\bar\alpha[D].\tau} \qquad \textsc{Inst}\ \frac{C, \Gamma \vdash e : \forall\bar\alpha[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash e : \tau} \qquad \textsc{Sub}\ \frac{C, \Gamma \vdash e : \tau' \qquad C \Vdash \tau' \leq \tau}{C, \Gamma \vdash e : \tau} \qquad \textsc{Hide}\ \frac{C, \Gamma \vdash e : \sigma \qquad \bar\alpha \# \mathrm{ftv}(\Gamma, \sigma)}{\exists\bar\alpha.C, \Gamma \vdash e : \sigma}$$

Clauses

$$\textsc{Clause}\ \frac{C \vdash p : \tau' \rightsquigarrow \exists\bar\beta[D]\Gamma' \qquad C \wedge D, \Gamma\Gamma' \vdash e : \tau \qquad \bar\beta \# \mathrm{ftv}(C, \Gamma, \tau)}{C, \Gamma \vdash p.e : \tau' \to \tau}$$

Figure 6: Typing rules

the guarded (bounded) existential quantification $\exists\bar{\beta}[D][]$, which ensures that the abstract type variables $\bar{\beta}$ remain local.

P-EqIn allows replacing the type τ with an arbitrary type τ', provided they are provably equal under C. We require $\tau = \tau'$, rather than $\tau \leq \tau'$ only: although the latter condition does not compromise type safety, it appears to create complications with type inference.

P-SubOut allows weakening the environment fragment produced by a pattern, in accordance with the subsumption ordering defined earlier.

P-Hide makes some type variables local to a subderivation, which helps manage names; it is analogous to Hide.

Example 4.21. The following is a valid derivation:

$$\frac{\mathit{Int} :: \forall\alpha[\alpha = \mathit{int}].\mathit{ty}(\alpha)}{\mathsf{true} \vdash \mathit{Int} : \mathit{ty}(\alpha) \rightsquigarrow \exists\varnothing[\alpha = \mathit{int}]\varnothing}\ \textsc{p-Cstr}$$

Its conclusion may be read as follows. First, it is valid to match a value of type $\mathit{ty}(\alpha)$ against the pattern *Int*. Furthermore, if successful, this test guarantees that α is *int*. The pattern *Int* does not introduce any abstract type variables or bind any variables. In the next example, we refer to this derivation as (d_1).

Here is another valid derivation:

$$\frac{\begin{array}{c}\forall i \in \{1,2\} \quad \dfrac{}{\alpha = \beta_1 \times \beta_2 \vdash t_i : \mathit{ty}(\beta_i) \rightsquigarrow (t_i : \mathit{ty}(\beta_i))}\ \textsc{p-Var} \\ \mathit{Pair} :: \forall\alpha\beta_1\beta_2[\alpha = \beta_1 \times \beta_2].\mathit{ty}(\beta_1) \cdot \mathit{ty}(\beta_2) \to \mathit{ty}(\alpha)\end{array}}{\mathsf{true} \vdash \mathit{Pair}\,(t_1, t_2) : \mathit{ty}(\alpha) \rightsquigarrow \exists\beta_1\beta_2[\alpha = \beta_1 \times \beta_2](t_1 : \mathit{ty}(\beta_1); t_2 : \mathit{ty}(\beta_2))}\ \textsc{p-Cstr}$$

Its conclusion may be read as follows. First, it is valid to match a value of type $\mathit{ty}(\alpha)$ against the pattern $\mathit{Pair}\,(t_1, t_2)$. Furthermore, if successful, this test guarantees that there exist types β_1 and β_2 such that α is $\beta_1 \times \beta_2$ and the variables t_1 and t_2 are bound to values of types $\mathit{ty}(\beta_1)$ and $\mathit{ty}(\beta_2)$, respectively. In the next example, we refer to this derivation as (d_2). ◇

Let us now briefly review the rules that concern expressions. They are standard, that is, identical to those of HM(X), up to minor cosmetic differences; see, for instance, Odersky et al. (1999) or Pottier and Rémy (2005). Our version of Fix allows *polymorphic recursion*, often an essential feature in programs that involve guarded algebraic data types, as illustrated in §2. Gen performs generalization, turning a type into a type scheme, while Inst performs the converse operation. Sub allows replacing a type τ' with an arbitrary type τ, provided the latter is provably a supertype of the former under C. Hide makes some type variables local to a subderivation, which helps manage names.

There remains to explain Clause, which assigns a function type $\tau' \to \tau$ to a clause $p.e$. The pattern p is checked against the argument type τ', yielding an environment fragment $\exists\bar{\beta}[D]\Gamma'$. Then, the expression e is required to have type τ, under an assumption augmented with D and an environment augmented with Γ'. By requiring the type variables $\bar{\beta}$ to be fresh, the third premise ensures that they remain abstract within e; this condition is identical to that found in the elimination construct for existential types (Läufer and Odersky, 1994). A key point, here, is the fact that e is typechecked under the augmented constraint $C \wedge D$. In other words, the type system exploits the presence of a *dynamic* check, namely pattern matching, to obtain new *static* information. As a result, in a function defined by cases, each clause may be typechecked assuming *different* constraints.

Example 4.22. Here is a valid derivation for the first clause in the definition of *print*, the generic printing function defined in §2.1. We assume that the environment Γ assigns type $int \to unit$ to the variable *print_int* and exploit the derivation (d_1) of Example 4.21.

$$\dfrac{(d_1)\quad \dfrac{\dfrac{\dfrac{\cdots}{\alpha = int, \Gamma \vdash \lambda x.print\_int\, x : int \to unit}}{\alpha = int \Vdash int \to unit \leq \alpha \to unit}}{\alpha = int, \Gamma \vdash \lambda x.print\_int\, x : \alpha \to unit}\ \textsc{Sub}}{\mathsf{true}, \Gamma \vdash Int.\lambda x.print\_int\, x : ty(\alpha) \to \alpha \to unit}\ \textsc{Clause}$$

The assumption $\alpha = int$, which appears in the conclusion of (d_1), is made available in the second premise of Clause, and is exploited by Sub. The derivation concludes that the clause $Int.\lambda x.print\_int\, x$ has type $ty(\alpha) \to \alpha \to unit$, where α is *unconstrained*: indeed, the hypothesis $\alpha = int$, which is necessary to typecheck the right-hand side of the clause, is local.

Here is a valid derivation for the second clause that defines *print*. (The intermediate call to *print_string* is omitted for brevity.) We assume that the environment Γ assigns type scheme $\forall\alpha.ty(\alpha) \to \alpha \to unit$ to *print*, so as to be able to typecheck the recursive calls to *print*. We write Γ' for the environment $(t_1 : ty(\beta_1); t_2 : ty(\beta_2))$.

$$\dfrac{(d_2)\quad \dfrac{\dfrac{\dfrac{\cdots}{\alpha = \beta_1 \times \beta_2, \Gamma\Gamma' \vdash \lambda(x_1, x_2).(print\, t_1\, x_1; print\, t_2\, x_2) : \beta_1 \times \beta_2 \to unit}}{\alpha = \beta_1 \times \beta_2 \Vdash \beta_1 \times \beta_2 \to unit \leq \alpha \to unit}}{\alpha = \beta_1 \times \beta_2, \Gamma\Gamma' \vdash \lambda(x_1, x_2).(print\, t_1\, x_1; print\, t_2\, x_2) : \alpha \to unit}\ \textsc{Sub}}{\mathsf{true}, \Gamma \vdash Pair\,(t_1, t_2).\lambda(x_1, x_2).(print\, t_1\, x_1; print\, t_2\, x_2) : ty(\alpha) \to \alpha \to unit}\ \textsc{Clause}$$

This derivation has identical structure. The type variables β_1 and β_2 do not appear in its conclusion: they are local to the subderivation rooted at Clause's second premise. The hypothesis $\alpha = \beta_1 \times \beta_2$ is also local to this subderivation.

By starting with the above two derivations and applying Abs, Gen, and Fix, it is straightforward to derive $\mathsf{true}, \Gamma_0 \vdash \mu print. \ldots : \forall\alpha.ty(\alpha) \to \alpha \to unit$, where Γ_0 assigns type $int \to unit$ to *print_int* and where the dots stand for the body of *print*'s definition. Thus, the function *print*, as defined in §2, is well-typed in (all instances of) HMG(X). ◇

We now conclude our description of the typing rules with a number of technical remarks concerning the formulation of the rules.

Remark 4.23. One could define another version of p-Or, whose premises produce two distinct environment fragments Δ_1 and Δ_2, and whose conclusion produces the disjunction $\Delta_1 + \Delta_2$. By reflexivity of $+$, by f-Lub, and by p-SubOut, the two formulations are equivalent. Disjunction is explicitly used in the constraint generation rules (§5). ◇

Remark 4.24. Some dialects of ML, such as Objective Caml, allow disjunction patterns to bind variables (as we do here). These programming languages provide the user with a choice between writing several clauses that have a common right-hand side, say $(K_1\, x).e$ and $(K_2\, x).e$, or writing a single clause, such as $(K_1\, x \vee K_2\, x).e$. In Objective Caml, the two idioms are semantically equivalent, but the latter admits fewer typings, because it requires both occurrences of x to have

a common type. By contrast, one may check that, in HMG(X), the two idioms do admit the same typings. Thus, our treatment of disjunction patterns is more precise than that of Objective Caml, even when no guarded algebraic data types are involved.

This seems to be the most natural way of dealing with disjunction patterns in HMG(X). It is somewhat costly, because it requires either generating a disjunction constraint (§5), thus reflecting the two possible manners in which x is defined, or textually duplicating e prior to type inference, as suggested above. Either method basically amounts to typechecking e twice. Fortunately, in the common case where p_1 and p_2 bind no variables and are associated with an ordinary (as opposed to guarded) algebraic data type, the disjunction is trivial (its members are identical), so e needs only be typechecked once. ◇

Remark 4.25. The reader might be surprised by the fact that P-CSTR's conclusion assigns type $\varepsilon(\bar{\alpha})$ to the pattern, thus requiring the type constructor ε to be applied to a vector of distinct type variables $\bar{\alpha}$, rather than to a vector of arbitrary types $\bar{\tau}$. In fact, by exploiting the non-syntax-directed rules P-EQIN, P-SUBOUT, and P-HIDE, as well as a weakening lemma (Lemma 4.30), it is possible to prove that this causes no loss of expressiveness.

The same technique is used in the formulation of INST, where a type scheme may be instantiated only with a vector of distinct type variables $\bar{\alpha}$, as opposed to a vector of arbitrary types $\bar{\tau}$. By exploiting SUB, HIDE, and Lemma 4.30, it is possible to prove that this causes no loss of expressiveness. This technique is standard: see, for instance, Sulzmann et al. (1999). ◇

Remark 4.26. One might wonder whether some non-syntax-directed rules such as P-SUBOUT and P-HIDE are really useful, that is, whether they really make the type system more expressive. Indeed, one may prove that these rules enjoy a few normalization properties, such as the following three. (A *shape* is a term whose nodes are names of typing rules and whose leaves are holes. We say that a shape *may be replaced* with a new shape if and only if, for every typing derivation that ends with an instance of the former, there exists a derivation of the same judgment that ends with an instance of the latter and otherwise has the same structure.)

- the shape P-CSTR(P-SUBOUT($\cdot$)) may be replaced with P-SUBOUT(P-CSTR($\cdot$));
- the shape CLAUSE(P-SUBOUT($\cdot$)) may be replaced with CLAUSE, modulo a weakening of the environment in CLAUSE's second premise;
- the shape CLAUSE(P-HIDE($\cdot$)) may be replaced with HIDE(CLAUSE($\cdot$)).

One might hope to establish, by proving enough normalization properties in this style, that all uses of P-SUBOUT, of P-HIDE, or of both rules, can be eliminated. However, this is not true in general. This is essentially due to the following two negative properties:

- the shape P-CSTR(P-HIDE($\cdot$)) may *not* in general be replaced with P-HIDE(P-CSTR($\cdot$)); such a replacement is possible when only ordinary algebraic data types are involved (see Remark 4.29);
- the shape P-HIDE(P-SUBOUT($\cdot$)) may *not* in general be replaced with P-SUBOUT(P-HIDE($\cdot$)).

For these reasons, it is not obvious that the present definition of HMG(X) can be made simpler.◇

Remark 4.27. The analogue of P-EQIN is missing in Xi et al.'s treatment (2003). In fact, in their typing rules for patterns (Xi et al., 2003, Figure 3), no rule allows exploiting the type equations contained in the assumption Δ_0. This causes some safe programs that involve nested patterns to be rejected. ◇

Remark 4.28. It is worth noting that P-CSTR propagates type information in a *top-down* manner, as previously pointed out, but not *sideways*. That is, the information gained by ensuring that $p_1, \ldots, p_i$ match *cannot* be exploited to prove that $p_{i+1}, \ldots, p_n$ are well-typed. This is apparent in the fact that every p_i is checked under the same assumption, namely $C \wedge D$.

As a result of this decision, some programs that might seem natural are ill-typed. Consider, for instance, an uncurried version of *print*:

```
let rec print : ∀α.ty(α) × α → unit = fun tx ->
  match tx with
  | (Int, x) ->
      print_int x
  | (Pair (t1, t2), (x1, x2)) ->
      print t1 x1; print_string " * "; print t2 x2
```

This version of *print* expects a *pair* of a runtime type representation and a value. If the first component of the pair is *Int*, then the second component must be an integer value x; if the first component is an application of *Pair*, then the second component must be a pair (x_1, x_2). So, the code appears to make perfect sense, and is perhaps even easier to read than in its original curried form.

It is, however, ill-typed in our system, because the second component of the pair *tx* must *unconditionally* be both an integer and a pair. It would be well-typed under a more liberal version of P-CSTR where type information is propagated in a left-to-right fashion: then, *tx*'s second component would be required to be an integer (resp. a pair) only when *tx*'s first component is an application of *Int* (resp. *Pair*).

It would be possible to prove that this relaxed version of P-CSTR is still consistent with the operational semantics given in §3, that is, it does not compromise type safety. So, why do we reject it?

The reason is as follows: suppose we adopt the relaxed rule, and view the above version of *print* as well-typed. Then, we must ensure that the compiler does not generate code that begins by examining the *second* component of the pair *tx* and blindly dereferences it, without checking whether it is an integer or a pair, to access x_1 and x_2. There seem to be two ways of guaranteeing this:

- either specify, in some way, that tuples are examined in a left-to-right manner;
- or allow integers and pairs to be distinguished at runtime.

The first option appears *ad hoc* (why left-to-right, rather than right-to-left, or some other strategy?). The second option requires every value to carry a type tag at runtime, which is unnecessary in ML, and undesirable for efficiency reasons.

One should perhaps point out that the semantics of pattern matching given in §3 *does* assume that values have unambiguous runtime representations, since (for instance) it specifies that K_1 does not match K_2, *even* if these (distinct) data constructors belong to *distinct* algebraic data types.

In ML, however, the type system enjoys the often unstated property that one never attempts, at runtime, to match K_1 against K_2 unless both are associated with the *same* algebraic data type. This property, which is stated by Lemma 4.40 in the present paper, is the reason why values need not carry runtime tags that identify their type. Although adopting the second option above would preserve type safety, it would violate this property, leading to a less efficient compilation scheme.

These somewhat subtle considerations are the reason why we stick to the strict version of p-Cstr. This choice in turn has implications on the design of the type inference algorithm. Indeed, adopting the relaxed version of p-Cstr would lead to different, perhaps simpler, constraint generation rules.

One should also point out that none of these problems arises if the language does not have *nested* patterns. Indeed, in a language where patterns are *shallow*, the above version of *print* cannot be written. Instead of a single, complex test, the programmer is in fact forced to perform a cascade of simple tests, where the ordering between tests is explicit. This eliminates the problem.◇

Remark 4.29. It is interesting to study how the type system degenerates when all data types are ordinary (as opposed to guarded) algebraic data types, that is, when every data constructor has a declaration of the form $K :: \forall\bar{\alpha}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$. Then, in every instance of p-Cstr, $\bar{\beta}$ and D must be $\varnothing$ and true, respectively, so that the rule may be written:

$$\text{p-Cstr}\quad \frac{\forall i \quad C \vdash p_i : \tau_i \rightsquigarrow \Delta_i \qquad K :: \forall\bar{\alpha}.\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})}{C \vdash K\, p_1 \cdots p_n : \varepsilon(\bar{\alpha}) \rightsquigarrow (\Delta_1 \times \cdots \times \Delta_n)}$$

Then, by exploiting the properties stated in Remark 4.26, one may prove that p-SubOut and p-Hide may be suppressed from the type system without affecting the set of valid judgments about expressions.

Let us also remove the pattern 0 from the language, since it does not exist in ML. Then, in the absence of 0 and of p-SubOut, and under the simplified version of p-Cstr above, all environment fragments must have the form $\exists\varnothing[\mathsf{true}]\Gamma$. Thus, judgments about patterns may take the simplified form $C \vdash p : \tau \rightsquigarrow \Gamma$, where Γ is a simple environment. This in turn allows simplifying Clause as follows:

$$\text{Clause}\quad \frac{C \vdash p : \tau' \rightsquigarrow \Gamma' \qquad C, \Gamma\Gamma' \vdash e : \tau}{C, \Gamma \vdash p.e : \tau' \to \tau}$$

This is a standard rule in HM(X): the expression e is typechecked in an environment extended with new bindings, but no fresh type variables are introduced, and the constraint assumption remains unchanged. ◇

4.7 Type soundness

We now establish several properties of the type system HMG(X), beginning with some standard weakening and normalization lemmas, and culminating with subject reduction and progress theorems.

Lemma 4.30 (Weakening). *Assume $C_1 \Vdash C_2$. If $C_2 \vdash p : \tau \rightsquigarrow \Delta$ (resp. $C_2, \Gamma \vdash ce : \sigma$) is derivable, then there exists a derivation of $C_1 \vdash p : \tau \rightsquigarrow \Delta$ (resp. $C_1, \Gamma \vdash ce : \sigma$) of the same structure.* ◇ Proof on page 50

LEMMA 4.31. *$C, \Gamma \vdash e : \sigma$ implies $C \Vdash \exists\sigma$.* ◇ Proof on page 50

Next come three auxiliary normalization lemmas. They are standard: they come unmodified from the theory of HM(X). The terminology employed in these statements was defined in Remark 4.26.

LEMMA 4.32. *The shape* INST(GEN($\cdot$)) *may be replaced with* HIDE(SUB($\cdot$)). ◇ Proof on page 50

LEMMA 4.33. *The shape* HIDE(GEN($\cdot$)) *may be replaced with* GEN(HIDE($\cdot$)). ◇ Proof on page 51

LEMMA 4.34. APP(SUB(ABS($\cdot_1$)), $\cdot_2$) *may be replaced with* SUB(APP(ABS($\cdot_1$), SUB($\cdot_2$))). ◇ Proof on page 51

Building upon these lemmas, we now establish the main normalization result. An instance of INST or GEN is *trivial* if its conclusion is identical to its premise. A typing derivation is *normal* if and only if (a) there are no trivial instances of INST or GEN; (b) every instance of GEN appears either at the root of the derivation or as a premise of a syntax-directed rule; (c) every instance of HIDE appears either at the root of the derivation or as a premise of GEN; and (d) at every subexpression of the form $(\lambda\bar{c})\, e$, ABS and APP are consecutive, that is, they are never separated by an instance of a non-syntax-directed rule.

LEMMA 4.35 (NORMALIZATION). *Every valid typing judgment admits a normal derivation.* ◇ Proof on page 51

We now prove that HMG(X) is sound, via Wright and Felleisen's syntactic approach (1994). We establish a few technical results, then give *subject reduction* and *progress* theorems. We begin with a basic substitution lemma, whose proof is straightforward:

LEMMA 4.36 (SUBSTITUTION). *$C, \Gamma[x \mapsto \sigma'] \vdash ce : \sigma$ and $C, \varnothing \vdash e : \sigma'$ imply $C, \Gamma \vdash [x \mapsto e]ce : \sigma$.* ◇ Proof on page 51

Next comes the key technical lemma that helps establishing subject reduction for pattern matching. We state it first, and explain it next.

LEMMA 4.37. *Assume v matches p and $C, \varnothing \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \Delta$ hold. Write Δ as $\exists\bar{\beta}[D]\Gamma$, where $\bar{\beta} \# \mathrm{ftv}(C)$. Then, there exists a constraint H such that $H \Vdash D$ and $C \equiv \exists\bar{\beta}.H$ and, for every $x \in \mathrm{dpv}(p)$, $H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$ holds.* ◇ Proof on page 52

To explain this complex statement, it is best to first consider the simple case where $\bar{\beta}$ is empty and D is true. In that case, we have $C \equiv H$. Thus, the lemma's statement may be specialized as follows: *if v matches p and $C, \varnothing \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \Gamma$ hold, then, for every $x \in \mathrm{dpv}(p)$, $C, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$ holds.* In other words, the value that x receives when matching v against p does indeed have the type that was predicted.

In the general case, the idea remains the same, but the statement must account for the abstract types $\bar{\beta}$. It still holds that $[p \mapsto v]x$ has type $\Gamma(x)$, albeit under a constraint H, which extends C with information about the type variables $\bar{\beta}$, as stated by the property $C \equiv \exists\bar{\beta}.H$. The exact amount of extra information carried by H is unknown, but is strong enough to guarantee that D holds, as stated by the property $H \Vdash D$.

$$
\begin{aligned}
\neg 0 &= 1 \\
\neg 1 &= 0 \\
\neg x &= 0 \\
\neg(K\, p_1 \cdots p_n) &= (\vee_{i\in[1,n]}\, K\, 1 \cdots 1 \cdot \neg p_i \cdot 1 \cdots 1) \\
&\quad \vee (\vee_{K'\sim K, K'\neq K}\, K'\, 1 \cdots 1) \\
\neg(p_1 \vee p_2) &= \neg p_1 \wedge \neg p_2 \\
\neg(p_1 \wedge p_2) &= \neg p_1 \vee \neg p_2
\end{aligned}
$$

Figure 7: Computing the complement of a pattern

Remark 4.38. The statement of Lemma 4.37 guarantees that $H \Vdash C \wedge D$ holds. One might wonder whether it could be simplified by requiring H to coincide with $C \wedge D$. It turns out that it cannot. Consider, for instance, an unparameterized algebraic data type ε, whose sole data constructor is $K :: \forall\beta.\beta \to \varepsilon$. (The type ε is an encoding of the existential type $\exists\beta.\beta$ in the style of Läufer and Odersky (1994).) Let v be $K\,3$, where 3 has type *int*; let p be $K\,x$; let C and D be true; let τ be ε; let Δ be $\exists\beta.(x : \beta)$. Then, the hypotheses of Lemma 4.37 are met, so there must exist a constraint H such that $H, \varnothing \vdash 3 : \beta$ holds. It is obvious that taking $H \equiv C \wedge D \equiv \mathsf{true}$ will not do, because 3 does not have type β for an arbitrary β. In fact, the weakest choice that meets this requirement is $H \equiv \mathit{int} \leq \beta$. This choice turns out to be acceptable. Indeed, the other two goals of the lemma, namely $\mathit{int} \leq \beta \Vdash \mathsf{true}$ and $\exists\beta.(\mathit{int} \leq \beta) \equiv \mathsf{true}$, are also met. In short, the constraint H provides a link between the abstract type (here, the type variable β) and its underlying concrete representation (here, the type *int*). ◇

Using the previous lemmas, it is possible to give a reasonably concise proof of subject reduction.

THEOREM 4.39 (SUBJECT REDUCTION). *$C, \varnothing \vdash e : \sigma$ and $e \to e'$ imply $C, \varnothing \vdash e' : \sigma$.* ◇ Proof on page 54

We now turn to the proof of the progress theorem. In programming languages equipped with pattern matching, such as ML, it is well-known that well-typedness alone does not ensure progress: indeed, a well-typed β-redex $(\lambda p_1.e_1 \cdots p_n.e_n)\, v$ may still be irreducible if none of $p_1, \ldots, p_n$ matches v. For this reason, we first establish progress under the assumption that every case analysis is *exhaustive*, as determined by a simple syntactic criterion. Then, we show how, in the presence of guarded algebraic data types, this criterion may be refined so as to take type information into account.

Our syntactic criterion for exhaustiveness is standard: it is, in fact, identical to that of ML. It uses almost no type information: it only requires being able to determine whether two data constructors K and K' are associated with the same algebraic data type ε. (We write $K \sim K'$ when they are.) It relies on the notion of *complement* of a pattern, which is standard (Xi, 2003) and whose definition is recalled in Figure 7. A case analysis $\lambda(p_1.e_1 \cdots p_n.e_n)$ is said to be *exhaustive* if and only if the pattern $\neg(p_1 \vee \cdots \vee p_n)$ is empty. How to determine whether a pattern is empty was discussed in §3.3.

It is important to note that the pattern $p \vee \neg p$ is in general *not* equivalent to 1: this is due to the definition of $\neg(K\, p_1 \cdots p_n)$, where *only* the data constructors compatible with K are enumerated. For instance, because the two data constructors associated with the algebraic data type constructor *ty* are *Int* and *Pair* (§2.1), we have $\mathit{Int} \vee \neg \mathit{Int} = \mathit{Int} \vee \mathit{Pair}\, 1 \cdot 1 \neq 1$.

The next lemma uses the type system to work around this difficulty. It guarantees that, if p has type τ, then $p \vee \neg p$ matches every value of type τ. In other words, in a well-typed program, the values that are matched against a pattern p cannot be arbitrary: they are guaranteed to match $p \vee \neg p$. This property allows dispensing with runtime type tags; this issue was discussed in Remark 4.28.

The hypotheses of the lemma are analogous to those of Lemma 4.37. It is, however, oriented towards proving progress, rather than subject reduction.

LEMMA 4.40. *If $C, \varnothing \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \Delta$ hold, where C is satisfiable, then v matches $p \vee \neg p$.* ◇ Proof on page 55

It is now straightforward to establish progress, under the hypothesis that every case analysis is exhaustive.

LEMMA 4.41. *If $E[e]$ is well-typed, then so is e.* ◇ Proof on page 56

THEOREM 4.42 (PROGRESS). *If e is well-typed and contains exhaustive case analyses only, then it is either reducible or a value.* ◇ Proof on page 56

A closed expression e is *stuck* if it is neither reducible nor a value; it is said to *go wrong* if it reduces to a stuck expression. We may now state a first type soundness result:

THEOREM 4.43 (TYPE SOUNDNESS). *If e is well-typed and contains exhaustive case analyses only, then it does not go wrong.* ◇ Proof on page 57

As promised earlier, we now turn to the definition of a more precise exhaustiveness criterion. In ML, nonexhaustive case analyses are either rejected or silently made exhaustive by extending them with a default clause whose right-hand side triggers a runtime error. In the presence of guarded algebraic data types, however, this purely syntactic criterion becomes unsatisfactory: although it remains correct, one can do better.

Indeed, the type assigned to a function may allow determining that some branches can never be taken: this is what Xi (1999) refers to as *dead code elimination*. For instance, the function $\lambda Int.3$ is not exhaustive, as per our syntactic criterion, because $\neg Int$ is $Pair\,1 \cdot 1$, which is nonempty. However, if the function is declared to have type $ty(int) \to int$, then pattern matching cannot fail, because no value of type $ty(int)$ matches $Pair\,1 \cdot 1$. If we were to extend the function with a clause guarded by the pattern $Pair\,1 \cdot 1$, then the right-hand side of that clause would be typechecked under the assumption $\exists \beta_1 \beta_2.(int = \beta_1 \times \beta_2)$, which is absurd, that is, equivalent to false. This allows the typechecker to recognize that such a clause is superfluous.

Thus, we proceed as follows: prior to typechecking, we automatically complete every case analysis with a default clause, so as to make it exhaustive. The right-hand side of every default clause consists of a special expression $\bot$, which is irreducible, but not a value: it is stuck, and models a runtime error. To statically prevent these runtime errors and preserve type safety, we ensure that $\bot$ is never well-typed: its associated typing rule is

$$\textsc{Dead}\quad \mathsf{false}, \Gamma \vdash \bot : \sigma$$

Thus, checking that the completed case analysis, as a whole, is well-typed, guarantees that the newly inserted default clause can never be selected at runtime. This in turn means that no code needs be generated for it: it only exists in the typechecker's eyes, not in the compiler's.

To formalize this discussion, let $\lfloor \cdot \rfloor$ be the procedure that completes every case analysis with a default clause, defined by letting

$$\lfloor \lambda(p_1.e_1 \cdots p_n.e_n) \rfloor = \lambda(p_1.\lfloor e_1 \rfloor \cdots p_n.\lfloor e_n \rfloor \cdot \neg(p_1 \vee \cdots \vee p_n).\bot)$$

and letting $\lfloor \cdot \rfloor$ be a homomorphism with respect to all other expression forms. Then, we may revisit the type soundness result as follows:

THEOREM 4.44 (PROGRESS REVISITED). *If $\lfloor e \rfloor$ is well-typed, then e is either reducible or a value.* ◇ Proof on page 57

THEOREM 4.45 (TYPE SOUNDNESS REVISITED). *If $\lfloor e \rfloor$ is well-typed then e does not go wrong.*◇ Proof on page 57

Let us stress that, according to Theorem 4.45, typechecking the modified program $\lfloor e \rfloor$, where every case analysis has been completed with a default clause, guarantees type soundness for the *original* program e. The syntactic notion of exhaustiveness defined earlier is no longer involved in this statement.

The ideas presented here are not new: see Xi (1999; 2003). However, a formal type soundness statement for a type system equipped with guarded algebraic data types and pattern matching does not seem to exist in the literature; Theorem 4.45 fills this gap.

Remark 4.46. One issue was left implicit in the above discussion: is our new, type-based criterion always at least as precise as the previous, syntactic one? The answer is positive, provided the pattern $\neg(p_1 \vee \cdots \vee p_n)$, which guards the default clause in the definition of $\lfloor \cdot \rfloor$, is *normalized* as per the rules of Figure 4. Indeed, consider a function $e = \lambda(p_1.e_1 \cdots p_n.e_n)$, and assume it is exhaustive, that is, $\neg(p_1 \vee \cdots \vee p_n)$ is empty. Then, applying the above procedure, we have $\neg(p_1 \vee \cdots \vee p_n) \rightsquigarrow^* 0$, so $\lfloor e \rfloor$ is $\lambda(p_1.\lfloor e_1 \rfloor \cdots p_n.\lfloor e_n \rfloor \cdot 0.\bot)$. Then, because $C, \Gamma \vdash 0.\bot : \tau_1 \to \tau_2$ holds for all C, Γ, τ_1 and τ_2, one may check that e and $\lfloor e \rfloor$ admit the same typings. ◇

Remark 4.47. More generally, normalizing patterns and *resolving sequentiality* (Xi, 2003) may improve the accuracy of type checking in the presence of guarded algebraic data types. These transformations may, however, cause an exponential increase in the size of patterns, and introduce many new disjunction patterns, so it is unclear whether they should be performed automatically or upon request of the programmer. ◇

Remark 4.48. In practice, $\lfloor e \rfloor$ may be ill-typed because the typechecker is unable to prove that some of the inserted $\bot$'s are unreachable. In that case, one can choose to accept the program, provided these instances of $\bot$ are compiled down to code that generates a runtime error. In ML, this concerns all instances of $\bot$, except those that are guarded by the pattern 0. In HMG(X), branches guarded by patterns other than 0 may be recognized by the typechecker as dead code: indeed, any pattern that gives rise to an empty environment fragment (one that contains the constraint false) qualifies. ◇

5 Type inference

We now turn to type inference, with the aim of reducing type inference to constraint solving.

Due to the presence of polymorphic recursion, well-typedness in HMG(X) is undecidable (Henglein, 1993). Thus, to begin, we restrict the language by requiring every μ-bound variable to be

explicitly annotated with a type scheme. This restriction is not necessary for type soundness, which explains why it was not made earlier.

Furthermore, looking ahead towards §6, it is also useful to require that every λ-abstraction carry an explicit type annotation. A cheap way of achieving this effect is to merge the μ and λ binders into a single construct, so that only one type annotation has to be written. Since, in practice, the μ binder is mainly used to define functions, this design choice seems acceptable.

Remark 5.1. Instead of merging μ and λ into a single construct, one could also keep them separate and require each of them to carry a type annotation. A simple form of *local type inference* (Pierce and Turner, 2000) could then be used to propagate a type annotation from one to the other when they are adjacent, allowing the user to provide only one type annotation in that case. We come back to this point in Remark 6.11. ◇

Thus, the language of expressions becomes:

$$e ::= x \mid \mu(x : \exists\bar{\beta}.\sigma).\lambda\bar{c} \mid K\,\bar{e} \mid e\,e \mid \mathsf{let}\,x = e\,\mathsf{in}\,e$$

Without loss of generality, we require the type scheme σ carried by every μ construct to be of the form $\forall\bar{\gamma}[C].\tau_1 \to \tau_2$, that is, to syntactically present an arrow type. We do *not* require σ to be closed. Instead, σ may have free type variables, which must be included in $\bar{\beta}$, so that the type annotation $\exists\bar{\beta}.\sigma$ is closed. (It would be straightforward to suppress the prefix $\exists\bar{\beta}$, allowing σ's free type variables to be bound elsewhere. This is an orthogonal issue.)

Not requiring σ to be closed seems important, for a couple of different reasons. The main reason is that this allows defining $\mu x.\lambda\bar{c}$ as syntactic sugar for $\mu(x : \exists\beta_1\beta_2.\beta_1 \to \beta_2).\lambda\bar{c}$. In other words, unannotated functions are still part of the language. They carry the uninformative type annotation $\exists\beta_1\beta_2.\beta_1 \to \beta_2$, which means *some (monomorphic) function type*. Note, however, that functions that exploit polymorphic recursion must carry a truly explicit, nontrivial type annotation. The second reason is that some functions do not admit a closed type scheme. This is often the case for functions that are nested inside another, larger function: see, for instance, `rmap_f` in §2. Unfortunately, in §6, we shall be led to further restrict the shape of type annotations, causing `rmap_f` to be rejected.

The typing rules Abs and Fix are replaced with the following new rule, a combination of Abs, Gen, and Fix, where the type scheme assigned to x is taken from the annotation instead of being guessed. This is the key point that makes type inference decidable again.

$$\textsc{FixAbs}\quad \frac{\forall i \quad C \wedge D, \Gamma[x \mapsto \sigma] \vdash c_i : \tau \qquad \bar{\alpha} \# \mathrm{ftv}(C, \Gamma) \qquad \sigma = \forall\bar{\alpha}[D].\tau}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).\lambda\bar{c} : \sigma}$$

Because the modified type system is a restriction of the original one, it is still sound. In the following, we show that type inference for it may be reduced to constraint solving.

5.1 Patterns

We begin our treatment of type inference by defining a procedure that computes principal typing judgments for patterns. It consists of two functions of a pattern p and a type τ, given in Figure 8. As usual, the new type variables that appear in the right-hand side of an equation must be chosen

Patterns (constraint generation)

$$
\begin{array}{rcl}
(\!|0 \downarrow \tau|\!) & = & \mathsf{true} \\
(\!|1 \downarrow \tau|\!) & = & \mathsf{true} \\
(\!|x \downarrow \tau|\!) & = & \mathsf{true} \\
(\!|p_1 \wedge p_2 \downarrow \tau|\!) & = & (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \\
(\!|p_1 \vee p_2 \downarrow \tau|\!) & = & (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \\
(\!|K\, p_1 \cdots p_n \downarrow \tau|\!) & = & \exists \bar{\alpha}.(\varepsilon(\bar{\alpha}) = \tau \wedge \forall \bar{\beta}.D \Rightarrow \wedge_i (\!|p_i \downarrow \tau_i|\!)) \\
 & & \textit{where } K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})
\end{array}
$$

Patterns (environment fragment generation)

$$
\begin{array}{rcl}
(\!|0 \uparrow \tau|\!) & = & \exists \varnothing[\mathsf{false}]\varnothing \\
(\!|1 \uparrow \tau|\!) & = & \exists \varnothing[\mathsf{true}]\varnothing \\
(\!|x \uparrow \tau|\!) & = & \exists \varnothing[\mathsf{true}](x \mapsto \tau) \\
(\!|p_1 \wedge p_2 \uparrow \tau|\!) & = & (\!|p_1 \uparrow \tau|\!) \times (\!|p_2 \uparrow \tau|\!) \\
(\!|p_1 \vee p_2 \uparrow \tau|\!) & = & (\!|p_1 \uparrow \tau|\!) + (\!|p_2 \uparrow \tau|\!) \\
(\!|K\, p_1 \cdots p_n \uparrow \tau|\!) & = & \exists \bar{\alpha}\bar{\beta}[\varepsilon(\bar{\alpha}) = \tau \wedge D](\times_i (\!|p_i \uparrow \tau_i|\!)) \\
 & & \textit{where } K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})
\end{array}
$$

Figure 8: Type inference for patterns

fresh for the parameters that appear on its left-hand side. Here, in the last equation of each group, $\bar{\alpha}$ and $\bar{\beta}$ must be fresh for τ.

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* (Pierce and Turner, 2000).)

The first three rules of each group directly reflect P-Empty, P-Wild, and P-Var.

The fourth rules of the first and second groups directly reflect P-And. The former states that it is legal to match a value of type τ against $p_1 \wedge p_2$ if and only if it is legal to match such a value against p_1 and against p_2 separately. The latter rule states that the knowledge thus obtained is the conjunction of the knowledge obtained by matching against p_1 and p_2 separately.

The fifth rules of the first and second groups reflect P-Or. More precisely, they directly reflect the variant of P-Or that was alluded to in Remark 4.23. The latter states that the knowledge obtained by matching a value against $p_1 \vee p_2$ is the disjunction of the knowledge obtained by matching against p_1 and p_2 separately. This is our first use of the fragment disjunction operator $+$.

The last rule of the first group may be read as follows: *it is legal to match a value of type τ against $K\, p_1 \cdots p_n$ if and only if, for some types $\bar{\alpha}$, τ is of the form $\varepsilon(\bar{\alpha})$ and, for all types $\bar{\beta}$ that satisfy D and for every $i \in \{1, \ldots, n\}$, it is legal to match a value of type τ_i against p_i.* The use of

universal quantification and of implication encodes the fact that the types $\bar{\beta}$ must be considered abstract, but may safely be assumed to satisfy D.

The last rule of the second group records the knowledge that, if $K\,p_1 \cdots p_n$ matches a value of type τ, then, for some types $\bar{\alpha}$ and $\bar{\beta}$, τ is of the form $\varepsilon(\bar{\alpha})$ and D is satisfied. This knowledge is combined, using the fragment conjunction operator, with that obtained by successfully matching the value against the subpatterns p_i.

The last two rules may be simplified when the expected type τ happens to be of the desired form, that is, of the form $\varepsilon(\bar{\alpha})$. This is stated by the next lemma.

LEMMA 5.2. *Assume* $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$. *Then, the two constraints* $(\!|K\,p_1 \cdots p_n \downarrow \varepsilon(\bar{\alpha})|\!)$ *and* $\forall\bar{\beta}.D \Rightarrow \wedge_i(\!|p_i \downarrow \tau_i|\!)$ *are equivalent. Furthermore, the two environment fragments* $(\!|K\,p_1 \cdots p_n \uparrow \varepsilon(\bar{\alpha})|\!)$ *and* $\exists\bar{\beta}[D](\times_i(\!|p_i \uparrow \tau_i|\!))$ *are equivalent.* ◇ Proof on page 57

Example 5.3. It is easy to check that the constraint $(\!|Int \downarrow ty(\alpha)|\!)$ is equivalent to true. Thus, it is legal to match a value of type $ty(\alpha)$ against the pattern Int, for an arbitrary α. Furthermore, the environment fragment $(\!|Int \uparrow ty(\alpha)|\!)$ is $\exists\alpha'[ty(\alpha') = ty(\alpha) \wedge \alpha' = int]\varnothing$, which is in fact equivalent to $\exists\varnothing[\alpha = int]\varnothing$. These results are consistent with the first derivation given in Example 4.21.

Here is another example. By Lemma 5.2, we find that $(\!|Pair(t_1, t_2) \downarrow ty(\alpha)|\!)$ is equivalent to

$$\forall\beta_1\beta_2.(\alpha = \beta_1 \times \beta_2) \Rightarrow ((\!|t_1 \downarrow ty(\beta_1)|\!) \wedge (\!|t_2 \downarrow ty(\beta_2)|\!))$$

Since every $(\!|t_i \downarrow ty(\beta_i)|\!)$ is true, the whole constraint is equivalent to true. Thus, it is valid to match a value of type $ty(\alpha)$ against the pattern $Pair\,(t_1, t_2)$. Similarly, $(\!|Pair(t_1, t_2) \uparrow ty(\alpha)|\!)$ is equivalent to $\exists\beta_1\beta_2[\alpha = \beta_1 \times \beta_2](t_1 : ty(\beta_1); t_2 : ty(\beta_2))$. These results are, again, consistent with the second derivation in Example 4.21. ◇

Lemmas 5.4 and 5.6 state that the rules give rise to judgments that are both *correct* and *complete* (that is, principal), respectively. To establish completeness, we exploit the auxiliary Lemma 5.5, which states that, under the assumption that τ and τ' are equal, they are interchangeable for the purposes of constraint generation.

LEMMA 5.4 (CORRECTNESS). $(\!|p \downarrow \tau|\!) \vdash p : \tau \rightsquigarrow (\!|p \uparrow \tau|\!)$. ◇ Proof on page 57

LEMMA 5.5. $\tau = \tau' \wedge (\!|p \downarrow \tau|\!) \Vdash (\!|p \downarrow \tau'|\!)$ *and* $\tau = \tau' \Vdash (\!|p \uparrow \tau|\!) \leq (\!|p \uparrow \tau'|\!)$ *hold.* ◇ Proof on page 58

LEMMA 5.6 (COMPLETENESS). $C \vdash p : \tau \rightsquigarrow \Delta$ *implies* $C \Vdash (\!|p \downarrow \tau|\!)$ *and* $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$. ◇ Proof on page 58

5.2 Expressions and clauses

Let us now turn to expressions and clauses. Given an environment Γ, an expression e and an expected type τ, the constraint $(\!|\Gamma \vdash e : \tau|\!)$ is intended to represent a necessary and sufficient condition for e to have type τ under environment Γ. Its definition appears in Figure 9. Again, the new type variables that appear in the right-hand side of an equation must be chosen fresh for the parameters that appear on its left-hand side.

The rules that govern expressions are standard: see, for instance, Sulzmann et al. (1999), Simonet (2003a), or Pottier and Rémy (2005).

The first rule, which deals with a variable x, requires the expected type τ to be an instance of the type scheme $\Gamma(x)$.

Expressions

$$
\begin{aligned}
(\!|\Gamma \vdash x : \tau|\!) &= \Gamma(x) \leq \tau \\
(\!|\Gamma \vdash e_1\, e_2 : \tau|\!) &= \exists\alpha.((\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!)) \\
(\!|\Gamma \vdash K\, e_1 \cdots e_n : \tau|\!) &= \exists\bar{\alpha}\bar{\beta}.(\wedge_i (\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau) \\
&\quad \textit{where } K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha}) \\
(\!|\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau|\!) &= (\!|\Gamma[x \mapsto \forall\alpha[C].\alpha] \vdash e_2 : \tau|\!) \wedge \exists\alpha.C \\
&\quad \textit{where } C \textit{ is } (\!|\Gamma \vdash e_1 : \alpha|\!) \\
(\!|\Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).\lambda\bar{c} : \tau|\!) &= \exists\bar{\beta}.((\forall\bar{\gamma}.C \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash \bar{c} : \tau_1 \to \tau_2|\!)) \wedge \sigma \leq \tau) \\
&\quad \textit{where } \sigma \textit{ is } \forall\bar{\gamma}[C].\tau_1 \to \tau_2
\end{aligned}
$$

Clauses

$$
\begin{aligned}
(\!|\Gamma \vdash p.e : \tau_1 \to \tau_2|\!) &= (\!|p \downarrow \tau_1|\!) \wedge \forall\bar{\beta}.D \Rightarrow (\!|\Gamma\Gamma' \vdash e : \tau_2|\!) \\
&\quad \textit{where } \exists\bar{\beta}[D]\Gamma' \textit{ is } (\!|p \uparrow \tau_1|\!)
\end{aligned}
$$

Figure 9: Type inference for expressions and clauses

The second rule, which deals with an application $e_1\, e_2$, ensures that the domain type of the function e_1 matches the type of the argument e_2 by using the fresh type variable α to stand for both of them.

The third rule may be viewed as a particular case of the previous two. Indeed, a data constructor application is dealt with in the same way as an application of a variable to n arguments. The only difference resides in the fact that the type scheme associated with K is fixed instead of found in the environment Γ.

The fourth rule deals with let-polymorphism. It typechecks e_2 in an environment extended with a binding of x to the type scheme $\forall\alpha[(\!|\Gamma \vdash e_1 : \alpha|\!)].\alpha$, which is a principal type scheme for e_1.

In the fifth rule, we write $(\!|\Gamma \vdash \bar{c} : \tau_1 \to \tau_2|\!)$ for $\wedge_i (\!|\Gamma \vdash c_i : \tau_1 \to \tau_2|\!)$ when $\bar{c}$ is $(c_1, \ldots, c_n)$. The rule implements polymorphic recursion by ensuring that the clauses $\bar{c}$ have type scheme σ under the hypothesis that x has type scheme σ. The context $\forall\bar{\gamma}.C \Rightarrow []$ is used to encode this requirement. The expected type τ is required to be an instance of σ. The type variables $\bar{\beta}$, whose value was not specified by the user, are existentially bound, so it is up to the constraint solver to determine their value.

Remark 5.7. For unannotated functions of the form $\mu x.\lambda\bar{c}$, which were defined to be syntactic sugar for $\mu(x : \exists\beta_1\beta_2.\beta_1 \to \beta_2).\lambda\bar{c}$, this gives rise to the following derived rule:

$$(\!|\Gamma \vdash \mu x.\lambda\bar{c} : \tau|\!) = \exists\beta_1\beta_2.((\!|\Gamma[x \mapsto \beta_1 \to \beta_2] \vdash \bar{c} : \beta_1 \to \beta_2|\!) \wedge \beta_1 \to \beta_2 \leq \tau)$$

This is a standard rule for monomorphic, recursive functions. ◇

We now turn to the last rule, which deals with clauses, and where most of the novelty resides. 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 $(\!|p \uparrow \tau_1|\!)$.

Remark 5.8. This rule does not exploit the presence of a type annotation at every λ-abstraction. Indeed, its parameters are τ_1 and τ_2 only. The information carried by $\bar{\gamma}$ and C, which are also supplied by the programmer at every "$\mu\lambda$" construct, is not used. The modified constraint generation rules in §6 do take advantage of this extra information. ◇

Example 5.9. Here is the constraint generated for the first clause in the definition of *print*, at type $ty(\alpha) \to \alpha \to unit$. As in Example 4.22, we assume that the environment Γ assigns type $int \to unit$ to the variable *print_int*. We implicitly exploit the results of Example 5.3. We write e_1 for $\lambda x.print\_int\, x$.

$$\begin{aligned}&(\!|\Gamma \vdash Int.e_1 : ty(\alpha) \to \alpha \to unit|\!)\\ &\quad\equiv\ \mathsf{true} \wedge \alpha = int \Rightarrow (\!|\Gamma \vdash e_1 : \alpha \to unit|\!)\end{aligned}$$

It is easy to check that the subconstraint $(\!|\Gamma \vdash \lambda x.print\_int\, x : \alpha \to unit|\!)$ is equivalent to $\alpha \leq int$. Indeed, for x to be a valid argument to *print_int*, its type must be a subtype of *int*. So, the above constraint reduces to $\alpha = int \Rightarrow \alpha \leq int$, which is equivalent to true.

Next, here is the constraint generated for the second clause in the definition of *print*, at type $ty(\alpha) \to \alpha \to unit$. (As in Example 4.22, the intermediate call to *print_string* is omitted for brevity.) We assume that the environment Γ assigns type scheme $\forall\alpha.ty(\alpha) \to \alpha \to unit$ to *print*, so as to be able to typecheck the recursive calls to *print*, and again implicitly exploit the results of Example 5.3. We write e_2 for $\lambda(x_1, x_2).(print\, t_1\, x_1; print\, t_2\, x_2)$.

$$\begin{aligned}&(\!|\Gamma \vdash Pair\,(t_1, t_2).e_2 : ty(\alpha) \to \alpha \to unit|\!)\\ &\quad\equiv \mathsf{true} \wedge \forall\beta_1\beta_2.\alpha = \beta_1 \times \beta_2 \Rightarrow (\!|\Gamma \vdash e_2 : \alpha \to unit|\!)\end{aligned}$$

Again, it can be checked that this constraint is equivalent to true.

The constraint generated for the entire function $\mu print.\ldots.$, in the environment Γ_0 of Example 4.22 and at a fresh type variable γ, is the following:

$$\begin{aligned}(\!|\Gamma_0 \vdash \mu print.\ldots. : \gamma|\!) \equiv \quad \forall\alpha.(\quad & \alpha = int \quad \Rightarrow \quad (\!|\Gamma \vdash e_1 : \alpha \to unit|\!)\\ & \wedge \forall\beta_1\beta_2.\alpha = \beta_1 \times \beta_2 \quad \Rightarrow \quad (\!|\Gamma \vdash e_2 : \alpha \to unit|\!))\\ & \wedge \exists\alpha.ty(\alpha) \to \alpha \to unit \leq \gamma\end{aligned}$$

The first part of the constraint, delimited by the universal quantifier $\forall\alpha$, ensures that the function admits the type scheme provided by the programmer, that is, $\forall\alpha.ty(\alpha) \to \alpha \to unit$. Each implication corresponds to one clause of the function. The second part of the constraint, delimited by the existential quantifier $\exists\alpha$, constrains the expected type γ to be an instance of this type scheme. ◇

There remains to prove that the constraint generation rules for expressions and clauses are correct and complete. Correctness is straightforward:

THEOREM 5.10 (CORRECTNESS). $(\!|\Gamma \vdash ce : \tau|\!), \Gamma \vdash ce : \tau$. ◇ Proof on page 58

The next two auxiliary lemmas state that $(\!|\Gamma \vdash ce : \tau|\!)$ is contravariant in Γ and covariant in τ. In other words, if the expected type is less precise, or if the environment is more precise, then the generated constraint is less restrictive.

LEMMA 5.11. $(\!|\Gamma \vdash ce : \tau|\!) \wedge \tau \leq \tau' \Vdash (\!|\Gamma \vdash ce : \tau'|\!)$. ◇ Proof on page 59

LEMMA 5.12. $\Gamma' \leq \Gamma \wedge (\!|\Gamma \vdash ce : \tau|\!) \Vdash (\!|\Gamma' \vdash ce : \tau|\!)$. ◇ Proof on page 59

The next auxiliary lemma states that the rule that deals with clauses exploits the environment fragment generated by invoking $(\!|p \uparrow \tau_1|\!)$ in a contravariant manner. In other words, if the environment fragment is more precise, then the generated constraint is less restrictive.

LEMMA 5.13. *Assume* $\bar{\beta}_1\bar{\beta}_2 \# \mathrm{ftv}(\Gamma, \tau)$. *We have* $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \forall\bar{\beta}_1.D_1 \Rightarrow (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$. ◇ Proof on page 59

These lemmas allow establishing completeness:

THEOREM 5.14 (COMPLETENESS). $C, \Gamma \vdash ce : \forall\bar{\alpha}[D].\tau$ *and* $\bar{\alpha} \# \mathrm{ftv}(\Gamma)$ *imply* $C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash ce : \tau|\!)$. ◇ Proof on page 59

Using Theorems 5.10 and 5.14, as well as Lemma 4.31, it is easy to prove that e is well-typed under environment Γ if and only if the constraint $\exists\alpha.(\!|\Gamma \vdash e : \alpha|\!)$, where α is fresh for Γ, is satisfiable. Thus, we have reduced type inference to constraint solving.

Type inference for HM(X) is usually reduced to constraint solving for a logic that includes basic predicates (such as subtyping), conjunction, and existential quantification. Here, we make use of more first-order connectives, including universal quantification and implication.

Nevertheless, this is enough to show that type inference for some instances of HMG(X) is decidable. For instance, assuming no basic predicates other than subtyping are available, and assuming subtyping is structural, constraint solving is decidable (Kuncak and Rinard, 2003). This includes the important case where subtyping is in fact interpreted as equality.

However, such a result remains weak, because the complexity of constraint solving for the first-order theory of equality is nonelementary (Vorobyov, 1996). To strengthen our result, we must work a bit more and ensure that we actually generate *tractable* constraints. This is the topic of §6.

6 Tractable type inference

Roughly speaking, one may identify three sources of complexity in the constraints that we generate.

One is that we have made the entire constraint language available to the programmer. Indeed, the constraints supplied by the programmer as part of data constructor declarations or type annotations eventually become components of the constraint generated by the type inference algorithm. Fortunately, we are willing to restrict the constraint language that is available to the programmer, so this source of complexity is easily eliminated.

Another is our use of logical disjunction $\vee$, hidden inside the fragment disjunction operator $+$, in the treatment of disjunction patterns. By construction, every such use of disjunction appears inside the left component of an implication. As a result, it is possible to lift it up and out of the implication, turning it into a conjunction: $\forall\bar{\beta}.(D_1 \vee D_2) \Rightarrow C$ is equivalent to $(\forall\bar{\beta}.D_1 \Rightarrow C) \wedge (\forall\bar{\beta}.D_2 \Rightarrow C)$. This operation, which duplicates the constraint C, corresponds to eliminating disjunction patterns in the source program, at the cost of some (possibly exponential) code duplication, as done by Xi (2003). In our constraint-based approach, the worst-case behavior is still exponential; however, an efficient constraint solver might simplify C before duplicating it, thus sharing much of the work. In an approach based on textual duplication of source expressions, every copy must be typechecked separately.

The last source of complexity, which is most problematic, is our frequent use of combined universal quantification and implication contexts, of the form $\forall\bar{\beta}.D \Rightarrow []$. If uncontrolled, these allow encoding negation, since $\neg D$ is $D \Rightarrow \mathsf{false}$. The bulk of this section is devoted to establishing that it is possible to use only *rigid implication*, of the form $\forall\bar{\beta}.D \Rightarrow []$, where $\mathrm{ftv}(D)$ *is a subset of* $\bar{\beta}$. A look at Figures 8 and 9 shows that none of the implication constructs that the current rules generate is, in general, rigid.

More precisely, we plan to reformulate the constraint generation rules so as to produce constraints that conform to the following restricted grammar:

$$\begin{array}{l} L ::= R \mid L \wedge L \mid L \vee L \\ R ::= \pi\,\bar{\tau} \mid R \wedge R \mid \exists\bar{\alpha}.R \mid \forall\bar{\beta}.L \Rightarrow R \qquad \text{where } \mathrm{ftv}(L) \subseteq \bar{\beta} \end{array}$$

We refer to constraints of the form R as *tractable*. They are made up of basic predicate application, conjunction, existential quantification, and *rigid* implication, of the form $\forall\bar{\beta}.L \Rightarrow R$, where every type variable that appears free in L is locally quantified, *i.e.* $\mathrm{ftv}(L) \subseteq \bar{\beta}$. The left-hand side of an implication, of the form L, may involve nested conjunctions and disjunctions of tractable constraints.

In order to be able to generate tractable constraints, we must require the type annotations supplied by the programmer to contain sufficient information. This requirement, as well as the modified constraint generation rules, are given in §6.1. Then, we argue that so-called tractable constraints are indeed easier to solve than arbitrary constraints, and define a constraint solving algorithm in the specific case of equality (§6.2).

6.1 Generating tractable constraints

As announced above, we restrict the constraint language available in data constructor declarations.

Requirement 6.1. In every declaration of the form $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$, the constraint D must in fact be of the form R. ◇

Because we wish to define a conservative extension of $\mathrm{HM}(X)$, and, more generally, of its extension with existential types in the style of Läufer and Odersky (1994), we cannot require programs that lie in this fragment of the language to carry type annotations. In other words, type annotations must be required only at functions that exploit polymorphic recursion or analyze a guarded algebraic data type. This leads us to begin with a precise definition of what it means for an algebraic data type to be guarded.

Definition 6.2. The data constructor $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ is *guarded* if and only if $\bar{\alpha} \cap \mathrm{ftv}(D) \neq \varnothing$. The algebraic data type constructor ε is *guarded* if and only if at least one of the data constructors associated with ε is guarded. A pattern p is *guarded* if and only if some data constructor that occurs in p is associated with a guarded algebraic data type. A clause $p.e$ is *guarded* if and only if p is guarded. A vector of clauses $\bar{c}$ is *guarded* if and only if one of its elements is guarded. ◇

Algebraic data types that are *not* guarded in the above sense are ordinary algebraic data types or encodings of existential types in the style of Läufer and Odersky (1994).

We are now able to express our requirement concerning type annotations:

Requirement 6.3. Every expression $\mu(x : \exists\bar{\beta}.\forall\bar{\gamma}[C].\tau_1 \to \tau_2).\lambda\bar{c}$ must satisfy:
(i) C is of the form R;
(ii) $\mathrm{ftv}(C) \subseteq \bar{\gamma}$;
(iii) if $\bar{c}$ is guarded, then $\mathrm{ftv}(\tau_1) \subseteq \bar{\gamma}$. ◇

Condition (i) restricts the constraint language that is available to the programmer, as announced earlier. Condition (ii) guarantees that the implication $\forall\bar{\gamma}.C \Rightarrow []$, which appears in the constraint generation rule for μ constructs (Figure 9), is rigid. The effect of condition (iii) is more technical, and is explained below when we describe the new constraint generation rules.

What are the practical implications of this requirement? Roughly speaking, it leads to distinguishing three cases. First, functions that do not involve polymorphic recursion and that do not perform case analysis on a guarded algebraic data type may carry no annotation—that is, they may carry the uninformative annotation $\exists\beta_1\beta_2.\beta_1 \to \beta_2$. Second, functions that do involve polymorphic recursion, but not guarded algebraic data types, must carry an annotation, subject to conditions (i) and (ii). Because these conditions are fulfilled when C is `true`, these functions may in particular carry any annotation of the form $\exists\bar{\beta}.\forall\bar{\gamma}.\tau_1 \to \tau_2$. Third, functions that do perform case analysis on a guarded algebraic data type must carry an annotation that satisfies all three conditions above. In such a case, only τ_2 may contain occurrences of the type variables $\bar{\beta}$. Experience seems to suggest that the extra flexibility afforded by this feature is minimal: not much expressiveness would be lost by requiring the type annotation to be *closed* (that is, to be of the form $\forall\bar{\gamma}[C].\tau_1 \to \tau_2$) in that case.

Remark 6.4. Requirement 6.3 may sound rather drastic. In fact, we spent considerable time exploring a more flexible approach, which turned out to be much more complex, for a marginal benefit. Some experience with an actual implementation seems necessary to tell whether the present restriction is acceptable in practice. Among the examples given in §2, only `rmap_f` violates the requirement. ◇

We are now ready to reformulate the constraint generation rules, focusing on the third case above, where the novelty lies, since the former two do not involve guarded algebraic data types. How do we, in that case, massage the existing rules so that they produce rigid implications only?

The main idea is to *fuse* adjacent implications. For instance, consider the context $\forall\bar{\gamma}.C \Rightarrow \forall\bar{\beta}.D \Rightarrow []$, where $\mathrm{ftv}(C) \subseteq \bar{\gamma}$ and $\mathrm{ftv}(D) \subseteq \bar{\gamma}\bar{\beta}$ hold. Then, while the first implication is rigid, the second one isn't. However, this context is equivalent to $\forall\bar{\gamma}\bar{\beta}.(C \wedge D) \Rightarrow []$, a rigid implication. More generally, we wish to fuse implications that are nested, but not adjacent, because some other connector lies in between. When this connector is a conjunction, we may push the external (rigid) implication into each component of the conjunction, by rewriting $\forall\bar{\gamma}.C \Rightarrow ([]_1 \wedge []_2)$ into $(\forall\bar{\gamma}.C \Rightarrow []_1) \wedge (\forall\bar{\gamma}.C \Rightarrow []_2)$. When it is an existential quantifier, we are, in general, stuck. However, if this existentially quantified constraint happens to be of the particular form $\exists\bar{\alpha}.(\varepsilon(\bar{\alpha}) = \tau \wedge [])$, where $\bar{\alpha}$ is fresh for τ, then, by Lemma 4.3, it may also be written $\exists\bar{\alpha}.(\varepsilon(\bar{\alpha}) = \tau) \wedge \forall\bar{\alpha}.(\varepsilon(\bar{\alpha}) = \tau) \Rightarrow []$. In this new formulation, no existential quantifier lies on the path from the root to the hole $[]$, so fusing can again take place.

We now propose a modified version of the constraint generation rules that incorporates these ideas, so as to directly produce tractable constraints. The modified rule set is given in Figure 10. (Again, the new type variables that appear in the right-hand side of an equation must be chosen fresh for the parameters that appear on its left-hand side.) In order to distinguish between the old and new rule sets, we now use square brackets $[\![\cdots]\!]$ instead of bananas $(\!|\cdots|\!)$.

Patterns

$$
\begin{aligned}
[\![0 \downarrow \forall\bar\gamma[C].\tau]\!] &= \mathsf{true} \\
[\![1 \downarrow \forall\bar\gamma[C].\tau]\!] &= \mathsf{true} \\
[\![x \downarrow \forall\bar\gamma[C].\tau]\!] &= \mathsf{true} \\
[\![p_1 \wedge p_2 \downarrow \forall\bar\gamma[C].\tau]\!] &= [\![p_1 \downarrow \forall\bar\gamma[C].\tau]\!] \wedge [\![p_2 \downarrow \forall\bar\gamma[C].\tau]\!] \\
[\![p_1 \vee p_2 \downarrow \forall\bar\gamma[C].\tau]\!] &= [\![p_1 \downarrow \forall\bar\gamma[C].\tau]\!] \wedge [\![p_2 \downarrow \forall\bar\gamma[C].\tau]\!] \\
[\![K\, p_1 \cdots p_n \downarrow \forall\bar\gamma[C].\tau]\!] &= (\forall\bar\gamma.C \Rightarrow \exists\bar\alpha.\varepsilon(\bar\alpha) = \tau) \wedge_i [\![p_i \downarrow \forall\bar\gamma\bar\alpha\bar\beta[C \wedge \varepsilon(\bar\alpha) = \tau \wedge D].\tau_i]\!] \\
&\quad \textit{where } K :: \forall\bar\alpha\bar\beta[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar\alpha)
\end{aligned}
$$

Clauses

$$
\begin{aligned}
[\![\Gamma \vdash p.e : \forall\bar\gamma[C].\tau_1 \to \tau_2]\!] &= [\![p \downarrow \forall\bar\gamma[C].\tau_1]\!] \wedge \forall\bar\gamma\bar\beta.(C \wedge D) \Rightarrow [\![\Gamma\Gamma' \vdash e : \tau_2]\!] \\
&\quad \textit{where } \exists\bar\beta[D]\Gamma' \textit{ is } (\!|p \uparrow \tau_1|\!)
\end{aligned}
$$

Expressions

$$
[\![\Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda\bar c : \tau]\!] = \begin{cases} \exists\bar\beta.([\![\Gamma[x \mapsto \sigma] \vdash \bar c : \sigma]\!] \wedge \sigma \leq \tau) & \text{if } \bar c \text{ is guarded} \\ \text{[omitted]} & \text{otherwise} \end{cases}
$$

Figure 10: Constraint generation, revisited

The new rules that associate constraints to patterns are parameterized not only by a type τ, as in Figure 8, but also by a set of so-called rigid type variables $\bar\gamma$ and a constraint C. The rules expect and preserve the invariant $\mathrm{ftv}(C, \tau) \subseteq \bar\gamma$. The idea is to perform constraint generation and to fuse the context $\forall\bar\gamma.C \Rightarrow []$ at once. This is made precise by the next lemma, which relates the old and new rules.

LEMMA 6.5. *$[\![p \downarrow \forall\bar\gamma[C].\tau]\!]$ is equivalent to $\forall\bar\gamma.C \Rightarrow (\!|p \downarrow \tau|\!)$.* ◇ Proof on page 61

LEMMA 6.6. *If $\mathrm{ftv}(C, \tau) \subseteq \bar\gamma$ holds, then $[\![p \downarrow \forall\bar\gamma[C].\tau]\!]$ is tractable.* ◇ Proof on page 61

The rules that associate environment fragments to patterns need not be modified: they already produce tractable constraints.

Similarly, the new rule that deals with clauses is parameterized not only by a type $\tau_1 \to \tau_2$, but by a type scheme $\forall\bar\gamma[C].\tau_1 \to \tau_2$, which comes directly from the type annotation carried by the enclosing function. The components $\bar\gamma$, C, and τ_1 are used to associate a constraint with the pattern at hand. Conditions (ii) and (iii) of Requirement 6.3 guarantee $\mathrm{ftv}(C, \tau_1) \subseteq \bar\gamma$, which is the invariant expected by the rules that deal with patterns. The (rigid) implication associated with the type annotation, of the form $\forall\bar\gamma.C \Rightarrow []$, is fused with the implication associated with pattern matching, of the form $\forall\bar\beta.D \Rightarrow []$. This yields an implication of the form $\forall\bar\gamma\bar\beta.(C \wedge D) \Rightarrow []$, which is rigid. Indeed, by construction, we have $\mathrm{ftv}(D) \subseteq \bar\beta \cup \mathrm{ftv}(\tau_1)$, which, together with conditions (ii) and (iii) of Requirement 6.3, implies $\mathrm{ftv}(C \wedge D) \subseteq \bar\gamma\bar\beta$.

The new rule for functions distinguishes two cases, depending on whether $\bar c$ is guarded. If it is (that is, if the case analysis involves a guarded algebraic data type), then the type scheme

found in the type annotation is used to associate a constraint with every clause, as described above. Otherwise, the function may be dealt with "as usual," so, for the sake of brevity, we omit the corresponding constraint generation rules (see Remark 6.7). The rules that concern all other expression forms remain identical to those given in Figure 8, except bananas $(\!|\cdots|\!)$ are replaced with square brackets $[\![\cdots]\!]$.

Remark 6.7. Technically, the rules that we omit are those that govern case analysis in an extension of HM(X) with polymorphic recursion, pattern matching with nested patterns, and (bounded) existential types in the style of Läufer and Odersky (1994). Although, to the best of our knowledge, these rules do not appear in the literature, they may be considered folklore, and are off topic in the present paper, where we wish to concentrate on *guarded* algebraic data types.

Unfortunately, the constraint generation rules given in §5 do *not* produce tractable constraints, even when only ordinary algebraic data types are involved. On the other hand, the rules given in Figure 10 do produce tractable constraints, but require $\mathrm{ftv}(\tau_1) \subseteq \bar{\gamma}$, so they cannot handle unannotated functions. This explains why, in order to produce tractable constraints, we end up with *two* independent rule sets, namely the rule set of Figure 10, which deals with functions that analyze a guarded algebraic data type, and the omitted rule set mentioned in the previous paragraph, which deals with ordinary functions.

This is a dark area in our presentation: despite substantial efforts, we have not been able to devise a (reasonably simple) rule set that covers both cases at once. ◇

Example 6.8. The constraint generated for $\mu print. \ldots$ in Example 5.9 is not tractable. Indeed, the left-hand side of each implication mentions the type variable α, which is certainly universally bound, but not locally: a conjunction lies on the path from the $\forall\alpha$ quantifier to the implication.

On the other hand, the new rule set does produce a tractable constraint:

$$\begin{array}{rrcl} [\![\Gamma_0 \vdash \mu print. \ldots : \gamma]\!] = & \forall\alpha.\alpha = \mathit{int} & \Rightarrow & (\!|\Gamma \vdash e_1 : \alpha \to \mathit{unit}|\!) \\ & \wedge\, \forall\alpha\beta_1\beta_2.\alpha = \beta_1 \times \beta_2 & \Rightarrow & (\!|\Gamma \vdash e_2 : \alpha \to \mathit{unit}|\!) \\ & \wedge\, \exists\alpha.\mathit{ty}(\alpha) \to \alpha \to \mathit{unit} \leq \gamma & & \end{array}$$

In comparison with the constraint produced in Example 5.9, the universal quantifier $\forall\alpha$ has been pushed into both components of the conjunction. Then, inside the second component, it has been fused with the quantifier $\forall\beta_1\beta_2$. ◇

The constraint generation rules of Figure 10 are equivalent to those of Figures 8 and 9, which ensures that they still define a correct and complete type inference algorithm. Furthermore, they produce tractable constraints.

THEOREM 6.9 (EQUIVALENCE). *$[\![\Gamma \vdash c : \forall\bar{\gamma}[C].\tau_1 \to \tau_2]\!]$ is equivalent to $\forall\bar{\gamma}.C \Rightarrow (\!|\Gamma \vdash c : \tau_1 \to \tau_2|\!)$. $[\![\Gamma \vdash e : \tau]\!]$ is equivalent to $(\!|\Gamma \vdash e : \tau|\!)$.* ◇ Proof on page 61

THEOREM 6.10. *$[\![\Gamma \vdash c : \forall\bar{\gamma}[C].\tau_1 \to \tau_2]\!]$ is tractable. $[\![\Gamma \vdash e : \tau]\!]$ is tractable.* ◇ Proof on page 61

Remark 6.11. For the sake of simplicity, our "$\mu\lambda$" construct combines recursion, abstraction, and case analysis. In other words, in a direct implementation of our framework, every function that analyzes a guarded algebraic data type must be of the restricted form:

```
let rec f : σ =
  fun x ->
    match x with
    ...
```

(We are assuming, for simplicity, that σ must be closed.) In practice, it is possible to relax this restriction slightly. For instance, one might allow f to accept multiple parameters and to analyze several of them at once, a common Objective Caml idiom:

```
let rec f : σ =
  fun x1 ... xn ->
    match (x_{i1}, ..., x_{ik}) with
    ...
```

It is straightforward to associate a tractable constraint with this composite construct. The key reason is that *the types of the parameters $x_1, \ldots, x_n$ are known*, since they are given by the type annotation σ. Thus, these types need not be inferred, which, in terms of constraints, means that no existential quantifier needs appear between the initial universal quantifier (which corresponds to the annotated `let rec` construct) and the implications (which correspond to the `match` construct). If such an existential quantifier did occur, it would prevent fusing the universal quantifier with the implications, so as to produce rigid implications, so we would be unable to produce a tractable constraint.

It is possible to further relax the syntax of functions that analyze a guarded algebraic data type. The rule that "no existential quantifier must arise between `let rec` and `match`" means that *no type inference can take place* between these constructs. Thus, by requiring these functions to exhibit simple structure, and by locally disallowing true type inference, we are really performing a kind of *local type inference* (Pierce and Turner, 2000). This was predicted by Remark 5.1. ◇

6.2 Solving tractable constraints in the case of equality

Why should so-called tractable constraints be easier to solve than arbitrary constraints? In particular, why should rigid implications be more tractable than arbitrary implications?

Let us begin with an informal explanation. One may say that the trouble with implication is that it involves a choice: satisfying an implication construct requires either refuting its left-hand side or satisfying its right-hand side. In the case of a rigid implication $\forall\bar{\beta}.L \Rightarrow R$, it is easy to resolve this choice. Indeed, because the constraint $\exists\bar{\beta}.L$ is closed, it must be equivalent to either false or true, and the solver is effectively able to tell which. If it is false, then the entire construct is equivalent to true and may be discarded. If it is true, the right-hand side R must be examined, under the simplifying assumption that L, which is satisfiable, has been rewritten to a solved form.

In the following, we give a more formal justification for the tractability of these constraints by describing a constraint solving algorithm, *in the specific case where no basic predicates other than equality are available*. It is difficult to argue about the general case, because the design of a constraint solver heavily depends on the basic predicates π at hand. Nevertheless, we do believe that the notion of rigid implication may be of interest in other cases: for instance, the first author has developed a similar notion, together with a constraint solving procedure, in the setting of structural subtyping (Simonet, 2003a).

From here on, we assume that the only basic predicate π is equality. Thus, we are looking at a particular instance of HMG(X), which we refer to as HMG(=). It is well-known that the corresponding instance of HM(X), namely HM(=), coincides with Hindley and Milner's type system (Pottier and Rémy, 2005). Thus, HMG(=) is an extension of Hindley and Milner's type system with guarded algebraic data types, as illustrated in §2.1.

$$
\begin{array}{rcll}
U & \rightarrowtriangle & U' & \textsc{s-unif} \\
 & & \textit{if } U \rightarrowtriangle_{\mathsf{u}} U' & \\
(L_1 \vee L_2) \wedge L_3 & \rightarrowtriangle & (L_1 \wedge L_3) \vee (L_2 \wedge L_3) & \textsc{s-and-or} \\
\forall \bar{\beta}.(L_1 \vee L_2) \Rightarrow R & \rightarrowtriangle & (\forall \bar{\beta}.L_1 \Rightarrow R) \wedge (\forall \bar{\beta}.L_2 \Rightarrow R) & \textsc{s-all-or} \\
\forall \bar{\beta}.\mathsf{false} \Rightarrow R & \rightarrowtriangle & \mathsf{true} & \textsc{s-all-false} \\
\forall \bar{\beta}_1 \bar{\beta}_2.R_1 \Rightarrow R_2 & \rightarrowtriangle & \forall \bar{\beta}_1.\exists \bar{\beta}_2.(R_1 \wedge R_2) & \textsc{s-all} \\
 & & \textit{if } \exists \bar{\beta}_2.R_1 \equiv \mathsf{true} \textit{ and } R_1 \textit{ determines } \bar{\beta}_2 &
\end{array}
$$

Figure 11: Solving constraints

Let us now explain how to solve tractable constraints in this particular case. We assume that a standard algorithm for first-order unification under a mixed prefix is given. That is, we assume that a constraint solving procedure is available for constraints defined by the following grammar:

$$U ::= \mathsf{false} \mid \tau = \tau \mid U \wedge U \mid \exists \bar{\alpha}.U \mid \forall \bar{\alpha}.U$$

The constraints U form a subset of the tractable constraints R, provided $\forall \bar{\alpha}.U$ is read as $\forall \bar{\alpha}.\mathsf{true} \Rightarrow U$. We assume that the unification algorithm consists of a reduction relation $\rightarrowtriangle_{\mathsf{u}}$ on constraints U, specified by the following definition.

Definition 6.12. Let $\rightarrowtriangle_{\mathsf{u}}$ be a relation on constraints U such that:
(i) $\rightarrowtriangle_{\mathsf{u}}$ is strongly normalizing;
(ii) $\rightarrowtriangle_{\mathsf{u}}$ is meaning preserving, that is, $U_1 \rightarrowtriangle_{\mathsf{u}} U_2$ implies $U_1 \equiv U_2$;
(iii) every normal form for $\rightarrowtriangle_{\mathsf{u}}$ is either satisfiable or false;
(iv) if U is a satisfiable normal form for $\rightarrowtriangle_{\mathsf{u}}$, then there exist $\bar{\beta}$ such that U determines $\bar{\beta}$ and $\exists \bar{\beta}.U$ is equivalent to true.

Conditions (i) to (iii) ensure that $\rightarrowtriangle_{\mathsf{u}}$ is indeed a constraint solving procedure. Condition (iv) characterizes the structure of solved forms: in short, it states that, for every assignment of the type variables other than $\bar{\beta}$, there exists a unique assignment of $\bar{\beta}$ that satisfies U. The type variables $\bar{\beta}$ are referred to as *eliminable* by Lassez, Maher, and Marriott (1988), while the type variables other than $\bar{\beta}$ are known as *parameters*. We do not give more details about first-order unification under a mixed prefix, because it is standard.

We now intend to show that the unification algorithm $\rightarrowtriangle_{\mathsf{u}}$ may be extended to solve tractable constraints of the form R. We present the extended algorithm as a reduction relation $\rightarrowtriangle$ on constraints, whose definition appears in Figure 11. Rewriting is performed modulo commutativity of conjunction and modulo an arbitrary context.

The definition of the extended algorithm may be explained as follows. Rule S-UNIF states that $\rightarrowtriangle$ contains $\rightarrowtriangle_{\mathsf{u}}$, allowing the underlying unification algorithm to be invoked at any time. Rules S-AND-OR and S-ALL-OR eliminate disjunctions by lifting them up through conjunctions and implications. As noted earlier, because S-ALL-OR duplicates the right-hand side R, an efficient strategy should rewrite R to a solved form first. Rules S-ALL-FALSE and S-ALL eliminate rigid implications. S-ALL-FALSE deals with implications whose left-hand side is unsatisfiable. According to the constraint generation rules given in §6.1, this situation indicates the presence of a dead

clause in the program, so one may wish to issue a warning whenever this rule is applied, unless the dead clause was automatically generated by the scheme described in §4.7. S-ALL deals with implications whose left-hand side is satisfiable, and has been reduced to a solved form. Its side conditions ensure that, when an assignment of $\bar{\beta}_1$ is fixed, there exists a unique assignment of $\bar{\beta}_2$ that satisfies R_1, so that $\forall\bar{\beta}_2.R_1 \Rightarrow []$ is in fact equivalent to $\exists\bar{\beta}_2.(R_1 \wedge [])$. Note that S-ALL's right-hand side no longer contains an implication.

Example 6.13. Consider the rigid implication $\forall\alpha\beta_1\beta_2.(\alpha = \beta_1 \times \beta_2) \Rightarrow R$. The left-hand side $\alpha = \beta_1 \times \beta_2$ is a solved form, where α is eliminable and β_1 and β_2 are parameters. In other words, $\exists\alpha.(\alpha = \beta_1 \times \beta_2)$ is equivalent to true and $\alpha = \beta_1 \times \beta_2$ determines α. Thus, S-ALL is applicable, and yields $\forall\beta_1\beta_2.\exists\alpha.(\alpha = \beta_1 \times \beta_2 \wedge R)$, eliminating the implication.

It is worth noting that the latter constraint is equivalent to $\forall\beta_1\beta_2.[\alpha \mapsto \beta_1 \times \beta_2]R$, where the substitution $[\alpha \mapsto \beta_1 \times \beta_2]$, a most general unifier of the constraint $\alpha = \beta_1 \times \beta_2$, is applied to R. Indeed, it would be possible to give a version of S-ALL where a most general unifier of R_1 is applied to R_2. Our presentation is equivalent, but purely constraint-based, which makes it more faithful to an efficient implementation. (Indeed, presentations that rely on substitutions do not allow reasoning about sharing. For instance, compare Robinson's substitution-based unification algorithm (1965), whose complexity is exponential, with Huet's graph-based algorithm (1976), whose complexity is quasi-linear, and whose most natural specification is constraint-based (Jouannaud and Kirchner, 1990; Pottier and Rémy, 2005).) ◇

The extended algorithm fulfills the exact same specification as the standard unification algorithm which we started with, as stated by the next theorem. This proves that $\twoheadrightarrow$ provides a constraint solving algorithm for so-called tractable constraints.

THEOREM 6.14. *The reduction relation $\twoheadrightarrow$ satisfies Definition 6.12, where every occurrence of the meta-variable U is replaced with R.* ◇

Proof on page 61

Let us stress that the structure of solved forms is the same for the extended algorithm and for the standard algorithm. In the context of type inference, this means that the structure of type schemes is the same in HMG(=) as in HM(=): that is, they are standard Hindley-Milner type schemes. In other words, extending ML with guarded algebraic data types does not affect the grammar of type schemes—an important point for programmers.

The complexity of constraint solving is exponential in the worst case, due to the presence of disjunctions. In their absence, we believe (but have not proved) that it is quasi-linear, as in the case of unification.

7 Conclusion

We have attempted to give a comprehensive account of the introduction of guarded algebraic data types into a typed programming language of the Hindley-Milner family. For the sake of generality, we have adopted a constraint-based approach, in the style of HM(X), so that type systems based on equality constraints, on subtyping constraints, or on arithmetic constraints, for example, are instances of our framework.

We have proved, in this very general setting, that the type system is sound (§4) and that, provided μ constructs carry a type annotation, type inference may be reduced to constraint solving

(§5). Then, because solving arbitrary constraints is expensive, we have further restricted the form of type annotations and proved that this allows producing so-called *tractable* constraints (§6.1). Last, in the specific setting of equality, we have explained how to solve tractable constraints (§6.2).

One of the strong points of our work is a clear separation of concerns. We have considered type soundness, decidable type inference, and tractable type inference as distinct goals, and gradually imposed the requirements necessary to reach each of them. In particular, our results show that, as far as *decidable* type inference is concerned, guarded algebraic data types do not introduce new difficulties beyond those already caused by polymorphic recursion. They do pose a new difficulty when pursuing *tractable* type inference: then, it appears necessary to request new type information, by placing more requirements on the form of type annotations.

We have chosen to deal with deep (nested) patterns, as opposed to only a shallow `case` construct, which solely examines the root of a term. This sometimes raises unexpectedly subtle issues (see Remark 4.28), and makes a few proofs significantly more involved (see *e.g.* Lemma 4.37).

A few weak points remain to be addressed. For instance, the requirement placed on type annotations in order to generate tractable constraints (Requirement 6.3) is strong: roughly speaking, functions that analyze a guarded algebraic data type must explicitly carry a closed type scheme. One might wonder whether this requirement is acceptable in practice, and to what extent it can be relaxed. Another inelegant aspect is the fact that our type inference algorithm uses two separate rule sets to deal with functions that analyze ordinary (Läufer-Odersky-style) algebraic data types, on the one hand, and with those that analyze guarded algebraic data types, on the other hand (see Remark 6.7).

We are currently working on a prototype implementation of HMG(=), that is, of an extension of ML with guarded algebraic data types. We hope to report on our experience in a later paper.

A Proofs

Proof of Lemma 3.1. Define the *weight* of a pattern as follows:

$$\begin{aligned} w(0) = w(1) = w(x) &= 3 \\ w(p_1 \wedge p_2) &= w(p_1) \times w(p_2) \\ w(p_1 \vee p_2) &= w(p_1) + 2 + w(p_2) \\ w(K\, p_1 \cdots p_n) &= 3(1 + w(p_1) \times \cdots \times w(p_n)) \end{aligned}$$

It is straightforward to check that every pattern has weight at least 3 and that each of the rules in Figure 4 is weight-decreasing. Furthermore, weight is preserved by associativity and commutativity of $\wedge$, by associativity of $\vee$, and is monotone with respect to arbitrary contexts. This proves that the length of any reduction sequence for $\rightsquigarrow$ is bounded by the weight of its initial term. We note that the weight of a pattern is at most exponential in its size. □

Proof of Lemma 3.2. By examination of each normalization rule (Figure 4) and by definition of extended substitution (Figure 2). □

Proof of Lemma 3.3. We begin with an analysis of the structure of the normal forms for $\rightsquigarrow$. It is straightforward to check that every normal form must be either 0 or a (multi-way) disjunction of one or more *definite patterns*, where a definite pattern is a (multi-way) conjunction of variables, 1's, and *at most one* data constructor pattern, whose subpatterns are again definite.

By structural induction, it is immediate that every definite pattern matches at least one value. Similarly, so does every disjunction of one or more definite patterns.

Now, assume $p \equiv 0$ holds. By Lemma 3.1, p has a normal form p'. By Lemma 3.2, $p' \equiv 0$ holds as well, so p' matches no value. By the previous paragraph, p' cannot be a disjunction of one or more definite patterns. So, by the first paragraph, p' must be 0. This proves that $p \rightsquigarrow^\star 0$ holds. (In fact, it proves that *every* normal form for p is 0, which is stronger and more useful, since $\rightsquigarrow$ is not confluent.)

Conversely, by Lemma 3.2, $p \rightsquigarrow^\star 0$ implies $p \equiv 0$. □

Proof of Lemma 4.3. It is clear that $\exists\bar{\alpha}.C_1 \wedge \forall\bar{\alpha}.C_1 \Rightarrow C_2$ entails $\exists\bar{\alpha}.(C_1 \wedge C_2)$. Conversely, assume C_1 determines $\bar{\alpha}$ (**1**). Let ρ satisfy $\exists\bar{\alpha}.(C_1 \wedge C_2)$. Then, there exists a ground assignment ρ'' that coincides with ρ outside $\bar{\alpha}$ and satisfies $C_1 \wedge C_2$. Now, pick an arbitrary ground assignment ρ' that coincides with ρ outside $\bar{\alpha}$ and satisfies C_1. Now, ρ' and ρ'' both satisfy C_1, and coincide outside $\bar{\alpha}$, so, by (1), ρ' and ρ'' must in fact coincide. Thus, ρ' satisfies C_2. Since this holds for an arbitrary ρ', we have in fact proved that ρ satisfies $\forall\bar{\alpha}.C_1 \Rightarrow C_2$. Since this holds for an arbitrary ρ, we have proved that $\exists\bar{\alpha}.(C_1 \wedge C_2)$ entails $\forall\bar{\alpha}.C_1 \Rightarrow C_2$. □

Proof of Lemma 4.5. Let ρ satisfy D. Then, by Definition 4.4, $\rho(\forall\bar{\alpha}[D].\tau)$ is a superset of $\uparrow\{\rho(\tau)\}$. Thus, ρ satisfies $\forall\bar{\alpha}[D].\tau \leq \tau$. □

Proof of Lemma 4.6. Left to the reader. One may consult the proof of Lemma 4.8, which follows, and is dual. □

Proof of Lemma 4.8. We have

$$\begin{array}{ll}
 & \rho \vdash \Delta' \leq \Delta \\
\iff & \downarrow\{\rho[\bar{\beta}' \mapsto \bar{t}'](\Gamma') \,/\, \rho[\bar{\beta}' \mapsto \bar{t}'] \vdash D'\} \subseteq \downarrow\{\rho[\bar{\beta} \mapsto \bar{t}](\Gamma) \,/\, \rho[\bar{\beta} \mapsto \bar{t}] \vdash D\} \\
 & \text{by definition of } \rho \vdash \Delta' \leq \Delta, \text{ of } \rho(\Delta'), \text{ and of } \rho(\Delta) \\
\iff & \{\rho[\bar{\beta}' \mapsto \bar{t}'](\Gamma') \,/\, \rho[\bar{\beta}' \mapsto \bar{t}'] \vdash D'\} \subseteq \downarrow\{\rho[\bar{\beta} \mapsto \bar{t}](\Gamma) \,/\, \rho[\bar{\beta} \mapsto \bar{t}] \vdash D\} \\
 & \text{by definition of } \downarrow \\
\iff & \forall\bar{t}' \quad \rho[\bar{\beta}' \mapsto \bar{t}'] \vdash D' \Rightarrow \exists\bar{t} \quad (\rho[\bar{\beta} \mapsto \bar{t}] \vdash D) \wedge (\rho[\bar{\beta}' \mapsto \bar{t}'](\Gamma') \leq \rho[\bar{\beta} \mapsto \bar{t}](\Gamma)) \\
\iff & \forall\bar{t}' \quad \rho[\bar{\beta}' \mapsto \bar{t}'] \vdash D' \Rightarrow \exists\bar{t} \quad \rho[\bar{\beta}' \mapsto \bar{t}'][\bar{\beta} \mapsto \bar{t}] \vdash (D \wedge \Gamma' \leq \Gamma) \\
 & \text{by exploiting } \bar{\beta} \# \mathrm{ftv}(\Gamma') \text{ and } \bar{\beta}' \# \mathrm{ftv}(\Delta) \\
\iff & \rho \vdash \forall\bar{\beta}'.D' \Rightarrow \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)
\end{array}$$

As a corollary, $C \Vdash \Delta' \leq \Delta$ is equivalent to $C \Vdash \forall\bar{\beta}'.D' \Rightarrow \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$. If $\bar{\beta}' \# \mathrm{ftv}(C)$ holds, then, by lifting the $\forall\bar{\beta}'$ quantifier up through the entailment symbol $\Vdash$, this may be written $C \Vdash D' \Rightarrow \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$, that is, $C \wedge D' \Vdash \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$. □

Proof of Lemma 4.15. Assume Δ is $\exists\bar{\beta}[D]\Gamma$ (**1**), where $\bar{\beta} \# \mathrm{ftv}(\bar{\alpha}, C)$ (**2**). We have

$$\begin{array}{ll}
 & \rho(\exists\bar{\alpha}[C]\Delta) \\
= & \rho(\exists\bar{\alpha}\bar{\beta}[C \wedge D]\Gamma) \\
 & \text{by (1), (2), and Definition 4.14} \\
= & \downarrow\{\rho[\bar{\alpha} \mapsto \bar{t}, \bar{\beta} \mapsto \bar{t}'](\Gamma) \,/\, \rho[\bar{\alpha} \mapsto \bar{t}, \bar{\beta} \mapsto \bar{t}'] \vdash C \wedge D\} \\
 & \text{by Definition 4.7} \\
= & \downarrow\{\rho[\bar{\alpha} \mapsto \bar{t}][\bar{\beta} \mapsto \bar{t}'](\Gamma) \,/\, \rho[\bar{\alpha} \mapsto \bar{t}][\bar{\beta} \mapsto \bar{t}'] \vdash D \wedge \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\} \\
 & \text{by (2)} \\
= & \cup\{\rho[\bar{\alpha} \mapsto \bar{t}](\Delta) \,/\, \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\} \\
 & \text{by (1) and Definition 4.7}
\end{array}$$

□

PROOF OF LEMMA 4.18. Assume Δ_1 and Δ_2 have disjoint domains, so that $\Delta_1 \times \Delta_2$ is defined. Assume Δ_1 is $\exists\bar{\beta}_1[D_1]\Gamma_1$ (**1**) and Δ_2 is $\exists\bar{\beta}_2[D_2]\Gamma_2$, (**2**) where $\bar{\beta}_1 \# \bar{\beta}_2$ (**3**), $\bar{\beta}_1 \# \text{ftv}(D_2, \Gamma_2)$ (**4**), and $\bar{\beta}_2 \# \text{ftv}(D_1, \Gamma_1)$ (**5**) hold. We have

$$
\begin{array}{ll}
 & \rho(\Delta_1 \times \Delta_2) \\
= & \rho(\exists\bar{\beta}_1\bar{\beta}_2[D_1 \wedge D_2](\Gamma_1 \times \Gamma_2)) \\
 & \text{by (1)-(5) and Definition 4.16} \\
= & \downarrow\{\rho[\bar{\beta}_1 \mapsto \bar{t}_1, \bar{\beta}_2 \mapsto \bar{t}_2](\Gamma_1 \times \Gamma_2) \;/\; \rho[\bar{\beta}_1 \mapsto \bar{t}_1, \bar{\beta}_2 \mapsto \bar{t}_2] \vdash D_1 \wedge D_2\} \\
 & \text{by Definition 4.7} \\
= & \downarrow\{\rho[\bar{\beta}_1 \mapsto \bar{t}_1](\Gamma_1) \times \rho[\bar{\beta}_2 \mapsto \bar{t}_2](\Gamma_2) \;/\; \rho[\bar{\beta}_1 \mapsto \bar{t}_1] \vdash D_1 \wedge \rho[\bar{\beta}_2 \mapsto \bar{t}_2] \vdash D_2\} \\
 & \text{by (4) and (5)} \\
= & \downarrow\{\rho[\bar{\beta}_1 \mapsto \bar{t}_1](\Gamma_1) \;/\; \rho[\bar{\beta}_1 \mapsto \bar{t}_1] \vdash D_1\} \times \downarrow\{\rho[\bar{\beta}_2 \mapsto \bar{t}_2](\Gamma_2) \;/\; \rho[\bar{\beta}_2 \mapsto \bar{t}_2] \vdash D_2\} \\
 & \text{by definition of } \times \text{ and by distributivity of } \downarrow \text{ with respect to } \times \\
= & \rho(\Delta_1) \times \rho(\Delta_2) \\
 & \text{by (1), (2), and Definition 4.7}
\end{array}
$$

□

PROOF OF LEMMA 4.19. Assume Δ_1 and Δ_2 have a common domain, so that $\Delta_1 + \Delta_2$ is defined. Assume Δ_1 is $\exists\bar{\beta}_1[D_1]\Gamma_1$ (**1**) and Δ_2 is $\exists\bar{\beta}_2[D_2]\Gamma_2$ (**2**), where $\bar{\beta}_1 \# \bar{\beta}_2$ (**3**), $\bar{\beta}_1 \# \text{ftv}(D_2, \Gamma_2)$ (**4**), and $\bar{\beta}_2 \# \text{ftv}(D_1, \Gamma_1)$ (**5**) hold. Let Γ map every variable in the domain of Δ_1 and Δ_2 to a distinct type variable in $\bar{\alpha}$ (**6**), where $\bar{\alpha} \# \text{ftv}(\bar{\beta}_1, \bar{\beta}_2, D_1, D_2, \Gamma_1, \Gamma_2)$ (**7**) holds. We have

$$
\begin{array}{ll}
 & \rho(\Delta_1 + \Delta_2) \\
= & \rho(\exists\bar{\beta}_1\bar{\beta}_2\bar{\alpha}[(D_1 \wedge \Gamma \leq \Gamma_1) \vee (D_2 \wedge \Gamma \leq \Gamma_2)]\Gamma) \\
 & \text{by (1)-(7) and Definition 4.17} \\
= & \downarrow\{\rho'(\Gamma) \;/\; \rho' \vdash \vee_i(D_i \wedge \Gamma \leq \Gamma_i)\} \\
 & \text{where } \rho' \text{ stands for } \rho[\bar{\beta}_1 \mapsto \bar{t}_1, \bar{\beta}_2 \mapsto \bar{t}_2, \bar{\alpha} \mapsto \bar{t}] \\
 & \text{by Definition 4.7} \\
= & \cup_i \downarrow\{\rho'(\Gamma) \;/\; \rho' \vdash D_i \wedge \Gamma \leq \Gamma_i\} \\
 & \text{by the interpretation of disjunction} \\
 & \text{by distributivity of } \downarrow \text{ with respect to } \cup \\
= & \cup_i \downarrow\{\bar{t} \;/\; \rho[\bar{\beta}_i \mapsto \bar{t}_i] \vdash D_i \wedge \bar{t} \leq \rho[\bar{\beta}_i \mapsto \bar{t}_i](\Gamma_i)\} \\
 & \text{by (4), (5), (6), and (7)} \\
= & \cup_i \downarrow\downarrow\{\rho[\bar{\beta}_i \mapsto \bar{t}_i](\Gamma_i) \;/\; \rho[\bar{\beta}_i \mapsto \bar{t}_i] \vdash D_i\} \\
 & \text{by definition of } \downarrow \\
= & \cup_i \rho(\Delta_i) \\
 & \text{by idempotency of } \downarrow, \text{ then by (1), (2), and Definition 4.7}
\end{array}
$$

□

PROOF OF LEMMA 4.20. By Lemmas 4.15, 4.18, and 4.19. □

PROOF OF LEMMA 4.30. By structural induction. □

PROOF OF LEMMA 4.31. By structural induction. □

PROOF OF LEMMA 4.32. Consider a derivation that ends with INST(GEN(·)). Its conclusion is $C, \Gamma \vdash e : \tau$ (**1**). The premises of INST are $C, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau$ (**2**) and $C \Vdash D$ (**3**). The derivation of (2) ends with an instance of GEN whose premises are $C' \wedge \theta D, \Gamma \vdash e : \theta\tau$ (**4**) and $\theta\bar{\alpha} \# \text{ftv}(\Gamma, C')$

(**5**), where $C \equiv C' \wedge \exists\bar{\alpha}.D$ (**6**) holds and the renaming θ is fresh for $\forall\bar{\alpha}[D].\tau$ (**7**). (By introducing θ, we account for the fact that Gen's conclusion might mention an arbitrary α-variant of the type scheme $\forall\bar{\alpha}[D].\tau$, namely $\forall(\theta\bar{\alpha})[\theta D].\theta\tau$.) Up to a renaming of Gen's premises, we may require $\theta\bar{\alpha} \# \mathrm{ftv}(D,\tau)$ (**8**). By Lemma 4.30, (4) yields $C' \wedge \theta D \wedge \theta\bar{\alpha} = \bar{\alpha}, \Gamma \vdash e : \theta\tau$ (**9**). By (7), we have $(\theta D \wedge \theta\bar{\alpha} = \bar{\alpha}) \equiv (D \wedge \theta\bar{\alpha} = \bar{\alpha})$ (**10**) and $\theta\bar{\alpha} = \bar{\alpha} \Vdash \theta\tau = \tau$ (**11**). By (9), (10), (11), and by Sub, we obtain $C' \wedge D \wedge \theta\bar{\alpha} = \bar{\alpha}, \Gamma \vdash e : \tau$ (**12**). By (5), (8), and Hide, (12) leads to $C' \wedge D \wedge \exists(\theta\bar{\alpha}).(\theta\bar{\alpha} = \bar{\alpha}), \Gamma \vdash e : \tau$ (**13**). Because θ is a renaming, the conjunct $\exists(\theta\bar{\alpha}).(\theta\bar{\alpha} = \bar{\alpha})$ is a tautology. Furthermore, according to (3) and (6), $C \equiv C' \wedge D$ holds. Thus, (13) is the goal (1).□

Proof of Lemma 4.33. Consider a derivation that ends with Hide(Gen(·)). Its conclusion is $\exists\bar{\alpha}_1.C_1, \Gamma \vdash e : \sigma$ (**1**). The premises of Hide are $C_1, \Gamma \vdash e : \sigma$ (**2**) and $\bar{\alpha}_1 \# \mathrm{ftv}(\Gamma,\sigma)$ (**3**). The derivation of (2) ends with an instance of Gen whose premises are $C_1' \wedge D, \Gamma \vdash e : \tau$ (**4**) and $\bar{\alpha} \# \mathrm{ftv}(\Gamma, C_1')$ (**5**) with $C_1 \equiv C_1' \wedge \exists\bar{\alpha}.D$ and $\sigma = \forall\bar{\alpha}[D].\tau$. Up to a renaming of Gen's premises, we may require $\bar{\alpha}_1 \# \bar{\alpha}$ (**6**). By (3) and (6), we have $\bar{\alpha}_1 \# \mathrm{ftv}(\Gamma,\tau)$ (**7**). By Hide, (4) and (7) imply $\exists\bar{\alpha}_1.(C_1' \wedge D), \Gamma \vdash e : \tau$ (**8**). By (3) and (6) again, we have $\bar{\alpha}_1 \# \mathrm{ftv}(D)$, so (8) may be written $\exists\bar{\alpha}_1.C_1' \wedge D, \Gamma \vdash e : \tau$. By (5) and Gen, this yields $\exists\bar{\alpha}_1.C_1' \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \sigma$ (**9**). The constraint $\exists\bar{\alpha}_1.C_1' \wedge \exists\bar{\alpha}.D$ may be written $\exists\bar{\alpha}_1.(C_1' \wedge \exists\bar{\alpha}.D)$, that is, $\exists\bar{\alpha}_1.C_1$. Thus, (9) is the goal (1). □

Proof of Lemma 4.34. Consider a derivation of shape App(Sub(Abs($\cdot_1$)), $\cdot_2$). Its conclusion is $C, \Gamma \vdash (\lambda\bar{c})\, e : \tau$ (**1**). The premises of App are $C, \Gamma \vdash \lambda\bar{c} : \tau' \to \tau$ (**2**) and $C, \Gamma \vdash e : \tau'$ (**3**). The derivation of (2) ends with an instance of Sub whose premises are $C, \Gamma \vdash \lambda\bar{c} : \tau_1' \to \tau_1$ (**4**) and $C \Vdash \tau_1' \to \tau_1 \leq \tau' \to \tau$ (**5**). (Because the judgment (4) is a consequence of Abs, it must exhibit an arrow type $\tau_1' \to \tau_1$.) By Requirement 4.10, (5) implies $C \Vdash \tau' \leq \tau_1'$ (**6**) and $C \Vdash \tau_1 \leq \tau$ (**7**). By (3), (6), and Sub, we have $C, \Gamma \vdash e : \tau_1'$ (**8**). By (4), (8), and App, we obtain $C, \Gamma \vdash (\lambda\bar{c})\, e : \tau_1$ (**9**). By (9), (7), and Sub, we reach the goal (1). □

Proof of Lemma 4.35. Let us consider an arbitrary typing derivation. Of course, every trivial instance of Gen or Inst may be suppressed, yielding a derivation that satisfies (a). Now, a nontrivial instance of Gen must be followed by either a syntax-directed rule or one of Inst, Hide. Furthermore, by Lemmas 4.32 and 4.33, Gen may be suppressed when it appears above Inst and pushed down when it appears above Hide. As a result, the derivation may be rewritten so as to satisfy (a) and (b). Next, by inspection of the rules in Figure 6, it is straightforward to check that, up to renamings of subderivations, Hide may be pushed down through every rule other than Gen while preserving (a) and (b). Furthermore, any number of consecutive instances of Hide may be collapsed into a single one. As a result, the derivation may be rewritten so as to satisfy (a), (b), and (c). At this point, one may check that, at every subexpression of the form $(\lambda\bar{c})\, e$, Abs and App may be separated only by zero or more instances of Sub. If there is one or more, then, by transitivity of subtyping, they may be collapsed to a single instance of Sub, which, by Lemma 4.34, may be eliminated without compromising (a), (b), or (c). Thus, the final derivation satisfies all four criteria. □

Proof of Lemma 4.36. By structural induction. All cases are straightforward. We give one of them, for the sake of illustration.

∘ *Case* Clause. ce is $p.e'$ and σ is $\tau' \to \tau$. We may assume, *w.l.o.g.*, that $x \notin \mathrm{dpv}(p)$ (**1**), so that $[x \mapsto e]ce$ is $p.[x \mapsto e]e'$. Clause's premises are $C \vdash p : \tau' \rightsquigarrow \exists\bar{\beta}[D]\Gamma'$ (**2**) and

$C \wedge D, \Gamma[x \mapsto \sigma']\Gamma' \vdash e' : \tau$ (**3**) and $\bar{\beta} \# \mathrm{ftv}(C, \Gamma, \sigma, \tau)$ (**4**). By (1), $\Gamma[x \mapsto \sigma']\Gamma'$ is $\Gamma\Gamma'[x \mapsto \sigma']$, so (3) may be written $C \wedge D, \Gamma\Gamma'[x \mapsto \sigma'] \vdash e : \tau$ (**5**). By Lemma 4.30, the hypothesis $C, \varnothing \vdash e : \sigma'$ yields $C \wedge D, \varnothing \vdash e : \sigma'$ (**6**). Applying the induction hypothesis to (5) and (6), we obtain $C \wedge D, \Gamma\Gamma' \vdash [x \mapsto e]e' : \tau$ (**7**). The goal follows by Clause from (2), (7), and (4). □

Proof of Lemma 4.37. To begin, let us prove that, if the above statement holds for a particular choice of $\bar{\beta}$, D, and Γ, then it holds for every such choice, that is, for every $\bar{\beta}'$, D', and Γ' such that $\exists\bar{\beta}[D]\Gamma$ and $\exists\bar{\beta}'[D']\Gamma'$ are α-equivalent and $\bar{\beta}' \# \mathrm{ftv}(C)$ holds. Indeed, let θ be the renaming that swaps $\bar{\beta}$ with $\bar{\beta}'$. θ maps D and Γ to D' and Γ', respectively. Thus, the property $H \Vdash D$ implies $\theta H \Vdash D'$. Furthermore, thanks to our freshness hypotheses, θ is fresh for C. Thus, we have $C \equiv \theta C \equiv \exists(\theta\bar{\beta}).\theta H \equiv \exists\bar{\beta}'.\theta H$, where the central step is permitted by applying θ to both sides of the property $C \equiv \exists\bar{\beta}.H$. Similarly, by applying θ to the property $H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$, we obtain $\theta H, \varnothing \vdash [p \mapsto v]x : \Gamma'(x)$. Thus, we have proved that the statement holds for $\bar{\beta}'$, D', and Γ', with θH as a witness. This initial remark is used several times later in the proof.

Next, we note that, since the proof of the present lemma is constructive, the witness H must satisfy, by construction, $\mathrm{ftv}(H) \subseteq \mathrm{ftv}(C, \tau, \Delta, \bar{\beta})$. This fact is used below to control the free type variables of the witnesses produced by invocations of the induction hypothesis.

By Lemma 4.35, we may assume, *w.l.o.g.*, that the derivation of $C, \varnothing \vdash v : \tau$ is normal. The proof proceeds with a structural induction on this derivation, where Hide forms the inductive case and all other cases are base cases.

∘ *Case* Hide. Our hypotheses are $\exists\bar{\alpha}.C, \varnothing \vdash v : \tau$ (**1**) and $\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). The judgment (1) follows from an instance of Hide whose premises are $C, \varnothing \vdash v : \tau$ (**3**) and $\bar{\alpha} \# \mathrm{ftv}(\tau)$ (**4**). We may assume, *w.l.o.g.*, $\bar{\alpha} \# \mathrm{ftv}(D, \Gamma)$ (**5**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 4.30 to (2) yields $C \vdash p : \tau \rightsquigarrow \Delta$ (**6**). Then, applying the induction hypothesis to (3) and (6) yields a constraint H such that $H \Vdash D$ (**7**) and $C \equiv \exists\bar{\beta}.H$ (**8**) and, for every $x \in \mathrm{dpv}(p)$, $H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$ (**9**) holds. By placing (7) within the context $\exists\bar{\alpha}.[]$ and exploiting (5), we obtain $\exists\bar{\alpha}.H \Vdash D$. Furthermore, (8) implies $\exists\bar{\alpha}.C \equiv \exists\bar{\beta}.\exists\bar{\alpha}.H$. Last, by applying Hide to (9) and (5), we obtain $\exists\bar{\alpha}.H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$. Thus, $\exists\bar{\alpha}.H$ is the desired witness.

We now reach the base case, where the derivation of $C, \varnothing \vdash v : \tau$ is normal and does not end with Hide. We proceed by induction on the derivation of $C \vdash p : \tau \rightsquigarrow \Delta$.

∘ *Case* p-Empty. Because the first hypothesis states that v matches p, this case cannot arise.

∘ *Case* p-Wild. Our hypotheses are $C, \varnothing \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}]\varnothing$. Let our witness be C; then, it is immediate to check that the goal holds.

∘ *Case* p-Var. Our hypotheses are $C, \varnothing \vdash v : \tau$ and $C \vdash x : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}](x \mapsto \tau)$. Let our witness be C; then, it is immediate to check that the goal holds. In particular, the third goal is the hypothesis $C, \varnothing \vdash v : \tau$.

∘ *Case* p-And. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p_1 \wedge p_2 : \tau \rightsquigarrow \Delta$ (**2**). The derivation (2) ends with an instance of p-And whose premises are, for all $i \in \{1, 2\}$, $C \vdash p_i : \tau \rightsquigarrow \Delta_i$ (**3**) where Δ is $\Delta_1 \times \Delta_2$ (**4**). Let us write Δ_i as $\exists\bar{\beta}_i[D_i]\Gamma_i$ (**5**), where $\bar{\beta}_i \# \mathrm{ftv}(C, \tau, \Delta_1, \Delta_2)$ (**6**) and $\bar{\beta}_1 \# \bar{\beta}_2$ (**7**); according to (4), and by (6) and (7), Δ is $\exists\bar{\beta}_1\bar{\beta}_2[D_1 \wedge D_2](\Gamma_1 \times \Gamma_2)$ (**8**). According to our initial remark, it is sufficient to establish the statement for this particular representation of Δ. By (5), applying the induction hypothesis to (1), (3) and (6), we obtain, for every $i \in \{1, 2\}$, a constraint H_i such that $H_i \Vdash D_i$ (**9**), $C \equiv \exists\bar{\beta}_i.H_i$ (**10**) and, for every $x \in \mathrm{dpv}(p_i)$, $H_i, \varnothing \vdash [p_i \mapsto v]x : \Gamma_i(x)$ (**11**). We also have, by construction, $\mathrm{ftv}(H_i) \subseteq \mathrm{ftv}(C, \tau, \Delta_i, \bar{\beta}_i)$, which by (6) and (7) implies $\bar{\beta}_j \# \mathrm{ftv}(H_i)$ when $\{i, j\} = \{1, 2\}$ (**12**).

Let us define H as $H_1 \wedge H_2$ and check that all three goals are met. The first goal is $H \Vdash D_1 \wedge D_2$, which follows from (9). The second goal is $C \equiv \exists \bar{\beta}_1 \bar{\beta}_2.(H_1 \wedge H_2)$, which follows from (10) and (12). Last, consider $x \in \mathrm{dpv}(p_1 \wedge p_2)$. There exists a unique i in $\{1, 2\}$ such that $x \in \mathrm{dpv}(p_i)$. Then, $[v \mapsto x]p$ is $[v \mapsto x]p_i$ and $(\Gamma_1 \times \Gamma_2)(x)$ is $\Gamma_i(x)$. Applying Lemma 4.30 to (11), we obtain $H_1 \wedge H_2, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$, which is the third goal.

∘ *Case* P-OR. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p_1 \vee p_2 : \tau \rightsquigarrow \Delta$ (**2**). Because v matches $p_1 \vee p_2$, there exists $i \in \{1, 2\}$ such that v matches p_i and $[p_1 \vee p_2 \mapsto v]$ is $[p_i \mapsto v]$. The derivation (2) ends with an instance of P-OR among whose premises we have $C \vdash p_i : \tau \rightsquigarrow \Delta$ (**3**). Then, applying the induction hypothesis to (1) and (3) yields the result.

∘ *Case* P-CSTR. Because a value that matches $K\, p_1 \cdots p_n$ must be of the form $K\, v_1 \cdots v_n$, our hypotheses are $C, \varnothing \vdash K\, v_1 \cdots v_n : \varepsilon(\bar{\alpha})$ (**1**) and $C \vdash K\, p_1 \cdots p_n : \varepsilon(\bar{\alpha}) \rightsquigarrow \Delta$ (**2**).

Because the derivation of (1) is normal and does not end with HIDE, it must end with the shape SUB(CSTR(·)). (Indeed, any number of consecutive occurrences of SUB may be expanded or collapsed to a single one.) Then, a straightforward analysis shows that SUB's second premise must be $C \Vdash \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha})$ (**3**), while CSTR's premises are $C, \varnothing \vdash v_i : \tau_i'$ (**4**), for every $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha}' \bar{\beta}' [D'].\tau_1' \cdots \tau_n' \to \varepsilon(\bar{\alpha}')$ (**5**), and $C \Vdash D'$ (**6**).

The derivation of (2) ends with an instance of P-CSTR whose premises are $C \wedge D \vdash p_i : \tau_i \rightsquigarrow \Delta_i$ (**7**), for every $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha} \bar{\beta} [D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ (**8**), and $\bar{\beta} \# \mathrm{ftv}(C)$ (**9**), where Δ is $\exists \bar{\beta}[D](\Delta_1 \times \cdots \times \Delta_n)$. Up to a renaming of P-CSTR's premises, we may assume $\bar{\beta} \# \bar{\alpha}' \bar{\beta}'$ (**10**). Let us write Δ_i as $\exists \bar{\beta}_i [D_i] \Gamma_i$, where $\bar{\beta}_i \# \mathrm{ftv}(C, \bar{\alpha}, \bar{\beta}, \bar{\alpha}', \bar{\beta}', \Delta_j, \bar{\beta}_j)$ (**11**) holds when $i \neq j$. Then, Δ is $\exists \bar{\beta} \bar{\beta}_1 \cdots \bar{\beta}_n [D \wedge_i D_i](\cup_i \Gamma_i)$ (**12**). According to our initial remark, it is sufficient to establish the statement for this particular representation of Δ.

By Lemma 4.30 and SUB, (4) yields $C \wedge D \wedge \tau_i' \leq \tau_i, \varnothing \vdash v_i : \tau_i$ (**13**). By Lemma 4.30 again, (7) yields $C \wedge D \wedge \tau_i' \leq \tau_i \vdash p_i : \tau_i \rightsquigarrow \Delta_i$ (**14**). Applying the induction hypothesis to (13) and (14), we obtain a constraint H_i such that $H_i \Vdash D_i$ (**15**) and $C \wedge D \wedge \tau_i' \leq \tau_i \equiv \exists \bar{\beta}_i.H_i$ (**16**) and, for every $x \in \mathrm{dpv}(p_i)$, $H_i, \varnothing \vdash [p_i \mapsto v_i]x : \Gamma_i(x)$ (**17**). We also have, by construction, $\mathrm{ftv}(H_i) \subseteq \mathrm{ftv}(C, D, \tau_i', \tau_i, \Delta_i, \bar{\beta}_i) \subseteq \mathrm{ftv}(C, \bar{\alpha}, \bar{\beta}, \bar{\alpha}', \bar{\beta}', \Delta_i, \bar{\beta}_i)$, which by (11) implies $\bar{\beta}_j \# \mathrm{ftv}(H_i)$ when $i \neq j$ (**18**). By Requirement 4.11, (5), (8), and (10) imply $D' \wedge \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}.(D \wedge_i \tau_i' \leq \tau_i)$ (**19**). Together with (6) and (3), this implies $C \Vdash \exists \bar{\beta}.(D \wedge_i \tau_i' \leq \tau_i)$ (**20**). Let us now define H as $D \wedge_i H_i$ and check that all three goals are met.

According to (12), the first goal is $H \Vdash D \wedge_i D_i$. It follows immediately from the definition of H and from (15).

According to (12), the second goal is $C \equiv \exists \bar{\beta} \bar{\beta}_1 \cdots \bar{\beta}_n.H$. Indeed, we have

$$\begin{array}{rcll} C & \equiv & \exists \bar{\beta}.(C \wedge D \wedge_i \tau_i' \leq \tau_i) & (\mathbf{21}) \\ & \equiv & \exists \bar{\beta}.(D \wedge_i \exists \bar{\beta}_i.H_i) & (\mathbf{22}) \\ & \equiv & \exists \bar{\beta} \bar{\beta}_1 \cdots \bar{\beta}_n.H & (\mathbf{23}) \end{array}$$

where (21) follows from (20) and (9); (22) follows from (16); (23) follows from (11), (18), and from the definition of H.

Last, consider x in $\mathrm{dpv}(p)$. There exists a unique i such that $x \in \mathrm{dpv}(p_i)$. Then, $[p \mapsto v]x$ is $[p_i \mapsto v_i]x$ and $(\cup_i \Gamma_i)(x)$ is $\Gamma_i(x)$. Applying Lemma 4.30 to (17), we obtain $H, \varnothing \vdash [p \mapsto v]x : (\cup_i \Gamma_i)(x)$, which is the third goal.

∘ *Case* P-EQIN. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). P-EQIN's premises are $C \vdash p : \tau' \rightsquigarrow \Delta$ (**3**) and $C \Vdash \tau = \tau'$ (**4**). Applying SUB to (1) and (4) yields a derivation of

$C, \varnothing \vdash v : \tau'$ (**5**), which still is normal and does not end with HIDE. There remains to apply the induction hypothesis to (5) and (3).

∘ *Case* P-SUBOUT. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). P-SUBOUT's premises are $C \vdash p : \tau \rightsquigarrow \Delta'$ (**3**) and $C \Vdash \Delta' \leq \Delta$ (**4**). By hypothesis, Δ is written $\exists\bar{\beta}[D]\Gamma$, where $\bar{\beta} \# \mathrm{ftv}(C)$ (**5**). Thanks to our initial remark, we may further require, *w.l.o.g.*, $\bar{\beta} \# \mathrm{ftv}(\tau, \Delta')$ (**6**). Let us write Δ' as $\exists\bar{\beta}'[D']\Gamma'$, where $\bar{\beta}' \# \mathrm{ftv}(C, \Delta, \bar{\beta})$ (**7**). Note that (6) and (7) imply $\bar{\beta} \# \mathrm{ftv}(\Gamma')$ (**8**), $\bar{\beta}' \# \mathrm{ftv}(D)$ (**9**), and $\bar{\beta}' \# \mathrm{ftv}(\Gamma)$ (**10**).

The induction hypothesis, applied to (1) and (3), yields a constraint H' such that $H' \Vdash D'$ (**11**) and $C \equiv \exists\bar{\beta}'.H'$ (**12**) and, for every $x \in \mathrm{dpv}(p)$, $H', \varnothing \vdash [p \mapsto v]x : \Gamma'(x)$ (**13**) holds. We also have, by construction, $\mathrm{ftv}(H') \subseteq \mathrm{ftv}(C, \tau, \Delta', \bar{\beta}')$, which by (5), (6), and (7) implies $\bar{\beta} \# \mathrm{ftv}(H')$ (**14**). Note also that (11) and (12) imply $H' \Vdash C \wedge D'$ (**15**).

Let us now define H as $D \wedge \exists\bar{\beta}'.(H' \wedge \Gamma' \leq \Gamma)$ and check that all three goals are met. The first goal, namely $H \Vdash D$, is immediate. Second, by Lemma 4.8, (4), (7), and (8) imply $C \wedge D' \Vdash \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$ (**16**). So, we have

$$\begin{array}{rcll} C & \equiv & \exists\bar{\beta}'.H' & (\mathbf{17}) \\ & \equiv & \exists\bar{\beta}'.(H' \wedge \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)) & (\mathbf{18}) \\ & \equiv & \exists\bar{\beta}.(D \wedge \exists\bar{\beta}'.(H' \wedge \Gamma' \leq \Gamma)) & (\mathbf{19}) \\ & \equiv & \exists\bar{\beta}.H & (\mathbf{20}) \end{array}$$

where (17) is exactly (12); (18) follows from (15) and (16); (19) is by (14) and (9); (20) is by definition of H. Thus, the second goal is met. Last, by Lemma 4.30 and SUB, (13) implies $H' \wedge \Gamma' \leq \Gamma, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$. By (10) and HIDE, this implies $\exists\bar{\beta}'.(H' \wedge \Gamma' \leq \Gamma), \varnothing \vdash [p \mapsto v]x : \Gamma(x)$. The third goal follows by Lemma 4.30 and by definition of H.

∘ *Case* P-HIDE. Our hypotheses are $\exists\bar{\alpha}.C, \varnothing \vdash v : \tau$ (**1**) and $\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). By hypothesis, Δ is written $\exists\bar{\beta}[D]\Gamma$, where $\bar{\beta} \# \mathrm{ftv}(\exists\bar{\alpha}.C)$ (**3**). The judgment (2) follows from an instance of P-HIDE whose premises are $C \vdash p : \tau \rightsquigarrow \Delta$ (**4**) and $\bar{\alpha} \# \mathrm{ftv}(\tau, \Delta)$ (**5**). We may assume, *w.l.o.g.*, that $\bar{\alpha}$ is fresh for $\bar{\beta}$ (**6**). Together, (3) and (6) imply $\bar{\beta} \# \mathrm{ftv}(C)$ (**7**), while (5) and (6) imply $\bar{\alpha} \# \mathrm{ftv}(D, \Gamma)$ (**8**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 4.30 to (1) yields $C, \varnothing \vdash v : \tau$ (**9**). Then, applying the induction hypothesis to (9), (4), and (7) yields a constraint H such that $H \Vdash D$ (**10**) and $C \equiv \exists\bar{\beta}.H$ (**11**) and, for every $x \in \mathrm{dpv}(p)$, $H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$ (**12**) holds. By placing (10) within the context $\exists\bar{\alpha}.[]$ and exploiting (8), we obtain $\exists\bar{\alpha}.H \Vdash D$. Furthermore, (11) implies $\exists\bar{\alpha}.C \equiv \exists\bar{\beta}.\exists\bar{\alpha}.H$. Last, by applying HIDE to (12) and (8), we obtain $\exists\bar{\alpha}.H, \varnothing \vdash [p \mapsto v]x : \Gamma(x)$. Thus, $\exists\bar{\alpha}.H$ is the desired witness. □

PROOF OF THEOREM 4.39. By Lemma 4.35, we may assume that the derivation of $C, \varnothing \vdash e : \sigma$ (**1**) is normal. Moreover, we may restrict our attention to the case where it ends with an instance of a syntax-directed rule; indeed, the general case follows immediately. We proceed by induction on the derivation of $e \to e'$.

∘ *Case* (β). e is $\lambda(p_1.e_1 \cdots p_n.e_n)\, v$ and e' is $[p_i \mapsto v]e_i$, for some $i \in \{1, \ldots, n\}$. The derivation of (1) ends with an instance of APP whose premises are $C, \varnothing \vdash \lambda(p_1.e_1 \cdots p_n.e_n) : \tau' \to \tau$ (**2**) and $C, \varnothing \vdash v : \tau'$ (**3**), where σ is τ. The derivation of (2) must end with an instance of ABS, whose premises include $C, \varnothing \vdash p_i.e_i : \tau' \to \tau$ (**4**). The derivation of (4) ends with an instance of CLAUSE whose premises are $C \vdash p_i : \tau' \rightsquigarrow \exists\bar{\beta}[D]\Gamma$ (**5**) and $C \wedge D, \Gamma \vdash e_i : \tau$ (**6**) and $\bar{\beta} \# \mathrm{ftv}(C, \tau)$ (**7**). By (3), (5), (7), and Lemma 4.37, there exists H such that $H \Vdash D$ (**8**) and $C \equiv \exists\bar{\beta}.H$ (**9**) and, for every $x \in \mathrm{dpv}(p_i)$, $H, \varnothing \vdash [p_i \mapsto v_i]x : \Gamma(x)$ (**10**) holds. By (9), we have $H \Vdash C$; together with

(8), this implies $H \Vdash C \wedge D$. Thus, applying Lemma 4.30 to (6), we find $H, \Gamma \vdash e_i : \tau$ (**11**). By Lemma 4.36, (10) and (11) imply $H, \varnothing \vdash [p_i \mapsto v_i]e_i : \tau$ (**12**). Applying HIDE to (12) and (7) and exploiting (9), we obtain $C, \varnothing \vdash [p_i \mapsto v_i]e_i : \tau$.

∘ *Case* (μ). e is $\mu x.v$ and e' is $[x \mapsto \mu x.v]v$. The derivation of (1) ends with an instance of FIX whose premise is $C, (x \mapsto \sigma) \vdash v : \sigma$. The result follows by Lemma 4.36.

∘ *Case* (let). e is $\mathsf{let}\, x = v \;\mathsf{in}\; e_1$ and e' is $[x \mapsto v]e_1$. The derivation of (1) must end with an instance of LET, whose premises are $C, \varnothing \vdash v : \sigma'$ and $C, (x \mapsto \sigma') \vdash e_1 : \sigma$. The result follows by Lemma 4.36.

∘ *Case* (context). By the induction hypothesis. □

PROOF OF LEMMA 4.40. By Lemma 4.35, we may assume, *w.l.o.g.*, that the derivation of $C, \varnothing \vdash v : \tau$ is normal. The proof proceeds with a structural induction on this derivation, where HIDE forms the inductive case and all other cases are base cases.

∘ *Case* HIDE. Our hypotheses are $\exists\bar{\alpha}.C, \varnothing \vdash v : \tau$ (**1**) and $\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta$ (**2**), where $\exists\bar{\alpha}.C$ is satisfiable. The judgment (1) follows from an instance of HIDE whose first premise is $C, \varnothing \vdash v : \tau$ (**3**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 4.30 to (2) yields $C \vdash p : \tau \rightsquigarrow \Delta$ (**4**). Because $\exists\bar{\alpha}.C$ is satisfiable, C is satisfiable as well. Applying the induction hypothesis to (3) and (4) yields the result.

We now reach the base case, where the derivation of $C, \varnothing \vdash v : \tau$ is normal and does not end with HIDE. We proceed by induction on the derivation of $C \vdash p : \tau \rightsquigarrow \Delta$.

∘ *Case* P-EMPTY. Then, p is 0, $\neg p$ is 1, so v matches $\neg p$.

∘ *Cases* P-WILD, P-VAR. Then, v matches p.

∘ *Case* P-AND. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p_1 \wedge p_2 : \tau \rightsquigarrow \Delta$ (**2**). The derivation of (2) ends with an instance of P-AND whose premises are, for every $i \in \{1, 2\}$, $C \vdash p_i : \tau \rightsquigarrow \Delta_i$ (**3**), where Δ is $\Delta_1 \times \Delta_2$. By the induction hypothesis, applied to (1) and (3), v matches $p_i \vee \neg p_i$, for every $i \in \{1, 2\}$. We conclude that v must match $\neg p_1 \vee \neg p_2 \vee (p_1 \wedge p_2)$, that is, $(p_1 \wedge p_2) \vee \neg(p_1 \wedge p_2)$.

∘ *Case* P-OR. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p_1 \wedge p_2 : \tau \rightsquigarrow \Delta$ (**2**). The derivation of (2) ends with an instance of P-OR whose premises are, for every $i \in \{1, 2\}$, $C \vdash p_i : \tau \rightsquigarrow \Delta$ (**3**). By the induction hypothesis, applied to (1) and (3), v matches $p_i \vee \neg p_i$, for every $i \in \{1, 2\}$. We conclude that v must match $p_1 \vee p_2 \vee (\neg p_1 \wedge \neg p_2)$, that is, $(p_1 \vee p_2) \vee \neg(p_1 \vee p_2)$.

∘ *Case* P-CSTR. Then, p is $K\, p_1 \cdots p_n$ and τ is $\varepsilon(\bar{\alpha})$. Because the derivation of $C, \varnothing \vdash v : \varepsilon(\bar{\alpha})$ is normal and does not end with HIDE, it must end with a syntax-directed rule, followed by SUB. (Indeed, any number of consecutive occurrences of SUB may be expanded or collapsed to a single one.) However, it cannot end with SUB(ABS(·)), because then an assertion of the form $C \Vdash \tau_1 \to \tau_2 \leq \varepsilon(\bar{\alpha})$ would hold—a contradiction, by Requirement 4.9, since C is satisfiable. So, it must end with SUB(CSTR(·)). As a result, v must be of the form $K'\, v_1 \cdots v_{n'}$. The data constructors K and K' cannot be associated with distinct data type declarations, because then an assertion of the form $C \Vdash \varepsilon'(\bar{\alpha}') \leq \varepsilon(\bar{\alpha})$, where ε and ε' are distinct, would hold—again, a contradiction. So, K' is associated with the data type ε.

If K and K' are distinct, then the pattern $K'\, 1 \cdots 1$ appears among the disjuncts in the definition of $\neg p$, so v matches $\neg p$.

Otherwise, K and n coincide with K' and n'. SUB's second premise is $C \Vdash \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha})$ (**1**), while CSTR's premises are $C, \varnothing \vdash v_i : \tau_i'$ (**2**), for every $i \in \{1, \ldots, n\}$, $K :: \forall\bar{\alpha}'\bar{\beta}'[D'].\tau_1' \cdots \tau_n' \to \varepsilon(\bar{\alpha}')$ (**3**), and $C \Vdash D'$ (**4**). The derivation of $C \vdash p : \tau \rightsquigarrow \Delta$ ends with an instance of P-CSTR

whose premises are $C \wedge D \vdash p_i : \tau_i \rightsquigarrow \Delta_i$ (**5**), for every $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ (**6**), and $\bar{\beta} \# \mathrm{ftv}(C)$ (**7**). We may further require, *w.l.o.g.*, $\bar{\beta} \# \bar{\alpha}'\bar{\beta}'$ (**8**). By Requirement 4.11, (3), (6), and (8) imply $D' \wedge \varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists\bar{\beta}.(D \wedge_i \tau'_i \leq \tau_i)$ (**9**). Together, (4), (1), (9), and (7) yield $C \Vdash \exists\bar{\beta}.(C \wedge D \wedge_i \tau'_i \leq \tau_i)$. Because C is satisfiable, this proves that $C \wedge D \wedge_i \tau'_i \leq \tau_i$ is satisfiable as well. Now, by Lemma 4.30 and by Sub, (2) and (5) yield $C \wedge D \wedge_i \tau'_i \leq \tau_i, \varnothing \vdash v_i : \tau_i$ and $C \wedge D \wedge_i \tau'_i \leq \tau_i \vdash p_i : \tau_i \rightsquigarrow \Delta_i$, respectively. Applying the induction hypothesis to these judgments, we find that v_i matches $p_i \vee \neg p_i$, for every $i \in \{1, \ldots, n\}$. Thus, v matches $K\,(p_1 \vee \neg p_1) \cdots (p_n \vee \neg p_n)$, which is contained in $p \vee \neg p$.

∘ *Case* p-EqIn. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). p-EqIn's premises are $C \vdash p : \tau' \rightsquigarrow \Delta$ (**3**) and $C \Vdash \tau = \tau'$ (**4**). Applying Sub to (1) and (4) yields a derivation of $C, \varnothing \vdash v : \tau'$ (**5**), which still is normal and does not end with Hide. There remains to apply the induction hypothesis to (5) and (3).

∘ *Case* p-SubOut. Our hypotheses are $C, \varnothing \vdash v : \tau$ (**1**) and $C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). p-SubOut's first premise is $C \vdash p : \tau \rightsquigarrow \Delta'$ (**3**). There remains to apply the induction hypothesis to (1) and (3).

∘ *Case* p-Hide. Our hypotheses are $\exists\bar{\alpha}.C, \varnothing \vdash v : \tau$ (**1**) and $\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta$ (**2**), where $\exists\bar{\alpha}.C$ is satisfiable. The judgment (2) follows from an instance of p-Hide whose first premise is $C \vdash p : \tau \rightsquigarrow \Delta$ (**3**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 4.30 to (1) yields $C, \varnothing \vdash v : \tau$ (**4**). Because $\exists\bar{\alpha}.C$ is satisfiable, C is satisfiable as well. Applying the induction hypothesis to (4) and (3) yields the result. □

Proof of Lemma 4.41. Left to the reader. □

Proof of Theorem 4.42. Suppose $C, \varnothing \vdash e : \sigma$ (**1**) where C is a satisfiable constraint. We may assume, *w.l.o.g.*, that the derivation of (1) is normal and ends with an instance of a syntax-directed-rule. The proof is by induction on the structure of e.

∘ *Case* e is x. Because well-typed expressions are closed, this case cannot occur.

∘ *Case* e is $\lambda(c_1 \cdots c_n).\ e$ is a value.

∘ *Case* e is $e_1\,e_2$. Because e is well-typed, so are e_1 and e_2. By the induction hypothesis, e_1 is either reducible or a value. In the former case, because $[]\,e_2$ is an evaluation context, e is reducible as well. Let us now assume the latter case. Then, by the induction hypothesis again, e_2 is either reducible or a value. In the former case, because e_1 is a value, $e_1\,[]$ is an evaluation context, so e is reducible. Let us now assume the latter case. The derivation of (1) ends with an instance of App whose premises are of the form $C, \varnothing \vdash e_1 : \tau' \to \tau$ (**2**) and $C, \varnothing \vdash e_2 : \tau'$ (**3**) for some satisfiable constraint C. We now reason by cases on the structure of the value e_1.

◇ *Sub-case* e_1 is x. Again, this case cannot occur.

◇ *Sub-case* e_1 is $K\,v_1 \cdots v_n$. Because any number of consecutive occurrences of Sub may be expanded or collapsed to a single one, we may assume that the derivation of (2) ends with the shape Sub(Cstr(·)). Sub's second premise must then be of the form $C \Vdash \varepsilon(\bar{\alpha}) \leq \tau' \to \tau$, a contradiction, by Requirement 4.9, since C is satisfiable.

◇ *Sub-case* e_1 is $\lambda(p_1.e'_1 \cdots p_n.e'_n)$. This case analysis must be exhaustive, which means that $\neg p_1 \wedge \cdots \wedge \neg p_n$ is empty. Thus, there exists $i \in \{1, \ldots, n\}$ such that the value e_2 does *not* match $\neg p_i$ (**4**). The derivation of (2) must end with an instance of Abs, preceded by instances of Clause, among whose premises we find $C \vdash p_i : \tau' \rightsquigarrow \Delta$ (**5**) for some Δ. Then, given (3), (5), and the satisfiability of C, Lemma 4.40 guarantees that e_2 matches $p_i \vee \neg p_i$ (**6**). Together, (4) and (6) show that e_2 matches p_i, which implies that e is reducible by (β).

∘ *Case* e is $\mu x.v.\ e$ is reducible by (μ).

∘ *Case* e is $\mathsf{let}\, x = e_1\ \mathsf{in}\ e_2$. Because e is well-typed, so is e_1. By the induction hypothesis, e_1 is either reducible or a value. In the former case, because $\mathsf{let}\, x = []\ \mathsf{in}\ e_2$ is an evaluation context, e is reducible as well. In the latter case, e is reducible by (let). □

Proof of Theorem 4.43. Suppose e reduces to e'. By Theorem 4.39, e' is well-typed. Because reduction preserves the property that all case analyses are exhaustive, Theorem 4.42 is applicable and guarantees that e' is not stuck. □

Proof of Theorem 4.44. Let $\lfloor e \rfloor$ be well-typed. Because, for every pattern p, $\neg p \wedge \neg\neg p$ is empty, every case analysis in $\lfloor e \rfloor$ is exhaustive. Thus, by Theorem 4.42, $\lfloor e \rfloor$ is either reducible or a value.

If $\lfloor e \rfloor$ is reducible, then it is straightforward to check that either e itself is reducible, or e is stuck and $\lfloor e \rfloor$ reduces to an expression of the form $E^*[\bot]$, where E^* stands for a stack of nested evaluation contexts. The latter case cannot arise, however, because $\lfloor e \rfloor$ is well-typed, while, by Lemma 4.41, $E^*[\bot]$ is not, contradicting the subject reduction property (Theorem 4.39). So, e is reducible.

If $\lfloor e \rfloor$ is a value, then it is straightforward to check, by induction on the definition of $\lfloor \cdot \rfloor$ that e is also a value. □

Proof of Theorem 4.45. Suppose e reduces to e'. In that case, it is straightforward to check that $\lfloor e \rfloor$ reduces to $\lfloor e' \rfloor$, so, by Theorem 4.39, $\lfloor e' \rfloor$ is well-typed. Then, Theorem 4.44 guarantees that e' is not stuck. □

Proof of Lemma 5.2. Left to the reader. □

Proof of Lemma 5.4. By induction on the structure of p.

∘ *Cases* p is 0, 1, or x. The goal follows immediately from p-Empty, p-Wild, or p-Var.

∘ *Case* p is $p_1 \wedge p_2$. By the induction hypothesis, for every $i \in \{1, 2\}$, we have $(\!|p_i \downarrow \tau|\!) \vdash p_i : \tau \rightsquigarrow (\!|p_i \uparrow \tau|\!)$. By Lemma 4.30 and by p-And, this implies the goal $(\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \vdash p_1 \wedge p_2 : \tau \rightsquigarrow (\!|p_1 \uparrow \tau|\!) \times (\!|p_2 \uparrow \tau|\!)$.

∘ *Case* p is $p_1 \vee p_2$. By the induction hypothesis, for every $i \in \{1, 2\}$, we have $(\!|p_i \downarrow \tau|\!) \vdash p_i : \tau \rightsquigarrow (\!|p_i \uparrow \tau|\!)$. By f-Lub and p-SubOut, $(\!|p_i \downarrow \tau|\!) \vdash p_i : \tau \rightsquigarrow (\!|p_1 \uparrow \tau|\!) + (\!|p_2 \uparrow \tau|\!)$ follows. By Lemma 4.30 and by p-Or, this implies the goal $(\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!) \vdash p_1 \vee p_2 : \tau \rightsquigarrow (\!|p_1 \uparrow \tau|\!) + (\!|p_2 \uparrow \tau|\!)$.

∘ *Case* p is $K\, p_1 \cdots p_n$. Let $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar{\alpha})$ (**1**), where $\bar{\alpha}\bar{\beta} \# \mathrm{ftv}(\tau)$ (**2**). By the induction hypothesis, for every $i \in \{1, \ldots, n\}$, we have $(\!|p_i \downarrow \tau_i|\!) \vdash p_i : \tau_i \rightsquigarrow (\!|p_i \uparrow \tau_i|\!)$ (**3**). Let C stand for $(\!|p_1 \downarrow \tau_1|\!) \wedge \cdots \wedge (\!|p_n \downarrow \tau_n|\!)$. We have $D \wedge \forall \bar{\beta}.D \Rightarrow C \Vdash C \Vdash (\!|p_i \downarrow \tau_i|\!)$ (**4**), where the left-hand entailment assertion is a logical tautology, while the right-hand assertion is by definition of C. Applying Lemma 4.30 to (3) and (4), we obtain $D \wedge \forall \bar{\beta}.D \Rightarrow C \vdash p_i : \tau_i \rightsquigarrow (\!|p_i \uparrow \tau_i|\!)$ (**5**). By p-Cstr, (5) and (1) imply $\forall \bar{\beta}.D \Rightarrow C \vdash p : \varepsilon(\bar{\alpha}) \rightsquigarrow \exists \bar{\beta}[D]\Delta$ (**6**), where Δ stands for $(\!|p_1 \uparrow \tau_1|\!) \times \cdots \times (\!|p_n \uparrow \tau_n|\!)$. Applying Lemma 4.30 and p-EqIn to (6), we find $\varepsilon(\bar{\alpha}) = \tau \wedge \forall \bar{\beta}.D \Rightarrow C \vdash p : \tau \rightsquigarrow \exists \bar{\beta}[D]\Delta$ (**7**).

Now, by f-Imply, we have $\tau = \varepsilon(\bar{\alpha}) \Vdash [D]\Delta \leq [D \wedge \tau = \varepsilon(\bar{\alpha})]\Delta$. By f-Hide and by transitivity of $\leq$, this implies $\tau = \varepsilon(\bar{\alpha}) \Vdash [D]\Delta \leq \exists \bar{\alpha}[D \wedge \tau = \varepsilon(\bar{\alpha})]\Delta$. By (2), $\bar{\beta}$ does not occur free in the left-hand side of this entailment assertion, which may thus be written $\tau = \varepsilon(\bar{\alpha}) \Vdash \forall \bar{\beta}.([D]\Delta \leq \exists \bar{\alpha}[D \wedge \tau = \varepsilon(\bar{\alpha})]\Delta)$. By f-Ex and by transitivity of entailment, this implies $\tau = \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}[D]\Delta \leq \exists \bar{\alpha}\bar{\beta}[D \wedge \tau = \varepsilon(\bar{\alpha})]\Delta$, that is, $\tau = \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}[D]\Delta \leq (\!|p \uparrow \tau|\!)$ (**8**).

By P-SubOut, (7) and (8) yield $\tau = \varepsilon(\bar\alpha) \wedge \forall\bar\beta.D \Rightarrow C \vdash p : \tau \rightsquigarrow (\!|p \uparrow \tau|\!)$. By (2) and P-Hide, this entails $\exists\bar\alpha.(\tau = \varepsilon(\bar\alpha) \wedge \forall\bar\beta.D \Rightarrow C) \vdash p : \tau \rightsquigarrow (\!|p \uparrow \tau|\!)$, that is, $(\!|p \downarrow \tau|\!) \vdash p : \tau \rightsquigarrow (\!|p \uparrow \tau|\!)$. □

Proof of Lemma 5.5. By structural induction on p. □

Proof of Lemma 5.6. By induction on the derivation of $C \vdash p : \tau \rightsquigarrow \Delta$ ($\mathcal{H}$).

∘ *Cases* P-Empty, P-Wild, P-Var. $(\!|p \downarrow \tau|\!)$ is true, so the first goal is a tautology. Furthermore, $(\!|p \uparrow \tau|\!)$ and Δ coincide, so the second goal follows from the reflexivity of $\leq$.

∘ *Case* P-And. ($\mathcal{H}$) is $C \vdash p_1 \wedge p_2 : \tau \rightsquigarrow \Delta_1 \wedge \Delta_2$. P-And's premises are $C \vdash p_i : \tau \rightsquigarrow \Delta_i$ (**1**), for every $i \in \{1, 2\}$. By the induction hypothesis, (1) implies $C \Vdash (\!|p_i \downarrow \tau|\!)$ (**2**) and $C \Vdash (\!|p_i \uparrow \tau|\!) \leq \Delta_i$ (**3**). The first goal, $C \Vdash (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!)$, follows from (2). The second goal, namely $C \Vdash (\!|p_1 \uparrow \tau|\!) \times (\!|p_2 \uparrow \tau|\!) \leq \Delta_1 \times \Delta_2$, follows from (3) by F-And.

∘ *Case* P-Or. ($\mathcal{H}$) is $C \vdash p_1 \vee p_2 : \tau \rightsquigarrow \Delta$. P-Or's premises are $C \vdash p_i : \tau \rightsquigarrow \Delta$ (**1**), for every $i \in \{1, 2\}$. By the induction hypothesis, (1) implies $C \Vdash (\!|p_i \downarrow \tau|\!)$ (**2**) and $C \Vdash (\!|p_i \uparrow \tau|\!) \leq \Delta$ (**3**). The first goal, $C \Vdash (\!|p_1 \downarrow \tau|\!) \wedge (\!|p_2 \downarrow \tau|\!)$, follows from (2). The second goal, namely $C \Vdash (\!|p_1 \uparrow \tau|\!) + (\!|p_2 \uparrow \tau|\!) \leq \Delta$, follows from (3) by F-Glb.

∘ *Case* P-Cstr. ($\mathcal{H}$) is $C \vdash K\, p_1 \cdots p_n : \varepsilon(\bar\alpha) \rightsquigarrow \exists\bar\beta[D](\Delta_1 \times \cdots \times \Delta_n)$. P-Cstr's premises are $C \vdash p_i : \tau_i \rightsquigarrow \Delta_i$ (**1**), for every $i \in \{1, \ldots, n\}$, $K :: \forall\bar\alpha\bar\beta[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar\alpha)$ (**2**) and $\bar\beta \# \mathrm{ftv}(C)$ (**3**). By the induction hypothesis, (1) implies $C \wedge D \Vdash (\!|p_i \downarrow \tau_i|\!)$ (**4**) and $C \wedge D \Vdash (\!|p_i \uparrow \tau_i|\!) \leq \Delta_i$ (**5**).

The assertions (4), where i ranges over $\{1, \ldots, n\}$, imply $C \wedge D \Vdash \wedge_i (\!|p_i \downarrow \tau_i|\!)$. This may be written $C \Vdash D \Rightarrow \wedge_i (\!|p_i \downarrow \tau_i|\!)$, and, by (3), $C \Vdash \forall\bar\beta.D \Rightarrow \wedge_i (\!|p_i \downarrow \tau_i|\!)$. By Lemma 5.2, this is exactly the first goal $C \Vdash (\!|K\, p_1 \cdots p_n \downarrow \varepsilon(\bar\alpha)|\!)$.

The assertions (5), where i ranges over $\{1, \ldots, n\}$, together with F-And, imply $C \wedge D \Vdash \times_i (\!|p_i \uparrow \tau_i|\!) \leq \times_i \Delta_i$. By F-Enrich, this implies $C \Vdash [D](\times_i (\!|p_i \uparrow \tau_i|\!)) \leq [D](\times_i \Delta_i)$. By (3), this may be written $C \Vdash \forall\bar\beta.([D](\times_i (\!|p_i \uparrow \tau_i|\!)) \leq [D](\times_i \Delta_i))$. By F-Ex and by transitivity of entailment, $C \Vdash \exists\bar\beta[D](\times_i (\!|p_i \uparrow \tau_i|\!)) \leq \exists\bar\beta[D](\times_i \Delta_i)$ follows. By Lemma 5.2, this is exactly the second goal $C \Vdash (\!|K\, p_1 \cdots p_n \uparrow \varepsilon(\bar\alpha)|\!) \leq \exists\bar\beta[D](\times_i \Delta_i)$.

∘ *Case* P-EqIn. P-EqIn's premises are $C \vdash p : \tau' \rightsquigarrow \Delta$ (**1**) and $C \Vdash \tau = \tau'$ (**2**). By the induction hypothesis, (1) implies $C \Vdash (\!|p \downarrow \tau'|\!)$ (**3**) and $C \Vdash (\!|p \uparrow \tau'|\!) \leq \Delta$ (**4**). By Lemma 5.5, we have $\tau = \tau' \wedge (\!|p \downarrow \tau'|\!) \Vdash (\!|p \downarrow \tau|\!)$ (**5**) and $\tau = \tau' \Vdash (\!|p \uparrow \tau|\!) \leq (\!|p \uparrow \tau'|\!)$ (**6**). By (2), (3) and (5), we obtain the first goal $C \Vdash (\!|p \downarrow \tau|\!)$. By (2) and (6), we get $C \Vdash (\!|p \uparrow \tau|\!) \leq (\!|p \uparrow \tau'|\!)$, which, combined with (4), yields the second goal $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$.

∘ *Case* P-SubOut. P-SubOut's premises are $C \vdash p : \tau \rightsquigarrow \Delta'$ (**1**) and $C \Vdash \Delta' \leq \Delta$ (**2**). By the induction hypothesis, (1) implies $C \Vdash (\!|p \downarrow \tau|\!)$ (**3**) and $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta'$ (**4**). The first goal is precisely (3). The second goal $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$ follows from (4) and (2).

∘ *Case* P-Hide. ($\mathcal{H}$) is $\exists\bar\alpha.C \vdash p : \tau \rightsquigarrow \Delta$. P-Hide's premises are $C \vdash p : \tau \rightsquigarrow \Delta$ (**1**) and $\bar\alpha \# \mathrm{ftv}(\tau, \Delta)$ (**2**). By the induction hypothesis, (1) implies $C \Vdash (\!|p \downarrow \tau|\!)$ (**3**) and $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$ (**4**). By (2), $\bar\alpha$ does not occur free in the right-hand sides of these entailment assertions. As a result, (3) and (4) respectively imply $\exists\bar\alpha.C \Vdash (\!|p \downarrow \tau|\!)$ and $\exists\bar\alpha.C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$, which are the first and second goals. □

Proof of Theorem 5.10. By induction on the structure of *ce*.

∘ *Case* *ce* is x. Write $\Gamma(x)$ as $\forall\bar\alpha[D].\tau'$, where $\bar\alpha \# \mathrm{ftv}(\Gamma, \tau)$ (**1**). By Var, Inst, and Sub, we have $D \wedge \tau' \leq \tau, \Gamma \vdash x : \tau$. By (1) and Hide, this implies $\exists\bar\alpha.(D \wedge \tau' \leq \tau), \Gamma \vdash x : \tau$, which is precisely the goal $\Gamma(x) \leq \tau, \Gamma \vdash x : \tau$.

∘ *Case ce* is $e_1\,e_2$. Let $\alpha \# \mathrm{ftv}(\Gamma,\tau)$ (**1**). By the induction hypothesis, we have $(\!|\Gamma \vdash e_1 : \alpha \to \tau|\!), \Gamma \vdash e_1 : \alpha \to \tau$ and $(\!|\Gamma \vdash e_2 : \alpha|\!), \Gamma \vdash e_2 : \alpha$. By Lemma 4.30 and App, this yields $(\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!), \Gamma \vdash e_1\,e_2 : \tau$. The result follows by (1) and Hide.

∘ *Case ce* is $K\,e_1 \cdots e_n$. Let $K :: \forall\bar\alpha\bar\beta[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar\alpha)$ (**1**), where $\bar\alpha\bar\beta \# \mathrm{ftv}(\Gamma,\tau)$ (**2**). By the induction hypothesis, $(\!|\Gamma \vdash e_i : \tau_i|\!), \Gamma \vdash e_i : \tau_i$ holds for every $i \in \{1,\ldots,n\}$. By Lemma 4.30, $\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar\alpha) \le \tau, \Gamma \vdash e_i : \tau_i$ (**3**) follows. Applying Cstr to (3), where i ranges over $\{1,\ldots,n\}$, and to (1), we obtain $\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar\alpha) \le \tau, \Gamma \vdash K\,e_1 \cdots e_n : \varepsilon(\bar\alpha)$. By Sub, (2), and Hide, this implies $\exists\bar\alpha\bar\beta.(\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar\alpha) \le \tau), \Gamma \vdash K\,e_1 \cdots e_n : \tau$, that is, $(\!|\Gamma \vdash K\,e_1 \cdots e_n : \tau|\!), \Gamma \vdash K\,e_1 \cdots e_n : \tau$.

∘ *Case ce* is $\mathsf{let}\,x = e_1\ \mathsf{in}\ e_2$. Let $\alpha \# \mathrm{ftv}(\Gamma,\tau)$ (**1**). Let C and σ stand respectively for $(\!|\Gamma \vdash e_1 : \alpha|\!)$ and $\forall\alpha[C].\alpha$. By the induction hypothesis, we have $C, \Gamma \vdash e_1 : \alpha$ (**2**) and $(\!|\Gamma[x \mapsto \sigma] \vdash e_2 : \tau|\!), \Gamma[x \mapsto \sigma] \vdash e_2 : \tau$ (**3**). Applying Gen to (2) and (1) yields $\exists\alpha.C, \Gamma \vdash e_1 : \sigma$ (**4**). The goal follows from (3) and (4) by Lemma 4.30 and Let.

∘ *Case ce* is $\mu(x : \exists\bar\beta.\sigma).\lambda(c_1,\ldots,c_n)$. Up to a renaming of $\bar\beta$ and σ, we may assume, *w.l.o.g.*, $\bar\beta \# \mathrm{ftv}(\Gamma,\tau)$ (**1**). Write σ as $\forall\bar\gamma[C].\tau_1 \to \tau_2$, where $\bar\gamma \# \mathrm{ftv}(\Gamma)$ (**2**). By the induction hypothesis, we have, for every $i \in \{1,\ldots,n\}$, $(\!|\Gamma[x \mapsto \sigma] \vdash c_i : \tau_1 \to \tau_2|\!), \Gamma[x \mapsto \sigma] \vdash c_i : \tau_1 \to \tau_2$ (**3**). Let D stand for $\forall\bar\gamma.C \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2|\!)$. Then, $C \wedge D$ entails $(\!|\Gamma[x \mapsto \sigma] \vdash c_i : \tau_1 \to \tau_2|\!)$, so, by Lemma 4.30, (3) implies $C \wedge D, \Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2$ (**4**). Furthermore, we have $\bar\gamma \# \mathrm{ftv}(D)$ (**5**). By (4), (2), (5), and FixAbs, we obtain $\exists\bar\gamma.C \wedge D, \Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda\bar c : \sigma$ (**6**). Because $\sigma \le \tau$ entails $\exists\bar\gamma.C$, (6) and Lemma 4.30 yield $\sigma \le \tau \wedge D, \Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda\bar c : \sigma$ (**7**). By Inst, Sub, and Hide, (7) implies $\sigma \le \tau \wedge D, \Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda\bar c : \tau$ (**8**). By (8), (1), and Hide, we obtain $\exists\bar\beta.(\sigma \le \tau \wedge D), \Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda\bar c : \tau$, which by definition of D is the goal.

∘ *Case ce* is *p.e*. Then, τ is of the form $\tau_1 \to \tau_2$. Write $(\!|p \uparrow \tau_1|\!)$ as $\exists\bar\beta[D]\Gamma'$, where $\bar\beta \# \mathrm{ftv}(\Gamma,\tau_1,\tau_2)$ (**1**). By the induction hypothesis, $(\!|\Gamma\Gamma' \vdash e : \tau_2|\!), \Gamma\Gamma' \vdash e : \tau_2$ (**2**) holds. Furthermore, by Lemma 5.4, we have $(\!|p \downarrow \tau_1|\!) \vdash p : \tau_1 \rightsquigarrow \exists\bar\beta[D]\Gamma'$ (**3**). Now, recall that, by definition, $(\!|\Gamma \vdash ce : \tau|\!)$ is $(\!|p \downarrow \tau_1|\!) \wedge \forall\bar\beta.D \Rightarrow (\!|\Gamma\Gamma' \vdash e : \tau_2|\!)$. As a result, by Lemma 4.30, (2) and (3) respectively imply $(\!|\Gamma \vdash ce : \tau|\!) \wedge D, \Gamma\Gamma' \vdash e : \tau_2$ (**4**) and $(\!|\Gamma \vdash ce : \tau|\!) \vdash p : \tau_1 \rightsquigarrow \exists\bar\beta[D]\Gamma'$ (**5**). By (4), (5), (1), and Clause, we obtain the goal $(\!|\Gamma \vdash ce : \tau|\!), \Gamma \vdash p.e : \tau_1 \to \tau_2$. □

Proof of Lemma 5.11. By induction on the structure of *ce*. □

Proof of Lemma 5.12. By induction on the structure of *ce*. □

Proof of Lemma 5.13. We assume $\bar\beta_1\bar\beta_2 \# \mathrm{ftv}(\Gamma,\tau)$ (**1**). Up to a renaming of the goal, we may assume, *w.l.o.g.*, $\bar\beta_1 \# \mathrm{ftv}(\exists\bar\beta_2[D_2]\Gamma_2)$ (**2**) and $\bar\beta_2 \# \Gamma_1$ (**3**). By Lemma 5.12, we have $\Gamma_1 \le \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$. By (1) and (3), $\bar\beta_2$ does not appear in the right-hand-side of this entailment assertion, so it can be existentially quantified in its left-hand-side, which yields $\exists\bar\beta_2.(\Gamma_1 \le \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!)) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$ (**4**). By (2) and (3), we have $\exists\bar\beta_1[D_1]\Gamma_1 \le \exists\bar\beta_2[D_2]\Gamma_2 \wedge D_1 \Vdash \exists\bar\beta_2.(D_2 \wedge \Gamma_1 \le \Gamma_2)$; then $\exists\bar\beta_1[D_1]\Gamma_1 \le \exists\bar\beta_2[D_2]\Gamma_2 \wedge D_1 \wedge \forall\bar\beta_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \exists\bar\beta_2.(\Gamma_1 \le \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!))$. By transitivity with (4), $\exists\bar\beta_1[D_1]\Gamma_1 \le \exists\bar\beta_2[D_2]\Gamma_2 \wedge D_1 \wedge \forall\bar\beta_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$ (**5**) follows. By (1), (2) and (3), $\bar\beta_1 \# \mathrm{ftv}(\exists\bar\beta_1[D_1]\Gamma_1 \le \exists\bar\beta_2[D_2]\Gamma_2 \wedge \forall\bar\beta_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!))$, so (4) can be rewritten into $\exists\bar\beta_1[D_1]\Gamma_1 \le \exists\bar\beta_2[D_2]\Gamma_2 \wedge \forall\bar\beta_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \forall\bar\beta_1.D_1 \Rightarrow (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$, which is the goal. □

Proof of Theorem 5.14. We proceed by induction on the derivation of $C, \Gamma \vdash ce : \forall\bar\alpha[D].\tau$ ($\mathcal{H}$). Let $\sigma = \forall\bar\alpha[D].\tau$. Because $\bar\alpha$ is α-convertible in the statement of the theorem, we may assume,

w.l.o.g., $\bar\alpha \# \mathrm{ftv}(C)$, so that the goal is equivalent to $C \wedge D \Vdash (\!|\Gamma \vdash ce : \tau|\!)$. For the same reason, in cases FixAbs and Gen below, we may assume that $\bar\alpha$ coincides with the vector of type variables that appears in the rule's premises.

∘ *Case* Var. Var's first premise is $\Gamma(x) = \forall\bar\alpha[D].\tau$. The goal $C \wedge D \Vdash \Gamma(x) \leq \tau$ follows from Lemma 4.5.

∘ *Case* Cstr. $(\mathcal{H})$ is $C, \Gamma \vdash K\, e_1 \cdots e_n : \varepsilon(\bar\alpha)$. Cstr's premises are $C, \Gamma \vdash e_i : \tau_i$ (**1**), for every $i \in \{1, \ldots, n\}$, $K :: \forall\bar\alpha\bar\beta[D].\tau_1 \cdots \tau_n \to \varepsilon(\bar\alpha)$ (**2**) and $C \Vdash D$ (**3**). By the induction hypothesis, (1) implies $C \Vdash (\!|\Gamma \vdash e_i : \tau_i|\!)$ (**4**). By (4) and (3), we obtain $C \Vdash \wedge_i (\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D$, whence $C \Vdash \exists\bar\beta.(\wedge_i (\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D)$ (**5**). Furthermore, we let the reader check that, by definition of constraint generation, $\exists\bar\beta.(\wedge_i (\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D)$ entails $(\!|\Gamma \vdash K\, e_1 \cdots e_n : \varepsilon(\bar\alpha)|\!)$ (**6**). Combining (5) and (6) yields the goal $C \Vdash (\!|\Gamma \vdash K\, e_1 \cdots e_n : \varepsilon(\bar\alpha)|\!)$.

∘ *Case* App. $(\mathcal{H})$ is $C, \Gamma \vdash e_1\, e_2 : \tau$. App's premises are $C, \Gamma \vdash e_1 : \tau' \to \tau$ (**1**) and $C, \Gamma \vdash e_2 : \tau'$ (**2**). By the induction hypothesis, (1) and (2) imply $C \Vdash (\!|\Gamma \vdash e_1 : \tau' \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \tau'|\!)$ (**3**). Pick $\alpha \notin \mathrm{ftv}(\Gamma, C, \tau', \tau)$ (**4**). By (4), we have $C \Vdash \exists\alpha.(C \wedge \alpha = \tau')$ (**5**). Furthermore, by Lemma 5.11, (3) implies $C \wedge \alpha = \tau' \Vdash (\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!)$ (**6**). Combining (5) and (6), we obtain $C \Vdash \exists\alpha.((\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!))$, that is, $C \Vdash (\!|\Gamma \vdash e_1\, e_2 : \tau|\!)$.

∘ *Case* FixAbs. $(\mathcal{H})$ is $C \wedge \exists\bar\alpha.D, \Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda e : \sigma$. FixAbs's premises are $C \wedge D, \Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2$ (**1**), $\bar\alpha \# \mathrm{ftv}(C, \Gamma)$ (**2**), and $\sigma = \forall\bar\alpha[D].\tau_1 \to \tau_2$ (**3**). By the induction hypothesis, (1) implies $C \wedge D \Vdash (\!|\Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2|\!)$, which, by (2), may be written $C \Vdash \forall\bar\alpha.D \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2|\!)$ (**4**). Furthermore, by (3) and by Lemma 4.5, we have $D \Vdash \sigma \leq \tau_1 \to \tau_2$ (**5**). Combining (4) and (5), we obtain $C \wedge D \Vdash (\forall\bar\alpha.D \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash \bar c : \tau_1 \to \tau_2|\!)) \wedge \sigma \leq \tau_1 \to \tau_2$ (**6**). We let the reader check that, by definition of constraint generation, the right-hand side of (6) entails $(\!|\Gamma \vdash \mu(x : \exists\bar\beta.\sigma).\lambda e : \tau_1 \to \tau_2|\!)$. The goal follows.

∘ *Case* Let. $(\mathcal{H})$ is $C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \sigma$. Let's premises are $C, \Gamma \vdash e_1 : \sigma'$ (**1**) and $C, \Gamma[x \mapsto \sigma'] \vdash e_2 : \sigma$ (**2**). Write σ as $\forall\bar\alpha[D].\tau$, where $\bar\alpha \# \mathrm{ftv}(\Gamma)$ (**3**). Write σ' as $\forall\bar\alpha'[D'].\tau'$, where $\bar\alpha' \# \mathrm{ftv}(\Gamma, C)$ (**4**). By the induction hypothesis, (1) and (4) imply $C \wedge D' \Vdash (\!|\Gamma \vdash e_1 : \tau'|\!)$ (**5**), while (2) and (3) imply $C \wedge D \Vdash (\!|\Gamma[x \mapsto \sigma'] \vdash e_2 : \tau|\!)$ (**6**). Pick $\alpha \notin \mathrm{ftv}(\Gamma, \tau, \tau')$ (**7**). Let H stand for $(\!|\Gamma \vdash e_1 : \alpha|\!)$. By (7), the constraint $(\!|\Gamma \vdash e_1 : \tau'|\!)$ entails $\exists\alpha.((\!|\Gamma \vdash e_1 : \tau'|\!) \wedge \alpha = \tau')$, which by Lemma 5.11 entails $\exists\alpha.(H \wedge \alpha \leq \tau')$. Combining this fact with (5), we obtain $C \wedge D' \Vdash \exists\alpha.(H \wedge \alpha \leq \tau')$. By (4), this may be written $C \Vdash \forall\bar\alpha'.D' \Rightarrow \exists\alpha.(H \wedge \alpha \leq \tau')$, which by (7) is $C \Vdash \forall\alpha[H].\alpha \leq \sigma'$ (**8**). By (6), (8), and Lemma 5.12, we obtain $C \wedge D \Vdash (\!|\Gamma[x \mapsto \forall\alpha[H].\alpha] \vdash e_2 : \tau|\!)$ (**9**). By Lemma 4.31, (1) implies $C \Vdash \exists\bar\alpha'.D'$, which, together with (8), yields $C \Vdash \exists\alpha.H$ (**10**). The goal $C \wedge D \Vdash (\!|\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau|\!)$ follows from (9) and (10).

∘ *Case* Clause. $(\mathcal{H})$ is $C, \Gamma \vdash p.e : \tau_1 \to \tau_2$. Clause's premises are $C \vdash p : \tau_1 \rightsquigarrow \exists\bar\beta[D]\Gamma'$ (**1**), $C \wedge D, \Gamma\Gamma' \vdash e : \tau_2$ (**2**), and $\bar\beta \# \mathrm{ftv}(C, \Gamma, \tau_2)$ (**3**). Up to a renaming of Clause's second premise, we may further assume, *w.l.o.g.*, $\bar\beta \# \mathrm{ftv}(\tau_1)$ (**4**). By applying Lemma 5.6 to (1), we obtain $C \Vdash (\!|p \downarrow \tau_1|\!)$ (**5**) and $C \Vdash (\!|p \uparrow \tau_1|\!) \leq \exists\bar\beta[D]\Gamma'$ (**6**). By the induction hypothesis, (2) implies $C \wedge D \Vdash (\!|\Gamma\Gamma' \vdash e : \tau_2|\!)$, which, by (3), may be written $C \Vdash \forall\bar\beta.D \Rightarrow (\!|\Gamma\Gamma' \vdash e : \tau_2|\!)$ (**7**). Write $(\!|p \uparrow \tau_1|\!)$ as $\exists\bar\beta_1[D_1]\Gamma'_1$, where $\bar\beta_1 \# \mathrm{ftv}(\Gamma, C, \tau_1, \tau_2, \bar\beta)$ (**8**). By (3) and (8), $\bar\beta\bar\beta_1 \# \mathrm{ftv}(\Gamma, \tau_2)$ (**9**) holds. Applying Lemma 5.13 to (9) and combining the result with (6) and (7), we find $C \Vdash \forall\bar\beta_1.D_1 \Rightarrow (\!|\Gamma\Gamma'_1 \vdash e : \tau_2|\!)$ (**10**). Combining (5) and (10), we obtain $C \Vdash (\!|p \downarrow \tau_1|\!) \wedge \forall\bar\beta_1.D_1 \Rightarrow (\!|\Gamma\Gamma'_1 \vdash e : \tau_2|\!)$. By (8), this is the goal $C \Vdash (\!|\Gamma \vdash p.e : \tau_1 \to \tau_2|\!)$.

∘ *Case* Gen. $(\mathcal{H})$ is $C \wedge \exists\bar\alpha.D, \Gamma \vdash ce : \forall\bar\alpha[D].\tau$. Gen's first premise is $C \wedge D, \Gamma \vdash ce : \tau$. By the induction hypothesis, this implies $C \wedge D \Vdash (\!|\Gamma \vdash ce : \tau|\!)$, which is precisely the goal.

∘ *Case* INST. $(\mathcal{H})$ is $C, \Gamma \vdash ce : \tau$. INST's premises are $C, \Gamma \vdash ce : \forall\bar{\alpha}[D].\tau$ (**1**) and $C \Vdash D$ (**2**). Let θ be a renaming of $\bar{\alpha}$ such that θ is fresh for $\forall\bar{\alpha}[D].\tau$ (**3**) and $\theta\bar{\alpha}$ # ftv(Γ) (**4**). By (3), (1) may be written $C, \Gamma \vdash ce : \forall(\theta\bar{\alpha})[\theta D].\theta\tau$, which by (4) and by the induction hypothesis implies $C \Vdash \forall\theta\bar{\alpha}.\theta D \Rightarrow (\!|\Gamma \vdash ce : \theta\tau|\!)$. We let the reader check that, using (2), the goal $C \Vdash (\!|\Gamma \vdash ce : \tau|\!)$ follows.

∘ *Case* SUB. $(\mathcal{H})$ is $C, \Gamma \vdash ce : \tau$. SUB's premises are $C, \Gamma \vdash ce : \tau'$ (**1**) and $C \Vdash \tau' \leq \tau$ (**2**). By the induction hypothesis, (1) implies $C \Vdash (\!|\Gamma \vdash ce : \tau'|\!)$ (**3**). Combining (3) and (2) and applying Lemma 5.11 yields the goal $C \Vdash (\!|\Gamma \vdash ce : \tau|\!)$.

∘ *Case* HIDE. $(\mathcal{H})$ is $\exists\bar{\beta}.C, \Gamma \vdash ce : \sigma$. HIDE's premises are $C, \Gamma \vdash ce : \sigma$ (**1**) and $\bar{\beta}$ # ftv(Γ, σ) (**2**). Write σ as $\forall\bar{\alpha}[D].\tau$, where $\bar{\alpha}$ # ftv(Γ). By the induction hypothesis, (1) implies $C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash ce : \tau|\!)$ (**3**). By (2), $\bar{\beta}$ does not occur free in the right-hand side of this entailment assertion. Thus, (3) implies the goal $\exists\bar{\beta}.C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash ce : \tau|\!)$. □

PROOF OF LEMMA 6.5. By structural induction on p. If p is 0, 1 or x, the goal is the tautology $\mathsf{true} \equiv \forall\bar{\gamma}.C \Rightarrow \mathsf{true}$. If p is $p_1 \wedge p_2$ or $p_1 \vee p_2$, the goal follows from the induction hypothesis and from the distributivity of $\Rightarrow$ and $\forall$ with respect to $\wedge$. If p is $K\, p_1 \cdots p_n$, the goal follows from the induction hypothesis, Lemma 4.3, and the distributivity of $\Rightarrow$ and $\forall$ with respect to $\wedge$. □

PROOF OF LEMMA 6.6. By structural induction on p and by inspection of the rules in Figure 10. Requirement 6.1 is exploited. □

PROOF OF THEOREM 6.9. By structural induction on clauses and expressions. If c is $p.e$, the goal follows from Lemma 6.5, the induction hypothesis, and the distributivity of $\Rightarrow$ and $\forall$ with respect to $\wedge$. If e is $\mu(x : \exists\bar{\beta}.\sigma).\lambda\bar{c}$, where $\bar{c}$ is guarded, the goal follows from the induction hypothesis and the distributivity of $\Rightarrow$ and $\forall$ with respect to $\wedge$. In all other cases, the goal is an immediate consequence of the induction hypothesis. □

PROOF OF THEOREM 6.10. By structural induction on clauses and expressions, by inspection of the rules in Figure 10, and by Lemma 6.6. Requirements 6.1 and 6.3 is exploited. □

PROOF OF THEOREM 6.14. *Point (i).* Define the *weight* of a tractable constraint by:

$$\begin{aligned} w(\mathsf{true}) = w(\mathsf{false}) = w(\tau_1 = \tau_2) &= 2 \\ w(L_1 \vee L_2) &= w(L_1) + w(L_2) + 2 \\ w(L_1 \wedge L_2) &= w(L_1) \times w(L_2) - 2 \\ w(\exists\bar{\alpha}.R) &= w(R) \\ w(\forall\bar{\beta}.L \Rightarrow R) &= w(R)^{w(L)-1} \end{aligned}$$

For every constraint L, we have $w(L) \geq 2$ and, for every unification constraint U, $w(U) = 2$ holds. Using these properties, it is straightforward to check that S-AND-OR and S-ALL-OR are weight decreasing, while the remaining rules are non-weight increasing. Thus, it is sufficient to check that S-UNIF, S-ALL-FALSE and S-ALL form a strongly normalizing system. The last two rules strictly decrease the number of implication constructs, which is preserved by S-UNIF. So, the result follows from point (i) of Definition 6.12.

Point (ii). It is sufficient to separately prove that every rule in Figure 11 is meaning preserving. All cases are straightforward, excepted that of S-ALL, which we now detail. Assume $\exists\bar{\beta}_2.R_1 \equiv \mathsf{true}$

and R_1 determines $\bar{\beta}_2$. Then, by Lemma 4.3, we have $\forall\bar{\beta}_2.R_1 \Rightarrow R_2 \equiv \exists\bar{\beta}_2.(R_1 \wedge R_2)$. This implies $\forall\bar{\beta}_1\bar{\beta}_2.R_1 \Rightarrow R_2 \equiv \forall\bar{\beta}_1.\exists\bar{\beta}_2.(R_1 \wedge R_2)$, which is the goal.

Points (iii) and (iv). Let R be a normal form for $\rightarrowtriangle$. Assume, by way of contradiction, that R is not a standard unification constraint, that is, not of the form U. Then, R must contain a rigid implication construct. Let us consider the innermost such construct. It must be of the form $\forall\bar{\beta}.L \Rightarrow R'$, where L is not true (otherwise this is a plain universal quantification construct). Because neither S-AND-OR nor S-ALL-OR is applicable, L cannot contain a disjunction; so, it must be a unification constraint U. Because S-UNIF is not applicable, U must be a normal form for $\rightarrowtriangle_{\mathsf{u}}$. Then, by points (iii) and (iv) of Definition 6.12, one of S-ALL-FALSE and S-ALL must be applicable, a contradiction. We conclude that R is in fact a standard unification constraint. Because it is a normal form for S-UNIF, it must be a normal form for $\rightarrowtriangle_{\mathsf{u}}$. Then, the result follows from points (iii) and (iv) of Definition 6.12. □

References

James Cheney and Ralf Hinze. First-class phantom types. Technical Report 1901, Cornell University, 2003.

Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, November 1998.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277, 1991.

Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω. PhD thesis, Université Paris 7, September 1976.

Mark P. Jones. From Hindley-Milner types to first-class structures. Research Report YALEU/DCS/RR-1075, Yale University, June 1995.

Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. Technical Report 561, Université Paris-Sud, April 1990.

Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *IEEE Symposium on Logic in Computer Science (LICS)*, June 2003.

Jean-Louis Lassez, Michael J. Maher, and Kim G. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kaufmann, 1988.

Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, July 2004.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999. Technical Report MIT/LCS/TR-783.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–67, January 1996.

Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

Christine Paulin-Mohring. Inductive definitions in the system Coq: rules and properties. Research Report RR1992-49, ENS Lyon, 1992.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 89–98, January 2004.

François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer Verlag, April 1994.

J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

Vincent Simonet. An extension of HM(X) with bounded existential and universal data-types. In *ACM International Conference on Functional Programming (ICFP)*, June 2003a.

Vincent Simonet. The Flow Caml system: documentation and user's manual. Technical Report 0282, INRIA, July 2003b.

Vincent Simonet. Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In *Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*. Springer Verlag, November 2003c.

Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, September 1996.

Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy (S&P)*, May 2004.

Sergei G. Vorobyov. An improved lower bound for the elementary theories of trees. In *International Conference on Automated Deduction (CADE)*, volume 1104 of *Lecture Notes in Computer Science*, pages 275–287. Springer Verlag, 1996.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Stephanie Weirich. Type-safe cast: Functional pearl. In *ACM International Conference on Functional Programming (ICFP)*, pages 58–67, September 2000.

Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, December 1998.

Hongwei Xi. Dead code elimination through dependent types. In *International Workshop on Practical Aspects of Declarative Languages (PADL)*, volume 1551 of *Lecture Notes in Computer Science*, pages 228–242. Springer Verlag, January 1999.

Hongwei Xi. Dependent ML, 2001.

Hongwei Xi. Dependently Typed Pattern Matching. *Journal of Universal Computer Science*, 9(8): 851–872, 2003.

Hongwei Xi. Applied type system. In *TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer Verlag, February 2004.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 224–235, January 2003.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 214–227, January 1999.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

Christoph Zenger. *Indizierte Typen*. PhD thesis, Universität Karlsruhe, July 1998.

Lantian Zheng and Andrew C. Myers. Dynamic security labels and noninterference. Technical Report 2004-1924, Cornell University, January 2004.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399