A Constraint-Based Approach to Guarded Algebraic Data Types

VINCENT SIMONET and FRANÇOIS POTTIER
INRIA

We study HMG(X), an extension of the constraint-based type system HM(X) with deep pattern matching, polymorphic recursion, and *guarded algebraic data types*. Guarded algebraic data types subsume the concepts known in the literature as *indexed types*, *guarded recursive datatype constructors*, *(first-class) phantom types*, and *equality qualified types*, and are closely related to *inductive types*. Their characteristic property is to allow every branch of a `case` construct to be typechecked under different assumptions about the type variables in scope. We prove that HMG(X) is sound and that, provided recursive definitions carry a type annotation, type inference can be reduced to constraint solving. Constraint solving is decidable, at least for some instances of X, but prohibitively expensive. Effective type inference for guarded algebraic data types is left as an issue for future research.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types; Data types and structures; Patterns*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Functional constructs; Type structure*

General Terms: Languages, Theory

1. INTRODUCTION

Members of the ML family of programming languages offer *type inference* in the style of Hindley [1969] and Milner [1978], making type annotations optional. Type inference can be decomposed into constraint generation and constraint solving phases, where constraints are, roughly speaking, systems of type equations. This remark has led to the definition of a family of constraint-based type systems, known as HM(X) [Odersky et al. 1999; Pottier and Rémy 2005], whose members exploit potentially more complex constraint languages, achieving greater expressiveness while still enjoying type inference in the style of Hindley and Milner.

These programming languages also provide high-level facilities for defining and manipulating data structures, namely *algebraic data types* and *pattern matching*. In the setting of an explicitly typed calculus, Xi, Chen, and Chen [2003] have recently introduced *guarded algebraic data types*, an extension that offers significant new expressive power to programmers.

The purpose of the present paper is to study how these two lines of research

Authors' address: INRIA, B.P. 105, 78153 Le Chesnay Cedex, France.

can be brought together. In order to introduce guarded algebraic data types into mainstream programming languages such as Haskell [Peyton Jones 2003] or Objective Caml [Leroy et al. 2005], it is necessary to study their interaction with Hindley-Milner-style type inference. Furthermore, because type inference is best understood in a constraint-based setting, and because constraint-based type systems are more general than Hindley and Milner's original type system, we believe it is worth studying an extension of HM(X) with guarded algebraic data types.

We proceed as follows. First, we present an untyped call-by-value λ-calculus featuring data constructors and pattern matching (§2). (We believe that our results could be transferred to a call-by-name calculus without difficulty.) Then, we define the type system HMG(X), which extends HM(X) with pattern matching, polymorphic recursion, and guarded algebraic data types, and establish *subject reduction* and *progress* theorems (§3). Last, we show that, provided recursive definitions carry a type annotation, type inference reduces to constraint solving (§4).

The design of HMG(X) and the development of its basic theory are the main contributions of this paper. By studying arbitrary constraints, including subtyping constraints, as well as pattern matching against deep patterns, we address issues that most other treatments of guarded algebraic data types avoid. In spite of these complications, we provide concise proofs of type soundness and of the reduction of type inference to constraint solving. In addition to the basic subject reduction and progress results, we discuss a few subtle aspects of type soundness. In particular, we study a type-based criterion for determining whether a case analysis is exhaustive, and prove that it preserves type soundness. We also prove that, even though our (standard) dynamic semantics implicitly assumes that values can be distinguished at runtime based on their nature (for instance, a λ-abstraction does not match a pair pattern), well-typed programs do not exploit this assumption, and do not require values to carry runtime type information. This theoretical framework is shared between all instances of HMG(X).

This work does not fully resolve the issue of combining guarded algebraic data types with type inference in the style of Hindley and Milner. Because HMG(X) is parameterized over the syntax and interpretation of constraints, we do not provide a constraint solver. Yet, constraint solving raises difficult issues. Because our constraints involve the implication connective, constraint solving is likely to be computationally expensive, and solved forms are likely to be complex. In fact, even in the simplest possible case, that of unification constraints, the complexity of constraint solving is nonelementary.

Thus, although HMG(X) is a safe type system, it must probably be restricted in order to ensure feasible type inference. We and other authors have begun studying such restrictions. We briefly discuss these works below (§1.5), but, because we believe that no definitive solution has emerged yet, we leave this issue open in the present paper.

The remainder of this introduction presents guarded algebraic data types (§1.1), reviews some of their applications (§1.2–§1.4), and discusses related work (§1.5).

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 can be derived from the declaration of ε, is of the form

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

where n is the arity of K. (This form includes so-called *nested* algebraic data types [Bird and Meertens 1998].) Then, the typing discipline for pattern matching can 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, here is the definition of an algebraic data type $\mathit{tree}(\alpha)$, describing binary trees whose internal nodes are labeled with values of type α. It consists of the following data constructors:

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

The arity of *Leaf* is 0; the arity of *Node* is 3. 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. Some extensions of ML allow more liberal 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 is of the form

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

The novelty resides in the fact that the argument types $\tau_1 \times \cdots \times \tau_n$ 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:

$$\mathit{Key} :: \forall\beta.\beta \times (\beta \to \mathit{int}) \to \mathit{key}$$

The values $\mathit{Key}\,(3, \lambda x.x{+}5)$ and $\mathit{Key}\,([1;2;3], \mathit{length})$ both have type *key*. Matching either of them against the pattern $\mathit{Key}\,(v, f)$ binds the variables v and f to values of type β and $\beta \to \mathit{int}$, for an abstract 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 receive *constrained* type schemes:

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

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 return 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 $\bar{\beta}$ such that D *is satisfied* and 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 refer to this flavor of algebraic data types as *guarded*, because their data constructors have guarded (constrained) type schemes.

Guarded algebraic data types are a fairly general concept, 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: in the following, we suggest a few.

1.2 Applications with unification constraints

In the simplest case, types are interpreted as finite trees, and constraints are *unification constraints*, made up of type equations, conjunction, and existential quantification.

An alternative form. In that case, the data constructors associated with guarded algebraic data types can in fact be assigned unconstrained type schemes of the form

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

where every type variable that appears free in $\tau_1, \ldots, \tau_n, \tau$ is a member of $\bar{\beta}$. In this form, no constraint is specified, but the type constructor ε can be applied to a vector of arbitrary types $\bar{\tau}$, instead of a vector of distinct type variables $\bar{\alpha}$. It is not difficult to check that the forms (3) and (4) offer equivalent expressiveness. Indeed, a declaration of the form (4) can be written

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

which is an instance of (3). Conversely, since every satisfiable unification constraint admits a most general unifier, a declaration of the form (3) either has no instance at all (making K inapplicable, a pathological case) or can be written under the form (4). We omit the details.

Appearances of this concept in the literature. Declarations of the form (4) are strongly reminiscent of the way *inductive types* are declared in the Calculus of Inductive Constructions [Paulin-Mohring 1992; Werner 1994]. The main difference lies in the fact that inductive types come with a positivity restriction, which ensures logical consistency, whereas guarded algebraic data types carry no such restriction. In a programming-language setting, logical consistency is not a concern: programs that do not terminate are considered acceptable.

Hanus [1988; 1989] designed a typed logic programming language where "functions" (data constructors, in our terminology) are introduced by declarations of the form (4). (In fact, in Hanus' system, it is even possible for the result type to be a type variable; it does not have to be an application of a data constructor ε.) This

yields expressiveness comparable to, and even in excess of, that of guarded algebraic data types. For instance, Hanus is able to typecheck programs that have been defunctionalized in the style of Warren [1982], while Pottier and Gauthier [2004] exploit guarded algebraic data types for the same purpose. However, Hanus' type system does not deal with pattern matching in a special way: that is, it does not exploit the fact that a successful match provides extra *static* type information. Hanus compensates for this weakness by performing *dynamic* type tests and backtracking when they fail. A programming language equipped with true guarded algebraic data types, on the other hand, statically checks that programs are safe, and requires no dynamic type tests. As a result, Hanus' approach is more expressive, but offers fewer static guarantees. It is also more costly at run time, although Hanus describes an optimization for types that adhere to a certain form.

One particular guarded algebraic data type, known as R, is exploited by Crary, Weirich and Morrisett [2002] to encode intensional type analysis into a programming language equipped with a type-erasure semantics. The examples that follow illustrate this idea.

Types introduced by declarations of the form (4) are referred to as *guarded recursive datatype constructors* by Xi et al. [2003], as *first-class phantom types* by Cheney and Hinze [2002; 2003] and by Hinze [2003], and as *equality qualified types* by Sheard [2004] and by Sheard and Pasalic [2004]. These works present a wealth of applications of guarded algebraic data types in the case of unification constraints.

Examples. We assume that type constructors for integers and binary pairs, *int* and $\times$, are available. (They too can be viewed as algebraic data type constructors.) Following Crary et al. [2002], we introduce a unary algebraic data type constructor ty, and declare the following data constructors for it:

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

These declarations are of the form (3), which we use in our theoretical development. In concrete syntax, one could (and should) allow programmers to use the form (4), that is, to write:

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

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: *$ty(\tau)$ can be interpreted 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 $\mathit{Pair}\, v_1\, v_2$, 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 gains 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 are 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
```

(The syntax of our examples mimics that of Objective Caml. We assume that a few standard library functions, such as `print_int` and `print_string`, are available.) 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` can 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` is in fact 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 $ty(\beta_i) \to \beta_i \to unit$, 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, it is natural to expect the type scheme $\forall\alpha.ty(\alpha) \to \alpha \to unit$ to be explicitly supplied by the programmer.

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 conversion* operator that relies on a *dynamic* comparison between type representations:

```
let rec convert : ∀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) -> (convert t1 u1 x1, convert t2 u2 x2)
  | _, _ ->
      raise ConvertException
```

If the type representations `t` and `u` match, then `convert t u x` returns a copy of the value `x`, otherwise it raises the exception `ConvertException`. Because it

copies its argument, `convert` is not quite a *cast* operator. More elaborate and more satisfactory versions of this idea have been studied by Weirich [2000] and by Cheney and Hinze [2002].

1.3 Applications with arithmetic constraints

We now 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 that stand for types and variables that stand for integers.

One can then declare the type of lists, *list*, as a binary algebraic data type constructor, whose parameters are respectively of type and integer kinds:

$$\begin{aligned} \mathit{Nil} &:: \forall\alpha\gamma[\gamma = 0].\mathit{list}(\alpha,\gamma) \\ \mathit{Cons} &:: \forall\alpha\gamma_1\gamma_2[\gamma_1 \geq 0 \wedge 1 + \gamma_1 = \gamma_2].\alpha \times \mathit{list}(\alpha,\gamma_1) \to \mathit{list}(\alpha,\gamma_2) \end{aligned}$$

The idea is that, while the parameter α is the type of the elements of the list, the parameter γ 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* [1997; 1998] and 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.

Examples. In the following examples, the data constructors *Nil* and *Cons* are written `[]` and `::` (infix). Our first example is `combine`, a function that transforms a pair of lists of matching length into a list of pairs:

```
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"
```

This clause is executed when `combine` is applied to lists of differing lengths. Here, it is unnecessary, because the type scheme ascribed to `combine` guarantees that `combine` is applied only to lists of matching length. If we added this clause anyway, it would be typechecked under the *inconsistent* assumption $\gamma = 0 \wedge \gamma_1 \geq 0 \wedge \gamma = 1 + \gamma_1$. That is, the type system would be able to prove that this clause is dead code. This explains why it can be omitted without compromising safety and without causing any compiler warning to be emitted. More details are given in §3.7.

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.

```
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. The auxiliary function `rmap_f` must be explicitly annotated because it involves polymorphic recursion.

In practice, the length of a list is often unknown or only partially known, so a possibly bounded *existential* type is required. For instance, the type scheme of a `filter` function over lists might be:

$$\forall\alpha\gamma_1.(\alpha \to \mathit{bool}) \to \mathit{list}(\alpha,\gamma_1) \to \exists\gamma_2[0 \leq \gamma_2 \leq \gamma_1].\mathit{list}(\alpha,\gamma_2)$$

Xi [1998] has studied partial type inference for existential types, so as to allow existential quantifiers to be implicitly introduced and eliminated. We do not address this issue: HMG(X) only has explicit (Läufer-Odersky-style) bounded existential types.

1.4 Applications with subtyping constraints

Guarded algebraic data types subsume the *bounded existential types* studied by the first author [Simonet 2003] 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. Guarded algebraic data types allow defining a singleton type for *dynamic security levels*, that is, runtime representations of security levels. Examining dynamic security levels at runtime allows recovering ordering constraints between static level variables—just as, in §1.2, examining runtime type representations allowed recovering equations between static type variables. Some more details appear in a technical report [Simonet and Pottier 2005, Section 2.3]. A similar approach is followed by Tse and Zdancewic [2004], who introduce *dynamic principals* into an information flow analysis.

1.5 Related work

1.5.1 *Related type systems.* Although we choose HM(X) [Odersky et al. 1999] as the foundation for our work, because of its elegance, it is by no means the only constraint-based type system in existence. Many others have been defined and studied [Mitchell 1984; Curtis 1990; Aiken and Wimmers 1993; Jones 1994; Smith 1994; Trifonov and Smith 1996; Pottier and Rémy 2005], of which it would be difficult to give a comprehensive list. One could in principle add guarded algebraic data types to any of them.

The usefulness of guarded algebraic data types is twofold. On the one hand, increasing the expressiveness of the type system means *accepting* more programs. On the other hand, it also allows writing more precise type specifications, thus *rejecting* some (safe, but incorrect) programs, in the spirit of *refinement types* [Freeman and Pfenning 1991].

One point should perhaps be stressed. Guarded algebraic data types yield 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. Refinement types *refine* ML, in the sense that every program that is well-typed in their presence is also a well-typed ML program. These properties are of course mutually exclusive. The type system presented in this paper subsumes indexed types in the style of Zenger [1997; 1998] and dependent types in the style of Xi and Pfenning [Xi 1998; Xi and Pfenning 1999], two type systems that refine ML. Nevertheless, it does not refine ML—on the contrary, it extends ML. This point is further clarified in the next paragraphs, which relate our work to Zenger's.

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]. One tells whether the constraint is satisfied, as usual, while the other 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, the satisfaction of C_1 must imply that of C_2, as usual, and *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 $C \Rightarrow \alpha_1 = \mathit{list}(\alpha_2, \gamma)$ is equivalent to $\exists \alpha_2' \gamma' . (\alpha_1 = \mathit{list}(\alpha_2', \gamma') \wedge (C \Rightarrow (\alpha_2 = \alpha_2' \wedge \gamma = \gamma')))$. In other words, this constraint requires α_1 to be a list, regardless of whether C is satisfied. Zenger's interpretation of implication is chosen so that the structure of types can be first discovered using unification, as in ML, and so that, thereafter, there only remains to solve a constraint over indices.

In the present paper, we cannot follow Zenger's approach: we must interpret implication in a standard way. Indeed, in our case, there is no distinction between indices and types. In the particular case of unification constraints (§1.2), for instance, there are no indices at all: we are dealing with equations between standard ML types. As a result, we cannot, like Zenger, distinguish *satisfiability* and *structural consistency*. Instead, we must interpret implication in terms of satisfiability alone. Therefore, we adopt the *standard* interpretation of implication. Because this interpretation is more liberal than Zenger's, more programs become well-typed. This explains why our type system does not refine ML, but extends it. Every program that is well-typed in Zenger's system is well-typed in ours, with the same type and possibly also with additional types.

Some other differences between Zenger's work and ours is that Zenger does not consider subtyping or nested patterns, while we do. 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 we believe was developed independently of Zenger's type system, is a close cousin of it. It maintains 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 exis-

tential 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. Like Zenger's, Xi's elaboration algorithm does, in the end, produce a constraint, whose satisfiability must then be determined.

In contrast with DML(C), 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). 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. Furthermore, as noted by Odersky and Läufer [1996, p. 62], this encoding is not *local*, making it difficult to use in modular programs.

First-class *universal* types with explicit introduction and elimination forms could be added to HMG(X) in the same manner as existential types; see, for instance, Rémy [1994], Jones [1995], Odersky and Läufer [1996], or Simonet [2003].

Several previous accounts of guarded algebraic data types only describe *type checking*, as opposed to *type inference*. This includes Xi et al.'s [2003], Cheney and Hinze'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 constraint domain, where constraint satisfiability and entailment must be decidable. These are also features of the type system presented in this paper.

1.5.2 *Towards type inference.* Some authors have attempted to fully address the pragmatic issue of introducing guarded algebraic data types into Haskell or ML without abandoning type inference. These include the present authors, in a previous version of this paper [Simonet and Pottier 2005], Peyton Jones, Washburn, and Weirich [2004], Peyton Jones, Vytiniotis, Weirich, and Washburn [2005], Pottier and Régis-Gianas [2006], and Stuckey and Sulzmann [2005]. Sheard's interpreter for the Ωmega programming language [Sheard 2005] also seems to perform type inference in the presence of guarded algebraic data types, but its algorithm is undocumented.

Simonet and Pottier [2005, Section 6] focus on the case of unification constraints and require every function that performs case analysis over a guarded algebraic data type to be explicitly annotated with a closed type scheme. Under these assumptions, they show that it is possible to generate so-called *tractable* constraints, which can be efficiently solved and admit most general unifiers. Unfortunately, the details of their constraint generation process are somewhat involved. Furthermore, requiring type annotations to be closed leads to a loss of expressive power that could be problematic in practice. `rmap_f` (§1.3) is an instance of a function that performs case analysis over a guarded algebraic data type and whose type scheme is *not* closed.

Peyton Jones et al. [2004] describe a working system that has been implemented in the Glasgow Haskell compiler. Although not stated in terms of constraints,

their approach appears to rest on the basic insight that a constraint of the form $C_1 \Rightarrow C_2$ can safely be simplified to $\phi(C_2)$, where ϕ is a most general unifier of C_1. Performing this simplification on the fly allows producing constraints that do not involve implications and are thus standard unification constraints. Unfortunately, this simplification step is not meaning-preserving: $C_1 \Rightarrow C_2$ is not in general equivalent to $\phi(C_2)$. In fact, different choices of ϕ can produce different residuals: for instance, $\alpha = \beta \Rightarrow \alpha = \mathit{int}$ can be simplified to $\alpha = \mathit{int}$ or to $\beta = \mathit{int}$. Furthermore, it does not commute with other constraint solving steps inside C_2: for instance, $\beta = \alpha \wedge (\alpha = \mathit{int} \Rightarrow \beta = \mathit{int})$ can be either directly simplified to $\beta = \alpha \wedge \beta = \mathit{int}$ or first rewritten to $\beta = \alpha \wedge (\alpha = \mathit{int} \Rightarrow \alpha = \mathit{int})$ which then simplifies to $\beta = \alpha \wedge \mathit{int} = \mathit{int}$, that is, $\beta = \alpha$. For these reasons, one must be careful to exploit this simplification rule only in a principled and predictable way. To this end, Peyton Jones et al. distinguish *wobbly* types, which are inferred and should not participate in simplification steps, because more might become known about them in the future, and *rigid* types—our terminology—which are explicitly provided by the programmer, can be relied upon, and can thus participate in simplification steps. Although these intuitions are good, Peyton Jones et al.'s formalization is complex and lacks a formal explanation for the need to distinguish wobbly and rigid types. Peyton Jones et al. [2005] describe a simplified version of the wobbly types system, which accepts slightly fewer programs, but is easier to implement and explain.

Inspired by Peyton Jones et al.'s original paper [2004], Pottier and Régis-Gianas develop a so-called *stratified* type inference system. The system combines an initial program transformation phase, known as *local shape inference*, with a traditional type inference phase, in the style of Hindley and Milner. Local shape inference gathers type information that is "evident" in the source program (such as that found in user-provided type annotations), propagates this information, and inserts new type annotations into the program, so that more information becomes "evident" in the transformed program. The transformed program is then submitted to a type inference algorithm that decides whether it is well-typed in a conservative extension of ML with guarded algebraic data types, known as MLGX. In short, MLGX marries *type inference* for ML with *type checking* for guarded algebraic data types: it understands guarded algebraic data types only where enough information has been made explicit, either by the programmer or during local shape inference. MLGX could be viewed as an instance of HMG(X) with unification constraints (as in §1.2), equipped with extra syntactic restrictions that guarantee that the implication connective never appears within constraints. As a result, type inference in MLGX is purely unification-based. It can be presented in terms of constraint generation and constraint solving, where constraints are made up of equations, conjunctions, and existential and universal quantifiers: implication is not required. The strength of stratified type inference is to separate a local shape inference algorithm, which possibly makes many *ad hoc* decisions about how type information should be propagated, and a clean, constraint-based type inference algorithm, which can be proven correct and complete with respect to the specification of MLGX. Several local shape inference algorithms, varying in precision and sophistication, can be used. One of them emulates wobbly types quite faithfully.

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

Fig. 1: Syntax of the calculus

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

Fig. 2: The variables defined by a pattern

Stuckey and Sulzmann's approach [2005] is explicitly constraint-based: their typing and constraint generation rules are close to those of HMG(X). Like Peyton Jones et al. [2004], they rely on rewriting $C_1 \Rightarrow C_2$ to $\phi(C_2)$, where ϕ is a most general unifier of C_1, in order to eliminate implications. Thus, their constraint solving technique is sound, but incomplete. Unlike Peyton Jones et al., they set up no machinery to tame the undesirable behaviors described above. Instead, they note that their constraint solver finds better solutions when more type annotations are available, and suggest a technique for pointing out where more type annotations might be needed. They also suggest another, more ambitious, constraint solving technique. Both of these techniques involve disjunctions, which means that they can produce multiple, incomparable answers, and that their cost may be high.

2. THE UNTYPED CALCULUS

We now introduce a call-by-value λ-calculus featuring data constructors and deep pattern matching.

We believe that the type system presented in §3 is also sound with respect to a call-by-name semantics. However, in order to keep things simple, we have not attempted to prove type soundness with respect to a semantics that captures both call-by-value and call-by-name.

We include nested patterns in the language, even though tests against deep patterns can be compiled down to cascades of tests against shallow patterns. Indeed, this compilation scheme is not unique: compilation can be performed left-to-right, right-to-left, or yet in other ways. We do not wish the typing rules for deep patterns to reflect a particular compilation scheme, because we do not wish to commit to one such scheme. The type system presented in §3 is *independent* of the way deep patterns are compiled: it *cannot* be viewed as the composition of some compilation scheme for deep patterns with some type system for shallow patterns. We come back to this issue at the end of §3.6.

2.1 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*. The syntax of *patterns*, *expressions*, *clauses*, and *values* is given in Figure 1. Patterns include the empty pattern 0, the wildcard pattern 1, variables, conjunction

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

Fig. 3: Extended substitution

and disjunction patterns, and data constructor applications. The empty pattern does not normally appear in source programs: it is used when *normalizing* patterns (§2.3). To a pattern p, we associate a set of *defined program variables* $\mathrm{dpv}(p)$, as specified in Figure 2. (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*. This construct is reminiscent of Peyton Jones' "fat bar" operator [1987]. A clause c is made up of a pattern p and an expression e and is written $p.e$; the variables in $\mathrm{dpv}(p)$ are bound within e. We occasionally use ce to stand for a clause or an expression. Values include functions and applications of a data constructor to values. Within patterns, expressions, and values, all applications of a data constructor must respect its arity: data constructors cannot be partially applied.

2.2 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 $[x \mapsto v]$, which justifies our abuse of notation. Extended substitution for other pattern forms is defined in Figure 3. 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 $\rightarrow$, is defined by the rules of Figure 4. It is standard. Rule (β) governs function application and pattern-matching:

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

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

Fig. 4: Operational semantics

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

Fig. 5: Normalizing patterns

$\lambda(p_1.e_1 \cdots p_n.e_n)\, v$ 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.

2.3 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 5, applied modulo associativity and commutativity of $\wedge$, modulo associativity of $\vee$, and under arbitrary contexts. (Note that $\vee$ cannot be considered commutative, since its semantics has left bias.) This process is terminating and meaning-preserving:

Proof on page 38

LEMMA 2.1. *The relation $\rightsquigarrow$ is strongly normalizing.* ◇

Proof on page 38

LEMMA 2.2. *$p_1 \rightsquigarrow p_2$ implies $p_1 \equiv p_2$.* ◇

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

Proof on page 38

LEMMA 2.3. *$p \equiv 0$ holds if and only if $p \rightsquigarrow^\star 0$ holds.* ◇

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 pattern normalization and typechecking is discussed in §3.7.

3. THE TYPE SYSTEM

We now equip our core calculus with a constraint-based type system featuring guarded algebraic data types. It is a conservative extension of HM(X) [Odersky et al. 1999], which we refer to as HMG(X).

As argued, for instance, by Pottier and Rémy [2005], explaining type inference in terms of constraints is beneficial for at least two reasons. First, constraints provide *a formal language for reasoning and computing about a program's type-correctness*. A logical interpretation gives meaning to sentences in this language. As a result, it is pleasant to reduce type inference to constraint solving: indeed, this allows stating *what* problem must be solved, without making an early commitment as to *how* it should be solved. This holds even when working within the simple setting of ML, where constraints are standard unification constraints. Second, working with constraints allows moving to *much more general settings*, where constraints may involve type class membership predicates, as in Haskell [Wadler and Blott 1989; Jones 1994], Presburger arithmetic, as in DML [Xi 1998], polynomials, as in Zenger's work [1997], subtyping, etc. Of course, in each case, one must write a different constraint solver, which can be hard work; but, at least, the theoretical framework and the type soundness proofs are shared.

In §3.1, 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 §3.2, 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 §3.3. 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 §3.3.

In §3.4, we introduce a few tools that allow manipulating environment fragments. Then, we review the typing judgments (§3.5) and typing rules (§3.6) of HMG(X), and establish type soundness (§3.7).

3.1 Syntax

In keeping with the HM(X) tradition, the type system is parameterized by a first-order 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 can 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). 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.

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

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

The set of basic predicates π is left unspecified, which allows the system to be instantiated in many ways. Every predicate is assumed to have a fixed arity. For instance, in an application to indexed types in the style of §1.3, equality with an integer constant would be a predicate of arity 1; the usual ordering of integers would be a predicate of arity 2; the usual binary operations over integers would be predicates of arity 3. 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. By convention, $\Rightarrow$ binds tighter than $\exists$ and $\forall$, which bind tighter than $\wedge$ and $\vee$.

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 $\bar{\alpha}$:

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

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 $\bullet$ for the empty environment. 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 (§3.5) and are meant to describe the static knowledge that is gained by successfully matching a value against a pattern.

3.2 Interpretation

The logic is interpreted in a fixed *model* $(T, \leq)$. T is a nonempty set whose elements t are referred to as *ground types*. It is equipped with a partial pre-order $\leq$, which is used to interpret the basic predicate $\leq$. Keep in mind that this pre-order 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 pre-order 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 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.

In order to guarantee type soundness, the model must satisfy a number of requirements, which we state in §3.3. Before doing so, however, we interpret type schemes and environment fragments, and explain how this gives rise to partial pre-orders on these objects.

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

Definition 3.1. 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 $\{t \mid \exists\bar{t} \quad \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D \wedge \rho[\bar{\alpha} \mapsto \bar{t}](\tau) \leq t\}$. It can also be written $\uparrow\{\rho[\bar{\alpha} \mapsto \bar{t}](\tau) \mid \rho[\bar{\alpha} \mapsto \bar{t}] \vdash D\}$, where $\uparrow$ is the upward closure operator. ◇

This definition gives rise to a partial pre-order 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 can verify that the type scheme $\forall\alpha.\alpha$ is a least element in this pre-order: 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 can 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 pre-order allows a universally quantified type variable to be instantiated (to become free).

The following property is used in a couple of proofs:

LEMMA 3.2. $D \Vdash (\forall\bar{\alpha}[D].\tau) \leq \tau$. ◇ Proof on page 39

Ordering constraints on type schemes can also be expressed in terms of ordering constraints on types. This is stated as follows:

LEMMA 3.3. *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.* ◇ Proof on page 39

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 (§3.6).

The pre-order on type schemes can 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 pre-order 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 3.4. 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 $\{g \mid \exists\bar{t} \quad \rho[\bar{\beta} \mapsto \bar{t}] \vdash D \wedge g \leq \rho[\bar{\beta} \mapsto \bar{t}](\Gamma)\}$. It can also be written $\downarrow\{\rho[\bar{\beta} \mapsto \bar{t}](\Gamma) \mid \rho[\bar{\beta} \mapsto \bar{t}] \vdash D\}$, where $\downarrow$ is the downward closure operator. ◇

Again, this definition gives rise to a partial pre-order 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 can 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 6) 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 pre-order allows a free type variable to become abstract (existentially quantified).

The interpretation of environment fragments, and the definition of their pre-order, 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 can be be expressed in terms of ordering constraints on types. This is stated by the following lemma:

Proof on page 39

LEMMA 3.5. *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 $C \wedge D' \Vdash \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$.* ◇

3.3 Requirements on the model

So far, the pre-order $\leq$ 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 3.28 and Theorem 3.30).

Requirement 3.6. Every constraint of the form $\tau_1 \rightarrow \tau_2 \leq \varepsilon(\bar{\tau})$ or $\varepsilon(\bar{\tau}) \leq \tau_1 \rightarrow \tau_2$ or $\varepsilon(\bar{\tau}) \leq \varepsilon'(\bar{\tau}')$, where ε and ε' are distinct, is unsatisfiable. ◇

Next, the arrow type constructor must be contravariant in its domain and covariant in its codomain. This is required for subject reduction to hold (Lemma 3.23). This requirement appears in all type systems equipped with subtyping.

Requirement 3.7. $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ 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, how-

ever; 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:

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

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 now state the variance requirements that bear on algebraic data type constructors. They are necessary to establish subject reduction and progress (Lemmas 3.26 and 3.28), and are standard in type systems featuring both subtyping and isorecursive types: see, for instance, Pottier and Rémy [2005] or Simonet [2003].

Requirement 3.8. For every data constructor K, if $K :: \forall \bar{\alpha} \bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ and $K :: \forall \bar{\alpha}' \bar{\beta}'[D'].\tau'_1 \times \cdots \times \tau'_n \to \varepsilon(\bar{\alpha}')$ 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 $\forall \bar{\alpha} \bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ amounts to declaring that the abstract data type $\varepsilon(\bar{\alpha})$ is *isomorphic* to a sum type of the form $(\exists \bar{\beta}[D]\, \tau_1 \times \cdots \times \tau_n) + \ldots$ A similar statement can be made about the α-equivalent declaration $K :: \forall \bar{\alpha}' \bar{\beta}'[D'].\tau'_1 \times \cdots \times \tau'_n \to \varepsilon(\bar{\alpha}')$. 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 $\exists \bar{\beta}'[D']\, \tau'_1 \times \cdots \times \tau'_n + \ldots$ and $\exists \bar{\beta}[D]\, \tau_1 \times \cdots \times \tau_n + \ldots$ In other words, since the sum type constructor $+$ is covariant, the law

$$\varepsilon(\bar{\alpha}') \leq \varepsilon(\bar{\alpha}) \Vdash \exists \bar{\beta}'[D']\, \tau'_1 \times \cdots \times \tau'_n \leq \exists \bar{\beta}[D]\, \tau_1 \times \cdots \times \tau_n$$

must hold. In fact, Requirement 3.8 could conceivably be stated in this manner. Under the hypothesis that $\bar{\beta}$ is fresh for every τ'_i, one proves, using Lemma 3.5, 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 above statement.

We have stated several requirements about the model, but have not explained how to construct a model. This is straightforward. Ground types are usually defined as finite trees, that is, types without variables. If subtyping is interpreted as equality, then Requirements 3.6–3.8 are trivially satisfied. If subtyping is interpreted as a nontrivial pre-order, one must check that its definition satisfies all three requirements.

3.4 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 3.9. If Δ is $\exists\bar{\beta}[D]\Gamma$ and $\bar{\beta} \# \text{ftv}(\bar{\alpha}, C)$ 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 3.15 and Figure 6).

Proof on page 39

LEMMA 3.10. $\rho(\exists\bar{\alpha}[C]\Delta) = \cup\{\rho[\bar{\alpha} \mapsto \bar{t}](\Delta) \mid \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\}$. ◇

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 next two lemmas, are simple.

Definition 3.11. 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 that, for every $i \in \{1, 2\}$, Δ_i is $\exists\bar{\beta}_i[D_i]\Gamma_i$, and provided that $\bar{\beta}_1 \# \bar{\beta}_2$, $\bar{\beta}_1 \# \text{ftv}(\Delta_2)$, and $\bar{\beta}_2 \# \text{ftv}(\Delta_1)$ hold. ◇

Definition 3.12. 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 $\bar{\beta}_1 \# \bar{\beta}_2$, $\bar{\beta}_1 \# \text{ftv}(\Delta_2)$, and $\bar{\beta}_2 \# \text{ftv}(\Delta_1)$ 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 $\bar{\alpha} \# \text{ftv}(\bar{\beta}_1, \bar{\beta}_2, \Delta_1, \Delta_2)$ holds. ◇

The next two lemmas are also low-level results, used only in the proof of the laws in Figure 6. 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 $\text{dom}(g_1) \cup \text{dom}(g_2)$ that maps x to $g_i(x)$ if $x \in \text{dom}(g_i)$ 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 $\{g_1 \times g_2 \mid g_1 \in G_1 \wedge g_2 \in G_2\}$.

$$
\begin{array}{rcll}
\mathsf{true} & \Vdash & \Delta \leq \exists \bar{\alpha}.\Delta & \text{(F-Hide)} \\
C_1 \Rightarrow C_2 & \Vdash & [C_1]\Delta \leq [C_2]\Delta & \text{(F-Imply)} \\
C \Rightarrow \Delta_1 \leq \Delta_2 & \Vdash & [C]\Delta_1 \leq [C]\Delta_2 & \text{(F-Enrich)} \\
\forall \bar{\alpha}.(\Delta_1 \leq \Delta_2) & \Vdash & \exists \bar{\alpha}.\Delta_1 \leq \exists \bar{\alpha}.\Delta_2 & \text{(F-Ex)} \\
\Delta_1 \leq \Delta_2 & \Vdash & \Delta \times \Delta_1 \leq \Delta \times \Delta_2 & \text{(F-And)} \\
\Delta_1 \leq \Delta_2 & \Vdash & \Delta + \Delta_1 \leq \Delta + \Delta_2 & \text{(F-Or)} \\
\Delta_1 \leq \Delta \wedge \Delta_2 \leq \Delta & \Vdash & \Delta_1 + \Delta_2 \leq \Delta & \text{(F-Glb)} \\
\mathsf{true} & \Vdash & \Delta_1 \leq \Delta_1 + \Delta_2 & \text{(F-Lub)}
\end{array}
$$

Fig. 6: Some properties of subsumption between environment fragments

Lemma 3.13. $\rho(\Delta_1 \times \Delta_2) = \rho(\Delta_1) \times \rho(\Delta_2)$. ◇ Proof on page 39

Lemma 3.14. $\rho(\Delta_1 + \Delta_2) = \rho(\Delta_1) \cup \rho(\Delta_2)$. ◇ Proof on page 40

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 (§4).

Lemma 3.15. *The entailment laws in Figure 6 are valid.* ◇ Proof on page 40

3.5 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 can be read: *under assumption C, it is legal to match a value of type τ against p; furthermore, if successful, this test guarantees that there exist types $\bar{\beta}$ that satisfy D such that Γ is a valid description of the values that the variables in* $\mathrm{dpv}(p)$ *receive.*

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 $K :: \forall \bar{\alpha}.\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$, 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 $K :: \forall \bar{\alpha}\bar{\beta}.\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$. 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 $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ holds, then we have $\mathsf{true} \vdash K\, x_1 \cdots x_n : \varepsilon(\bar{\alpha}) \rightsquigarrow \exists \bar{\beta}[D](x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n)$.

Patterns (syntax-directed)

$$\textsc{p-Empty}\quad C \vdash 0 : \tau \rightsquigarrow \exists\varnothing[\mathsf{false}]\bullet$$

$$\textsc{p-Wild}\quad C \vdash 1 : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}]\bullet$$

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

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

$$\textsc{p-Or}\quad \frac{\forall i \quad C \vdash p_i : \tau \rightsquigarrow \Delta}{C \vdash p_1 \vee p_2 : \tau \rightsquigarrow \Delta}$$

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

Patterns (non-syntax-directed)

$$\textsc{p-EqIn}\quad \frac{C \vdash p : \tau' \rightsquigarrow \Delta \qquad C \Vdash \tau = \tau'}{C \vdash p : \tau \rightsquigarrow \Delta}$$

$$\textsc{p-SubOut}\quad \frac{C \vdash p : \tau \rightsquigarrow \Delta' \qquad C \Vdash \Delta' \leq \Delta}{C \vdash p : \tau \rightsquigarrow \Delta}$$

$$\textsc{p-Hide}\quad \frac{C \vdash p : \tau \rightsquigarrow \Delta \qquad \bar{\alpha} \# \mathrm{ftv}(\tau, \Delta)}{\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta}$$

Expressions (syntax-directed)

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma}$$

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

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

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

$$\textsc{Fix}\quad \frac{C, \Gamma[x \mapsto \sigma] \vdash v : \sigma}{C, \Gamma \vdash \mu x.v : \sigma}$$

$$\textsc{Let}\quad \frac{C, \Gamma \vdash e_1 : \sigma' \qquad C, \Gamma[x \mapsto \sigma'] \vdash e_2 : \sigma}{C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \sigma}$$

Expressions (non-syntax-directed)

$$\textsc{Gen}\quad \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \# \mathrm{ftv}(\Gamma, C)}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}$$

$$\textsc{Inst}\quad \frac{C, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash e : \tau}$$

$$\textsc{Sub}\quad \frac{C, \Gamma \vdash e : \tau' \qquad C \Vdash \tau' \leq \tau}{C, \Gamma \vdash e : \tau}$$

$$\textsc{Hide}\quad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{\alpha} \# \mathrm{ftv}(\Gamma, \sigma)}{\exists\bar{\alpha}.C, \Gamma \vdash e : \sigma}$$

Clauses

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

Fig. 7: Typing rules

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 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 $C, \bullet \vdash e : \sigma$ holds for some *satisfiable* constraint C.

3.6 Typing rules

Whether a judgment is valid is defined by the rules in Figure 7, which we now review, beginning with the rules that concern patterns.

P-Empty and P-Wild tell that the patterns 0 and 1 can 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 can 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. 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 (§4).

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, 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 wrapping them within the guarded (bounded) existential quantifier $\exists\bar{\beta}[D]$, 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 3.16. The following is a valid derivation:

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

Its conclusion can be read as follows. First, it is valid to match a value of type $ty(\alpha)$ 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. This derivation is referred to as (d_1) in Example 3.17.

Here is another valid derivation:

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

Its conclusion can be read as follows. First, it is valid to match a value of type $ty(\alpha)$ 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 $ty(\beta_1)$ and $ty(\beta_2)$, respectively. This derivation is referred to as (d_2) in Example 3.17. ◇

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]. The premise $C \Vdash \exists\sigma$ in Var is a minor technicality: it allows establishing Lemma 3.20 without requiring a hypothesis on Γ. Cstr typechecks a data constructor application exactly as if it were 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. Abs requires all clauses to have the same (function) type. Fix allows *polymorphic recursion*, an essential feature in programs that involve guarded algebraic data types, as illustrated in §1. 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 3.17. Here is a valid derivation for the first clause in the definition of *print*, the generic printing function defined in §1.2. We assume that the environment Γ assigns type $\mathit{int} \to \mathit{unit}$ to the variable *print_int* and exploit the derivation (d_1)

of Example 3.16. We write e_1 for $\lambda x.\mathit{print\_int}\, x$.

$$\dfrac{(d_1)\qquad \dfrac{\dfrac{\dfrac{\cdots}{\alpha = \mathit{int}, \Gamma \vdash e_1 : \mathit{int} \to \mathit{unit}}}{\alpha = \mathit{int} \Vdash \mathit{int} \to \mathit{unit} \leq \alpha \to \mathit{unit}}\ \textsc{Sub}}{\alpha = \mathit{int}, \Gamma \vdash e_1 : \alpha \to \mathit{unit}}}{\mathsf{true}, \Gamma \vdash \mathit{Int}.e_1 : \mathit{ty}(\alpha) \to \alpha \to \mathit{unit}}\ \textsc{Clause}$$

The assumption $\alpha = \mathit{int}$, 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 $\mathit{Int}.e_1$ has type $\mathit{ty}(\alpha) \to \alpha \to \mathit{unit}$, where α is *unconstrained*: indeed, the hypothesis $\alpha = \mathit{int}$, 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*. We assume that the environment Γ assigns type scheme $\forall\alpha.\mathit{ty}(\alpha) \to \alpha \to \mathit{unit}$ to *print*, so as to be able to typecheck the recursive calls to *print*. We let Γ' stand for the environment $(t_1 \mapsto \mathit{ty}(\beta_1); t_2 \mapsto \mathit{ty}(\beta_2))$. We write e_2 for $\lambda(x_1, x_2).(\mathit{print}\, t_1\, x_1; \ldots; \mathit{print}\, t_2\, x_2)$.

$$\dfrac{(d_2)\qquad \dfrac{\dfrac{\dfrac{\cdots}{\alpha = \beta_1 \times \beta_2, \Gamma\Gamma' \vdash e_2 : \beta_1 \times \beta_2 \to \mathit{unit}}}{\alpha = \beta_1 \times \beta_2 \Vdash \beta_1 \times \beta_2 \to \mathit{unit} \leq \alpha \to \mathit{unit}}\ \textsc{Sub}}{\alpha = \beta_1 \times \beta_2, \Gamma\Gamma' \vdash e_2 : \alpha \to \mathit{unit}}}{\mathsf{true}, \Gamma \vdash \mathit{Pair}\,(t_1, t_2).e_2 : \mathit{ty}(\alpha) \to \alpha \to \mathit{unit}}\ \textsc{Clause}$$

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 $\mathsf{true}, \Gamma_0 \vdash \mu\mathit{print}\ldots. : \forall\alpha.\mathit{ty}(\alpha) \to \alpha \to \mathit{unit}$, where Γ_0 assigns type $\mathit{int} \to \mathit{unit}$ to *print_int* and where the dots stand for the body of *print*'s definition. Thus, the function *print*, as defined in §1.2, is well-typed in (all instances of) HMG(X). ◇

Remark 3.18. 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 $K :: \forall\bar{\alpha}.\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$. Then, in every instance of p-Cstr, $\bar{\beta}$ and D must be $\varnothing$ and true, respectively, so that the rule may be written:

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

Then, one proves that p-SubOut and p-Hide can 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 take the simplified form $C \vdash p : \tau \rightsquigarrow \Gamma$, where Γ is a simple environment. This in turn allows simplifying CLAUSE as follows:

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

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. ◇

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, the following 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 function 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). At first sight, this code seems to make perfect sense.

It is, however, ill-typed in our system, because, *in the absence of any hypotheses about* α, a value of type α cannot be matched against the pair pattern (x_1, x_2). Our type system only accepts the following variant, where it is clear that the runtime type representation must be examined *before* x can be deemed to be a pair:

```
  | (Pair (t1, t2), x) ->
      let (x1, x2) = x in
      ...
```

This design may seem surprising. Instead, we could make P-CSTR more liberal and allow type information to propagate in a left-to-right fashion. The first uncurried version of *print* would then be considered well-typed, since the hypothesis $\alpha = \beta_1 \times \beta_2$ would be available when the pattern (x_1, x_2) is typechecked. Furthermore, adopting such a relaxed version of P-CSTR would not compromise type safety. So, why stick with a stricter rule?

Our reason is as follows. Suppose we adopt the relaxed rule, and view the first uncurried 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? In some call-by-value programming languages, such as Objective Caml [Leroy et al. 2005], the evaluation order of pattern matching is unspecified. This is a good thing, because the compiler is free to schedule tests in whichever order appears most efficient. It would be undesirable for the type system to impose tight constraints on the compilation strategy. In a call-by-name language such as Haskell [Peyton Jones 2003], the evaluation strategy is already specified as left-to-right, so it makes sense to adopt a relaxed version of P-Cstr, as indeed Peyton Jones et al. [2004] do.

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 §2 *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 3.28 in the present paper, is the reason why values need not carry runtime tags that identify their type. Although adopting the second option would preserve type safety, it would violate this property, leading to a less efficient compilation scheme.

One should point out that this problem does not arise if the language does not have *nested* patterns. Indeed, in a language where patterns are *shallow*, the above versions of *print* cannot be written. Instead of a single, complex test, the programmer is forced to write a cascade of simple tests, where the sequencing is explicit. This eliminates the problem. Bringing this problem to light and explicitly addressing it is the reason why we include nested patterns in our calculus.

3.7 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 3.19 (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 40

Lemma 3.20. *$C, \Gamma \vdash e : \sigma$ implies $C \Vdash \exists\sigma$.* ◇ Proof on page 40

Next come three auxiliary normalization lemmas. They are standard: they come unmodified from the theory of HM(X). A *shape* is a term whose nodes are names of typing rules and whose leaves are holes. We say that a shape *can 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.

Lemma 3.21. *The shape* Inst(Gen($\cdot$)) *can be replaced with* Hide(Sub($\cdot$)). ◇ Proof on page 40

Proof on page 41

LEMMA 3.22. *The shape* HIDE(GEN(·)) *can be replaced with* GEN(HIDE(·)). ◇

Proof on page 41

LEMMA 3.23. *The shape* APP(SUB(ABS(·$_1$)), ·$_2$) *can be replaced with the shape* SUB(APP(ABS(·$_1$), SUB(·$_2$))). ◇

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.

Proof on page 41

LEMMA 3.24 (NORMALIZATION). *Every valid typing judgment admits a normal derivation.* ◇

We now prove that HMG(X) is sound, following 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:

Proof on page 41

LEMMA 3.25 (SUBSTITUTION). $C, \Gamma[x \mapsto \sigma'] \vdash ce : \sigma$ *and* $C, \bullet \vdash e : \sigma'$ *imply* $C, \Gamma \vdash [x \mapsto e]ce : \sigma$. ◇

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

Proof on page 42

LEMMA 3.26. *Assume* v *matches* p *and* $C, \bullet \vdash v : \tau$ *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* $x \in \mathrm{dpv}(p)$, $H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$ *holds.* ◇

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 can be specialized as follows: *if* v *matches* p *and* $C, \bullet \vdash v : \tau$ *and* $C \vdash p : \tau \rightsquigarrow \Gamma$ *hold, then, for every* $x \in \mathrm{dpv}(p)$, $C, \bullet \vdash [p \mapsto v]x : \Gamma(x)$ *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 $H \Vdash D$.

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

Proof on page 45

THEOREM 3.27 (SUBJECT REDUCTION). $C, \bullet \vdash e : \sigma$ *and* $e \to e'$ *imply* $C, \bullet \vdash e' : \sigma$. ◇

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

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

Fig. 8: Computing the complement of a pattern

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 can 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 8. 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 §2.3.

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* (§1.2), we have $\mathit{Int} \vee \neg \mathit{Int} = \mathit{Int} \vee \mathit{Pair}\, 1 \cdot 1 \neq 1$. For the same reason, an exhaustiveness condition of the form $\neg p \equiv 0$ is not equivalent to $p \equiv 1$.

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 §3.6.

The hypotheses of the lemma are analogous to those of Lemma 3.26. It is, however, oriented towards proving progress, rather than subject reduction.

LEMMA 3.28. *If $C, \bullet \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \Delta$ hold, where C is satisfiable, then v matches $p \vee \neg p$.* ◇ Proof on page 45

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

LEMMA 3.29. *If $E[e]$ is well-typed, then so is e.* ◇ Proof on page 46

THEOREM 3.30 (PROGRESS). *If e is well-typed and contains exhaustive case analyses only, then it is either reducible or a value.* ◇ Proof on page 47

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 now state a first type soundness result:

Proof on page 47

THEOREM 3.31 (TYPE SOUNDNESS). *If e is well-typed and contains exhaustive case analyses only, then it does not go wrong.* ◇

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

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

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 revisit the type soundness result as follows:

Proof on page 47

THEOREM 3.32 (PROGRESS REVISITED). *If $\lfloor e \rfloor$ is well-typed, then e is either reducible or a value.* ◇

Proof on page 48

THEOREM 3.33 (TYPE SOUNDNESS REVISITED). *If $\lfloor e \rfloor$ is well-typed, then e does not go wrong.* ◇

Let us stress that, according to Theorem 3.33, 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 3.33 fills this gap.

Remark 3.34. 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 5. 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, 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 can check that e and $\lfloor e \rfloor$ admit the same typings. ◇

4. 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.

Thus, the language of expressions becomes:

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

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.

In practice, one usually requires type annotations of the form $\mu(x : \sigma).e$, where σ possibly contains free type variables, and one introduces the expression forms $\exists \bar{\beta}.e$ and $\forall \bar{\beta}.e$ to allow programmers to explicitly bind these type variables. This more general treatment is well understood—see, for instance, Peyton Jones and Shields [2004]—and is orthogonal to guarded algebraic data types, so we omit it.

Not requiring σ to be closed is important, for a couple of different reasons. The main reason is that this allows defining $\mu x.e$ as syntactic sugar for $\mu(x : \exists \beta.\beta).e$. In other words, unannotated fixpoints are still part of the language. They carry the uninformative type annotation $\exists \beta.\beta$, which means *some (monomorphic) type*. Note, however, that fixpoints that exploit polymorphic recursion must carry a truly explicit, nontrivial type annotation. The second reason is that some fixpoints do not admit a closed type scheme. This is often the case for recursive functions that are nested inside another, larger function: see, for instance, `rmap_f` in §1.3.

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

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

Patterns (constraint generation)

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

Patterns (environment fragment generation)

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

Fig. 9: Type inference for patterns

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 can be reduced to constraint solving.

4.1 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 9. As usual, the type variables that are bound in the right-hand side of an equation must be chosen 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. 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 can 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 can 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 sub-patterns p_i.

The last two rules can 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 4.1. *Assume $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$. 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 48

Example 4.2. 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 $\exists\alpha'[ty(\alpha') = ty(\alpha) \wedge \alpha' = int]\bullet$, which is equivalent to $\exists\varnothing[\alpha = int]\bullet$. These results are consistent with the first derivation given in Example 3.16.

Here is another example. By Lemma 4.1, 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 3.16. ◇

Lemmas 4.3 and 4.5 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 4.4, which states that, under the assumption that τ and τ' are equal, they are interchangeable for the purposes of constraint generation.

LEMMA 4.3 (CORRECTNESS). $(\!|p \downarrow \tau|\!) \vdash p : \tau \rightsquigarrow (\!|p \uparrow \tau|\!)$. ◇ Proof on page 48

LEMMA 4.4. $\tau = \tau' \wedge (\!|p \downarrow \tau|\!) \Vdash (\!|p \downarrow \tau'|\!)$ *and* $\tau = \tau' \Vdash (\!|p \uparrow \tau|\!) \leq (\!|p \uparrow \tau'|\!)$ *hold.* ◇ Proof on page 48

LEMMA 4.5 (COMPLETENESS). $C \vdash p : \tau \rightsquigarrow \Delta$ *implies* $C \Vdash (\!|p \downarrow \tau|\!)$ *and* $C \Vdash (\!|p \uparrow \tau|\!) \leq \Delta$. ◇ Proof on page 48

4.2 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 10. Again, the type variables that are bound in the

Expressions

$$
\begin{aligned}
(\!|\Gamma \vdash x : \tau|\!) &= \Gamma(x) \leq \tau \\
(\!|\Gamma \vdash \lambda\bar{c} : \tau|\!) &= \exists\alpha_1\alpha_2.((\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau) \\
(\!|\Gamma \vdash e_1\, e_2 : \tau|\!) &= \exists\alpha.((\!|\Gamma \vdash e_1 : \alpha \to \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!)) \\
(\!|\Gamma \vdash K\, e_1 \cdots e_n : \tau|\!) &= \exists\bar{\alpha}\bar{\beta}.(\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau) \\
&\quad \textit{where } K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha}) \\
(\!|\Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \tau|\!) &= \exists\bar{\beta}.((\!|\Gamma[x \mapsto \sigma] \vdash e : \sigma|\!) \wedge \sigma \leq \tau) \\
(\!|\Gamma \vdash e : \forall\bar{\gamma}[C].\tau|\!) &= \forall\bar{\gamma}.C \Rightarrow (\!|\Gamma \vdash e : \tau|\!) \\
(\!|\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau|\!) &= (\!|\Gamma[x \mapsto \forall\alpha[C].\alpha] \vdash e_2 : \tau|\!) \wedge \exists\alpha.C \\
&\quad \textit{where } C \textit{ is } (\!|\Gamma \vdash e_1 : \alpha|\!)
\end{aligned}
$$

Clauses

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

Fig. 10: Type inference for expressions and clauses

right-hand side of an equation must be chosen fresh for the parameters that appear on its left-hand side. Note that τ can be a type variable, so we do "infer types", even though an "expected type" has to be provided in this formulation.

The rules that govern expressions are standard: see, for instance, Sulzmann et al. [1999], Simonet [2003], 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 $\Gamma(x)$.

The second rule, which deals with a λ-abstraction $\lambda\bar{c}$, requires the expected type τ to be (a supertype of) an arrow type $\alpha_1 \to \alpha_2$, and requires every clause to have type $\alpha_1 \to \alpha_2$. We write $(\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!)$ for the conjunction $\wedge_i(\!|\Gamma \vdash c_i : \alpha_1 \to \alpha_2|\!)$ when $\bar{c}$ is $(c_1, \ldots, c_n)$.

The third 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 fourth rule deals with a data constructor application exactly as if it were 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 fifth rule implements polymorphic recursion by ensuring that the body e admits the type scheme σ under the hypothesis that x has type scheme σ. (The sixth rule defines the notation $(\!|\Gamma \vdash e : \sigma|\!)$.) 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. For unannotated fixpoints of the form $\mu x.e$, which were defined to be syntactic sugar for $\mu(x : \exists\beta.\beta).e$, this gives rise to the following derived rule:

$$(\!|\Gamma \vdash \mu x.e : \tau|\!) = \exists\beta.((\!|\Gamma[x \mapsto \beta] \vdash e : \beta|\!) \wedge \beta \leq \tau)$$

This is a standard rule for monomorphic fixpoints.

The seventh 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.

We now turn to the last rule, which deals with clauses. This is where 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 $(\!|p \uparrow \tau_1|\!)$.

Example 4.6. 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 3.17, we assume that the environment Γ assigns type $int \to unit$ to the variable *print_int*. We implicitly exploit the results of Example 4.2. We write e_1 for $\lambda x.print\_int\, x$.

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

It is easy to check that the subconstraint $(\!|\Gamma \vdash e_1 : \alpha \to unit|\!)$ 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 3.17, 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 4.2. We write e_2 for $\lambda(x_1, x_2).(print\, t_1\, x_1; \ldots; print\, t_2\, x_2)$.

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

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 3.17 and at a fresh type variable γ, is the following:

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

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 4.7 (CORRECTNESS). $(\!|\Gamma \vdash ce : \tau|\!), \Gamma \vdash ce : \tau$. ◇ Proof on page 49

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 4.8. $(\!|\Gamma \vdash ce : \tau|\!) \wedge \tau \leq \tau' \Vdash (\!|\Gamma \vdash ce : \tau'|\!)$. ◇ Proof on page 50

Proof on page 50

LEMMA 4.9. $\Gamma' \leq \Gamma \wedge (\!|\Gamma \vdash ce : \tau|\!) \Vdash (\!|\Gamma' \vdash ce : \tau|\!)$. ◇

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.

Proof on page 50

LEMMA 4.10. *Assume* $\bar{\beta}_1\bar{\beta}_2 \# \mathrm{ftv}(\Gamma, \tau)$. *We have* $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \forall\bar{\beta}_1.D_1 \Rightarrow (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$. ◇

These lemmas allow establishing completeness:

Proof on page 50

THEOREM 4.11 (COMPLETENESS). $C, \Gamma \vdash ce : \forall\bar{\alpha}[D].\tau$ *and* $\bar{\alpha} \# \mathrm{ftv}(\Gamma)$ *imply* $C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash ce : \tau|\!)$. ◇

Using Theorems 4.7 and 4.11, as well as Lemma 3.20, 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.

It is worth noting that the type inference rules are much simpler and clearer than the typing rules of §3. This was true in HM(X) already, and is even more so in HMG(X). In fact, one of the anonymous referees found this state of affairs to be "quite frustrating." One might be tempted to abandon the typing rules altogether and to adopt the type inference rules as the sole definition of the type system. However, defining both sets of rules, and proving that they are equivalent, is a useful "sanity check," which helps ensure that the type system is defined in a sensible way. Also, some proofs, such as type soundness, are probably more easily carried out in terms of the typing rules of §3 rather than in terms of the constraint production rules.

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 that no basic predicates other than equality are available, and assuming that types receive their standard interpretation as elements of a free algebra of (finite or infinite) terms, constraint solving is decidable [Maher 1988; Comon and Lescanne 1989]. More generally, assuming that no basic predicates other than subtyping are available, and assuming that subtyping receives a *structural* interpretation, constraint solving remains decidable [Kuncak and Rinard 2003].

Unfortunately, these decidability results are only of theoretical interest, because the complexity of constraint solving for the first-order theory of term equality is nonelementary [Vorobyov 1996]. In other words, even in the simplest case above, where no basic predicates other than equality are available, the constraints that we produce belong to a class whose satisfiability problem is intractable. Of course, this lower complexity bound also applies to cases where more predicates, such as structural subtyping, are introduced.

This fact may come as a surprise. Indeed, for simple program fragments such as *print*, the constraints that we produce are so simple as to appear "obviously" satisfiable to a human observer. Is the problem really so difficult in the general case? Do the constraints that we produce really exploit the full expressive power of the first-order theory of term equality?

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. Of course, it is reasonable 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 use of implication. If uncontrolled, implication allows encoding negation, since $\neg D$ is $D \Rightarrow \mathsf{false}$. Taming implication is an issue that we leave open in the present paper. As mentioned in §1.5, several potential answers have been suggested in the particular case of equality constraints. In a technical report [Simonet and Pottier 2005], we suggest exploiting only *rigid implication*, of the form $\forall\bar{\beta}.C_1 \Rightarrow C_2$, where $\mathrm{ftv}(C_1) \subseteq \bar{\beta}$. Stuckey and Sulzmann [2005] suggest replacing $C_1 \Rightarrow C_2$ with $\phi(C_2)$, where ϕ is C_1's most general unifier—a correct but incomplete simplification step. Peyton Jones et al. [2004; 2005] and Pottier and Régis-Gianas [2006] adopt approaches that can be understood as the combination of an incomplete (but hopefully predictable) local inference algorithm with a complete, unification-based type inference algorithm. We believe that more experience is needed before a definitive answer emerges.

5. CONCLUSION

We have extended the constraint-based type system HM(X) with deep pattern matching, polymorphic recursion, and guarded algebraic data types, in an attempt to study, in a general setting, the interaction between guarded algebraic data types and type inference in the style of Hindley and Milner. We have proved that HMG(X) is sound and that, provided recursive definitions carry a type annotation, type inference can be reduced to constraint solving. This provides a solid theoretical foundation for many applications involving equality constraints, arithmetic constraints, subtyping constraints, and so on.

The main issue left unanswered in this paper is how to efficiently solve a constraint that encodes a type inference problem. Our philosophy is that one should not develop an incomplete solver—that is, a solver that sometimes rewrites a constraint to a more restrictive constraint—because this means abandoning the primary benefit of constraints, namely the fact that the *logical meaning* of a constraint, regardless of its syntax, is enough to tell whether a program is type-correct. We believe that one should, instead, strive to produce simpler constraints, whose satisfiability can be efficiently determined by a (correct and complete) solver. Inspired by Peyton Jones et al.'s wobbly types [2004; 2005], recent work by Pottier and Régis-Gianas [2006] proposes one way of doing so, by relying on explicit, user-provided type annotations and on an *ad hoc* local shape inference phase. It would be interesting to know whether it is possible to do better, that is, *not* to rely on an ad hoc preprocessing phase.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous referees for their useful comments and suggestions.

A. PROOFS

Proof of Lemma 2.1. 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 5 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 2.2. By examination of each normalization rule (Figure 5) and by definition of extended substitution (Figure 3). □

Proof of Lemma 2.3. 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 2.1, p has a normal form p'. By Lemma 2.2, $p' \equiv 0$ 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 2.2, $p \rightsquigarrow^\star 0$ implies $p \equiv 0$. □

Proof of Lemma 3.2. Let ρ satisfy D. Then, by Definition 3.1, $\rho(\forall\bar{\alpha}[D].\tau)$ is a superset of $\uparrow\{\rho(\tau)\}$. Thus, ρ satisfies $(\forall\bar{\alpha}[D].\tau) \leq \tau$. □

Proof of Lemma 3.3. Left to the reader. The proof of Lemma 3.5, which follows, is dual. □

Proof of Lemma 3.5. We have

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

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 can 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 3.10. Assume Δ is $\exists\bar{\beta}[D]\Gamma$ (**1**), where $\bar{\beta} \# \mathrm{ftv}(\bar{\alpha}, C)$ (**2**). We have

$$
\begin{array}{ll}
 & \rho(\exists\bar{\alpha}[C]\Delta) \\
= & \rho(\exists\bar{\alpha}\bar{\beta}[C \wedge D]\Gamma) \\
 & \text{by (1), (2), and Definition 3.9} \\
= & \downarrow\{\rho[\bar{\alpha} \mapsto \bar{t}, \bar{\beta} \mapsto \bar{t}'](\Gamma) \mid \rho[\bar{\alpha} \mapsto \bar{t}, \bar{\beta} \mapsto \bar{t}'] \vdash C \wedge D\} \\
 & \text{by Definition 3.4} \\
= & \downarrow\{\rho[\bar{\alpha} \mapsto \bar{t}][\bar{\beta} \mapsto \bar{t}'](\Gamma) \mid \rho[\bar{\alpha} \mapsto \bar{t}][\bar{\beta} \mapsto \bar{t}'] \vdash D \wedge \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\} \\
 & \text{by (2)} \\
= & \cup\{\rho[\bar{\alpha} \mapsto \bar{t}](\Delta) \mid \rho[\bar{\alpha} \mapsto \bar{t}] \vdash C\} \\
 & \text{by (1) and Definition 3.4}
\end{array}
$$

□

Proof of Lemma 3.13. 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**), $\bar\beta_1 \# \mathrm{ftv}(D_2, \Gamma_2)$ (**4**), and $\bar\beta_2 \# \mathrm{ftv}(D_1, \Gamma_1)$ (**5**) hold. We have

$$
\begin{array}{ll}
& \rho(\Delta_1 \times \Delta_2) \\
= & \rho(\exists \bar\beta_1 \bar\beta_2 [D_1 \wedge D_2](\Gamma_1 \times \Gamma_2)) \\
& \text{by (1)-(5) and Definition 3.11} \\
= & \downarrow\{\rho[\bar\beta_1 \mapsto \bar t_1, \bar\beta_2 \mapsto \bar t_2](\Gamma_1 \times \Gamma_2) \mid \rho[\bar\beta_1 \mapsto \bar t_1, \bar\beta_2 \mapsto \bar t_2] \vdash D_1 \wedge D_2\} \\
& \text{by Definition 3.4} \\
= & \downarrow\{\rho[\bar\beta_1 \mapsto \bar t_1](\Gamma_1) \times \rho[\bar\beta_2 \mapsto \bar t_2](\Gamma_2) \mid \rho[\bar\beta_1 \mapsto \bar t_1] \vdash D_1 \wedge \rho[\bar\beta_2 \mapsto \bar t_2] \vdash D_2\} \\
& \text{by (4) and (5)} \\
= & \downarrow\{\rho[\bar\beta_1 \mapsto \bar t_1](\Gamma_1) \mid \rho[\bar\beta_1 \mapsto \bar t_1] \vdash D_1\} \times \downarrow\{\rho[\bar\beta_2 \mapsto \bar t_2](\Gamma_2) \mid \rho[\bar\beta_2 \mapsto \bar t_2] \vdash D_2\} \\
& \text{by definition of } \times \text{ and by distributivity of } \downarrow \text{ with respect to } \times \\
= & \rho(\Delta_1) \times \rho(\Delta_2) \\
& \text{by (1), (2), and Definition 3.4}
\end{array}
$$

□

Proof of Lemma 3.14. 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**), $\bar\beta_1 \# \mathrm{ftv}(D_2, \Gamma_2)$ (**4**), and $\bar\beta_2 \# \mathrm{ftv}(D_1, \Gamma_1)$ (**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 $\bar\alpha \# \mathrm{ftv}(\bar\beta_1, \bar\beta_2, D_1, D_2, \Gamma_1, \Gamma_2)$ (**7**) holds. We have

$$
\begin{array}{ll}
& \rho(\Delta_1 + \Delta_2) \\
= & \rho(\exists \bar\beta_1 \bar\beta_2 \bar\alpha [(D_1 \wedge \Gamma \leq \Gamma_1) \vee (D_2 \wedge \Gamma \leq \Gamma_2)]\Gamma) \\
& \text{by (1)-(7) and Definition 3.12} \\
= & \downarrow\{\rho'(\Gamma) \mid \rho' \vdash \vee_i (D_i \wedge \Gamma \leq \Gamma_i)\} \\
& \text{where } \rho' \text{ stands for } \rho[\bar\beta_1 \mapsto \bar t_1, \bar\beta_2 \mapsto \bar t_2, \bar\alpha \mapsto \bar t] \\
& \text{by Definition 3.4} \\
= & \cup_i \downarrow\{\rho'(\Gamma) \mid \rho' \vdash D_i \wedge \Gamma \leq \Gamma_i\} \\
& \text{by the interpretation of disjunction} \\
& \text{by distributivity of } \downarrow \text{ with respect to } \cup \\
= & \cup_i \downarrow\{\bar t \mid \rho[\bar\beta_i \mapsto \bar t_i] \vdash D_i \wedge \bar t \leq \rho[\bar\beta_i \mapsto \bar t_i](\Gamma_i)\} \\
& \text{by (4), (5), (6), and (7)} \\
= & \cup_i \downarrow\downarrow\{\rho[\bar\beta_i \mapsto \bar t_i](\Gamma_i) \mid \rho[\bar\beta_i \mapsto \bar t_i] \vdash D_i\} \\
& \text{by definition of } \downarrow \\
= & \cup_i \rho(\Delta_i) \\
& \text{by idempotency of } \downarrow, \text{ then by (1), (2), and Definition 3.4}
\end{array}
$$

□

Proof of Lemma 3.15. By Lemmas 3.10, 3.13, and 3.14. □

Proof of Lemma 3.19. By structural induction. □

Proof of Lemma 3.20. By structural induction. □

Proof of Lemma 3.21. 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 $\theta\bar\alpha \# \mathrm{ftv}(\Gamma, C')$ (**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 can

require $\theta\bar{\alpha} \# \mathrm{ftv}(D, \tau)$ (**8**). By Lemma 3.19, (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 3.22. 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 can 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) can 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 $\exists\bar{\alpha}_1.C_1' \wedge \exists\bar{\alpha}.D$ can 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 3.23. Consider a derivation of shape App(Sub(Abs(·$_1$)), ·$_2$). 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 3.7, (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 3.24. Let us consider an arbitrary typing derivation. Of course, every trivial instance of Gen or Inst can 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 3.21 and 3.22, Gen can be suppressed when it appears above Inst and pushed down when it appears above Hide. As a result, the derivation can be rewritten so as to satisfy (a) and (b). Next, by inspection of the rules in Figure 7, it is straightforward to check that, up to renamings of subderivations, Hide can be pushed down through every rule other than Gen while preserving (a) and (b). Furthermore, any number of consecutive instances of Hide can be collapsed into a single one. As a result, the derivation can be rewritten so as to satisfy (a), (b), and (c). At this point, one can check that, at every subexpression of the form $(\lambda\bar{c})\, e$, Abs and App can 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 3.23, can be eliminated without compromising (a), (b), or (c). Thus, the final derivation satisfies all four criteria. □

Proof of Lemma 3.25. 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 can 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) can be written $C \wedge D, \Gamma\Gamma'[x \mapsto \sigma'] \vdash e : \tau$ (**5**). By Lemma 3.19, the hypothesis $C, \bullet \vdash e : \sigma'$ yields $C \wedge D, \bullet \vdash e : \sigma'$ (**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 3.26. 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 $H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$, we obtain $\theta H, \bullet \vdash [p \mapsto v]x : \Gamma'(x)$. 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 3.24, we can assume, *w.l.o.g.*, that the derivation of $C, \bullet \vdash v : \tau$ 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 $\exists\bar{\alpha}.C, \bullet \vdash v : \tau$ (**1**) and $\exists\bar{\alpha}.C \vdash p : \tau \rightsquigarrow \Delta$ (**2**). The judgment (1) follows from an instance of Hide whose premises are $C, \bullet \vdash v : \tau$ (**3**) and $\bar{\alpha} \# \mathrm{ftv}(\tau)$ (**4**). We can assume, *w.l.o.g.*, $\bar{\alpha} \# \mathrm{ftv}(D, \Gamma)$ (**5**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 3.19 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 $x \in \mathrm{dpv}(p)$, $H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$ (**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 $\exists\bar{\alpha}.H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$. Thus, $\exists\bar{\alpha}.H$ is the desired witness.

We now reach the base case, where the derivation of $C, \bullet \vdash v : \tau$ 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 $C, \bullet \vdash v : \tau$ and $C \vdash p : \tau \rightsquigarrow \exists\varnothing[\mathsf{true}]\bullet$. Let our witness be C; then, it is immediate to check that the goal holds.

∘ *Case* p-Var. Our hypotheses are $C, \bullet \vdash v : \tau$ 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 $C, \bullet \vdash v : \tau$.

∘ *Case* p-And. Our hypotheses are $C, \bullet \vdash v : \tau$ (**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 $x \in \mathrm{dpv}(p_i)$, $H_i, \bullet \vdash [p_i \mapsto v]x : \Gamma_i(x)$ (**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 3.19 to (11), we obtain $H_1 \wedge H_2, \bullet \vdash [p \mapsto v]x : \Gamma(x)$, which is the third goal.

∘ *Case* P-Or. Our hypotheses are $C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash K\,v_1 \cdots v_n : \varepsilon(\bar{\alpha})$ (**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($\cdot$)). (Indeed, any number of consecutive occurrences of Sub can 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 $C, \bullet \vdash v_i : \tau_i'$ (**4**), for every $i \in \{1, \ldots, n\}$, $K :: \forall\bar{\alpha}'\bar{\beta}'[D'].\tau_1' \times \cdots \times \tau_n' \to \varepsilon(\bar{\alpha}')$ (**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 $i \in \{1, \ldots, n\}$, $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**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 can 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 3.19 and Sub, (4) yields $C \wedge D \wedge \tau_i' \leq \tau_i, \bullet \vdash v_i : \tau_i$ (**13**). By Lemma 3.19 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 $x \in \mathrm{dpv}(p_i)$, $H_i, \bullet \vdash [p_i \mapsto v_i]x : \Gamma_i(x)$ (**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 3.8, (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

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

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 3.19 to (17), we obtain $H, \bullet \vdash [p \mapsto v]x : (\cup_i \Gamma_i)(x)$, which is the third goal.

∘ *Case* P-EqIn. Our hypotheses are $C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash v : \tau'$ (**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 $C, \bullet \vdash v : \tau$ (**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 can 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 $x \in \mathrm{dpv}(p)$, $H', \bullet \vdash [p \mapsto v]x : \Gamma'(x)$ (**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 3.5, (4), (7), and (8) imply $C \wedge D' \Vdash \exists\bar{\beta}.(D \wedge \Gamma' \leq \Gamma)$ (**16**). So, we have

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

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 3.19 and Sub, (13) implies $H' \wedge \Gamma' \leq \Gamma, \bullet \vdash [p \mapsto v]x : \Gamma(x)$. By (10) and Hide, this implies $\exists\bar{\beta}'.(H' \wedge \Gamma' \leq \Gamma), \bullet \vdash [p \mapsto v]x : \Gamma(x)$. The third goal follows by Lemma 3.19 and by definition of H.

∘ *Case* P-Hide. Our hypotheses are $\exists\bar{\alpha}.C, \bullet \vdash v : \tau$ (**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 can 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 3.19 to (1) yields $C, \bullet \vdash v : \tau$ (**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 $x \in \mathrm{dpv}(p)$, $H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$

(**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 $\exists\bar{\alpha}.H, \bullet \vdash [p \mapsto v]x : \Gamma(x)$. Thus, $\exists\bar{\alpha}.H$ is the desired witness. □

PROOF OF THEOREM 3.27. By Lemma 3.24, we can assume that the derivation of $C, \bullet \vdash e : \sigma$ (**1**) is normal. Moreover, we can 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 $C, \bullet \vdash \lambda(p_1.e_1 \cdots p_n.e_n) : \tau' \to \tau$ (**2**) and $C, \bullet \vdash v : \tau'$ (**3**), where σ is τ. The derivation of (2) must end with an instance of ABS, whose premises include $C, \bullet \vdash p_i.e_i : \tau' \to \tau$ (**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 3.26, there exists H such that $H \Vdash D$ (**8**) and $C \equiv \exists\bar{\beta}.H$ (**9**) and, for every $x \in \mathrm{dpv}(p_i)$, $H, \bullet \vdash [p_i \mapsto v_i]x : \Gamma(x)$ (**10**) holds. By (9), we have $H \Vdash C$; together with (8), this implies $H \Vdash C \wedge D$. Thus, applying Lemma 3.19 to (6), we find $H, \Gamma \vdash e_i : \tau$ (**11**). By Lemma 3.25, (10) and (11) imply $H, \bullet \vdash [p_i \mapsto v_i]e_i : \tau$ (**12**). Applying HIDE to (12) and (7) and exploiting (9), we obtain $C, \bullet \vdash [p_i \mapsto v_i]e_i : \tau$.

∘ *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 3.25.

∘ *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 $C, \bullet \vdash v : \sigma'$ and $C, (x \mapsto \sigma') \vdash e_1 : \sigma$. The result follows by Lemma 3.25.

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

PROOF OF LEMMA 3.28. By Lemma 3.24, we can assume, *w.l.o.g.*, that the derivation of $C, \bullet \vdash v : \tau$ 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 $\exists\bar{\alpha}.C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash v : \tau$ (**3**). Because C entails $\exists\bar{\alpha}.C$, applying Lemma 3.19 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 $C, \bullet \vdash v : \tau$ 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 $C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash v : \varepsilon(\bar{\alpha})$ 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 can 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 3.6, 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 $C, \bullet \vdash v_i : \tau_i'$ (**2**), for every $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha}' \bar{\beta}' [D'].\tau_1' \times \cdots \times \tau_n' \to \varepsilon(\bar{\alpha}')$ (**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 $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha} \bar{\beta} [D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**6**), and $\bar{\beta} \# \mathrm{ftv}(C)$ (**7**). We can further require, *w.l.o.g.*, $\bar{\beta} \# \bar{\alpha}' \bar{\beta}'$ (**8**). By Requirement 3.8, (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 3.19 and by SUB, (2) and (5) yield $C \wedge D \wedge_i \tau_i' \leq \tau_i, \bullet \vdash v_i : \tau_i$ 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 $C, \bullet \vdash v : \tau$ (**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 $C, \bullet \vdash v : \tau'$ (**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 $C, \bullet \vdash v : \tau$ (**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 $\exists \bar{\alpha}.C, \bullet \vdash v : \tau$ (**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 3.19 to (1) yields $C, \bullet \vdash v : \tau$ (**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 3.29. Left to the reader. □

PROOF OF THEOREM 3.30. Suppose $C, \bullet \vdash e : \sigma$ (**1**) where C is a satisfiable constraint. We can 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 $C, \bullet \vdash e_1 : \tau' \to \tau$ (**2**) and $C, \bullet \vdash e_2 : \tau'$ (**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 can be expanded or collapsed to a single one, we can assume that the derivation of (2) ends with the shape SUB(CSTR($\cdot$)). SUB's second premise must then be of the form $C \Vdash \varepsilon(\bar{\alpha}) \leq \tau' \to \tau$, a contradiction, by Requirement 3.6, 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 3.28 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 3.31. Suppose e reduces to e'. By Theorem 3.27, e' is well-typed. Because reduction preserves the property that all case analyses are exhaustive, Theorem 3.30 is applicable and guarantees that e' is not stuck. □

PROOF OF THEOREM 3.32. 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 3.30, $\lfloor e \rfloor$ 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 3.29, $E^*[\bot]$ is not, contradicting the subject reduction property (Theorem 3.27). 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 3.33. 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 3.27, $\lfloor e' \rfloor$ is well-typed. Then, Theorem 3.32 guarantees that e' is not stuck. □

Proof of Lemma 4.1. Left to the reader. □

Proof of Lemma 4.3. 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 3.19 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 3.19 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 $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**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 3.19 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 3.19 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 can 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 4.4. By structural induction on p. □

Proof of Lemma 4.5. 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 $i \in \{1, \ldots, n\}$, $K :: \forall \bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**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 can 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 4.1, 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 can 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 4.1, 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 4.4, 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 4.7. 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 $\lambda \bar{c}$. Let $\alpha_1 \alpha_2 \# \mathrm{ftv}(\Gamma, \tau)$ (**1**). Applying the induction hypothesis in turn to each member of $\bar{c}$ yields $(\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!), \Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2$, which by ABS implies $(\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!), \Gamma \vdash \lambda \bar{c} : \alpha_1 \to \alpha_2$ (**2**). By Lemma 3.19 and by SUB, (2) implies $(\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau, \Gamma \vdash \lambda \bar{c} : \tau$. By (1) and by HIDE, this implies $\exists \alpha_1 \alpha_2.((\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau), \Gamma \vdash \lambda \bar{c} : \tau$, which is precisely the goal $(\!|\Gamma \vdash \lambda \bar{c} : \tau|\!), \Gamma \vdash \lambda \bar{c} : \tau$.

∘ *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 3.19 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 $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**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 3.19, $\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau, \Gamma \vdash e_i : \tau_i$ (**3**) follows. Applying Cstr to (3), where i ranges over $\{1, \ldots, n\}$, and to (1), we obtain $\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau, \Gamma \vdash K\,e_1 \cdots e_n : \varepsilon(\bar{\alpha})$. By Sub, (2), and Hide, this implies $\exists\bar{\alpha}\bar{\beta}.(\wedge_i(\!|\Gamma \vdash e_i : \tau_i|\!) \wedge D \wedge \varepsilon(\bar{\alpha}) \leq \tau), \Gamma \vdash K\,e_1 \cdots e_n : \tau$, that is, $(\!|\Gamma \vdash K\,e_1 \cdots e_n : \tau|\!), \Gamma \vdash K\,e_1 \cdots e_n : \tau$.

∘ *Case ce* is $\mu(x : \exists\bar{\beta}.\sigma).e$. By convention, we have $\bar{\beta} \# \mathrm{ftv}(\Gamma, \tau)$ (**1**). Write σ as $\forall\bar{\gamma}[C].\tau_1$, where $\bar{\gamma} \# \mathrm{ftv}(\Gamma)$ (**2**). By the induction hypothesis, we have $(\!|\Gamma[x \mapsto \sigma] \vdash e : \tau_1|\!), \Gamma[x \mapsto \sigma] \vdash e : \tau_1$ (**3**). Let D stand for $\forall\bar{\gamma}.C \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash e : \tau_1|\!)$. Then, $C \wedge D$ entails $(\!|\Gamma[x \mapsto \sigma] \vdash e : \tau_1|\!)$, so, by Lemma 3.19, (3) implies $C \wedge D, \Gamma[x \mapsto \sigma] \vdash e : \tau_1$ (**4**). Furthermore, we have $\bar{\gamma} \# \mathrm{ftv}(D)$ (**5**). By (4), (2), (5), and FixAnnot, we obtain $\exists\bar{\gamma}.C \wedge D, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \sigma$ (**6**). Because $\sigma \leq \tau$ entails $\exists\bar{\gamma}.C$, (6) and Lemma 3.19 yield $\sigma \leq \tau \wedge D, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \sigma$ (**7**). By Inst, Sub, and Hide, (7) implies $\sigma \leq \tau \wedge D, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \tau$ (**8**). By (8), (1), and Hide, we obtain $\exists\bar{\beta}.(\sigma \leq \tau \wedge D), \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \tau$, which by definition of D is the goal.

∘ *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 3.19 and Let.

∘ *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 4.3, 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 3.19, (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 4.8. By induction on the structure of *ce*. □

Proof of Lemma 4.9. By induction on the structure of *ce*. □

Proof of Lemma 4.10. We assume $\bar{\beta}_1\bar{\beta}_2 \# \mathrm{ftv}(\Gamma, \tau)$ (**1**). Up to a renaming of the goal, we can 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 4.9, we have $\Gamma_1 \leq \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$. 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 $\exists\bar{\beta}_2.(\Gamma_1 \leq \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!)) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$ (**4**). By (2) and (3), we have $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge D_1 \Vdash \exists\bar{\beta}_2.(D_2 \wedge \Gamma_1 \leq \Gamma_2)$; then $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge D_1 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \exists\bar{\beta}_2.(\Gamma_1 \leq \Gamma_2 \wedge (\!|\Gamma\Gamma_2 \vdash e : \tau|\!))$. By transitivity with (4), $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge D_1 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$ (**5**) follows. By (1), (2) and (3), $\bar{\beta}_1 \# \mathrm{ftv}(\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!))$, so (4) can be rewritten into $\exists\bar{\beta}_1[D_1]\Gamma_1 \leq \exists\bar{\beta}_2[D_2]\Gamma_2 \wedge \forall\bar{\beta}_2.D_2 \Rightarrow (\!|\Gamma\Gamma_2 \vdash e : \tau|\!) \Vdash \forall\bar{\beta}_1.D_1 \Rightarrow (\!|\Gamma\Gamma_1 \vdash e : \tau|\!)$, which is the goal. □

Proof of Theorem 4.11. 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 can 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 FixAnnot and Gen below, we can 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 3.2.

∘ *Case* Abs. $(\mathcal{H})$ is $C, \Gamma \vdash \lambda\bar{c} : \tau_1 \to \tau_2$. Abs' premise is $C, \Gamma \vdash \bar{c} : \tau_1 \to \tau_2$, which by the induction hypothesis implies $C \Vdash (\!|\Gamma \vdash \bar{c} : \tau_1 \to \tau_2|\!)$ (**1**). Pick $\alpha_1\alpha_2 \# \mathrm{ftv}(C, \tau_1, \tau_2)$. Then, we have $C \Vdash \exists\alpha_1\alpha_2.(C \wedge \alpha_1 = \tau_1 \wedge \alpha_2 = \tau_2)$ (**2**). Furthermore, by Lemma 4.8, (1) implies $C \wedge \alpha_1 = \tau_1 \wedge \alpha_2 = \tau_2 \Vdash (\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau_1 \to \tau_2$ (**3**). Combining (2) and (3), we obtain $C \Vdash \exists\alpha_1\alpha_2.((\!|\Gamma \vdash \bar{c} : \alpha_1 \to \alpha_2|\!) \wedge \alpha_1 \to \alpha_2 \leq \tau_1 \to \tau_2)$, that is, $C \Vdash (\!|\Gamma \vdash \lambda\bar{c} : \tau_1 \to \tau_2|\!)$.

∘ *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 $i \in \{1, \ldots, n\}$, $K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \varepsilon(\bar{\alpha})$ (**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 $\alpha \notin \mathrm{ftv}(C, \tau')$ (**4**). By (4), we have $C \Vdash \exists\alpha.(C \wedge \alpha = \tau')$ (**5**). Furthermore, by Lemma 4.8, (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* FixAnnot. $(\mathcal{H})$ is $C \wedge \exists\bar{\alpha}.D, \Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \sigma$. FixAnnot's premises are $C \wedge D, \Gamma[x \mapsto \sigma] \vdash e : \tau$ (**1**), $\bar{\alpha} \# \mathrm{ftv}(C, \Gamma)$ (**2**), and $\sigma = \forall\bar{\alpha}[D].\tau$ (**3**). By the induction hypothesis, (1) implies $C \wedge D \Vdash (\!|\Gamma[x \mapsto \sigma] \vdash e : \tau|\!)$, which, by (2), can be written $C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma[x \mapsto \sigma] \vdash e : \tau|\!)$, that is, $C \Vdash (\!|\Gamma[x \mapsto \sigma] \vdash e : \sigma|\!)$ (**4**). Furthermore, by (3) and by Lemma 3.2, we have $D \Vdash \sigma \leq \tau$ (**5**). Combining (4) and (5), we obtain $C \wedge D \Vdash (\!|\Gamma[x \mapsto \sigma] \vdash e : \sigma|\!) \wedge \sigma \leq \tau$. This implies $C \wedge D \Vdash \exists\bar{\beta}.((\!|\Gamma[x \mapsto \sigma] \vdash e : \sigma|\!) \wedge \sigma \leq \tau)$, that is, $C \wedge D \Vdash (\!|\Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \tau|\!)$ (**6**). The goal $C \wedge \exists\bar{\alpha}.D \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash \mu(x : \exists\bar{\beta}.\sigma).e : \tau|\!)$ follows from (2) and (6).

∘ *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 4.8 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 can 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 4.9, we obtain $C \wedge D \Vdash (\!|\Gamma[x \mapsto \forall\alpha[H].\alpha] \vdash e_2 : \tau|\!)$ (**9**). By Lemma 3.20, (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 can further assume, *w.l.o.g.*, $\bar{\beta} \# \mathrm{ftv}(\tau_1)$ (**4**). By applying Lemma 4.5 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), can 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 4.10 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 $\theta\bar{\alpha} \# \mathrm{ftv}(\Gamma)$ (**4**). By (3), (1) can 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 4.8 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 $\bar{\beta} \# \mathrm{ftv}(\Gamma, \sigma)$ (**2**). Write σ as $\forall\bar{\alpha}[D].\tau$, where $\bar{\alpha} \# \mathrm{ftv}(\Gamma)$. 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 $\exists\bar{\beta}.C \Vdash \forall\bar{\alpha}.D \Rightarrow (\!|\Gamma \vdash ce : \tau|\!)$. □

REFERENCES

AIKEN, A. S. AND WIMMERS, E. L. 1993. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*. ACM Press, 31–41.

BIRD, R. AND MEERTENS, L. 1998. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*. Lecture Notes in Computer Science, vol. 1422. Springer Verlag, 52–67.

CHENEY, J. AND HINZE, R. 2002. A lightweight implementation of generics and dynamics. In *Haskell workshop*.

CHENEY, J. AND HINZE, R. 2003. First-class phantom types. Tech. Rep. 1901, Cornell University.

COMON, H. AND LESCANNE, P. 1989. Equational problems and disunification. *Journal of Symbolic Computation 7*, 371–425.

CRARY, K., WEIRICH, S., AND MORRISETT, G. 2002. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming 12,* 6 (Nov.), 567–600.

CURTIS, P. 1990. Constrained quantification in polymorphic type analysis. Ph.D. thesis, Cornell University.

FREEMAN, T. AND PFENNING, F. 1991. Refinement types for ML. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 268–277.

HANUS, M. 1988. Horn clause specifications with polymorphic types. Ph.D. thesis, Fachbereich Informatik, Universität Dortmund.

HANUS, M. 1989. Horn clause programs with polymorphic types: Semantics and resolution. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*. Lecture Notes in Computer Science, vol. 352. Springer Verlag, 225–240.

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

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

Hinze, R. 2003. Fun with phantom types. In *The Fun of Programming*, J. Gibbons and O. de Moor, Eds. Palgrave Macmillan, 245–262.

Jones, M. P. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press.

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

Jones, S. P., Vytiniotis, D., Weirich, S., and Washburn, G. 2005. Simple unification-based type inference for GADTs. Submitted.

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

Läufer, K. and Odersky, M. 1994. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems 16,* 5 (Sept.), 1411–1430.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. 2005. *The Objective Caml system*.

Maher, M. J. 1988. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *IEEE Symposium on Logic in Computer Science (LICS)*. 348–357.

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

Mitchell, J. C. 1984. Coercion and type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*. 175–185.

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

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

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

Peyton Jones, S. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall.

Peyton Jones, S., Ed. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Peyton Jones, S. and Shields, M. 2004. Lexically-scoped type variables. Manuscript.

Peyton Jones, S., Washburn, G., and Weirich, S. 2004. Wobbly types: type inference for generalised algebraic data types. Tech. Rep. MS-CIS-05-26, University of Pennsylvania. July.

Pierce, B. C. and Turner, D. N. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems 22,* 1 (Jan.), 1–44.

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

Pottier, F. and Régis-Gianas, Y. 2006. Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL)*.

Pottier, F. and Rémy, D. 2005. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT Press, Chapter 10, 389–489.

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

Sheard, T. 2004. Languages of the future. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 116–119.

Sheard, T. 2005. *Ωmega*.

Sheard, T. and Pašalić, E. 2004. Meta-programming with built-in type equality. In *Workshop on Logical Frameworks and Meta-Languages (LFM)*.

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

Simonet, V. and Pottier, F. 2005. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA. Jan.

Smith, G. S. 1994. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming 23,* 2–3 (Dec.), 197–226.

Stuckey, P. J. and Sulzmann, M. 2005. Type inference for guarded recursive data types. Manuscript.

Sulzmann, M. 2000. A general framework for Hindley/Milner type systems with constraints. Ph.D. thesis, Yale University, Department of Computer Science.

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

Trifonov, V. and Smith, S. 1996. Subtyping constrained types. In *Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 1145. Springer Verlag, 349–365.

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

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

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

Warren, D. H. D. 1982. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, J. E. Hayes, D. Michie, and Y.-H. Pao, Eds. Ellis Horwood, 441–454.

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

Werner, B. 1994. Une théorie des constructions inductives. Ph.D. thesis, Université Paris 7.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115,* 1 (Nov.), 38–94.

Xi, H. 1998. Dependent types in practical programming. Ph.D. thesis, Carnegie Mellon University.

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

Xi, H. 2001. Dependent ML.

Xi, H. 2003. Dependently Typed Pattern Matching. *Journal of Universal Computer Science 9,* 8, 851–872.

Xi, H. 2004. Applied type system. In *TYPES 2003*. Lecture Notes in Computer Science, vol. 3085. Springer Verlag, 394–408.

Xi, H., Chen, C., and Chen, G. 2003. Guarded recursive datatype constructors. In *ACM Symposium on Principles of Programming Languages (POPL)*. 224–235.

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

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

Zenger, C. 1998. Indizierte typen. Ph.D. thesis, Universität Karlsruhe.