Stratified type inference for generalized algebraic data types

François Pottier Yann Régis-Gianas

INRIA

{Francois.Pottier, Yann.Regis-Gianas}@inria.fr

Abstract

We offer a solution to the type inference problem for an extension of Hindley and Milner's type system with generalized algebraic data types. Our approach is in two *strata*. The bottom stratum is a core language that marries type *inference* in the style of Hindley and Milner with type *checking* for generalized algebraic data types. This results in an extremely simple specification, where case constructs must carry an explicit type annotation and type conversions must be made explicit. The top stratum consists of (two variants of) an independent *shape inference* algorithm. This algorithm accepts a source term that contains some explicit type information, propagates this information in a local, predictable way, and produces a new source term that carries more explicit type information. It can be viewed as a preprocessor that helps produce some of the type annotations required by the bottom stratum. It is proven *sound* in the sense that it never inserts annotations that could contradict the type derivation that the programmer has in mind.

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

1. Introduction

1.1 Generalized algebraic data types

Generalized algebraic data types are a simple generalization of the algebraic data types of ML and Haskell. They are strongly reminiscent of the *inductive types* that have long existed in the Calculus of Inductive Constructions [10, 27]. In the programming languages area, Crary, Weirich and Morrisett [3] have exploited one particular generalized algebraic data type, known as R, to encode a correspondence between compile-time types and run-time type representations. More recently, generalized algebraic data types have been put to a large variety of uses, under diverse names, by many authors, among whom Cheney and Hinze [1, 2], Xi *et al.* [28], Hinze [5], Sheard [21], Sheard and Pasalic [22], Pottier and Gauthier [14], and Pottier and Régis-Gianas [16].

A typical use of generalized algebraic data types is writing a safe evaluator for a simply-typed object language that does not require values to carry run-time tags. The algebraic data type $term$ of abstract syntax trees is given a type parameter α so that values of type $term\ \alpha$ are abstract syntax trees for object-level expressions of type α. For instance, Peyton Jones, Washburn, and Weirich [12] define $term$ by associating the following data constructors with it:

$$\begin{aligned}
Lit &:: int \rightarrow term\ int\\
Inc &:: term\ int \rightarrow term\ int\\
IsZ &:: term\ int \rightarrow term\ bool\\
If &:: \forall\alpha.term\ bool \rightarrow term\ \alpha \rightarrow term\ \alpha \rightarrow term\ \alpha\\
Pair &:: \forall\alpha\beta.term\ \alpha \rightarrow term\ \beta \rightarrow term\ (\alpha \times \beta)\\
Fst &:: \forall\alpha\beta.term\ (\alpha \times \beta) \rightarrow term\ \alpha\\
Snd &:: \forall\alpha\beta.term\ (\alpha \times \beta) \rightarrow term\ \beta
\end{aligned}$$

This definition allows writing an evaluator that does not perform any tagging or untagging of object-level values:

$$\begin{array}{l}
\mu(eval : \forall\alpha.term\ \alpha \rightarrow \alpha).\lambda t.\\
\quad \mathsf{case}\ t\ \mathsf{of}\\
\qquad \mid Lit\ i \rightarrow i\\
\qquad \mid Inc\ t \rightarrow eval\ t + 1\\
\qquad \mid IsZ\ t \rightarrow eval\ t = 0\\
\qquad \mid If\ b\ t\ e \rightarrow \mathsf{if}\ eval\ b\ \mathsf{then}\ eval\ t\ \mathsf{else}\ eval\ e\\
\qquad \mid Pair\ a\ b \rightarrow (eval\ a, eval\ b)\\
\qquad \mid Fst\ t \rightarrow fst\ (eval\ t)\\
\qquad \mid Snd\ t \rightarrow snd\ (eval\ t)
\end{array}$$

This program is well-typed in an extension of Milner and Mycroft's type system [7] with generalized algebraic data types, such as the type system MLGI defined in §3. A key mechanism is the introduction, at case constructs, of type equations into the typing context. For instance, in the first branch of $eval$, the variable t, which has type $term\ \alpha$, is known to match the pattern $Lit\ i$, which, according to the declaration of Lit, has type $term\ int$. As a result, the equation $\alpha = int$ must hold within that branch. This equation is added to the typing context and exploited by the typechecker to prove that the outcome of this branch, that is, the integer variable i, has type α, as required by $eval$'s signature. Nontrivial equations are also exploited in the Inc, IsZ, and $Pair$ cases.

1.2 Type inference

The papers cited above explain at length why generalized algebraic data types are useful. Here, we take this fact for granted and focus on the *type inference* problem. It is well-known that, provided recursive definitions carry an explicit type annotation, type inference for Milner and Mycroft's type system reduces to first-order unification under a mixed prefix, that is, to satisfiability of formulas made up of equations between finite trees, conjunction, and existential and universal quantification. Now, what happens when generalized algebraic data types are thrown into the mix?

In short, "things become more difficult." Indeed, when an equation such as $\alpha = int$ is available, a type inference system is faced with choices. For instance, the integer variable i can be assigned type int or type α. Outside of this case branch, α and int cannot be considered equal, so this choice becomes observable. How do we resolve it without resorting to combinatorial search? Furthermore, how do we know that the equation $\alpha = int$ is available?

That is, how do we determine which equations are available in the first place? Inferring which equations are introduced at a `case` construct requires inferring the type of the scrutinee. Thus, inference of types and inference of equation systems are interdependent.

1.3 Related work

Simonet and Pottier [23] show that type inference for $\mathrm{HMG}(X)$, an extension of $\mathrm{HM}(X)$ [8] with generalized algebraic data types, can be reduced to satisfiability of formulas in the first-order theory of equality of finite trees, that is, of formulas made up of equations between finite trees, conjunction, existential and universal quantification, and *implication*. Unfortunately, this problem is intractable [26]. Simonet and Pottier suggest relying on *explicit type annotations* in order to produce formulas in a less expressive, more tractable theory, but leave that as future work.

Stuckey and Sulzmann [25, 24] also reduce type inference to solving constraints that involve implications. They then develop incomplete constraint solvers. Unfortunately, this approach seems expensive, because solving can involve combinatorial search, and makes it hard to understand why a program is accepted or rejected.

Peyton Jones, Washburn, and Weirich [12] describe a proposal, implemented in version 6.4 of the Glasgow Haskell compiler, which appears to work well in practice. They also suggest exploiting programmer-supplied type annotations. They argue that, in the definition of $eval$, it is “clear” that t has type $term\ \ \alpha$, which implies that the equation $\alpha = int$ becomes available in the Lit branch. It is similarly “clear” that the outcome of this branch should have type α. Instead of reasoning with implications, Peyton Jones *et al.* apply the substitution $[\alpha \mapsto int]$ to the expected type α, so that the expression i is checked against the expected type int, with success.

Although Peyton Jones *et al.*’s system aims at simplicity, its definition is quite involved. To ensure that “*type refinement [...] is insensitive to the order in which the inference algorithm traverses the tree,*” they distinguish *wobbly types*, which are inferred via first-order unification, and *rigid* types (our terminology), which are also inferred, but in a simpler and hopefully more *predictable* way: rigid types are found in explicit type annotations and propagated up and down the tree according to a predetermined set of rules.

Wobbly types are identified by a dedicated type constructor, written $\boxed{\cdot}$. This type constructor has no computational meaning. It sometimes has to be erased or pushed out of the way: for instance, $\boxed{\tau_1 \rightarrow \tau_2}$ is converted to $\boxed{\tau_1} \rightarrow \boxed{\tau_2}$ at application nodes. Its behavior is sometimes surprising: for instance, the application of a type substitution to a wobbly type $\boxed{\tau}$ is defined to be $\boxed{\tau}$ itself. These and other aspects make it difficult to understand the deep meaning of “wobbliness” and to predict when a type has to be wobbly.

1.4 Our approach

In short, we believe that the ideas behind “wobbly types” are good, but that they could and should be better presented and explained. In particular, one should clearly separate two subsystems:

- one that performs *traditional type inference* in the style of Hindley and Milner, and enjoys a *principal types* property, so that it produces types that are indeed “*insensitive to the order in which the inference algorithm traverses the tree,*” and
- one that performs *local type inference*, that is, *ad hoc* propagation of explicit type annotations, either in Peyton Jones *et al.*’s bidirectional style, or in other ways.

This separation makes the presentation of the system more modular and more compelling. Furthermore, it allows identifying choices and issues in the design of the local type inference component that were not explicit in Peyton Jones *et al.*’s paper. In particular, we highlight and discuss a *soundness* issue that was not addressed by Peyton Jones *et al.*

We first set the stage by introducing a type system that extends Milner and Mycroft’s type system with generalized algebraic data types (§2 and §3). This type system defines the programs that we deem sound and would ultimately like to accept. It does not require any explicit type annotations (except, following common practice, at recursive definitions), because it is not meant to allow type inference. We refer to it as MLGI (read: “**ML** with **g**eneralized algebraic data types in **i**mplicit style”).

Then (§4), we introduce a type system that restricts MLGI by requiring explicit type annotations wherever generalized algebraic data types are involved. More specifically, type equations can be *introduced* into the typing context, at a `case` construct, only if the scrutinee carries an explicit type annotation. Furthermore, type equations can be *exploited* only via explicit *type coercions*. We refer to this type system as MLGX (read: “**ML** with **g**eneralized algebraic data types in e**x**plicit style”). The benefit of these restrictions is that MLGX enjoys traditional type inference in the style of Hindley and Milner: that is, type inference for MLGX can be reduced to first-order unification under a mixed prefix.

Programming directly in MLGX would be quite painful, because of the many explicit type annotations that MLGX requires. To alleviate this burden, we next design *local type inference* algorithms that accept a program that contains some explicit type annotations, propagates this information in a predictable way, and produces a new program that carries more type annotations. These algorithms can be viewed as preprocessors that help produce some of the type annotations required by MLGX.

Our local type inference algorithms propagate *shapes* (§5). Roughly speaking, shapes are types that contain holes. A hole appears where a wobbly type would appear in Peyton Jones *et al.*’s proposal. This formalizes the intuition that the types “inside the wobbly boxes” should be invisible to the local type inference component. Our shapes have pleasant algebraic properties and seem particularly well-suited for expressing *incomplete* knowledge about types in (variants of) Hindley and Milner’s type system.

We present two local type inference (or *shape inference*) algorithms. The former (§6) closely follows Peyton Jones *et al.*’s bidirectional propagation technique. The latter (§7) enhances the former by performing checking and inference *simultaneously* and by supporting *iterated* shape inference. These features lead, in particular, to a more accurate and less *ad hoc* treatment of application.

All proofs are omitted. Some can be found in an extended version of this paper [15].

2. Preliminary definitions

Algebraic data type constructors We assume that a number of algebraic data type constructors, written ε, are given. Every algebraic data type constructor ε is parameterized over *two* groups of type parameters: that is, applications of ε are of the form $\varepsilon\,\bar\tau_1\,\bar\tau_2$, where $\bar\tau_1$ and $\bar\tau_2$ are vectors of types. We refer to parameters in the first group as *ordinary* and to parameters in the second group as *generalized*. When the second group is empty, ε is said to be an ordinary algebraic data type; when it is nonempty, ε is a generalized algebraic data type. For instance, the algebraic data type constructor $term$ of §1 has one generalized type parameter and no ordinary type parameters.

Data constructors We assume that every algebraic data type constructor ε comes with a number of data constructors, written K. Every data constructor K is assigned a closed type scheme by a declaration of the form

$$K :: \forall\bar\alpha\bar\beta.\tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon\,\bar\alpha\,\bar\tau,$$

Types	$\tau ::=$
Type variable	$\quad\alpha$
Function type	$\quad\mid \tau \rightarrow \tau$
Algebraic data type	$\quad\mid \varepsilon\,\bar{\tau}\,\bar{\tau}$
Type schemes	$\sigma ::= \forall\bar{\alpha}.\tau$
Simple type annotations	$\theta ::= \exists\bar{\gamma}.\tau$
Polymorphic type annotations	$\varsigma ::= \exists\bar{\gamma}.\sigma$
Type coercions	$\kappa ::= \exists\bar{\gamma}.(\tau \triangleright \tau)$
Terms	$t ::=$
Variable	$\quad x$
Function	$\quad\mid \lambda(x : \theta).t$
Function application	$\quad\mid t\;t$
Local definition	$\quad\mid \mathsf{let}\,x\;=\;t\,\mathsf{in}\,t$
Fixpoint	$\quad\mid \mu(x : \varsigma).t$
Data constructor application	$\quad\mid K\,t\ldots t$
Case analysis	$\quad\mid \mathsf{case}\,t\,\mathsf{of}\,\bar{c}$
Type variable introduction	$\quad\mid \forall\bar{\alpha}.t$
Type annotation	$\quad\mid (t : \theta)$
Type coercion	$\quad\mid (t : \kappa)$
Clauses	$c ::= p.t$
Patterns	$p ::= K\,\bar{\beta}\,\bar{x}$
Equation systems	$E ::= \mathsf{true} \mid \tau = \tau \mid E \wedge E$
Constraints	$C ::=$
Equations and conjunction	$\quad\mathsf{true} \mid \tau = \tau \mid C \wedge C$
Existential quantification	$\quad\mid \exists\bar{\gamma}.C$
Universal quantification	$\quad\mid \forall\bar{\alpha}.C$

Figure 1. Types, terms, constraints

where $\bar{\alpha}\,\#\,\bar{\beta}$ and $\mathrm{ftv}(\bar{\tau}) \subseteq \bar{\beta}$. Here and elsewhere, $\bar{\alpha}$ stands for a vector of distinct type variables. The length of the vector $\bar{\beta}$ is the number of type variables *introduced by* K. When ε is an ordinary algebraic data type constructor, it is zero. We write $K \preceq \sigma$ when σ is an instance of the type scheme assigned to K.

Types A number of syntactic categories that are used throughout the paper are defined in Figure 1. A *type* τ is a type variable α, a function type $\tau_1 \rightarrow \tau_2$, or an application of an algebraic data type constructor ε to vectors of type parameters $\bar{\tau}_1$ and $\bar{\tau}_2$. A *type scheme* $\forall\bar{\alpha}.\tau$ binds a vector $\bar{\alpha}$ of type variables within a type τ. Every type of the form $[\bar{\alpha} \mapsto \bar{\tau}]\tau$ is an *instance* of the type scheme $\forall\bar{\alpha}.\tau$. We write $\sigma \preceq \tau$ when τ is an instance of σ. Similarly, a *simple type annotation* $\exists\bar{\gamma}.\tau$ binds $\bar{\gamma}$ within a type τ; a *polymorphic type annotation* $\exists\bar{\gamma}.\sigma$ binds $\bar{\gamma}$ within a type scheme σ; and a *coercion* $\exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)$ binds $\bar{\gamma}$ within a pair of types $(\tau_1 \triangleright \tau_2)$. The instance relations $\theta \preceq \tau$, $\varsigma \preceq \sigma$, and $\kappa \preceq (\tau_1 \triangleright \tau_2)$ are defined accordingly.

Terms In every λ-abstraction, the bound variable x carries a simple type annotation θ. The unannotated abstraction $\lambda x.t$ can be defined as syntactic sugar for $\lambda(x : \exists\gamma.\gamma).t$, since every type is an instance of the uninformative annotation $\exists\gamma.\gamma$. Similarly, in every fixpoint, the bound variable x carries a polymorphic type annotation ς, so as to avoid the difficulties associated with polymorphic recursion in the absence of any annotation [4]. When ς is $\exists\gamma.\gamma$, the type of x is inferred, but must be monomorphic.

Every case construct involves a vector of clauses $\bar{c}$. A *clause* is a pair of a *pattern* of the form $K\,\bar{\beta}\,\bar{x}$ and of a term t, where $\bar{\beta}$ and $\bar{x}$ are bound within t. For simplicity, we deal with shallow patterns only. The length of $\bar{\beta}$ must match the number of type variables introduced by K.

In the construct $\forall\bar{\alpha}.t$, the type variables $\bar{\alpha}$ are bound within t. They are interpreted as universally bound, which means that t should be well-typed under every instantiation of these type variables. In practice, one should also introduce the dual construct $\exists\bar{\alpha}.t$, where the type variables are existentially bound, so that t must be well-typed under some instantiation of them. Instead, in this paper, we build existential quantifiers into type annotations θ and ς and type coercions κ. This simplifies our presentation, because every type variable that appears free inside a type annotation or type coercion can be assumed to be *rigid*, that is, to be universally quantified somewhere up in the term.

Equation systems and constraints An *equation system* E is a (possibly empty) conjunction of type equations. Equation systems are used in typing judgments to keep track of the type equations introduced at case constructs. *Constraints* encode unification problems, where "unification" means first-order unification under a mixed prefix. Constraints are used to express type inference problems. *Satisfiability*, *entailment*, and *equivalence* of constraints are defined via a standard interpretation in the Herbrand universe, that is, in the finite tree model. We write $C_1 \Vdash C_2$ when C_1 entails C_2. We also define $C \Vdash \exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)$ as syntactic sugar for $C \Vdash \forall\bar{\gamma}.(\tau_1 = \tau_2)$. Intuitively, $C \Vdash \kappa$ means that, according to the constraint C, all instances of the coercion κ are valid. This is further explained in §3.

3. An ideal type system

MLGI is an extension of Milner and Mycroft's type system with explicit type annotations and generalized algebraic data types. It is the "ideal" type system that we are interested in. MLGI is expressive, but type inference for it is not easy. As a result, we later develop type systems where some type annotations are mandatory, and prove them sound with respect to MLGI.

Presentation MLGI's typing judgments are of the form $E, \Gamma \vdash t : \sigma$, where E is an equation system, Γ assigns type schemes to variables, t is a term, and σ is a type scheme. The type system is defined in Figure 2.

The presence of generalized algebraic data types in the language requires keeping track of the equations that have been discovered at case constructs. This is the role of E. This equation system is augmented by rule CLAUSE and is exploited by rules CONV and COERCE, which perform implicit and explicit type coercions, respectively.

CONV allows replacing a type τ_1 with another type τ_2 at any time, provided $E \Vdash \tau_1 = \tau_2$ holds, that is, provided E guarantees that this conversion is valid. This rule is not syntax-directed. COERCE serves the same purpose, but is syntax-directed: the special construct $(t : \kappa)$ is interpreted as an explicit request for a coercion. In the simplest case, κ is of the form $(\tau_1 \triangleright \tau_2)$. In that case, the second premise vanishes, and the third premise becomes $E \Vdash \tau_1 = \tau_2$, so COERCE and CONV have identical premises. In the general case, the structure of τ_1 and τ_2 is only partially specified by the programmer, that is, κ is of the form $\exists\bar{\gamma}.(\tau_1' \triangleright \tau_2')$. This is interpreted as a request to convert between τ_1' and τ_2', for *some* value of the "flexible" type variables $\bar{\gamma}$. To ensure soundness, *all* such coercions should be valid, which is why $E \Vdash \kappa$ is defined in that case as $E \Vdash \forall\bar{\gamma}.\tau_1' = \tau_2'$.

Explicit type coercions really are of no use in MLGI, since implicit coercions are also allowed. They become essential in MLGX (§4), where CONV is suppressed.

Rule CLAUSE is invoked by rule CASE to typecheck a clause of the form $p.t$. The pattern p binds variables and type variables within the term t, and also introduces new equations. To reflect this, CLAUSE's first premise confronts p with the type τ_1 of the scrutinee, giving rise to new type variables $\bar{\beta}$, equations E', and variable bindings Γ', which are used in the second premise to typecheck the term t.

Rule PAT confronts a pattern $K\,\bar{\beta}\,x_1 \ldots x_n$ with the type of the scrutinee. Obviously, this type must be of the form $\varepsilon\,\bar{\tau}_1\,\bar{\tau}_2$, where

$$\textsc{Var}\quad \frac{(x : \sigma) \in \Gamma}{E, \Gamma \vdash x : \sigma}$$

$$\textsc{Lam}\quad \frac{E, (\Gamma; x : \tau_1) \vdash t : \tau_2 \qquad \theta \preceq \tau_1}{E, \Gamma \vdash \lambda(x : \theta).t : \tau_1 \rightarrow \tau_2}$$

$$\textsc{App}\quad \frac{\begin{array}{c} E, \Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \\ E, \Gamma \vdash t_2 : \tau_1 \end{array}}{E, \Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\textsc{Let}\quad \frac{E, \Gamma \vdash t_1 : \sigma \qquad E, (\Gamma; x : \sigma) \vdash t_2 : \tau}{E, \Gamma \vdash \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 : \tau}$$

$$\textsc{Fix}\quad \frac{E, (\Gamma; x : \sigma) \vdash t : \sigma \qquad \varsigma \preceq \sigma}{E, \Gamma \vdash \mu(x : \varsigma).t : \sigma}$$

$$\textsc{Cstr}\quad \frac{\begin{array}{c} K \preceq \tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \\ \forall i \quad E, \Gamma \vdash t_i : \tau_i \end{array}}{E, \Gamma \vdash K\, t_1 \ldots t_n : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2}$$

$$\textsc{Case}\quad \frac{\begin{array}{c} E, \Gamma \vdash t : \tau_1 \\ \forall i \quad E, \Gamma \vdash c_i : \tau_1 \rightarrow \tau_2 \end{array}}{E, \Gamma \vdash \mathsf{case}\, t \,\mathsf{of}\, c_1 \ldots c_n : \tau_2}$$

$$\textsc{Forall}\quad \frac{E, \Gamma \vdash t : \tau \qquad \bar{\alpha} \# \operatorname{ftv}(E, \Gamma)}{E, \Gamma \vdash \forall \bar{\alpha}.t : \forall \bar{\alpha}.\tau}$$

$$\textsc{Annot}\quad \frac{E, \Gamma \vdash t : \tau \qquad \theta \preceq \tau}{E, \Gamma \vdash (t : \theta) : \tau}$$

$$\textsc{Coerce}\quad \frac{E, \Gamma \vdash t : \tau_1 \qquad \kappa \preceq (\tau_1 \triangleright \tau_2) \qquad E \Vdash \kappa}{E, \Gamma \vdash (t : \kappa) : \tau_2}$$

$$\textsc{Conv}\quad \frac{E, \Gamma \vdash t : \tau_1 \qquad E \Vdash \tau_1 = \tau_2}{E, \Gamma \vdash t : \tau_2}$$

$$\textsc{Gen}\quad \frac{E, \Gamma \vdash t : \tau \qquad \bar{\alpha} \# \operatorname{ftv}(E, \Gamma, t)}{E, \Gamma \vdash t : \forall \bar{\alpha}.\tau}$$

$$\textsc{Inst}\quad \frac{E, \Gamma \vdash t : \sigma \qquad \sigma \preceq \tau}{E, \Gamma \vdash t : \tau}$$

$$\textsc{Clause}\quad \frac{\begin{array}{c} p : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma\Gamma' \vdash t : \tau_2 \\ \bar{\beta} \# \operatorname{ftv}(E, \Gamma, \tau_2) \end{array}}{E, \Gamma \vdash p.t : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \rightarrow \tau_2}$$

$$\textsc{Pat}\quad \frac{K \preceq \forall \bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon\, \bar{\tau}_1\, \bar{\tau} \qquad \bar{\beta} \# \operatorname{ftv}(\bar{\tau}_1, \bar{\tau}_2)}{K\, \bar{\beta}\, x_1 \ldots x_n : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \vdash (\bar{\beta}, \bar{\tau}_2 = \bar{\tau}, (x_1 : \tau_1; \ldots; x_n : \tau_n))}$$

Figure 2. **ML** with **g**eneralized algebraic data types in **i**mplicit style (MLGI)

ε is the algebraic data type constructor that K is associated with. The rule is simple but subtle: the main point is that ordinary and generalized type parameters are dealt with in different ways. Let the type scheme associated with K be of the form

$$K :: \forall \bar{\alpha}\bar{\beta}. \star \times \ldots \times \star \rightarrow \varepsilon\, \bar{\alpha}\, \bar{\tau}$$

(Note that we choose the type variables $\bar{\beta}$ that appear in this type scheme to be the same as the type variables $\bar{\beta}$ that appear in the pattern. Furthermore, here and elsewhere in the paper, every occurrence of $\star$ stands for a distinct anonymous metavariable. We exploit this convention to avoid assigning explicit names to entities that we are not interested in.) PAT's first premise is somewhat different:

$$K \preceq \forall \bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon\, \bar{\tau}_1\, \bar{\tau}$$

That is, we take an instance of the type scheme associated with K by substituting the actual ordinary type parameters $\bar{\tau}_1$ for the formal ordinary type parameters $\bar{\alpha}$. (This leaves the generalized type parameters $\bar{\tau}$ unaffected, since their free type variables form a subset of $\bar{\beta}$.) This substitution determines the types $\tau_1, \ldots, \tau_n$. We are now ready to read PAT's conclusion: the body of the clause guarded by this pattern should be typechecked in the scope of the rigid type variables $\bar{\beta}$, under the assumption that the equations $\bar{\tau}_2 = \bar{\tau}$ hold, and under the assumption that every x_i has type τ_i. The equations $\bar{\tau}_2 = \bar{\tau}$ are obtained by confronting the generalized type parameters found in the scrutinee's type, namely $\bar{\tau}_2$, with those found in the definition of K, namely $\bar{\tau}$.

When ε is an ordinary algebraic data type, $\bar{\beta}$ is empty, and there are no generalized type parameters. No new equations appear. In that case, CLAUSE degenerates to the standard rule for pattern matching in Hindley and Milner's type system.

Soundness A closed term, or *program*, is well-typed if it admits a type under an empty equation system and in the empty environment. Programs can be given a call-by-name or call-by-value semantics using operational or denotational techniques; this defines what it means for a program to "go wrong."

Claim 3.1 (Soundness for MLGI) *Well-typed MLGI programs do not go wrong.* ◇

4. A type system with explicit annotations

We now define a type system, known as MLGX, where the difficulties associated with generalized algebraic data types are avoided thanks to mandatory type annotations. The idea is simple. First, we forbid implicit type conversions, so the only way of exploiting E is now via explicit coercions. Second, we require every case scrutinee to carry a type annotation, so that it becomes easy to determine E. In short, MLGX could be described as a type system that marries traditional *type inference* for Hindley and Milner's type system and *type checking* for generalized algebraic data types.

Presentation MLGX is defined in Figure 3. Most of the rules are shared with MLGI and are not repeated.

As announced earlier, implicit type conversions are disallowed in MLGX. That is, rule CONV is suppressed, so that COERCE must be used instead. We maintain the invariant that *the equation system is rigid*, that is, all of the type variables that appear within E can be interpreted as universally bound. As a result, in terms of type inference, COERCE's last premise, $E \Vdash \kappa$, now means *check that E implies the validity of κ*, rather than *solve for the flexible type variables within E so that E implies the validity of κ*. This is the key idea that drives the design of MLGX.

We do still allow implicit conversions in one specific situation, namely, when E is false. Indeed, rule X-FALSE states that, under such an inconsistent assumption, every term t admits every type τ. This is convenient, because it allows the local type inference algorithms described later in the paper to give up as soon they find that E is inconsistent. Although X-FALSE is not a syntax-directed rule, its presence does not give rise to combinatorial choices.

Rule CASE is suppressed and replaced with X-CASE. In the new rule, the term t must carry an explicit type annotation θ. The first premise passes θ on to ANNOT, thus checking that the type ascribed to t is indeed an instance of θ. The second premise passes θ on to X-CLAUSE, where it is exploited to determine which new equations arise inside the clause.

We define the unannotated case construct $\mathsf{case}\, t \,\mathsf{of}\, c_1 \ldots c_n$ as syntactic sugar for $\mathsf{case}\, (t : \exists \bar{\gamma}_1 \bar{\gamma}_2.\varepsilon\, \bar{\gamma}_1\, \bar{\gamma}_2) \,\mathsf{of}\, c_1 \ldots c_n$, where the appropriate type constructor ε is determined by examining the

X-False

$$\frac{\mathrm{fv}(t) \subseteq \mathrm{dom}(\Gamma)}{\mathsf{false}, \Gamma \vdash t : \sigma}$$

X-Case

$$\frac{E, \Gamma \vdash (t : \theta) : \tau_1 \qquad \forall i \quad E, \Gamma \vdash (p_i : \theta).t_i : \tau_1 \to \tau_2}{E, \Gamma \vdash \mathsf{case}\,(t : \theta)\ \mathsf{of}\ p_1.t_1 \ldots p_n.t_n : \tau_2}$$

X-Clause

$$\frac{p : \varepsilon\,\bar{\tau}_1\,\bar{\tau}_2' \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma\Gamma' \vdash t : \tau_2 \qquad \bar{\beta} \,\#\, \mathrm{ftv}(E, \Gamma, \tau_2) \qquad \bar{\gamma} \,\#\, \mathrm{ftv}(E, \Gamma, t, \tau_2)}{E, \Gamma \vdash (p : \exists\bar{\gamma}.\varepsilon \star \bar{\tau}_2').t : \varepsilon\,\bar{\tau}_1 \star \to \tau_2}$$

All of the rules that define MLGI, except Conv, Case, and Clause, repeated here.

Figure 3. **ML** with **g**eneralized algebraic data types in e**x**plicit style (MLGX)

patterns that guard the clauses $c_1 \ldots c_n$. Thanks to this convention, legacy programs that do not exploit generalized algebraic data types need not be annotated at all: that is, MLGX is a conservative extension of ML.

Rule Clause is suppressed and replaced with X-Clause. The key change is in the first premise. In Clause, the generalized type parameters $\bar{\tau}_2$, *found in the type of the scrutinee*, are used to determine which new equations appear. In terms of type inference, this is problematic, since the types $\bar{\tau}_2$ are initially unknown and have to be inferred. For this reason, in X-Clause, these types are disregarded—which we emphasize by writing $\star$ instead of $\bar{\tau}_2$ in the rule's conclusion. Instead, the generalized type parameters $\bar{\tau}_2'$, *found in the explicit type annotation*, are used to determine which new equations appear. In terms of type inference, this is good—no guessing is involved.

The types $\bar{\tau}_2'$ contain occurrences of the type variables $\bar{\gamma}$. As a result, so does the equation system E'. These type variables stand for yet unknown types, so they must be considered abstract when typechecking t, that is, in the second premise. This is guaranteed by the last side condition. Thus, the invariant that the equation system is rigid is maintained in the second premise.

Determining which new equations arise by relying on a possibly incomplete type annotation, as in MLGX, instead of on the actual type of the scrutinee, as in MLGI, entails a loss of information. Indeed, $\bar{\tau}_2$ is a possibly strict instance of $\exists\bar{\gamma}.\bar{\tau}_2'$, so the new equations obtained by relying on $\bar{\tau}_2'$ are implied by, but possibly weaker than, those that would be obtained by relying on $\bar{\tau}_2$. As a result, the equations available in an MLGX type derivation are in general weaker than those available in an analogous MLGI type derivation. On the other hand, when the type annotations are sufficiently specific, that is, when $\bar{\gamma} \,\#\, \mathrm{ftv}(\bar{\tau}_2')$ holds, then the vectors $\bar{\tau}_2$ and $\bar{\tau}_2'$ coincide, so the equations available in MLGX are identical to those available in MLGI.

Soundness and completeness It is straightforward to show that every well-typed MLGX program is a well-typed MLGI program. In combination with Claim 3.1, this implies that MLGX is sound, that is, well-typed MLGX programs do not go wrong.

Theorem 4.1 (Soundness for MLGX) *If $E, \Gamma \vdash t : \sigma$ holds in MLGX, then it holds in MLGI as well.* ◇

It is also clear that every well-typed MLGI program can be turned into a well-typed MLGX program by adding enough type annotations. In short, it is sufficient to replace every implicit type conversion with an explicit type coercion, to add an explicit type annotation to every case expression over a generalized algebraic data type, and to explicitly bind the type variables that appear inside these new annotations.

Theorem 4.2 (Completeness with assistance for MLGX) *Define* equivalence up to annotations, *written $\equiv$, as the reflexive and congruence closure of the following axioms:*

$$\frac{t \equiv t' \qquad \bar{\alpha} \,\#\, \mathrm{ftv}(t)}{t \equiv \forall\bar{\alpha}.t'} \qquad \frac{t \equiv t'}{t \equiv (t' : \theta)} \qquad \frac{t \equiv t'}{t \equiv (t' : \kappa)}$$

If $E, \Gamma \vdash t : \sigma$ holds in MLGI, then there exists a term t' such that $t \equiv t'$ holds and $E, \Gamma \vdash t' : \sigma$ holds in MLGX. ◇

Example Here is the $eval$ example of §1, augmented with enough explicit type annotations to make it a well-typed MLGX term:

$$\begin{array}{l}
\mu(eval : \forall\alpha.term\ \alpha \to \alpha).\forall\alpha.\lambda t.\\
\quad \mathsf{case}\,(t : term\ \alpha)\ \mathsf{of}\\
\qquad \mid Lit\ i \to (i : (int \triangleright \alpha))\\
\qquad \mid Inc\ t \to (eval\ t + 1 : (int \triangleright \alpha))\\
\qquad \mid IsZ\ t \to (eval\ t = 0 : (bool \triangleright \alpha))\\
\qquad \mid If\ b\ t\ e \to \mathsf{if}\ eval\ b\ \mathsf{then}\ eval\ t\ \mathsf{else}\ eval\ e\\
\qquad \mid Pair\ \beta_1\ \beta_2\ a\ b \to ((eval\ a, eval\ b) : (\beta_1 \times \beta_2 \triangleright \alpha))\\
\qquad \mid Fst\ \beta_2\ t \to fst\ (eval\ t)\\
\qquad \mid Snd\ \beta_1\ t \to snd\ (eval\ t)
\end{array}$$

The first change is the explicit introduction of the type variable α, on the first line. This is required in order to allow references to α in the type annotations that follow. In a surface language, one could add sugar and adopt the convention that the first occurrence of $\forall\alpha$ binds α not only in the type $term\ \alpha \to \alpha$, but also in the term that follows. Glasgow Haskell, for instance, allows this. In fact, this turns out to be helpful for local type inference, so we introduce this convention (and write $\mu^\star$ instead of μ) in §6 and §7.

The case scrutinee t now carries the explicit type annotation $term\ \alpha$, so as to allow X-Case and X-Clause to determine which type equations arise within each clause.

In the Lit clause, the variable i has type int, which we want to convert to α, so an explicit type coercion is required. Analogous coercions appear in the Inc, IsZ, and $Pair$ clauses. In the $Pair$ clause, α is known to be equal to a product type $\beta_1 \times \beta_2$.

Type inference for MLGX Type inference for MLGX is analogous to type inference for an extension of Milner and Mycroft's type system with explicit type annotations. There exists a standard reduction of the latter to solving constraints, that is, to first-order unification under a mixed prefix [18, 17]. It can be presented as a transformation, written $(\!|\cdot|\!)$ and known as *constraint generation*, that maps a candidate judgement $E, \Gamma \vdash t : \tau$ to a constraint. In short, Γ and τ can be thought of as an "expected typing" for the term t, and the constraint $(\!| E, \Gamma \vdash t : \tau |\!)$ expresses the requirements that the type variables in $\mathrm{ftv}(\Gamma, t, \tau)$ must meet for this typing to become valid.

We do not repeat the standard reduction of type inference to constraint solving. Instead, we show how it is extended to cover the new constructs in MLGX. This requires four constraint generation rules (Figure 4). For clarity, the side conditions that require all type variables to be chosen "sufficiently fresh" are omitted.

Rule G-False states that, if the equation system E collected so far is inconsistent, then it is permissible to stop examining the term t and simply produce the constraint true. This rule is not syntax-directed: it is applied in preference to all other constraint generation rules.

Rule G-Coerce first checks that E entails the validity of the coercion $\exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)$. This check is easy to implement: provided $\bar{\gamma} \,\#\, \mathrm{ftv}(E)$ holds, it is equivalent to computing a most general unifier of E and checking that it is also a unifier of $\tau_1 = \tau_2$.

$$
\begin{array}{ll}
 & \text{G-F}\small{\text{ALSE}} \\
 & \llparenthesis\, \mathsf{false}, \Gamma \vdash t : \tau \,\rrparenthesis \\
= & \mathsf{true} \\[1em]
 & \text{G-C}\small{\text{OERCE}} \\
 & \llparenthesis\, E, \Gamma \vdash (t : \exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)) : \tau \,\rrparenthesis \\
= & \exists\bar{\gamma}.(\llparenthesis\, E, \Gamma \vdash t : \tau_1 \,\rrparenthesis \wedge \tau_2 = \tau) \\
\text{if} & E \Vdash \forall\bar{\gamma}.\tau_1 = \tau_2 \\[1em]
 & \text{G-C}\small{\text{ASE}} \\
 & \llparenthesis\, E, \Gamma \vdash \mathsf{case}\,(t : \theta)\,\mathsf{of}\; p_1.t_1 \ldots p_n.t_n : \tau \,\rrparenthesis \\
= & \exists\gamma.(\llparenthesis\, E, \Gamma \vdash (t : \theta) : \gamma \,\rrparenthesis \wedge \\
 & \qquad \bigwedge_i \llparenthesis\, E, \Gamma \vdash (p_i : \theta).t_i : \gamma \rightarrow \tau \,\rrparenthesis) \\[1em]
 & \text{G-C}\small{\text{LAUSE}} \\
 & \llparenthesis\, E, \Gamma \vdash (K\,\bar{\beta}\,x_1 \ldots x_n : \exists\bar{\gamma}.\varepsilon \star \bar{\tau}'_2).t : \tau' \rightarrow \tau \,\rrparenthesis \\
= & \exists\bar{\alpha}.(\exists\bar{\gamma}'_2.(\tau' = \varepsilon\,\bar{\alpha}\,\bar{\gamma}'_2) \wedge \\
 & \qquad \forall\bar{\beta}\bar{\gamma}.\llparenthesis\, E \wedge \bar{\tau}'_2 = \bar{\tau}, \Gamma; x_1 : \tau_1; \ldots; x_n : \tau_n \vdash t : \tau \,\rrparenthesis) \\
\text{if} & K :: \forall\bar{\alpha}\bar{\beta}.\tau_1 \times \ldots \times \tau_n \rightarrow \varepsilon\,\bar{\alpha}\,\bar{\tau}
\end{array}
$$

Figure 4. Constraint generation for MLGX (excerpt)

If the check fails, constraint generation fails as well. If the check succeeds, then constraint generation proceeds. The term t is now expected to have type τ_1, but the type that is made visible to the outside is τ_2. The constraint that is produced is identical to what would be obtained by applying a function of type $\forall\bar{\gamma}.\tau_1 \rightarrow \tau_2$ to t.

G-CASE is simple. The type annotation θ is transmitted down both sides. The flexible type variable γ stands for the unknown type of the scrutinee.

G-CLAUSE paraphrases X-CLAUSE. The type scheme assigned to K is looked up. The constraint first binds the ordinary type parameters $\bar{\alpha}$ *existentially*: they are inferred. Then comes a conjunction. The conjunct $\exists\bar{\gamma}'_2.(\tau' = \varepsilon\,\bar{\alpha}\,\bar{\gamma}'_2)$ determines appropriate values for $\bar{\alpha}$ by equating the scrutinee's expected type τ' with $\varepsilon\,\bar{\alpha}\,\bar{\gamma}'_2$. The type variables $\bar{\gamma}'_2$ do not occur elsewhere: they serve only to discard the generalized type parameters. In the second conjunct, E is augmented with the equations $\bar{\tau}'_2 = \bar{\tau}$, obtained by confronting the type annotation with the type scheme assigned to K. The type variables that occur within these equations are (a subset of) $\bar{\beta}\bar{\gamma}$. These type variables are *universally* quantified up front, maintaining the invariant that the equation system is rigid. The typing environment Γ is extended with appropriate bindings, and a constraint that requires the clause body t to have type τ is produced.

It is worth noting that the expected type τ' of the scrutinee does not influence the new equations that arise. As in MLGX, these are determined solely by exploiting the information $\bar{\tau}'_2$ found in the explicit type annotation.

The constraint generation rules can be proven sound and complete with respect to the specification of MLGX. The extra proof cases that must be added to the standard proof are straightforward.

Theorem 4.3 (Type inference for MLGX) *Let ϕ be a type substitution whose domain is disjoint with $\mathrm{ftv}(E, t)$. Then, ϕ is a unifier of $\llparenthesis\, E, \Gamma \vdash t : \tau \,\rrparenthesis$ if and only if $E, \phi(\Gamma) \vdash t : \phi(\tau)$ holds in MLGX.* ◇

This means, in particular, that MLGX has principal type schemes, like Hindley and Milner's type system.

Comparison with wobbly types The flexible type variables in our type annotations (and in our shapes, see §5) play exactly the same rôle as wobbly types in Peyton Jones *et al.*'s proposal [12]. They write: "*when performing match-unification* [that is, when determining which new equations arise, in this paper's terminology], *we make no use of information inside wobbly types.*" In our presentation, this goes without saying: flexible type variables are type variables—that is, they stand for unknown types—so of course they carry no information!

Peyton Jones *et al.* further write: "*This simple intuition is surprisingly tricky to formalise.*" Indeed, their formalization requires excising the wobbly types, performing unification, and re-inserting the excised types back into the result. Technically, this involves computing, composing, and restricting type substitutions—tricky business. Here, no such tricks are necessary, because the unwanted information simply isn't there in the first place. In other words, in MLGX, the building of equation systems E, which are used to validate coercions, is *entirely separate* from the production (and later solving) of constraints C, which are used to perform traditional type inference in the style of Hindley and Milner. In Peyton Jones *et al.*'s presentation, the two are mixed, at least in appearance. (They are also mixed with a form of local type inference, which we discuss later on.)

There is in fact a slight difference between the flexible type variables in our type annotations and Peyton Jones *et al.*'s wobbly types: a type variable has *identity*, whereas a "wobbly box" doesn't. For instance, the type annotation $\exists\gamma.term\ (\gamma \times \gamma)$ has two holes, but the two have the same identity γ. This annotation is more precise than $\exists\gamma_1\gamma_2.term\ (\gamma_1 \times \gamma_2)$, which appears to correspond to $term\ (\boxed{\tau_1} \times \boxed{\tau_2})$ in Peyton Jones *et al.*'s formalization. We conjecture that, in the Glasgow Haskell implementation, wobbly boxes *do* have an identity—indeed, Peyton Jones *et al.* write: "*wobbly types [...] simply* are *the flexible meta variables that the inference engine already uses.*" Our formalization may well be more faithful with respect to such an implementation.

Towards stratified type inference The strength of MLGX lies in its simplicity and in the fact that it enjoys type inference and principal types in the style of Hindley and Milner. Its design, which marries type inference for the core language with type checking for generalized algebraic data types, appears to be robust, in the sense that no variations have come to mind so far.

Unfortunately, from a user's standpoint, MLGX is not very expressive. In the $eval$ example, it requires a lot of explicit type information. One might say that MLGX does not attempt to do type inference *for* generalized algebraic data types; it only does type inference *in their presence*.

In the case of $eval$, it should not be very hard to guess which explicit type annotations must be added to the program. The signature given at the μ binder specifies that $eval$ has type $term\ \alpha \rightarrow \alpha$, so it is "clear" that the variable t should have type $term\ \alpha$ and that every branch of the case construct should have type α. The former remark allows inserting the type annotation $(t : term\ \alpha)$. The latter, applied to the Lit branch and combined with the fact that i "clearly" has type int, allows inserting the coercion $(int \triangleright \alpha)$. It is similarly "clear" which coercions should be added to the other branches. Isn't it a shame for a type inference system to be unable to take advantage of information that is so "clearly" apparent in the program?

Our answer is to design a separate transformation that discovers the explicit type information in the original program, propagates it, and exploits it to produce a transformed program that contains *more* explicit type annotations and coercions. The transformed program can then be passed on to MLGX's type inference algorithm. We refer to this two-stage approach as *stratified type inference*.

Contrary to what one might think, propagating explicit type information is not easy—if it were, we would have built this feature into MLGX in the first place. Many design choices soon arise, and most designs are incomplete, that is, reject programs that are valid in MLGI. From this remark, we draw two conclusions:

$$
\begin{array}{lll}
\bot & = & \gamma.\gamma \\
(\bar{\gamma}_1.\tau_1) \to (\bar{\gamma}_2.\tau_2) & = & \bar{\gamma}_1\bar{\gamma}_2.\tau_1 \to \tau_2 \\
 & & \bar{\gamma}_1 \# \operatorname{ftv}(\tau_2), \bar{\gamma}_2 \# \operatorname{ftv}(\tau_1) \\
\mathcal{D}(\bot) & = & \bot \\
\mathcal{D}(\bar{\gamma}.\tau_1 \to \star) & = & \bar{\gamma}.\tau_1 \\
\mathcal{C}(\bot) & = & \bot \\
\mathcal{C}(\bar{\gamma}.\star \to \tau_2) & = & \bar{\gamma}.\tau_2
\end{array}
$$

Figure 5. Basic operations on shapes

- it is worth *separating* the robust, well-understood back-end (MLGX) from the more *ad hoc* front-end.
- because its design is *ad hoc*, the front-end should be *simple* and *predictable*.

Ad hoc methods of type inference have been studied, for instance, by Pierce and Turner [13], who introduce *local* type inference as a means of achieving simplicity and predictability. They write: "*missing annotations [should be] recovered using only information from adjacent nodes in the syntax tree, without long-distance constraints such as unification variables*." Other, more recent type inference systems, such as Peyton Jones and Shields' approach to introducing arbitrary-rank predicate polymorphism into Haskell [11], or Peyton Jones *et al.*'s "wobbly types" proposal [12], also use forms of local type inference.

In the following, we design two such local type inference systems. The first design (§6) closely follows Peyton Jones *et al.*'s "wobbly types" proposal, with a few changes and improvements, and explains how "wobbly types" are recast in terms of stratified type inference. The second design (§7) addresses improves in accuracy over the previous one. Neither is definitive: many more could be imagined.

Because the program produced by the front-end is submitted to MLGX, the local type inference algorithm has no obligation of rejecting invalid programs, or of fully determining the type of every expression. Instead, it is perfectly fine for it to manipulate *incomplete* (that is, *approximate*) type information, and to produce new type annotations and coercions only where enough information is available. For this reason, both of our designs are based on *shapes*, that is, approximate type schemes. Shapes, introduced next (§5), can share *rigid* type variables, but cannot share *flexible* ("*unification*") variables. This is why shape-based algorithms can be deemed "local."

5. Shapes

Shapes are defined by

$$s ::= \bar{\gamma}.\tau$$

where the type variables $\bar{\gamma}$ are bound within the type τ. We refer to the type variables $\bar{\gamma}$ as *flexible*. A flexible type variable represents a type that is either unknown (so the shape $\gamma.\gamma \to \gamma$ would adequately describe a value of type, say, $int \to int$) or a polymorphic type variable (so the shape $\gamma.\gamma \to \gamma$ also describes the polymorphic identity function, whose type scheme is $\forall\gamma.\gamma \to \gamma$). Shapes are not necessarily closed. Their free type variables are interpreted as *rigid*—that is, they are type variables that have been explicitly universally quantified by the programmer. For instance, the shape $\gamma.\alpha \times \gamma$ describes a pair whose first component has type α, where the rigid type variable α was introduced by the programmer, and whose second component has unknown type.

Shapes bear close resemblance to simple type annotations. We often implicitly convert the simple type annotation $\exists\bar{\gamma}.\tau$ into the shape $\bar{\gamma}.\tau$. We also convert polymorphic type annotations $\exists\bar{\gamma}.\forall\bar{\alpha}.\tau$ into shapes, but that is done explicitly.

Figure 5 introduces a few basic operations on shapes. The *bottom* shape $\gamma.\gamma$ is written $\bot$. This shape carries no information whatsoever. Out of two arbitrary shapes s_1 and s_2, one can build a *function* shape $s_1 \to s_2$. Conversely, out of an arbitrary shape s, one can attempt to extract *domain* and *codomain* shapes $\mathcal{D}(s)$ and $\mathcal{C}(s)$. These operations are defined if s is the bottom shape or a function shape, and undefined otherwise.

5.1 Ordering shapes

Shapes are equipped with a standard instantiation ordering, defined by the single axiom

$$\frac{\bar{\gamma}_2 \# \operatorname{ftv}(\bar{\gamma}_1.\tau_1)}{\bar{\gamma}_1.\tau_1 \preceq \bar{\gamma}_2.[\bar{\gamma}_1 \mapsto \bar{\tau}_1]\tau_1}$$

Example 5.1 We have $(\gamma_1.\alpha \times \gamma_1) \preceq (\gamma_2.\alpha \times (\alpha \to \gamma_2))$. ◇

This confers a rich structure to the set of shapes [6, chapter 5]:

Theorem 5.2 (Huet) *Shapes form a well-founded lower semi-lattice, whose least element is $\bot$.* ◇

This result implies that any finite set of shapes that admits an upper bound must in fact admit a least upper bound. The least upper bound of two shapes s_1 and s_2 is written $s_1 \sqcup s_2$, when it exists, and can be computed via first-order unification.

Example 5.3 Recall that $int \to \bot$ stands for $\gamma.int \to \gamma$. Then, it is easy to check that $(\gamma.\gamma \to \gamma) \sqcup (int \to \bot)$ is $int \to int$. ◇

Shapes do not share flexible type variables, so that no "long-distance" unification takes place during shape inference: this was announced as the key property that makes inference "local." Yet, the least upper bound operation over shapes *does* involve unification. This is important: a local type inference algorithm that did not involve *any* kind of unification would be quite imprecise.

The definition of the ordering can be generalized so as to make it relative to an equation system E. The original definition is recovered when E is `true`.

Definition 5.4 *We write $E \Vdash s_1 \preceq s_2$ if and only if there exists a shape s such that $s_1 \preceq s$ and $E \Vdash s = s_2$ hold. We write $E \Vdash s_1 = s_2$ when $E \Vdash s_1 \preceq s_2$ and $E \Vdash s_2 \preceq s_1$ hold.* ◇

Example 5.5 Let s_1 be $\gamma_1.\alpha \times \gamma_1$ and s_2 be $\gamma_2.int \times (\alpha \to \gamma_2)$. Then, $s_1 \preceq s_2$ does not hold, because the rigid type variable α cannot be instantiated to int, but $\alpha = int \Vdash s_1 \preceq s_2$ does. ◇

5.2 Normalization

Shapes that are syntactically incompatible (that is, do not have a common upper bound) should sometimes be viewed as compatible. For instance, let E consist of the equation $\alpha = \beta_1 \to \beta_2$. If some expression is found to have both shape α and shape $\gamma.\beta_1 \to \gamma$, then a sensible shape inference algorithm should not fail, nor should it conclude that this expression has shape $\bot$. Instead, the two shapes should be successfully combined, yielding $\beta_1 \to \beta_2$, as opposed to α, because the latter is *more informative*: it exposes the fact that the expression can only evaluate to a function. (If the domain operator $\mathcal{D}(\cdot)$ is later applied to $\beta_1 \to \beta_2$, it will successfully yield β_1, whereas applying $\mathcal{D}(\cdot)$ to α would fail.)

To extract as much information as possible out of a shape, we *normalize* it with respect to E. When E contains an equation $\alpha = \tau$, where τ is not a type variable, then normalization rewrites α into τ.

The definition of normalization is simple, but introduces a measure of arbitrariness into the system: indeed, when E relates two type variables α and β, a choice has to be made between rewriting α to β or vice versa. This choice influences how the program is transformed by the front-end, which means that it also has an impact on the type error messages produced by the back-end (MLGX) when the transformed program doesn't typecheck.

In the following, we assume that E isn't equivalent to false. Because we interpret constraints in a finite tree model, this implies that E is acyclic. This hypothesis guarantees that normalization, as defined below, terminates. It is in fact possible to deal with cyclic equation systems, and doing so is indeed necessary when the type system features equirecursive types. We omit this discussion.

Definition 5.6 *Let $<$ be a fixed, arbitrarily chosen total ordering over $\mathrm{ftv}(E)$. Then, the rewriting relation $\leadsto_E$ on types is generated by the axioms:*

$$\begin{array}{llll} \alpha & \leadsto_E & \alpha' & \textit{if } E \Vdash \alpha = \alpha' \textit{ and } \alpha' < \alpha \\ \alpha & \leadsto_E & \varepsilon\,\bar{\tau}_1\,\bar{\tau}_2 & \textit{if } E \Vdash \alpha = \varepsilon\,\bar{\tau}_1\,\bar{\tau}_2 \\ \alpha & \leadsto_E & \tau_1 \rightarrow \tau_2 & \textit{if } E \Vdash \alpha = \tau_1 \rightarrow \tau_2 \end{array}$$

This relation is confluent and terminating. We write $\tau\downharpoonleft_E$ for the normal form of the type τ. We write $s\downharpoonleft_E$ for $\bar{\gamma}.(\tau\downharpoonleft_E)$ when s is $\bar{\gamma}.\tau$ and $\bar{\gamma}\,\#\,\mathrm{ftv}(E)$ holds. The notations $\theta\downharpoonleft_E$ and $\varsigma\downharpoonleft_E$ are defined similarly. ◇

In Peyton Jones *et al.*'s proposal [12], normalization is performed by picking an (arbitrary) most general unifier ϕ of E and applying it to the type that should be normalized. This substitution process stops at "wobbly boxes," that is, $\phi(\boxed{\tau})$ is defined as $\boxed{\tau}$. Here, this corresponds to the fact that normalization does not affect flexible type variables.

5.3 Pruning

One problem still hasn't been discussed: can we guarantee that the front-end is *sound*, that is, that the type annotations and coercions inserted by the front-end are *correct* with respect to the programmer's intent?

Assume the original program is well-typed in MLGI (although perhaps not in MLGX, by lack of explicit type information). We certainly cannot expect the transformed program to always be well-typed in MLGX, because that would amount to requiring the front-end to perform *complete* type inference for MLGI. However, we should be able to guarantee that the transformed program is also well-typed in MLGI. Indeed, if that is not the case, then the transformation is counter-productive: it inserts annotations that break the program! We claim that, when in doubt, one should insert no annotations at all, rather than insert incorrect ones.

Achieving soundness requires some care. Imagine that the equation $\alpha = \beta$ is made available within a case branch. Imagine further that this branch has type α. In MLGI, it is also true that this branch has type β. As a result, it is fine to reason with shapes that are correct only "up to E." A shape s implicitly denotes the set of types τ such that $E \Vdash s \preceq \tau$ holds. However, outside of the branch, the equation $\alpha = \beta$ is no longer available, so it *does* make a difference whether the branch is deemed to have shape α or β. That is, *interpreting shapes "up to E" requires extra care when E shrinks*. An arbitrary choice between α and β could produce a transformed program that is ill-typed in MLGI. Instead, one must abandon this unreliable piece of information and report that the branch has shape $\bot$, which certainly is a sound approximation. We refer to this process as *pruning*.

Definition 5.7 *The denotation of s under E is the set of all types τ such that $E \Vdash s \preceq \tau$ holds. The shape obtained by pruning s' with respect to E and E', written $s' \upharpoonleft_{E,E'}$, is the least upper bound of the shapes s such that $s \preceq s'$ holds and the denotation of s under E contains that of s' under $E \wedge E'$.* ◇

Pruning is performed at the boundary between an equation system E and a richer equation system $E \wedge E'$. A shape s' is given. The denotation of s' under E is always a subset of its denotation under $E \wedge E'$. If we are unlucky, it is a strict subset, which means that the denotation of s' silently changes when we move from $E \wedge E'$ back to E. Pruning s' consists in discarding information in order to avoid this phenomenon, that is, in determining the most precise shape s such that $s \preceq s'$ holds and the denotation of s under E contains that of s' under $E \wedge E'$.

Example 5.8 Let E be true and E' be $\alpha = \beta_1 \times \beta_2$. Then, pruning the shape $s' = \gamma.\alpha \rightarrow \gamma$ with respect to E and E' yields the shape $s = \gamma_1\gamma_2.\gamma_1 \rightarrow \gamma_2$. Indeed, the denotation of s' under $E \wedge E'$ contains all types of the form $(\beta_1 \times \beta_2) \rightarrow \tau$, which its denotation under E does not contain, so the sub-term α must be pruned. The denotation of s under E does contain all such types.

Of course, pruning cannot guarantee soundness unless we have exact knowledge of the current equation system. Indeed, pruning with respect to an under-approximation of E and E' might lead to keeping sub-terms that would be discarded when pruning with respect to E and E'. In other words, pruning with respect to under-approximations of the equation systems is just as good as no pruning at all: it is unsound! As a result, in §6 and §7, we insist on determining the current equation system with precision. An opposite decision is made by Peyton Jones *et al.* [12]. We compare the two alternatives in §6.

6. The shape inference system W

The shape toolbox developed in §5 provides the building blocks to develop a local type inference (or *shape inference*) algorithm that can be placed in front of MLGX in a stratified type inference system. In fact, it is easy to think of *many* such algorithms that differ in how information is propagated through the abstract syntax tree. Here, in §6, we describe one such algorithm, which we call W. It is intended to emulate Peyton Jones *et al.*'s "wobbly types" proposal [12], with a few differences. Next, in §7, we describe another, more accurate algorithm.

Following Peyton Jones *et al.*, W is *bidirectional*: it operates either in *inference mode* or in *checking mode*. An inference mode judgement takes the form $E, \Gamma \vdash t \Uparrow s \leadsto t'$. Its inputs are the equation system E, the environment Γ, which maps variables to shapes, and the term t. Its outputs are the inferred shape s and the transformed term t'. A checking mode judgement takes the form $E, \Gamma \vdash t \Downarrow s \leadsto t'$. It is analogous to an inference mode judgement, except the expected shape s is now an input. The definition of the judgments appears in Figure 6.

An invariant is that, in either mode, the shape s is normalized with respect to E. As explained in §5.2, normalizing shapes is required in order to avoid "silly" unification errors. When we write $s_1 \sqcup s_2$, we ensure that s_1 and s_2 are both normalized. When we write $\mathcal{D}(s)$ or $\mathcal{C}(s)$, we ensure that s is normalized. The shapes that appear in Γ are *not* necessarily normalized.

In general, the transformed term t' is identical to t, except (i) all explicit type annotations are normalized, (ii) new type annotations are inserted around case scrutinees, and (iii) type coercions are inserted at uses of variables and around some case clauses. Normalizing type annotations can be viewed as a heuristic that attempts to increase the likelihood that the transformed term is well-typed in MLGX.

We suppress the construct $\mu(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau).t$ and replace it with the new construct $\mu^\star(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau).t$, which is identical, except the type variables $\bar{\alpha}$ are considered bound not only in τ, but also in t. In other words, the new construct can be viewed as syntactic sugar for $\mu(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau).\forall\bar{\alpha}.t$. This is exploited in the formulation of the rules FIX-⇑ and FIX-⇓.

We disallow explicit type coercions in *source* terms, because they are redundant with type annotations. Indeed, for $\exists\bar{\gamma}.(\tau_1 \triangleright \tau_2)$

False-⇑
$$\mathsf{false}, \Gamma \vdash t \Uparrow \bot \leadsto t$$

False-⇓
$$\mathsf{false}, \Gamma \vdash t \Downarrow s \leadsto t$$

Var-⇑
$$\frac{(x : s) \in \Gamma}{E, \Gamma \vdash x \Uparrow s \lfloor_E \leadsto (x \downarrow_E s)}$$

Var-⇓
$$\frac{(x : s) \in \Gamma}{E, \Gamma \vdash x \Downarrow s' \leadsto (x \downarrow_E s)}$$

Lam-⇑
$$\frac{E, \Gamma; x : \theta \lfloor_E \vdash t \Uparrow s \leadsto t'}{E, \Gamma \vdash \lambda(x : \theta).t \Uparrow (\theta \lfloor_E \to s) \leadsto \lambda(x : \theta \lfloor_E).t'}$$

Lam-⇓
$$\frac{\begin{array}{c} s' = s \sqcup (\theta \lfloor_E \to \bot) \\ E, \Gamma; x : \mathcal{D}(s') \vdash t \Downarrow \mathcal{C}(s') \leadsto t' \end{array}}{E, \Gamma \vdash \lambda(x : \theta).t \Downarrow s \leadsto \lambda(x : \theta \lfloor_E).t'}$$

App-⇑
$$\frac{\begin{array}{c} E, \Gamma \vdash t_1 \Uparrow s \leadsto t_1' \\ E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s) \leadsto t_2' \end{array}}{E, \Gamma \vdash t_1\ t_2 \Uparrow \mathcal{C}(s) \leadsto t_1'\ t_2'}$$

App-⇓
$$\frac{\begin{array}{c} E, \Gamma \vdash t_1 \Uparrow s_1 \leadsto t_1' \\ E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s_1 \sqcup (\bot \to s)) \leadsto t_2' \end{array}}{E, \Gamma \vdash t_1\ t_2 \Downarrow s \leadsto t_1'\ t_2'}$$

Let-⇕
$$\frac{\begin{array}{c} E, \Gamma \vdash t_1 \Uparrow s_1 \leadsto t_1' \\ E, \Gamma; x : s_1 \vdash t_2 \Updownarrow s_2 \leadsto t_2' \end{array}}{E, \Gamma \vdash \mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2 \Updownarrow s_2 \leadsto \mathsf{let}\, x = t_1' \,\mathsf{in}\, t_2'}$$

Fix-⇑
$$\frac{\begin{array}{c} \bar{\alpha} \# \operatorname{ftv}(E, \Gamma) \\ E, \Gamma; x : \bar{\gamma}\bar{\alpha}.\tau \lfloor_E \vdash t \Downarrow \bar{\gamma}.\tau \lfloor_E \leadsto t' \end{array}}{E, \Gamma \vdash \mu^\star(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau).t \Uparrow \bar{\gamma}\bar{\alpha}.\tau \lfloor_E \leadsto \mu^\star(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau \lfloor_E).t'}$$

Fix-⇓
$$\frac{\begin{array}{c} \bar{\alpha} \# \operatorname{ftv}(E, \Gamma, s) \\ E, \Gamma; x : (\bar{\gamma}\bar{\alpha}.\tau \lfloor_E \sqcup s) \vdash t \Downarrow (\bar{\gamma}.\tau \lfloor_E \sqcup s) \leadsto t' \end{array}}{E, \Gamma \vdash \mu^\star(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau).t \Downarrow s \leadsto \mu^\star(x : \exists\bar{\gamma}.\forall\bar{\alpha}.\tau \lfloor_E).t'}$$

Cstr-⇑
$$\frac{K :: s \qquad \forall i \quad E, \Gamma \vdash t_i \Uparrow s_i \leadsto t_i'}{\begin{array}{l} E, \Gamma \vdash K\, t_1 \ldots t_n \Uparrow \mathcal{C}(s \sqcup (s_1 \times \ldots \times s_n \to \bot)) \leadsto \\ \qquad K\, t_1' \ldots t_n' \end{array}}$$

Cstr-⇓
$$\frac{K :: s \qquad \forall i \quad E, \Gamma \vdash t_i \Downarrow \mathcal{D}_i(s \sqcup (\bot \times \ldots \times \bot \to s')) \leadsto t_i'}{E, \Gamma \vdash K\, t_1 \ldots t_n \Downarrow s' \leadsto K\, t_1' \ldots t_n'}$$

Case-⇑
$$\frac{E, \Gamma \vdash t \Uparrow s' \leadsto t' \qquad \forall i \quad E, \Gamma \vdash (p_i : s').t_i \Uparrow s_i \leadsto p_i.t_i'}{\begin{array}{l} E, \Gamma \vdash \mathsf{case}\, t \,\mathsf{of}\, p_1.t_1 \ldots p_n.t_n \Uparrow \sqcup_i s_i \leadsto \\ \qquad \mathsf{case}\, (t' : s') \,\mathsf{of}\, p_1.t_1' \ldots p_n.t_n' \end{array}}$$

Case-⇓
$$\frac{E, \Gamma \vdash t \Uparrow s' \leadsto t' \qquad \forall i \quad E, \Gamma \vdash (p_i : s').t_i \Downarrow s \leadsto p_i.t_i'}{\begin{array}{l} E, \Gamma \vdash \mathsf{case}\, t \,\mathsf{of}\, p_1.t_1 \ldots p_n.t_n \Downarrow s \leadsto \\ \qquad \mathsf{case}\, (t' : s') \,\mathsf{of}\, p_1.t_1' \ldots p_n.t_n' \end{array}}$$

Forall-⇑
$$\frac{\begin{array}{c} \bar{\alpha} \# \operatorname{ftv}(E, \Gamma) \\ E, \Gamma \vdash t \Uparrow s \leadsto t' \end{array}}{E, \Gamma \vdash \forall\bar{\alpha}.t \Uparrow \bar{\alpha}.s \leadsto \forall\bar{\alpha}.t'}$$

Forall-⇓
$$\frac{\begin{array}{c} \bar{\alpha} \# \operatorname{ftv}(E, \Gamma, s) \\ E, \Gamma \vdash t \Downarrow s \leadsto t' \end{array}}{E, \Gamma \vdash \forall\bar{\alpha}.t \Downarrow s \leadsto \forall\bar{\alpha}.t'}$$

Annot-⇑
$$\frac{E, \Gamma \vdash t \Downarrow \theta \lfloor_E \leadsto t'}{\begin{array}{c} E, \Gamma \vdash (t : \theta) \Uparrow \theta \lfloor_E \leadsto \\ (t' : \theta \lfloor_E) \end{array}}$$

Annot-⇓
$$\frac{E, \Gamma \vdash t \Downarrow (\theta \lfloor_E \sqcup s) \leadsto t'}{E, \Gamma \vdash (t : \theta) \Downarrow s \leadsto (t' : \theta \lfloor_E)}$$

Clause-⇑
$$\frac{\begin{array}{c} p : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\bar{\gamma}.\Gamma') \vdash t \Uparrow s \leadsto t' \\ \bar{\beta} \# \operatorname{ftv}(E, \Gamma, s \upharpoonright_{E,E'}) \qquad \bar{\gamma} \# \operatorname{ftv}(E, \Gamma, \bar{\tau}_2, t) \end{array}}{E, \Gamma \vdash (p : \bar{\gamma}.\varepsilon\, \bar{\tau}_1\, \bar{\tau}_2).t \Uparrow s \upharpoonright_{E,E'} \leadsto p.t'}$$

Clause-⇓
$$\frac{\begin{array}{c} p : \varepsilon\, \bar{\tau}_1\, \bar{\tau}_2 \vdash (\bar{\beta}, E', \Gamma') \qquad E \wedge E', \Gamma(\bar{\gamma}.\Gamma') \vdash t \Downarrow s \lfloor_{E \wedge E'} \leadsto t' \\ \bar{\beta} \# \operatorname{ftv}(E, \Gamma, s) \qquad \bar{\gamma} \# \operatorname{ftv}(E, \Gamma, \bar{\tau}_2, t, s) \end{array}}{E, \Gamma \vdash (p : \bar{\gamma}.\varepsilon\, \bar{\tau}_1\, \bar{\tau}_2).t \Downarrow s \leadsto p.(t' \uparrow_{E \wedge E'} s)}$$

Figure 6. The shape inference system W

to be a valid coercion, $E \Vdash \forall\bar{\gamma}.\tau_1 = \tau_2$ must hold, which implies that normalizing τ_1 and τ_2 produces the same result. Because W normalizes all programmer-supplied types, a type coercion in the source term would degenerate to a simple type annotation in the transformed term.

Presentation Rules False-⇑ and False-⇓ state that the transformation stops when E is found to be (equivalent to) false. These rules are not syntax-directed: they are applied in preference to all other rules. This allows us to assume, elsewhere, that E is satisfiable.

Rule Var-⇑ looks up the shape s associated with x in the environment. It produces the inferred shape $s \lfloor_E$, thus satisfying the invariant that the inferred shape is normalized with respect to E. This normalization step corresponds to a type conversion: the type of x, an instance of s, is turned into an instance of $s \lfloor_E$. This must be reflected in the transformed term by inserting an explicit type coercion, so that the MLGX back-end knows what is going on. The rule produces the term $(x \downarrow_E s)$, where $(t \downarrow_E \bar{\gamma}.\tau)$ is defined as syntactic sugar for $(t : \exists\bar{\gamma}.(\tau \triangleright \tau \lfloor_E))$, provided $\bar{\gamma} \# \operatorname{ftv}(E)$ holds. That is, for some value of the flexible type variables $\bar{\gamma}$, to be inferred by MLGX, the type τ is being converted to $\tau \lfloor_E$. Rule Var-⇓ is analogous. The expected shape s' is ignored.

Rule Lam-⇑ extracts the explicit type annotation θ that decorates x, and replaces it with $\theta \lfloor_E$ in the transformed term. Accordingly, the shape environment Γ is extended with the binding $x : \theta \lfloor_E$, and the function's inferred shape is $\theta \lfloor_E \to s$ if t's inferred shape is s. Rule Lam-⇓ is analogous, but combines the expected shape s with the information contained in the type annotation. For instance, if s is $\gamma.\gamma \to \gamma$ and θ is int, then the combination yields $s' = int \to int$, so that the binding $x : int$ is added to the environment and and t is checked with expected shape int.

Following Peyton Jones *et al.*, both App-⇑ and App-⇓ *infer* the function's shape and use this information to *check* the argument's shape. In App-⇓, the argument's shape s_1 is combined with the shape $\bot \to s$, reflecting the fact that the application's result shape is known. As noted by Peyton Jones *et al.* [12, Section 4.6], these rules are not very “smart:” the shape inferred for $id\ x$, where id has shape $\gamma.\gamma \rightarrow \gamma$ and x has shape int, is $\bot$. Indeed, because x is examined in checking mode, the information that x has shape int is discarded. The algorithm in §7 is designed specifically to address this deficiency.

Rule Let-$\Updownarrow$ is straightforward. The metavariable $\Updownarrow$ stands for one of $\Uparrow$ and $\Downarrow$. No generalization in the style of Hindley and Milner takes place, because there is nothing to generalize: the only free type variables in a shape are rigid type variables.

Rule Fix-$\Uparrow$ exploits the type annotation carried by the $\mu^\star$ construct to examine t in checking mode. The subtlety is that the polymorphic type annotation $\exists\bar{\gamma}.\forall\bar{\alpha}.\tau$ is turned into two different shapes. The shape inferred for the entire construct is (the normalized form of) $\bar{\gamma}\bar{\alpha}.\tau$, a shape where the type variables $\bar{\alpha}$ are bound. This shape is also ascribed to x in the environment, so that x can be used at several different types within its own definition. However, the shape that is expected of t is more precise: it is (the normalized form of) $\bar{\gamma}.\tau$, a shape where the type variables $\bar{\alpha}$ are exposed. This makes sense only thanks to our convention that $\mu^\star$ binds $\bar{\alpha}$ within t—in other words, the type variables $\bar{\alpha}$ are rigid within t. Fix-$\Downarrow$ is analogous, but combines the type annotation with the expected shape s.

Applications of data constructors could be treated like function applications. Instead, in Cstr-$\Uparrow$, we adopt a different approach, which is reminiscent of Peyton Jones *et al.*’s “smart application” rule Appn [12, Section 4.6]. The arguments are examined in inference mode, rather than in checking mode, yielding shapes $s_1, \ldots, s_n$. The data constructor’s type scheme, viewed as a shape s, is then unified with the shape $s_1 \times \ldots \times s_n \rightarrow \bot$. This yields an appropriate instance of s, whose codomain is the desired inferred shape. For instance, assuming that $Some$ has type scheme $\forall\alpha.\alpha \rightarrow option\ \alpha$ and x has shape int, this rule allows inferring that $Some\,x$ has shape $option\ int$. Rule Cstr-$\Downarrow$ is analogous to App-$\Downarrow$. The shape operator $\mathcal{D}_i(\cdot)$ extracts the i-th component of the domain of its argument: its definition is analogous to that of $\mathcal{D}(\cdot)$.

Rules Case-$\Uparrow$ and Case-$\Downarrow$ are straightforward. The term t is always examined in inference mode, yielding a shape s'. This shape is passed down to Clause-$\Uparrow$ or Clause-$\Downarrow$, where it is exploited to determine which new equations arise.

In Forall-$\Uparrow$, the rigid type variables $\bar{\alpha}$ can occur free in the shape s, so they are abstracted away in the inferred shape $\bar{\alpha}.s$. In Forall-$\Downarrow$, the expected shape s may involve quantified type variables $\bar{\gamma}$, but we cannot guess how to match these up with $\bar{\alpha}$, so all we can do is pass s down unchanged.

Rules Clause-$\Uparrow$ and Clause-$\Downarrow$ are rather similar to X-Clause in Figure 3. We do, however, introduce an important restriction. The new side condition $\bar{\gamma} \mathrel{\#} \mathrm{ftv}(\bar{\tau}_2)$ requires the (inferred) shape of the scrutinee to be *fully explicit* about the generalized type parameters: they cannot be (or contain) flexible type variables. As explained in §4, this ensures that we have full knowledge of E.

The first premise in Clause-$\Uparrow$ and Clause-$\Downarrow$ confronts the pattern p with the scrutinee’s shape to obtain new rigid type variables $\bar{\beta}$, new equations E', and a new *type* environment Γ'. The flexible type variables $\bar{\gamma}$ can occur free in Γ', so we abstract them away, pointwise, to produce a *shape* environment $\bar{\gamma}.\Gamma'$.

The second premise in Clause-$\Uparrow$ and Clause-$\Downarrow$ examines the sub-term t. The two rules exhibit subtly different formulations.

In Clause-$\Downarrow$, the expected shape s is normalized with respect to $E \wedge E'$, so as to maintain the invariant that the expected shape is normalized with respect to the current equation system. As in rules Var-$\Uparrow$ and Var-$\Downarrow$, this normalization step corresponds to a type conversion: on the outside, the type of this branch appears to be an instance of s, but on the inside, it is an instance of $s\downharpoonleft_{E\wedge E'}$. Again, this is reflected in the transformed term by inserting an explicit type coercion: $(t\ \uparrow_E\ \bar{\gamma}.\tau)$ is defined as syntactic sugar for $(t : \exists\bar{\gamma}.(\tau\downharpoonleft_E\ \triangleright \tau))$, provided $\bar{\gamma} \mathrel{\#} \mathrm{ftv}(E)$ holds.

In Clause-$\Uparrow$, no coercion is inserted: because no expected shape is initially available, no normalization takes place. Instead, t’s inferred shape s is *pruned* to produce an inferred shape $s\restriction_{E,E'}$ for the clause. The need for pruning was discussed earlier (§5.3).

The main differences between Peyton Jones *et al.*’s original type system and W are (i) our use of unification, implicit in the least upper bound operator over shapes, which makes the algorithm more accurate (unification was also present in Peyton Jones *et al.*’s Appn); and (ii) our insistence on full knowledge of the equations that arise at case constructs and our use of pruning.

Every well-typed MLGI program can be turned into a program that is well-typed with respect to the combination of W and MLGX by adding enough type annotations. We omit a formal statement.

Example Consider again the $eval$ example of §1, where μ is replaced with $\mu^\star$. Let us attack this term in inference mode. Fix-$\Uparrow$ switches to checking mode for the sub-term $\lambda t. \ldots$, with expected shape $term\ \alpha \rightarrow \alpha$. Lam-$\Downarrow$ determines that t has shape $term\ \alpha$ and checks the case construct against shape α. Case-$\Downarrow$ looks up the environment and infers that t has shape $term\ \alpha$, which allows inserting the annotation $(t : term\ \alpha)$ in the transformed term. Every branch is then examined by Clause-$\Downarrow$, with knowledge that the scrutinee has shape $term\ \alpha$ and that the branch has expected shape α. In the Lit branch, for instance, the equation $\alpha = int$ becomes available, so the expected shape α is normalized to int upon entry, and the coercion $(i : (int \triangleright \alpha))$ is inserted. The variable i is then successfully checked against shape int.

This explains how the $eval$ example of §1 is automatically transformed into the annotated version of §4. Then, the transformed program is successfully submitted to MLGX type inference.

Soundness We now give a formal soundness statement for W. The statement assumes that a derivation of $E, \Gamma \vdash t : \sigma$ in MLGI is given. Naturally, in practice, this derivation is not known: it only exists in the programmer’s mind! Here, it is used as an oracle: the assertion that a shape s is a sound approximation of t’s “true” type is encoded by the statement $E \Vdash s \preceq \sigma$. The assertion that the annotations and coercions inserted by the algorithm are sound is encoded by the statement that the transformed term t' still has type σ in MLGI. Thus, Item 1 of Theorem 6.1 can be read: *if W is invoked in inference mode and supplied with sound assumptions, then it produces a sound shape and inserts sound annotations and coercions.* Item 2 makes a similar statement about checking mode.

Theorem 6.1 (Soundness) *Let $E, \Gamma \vdash t : \sigma$ hold in MLGI. Let $E \Vdash \Gamma' \preceq \Gamma$ hold. Then,*

1. *If $E, \Gamma' \vdash t \Uparrow s \rightsquigarrow t'$ holds in W, then $E \Vdash s \preceq \sigma$ holds and $E, \Gamma \vdash t' : \sigma$ holds in MLGI.*
2. *If $E, \Gamma' \vdash t \Downarrow s \rightsquigarrow t'$ holds in W, $E \Vdash s \preceq \sigma$ holds, and s is normalized w.r.t. E, then $E, \Gamma \vdash t' : \sigma$ holds in MLGI.* ◇

Soundness comes at a price. It rests upon pruning, which demands exact knowledge of the current equation system. This led to requiring, in Clause-$\Uparrow$ and Clause-$\Downarrow$, that the (inferred) shape of the case scrutinee be fully explicit about the generalized type parameters. If one gave up soundness, one could design more liberal versions of these rules where the shape of the scrutinee is allowed to be incomplete, giving rise to a weaker equation system within the clause. This route is followed by Peyton Jones *et al.* [12], whose “wobbly unification” algorithm “*may do less type refinement than would be justified in an explicitly-typed program.*” On the one hand, because Peyton Jones *et al.*’s system is able to work with an underapproximation of the current equation system, it accepts more programs; on the other hand, because it does not do any pruning, it sometimes infers unsound shapes.

Z-False
$$\textsf{false}, \Gamma \vdash t \Downarrow s \Uparrow s \rightsquigarrow t$$

Z-Var
$$\frac{(x : s') \in \Gamma}{E, \Gamma \vdash x \Downarrow s \Uparrow (s \sqcup s' \lfloor_E\,) \rightsquigarrow (x \downarrow_E s')}$$

Z-Lam
$$\frac{s' = s \sqcup (\theta \lfloor_E \,\to \bot) \qquad E, \Gamma; x : \mathcal{D}(s') \vdash t \Downarrow \mathcal{C}(s') \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash \lambda(x : \theta).t \Downarrow s \Uparrow (s' \sqcup (\bot \to s'')) \rightsquigarrow \lambda(x : \theta \lfloor_E\,).t'}$$

Z-App
$$\frac{s' = s_1 \sqcup (s_2 \to s) \qquad E, \Gamma \vdash t_1 \Downarrow s' \Uparrow s'_1 \rightsquigarrow t'_1 \qquad E, \Gamma \vdash t_2 \Downarrow \mathcal{D}(s'_1) \Uparrow s'_2 \rightsquigarrow t'_2}{E, \Gamma \vdash (t_1 : s_1)\ (t_2 : s_2) \Downarrow s \Uparrow \mathcal{C}(s'_1 \sqcup (s'_2 \to \bot)) \rightsquigarrow (t'_1 : s'_1)\ (t'_2 : s'_2)}$$

Z-Let
$$\frac{E, \Gamma \vdash t_1 \Downarrow s_1 \Uparrow s'_1 \rightsquigarrow t'_1 \qquad E, \Gamma; x : s'_1 \vdash t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow t'_2}{E, \Gamma \vdash \textsf{let}\, x = (t_1 : s_1)\, \textsf{in}\, t_2 \Downarrow s \Uparrow s_2 \rightsquigarrow \textsf{let}\, x = (t'_1 : s'_1)\, \textsf{in}\, t'_2}$$

Z-Fix
$$\frac{\bar\alpha \mathrel{\#} \mathrm{ftv}(E, \Gamma, s) \qquad E, \Gamma; x : (\bar\gamma\bar\alpha.\tau \lfloor_E \,\sqcup s) \vdash t \Downarrow (\bar\gamma.\tau \lfloor_E \,\sqcup s) \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash \mu^\star(x : \exists\bar\gamma.\forall\bar\alpha.\tau).t \Downarrow s \Uparrow \bar\alpha.s'' \rightsquigarrow \mu^\star(x : \exists\bar\gamma.\forall\bar\alpha.\tau \lfloor_E\,).t'}$$

Z-Case
$$\frac{E, \Gamma \vdash t \Downarrow s' \Uparrow s'' \rightsquigarrow t' \qquad \forall i \quad E, \Gamma \vdash (p_i : s'').t_i \Downarrow s \Uparrow s_i \rightsquigarrow p_i.t'_i}{E, \Gamma \vdash \textsf{case}\, (t : s')\, \textsf{of}\, p_1.t_1 \ldots p_n.t_n \Downarrow s \Uparrow \sqcup_i s_i \rightsquigarrow \textsf{case}\, (t' : s'')\, \textsf{of}\, p_1.t'_1 \ldots p_n.t'_n}$$

Z-Forall
$$\frac{E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t' \qquad \bar\alpha \mathrel{\#} \mathrm{ftv}(\Gamma, E, s)}{E, \Gamma \vdash \forall\bar\alpha.t \Downarrow s \Uparrow \bar\alpha.s' \rightsquigarrow \forall\bar\alpha.t'}$$

Z-Annot
$$\frac{E, \Gamma \vdash t \Downarrow (\theta \lfloor_E \,\sqcup s) \Uparrow s'' \rightsquigarrow t'}{E, \Gamma \vdash (t : \theta) \Downarrow s \Uparrow s'' \rightsquigarrow (t' : \theta \lfloor_E\,)}$$

Z-Coerce
$$\frac{E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'}{E, \Gamma \vdash (t : \kappa) \Downarrow s \Uparrow s' \rightsquigarrow t'}$$

Z-Clause
$$\frac{p : \varepsilon\, \bar\tau_1\, \bar\tau_2 \vdash (\bar\beta, E', \Gamma') \qquad E \wedge E', \Gamma(\bar\gamma.\Gamma') \vdash t \Downarrow s \lfloor_{E \wedge E'} \Uparrow s' \rightsquigarrow t' \qquad s'' = s' \upharpoonright_{E,E'} \sqcup s \qquad \bar\beta \mathrel{\#} \mathrm{ftv}(E, \Gamma, s'') \qquad \bar\gamma \mathrel{\#} \mathrm{ftv}(E, \Gamma, \bar\tau_2, t, s)}{E, \Gamma \vdash (p : \bar\gamma.\varepsilon\, \bar\tau_1\, \bar\tau_2).t \Downarrow s \Uparrow s'' \rightsquigarrow p.(t' \uparrow_{E \wedge E'} s'')}$$

Figure 7. The shape inference system Z

7. The shape inference system Z

Although W successfully turns the *eval* example of §1 into a well-typed MLGX term, it suffers from a severe shortcoming in its treatment of application. As noted in §6, it *infers* the function's shape and uses this information to examine the argument in *checking* mode. Our rule CSTR-⇑ and Peyton Jones *et al.*'s APPN [12] make an opposite choice and examine the arguments in *inference* mode. In fact, either choice is *ad hoc*. Ideally, shape information should be allowed to flow from function to argument *and vice versa*. We now describe a shape inference system, called Z, that subsumes W and is designed to allow propagation both ways.

Example 7.1 The following term illustrates W's shortcoming. We assume that the data constructor I has type $ty\ \ int$, so that the equation $\alpha = int$ is available inside the case construct.

$$\mu^\star(double : \forall\alpha.ty\ \alpha \to list\ \alpha \to list\ \alpha).\lambda t.\lambda l.$$
$$map\ (\lambda x.\textsf{case}\, t\, \textsf{of}\ I \to x + x)\ l$$

Three explicit coercions are necessary to turn this into a well-typed MLGX term. At both occurrences of x, α must be converted to int. Furthermore, $x + x$ must be coerced from int back to α, so as to satisfy the programmer-supplied annotation, which requires the anonymous function's return type to be α.

However, W is unable to insert any coercion. The term contains a double application of map. W attacks the outermost application in checking mode with expected type $list\ \ \alpha$. By APP-⇓, this requires first *inferring* a shape for the innermost application $map\ \ (\lambda x.\ldots)$, then *checking* the argument l. Thus, APP-⇑ is applied to the innermost application. The shape inferred for map is $\gamma_1\gamma_2.(\gamma_1 \ \to\ \gamma_2) \ \to\ list\ \ \gamma_1 \ \to\ list\ \ \gamma_2$. This leads to checking that $\lambda x.\ldots$ has shape $\gamma_1\gamma_2.\gamma_1 \ \to\ \gamma_2$. This imprecise shape does not provide any information about the type of x or about the anonymous function's return type. As a consequence, W is unable to insert any coercion into the function.

In this example, the "right" thing to do at the outermost application is to first examine l in *inference* mode, yielding the shape $list\ \alpha$, and to exploit this information to examine the innermost application in *checking* mode with expected shape $list\ \ \alpha \ \to\ list\ \ \alpha$. The "right" thing to do at the innermost application is to examine map in *inference* mode, since its type is known, and to exploit this information to examine $\lambda x.\ldots$ in *checking* mode, with expected shape $\alpha \to \alpha$. This shows that committing to either left-to-right or right-to-left propagation is a bad idea. ◇

To avoid such a commitment, we suggest dealing with application (and, in general, with binary constructs) in a *symmetric* way. This becomes possible if shape inference is broken up into two passes. The idea is, roughly speaking, as follows. During a first pass, both function and argument are examined in *inference* mode. During a second pass, both are examined in *checking* mode. The shape inferred for the function during the first pass is used during the second pass to predict the argument's expected type, and vice versa. This allows information to propagate both ways.

Another, independent idea is to abandon the distinction between inference and checking modes altogether. Indeed, when in inference mode, why refuse to take advantage of the information offered by an expected shape? Conversely, when in checking mode, why refuse to produce an inferred shape which potentially could be more precise than the initially expected shape? Our answer is to perform both checking and inference at once. Judgements in Z take the form $E, \Gamma \vdash t \Downarrow s \Uparrow s' \rightsquigarrow t'$, where the inferred shape s', an output parameter, is always at least as precise as the expected shape s, an input parameter—that is, $s \preceq s'$ holds. Thus, Z is bidirectional, like W, but runs *in both modes simultaneously*. This is reminiscent of *colored local type inference* [9].

Our rough initial statement that the first pass runs in inference mode, while the second pass runs in checking mode, can now be made more precise. Both passes are in fact identical—that is, they are defined by the exact same set of rules—and both simultaneously perform shape checking and inference. The first pass annotates the immediate sub-terms of all application nodes with their inferred shapes. The second pass then exploits these annotations to infer more accurate shapes.

Thus, Z really consists of a single pass than can be *iterated* as many times as desired. In practice, iterating at least twice is required for information to be propagated from function to argument, and vice versa, at application nodes. Iterating more than twice might allow accepting more programs, but might also compromise the algorithm's perceived simplicity and predictability.

Presentation The rules that define Z are shown in Figure 7. Rule Z-APP expects the function t_1 and the argument t_2 to carry explicit type annotations s_1 and s_2. If, in fact, there is no such annotation, then $\bot$ is used. In practice, there typically is no annotation before the first pass. The inferred shapes s'_1 and s'_2 are recorded as type annotations, to be exploited during the next pass, if there is one.

By assumption, there are no coercions in the source term. Rule Z-COERCE states that each pass erases the coercions inserted during the previous pass. Indeed, since each new pass has better shape information than the previous pass, it is able to produce more accurate coercions.

We omit a detailed explanation of the other rules. In most cases, the inference and checking variants of every rule of W are superimposed to produce a rule that does checking and inference simultaneously.

Example 7.2 Consider the $double$ example again:

$$\begin{array}{l}\mu^\star(double : \forall\alpha.ty\ \alpha \to list\ \alpha \to list\ \alpha).\lambda t.\lambda l.\\ \quad map\ (\lambda x.\mathsf{case}\,t\,\mathsf{of}\ I \to x + x)\ l\end{array}$$

The first pass of algorithm Z attacks the outermost application with expected shape $list\ \alpha$. First, it examines the left-hand side, that is, the innermost application, with expected shape $\bot \to list\ \alpha$. This leads to examining map with expected shape $\bot \to \bot \to list\ \alpha$. Rule Z-VAR combines this with map's known shape, yielding the inferred shape $\gamma.(\gamma \to \alpha) \to list\ \gamma \to list\ \alpha$. The sub-term map is annotated with this shape. Then, the anonymous function is entered, with expected shape $\gamma.\gamma \to \alpha$. The case construct is examined with expected shape α, which leads Z-CLAUSE to inserting a coercion of int back to α around the clause. The outcome of the first pass is the term

$$\begin{array}{l}\mu^\star(double : \forall\alpha.ty\ \alpha \to list\ \alpha \to list\ \alpha).\lambda t.\lambda l.\\ \quad \big((map : \gamma.(\gamma \to \alpha) \to list\ \gamma \to list\ \alpha)\\ \quad\ \ ((\lambda x.\mathsf{case}\,t\,\mathsf{of}\ I \to (x + x : int \triangleright \alpha)) : \gamma.\gamma \to \alpha)\\ \quad\ \ : \gamma.list\ \gamma \to list\ \alpha\big)\\ \quad (l : list\ \alpha)\end{array}$$

In the second pass, at the outermost application, the function's inferred shape $\gamma.list\ \gamma \to list\ \alpha$ is combined with the argument's inferred shape $list\ \alpha$. This leads to examining the innermost application with expected shape $list\ \alpha \to list\ \alpha$. At the innermost application, this information allows determining that map is being used at type $(\alpha \to \alpha) \to list\ \alpha \to list\ \alpha$, which leads to examining the anonymous function $\lambda x. \ldots$ with expected shape $\alpha \to \alpha$. Z's second pass is now able to determine that x has shape α, which allows Z-VAR to insert coercions from α to int at both uses of x. The term produced by the second pass is:

$$\begin{array}{l}\mu^\star(double : \forall\alpha.ty\ \alpha \to list\ \alpha \to list\ \alpha).\lambda t.\lambda l.\\ \quad \big((map : (\alpha \to \alpha) \to list\ \alpha \to list\ \alpha)\\ \quad\ \ ((\lambda x.\mathsf{case}\,t\,\mathsf{of}\\ \qquad\qquad I \to ((x : \alpha \triangleright int) + (x : \alpha \triangleright int) : int \triangleright \alpha))\\ \quad\ \ \ : \alpha \to \alpha)\\ \quad\ \ : list\ \alpha \to list\ \alpha\big)\\ \quad (l : list\ \alpha)\end{array}$$

It is well-typed in MLGX, which means that this definition of $double$ is accepted by the stratified type inference system that combines Z and MLGX. $\diamond$

Statements Just like W, Z enjoys a soundness theorem, which we do not state here. Similarly, every well-typed MLGI program can be turned into a program that is well-typed with respect to the combination of Z and MLGX by adding enough type annotations. The two theorems below state that the first iteration of Z alone yields shape information that is more precise than that offered by W, and that each further iteration of Z refines this information.

Theorem 7.3 (*Z* subsumes *W*) *If $E, \Gamma \vdash t \Uparrow s \rightsquigarrow t'$ holds in W, then there exist a shape s' and a term t'' such that $E, \Gamma \vdash t \Downarrow \bot \Uparrow s' \rightsquigarrow t''$ holds in Z and $s \preceq s'$.* $\diamond$

Theorem 7.4 (Iteration) *$E, \Gamma \vdash t \Downarrow \bot \Uparrow s \rightsquigarrow t'$ and $E, \Gamma \vdash t' \Downarrow \bot \Uparrow s' \rightsquigarrow t''$ imply $s \preceq s'$.* $\diamond$

8. Conclusion

We have introduced *stratified type inference*, which separates traditional type inference in the style of Hindley and Milner from local propagation of explicit type information, and illustrated this idea in the case of type inference for generalized algebraic data types. An analogous idea is developed by Rémy [20] in the case of type inference for arbitrary-rank predicative polymorphism.

Our bottom stratum, MLGX, extends Milner and Mycroft's type system in a minimal way so as to accommodate generalized algebraic data types. Our top strata, W and Z, are defined using a common toolbox of operations on *shapes*, which seem particularly well-suited for expressing *approximate* knowledge about types.

A prototype implementation of our proposal, written by the second author, is available and can be used online [19].

We improve upon Simonet and Pottier's work [23] by uniformly dealing with ordinary and generalized algebraic data types, by accepting arbitrary "lexically scoped" type annotations, and by avoiding implication constraints entirely. We improve upon Peyton Jones *et al.*'s [12] by offering a more modular presentation and by performing more accurate shape inference.

One could further enhance our shape inference algorithm, Z. For instance, Z's judgements mention both an expected shape and an inferred shape. For symmetry, one could also use both a *given environment* (an input parameter) and a *requested environment* (an output parameter). That would help deal with let constructs in a more precise way. Second, it should sometimes be possible to infer the shape of a case construct by *reconciling* the shapes of the various branches, even when these are incompatible. This involves examining the equation systems available within each branch and performing a form of anti-unification.

Our systematic use of normalization, which follows Peyton Jones *et al.*, is not always satisfactory. Assume that the equation $\alpha = int$ is available. When one writes $(x : \alpha)$, the shape inference systems W and Z behave exactly as if one had written $(x : int)$. Some valuable information is discarded: perhaps the programmer really intended to tell the system that x is being used at type α, not int. This behavior makes the meaning of a type annotation dependent upon E. As a result, moving a type annotation into or out of a case construct can change its meaning! Yet, it is not entirely clear how to avoid this shortcoming without sacrificing accuracy.

References

[1] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell workshop*, 2002.

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

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

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

[5] Ralf Hinze. Fun with phantom types. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan, March 2003.

[6] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris 7, September 1976.

[7] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer Verlag, April 1984.

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

[9] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

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

[11] Simon Peyton Jones and Mark Shields. Practical type inference for arbitrary-rank types. Manuscript, April 2004.

[12] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Manuscript, July 2004.

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

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

[15] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types (extended version). `http://cristal.inria.fr/~regisgia/publis/prg-long.pdf`, July 2005.

[16] François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. Manuscript, April 2005.

[17] François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

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

[19] Yann Régis-Gianas. A prototype typechecker for ML with generalized algebraic data types. `http://cristal.inria.fr/~regisgia/software/`, July 2005.

[20] Didier Rémy. Simple, partial type inference for system F based on type containment. In *ACM International Conference on Functional Programming (ICFP)*, September 2005.

[21] Tim Sheard. Languages of the future. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 116–119, October 2004.

[22] Tim Sheard and Emir Pašalić. Meta-programming with built-in type equality. In *Workshop on Logical Frameworks and Meta-Languages (LFM)*, July 2004.

[23] Vincent Simonet and François Pottier. Constraint-based type inference for guarded algebraic data types. Research Report 5462, INRIA, January 2005.

[24] Peter J. Stuckey and Martin Sulzmann. Solutions of implication constraints yield type inference for more general algebraic data types. Manuscript, April 2005.

[25] Peter J. Stuckey and Martin Sulzmann. Type inference for guarded recursive data types. Manuscript, February 2005.

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

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

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