Concoqtion: Mixing Indexed Types and Hindley-Milner Type Inference

Emir Pašalić Jeremy Siek Walid Taha
Rice University
{pasalic,Jeremy.G.Siek,taha}@rice.edu

Abstract

This paper addresses the question of how to extend OCaml's Hindley-Milner type system with types indexed by logical propositions and proofs of the Coq theorem prover, thereby providing an expressive and extensible mechanism for ensuring fine-grained program invariants. We propose adopting the approached used by Shao et al. for certified binaries. This approach maintains a phase distinction between the computational and logical languages, thereby limiting effects and non-termination to the computational language, and maintaining the decidability of the type system. The extension subsumes language features such as impredicative first-class (higher-rank) polymorphism and type operators, that are notoriously difficult to integrate with the Hindley-Milner style of type inference that is used in OCaml. We make the observation that these features can be more easily integrated with type inference if the inference algorithm is free to adapt the order in which it solves typing constraints to each program. To this end we define a novel "order-free" type inference algorithm. The key enabling technology is a graph representation of constraints and a constraint solver that performs Hindley-Milner inference with just three graph rewrite rules.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Polymorphism, Data types and structures

General Terms Languages, Theory

Keywords Type Inference, Polymorphism, Type Operators, System F, Dependent Types, Indexed Types, Theorem Proving

1. Introduction

While there have been several approaches to introducing ideas from dependent type theory into traditional programming language deisgn [2, 3, 17, 31] it is often an explicit goal of these approaches to automate the construction of proofs [5, 14, 31, 38]. In principle, automating the proofs can make programming easier. In reality, whether or not this is the case is a complex human-factors question. An alternative approach equally worthy of exploration to make the underlying proof explict, to use a well-developed and well-establishing proof theory, and to integrate the proof theory direclty into a tradtional programming language design. In particular, predictability is a goal both in the design of type systems and proof theory, and direct approach to combining the fruits of both may stand the best chance of yielding the most predictable design. In contrast, decision procedures are often the result of of a deep understanding of particular properties of specific subsets of provable propositions, and it making them predictable to a programmer might require explaining many of their internal (and algorithmic) details.

[copyright notice will appear here]

Our goal is to explore the language design approach that first proposed by Shao et al. in the context of intermediate language design [30], and was later advocated in the context of source-language design by Pasalic, Taha and Sheard [24]. For the purposes of source language design, however, both these works assumed that the computational language being extended is explicitly typed, rather than being based on Hindley-Milner inference. This paper describes Concoqtion, a conservative extension of OCaml's type system with the terms of the Coq proof theory, and addresses the key technical problems that arise in this setting. This is particularly challenging because the extension subsumes impredicative first-class polymorphism and type operators. Considerable research has gone into inference in the presence of first-class polymorphism [12, 13, 20, 25, 29, 36], whereas much less research has gone into integrating type inference with type operators, with Pfenning's partial inference as a notable exception [27].

What distinguishes our appoach is its simplicity, predictability, and indisputable expressive power. Instead of developing a new type system with incremental goals, we focus on developing a more extensible constraint-based inference algorithm for the Hindley-Milner type system. This allows us to directly integrate a Hindley-Milner system with, for example, the Coq proof theory. The key technical insight behind the new inference algorithm is to represent constraints and types in a directed acyclic graph and to accomplish constraint solving by graph rewriting. Only three rewrite rules are required to solve the Hindley-Milner constraints. We then extend the inference algorithm to handle indexed types. This extension adds new kinds of constraints and rewrite rules but does not change or interfere with the Hindley-Milner constraints or rewrite rules.

Background Linking the computational to the logical language involves adding first-class polymorphism to the computational language, where the parameters may range over entities in the logical language (which includes but is not limited to the the computational language types). Furthermore, Coq functions may be used inside the computational type expressions as type operators, making Concoqtion effectively a superset of F_ω. Our key observation is that Hindley-Milner type inference can be made more robust by making the algorithm *order-free*, that is, if the algorithm is free to adapt the order in which it infers types for expressions within the program. Consider the following program (`Λa. e` stands for type

abstraction)

```
(fun f →f .|int| 1) (Λa. fun x:a →x)
```

If the inference algorithm traverses the syntax in the usual order (as in algorithm $\mathcal{W}$), then it can not infer the type of `f .|int|` because the type of `f` is unknown. However, if the inferencer first considers `Λa. fun x:a →x` and the topmost application, it can infer the type `∀a. a→a` for `f`, which then allows the inferencer to infer the type `int→int` for the type application.

Constraint-based inference algorithms, such as HM(X) of Odersky, Sulzmann, and Wehr [21], provide a first step towards an order-free algorithm by separating inference into two parts: constraint generation and constraint solving. However, for the Hindley-Milner type system, it is non-trivial to completely separate constraint generation and solving. Last year Pottier and Rémy [8] defined the first constraint-based Hindley-Milner inference algorithm with complete separation.

The constraint solver of Pottier and Rémy, while mostly order free, still imposes some ordering restrictions, presumably for efficiency reasons. However, their ordering restrictions are more than necessary for performance. In this paper we lift the few remaining order restrictions. The key is to represent constraints in a directed acyclic graph (including instantiation constraints) and to express the constraint solver as a set of graph rewriting rules. The resulting algorithm is order-free and it is simple and intuitive, requiring only three rewrite rules. Also, the graph-based formalism closely matches the data-structures used in Concoqtion type inferencer implementation.

With an order-free type inference algorithm in hand, it is straightforward to perform inference in the presence of first-class polymorphism and type operators. The solving of constraints involving type application and type operators is delayed until the input types become known. In the case of a type application `e .|t|`, the result type remains unknown (a type variable) until a universal type is inferred for `e`, at which time the result type is updated to the instantiation of the universal with `t`. In the case of a type expression that is a type operator application, unification with other types is delayed until the input to the operator has been inferred. Of course, unification of two identical type operator applications proceeds immediately.

Like the other approaches to inference in the presence of first-class polymorphism, our type inferencer does not guess universal types, but instead requires some annotations from the programmer and then propagates the annotations. Unlike other approaches, we use our order-free Hindley-Milner inference algorithm to propagate annotations. Most other approaches rely on bidirectional type inference (part of local inference) to propagate annotations [28, 36, 38]. We hypothesize that our approach propagates annotations to more locations and is easier to understand for functional programmers familiar with ML-style inference as they can continue to rely on their intuitions. In Section 5 we compare our approach with these other techniques in more detail.

Road Map and Contributions To help give the reader an intuition of what it is like to program in Concoqtion, we begin with a brief introduction to the language, illustrating its features with examples (Section 2).

Our first contribution is the first order-free inference algorithm for the Hindley-Milner type system.

The algorithm is sound, complete, and terminates. We present the algorithm first in the simpler setting of OCaml-the-calculus (Section 3) for two reasons: first, to present the ideas without the distractions of the Concoqtion extensions; second, to underscore the essential modularity of our formal development: the sets of constraint generation and solving rules are augmented, but the existing rules are not changed, by the subsequent extensions of the inference algorithm. We hypothesize that our Hindley-Milner inference algorithm can be used as a departure point for many other extensions to the Hindley-Milner type system both in theory and implementation.

Our second contribution is to extend the order-free inference algorithm to handle the additional features of Concoqtion (Section 4), which include impredicative first-class polymorphism and type operators. We characterize the extended inference algorithm by establishing its relationship to two type systems: the first type system (Section 4.2) has a traditional presentation and provides a lower bound on the capabilities of the inference algorithm. The second type system (Section 4.5) is novel, and accepts *exactly* the same set of programs that our inference algorithm accepts. The two type system differ in that the first only treats the subset of Concoqtion where all type applications are explicitly annotated with the universal type being eliminated. The extended inference algorithm terminates, and is sound and complete with respect to both type systems.

In other words, the inference algorithm presented here is predictable the sense that in that it propagates types (and type annotations) according to the Hindley-Milner constraints that many functional programmers are familiar with. This is formalized by showing completeness with respect to the Concoqtion type system, for which there exists a declarative specification for our inference algorithm. Specifying this type system requires care, because the naive combination of the Hindley-Milner rules with the F_ω rules results in a type system for which there can be no complete algorithm [35]. The problem can be reduced to higher-order unification [27], which is also undecidable problem, and using a semi-decision procedure results in unpredictable behavior. To deal with this problem, and following the standard solution, we refrain from guessing universal types. For example, the following Concoqtion term is not well-typed because there is no way to propagate (using Hindley-Milner rules) a universal type to `f`. (The concrete Concoqtion syntax uses `e .|t|` for type application.)

```
fun f →f .|int| 1
```

Unlike other approaches, we exhibit a type system that precisely specifies the terms that are accepted by our inference algorithm, that is, we provide completeness. This is achieved by a novel type system that is not allowed to guess polymorphism.

A prototype implementation of Concoqtion, implemented as a modification of the OCaml 3.08 compiler, is available online [1]. An accompanying technical report [23] details the proofs and formal definitions that we did not have the space to present in this paper.

2. Design of Concoqtion

Concoqtion extends OCaml by allowing Coq [34] terms to appear in OCaml types. A Coq term `c` can appear in a Concoqtion type in the form of a *tick type*, written `'(c)`. Concoqtion is explicitly kinded, and uses Coq terms for its kinds. In particular, there is a Coq constant kind **`ot`**`:Set`[1]. Furthermore, for each OCaml type constructor `(t1:k1,...,tn:kn) t`, there is a constant in Coq, named `OT_t : k1 →... →kn →`**`ot`** that embeds the type into Coq. For example, the types `'(OT_int)` and `int` are interchangeable in Concoqtion.

Concoqtion has explicit first-class polymorphic types, where type variables can range over both OCaml types and Coq values. Similarly, we extend OCaml data-types so that they can be parameterized over particular Coq values as well as OCaml types. This allows us to connect particular values in OCaml with values in Coq.

For example, consider the Concoqtion data-type for lists of specific length `n` (Figure 1a). The type constructor `listN` takes two parameters: the first is the type of the list element; the second,

[1] The name **`ot`** stands for 'OCaml type.'

(a) Concoqtion

```
type ('a, 'n:'(nat)) listN =
  | Nil : ('a, '(0)) listN
  | Cons of let 'm:'(nat) in 'a * ('a, '(m)) listN
        : ('a, '(1+m)) listN

let rec app .|m:'(nat)| .|n:'(nat)|
  (l1 : (_,'(m)) listN) (l2 : (_,'(n)) listN)
  : (_,'(m+n)) listN =
 match l1 as ('a:'(ot),'i:'(nat)) listN
             : ('(a),'(i+n)) listN with
 | Nil → l2
 | Cons .|m2:'(nat)| (x,xs) →
    Cons .|'(m2+n) |
         (x, app .|'(m2),'(n) | xs l2)
```

(b) Haskell (GHC) using GADTs

```
data Z
data S x

data ListN a n where
  Nil :: ListN a Z
  Cons :: a → ListN a m → ListN a (S m)

data Sum m n s where
  SumZ :: Sum Z n n
  SumS :: (Sum a n r) → Sum (S a) n (S r)

data PlusLenL a m n where
  PP :: (Sum m n sum) → (ListN a sum) → PlusLenL a m n

app :: ListN a m → ListN a n → PlusLenList a m n
app Nil ys = PP SumZ ys
app (Cons x xs) ys =
   case app xs ys of
   PP sum rest → PP (SumS sum) (Cons x rest)
```

Figure 1. Lists with length in Concoqtion and in Haskell (GHC)

a Coq term of type **nat** is the length of the list. The data constructor `Nil` is the empty list and thus has the type `('a, '(0)) listN`. The data-constructor `Cons` has a universally quantified type: for any natural number `m`, it takes a value of the element type `'a`, a list of length `m`, and constructs a list of length `(1+m)`.

We can write functions over these extended data-types using an extended form of pattern matching. For example, appending two lists of length `m` and `n`, respectively, results in a list of length `m+n`. This invariant is captured in the type of the concatenation function:

```
app : ∀m,n : '(nat). ('a,'(m)) listN
      → ('a,'(n)) listN → ('a,'(m+n)) listN
```

Note also the explicit universal quantification, written `∀X:kind.t`. Values of universally quantified types are constructed by explicit type abstraction (written `ΛX:k.e`). Analogously to function in OCaml, Concoqtion provides OCaml-style let-bound syntax for type application, written
`let f .|X:k| ... = e`.
The same extended syntax can be used to give explicit type signatures for functions with first-class polymorphism.

The function `app` deconstructs the first list `l1` (of length `n`) using a **`match`** form. In each case, the signatures of `listN` data-constructors are used to introduce more assumptions about type parameters. Thus, if `l1` is the empty list (`Nil`), then the first branch can rely on the fact that `n` is zero. It is easily verified that the result of the branch `l2` has the required type (`('a,'(n+m)) listN`, since `'(0+m) = '(m)`). In the second branch, the length of `l1` is assumed to be `'(1+m2)` and the sub-list `xs` has the type `('a,'(m2)) listN`. The result of the branch has the length `(1+(m2+n))` which is equal to the desired length `'(m+n)`.

Let us compare the Concoqtion implementation of `app` to a similar implementation in Haskell using GADTs [26] (Figure 1b, following an example of Sheard's [31]). The data-type `ListN` plays the same role as `listN` in Concoqtion, except that the integer values of the length index are encoded as Haskell types built up of type constructors `Z` and `S`. Aside from surface syntactic differences with Concoqtion, in the sub-index `m` in the constructor `Cons` is quantified implicitly in Haskell. Similarly, when constructing values with `Cons` in Haskell, the type application is implicitly inferred by the type-checker.

The Concoqtion type of `app` directly expresses the intuitive point that the length of two appended lists is the sum of their length. In Haskell, however, we have no way of directly writing down the type index `m+n`. Instead, we first encode what is means to be a sum of two numbers in the auxiliary data-type `Sum m n s`: if we can construct a value of type `Sum m n s`, than we have a proof that `m+n = s`. Next, the Concoqtion type `('a,'(m+n)) listN` is represented by the Haskell type `∃s. (Sum m n s, List a s)`.

Both Concoqtion and Haskell examples use a kind of GADT for representing lists which are computational data. In Haskell, GADTs are used to encode propositions in a "relational style." In Concoqtion, on the other hand, we are free to use GADT-like notation for list values (for which GADTs are well suited), but use the more concise and clear notation of Coq for properties. Moreover, since proofs are constructed entirely in Coq and act only at the level of types, they can be erased by the Concoqtion compiler and incur no runtime overhead.

2.1 The Burden of Proof

Suppose we with to call a function in Haskell that took a list of length $m+n$ (`PlusLenL a m n`) but all we have is a list of length $n+m$ (`PlusLenL a n m`). To use the value available, we would have to explicitly prove that addition is commutative by providing a function of type `Sum m n s→Sum n m s`. Such a function can indeed be built by recursively deconstructing an object of type `Sum m n s` and rebuilding a `Sum m n s`.

What about Concoqtion? Again, suppose we had a value `x` of type `('a, '(m+n)) listN` and what we really need is a value of type `('a,'(n+m)) listN`. Somehow, we must use the fact that addition is commutative to convert between the two types. Note, however, that the two types are *not* convertible (modulo Coq reduction relations) to each other: we will have to prove them equal, and use that proof to cast from one type to another. Such a casting operator is a part of the standard Concoqtion library and has the type `∀a:'(ot). ∀b:'(ot). ∀p:'(a=b). '(a) →'(b)`.

In Concoqtion programs, we can use a special kind of declarations, the **`coq ... end`** sections programs to construct such proofs by inlining Coq proof scripts:

```
coq
  Require Import Arith.
  Lemma lemma1 : ∀elem, ∀m n, (m = n) →
       (OT_listN elem m) → (OT_listN elem n).
   intros; eauto. Qed.
end
```

Finally, we can combine `lemma1` with a standard Coq library theorem `plus_comm` to obtain the following function:

```
let comm .|a , m:'(nat), n:'(nat)|
    (x:('(a), '(m+n)) listN) : ('(a),'(n+m)) listN =
   cast
     .|'(OT_listN a (m+n)), '(OT_listN a (n+m)),
       '(lemma1 a (m+n) (n+m) (plus_comm m n)) | x
```

We note two points. First, the type-safe cast function in Concoqtion is generic and works for any types which we can prove

equal in Coq. Moreover, the proof of the commutativity of addition is already a theorem in Coq and we can simply refer to it without ever having to reprove it. Second, these proofs and properties live entirely in the logical language and have no runtime cost, in contrast to Haskell, where proof objects for various logical properties expressed as Haskell data-types must be manipulated at runtime.

2.2 Using Decision Procedures

Consider writing the function `comm` in DML [38] or Zenger's indexed type language [39]. The cast would not be necessary and the equivalence `m+n = n+m` would be proved automatically by a Presburger arithmetic decision procedure that is built into the type checker. Clearly, this is less burdensome on the programmer than our Concoqtion example above. However, in Concoqtion, the programmer has access to the decision procedures of Coq, which greatly reduces the burden of proof. For example, the `comm` function can be written more concisely as follows:

```
let comm .|a, m:'(nat), n:'(nat)|
    (x:('(a), '(m+n)) listN) : ('(a),'(n+m)) listN =
   cast .|^(eauto), ^(eauto), ^(omega;eauto)| x
```

In this example, the programmer uses an alternative form of the tick type, written `^( [ goal| ] script)`. The (optional) goal argument specifies a proposition (in Concoqtion this is a *kind*), e.g., `(_,'(m+n))listN =(_,'(n+m)) listN`.
Often this annotation too can be omitted, as it can be inferred from the context. Next, the `script` argument is a (usually very short) Coq proof script which instructs the theorem prover to use a particular decision procedure. For example, the above `^(omega;eauto)` instructs Coq to use the Presburger arithmetic decision procedure `omega` together with the standard propositional manipulation package `eauto`.

The advantage of Concoqtion over the languages with the built-in decision procedures is the support for greater logical expressiveness and graceful degradation: sometimes the proof that is needed cannot be constructed by any one decision procedure, but can be obtained by applying several such procedures, with minimal, but *necessary* guidance by the user. For example, if the DML-style type checker had to show that `'(x*x+2x+1)=(x+1)*(x+1)`, it would fail; in Concoqtion this can be proven in Coq as a theorem, added to the standard simpset, and let `eauto` use it.

Other proposals, including ATS [5], support programmer constructed proofs. The advantages of Concoqtion over ATS are primarily pragmatic: rather than requiring the programmer to build his proofs painstakingly from the ground up, using a relational notation, he can simply make use of the numerous libraries, proofs and decision procedures for Coq. In future versions of Concoqtion, we also plan to integrate support for "proofs" by assertions and runtime checks [11, 14, 24] as a kind of "rapid prototyping" for proofs.

2.3 The Hindley-Milner Type Inference

First an foremost, Concoqtion is backwards compatible with OCaml, so let-polymorphism continues to work as expected:

```
let id x = x
let _ = (id 1, id false)
let rec map x ls = ...
let _ = map (fun x → x) [1;2;3]
```

First-class polymorphism in Concoqtion is treated separately from let-polymorphism, with universal types separate from type schemes. Concoqtion provides introduction and elimination forms in the style of System F.

```
let f1 = Λa. fun (x:'(a)) → x
let f2 .|a| (x:'(a)) = x
let _ = f1.|int| 1
let _ = f2.|bool| false
let _ = f1 .|'(1+1)| 2 (* kind error! *)
let f (get : (∀ a. '(a) list →'(a))) =
  (get .|int| [1;2], get .|char| ['a','b','c'])
let church_num_list =
  let zero .|a| f (x:'(a)) = x in
  let one .|a| f x = f x in [zero; one]
let _ =
  (List.hd church_num_list) .|int| (succ) 0
```

Concoqtion can perform partial inference at type application: a programmer may omit the type argument and just indicate the location of the type application.

```
let _ = f1 .|| 1
```

The first-class polymorphism interacts with Hindley-Milner type inference in several ways. First, inference can propagate a universal type to where it is needed in a type application. Note that since we use an order-free inference algorithm, the annotations can propagate in many directions and non-locally. If a universal type cannot be propagated to the type application then the inferencer reports an error.

```
let _ = (fun f → f .|int| 1)(Λa.fun x:'(a) → x)
let _ = fun f → f .|int| 1 (* error !*)
let h (f : (∀ a. '(a) →'(a))) = f
let _ = fun g → let f x = g .|int| x in h g
```

In addition, let-polymorphic variables may be implicitly coerced to have a universal type. In the following, the function `h` expects a universally quantified argument.

```
let g x = x in h g
let succ x = x + 1 in h g (* error! *)
```

Tick types interact with type inference in interesting ways. Consider the following example that uses a type operator `f` to describe the type of the higher-order argument `z`.

```
(* choose : 'a →'a →'a *)
let choose x y = if true then x else y

let g .|f :'(ot →ot)|
    (z: ∀t. '(t) →'(f t)) (x:'a) (y:'b) =
  let _ = choose x y
  in choose (z .|'a| x) (z .|'a| y)
```

From `choose x y` the type inference algorithm deduces that the types of `x` and `y` are the same, that is `'a = 'b`. Applying `z` to `x` and `y` gives result types `f 'a` and `f 'b`. We pass the results to `choose`, so the inference algorithm needs to deduce that `f 'a = f 'b`, which in fact it does given that `'a = 'b`.

The following example, which appears as a motivating example for ML^F [13], is particularly challenging for type inferencers, and serves to demonstrate the order-free nature of the Concoqtion type inference algorithm. Note also that `auto` function below requires impredicative polymorphism, since `x` is type applied to a universal type.

```
type idT = ∀a. '(a) →'(a)
let auto (x : idT) = x .|idT| x
let _ = ((choose id) succ, (choose id) auto)
```

The interesting point is that the type inferencer should not choose how to instantiate `choose` too soon, i.e., at the first application to `id`. Instead, the inferencer needs to look forward to the second application, to `succ` in the first case and `auto` in the second. So `choose` should be instantiated with `'a = int→int` in the first case and `'a = idT→idT` in the second. The Concoqtion type inference succeeds on this example because it does not prematurely commit to a particular instantiation and instead proceeds to solve constraints in an order-free manner.

$$(\textsc{MonoVar})\ \frac{x \notin \mathrm{dom}(\Gamma')}{\Gamma, \textbf{fun}\ x : \tau, \Gamma' \vdash x : \tau} \qquad (\textsc{PolyVar})\ \frac{\tau = [\overline{\alpha_n \mapsto \tau_n}]\sigma \quad x \notin \mathrm{dom}(\Gamma') \quad \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{TV}(\Gamma) = \emptyset}{\Gamma, \textbf{let}\ x : \sigma, \Gamma' \vdash x : \tau}$$

$$(\textsc{App})\ \frac{\Gamma \vdash e : \sigma \to \tau \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash (e\ e') : \tau} \qquad (\textsc{Lam})\ \frac{\Gamma, \textbf{fun}\ x : \rho \vdash e : \sigma}{\Gamma \vdash (\textbf{fun}\ x \to e) : \rho \to \sigma} \qquad (\textsc{Let})\ \frac{\Gamma \vdash e : \rho \quad \Gamma, \textbf{let}\ x : \rho \vdash e' : \sigma}{\Gamma \vdash (\textbf{let}\ x = e\ \textbf{in}\ e') : \sigma}$$

Figure 2. Milner's type system.

3. Order-Free Hindley-Milner Type Inference

In this section we present a new sound and complete inference algorithm for the Hindley-Milner type system. We show how Hindley-Milner type inference can be reduced to (a) constructing a constraint graph based on an input term; (b) solving the constraint graph by means of applying three simple rewrite rules. This presentation of the algorithm forms the basis for extending the inference algorithm to Concoqtion: the constraint generation and solving is simply extended by adding new rules, but the core Hindley-Milner subsystem remains unchanged.

The syntax of OCaml-the-calculus is shown below.

x	$\in \mathbb{X}$		Term variables
α	$\in \mathbb{V}$		Type variables
g	$\in \mathbb{G}$	$\supseteq \{\to, \times, \texttt{int}, \texttt{bool}\}$	Type constructors
ρ, σ, τ	$::=$	$\alpha \mid g(\tau, \ldots, \tau)$	Types
e	$::=$	$x \mid \textbf{fun}\ x \to e \mid e\ e$	Terms
	$\mid$	$\textbf{let}\ x = e\ \textbf{in}\ e$	

We often use infix notation instead of prefix notation for types and drop parenthesis when the argument list is empty: $\texttt{int} \to \texttt{int}$ instead of $\to(\texttt{int}(), \texttt{int}())$.

We find Milner's original presentation [19] of the Hindley-Milner type system (Figure 2) better suited to our purposes than the more commonly used Damas-Milner formulation [10]. There are several differences between Milner's original formulation and the Damas-Milner variant: Milner's original differentiates between function and let-bound variables in the environment, so an environment Γ is a sequence of bindings of the form $\textbf{fun}\ x : \tau$ or $\textbf{let}\ x : \tau$. Generalization (creation of type schemes) is never explicitly performed and instantiation is folded into the rule for let-bound variables (PolyVar). An explicit syntactic form for type schemas is not necessary in this formulation. We write $\mathrm{TV}(\Gamma)$ for the type variables occurring in the types of environment Γ. The notation $[\overline{\alpha_n \mapsto \rho_n}]\tau$ stands for the type that results from simultaneously replacing all free occurrences of α_i in τ with ρ_i, for $i \in 1..n$.

3.1 Inferring Hindley-Milner Types by Graph Rewriting

Our type inference algorithm consists of two stages: constraint generation and constraint solving. Our algorithm takes as input a term graph [2] which has a type variable vertex for the type of each subterm and a box grouping together the types and constraints on the terms in the right-hand side of a let. This box corresponds to the constrained type schemas in the work of Pottier and Rémy [8].

The constraint generator then adds vertices and edges that represent types and that express the typing constraints. Type vertices are labeled with the appropriate type constructor, or with $\circ$ in the case of type variables. Constraints are represented by thick, red, undirected *equality edges* and by thick red directed *instantiation edges*. The constraint solver applies graph rewrite rules until no more rules apply. At that point, the graph will either be in a solved form or it will be erroneous.

[2] The term graph is trivially constructed from an expression by considering its abstract syntax tree as a graph.

Constraint generation. Figure 4 summarizes the constraint generation rules. The constraint generator takes as input a graph and returns an updated graph. In the graphical notation we use here, the rectangular vertices represent term AST vertices. When describing the constraint generation graphically, we use shadowing to represent the vertices that are created by the constraint generator. Similarly, the thick red edges are added by the constraint generator. The constraint generation rules are more formally defined in the accompanying technical report [23].

We walk through the steps for the following term:

```
let f = fun x →x in f 1
```

For each occurrence of a let-bound variable, such as the `f` in `f 1`, the generator creates an instantiation edge from the box for right-hand side of the let to the type of `f`. Moving outwards one subexpression, the application `f 1` requires its function to have a function type and that its domain equal the argument's type. This is expressed by creating a new vertices for a function type and for its *dom* and *cod*, and then placing an equality edge between the *dom* and the argument's type. Also, we add an equality edge connecting the type of `f` to the new function type.

For functions, such as `fun x →x`, we create a function type whose *dom* points to the type of the parameter and whose *cod* points to the type of the body. An equality edge is added to connect the type of the function to the new function type. A function-bound variable such as `x` are required to have the same type as the binding parameter, so an equality edge is added to connect them. The type of a let expression is the type of its body, which we express by connecting them with an equality edge. The constraint graph for this term is shown in Figure 3a).

Definition 1 (Hindley-Milner constraint generator). We write $G \Rightarrow G'$ whenever G' is derived by applying the rules of Figure 4 to all term vertices in the graph G.

Remark 1. The constraint generator terminates on all term graphs.

Constraint solving. The constraint solver consists of just three graph rewrite rules: unification, schema removal, and instantiation. These rewrite rules are graphically depicted in Figure 5. The first rewrite, *unification*, corresponds to the term reduction and variable elimination steps in Martelli and Montanari's classic unification algorithm [16]. The unification rule relies on the following two definitions.

Definition 2 (Vertex label compatibility). Two vertex labels a and b are compatible (written $a \approx b$) iff $a = b$ or $a = \circ$ or $b = \circ$.

Definition 3 (Label merging).

$$\mathrm{merge}(a, b) = \begin{cases} a & \text{if } a = b \vee b = \circ \\ b & \text{otherwise} \end{cases}$$

The unification rule applies when two vertices are connected by an equality edge and have matching labels, and merges the two vertices into one vertex. Also, for each pair of out-edges with the same label, it adds an equality edge connecting the targets of the two out-edges. Figure 3a contains five equality edges. Applying the unification rewrite to each of these results in Figure 3b.

The second rewrite rule, *instantiation* (Figure 5), removes an instantiation edge. The rule duplicates the box associated with the

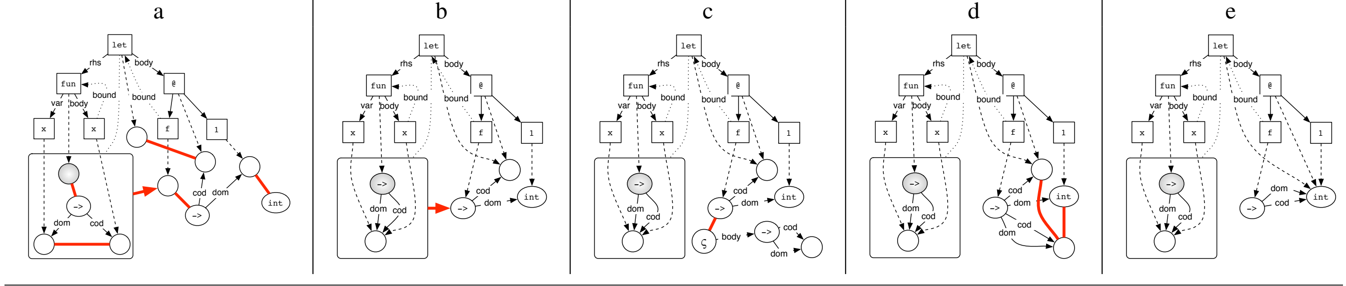

Figure 3. Constraint solving example.

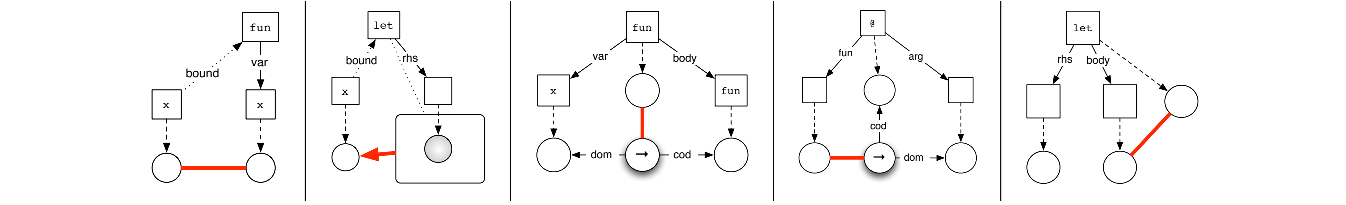

Figure 4. Constraint generation.

right-hand-side of the let (copying all the vertices an edges in the cloud) and then adds a type scheme vertex (labeled ς) and connects it to the root of the duplicated box with an edge labeled "body." Finally, it then adds an equality edge between the type scheme and the type of the variable. Duplicating the box includes duplicating all *edges* that connect vertices inside the box to vertices outside the box. For Hindley-Milner inference, the type scheme vertex is not strictly necessary, but it will serve a purpose once we extend the inference algorithm to Concoqtion. Applying the instantiation rewrite rule to the graph Figure 3b produces the graph in Figure 3c.

The type scheme rewrite rule removes a type scheme vertex when it is connected to a non-scheme vertex via an equality edge. We apply this rewrite followed by a unification rewrite to go from Figure 3c to Figure 3d. Two more applications of the unification rewrite produce the final Figure 3e.

Definition 4 (Hindley-Milner constraint solver). Apply the rewrite rules of Figure 5 in any order to any redex until there are no more redexes. As a post-processing step, any type scheme vertices (ς) are removed and replaced by their body vertex. We write $G \hookrightarrow G'$ if the constraint solver applied to G produces G'.

Remark 2. The graph rewriting system is confluent.

Definition 5 (Erroneous and Solved Graphs). If a graph G does not contain any redexes and there are red edges (equality or instantiation edges) then we call the graph G *erroneous*, *solved* otherwise.

An interesting aspect of our order-free inference algorithm is that, in particular, the unification rewrites are orthogonal to the instantiation rewrites. Unlike the inference algorithm of Pottier and Rémy [8], our algorithm can perform unification rewrites in the body of a let expression before solving all of the constraints in the right-hand side of the let. Note that from an efficiency standpoint, it is always safe to perform unification rewrites as they strictly decrease the amount of work to be done. On the other hand, one does need to take care with the order in which instantiation rewrites are performed to avoid poor performance (although the ordering doesn't matter for correctness and termination). To minimize work, one should always choose first an instantiation whose let's right-hand side is not the target of any other instantiation.

Our presentation immediately merges of nodes during unification, which is inefficient. The performance can be straightforwardly

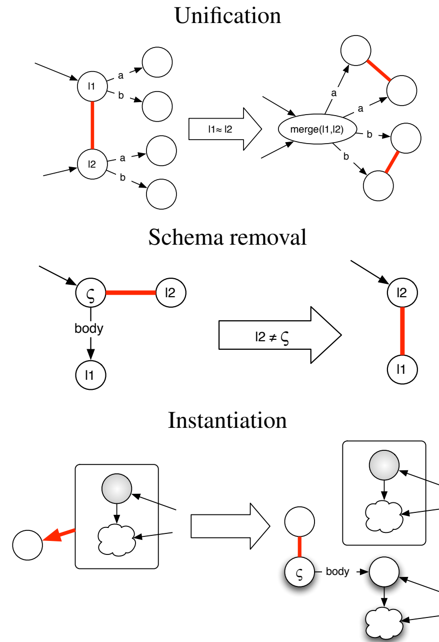

Figure 5. Constraint solving by graph rewriting.

increased by using the standard union-find data-structure and algorithm [33], which we do not show here as it would needlessly complicate the presentation.

3.2 Soundness and Completeness

In this section we present the properties of order-free Hindley-Milner type inference. Detailed proofs are in the accompanying technical report [23].

Definition 6. We write $(G, u) = \mathcal{G}(e)$ to create a graph G with a distinguished vertex u from the term e. The graph G includes a separate type variable vertex for each term. We write $\mathcal{T}(u)$ to

$$
\begin{array}{rcll}
c & ::= & \ldots \mid X \mid ?_\alpha \mid \;\mid \star \ldots & \text{Coq Terms} \\
\tau & ::= & \ldots & \text{Types} \\
 & \mid & \forall\, X\!:\!c.\ \tau & \text{Universals} \\
 & \mid & '\,(c) & \text{Tick types} \\
\gamma & ::= & & \text{Type Patterns} \\
 & \mid & \alpha & \text{Existential type variables} \\
 & \mid & '\,X\!:\!c & \\
 & \mid & g(\gamma, \ldots, \gamma) & \text{Type constructors} \\
C & \in \mathbb{C} & & \text{Constructors} \\
e & ::= & \ldots & \text{Terms} \\
 & \mid & \Lambda\, X\!:\!c.\ e & \text{Type abstraction} \\
 & \mid & e.|\tau| & \text{Type application} \\
 & \mid & C.|\tau, \ldots|(e_1, \ldots, e_n) & \text{GADTs} \\
 & \mid & e : \tau & \text{Type ascription} \\
 & \mid & \textbf{match}\ e\ \texttt{as}\ \gamma\!:\!\tau\ \texttt{with} & \text{Case} \\
 & & \quad \overline{\pi \to e} & \\
\pi & ::= & & \text{Patterns} \\
 & \mid & x & \\
 & \mid & C.|\alpha\!:\!c, \ldots|(\pi, \ldots) &
\end{array}
$$

Figure 6. Syntax: Concoqtion extensions

convert from the graph representation of a type (rooted at u) to the corresponding type.

Theorem 1. *(Type inference terminates) If $(G, u) = \mathcal{G}(e)$ and $G \Rightarrow G'$ then $G \hookrightarrow G'$.*

Theorem 2. *(Type inference is sound) Let $(G, u) = \mathcal{G}(e)$ and $G \Rightarrow G'$. If $G' \hookrightarrow G''$ and G'' is solved then $\emptyset \vdash e : \mathcal{T}(type(u))$.*

Theorem 3. *(Type inference is complete) Suppose $\emptyset \vdash e : \tau$ and let $(G, u) = \mathcal{G}(e)$ and let $G \Rightarrow G'$. Then $G' \hookrightarrow G''$ and G'' is solved and there is a substitution S where $S(\mathcal{T}(type(u))) = \tau$.*

4. Formalizing Core Concoqtion

In this section, we formalize a core calculus for Concoqtion and extend our order-free type inference algorithm to handle the additional language features: tick types, first-class polymorphism, and extended data types. We characterize our inference algorithm by studying its relationship with two type systems.

The first type system is for a subset of Concoqtion. The only restriction to terms is that the expression in a type application (universal elimination) must be fully annotated. We call this subset E-Annotated Concoqtion. The type system for E-Annotated Concoqtion has the advantage of being described naturally as a type system that results directly from combining the rules of a HM language and F_ω. We present an inference algorithm and show that it is sound and complete for this system.

The second type system is for the whole of Concoqtion, allowing programmers to omit some type annotations in type applications. This second type system precisely defines the well-typed terms of Concoqtion. We show that the same inference algorithm is sound and complete with respect to this type system

4.1 Syntax

The definition of Concoqtion's syntax assumes that we can refer to an existing syntax for *Coq terms*. It further assumes that Coq term syntax includes three specific productions: X to denote the use of a Coq variable, $?_\alpha$ to denote the use of a constant relating to a type variable in the Hindley-Milner world, and $\star$ to denote the universe of Hindley-Milner types. In other words, $\star$ will represent the Concoqtion kind for Hindley-Milner types, which we called **`ot`** in examples that used Concoqtion's concrete syntax.

These requirements are expressed in the first line in Figure 6. In reality, of course, the set of Coq terms is a powerful and expressive language in its own right (cf. [34]), but these are the only relevant syntactic assumptions needed to define this extension.

The rest of the figure defines the syntax for the calculus that we will use for the meta-theoretic treatment in the rest of this paper. Concoqtion types extend the types in an underlying Hindley-Milner language, modeled in this case using types introduced in Section 3. Concoqtion only adds two new productions to types: $\forall X : c.\tau$ to represent universal quantification in the style of System F, and $'\,(c)$ to include arbitrary Coq terms in Concoqtion types. Note that kind annotations are Coq terms. For convenience, kind annotations will ocassionally be omitted, in which case the kind is assumed to be $\star$.

The *Terms* of Concoqtion are similarly an extension of those of the Hindley-Milner language. For the type $\forall X : c.\tau$, we provide $\Lambda\ X\!:\!c.\ e$ as the introduction construct, and $e.|\tau|$ for elimination. For E-Annotated Concoqtion, the elimination form is $(e : \forall X.\tau).|\tau|$. The type inference algorithm would be incomplete with respect to the E-Annotated Type System if this restriction were removed. In particular, this type system would require higher-order unification to infer it in all cases. At the same time, it should be noted that full annotation is overly conservative, since it can often be inferred from other parts of the program via our order-free inference algorithm. In fact, these annotations are not required in the source Concoqtion language. Precisely specifying when these annotations can be omitted is nontrivial and the topic of Section 4.5. With this in mind, we present the fully annotated system here because it allows us to express important aspects of our formalism without overwhelming the reader with detail of the meta-theory of the formulation presented in the next section.

Concoqtion Terms also include support for introducing and consuming values of a generalized algebraic datatype (GADT). The data constructor $C.|\tau, \ldots|(e_1, \ldots, e_n)$ follows the OCaml notational convention, but is extended with a type application that denotes the type application of any locally bound type variables in the type of the data constructor (See for example the constructors of `listN` in Figure 1a). The **`match`** e `as` $\gamma\!:\!\tau$ `with` $\cdots$ expression extends typical pattern matching used in a Hindley-Milner system. The type pattern γ is used to bind the type indexes of the discriminated expression type to a set of Coq type variables in scope of the type τ. The type τ itself specifies the return type of the **`match`** (See for example `app` in Section **??**). The pattern (π) for each case binds local type variables as specified in the data constructors.

4.2 A Type System for E-Annotated Concoqtion

Figure 7 presents the type system for the core calculus of Concoqtion. First, we assume that there is a signature Σ which provides the types of the data-constructors and kinds of type constructors. The judgment $\Delta; \Gamma \vdash e : \tau$ takes a type assignment Γ (as in Figure 2) as well as a Coq type assignment Δ.

Universal quantification. The Coq type assignment Δ is extended by the TABS rule when a new Coq variable X is bound. The auxiliary judgment $\Delta \vdash_C c : \texttt{Set}$ (not shown here) is the Coq typing judgment for constructions; here it is used as kinding judgment for the type variable X.

Conversion The conversion rule TCONV uses the Coq convertibility judgment $\Delta \vdash \cdot = \cdot$ to compare types for equality. This is done by first representing the two types τ_1 and τ_2 as Coq constructions. In this way, we can reuse Coq's notion of convertibility without having to explicitly define it for Concoqtion types.

Definition 7 (Forth and back). There are two functions, forth ($F(\cdot) : \tau \to c$) and back ($B(\cdot) : c \to \tau$) which take a type τ and convert it to a Coq construction, of type $\star$and vice-versa. Back and forth are inverses of each other ([23]).

$$\boxed{\Delta;\Gamma \vdash e : \tau}$$

All the rules from Figure 2 modified to accept but ignore Δ.

$$(\textsc{TAbs})\ \frac{\Delta, X;\Gamma \vdash e : \tau \quad \Delta \vdash_C c : \texttt{Set}}{\Delta;\Gamma \vdash (\Lambda X\!:\!c.\ e) : \forall\ X\!:\!c.\ \tau} \qquad (\textsc{TApp})\ \frac{\Delta;\Gamma \vdash_C \tau : c \quad \Delta;\Gamma \vdash e : \rho \quad \rho = \forall X\!:\!c.\ \sigma}{\Delta;\Gamma \vdash ((e\!:\!\rho)\texttt{.|}\tau\texttt{|}) : \sigma[X \mapsto \tau]}$$

$$(\textsc{TConv})\ \frac{\Delta;\Gamma \vdash e : \tau_1 \quad \Delta \vdash_C F(\tau_1) = F(\tau_2)}{\Delta;\Gamma \vdash e : \tau_2} \qquad (\textsc{TAscr})\ \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash (e : \tau) : \tau}$$

$$(\textsc{TCons})\ \frac{\Sigma(C) = (Y_0 : c_0, \ldots, Y_n : c_k)C\texttt{.|}\overline{X : c}\texttt{|}\ \texttt{of}\ \overline{\rho} : \rho' \quad \Delta;\Gamma \vdash_C F(\tau_i) : c_i \text{ for } i = 1 \ldots n \quad \Delta;\Gamma \vdash e_j : \rho_j[\overline{X}/\overline{\tau}] \text{ for } j = 1 \ldots m}{\Delta;\Gamma \vdash C\texttt{.|}\tau_1, \ldots, \tau_n\texttt{|}(e_1, \ldots, e_m) : \rho'[\overline{X}/\overline{\tau}]}$$

$$(\textsc{TMatch})\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e : g(\tau_1, \ldots, \tau_n) \quad \Sigma(g) = (c_1, \ldots, c_n) \\ \Delta \vdash_C F(\tau_i) : c_i \text{ for } i = 1 \ldots n \\ S(\gamma) = g(\tau_1, \ldots, \tau_n) \\ \Delta \vdash \gamma \qquad \Delta \vdash_C S(\tau) : \star \\ \Delta;\Gamma;\gamma;g(\tau_1, \ldots, \tau_n);\tau \vdash \pi_j \texttt{ -> } e_j \text{ for } j = 1 \ldots m\end{array}}{\Delta;\Gamma \vdash \textbf{match}\ e\ \texttt{as}\ \gamma\!:\!\tau\ \texttt{with}\ \overline{(\pi \texttt{ -> } e)}_m : S(\tau)}$$

$$\boxed{\Delta;\Gamma;\gamma;\tau;\tau \vdash \pi \texttt{ -> } e}$$

$$\frac{\Delta;\Gamma \vdash \pi : \tau \Rightarrow \varsigma;\Delta';\Gamma' \quad S'(\gamma) = \varsigma \quad \Delta,\Delta';\Gamma,\Gamma' \vdash e : S'(\tau_r)}{\Delta;\Gamma;\gamma;\tau;\tau_r \vdash \pi \texttt{ -> } e}$$

$$\boxed{\Delta;\Gamma \vdash \pi : \tau \Rightarrow \varsigma;\Delta;\Gamma}$$

$$\frac{}{\Delta;\Gamma \vdash x : \tau \Rightarrow \tau;\cdot;x : \tau}$$

$$\frac{\Sigma(C) = (\overline{Z_k : c_k})C\texttt{.|}\overline{Y_m : c'_m}\texttt{|}\ \texttt{of}\ \overline{\rho}_n : g(\overline{\tau'_k}) \quad \Delta, \overline{X : c'};\Gamma \vdash \pi_j : \rho_j[\overline{Z}/\overline{\tau}, \overline{Y}/\overline{X}] \Rightarrow \varsigma_j;\Delta'_j;\Gamma'_j \text{ for } j = 1 \ldots n}{\Delta;\Gamma \vdash C\texttt{.|}\overline{X}\texttt{|}(\overline{\pi}) : g(\overline{\tau}) \Rightarrow g(\overline{\varsigma_j});\cup_j \Delta'_j;\cup_j \Gamma'_j}$$

$$\boxed{\Delta \vdash_C \tau = \tau} \qquad \boxed{\Delta \vdash_C c : c}$$

Figure 7. Type system for E-Annotated Concoqtion.

Definition 8 (Predefined Coq constants). For each type constructor f, there exists a Coq constant, named `OT_f`, of appropriate type, which represents it in Coq. For example, the $\forall$ type is represented by the constant `OT_∀: ∀A:Set, (A →ot) →ot`.

Pattern matching. In Concoqtion pattern matching, additional assumptions about the type $g(\tau_1, \ldots, \tau_n)$ of the discriminated expression are introduced into the environment of each branch. For example, recall the function `app` of Figure 1a: the type checker knows that type parameter `n` is `'(0)` in the `Nil` branch.

The declaration $\cdots$ `as` $\gamma : \tau \cdots$ specifies *how* the types of each branch may vary depending on this extra contextual information introduced by the patterns. The type pattern γ binds type variables that are used in τ, and is matched against the the discriminated type $g(\tau_1, \ldots, \tau_n)$; the resulting substitution is applied to the return type τ to determine the overall result type of the **match** statement.

Let us look more closely at the rule TMatch. The first two lines of antecedents simply check the discriminated expression, and make sure that its type is well-kinded with respect to the signature of the type constructor g. Next, the type pattern γ is matched against the type of the discriminated expression to obtain a substitution S which ranges over the type variables in the type pattern γ. The next line ensures that the specified return type is well kinded. The result of the entire match statement is the substitution S applied to the declared return type τ.

Each case is checked with the auxiliary judgment

$$\Delta;\Gamma;\gamma;\tau_{in};\tau \vdash \pi \texttt{ -> } e$$

This judgment first determines the type of the pattern based on the signature of data-constructors used in the patterns and matches this type against the type pattern γ, obtaining the local context substitution S'. The result of the branch is then $S'(\tau)$.

4.3 Type inference for Concoqtion

With an order-free inference algorithm established for the OCaml core (Section 3), it becomes straightforward to extend the inference algorithm to handle the language features of Concoqtion: tick types, first-class polymorphism, and dependent data types. We extend the algorithm by adding constraint generations rules for each of the new syntactic forms, several new kinds of constraint edges, and several graph rewrite rules.

Constraint generation. Figure 8 shows the new cases for constraint generation.

Type ascription rule simply introduces an equality edge between the type of the expression `e` and the type annotation `t`. The rule for type abstraction inserts a universal type and an equality edge connecting it to the type of the type abstraction. The rule for type application inserts a *universal instantiation edge* from the type of `e` to the type of the type application expression. The instantiation edge also points to the type argument, which will be needed for the universal instantiation rewrite rule. The rule for data-type constructor application finds the signature for the constructor `C` in the environment and then substitutes for each type parameter the given type arguments. This produces the gray clouds shown in the diagram. We add equality edges for each of the arguments and for the return type.

Finally, we come to the **match** expression. Constraint generation for **match** is rather involved but not inherently difficult. Recall that the unusual aspect of a type-indexed match is in each one of the cases new information is introduced to the local scope by the pattern and the result type of each case may be different. The result type of the match itself is independent of the cases (the local information should not escape), and instead is specified by the annotation γ and τ. The result type of the match is the annotation τ, but with the type-indexed variables inferred from matching the type of e with γ. We insert an equality edge between the type of e and γ and between a copy of τ and the type of the match.

For each match case, we construct the subgraph on the right: from the signature of the data-type and the pattern π_i we construct the pattern type (a cloud inside a dashed boundary). This type is a "more specific" version of the discriminated expression e's type in that additional information may be obtained from the structure of the pattern and the signatures of the data-constructors that appear in them. For example, in Figure 1 we know that the `Nil` pattern makes the type parameter `n` equal to `'(0)`. Additionally, the types produced from the pattern depend on the type of the discriminated expression (but not the other way around). To accomplish a one-way transfer of information, we create a new kind of edge, the *match edge*. The pattern type thus constructed (represented as a cloud inside a dashed rectangle) type is unified with the γ to get the bindings for the type-index variables in τ which provides a return

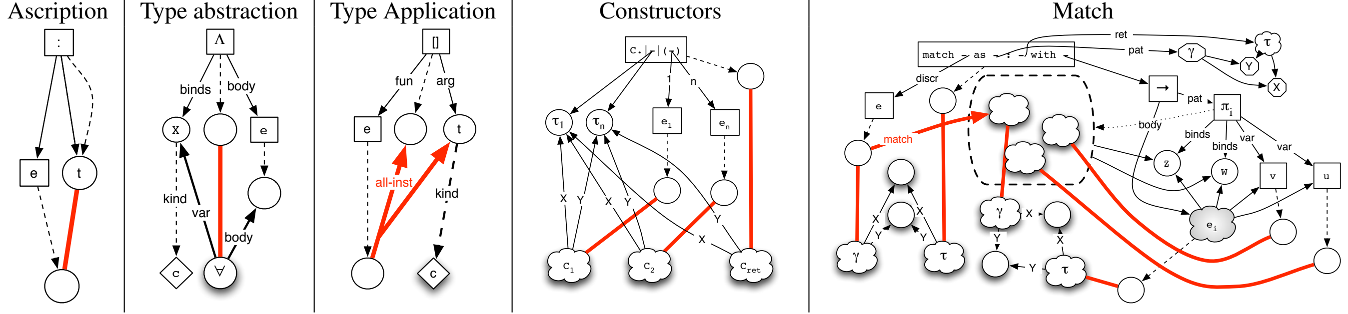

Figure 8. Generating the constraint graph for Concoqtion. We add the cases for type ascription, type abstraction and type application, (data) constructor application and pattern matching to the cases in Figure 4.

type specific to the particular case. The return type be equal with the type of the body of the case (e_i), so we draw an equality edge between them.

Definition 9 (Concoqtion constraint generator). Apply the rules of Figure 8 to each term vertex in the graph G to create a new graph G'. We write $G \Rightarrow_{cq} G'$ for this process.

Constraint Solving Figure 9 shows the additional rewriting rules used by the Concoqtion constraint solver.

Tick type rules. The top four rules deal with tick types. First, if a tick type contains an application of a type constructor (a Coq `OT_` constant), we replace the tick with with add a vertex for the type constructor and tick types for the arguments. Second, a Coq existential variable, which is used to represent a Hindley-Milner type variable, is expanded to the actual type variable vertex it represents. Third, the representation of a universal type is expanded to a $\forall$ vertex with a tick type in the body. Finally, if a construction c_1 can be reduced (using the semantics of Coq) to the construction c_2 (where $c_1 \not\equiv c_2$), then we update the tick type to c_2. This allows the constructions to be evaluated to weak-head normal form, enabling other rewrites to fire.

The Coq construction in a tick type is the label of the vertex. Thus, two ticked type vertices connected by an equality edge are merged only if they are equal (i.e., convertible in Coq). Otherwise, the unification rule does not apply until either both tick types are expanded to other kinds of vertices, or until they become equal through the application of the fourth tick rewrite rule. If the solver can never remove the equality edge, the graph is (as defined before) *erroneous*.

Type application. This rewrite rule eliminates an all-inst edge. The body of the universal type is copied, with edges connecting to the type parameter redirected to the type argument. We add an equality edge connecting the new body with the target vertex of the all-inst edge.

Universal/Scheme merging. This rewrite rule allows for interoperability between type schemes introduced by let-polymorphism and explicit universal types. This rewrite rule applies when there is an equality edge connecting a universal type and a type scheme. The rewrite merges the type scheme vertex into the vertex for the universal type and adds an equality edge between the targets of their respective *body* out-edges.

Matching. The two rules for `match` edges are similar to unification, except that information is allowed to flow only one way: if the source and target labels match, the match edges are propagated to their children, as with the unification rewrite rule. If the target edge is an empty vertex (type variable) then the target is replaced by a copy of the source.

Definition 10 (Concoqtion constraint solver). Apply the rewrite rules of Figure 9 in any order to any redex until there are no more

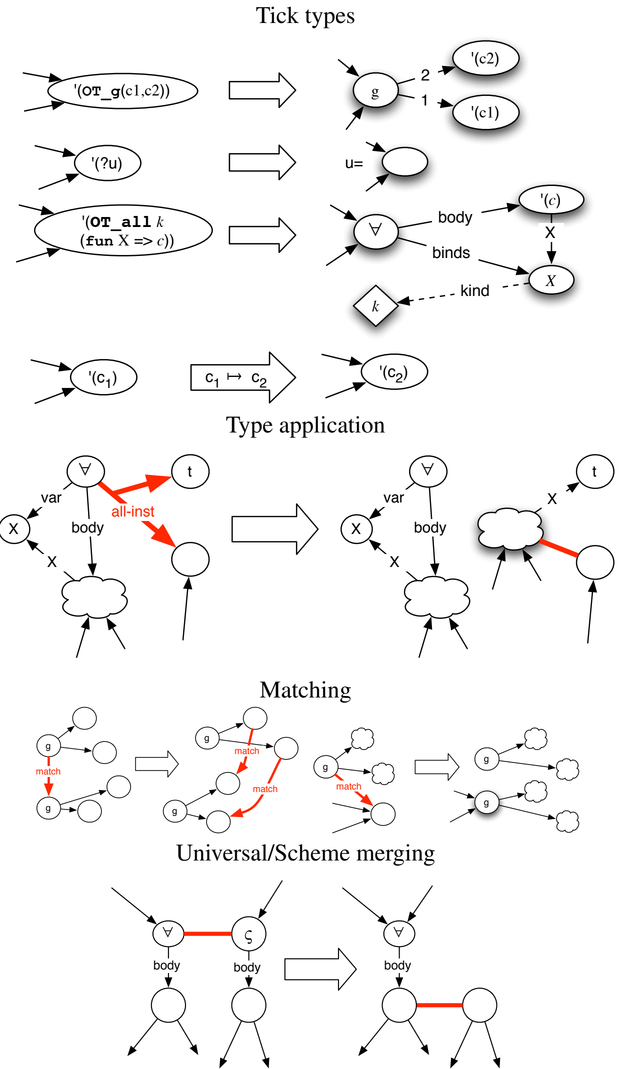

Figure 9. Graph rewrite rules for Concoqtion constraint solver

redexes. As a post-processing step, any type scheme vertices (ς) are removed and replaced by their body vertex. We write $G \hookrightarrow_{cq} G'$ if the constraint solver applied to G produces G'.

4.4 Properties of Concoqtion Type Inference

The Concoqtion type inferencer terminates and it is sound and complete with respect to the type system of Core Concoqtion (Figure 7).

Theorem 4. *(Type inference is sound) Let* $(G, u) = \mathcal{G}(e)$ *and* $G \Rightarrow_{cq} G'$. *If* $G' \hookrightarrow_{cq} G''$ *and* G'' *is solved then* $\emptyset; \emptyset \vdash e : \mathcal{T}(\mathit{type}(u))$.

Theorem 5. *(Type inference terminates) If* $(G, u) = \mathcal{G}(e)$ *and* $G \Rightarrow_{cq} G'$ *then* $G \hookrightarrow_{cq} G'$.

Theorem 6. *(Type inference is complete) Suppose* $\emptyset; \emptyset \vdash e : \tau$ *and let* $(G, u) = \mathcal{G}(e)$ *and let* $G \Rightarrow_{cq} G'$. *Then* $G' \hookrightarrow_{cq} G''$ *and* G'' *is solved and there is a substitution* S *where* $S(\mathcal{T}(\mathit{type}(u))) = \tau$.

4.5 A Type System for Concoqtion

The language Concoqtion does not require type applications to be annotated with the type of e, as in required by the conservative type system. It is an interesting challenge to create a type system (specification) that defines the well-typed terms of Concoqtion in a way that exactly coincides with the Concoqtion type inferencer. Naively adapting the type system of Figure 7 to cover un-annotated type applications is a dead end because that type system would be too powerful: it has the ability guess polymorphic types. Instead we need to level the playing field and define a type system that propagates types according to the rules of Hindley-Milner, but does not guess types.

The relationship between a type inference algorithm and a type system can be viewed as follows: the inferencer produces a *certificate* for the solution, and the type system *verifies* the certificate. Traditionally, this means that the inferencer produces a type assignment and the type system checks whether the type assignment is valid. For our precise type system, we change the notion of certificate: the inference algorithm produces proofs that shows why terms have particular types. The type system is simply the proof theory (collection of inference rules) that specifies what constitutes a valid proof that one type equals another type.

The judgment $\Gamma \vdash e_\alpha \Rightarrow E$ produces a set of equations E from a term where each subterm is labeled with a unique type variable to serve as an unknown for its type. The judgment $E \vdash \tau = \tau$ defines the inference rules for types equality under the equations E. The rules for both judgments are defined in Figure 10.

An expression e_α is well-typed with τ if the set of equations is *consistent* (there is no proof of false) and if τ is the most specific type with a proof $E \vdash \alpha = \tau$. The reason we say "most specific", and not just any type, is that trivially we have $E \vdash \alpha = \alpha$, but we do not want to say that α is the type of e.

Definition 11. (Concoqtion's Type System) We say that a Concoqtion term e is well-typed with τ, written $\vdash_{cq} e : \tau$ if $\vdash_{cq} e_\alpha \Rightarrow E$ and $E \not\vdash \bot$ and $\tau = \max\{\sigma \mid E \vdash_{cq} \alpha = \sigma\}$.

Theorem 7. *(Concoqtion type inference is sound) Let* $(G, u) = \mathcal{G}(e)$ *and* $G \Rightarrow_{cq} G'$. *If* $G' \hookrightarrow_{cq} G''$ *and* G'' *is solved then* $\vdash_{cq} e : \mathcal{T}(\mathit{type}(u))$.

Theorem 8. *(Concoqtion type inference is complete) Suppose* $\vdash_{cq} e : \tau$ *and let* $(G, u) = \mathcal{G}(e)$ *and* $G \Rightarrow_{cq} G'$. *Then* $G' \hookrightarrow_{cq} G''$ *and* G'' *is solved and* $\mathcal{T}(\mathit{type}(u)) = \tau$ *up to renaming free type variables.*

5. Related work

Full spectrum dependent types Several lines of research involve enriching type systems by enabling types to depend on values. One approach involves the notion of *full spectrum type dependency* [18] in programming languages that are based on type theories [9, 15, for example]. In this approach, types may depend on arbitrary terms, thereby intermingling types and terms in a single language. Cayenne [2] and Epigram [17] are examples of such programming languages. The down side of full spectrum dependent typing is that the language must be strongly-normalizing (and therefore *not* Turing complete) to maintain decidability of type checking, and the language must be pure (effect free) to maintain type soundness.

Generalized algebraic datatypes (GADTs) GADTs extend the standard notion of algebraic data-types so that the type of a data value may depend on what constructor is used to build it. This basic form of dependency can be used to encode a surprising variety of properties [4]. One of the first versions of GADTs was introduced by Xi (called *guarded recursive data-types* [37]). Integrating generalized algebraic data-types with ML-style type inference has been the focus of recent research (Pottier and Régis-Gianas [7], Stuckey and Sulzmann [32], and Vytiniotis, Weirich, and Peyton Jones [36]). Several languages with type systems directly supporting GADTs have been proposed, such as First-Class Phantom Types of Cheney and Hinze [6], extensions to the GHC Haskell compiler [36], and Ωmega of Sheard [31].

The advantages of GADTs are they are a natural extension of existing language features and type inference can be extended to include them. The down side of GADTs is that to represent complex invariants, one must encode them in many new data-types. The proofs of such invariants must be constructed and manipulated by programs, as values of those data-types. This can lead to complex and verbose programs, as we show in Section 2.

Inference for First-Class Polymorphism Pfenning defines a partial inference algorithm [27] for F_ω that infers types for type application but still requires the location of type application to be annotated, as is the case for our system. Pfenning's algorithm guesses universal types, reducing the problem to higher-order unification, which is undecidable, but there are semi-decision procedures. In contrast, our algorithm does not guess universal types.

Local type inference, as developed by Pierce and Turner [28] and later refined by Odersky, Zenger, and Zenger [22] is not conservative over ML programs, so it is not directly suitable for Concoqtion, which must remain backwards compatible with OCaml.

Odersky and Läufer [20] define an extension of the Hindley-Milner type system with predicative first-class polymorphism. Their algorithm infers the introduction and elimination of first-class polymorphism and they allow for polymorphic coercions (via instantiation) under function arrows. Their algorithm reduces the type inference problem to first-order unification under a mixed prefix. Odersky and Läufer do not treat type operators or impredicative polymorphism, as we do in this work.

Garrigue and Rémy [12] conservatively extend ML with first-class polymorphism, with explicit introduction and semi-explicit elimination forms, and they distinguish between type schemes and universal types, as we do. However, their inference algorithm is not order-free, so their algorithm rejects the following program, whereas ours accepts.

```
fun f →if true then f.|| f
       else (f : ∀a. '(a) →'(a))
```

Garrigue and Rémy do not address type operators in this work.

Le Botlan and Rémy [13] define an extension to the Hindley-Milner type system with impredicative first-class polymorphism, called ML^F. Their type inferencer infers type applications, whereas in our system the programmer must identify the location of type applications. On the other hand, our inferencer handles type operators while ML^F does not. Our inferencer is similar to ML^F in

Equation Generation $\Gamma \vdash e_\alpha \Rightarrow E$

$$\frac{x \notin \mathrm{dom}(\Gamma')}{\Gamma, \mathbf{fun}\ x_\alpha, \Gamma' \vdash x_\beta \Rightarrow \{\alpha = \beta\}} \quad \frac{x \notin \mathrm{dom}(\Gamma') \quad \mathrm{dom}(S) \cap \mathrm{FTV}(\Gamma) = \emptyset}{\Gamma, \mathbf{let}\ x_\alpha, \Gamma' \vdash x_\beta \Rightarrow \{S(\alpha) = \beta\}} \quad \frac{\Gamma, \mathbf{fun}\ x_\alpha \vdash e_\beta \Rightarrow E}{\Gamma \vdash (\mathbf{fun}\ x_\alpha \rightarrow e_\beta)_\gamma \Rightarrow E \cup \{\gamma = \alpha \rightarrow \beta\}}$$

$$\frac{\Gamma \vdash e_\alpha \Rightarrow E_1 \quad \Gamma \vdash e'_\beta \Rightarrow E_2}{\Gamma \vdash (e_\alpha\ e'_\beta)_\gamma \Rightarrow E_1 \cup E_2 \cup \{\alpha = \beta \rightarrow \gamma\}} \quad \frac{\Gamma \vdash e_\alpha \Rightarrow E \quad \Gamma, \mathbf{let}\ x_\alpha \vdash e'_\beta \Rightarrow E'}{\Gamma \vdash (\mathbf{let}\ x = e_\alpha\ \mathbf{in}\ e'_\beta)_\gamma \Rightarrow E \cup E' \cup \{\beta = \gamma\}} \quad \frac{\Gamma, X : c \vdash_{cq} e_\alpha \Rightarrow E}{\Gamma \vdash_{cq} (\Lambda\, X{:}c.\ e_\alpha)_{\forall\, X:c.\ \alpha} \Rightarrow E}$$

$$\frac{\Gamma \vdash_{cq} e_\alpha \Rightarrow E}{\Gamma \vdash_{cq} (e_\alpha\,.\,|\,\tau\,|)_\beta \Rightarrow E \cup \{(\alpha = \forall\, X{:}c.\ \sigma \Longrightarrow \beta = \sigma[X \mapsto \tau])\}} \quad \frac{\Gamma \vdash e_\alpha \Rightarrow E}{\Gamma \vdash (e_\alpha{:}\tau)_\gamma \Rightarrow E \cup \{\gamma = \tau, \alpha = \tau\}} \quad \ldots$$

Equational Theory of Types $E \vdash \tau = \tau$

$$(\mathrm{AX})\frac{\tau_1 = \tau_2 \in E}{E \vdash \tau_1 = \tau_2} \quad (\mathrm{REFL})\frac{}{E \vdash \tau = \tau} \quad (\mathrm{SYM})\frac{E \vdash m_1 = \sigma = \tau = m_2}{E \vdash m_1 = \tau = \sigma = m_2} \quad (\mathrm{TRANS})\frac{E \vdash \tau = m_1 \quad E \vdash \tau = m_2}{E \vdash \tau = m_1 = m_2}$$

$$(\mathrm{UNIFY})\frac{E \vdash g(\tau_1, \ldots, \tau_n) = g(\sigma_1, \ldots, \sigma_n) = m}{E \vdash \tau_i = \sigma_i} \quad (\mathrm{S\text{-}FUN})\frac{E \vdash \tau_1 = \tau_2 = m}{E \vdash S(\tau_1) = S(\tau_2)} \quad (\mathrm{S\text{-}CONG})\frac{E \vdash S(g(\overline{\tau_n})) = g(\overline{\sigma_n}) = m}{E \vdash S(\tau_i) = \sigma_i}$$

$$(\mathrm{S\text{-}BND})\frac{\alpha \notin \mathrm{dom}(S)}{E \vdash S(\alpha) = \alpha} \quad (\mathrm{ERROR})\frac{E \vdash g(\tau_1, \ldots, \tau_n) = g'(\sigma_1, \ldots, \sigma_n) \quad g \neq g'}{E \vdash \bot} \quad (\mathrm{MP})\frac{(\tau_1 = \tau_2 \Longrightarrow \tau_3 = \tau_4) \in E \quad E \vdash_{cq} \tau_1 = \tau_2}{E \vdash_{cq} \tau_3 = \tau_4}$$

Figure 10. Precise Type System for Concoqtion

that both have the ability to postpone solving constraints until more information becomes available, as we showed in Section 2.3. The ML^F inferencer achieves this by enhancing universal types with constraints which can summarize partial solutions. In contrast, our approach uses standard universal types but uses an order-free inference algorithm.

Vytiniotis, Weirich, and Peyton Jones introduce Boxy Types [36] as a way to integrate impredicative first-class polymorphism with the Hindley-Milner type system. They do not address type operators. Their system uses local type inference to infer type applications at function applications and to propagate annotations. Our system does not infer type applications but it propagates annotations to more locations. For example, with Boxy types, only monotypes are inferred for function parameters when the function is processed in inference mode. So, for example, they do not infer a type for `g` in the following example, whereas the Concoqtion inferencer does infer a type for `g`.

```
let h (f : (∀ a. '(a) →'(a))) = f
let _ = fun g →h g
```

They show that their inference algorithm is sound and complete with respect to their "boxy" type system. We plan to investigate the relationship between their type system and the type system of Concoqtion.

Peyton Jones, Vytiniotis, Weirich, and Shields [25] in earlier work integrated predicative first-class polymorphism with the Hindley-Milner type system using the system of Odersky and Läufer as their starting point. As in the above, they use local type inference to propagate constraints, and so this system also does not infer a type for the above program.

Rémy designed a stratified approach to integrating impredicative first-class polymorphism with Hindley-Milner inference that is conservative over ML [29]. The first phase performs "shape" inference and the second phase performs first-order type inference. His algorithm infers predicative type applications but not impredicative type applications. Additionally, it allows for polymorphic coercions (via instantiation) under function arrows, according to the type containment relation, whereas ours does not. Rémy's system does not treat type operators and his inference algorithm is not complete.

6. Conclusion

The Concoqtion project aims to produce an industrial-strength programming language with indexed types that allow the programmer to cleanly express, prove and statically check important invariants of real OCaml programs. The implementation of Concoqtion consists of a modified OCaml compiler that includes the Coq proof assistant as a subcomponent and allows the programmer to harness the power and convenience of Coq in type-checking his programs. A prototype implementation of Concoqtion (and the accompanying technical report [23]) is available online [1].

We consider it a critically important goal for Concoqtion to be fully backward compatible with existing OCaml code bases. Satisfying our design goals requires addressing a key technical problem: sound and complete integration of Hindley-Milner type inference with the Calculus of Inductive constructions. We propose a solution to this problem, and prove it correct. The key insight for carrying this out was to formulate an order-free Hindley-Milner inference algorithm based on a small number of graph-rewriting rules.

6.1 Future work.

We single out some important directions for future work.

Syntax and type annotations. Our type inference algorithm can in practice make the explicit type annotation burden surprisingly light (see the Church numeral example in Section 2.3). However, we have done little explicitly to address the sometimes redundant type annotations required by **`match`** expressions, as was done in the work of Pottier and Régis-Gianas [7]. Removing the need for some of these annotation will make programming with indexed types easier.

Singleton types. One notable feature of Concoqtion programs is that they often have to duplicate information in both the computational and logical worlds. For example, natural numbers indexed by Coq naturals are encoded as the following Concoqtion data-type:

```
type 'n:'(nat) mynat = Z : '(0) mynat
 | S of let 'm:'(nat) in '(m) mynat : '(S m) mynat
```

This leads to the programmer having to duplicate much work in reflecting Coq sets to the computational level. We have experimented with an automatic way to derive singleton types such as `mynat` from Coq inductive families to spare the programmer from having to do it manually. The problem, however, is that in manually defin-

ing such types, the programmer can chose to omit unnecessary information from the type indexes, encoding only the properties he really needs.

Decision procedures. Exploring further ways to make Coq decision procedures available to the programmer is a very important future direction

References

[1] The Concoqtion web site. Online at http://www.metaocaml.org/concoqtion, July 2006.

[2] L. Augustsson. Cayenne: a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

[3] L. Cardelli. The quest language and system, 1994.

[4] C. Chen, R. Shi, and H. Xi. Implementing Typeful Program Transformations. *Fundamenta Informaticae*, 69(1-2):103–121, 2005.

[5] C. Chen and H. Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, New York, NY, USA, 2005. ACM Press.

[6] J. Cheney and R. Hinze. Phantom types, May 25 2003.

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

[8] F. cois Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005. A draft extended version is also available.

[9] T. Coquand and G. Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. Colloque de Logique, Orsay (July 1985), North-Holland, forthcoming.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[11] C. Flanagan. Hybrid type checking. In *POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, Charleston, South Carolina, January 2006.

[12] J. Garrigue and D. Rémy. Semi-explicit first-class polymorphism for ml. *Information and Computation*, 155(1-2):134–169, 1999.

[13] D. Le Botlan and D. Rémy. MLF: Raising ML to the power of System-F. In *Proceedings of the International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 27–38. ACM Press, aug 2003.

[14] D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University Department of Computer Science, 2005.

[15] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*, volume 11 of *Int. Series of Monographs in Computer Science*. Clarendon Press, Oxford, 1994.

[16] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[17] C. McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

[18] J. McKinna. Why dependent types matter. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–1, New York, NY, USA, 2006. ACM Press.

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

[20] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM Press, 1996.

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

[22] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 41–53, New York, NY, USA, 2001. ACM Press.

[23] E. Pasalic, J. Siek, and W. Taha. Concoqtion: Mixing dependent types and Hindley-Milner type inference (extended version). Technical report, Rice University, 2006.

[24] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *the International Conference on Functional Programming (ICFP '02)*, Pittsburgh, USA, October 2002. ACM.

[25] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. Accepted for publication to the Journal of Functional Programming, 2005.

[26] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, April 2006. submitted to ICFP 2006.

[27] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM Press.

[28] B. C. Pierce and D. N. Turner. Local type inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 252–265, New York, NY, USA, 1998. ACM Press.

[29] D. Rémy. Simple, partial type-inference for system f based on type-containment. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 130–143, New York, NY, USA, 2005. ACM Press.

[30] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.

[31] T. Sheard. Languages of the future. *SPNOTICES: ACM SIGPLAN Notices*, 39, 2004.

[32] P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types. Technical report, National University of Singapore, 2005.

[33] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[34] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, Apr. 2004. `http://coq.inria.fr`.

[35] P. Urzyczyn. Type reconstruction in f[omega]. *Mathematical Structures in Computer Science*, 7:329–358, 1997.

[36] D. Vytiniotis, S. Weirich, and S. P. Jones. Boxy types: type inference for higher-rank types and impredicativity. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006.

[37] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.

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

[39] C. Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.