Hindley-Milner

Abstract

Elaboration in Applicative Style

Functional pearl

François Pottier

INRIA
Francois.Pottier@inria.fr

- If t is a function $\lambda x.u$, then, for some types τ_1 and τ_2,

Type inference—the problem of determining whether a program is well-typed—is well-understood. In contrast, elaboration—the task of constructing an explicitly-typed representation of the program—seems to have received relatively little attention, even though, in a non-local type inference system, it is non-trivial. We show that the constraint-based presentation of Hindley-Milner type inference can be extended to deal with elaboration, while preserving its elegance. This involves introducing a new notion of "constraint with a value", which forms an applicative functor.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

Keywords Type inference; elaboration; polymorphism; constraints

1. Prologue

It was a bright morning. The Advisor was idle, when his newest student suddenly entered the room. "I need your help," she began. "I am supposed to test my type-preserving compiler for ML21," the Student continued, "but I can't conduct any experiments because I don't know how to connect the compiler with the front-end."

"Hmm," the Advisor thought. This experimental compiler was supposed to translate an *explicitly-typed* presentation of ML21 all the way down to typed assembly language. The stumbling block was that the parser produced abstract syntax for an *implicitly-typed* presentation of ML21, and neither student nor advisor had so far given much thought to the issue of converting one presentation to the other. After all, it was just good old Hindley-Milner type inference [15], wasn't it?

"So," the Student pressed. "Suppose the term t carries no type annotations. How do I determine whether t admits the type τ? And if it does, which type-annotated term t' should I produce?"

The Advisor sighed. Such effrontery! At least, the problem had been stated in a clear manner. He expounded: "Let us consider just simply-typed λ-calculus, to begin with. The answers to your questions are very simple." On the whiteboard, he wrote:

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628145

- the types τ and $\tau_1 \rightarrow \tau_2$ should be equal,
- assuming that the type of x is τ_1, u should have type τ_2,

and t' should be the type-annotated abstraction $\lambda x : \tau_1.u'$.

- If t is an application $t_1\ t_2$, then, for some type τ_2,
  - t_1 should have type $\tau_2 \rightarrow \tau$,
  - t_2 should have type τ_2,

  and t' should be $t'_1\ t'_2$.
- If t is a variable x, and if the type of x is θ, then
  - the types τ and θ should be equal,

  and t' should be x.

"There is your algorithm," the Advisor declared, setting the pen down and motioning towards the door. "It *can't* be any more complicated than this."

"This is a declarative specification," the Student thought. "It is not quite obvious whether an executable algorithm could be written in this style." It then occurred to her that the Advisor had not addressed the most challenging part of the question. "Wait," she said. "What about polymorphism?"

The Advisor pondered. He was not quite sure, offhand, how to extend this description with Hindley-Milner polymorphism.

"Let's see," he thought. So far, he had been implicitly thinking in terms of constraints $C ::= \texttt{true} \mid C \wedge C \mid \tau = \tau \mid \exists\alpha.C$ [25]. When he wrote "the types τ and θ should be equal", he had in mind an equality constraint $\tau = \theta$. When he wrote "for some type τ_2," he had in mind an existentially quantified constraint $\exists\alpha_2.\ldots$ (and he rather conveniently ignored the distinction between the type variable α_2 and the type τ_2 that he was really after). When he wrote "t has type τ", he had in mind a constraint, which, once t and τ are given, can be systematically constructed: on the whiteboard was a recursive description of this constraint generation process.

Now, one way of understanding Hindley-Milner polymorphism is to construct the predicate $\lambda\alpha.(t \text{ has type } \alpha)$. This is a constraint, parameterized over one type variable; in other words, a *constraint abstraction* [7]. A key theorem is that every satisfiable constraint abstraction $\lambda\alpha.C$ can be transformed to an equivalent canonical form, $\lambda\alpha.\exists\vec{\beta}.(\alpha = \theta)$, for suitably chosen type variables $\vec{\beta}$ and type θ. In traditional parlance, this canonical form is usually known as a *type scheme* [8] and written $\forall\vec{\beta}.\theta$. A type that satisfies the predicate $\lambda\alpha.\exists\vec{\beta}.(\alpha = \theta)$ is usually referred to as an *instance* of the type scheme $\forall\vec{\beta}.\theta$. The existence of such canonical forms for constraint abstractions is the *principal type scheme* property [8, 2].

"Jolly good," the Advisor resumed. "Let me amend the case of variables as follows."

- If t is x, and if the type scheme of x is $\forall\vec{\beta}.\theta$, then
  - for some vector $\vec{\tau}$, the types τ and $[\vec{\tau}/\vec{\beta}]\theta$ should be equal,

  and t' should be the type application $x\ \vec{\tau}$.

"And let me add a new case for let bindings." Somewhat more hesitantly, he wrote:

- If t is $\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$, then:
  - the constraint abstraction "$\lambda\alpha.(t_1$ has type $\alpha)$" should have some canonical form $\forall\vec{\beta}.\theta$,
  - assuming that the type scheme of x is $\forall\vec{\beta}.\theta$, the term t_2 should have type τ,

  and t' should be $\mathsf{let}\ x = \Lambda\vec{\beta}.t'_1\ \mathsf{in}\ t'_2$.

"There you are now." The Advisor seemed relieved. Apparently he had been able to write something plausible.

"This looks reasonably pretty, but is really still quite fuzzy," the Student thought. "For one thing, which type variables are supposed, or not supposed, to occur in the term t'? Is it clear that Λ-abstracting the type variables $\vec{\beta}$ in t'_1 is the right thing to do?" Indeed, the Advisor's specification would turn out to be incorrect or misleading (§B). "And," the Student thought, "it is now even less obvious how this description could be turned into executable code without compromising its elegance."

As if divining her thought, the Advisor added: "ML21 is a large language, whose design is not fixed. It is quite important that the elaboration code be as simple as possible, so as to evolve easily.

Split it into a constraint generator, along the lines of the whiteboard specification, and a constraint solver. The generator will be specific of ML21, but will be easy to adapt when the language evolves. The solver will be independent of ML21."

The Student shrugged imperceptibly. Such amazing confidence! Her advisor was a constraint buff. He probably thought constraints could save the world!

2. Constraints: a recap

The Student was well schooled, and knew most of what had been spelled out on the whiteboard. Why didn't her advisor's answer fully address her concerns?

Type inference in the simply-typed case can be reduced to solving a conjunction of type equations [25], or in other words, to solving constraints of the form $C ::= \mathsf{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C$. In its simplest formulation, the problem is to determine whether the equations (or the constraint) are satisfiable. In a more demanding formulation, the problem is to compute a most general unifier of the equations, or in other words, to bring the constraint into an equivalent solved form. These problems are collectively known as *first-order unification*. They are solved in quasi-linear time by Huet's first-order unification algorithm [9], which relies on Tarjan's efficient union-find data structure [23].

Type inference with Hindley-Milner polymorphism can also be considered a constraint solving problem, for a suitably extended constraint language [7, 19]:

$$\begin{array}{rl} \tau & ::= \alpha \mid \tau \to \tau \mid \ldots \\ C & ::= \mathsf{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha.C \end{array}$$

$$
\begin{array}{l}
\mid \mathsf{let}\ x = \lambda\alpha.C\ \mathsf{in}\ C \\
\mid x\ \tau
\end{array}
$$

The extension is quite simple. The let construct binds the variable x to the constraint abstraction $\lambda\alpha.C$. The instantiation construct $x\ \tau$ applies the constraint abstraction denoted by x to the type τ. One way of defining or explaining the meaning of these constructs is to expand them away via the following substitution law:

$$\mathsf{let}\ x = \lambda\alpha.C_1\ \mathsf{in}\ C_2 \quad \equiv \quad \exists\alpha.C_1 \wedge [\lambda\alpha.C_1/x]C_2$$

That is, the let constraint on the left-hand side is equivalent to (a) requiring that there exist at least one value of α for which C_1 holds; and (b) replacing[1] every reference to x in C_2 with a copy of the constraint abstraction $\lambda\alpha.C_1$.

According to the accepted wisdom, as repeated by the Advisor, one should write a constraint generator, which maps an unannotated term t to a constraint C, and a constraint solver, mapping a constraint C to a "satisfiable" or "unsatisfiable" answer. By composing the generator and the solver, one can determine whether t is well-typed.

In greater detail, the constraint generator takes the form of a recursive function that maps a term t and a type τ to a constraint $[\![t : \tau]\!]$, which informally means "t has type τ". It can be defined as follows [19]:

$$
\begin{aligned}
[\![x : \tau]\!] &= x\ \tau \\
[\![\lambda x.u : \tau]\!] &= \exists\alpha_1\alpha_2.\left(\begin{array}{l} \tau = \alpha_1 \to \alpha_2 \wedge \\ \mathsf{def}\ x = \alpha_1\ \mathsf{in}\ [\![u : \alpha_2]\!] \end{array}\right) \\
[\![t_1\ t_2 : \tau]\!] &= \exists\alpha.([\![t_1 : \alpha \to \tau]\!] \wedge [\![t_2 : \alpha]\!]) \\
[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \tau]\!] &= \mathsf{let}\ x = \lambda\alpha.[\![t_1 : \alpha]\!]\ \mathsf{in}\ [\![t_2 : \tau]\!]
\end{aligned}
$$

There, $\mathsf{def}\ x = \tau\ \mathsf{in}\ c$ is a short-hand for $\mathsf{let}\ x = \lambda\alpha.(\alpha = \tau)\ \mathsf{in}\ c$. A variable x that occurs free in the term t also occurs free in the constraint $[\![t : \tau]\!]$, where it now stands for a constraint abstraction. It is convenient to keep the name x, since the term t and the constraint $[\![t : \tau]\!]$ have the same binding structure.

This resembles the Advisor's whiteboard specification, but solves only the *type inference* problem, that is, the problem of determining whether a program is well-typed. It does not solve the *elaboration* problem, that is, the problem of constructing an explicitly-typed representation of the program. If the solver returns only a Boolean answer, how does one construct a type-annotated term t'? How does one obtain the necessary type information? A "satisfiable" or "unsatisfiable" answer is not nearly enough.

One may object that a solver should not just produce a Boolean answer, but also transform a constraint C into an equivalent solved form. However, if the term t is closed, then the constraint $[\![t : \alpha]\!]$ has just one free type variable, namely α. This implies that a solved form of $[\![t : \alpha]\!]$ cannot constrain any type variables other than α. Such a solved form could be, for instance, $\alpha = \mathit{unit}$, which tells us that t has type *unit*, but does not tell us how to construct t', which presumably must contain many type annotations.

A more promising idea, or a better formulation of this idea, would be to let the solver produce a satisfiability witness W, whose shape is dictated by the shape of C. (This could be implemented simply by annotating the constraint with extra information.) One would then write an elaboration function, mapping t and W to an explicitly-typed term t'.

Certainly, this approach is workable: the solution advocated in this paper can be viewed as a nicely-packaged version of it. If implemented plainly in the manner suggested above, however, it seems unsatisfactory. For one thing, the elaboration function expects two arguments, namely a term t and a witness W, and must deconstruct them in a "synchronous" manner, keeping careful track of the correlation between them. This is unpleasant[2]. Furthermore, in this approach, the type inference and elaboration process is split

[1] Technically, one defines $[\lambda\alpha.C/x](x\ \tau)$ as $[\tau/\alpha]C$; that is, the β-redex $(\lambda\alpha.C)\ \tau$ is reduced on the fly as part of the substitution.

[2] Rémy and Yakobowski's elaboration of eMLF into xMLF [21] merges the syntax of terms, constraints, and witnesses. Similarly, Gundry suggests "identifying the syntactic and linguistic contexts" [6, §2.4]. If one follows them, then the constraint C carries more information than the term t, and the witness W in turn carries more information than C. This means that the elaboration phase does not have to be a function of two arguments: it maps

in three phases, namely constraint generation, constraint solving, and elaboration. Only the second phase is independent of the programming language at hand. The first and last phases are not. For our Student, this means that, at every evolution of ML21, two places in the code have to be consistently updated. This is not as elegant as we (or the Student's exacting advisor) would like.

In summary, the traditional presentation of type inference as a constraint solving problem seems to fall a little short of offering an elegant solution to the elaboration problem.

3. Constraints with a value

The fundamental reason why there must be three separate phases (namely generation, solving, elaboration) is that constraint solving is a non-local process. In a constraint of the form $(\exists\alpha.C_1) \wedge C_2$, for instance, the final value of α cannot be determined by inspecting just C_1: the solver must inspect also C_2. In other words, when looking at a constraint of the form $\exists\alpha.C$, the final value of α cannot be determined by examining C alone: this value can be influenced by the surrounding context. Thus, one must wait until constraint solving is finished before one can query the solver about the value of α. One cannot query it and obtain an answer right away.

Yet, the pseudo-code on the whiteboard (§1) seems to be written *as if* this was possible. It wishes for some type τ to exist, subject to certain constraints, then goes on and uses τ in the construction of the term t'. In other words, even though phases 1 and 3 (that is, generation and elaboration) must be separately *executed,* we wish to *express* them together. This is the key reason why this pseudo-code seems concise, compositional, and maintainable (i.e., when ML21 evolves, only one piece of code must be updated).

Fortunately, one *can* give precise, executable meaning to the Advisor's style of expression. This is what the Student discovered and worked out, confirming that his Advisor was on the right track, even though he most likely did not have a very clear idea of the difficulties involved.

Described in high-level, declarative terms, what is desired is a language of "constraints with a value", that is, constraints that not only impose certain requirements on their free type variables, but also (provided these requirements are met) produce a result. Here, this result is an explicitly-typed term. In general, though, it could be anything. The language of constraints-with-a-value can (and should) be independent of the nature of the values that are computed. For any type α of the meta-language[3], we would like to be able to construct "α-constraints", that is, constraints which (once satisfied) produce a result of type α.

Described in lower-level, operational terms, one wishes to bring together the code of phase 1, which builds a constraint, and the code of phase 3, which (by exploiting the information provided by the solver) produces a result. So, one could think of an "α-constraint" as a pair of (a) a raw constraint (which can be submitted to the solver) and (b) a function which (after the solver has finished) computes a value of type α. Our OCaml implementation (§4) is based on this representation.

We propose the following syntax of constraints-with-a-value:

$$\begin{array}{rl} C ::= & \\ & \mid \mathsf{true} \mid C \wedge C \mid \tau = \tau \mid \exists \alpha . C \\ & \mid \mathsf{let}\ x = \lambda \alpha . C\ \mathsf{in}\ C \\ & \mid x\ \tau \\ & \mid \mathit{map}\ f\ C \end{array}$$

This syntax is identical to that of raw constraints (§2), with one addition. A new construct appears: $map\ f\ C$, where f is a meta-language function. The intention is that this constraint is satisfied when C is satisfied, and if the constraint C produces some value V, then $map\ f\ C$ produces the value $f\ V$.

The other constructs retain their previous logical meaning, and in addition, acquire a new meaning as producers of meta-language values. At this point, let us give only an informal description of the value that each construct produces. Things are made more precise when we present the high-level interface of the OCaml library (§4.3). Furthermore, to the mathematically inclined reader, an appendix (§A) offers a formal definition of the meaning of constraints-with-a-value, that is, when they are satisfied, and what value they produce. This allows us to specify what the OCaml code is supposed to compute.

As usual, a conjunction $C_1 \wedge C_2$ is satisfied if and only if C_1 and C_2 are satisfied. In addition, if C_1 and C_2 respectively produce the values V_1 and V_2, then the conjunction $C_1 \wedge C_2$ produces the pair (V_1, V_2).

The constraints `true` and $\tau_1 = \tau_2$ produce a unit value.

Existential quantification is more interesting. If C produces the value V, then $\exists\alpha.C$ produces the pair (T, V), where T is the witness, that is, the value that must be assigned to the type variable α in order to satisfy the constraint C. (We write T for a "decoded type". This notion is clarified in §4, from an OCaml programmer's point of view, and in §A, from a more formal point of view.) The type T may have free "decoded type variables",

just W to t'. A disadvantage of this approach, though, is that the syntax of constraints is no longer independent of the programming language at hand.

[3] In our implementation (§4, §5), the meta-language is OCaml.

which we write a. The reader may wonder where and how these variables are supposed to be introduced. This is answered below in the discussion of let constraints.

An instantiation constraint $x\ \tau$ produces a vector $\vec{T}$ of decoded types. These are again witnesses: they indicate how to the type scheme associated with x must be instantiated in order to obtain the type τ.

A constraint of the form let $x = \lambda\alpha.C_1$ in C_2 produces a tuple of three values:

1. The canonical form of the constraint abstraction $\lambda\alpha.C_1$. In other words, this is the type scheme that was inferred for x, and that was associated with x while solving C_2. It is a "decoded type scheme", of the form $\forall\vec{b}.T$.
2. A value of the form $\Lambda\vec{a}.V_1$, if V_1 is the value produced by C_1.
3. The value V_2 produced by C_2.

In order to understand the binder "$\Lambda\vec{a}$" in the second item, one must note that, in general, the value V_1 may have free decoded type variables. For instance, if C_1 begins with an existential quantifier $\exists\alpha.\ldots$, then V_1 is a pair $(T, \ldots)$, where the decoded type T may have free decoded type variables. By introducing the binder "$\Lambda\vec{a}$", the solver is telling the user that, at this particular place, the type variables $\vec{a}$ should be introduced. (In the OCaml code, the solver separately returns $\vec{a}$ and V_1, and the user is responsible for building an appropriate abstraction.)

The reader may wonder whether there should be a connection between the vectors $\vec{a}$ and $\vec{b}$. The short answer is, in general, $\vec{b}$ is a subset of $\vec{a}$. This is discussed in detail in the appendices (§A, §B).

4. Solving constraints with a value

We have implemented our proposal as an OCaml library, whose code is available online [17]. It is organized in two layers. The low-level layer (§4.2) solves a raw constraint, exports information via write-once references, and offers facilities to decode this information. The high-level layer (§4.3) hides many of these low-level details. It allows the client to construct constraints-with-a-value and offers a single function *solve*; nothing else is needed.

```
module type TEVAR = sig
  type tevar
  val compare: tevar → tevar → int
end
```

Figure 1. Term variables

```
module type STRUCTURE = sig
  type α structure
  val map: (α → β) → α structure → β structure
  val iter: (α → unit) → α structure → unit
  val fold: (α → β → β) → α structure → β → β
  exception Iter_2
  val iter_2: (α → β → unit) → α structure → β structure → unit
end
```

Figure 2. Shallow structure of types

```
module type OUTPUT = sig
  type tyvar = int
  type α structure
  type ty
  val variable: tyvar → ty
  val structure: ty structure → ty
  val mu: tyvar → ty → ty
  type scheme = tyvar list × ty
end
```

Figure 3. Decoded representation of types

4.1 Parameters

The low-level and high-level solvers are functors, parameterized over three arguments.

The first argument (Figure 1) provides the type *tevar* of term variables. This type must be equipped with a total ordering.

The second argument (Figure 2) provides a type α *structure*, which defines the first-order universe over which type variables are interpreted. A value of type α *structure* is a shallow type: it represents an application of a constructor (say, arrow, or product) to a suitable number of arguments of type α. It must be equipped with a *map* function (as well as *iter* and *fold*, which in principle can be derived from *map*) and with $iter_2$, which is expected to fail if its arguments exhibit distinct constructors.

The last argument (Figure 3) provides the types *tyvar* and *ty* of decoded type variables and decoded types. For simplicity, the definition of *tyvar* is fixed: it is just *int*. That is, a decoded type variable is represented as an integer name. The type *ty* is the client's representation of types. It must be able to express type variables (the function *variable* is an injection of *tyvar* into *ty*) as well as types built by applying a constructor to other types (the function *structure* is an injection of *ty structure* into *ty*).

The type *ty* must also come with a function *mu*, which allows constructing recursive types. If *a* is a type variable and *t* represents an arbitrary type, then *mu a t* should represent the recursive type

$\mu a.t$. This feature is required for two reasons: (a) the solver optionally supports recursive types, in the style of `ocaml -rectypes`; and (b) even if this option is disabled, the types carried by the solver exceptions *Unify* and *Cycle* (Figure 5) can be cyclic.

The last line of Figure 3 specifies that a decoded type scheme is represented as a pair of a list of type variables (the universal quantifiers) and a type (the body).

```
module Make
  (X : TEVAR)
  (S : STRUCTURE)
  (O : OUTPUT with type α structure = α S.structure)
: sig
  open X
  open S
  open O
  type variable
  val fresh: variable structure option → variable

  type ischeme
  type rawco =
  | CTrue
  | CConj of rawco × rawco
  | CEq of variable × variable
  | CExist of variable × rawco
  | CInstance of tevar × variable × variable list WriteOnceRef.t
  | CDef of tevar × variable × rawco
  | CLet of (tevar × variable × ischeme WriteOnceRef.t) list
            × rawco
            × rawco
            × variable list WriteOnceRef.t
  exception Unbound of tevar
  exception Unify of variable × variable
  exception Cycle of variable
  val solve: bool → rawco → unit

  val decode_variable: variable → tyvar
  type decoder = variable → ty
  val new_decoder: bool → decoder
  val decode_scheme: decoder → ischeme → scheme
end
```

Figure 4. The solver's low-level interface

4.2 Low-level interface

As the low-level layer is not a contribution of this paper, we describe it rather briefly. The reader who would like to know more may consult its code online [17]. Its interface appears in Figure 4.

The types *variable* and *ischeme* are abstract. They are the solver's internal representations of type variables and type schemes. Here, a type variable can be thought of as a vertex in the graph maintained by the first-order unification algorithm. The function *fresh* allows the client to create new vertices. It can be applied to *None* or to *Some t*, where *t* is a shallow type. In the former case, the new vertex can be thought of as a fresh unification variable; in the latter case, it can be thought of as standing for the type *t*.

The type *rawco* is the type of raw constraints. Their syntax is as previously described (§2), except that *CLet* allows binding several term variables at once, a feature that we do not describe in this paper. A couple of low-level aspects will be later hidden in the high-level interface, so we do not describe them in detail:

- In *CExist* (v, c), the type variable v must be fresh and unique. A similar requirement bears on the type variables carried by *CLet*.
- *CInstance* and *CLet* carry write-once references (i.e., references to an option), which must be fresh (uninitialized) and unique. The solver sets these references[4].

[4] Instead of setting write-once references, the solver could build a witness, a copy of the constraint that carries more information. That would be somewhat more verbose and less efficient, though. Since these details are ultimately hidden, we prefer to rely on side effects.

The function *solve* expects a closed constraint and determines whether it is satisfiable. The Boolean parameter indicates whether recursive types, in the style of `ocaml -rectypes`, are legal.

If the constraint is unsatisfiable, an exception is raised. The exception *Unify* (v_1, v_2) means that the type variables v_1 and v_2 cannot be unified; the exception *Cycle* v means that a cycle in the type structure has been detected, which the type variable v participates in.

If the constraint is satisfiable, the solver produces no result, but annotates the constraint by setting the write-once references embedded in it.

The type information that is made available to the client, either via the exceptions *Unify* and *Cycle* or via the write-once references, consists of values of type *variable* and *ischeme*. These are abstract types: we do not wish to expose the internal data structures used by the solver. Thus, the solver must also offer facilities for decoding this information, that is, for converting it to values of type *tyvar*, *ty*, etc. These decoding functions are supposed to be used only after the constraint solving phase is finished.

The function *decode_variable* decodes a type variable. As noted earlier, the type *tyvar* is just *int*: a type variable is decoded to its unique integer identifier.

The function *new_decoder* constructs a new type decoder, that is, a function of type *variable* → *ty*. (The Boolean parameter tells whether the decoder should be prepared to support cyclic types.) This decoder has persistent state. Indeed, decoding consists in traversing the graph constructed by the unification algorithm and turning it into what appears to be a tree (a value of type *ty*) but is really a DAG. The decoder internally keeps track of the visited vertices and their decoded form (i.e., it maintains a mapping of *variable* to *ty*), so that the overall cost of decoding remains linear in the size of the graph[5].

We lack space to describe the implementation of the low-level solver, and it is, anyway, beside the point of the paper. Let us just emphasize that it is is modular: (a) at the lowest layer lies Tarjan's efficient union-find algorithm [23]; (b) above it, one finds Huet's first-order unification algorithm [9]; (c) then comes the treatment of generalization and instantiation, which exploits Rémy's integer ranks [20, 12, 11] to efficiently determine which type variables must be generalized; (d) the last layer interprets the syntax of constraints. The solver meets McAllester asymptotic time bound [12]: under the assumption that all of the type schemes that are ever constructed have bounded size, its time complexity is $O(nk)$, where n is the size of the constraint and k is the left-nesting depth of *CLet* nodes.

4.3 High-level interface

The solver's high-level interface appears in Figure 5. It abstracts away several details of raw constraints, including the transmission of information from solver to client via write-once references and the need to decode types. In short, it provides: (a) an abstract type α *co* of constraints that produce a value of type α; (b) a number of ways of constructing such constraints; and (c) a single function, *solve*, that solves and evaluates such a constraint and (if successful) produces a final result of type α.

The type α *co* is internally defined as follows:

type α *co* =
  rawco $\times$ (*env* $\rightarrow \alpha$)

That is, a constraint-with-a-value is a pair of a raw constraint *rc* and a continuation *k*, which is intended to be invoked after the

```
module Make
  (X : TEVAR)
  (S : STRUCTURE)
  (O : OUTPUT with type α structure = α S.structure)
: sig
  open X
  open S
  open O
  type variable

  type α co
  val pure: α → α co
  val (^&): α co → β co → (α × β) co
  val map: (α → β) → α co → β co
  val (--): variable → variable → unit co
  val (---): variable → variable structure → unit co
  val exist: (variable → α co) → (ty × α) co
  val instance: tevar → variable → ty list co
  val def: tevar → variable → α co → α co
  val let1: tevar → (variable → α co) → β co →
                 (scheme × tyvar list × α × β) co

  exception Unbound of tevar
```

[5] This is true when support for cyclic types is disabled. When it is enabled, one must place μ binders in a correct manner, and this seems to prevent the use of persistent state. We conjecture that the unary μ is too impoverished a construct: it does not allow describing arbitrary cyclic graphs without a potential explosion in size.

```
  exception Unify of ty × ty
  exception Cycle of ty
  val solve: bool → α co → α
end
```

Figure 5. The solver's high-level interface

constraint solving phase is over, and is expected to produce a result of type α. The continuation receives an environment which, in the current implementation, contains just a type decoder:

```
type env =
  decoder
```

If one wished to implement α *co* in a purely functional style, one would certainly come up with a different definition of α *co*. Perhaps something along the lines of *rawco* × (*witness* → α *m*), where *witness* is the type of the satisfiability witness produced by the low-level solver (no more write-once references!) and α *m* is a suitable monad, so as to allow threading the state of the type decoder through the elaboration phase. Perhaps one might also wish to use a dependent type, or a GADT, to encode the fact that the shape of the witness is dictated by the shape of the raw constraint. We use OCaml's imperative features because we can, but the point is, the end user does not need to know; the abstraction that we offer is independent of these details, and is not inherently imperative.

The combinators (*pure*, ..., *let*$_1$) allow building constraints-with-a-value. Most of them produce a little bit of the underlying raw constraint, together with an appropriate continuation. The only exception is *map*, which installs a continuation but does not affect the underlying raw constraint.

The constraint *pure a* is always satisfied and produces the value *a*. It is defined as follows:

```
let pure a =
  CTrue,
  fun env → a
```

If c_1 and c_2 are constraints of types α *co* and β *co*, then c_1 ^& c_2 is a constraint of type $(\alpha \times \beta)$ *co*. It represents the conjunction of the underlying raw constraints, and produces a pair of the results produced by c_1 and c_2.

```
let (^&) (rc1, k1) (rc2, k2) =
  CConj (rc1, rc2),
  fun env → (k1 env, k2 env)
```

If c is a constraint of type α *co* and if the user-supplied function f maps α to β, then *map f c* is a constraint of type β *co*. Its logical meaning is the same as that of c.

```
let map f (rc, k) =
  rc,
  fun env → f (k env)
```

Equipped with the combinators *pure*, ^&, and *map*, the type constructor *co* is an applicative functor. More specifically, it is an instance of McBride and Paterson's type class *Monoidal* [13, §7]. Furthermore, the combinator ^& is commutative, that is, it enjoys the following law:

$$c_1 \mathbin{\hat{}\&} c_2 \equiv \mathit{map}\ \mathit{swap}\ (c_2 \mathbin{\hat{}\&} c_1)$$

where *swap* (a_2, a_1) is (a_1, a_2). This law holds because *CConj* is commutative; the order in which the members of a conjunction are considered by the solver does not influence the final result.

It is worth noting that *co* is not a monad, as there is no sensible way of defining a *bind* operation of type $\alpha\ co \to (\alpha \to \beta\ co) \to \beta\ co$. In an attempt to define *bind* $(rc_1, k_1)\ f_2$, one would like to construct a raw conjunction *CConj* (rc_1, rc_2). In order to obtain rc_2, one must invoke f_2, and in order to do that, one needs a value of type α, which must be produced by k_1. But the continuation k_1 must not be invoked until the raw constraint rc_1 has been solved. In summary, a constraint-with-a-value is a pair of a static component (the raw constraint) and a dynamic component (the continuation), and this precludes a definition of *bind*. This phenomenon, which was observed in Swierstra and Duponcheel's LL(1) parser combinators [22], was one of the motivations that led to the recognition of arrows [10] and applicative functors [13] as useful abstractions.

Although we do not have *bind*, we have *map*. When one builds a constraint *map f c*, one is assured that the function *f* will be run after the constraint solving phase is finished. In particular, *f* is run after *c* has been solved. We emphasize this by defining a version of *map* with reversed argument order:

```
let (<$$>) a f =
  map f a
```

The combinators -- and --- construct equations, i.e., unification constraints. The constraint v_1 -- v_2 imposes an equality between the variables v_1 and v_2, and produces a unit value. Its definition is straightforward:

```
let (--) v1 v2 =
  CEq (v1, v2),
  fun env → ()
```

The constraint v_1 --- t_2 is also an equation, whose second member is a shallow type.

The next combinator, *exist*, builds an existentially quantified constraint $\exists\alpha.C$. Its argument is a user-defined function f which, once supplied with a fresh type variable α, must construct C. It is defined as follows:

```
let exist f =
  let v = fresh None in
  let rc, k = f v in
  CExist (v, rc),
  fun env →
    let decode = env in
    (decode v, k env)
```

At constraint construction time, we create a fresh variable *v* and pass it to the client by invoking f *v*. This produces a constraint, i.e., a pair of a raw constraint *rc* and a continuation *k*. We can then construct the raw constraint *CExist* (*v*, *rc*). We define a new continuation, which constructs a pair of the decoded value of *v* and the value produced by *k*. As a result, the constraint *exist f* has type (*ty* × α) *co*: it produces a pair whose first component is a decoded type. The process of decoding types has been made transparent to the client.

The combinator *instance* constructs an instantiation constraint *x v*, where *x* is a term variable and *v* is a type variable. The type of *instance x v* is *ty list co*: this constraint produces a vector of decoded types, so as to indicate how the type scheme associated with *x* was instantiated. This combinator is implemented as follows:

```
let instance x v =
  let witnesses = WriteOnceRef.create() in
```

```
CInstance (x, v, witnesses),
fun env →
  let decode = env in
  List.map decode (WriteOnceRef.get witnesses)
```

At constraint construction time, we create an empty write-once reference, *witnesses*, and construct the raw constraint *CInstance* (*x*, *v*, *witnesses*), which carries a pointer to this write-once reference. During the constraint solving phase, this reference is written by the solver, so that, when the continuation is invoked, we may read the reference and decode the list of types that it contains. Thus, the transmission of information from the solver to the client via write-once references has been made transparent.

The last combinator, let_1, builds a let constraint. It should be applied to three arguments, namely: (a) a term variable *x*; (b) a user-supplied function f_1, which denotes a constraint abstraction $\lambda\alpha.c_1$ (i.e., when applied to a fresh type variable α, this function constructs the constraint c_1); (c) a constraint c_2. We omit its code, which is in the same style as that of *instance* above. As promised earlier (§3), this constraint produces the following results:

- A decoded type scheme, $\forall\vec{b}.T$, of type *scheme*. It can be viewed as the canonical form of the constraint abstraction $\lambda\alpha.c_1$. This type scheme has been associated with *x* while solving c_2. We guarantee that $\vec{b}$ is a subset of $\vec{a}$ (see §A and §B for details).
- A vector of decoded type variables $\vec{a}$, of type *tyvar list*, and a value V_1, of type α, produced by c_1. The type variables $\vec{a}$ may occur in V_1. The user is responsible for somehow binding them in V_1, so as to obtain the value referred to as "$\Lambda\vec{a}.V_1$" in §3.
- A value V_2, produced by c_2, of type β.

The function *solve* takes a constraint of type α *co* to a result of type α. If the constraint is unsatisfiable, then the exception that is raised (*Unify* or *Cycle*) carries a decoded type, so that (once again) the decoding process is transparent. Thus, in the implementation, we redefine *Unify* and *Cycle*:

```
exception Unify of O.ty × O.ty
exception Cycle of O.ty
```

and implement *solve* as follows:

```
let solve rectypes (rc, k) =
  begin try
    Lo.solve rectypes rc
  with
  | Lo.Unify (v1, v2) →
        let decode = new_decoder true in
        raise (Unify (decode v1, decode v2))
  | Lo.Cycle v →
        let decode = new_decoder true in
        raise (Cycle (decode v))
  end;
  let decode = new_decoder rectypes in
  let env = decode in
  k env
```

The computation is in two phases. First, the low-level solver, *Lo.solve*, is applied to the raw constraint *rc*. Then, elaboration

```
type tevar = string
type term =
  | Var of tevar
  | Abs of tevar × term
  | App of term × term
  | Let of tevar × term × term
```

Figure 6. Syntax of the untyped calculus (ML)

```
type (α, β) typ =
  | TyVar of α
  | TyArrow of (α, β) typ × (α, β) typ
  | TyProduct of (α, β) typ × (α, β) typ
  | TyForall of β × (α, β) typ
  | TyMu of β × (α, β) typ
type tyvar = int
type nominal_type = (tyvar, tyvar) typ
type tevar = string
type (α, β) term =
  | Var of tevar
  | Abs of tevar × (α, β) typ × (α, β) term
  | App of (α, β) term × (α, β) term
  | Let of tevar × (α, β) term × (α, β) term
  | TyAbs of β × (α, β) term
  | TyApp of (α, β) term × (α, β) typ
type nominal_term = (tyvar, tyvar) term

let ftyabs vs t =
  List.fold_right (fun v t → TyAbs (v, t)) vs t
```

```
let ftyapp t tys =
  List.fold_left (fun t ty → TyApp (t, ty)) t tys
```

Figure 7. Syntax of the typed calculus (F)

takes place: the continuation *k* is invoked. It is passed a fresh type decoder, which has persistent state[6], so that (as announced earlier) the overall cost of decoding is linear.

If the raw constraint *rc* is found to be unsatisfiable, the exception raised by the low-level solver (*Lo.Unify* or *Lo.Cycle*) is caught; its arguments are decoded, and an exception (*Unify* or *Cycle*) is raised again. The decoder that is used for this purpose must support recursive types, even if *rectypes* is *false*. Obviously, the argument carried by *Cycle* is a vertex that participates in a cycle! Perhaps more surprisingly, the arguments carried by *Unify* may participate in cycles too, as the occurs check is performed late (i.e., only at *CLet* constraints) and in a piece-wise manner (i.e., only on the so-called "young generation").

The high-level solver has asymptotic complexity $O(nk)$, like the low-level solver. Indeed, the cost of constructing and invoking continuations is $O(n)$, and the cost of decoding is linear in the total number of type variables ever created, that is, $O(nk)$.

5. Elaborating ML into System F

We now show how to perform elaboration for an untyped calculus ("ML", shown in Figure 6) and translate it to an explicitly-typed form ("F", shown in Figure 7). The code (Figure 8) is a formal and rather faithful rendition of the Advisor's pseudo-code (§1).

5.1 Representations of type variables and binders

In both calculi, the representation of term variables is nominal. (Here, they are just strings. One could use unique integers instead.) The representation of type variables in System F is not fixed: the

```
let rec hastype (t : ML.term) (w : variable) : F.nominal_term co
= match t with
  | ML.Var x ->
        instance x w <$$> fun tys ->
        F.ftyapp (F.Var x) tys
  | ML.Abs (x, u) ->
        exist (fun v1 ->
          exist (fun v2 ->
            w --- arrow v1 v2 ^&
            def x v1 (hastype u v2)
          )
        ) <$$> fun (ty1, (ty2, ((), u'))) ->
        F.Abs (x, ty1, u')
  | ML.App (t1, t2) ->
        exist (fun v ->
          lift hastype t1 (arrow v w) ^&
          hastype t2 v
        ) <$$> fun (ty, (t1', t2')) ->
        F.App (t1', t2')
  | ML.Let (x, t, u) ->
        let1 x (hastype t)
          (hastype u w)
        <$$> fun ((b, _), a, t', u') ->
        F.Let (x, F.ftyabs a t',
```

[6] Provided *rectypes* is *false* (§4.2).

```
    F.Let (x, coerce a b (F.Var x),
    u'))
```

Figure 8. Type inference and translation of ML to F

syntax is parametric in α (a type variable occurrence) and β (a type variable binding site). A nominal representation is obtained by instantiating α and β with *tyvar* (which is defined as *int*, so a type variable is represented by a unique integer identifier), whereas de Bruijn's representation (not shown) is obtained by instantiating α with *int* (a de Bruijn index) and β with *unit*.

In the following, we construct type-annotated terms under a nominal representation. This is natural, because the constraint solver internally represents type variables as mutable objects with unique identity, and presents them to us as unique integers. One may later perform a conversion to de Bruijn's representation, which is perhaps more traditional for use in a System F type-checker.

5.2 Translation

Thanks to the high-level solver interface, type inference for ML and elaboration of ML into F are performed in what appears to be one pass. The code is simple and compositional: it takes the form of a single recursive function, *hastype*. This function maps an ML term *t* and a unification variable *w* to a constraint of type *F.nominal_term co*. This constraint describes, at the same time, a raw constraint (what is a necessary and sufficient condition for the term *t* to have type *w*?) and a process by which (if the raw constraint is satisfied) a term of System F is constructed.

The function *hastype* appears in Figure 8. Most of it should be clear, since it corresponds to the whiteboard specification of §1. Let us explain just a few points.

The auxiliary function *lift* (not shown) transforms a function of type $\alpha \rightarrow$ *variable* $\rightarrow \beta$ *co* into one of type $\alpha \rightarrow$ *variable structure* $\rightarrow \beta$ *co*. It can be defined in terms of *map*, *exist*, and ---. Thus, whereas the second argument of *hastype* is a type variable, the second argument of *lift hastype* is a shallow type. This offers a convenient notation in the case of *ML.App*.

In the case of *ML.Let*, the partial application *hastype t* is quite literally a constraint abstraction! The construct *ML.Let* is translated to *F.Let*. (One could encode *F.Let* as a β-redex, at the cost of extra type annotations.) The type variables a are explicitly Λ-abstracted in the term t', as explained in the description of let_1 (§4.3).

In the next-to-last line of Figure 8, the variable x is re-bound to an application of a certain coercion to x, which is constructed by the function call *coerce a b* (*F.Var x*). This coercion has no effect when the vectors of type variables a and b are equal. The case where they differ is discussed in the appendix (§B).

6. Conclusion

What have we achieved? We have started with a language of "raw" constraints (§2) that can express the type inference problem in a concise and elegant manner. Its syntax is simple. Yet, it requires a non-trivial and non-local constraint solving procedure, involving first-order unification as well as generalization and instantiation. We have argued that it does not solve the elaboration problem. A "satisfiable" or "unsatisfiable" answer is not enough, and asking the solver to produce more information typically results in a low-level interface (§4.2) that does not directly allow us to express elaboration in an elegant manner. The key contribution of this paper is a high-level solver interface (§4.3) that allows (and forces) the user to tie together the constraint generation phase and the elaboration phase, resulting in concise and elegant code (§5). The high-level interface offers one key abstraction, namely the type α *co* of "constraints with a value". Its meaning can be specified in a declarative manner (§A). It is an applicative functor, which suggests that it is a natural way of structuring an elaboration algorithm that has the side effect of emitting and solving a constraint. The high-level interface can be modularly constructed above the low-level solver, without knowledge of how the latter is implemented, provided the low-level solver produces some form of satisfiability witness.

The idea of "constraints with a value" is not specific of the Hindley-Milner setting. It is in principle applicable and useful in other settings where constraint solving is non-local. For instance, elaboration in programming languages with dependent types and implicit arguments [6], which typically relies on higher-order pattern unification [14], could perhaps benefit from this approach.

The constraint language is small, but powerful. As one scales up to a real-world programming language in the style of ML, the constraint language should not need to grow much. The current library [17] already offers a combinator *letn* for defining a constraint abstraction with n entry points; this allows dealing with ML's "let $p = t_1$ in t_2", which simultaneously performs generalization and pattern matching. Two simple extensions would be universal quantification in constraints [18, §1.10], which allows dealing with "rigid", user-provided type annotations, and rows [19, §10.8], which allow dealing with structural object types in the style of OCaml. The value restriction requires no extension to the library, but the relaxed value restriction [3] would require one: the solver would have to be made aware of the variance of every type constructor. Higher-rank polymorphism [4, 16], polymorphism in the style of MLF [21], and GADTs [24, 5] would require other extensions, which we have not considered.

The current library has limited support for reporting type errors, in the form of the exceptions *Cycle* and *Unify*. The unification algorithm is transactional. Equations are submitted to it one by one, and each submission either succeeds and updates the algorithm's current state, or fails and has no effect. This means that the types carried by the exception *Unify* reflect the state of the solver just before the problematic equation was encountered. The library could easily be extended with support for embedding source code locations (of a user-specified type) in constraints. This should allow displaying type error messages of roughly the same quality as those of the OCaml type-checker. A more ambitious treatment of type errors might require a different constraint solver, which hopefully would offer the same interface as the present one, so that the elaboration code need not be duplicated.

Our claim that the elaboration of ML into System F has complexity $O(nk)$ (§4.3) must be taken with a grain of salt. In our current implementation, this is true because elaboration does not produce a System F *term*: it actually constructs a System F *DAG*, with sharing in the type annotations. Displaying this DAG in a naive manner, or converting it in a naive way to another representation, such as de Bruijn's representation, causes an increase in size, which in the worst case could be exponential. One could address this issue by extending System F with a local type abbreviation construct, of the form $\mathsf{let}\ a = T\ \mathsf{in}\ t$, where a is a type variable and T is a type. The elaboration algorithm would emit this construct at let nodes. (The low-level and high-level solver interfaces would have to be adapted. The solver would publish, at every let node, a set of local type definitions.) All type annotations (at λ-abstractions and at type applications) would then be reduced to type variables. This could be an interesting avenue for research, as this extension of System F might enjoy significantly faster type-checking.

References

[1] Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Principles of Programming Languages (POPL)*, pages 361–372, 2012.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[3] Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2004.

[4] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Information and Computation*, 155(1):134–169, 1999.

[5] Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with GADTs. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013.

[6] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[7] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*. Springer, 2001.

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

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

[10] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[11] George Kuan and David MacQueen. Efficient type inference using ranked type variables. In *ACM Workshop on ML*, pages 3–14, 2007.

[12] David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2003.

[13] Conor McBride and Ross Paterson. Applicative programming with

effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[14] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

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

[16] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1):1–82, 2007.

[17] François Pottier. Inferno: a library for Hindley-Milner type inference and elaboration, February 2014. http://gallium.inria.fr/~fpottier/inferno/inferno.tar.gz.

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

[19] 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.

[20] Didier Rémy. Extending ML type system with a sorted equational theory. Technical Report 1766, INRIA, 1992.

[21] Didier Rémy and Boris Yakobowski. A Church-style intermediate language for MLF. *Theoretical Computer Science*, 435(1):77–105, 2012.

[22] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.

[23] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[24] Dimitrios Vytiniotis, Simon L. Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4–5):333–412, 2011.

[25] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticæ*, 10:115–122, 1987.

A. Semantics of constraints with a value

In order to clarify the definition that follows, we must distinguish two namespaces of type variables. In the syntax of constraints (§2, §3), we have been using α to denote a type variable. Such a variable may appear in a type τ and in a constraint C. It represents a type to be determined; one could refer to it informally as a "unification variable". In contrast, the explicitly-typed terms that we wish to construct also contain type variables, but those do not stand for types to be determined; they are type constants, so to speak. For the sake of clarity, we use distinct meta-variables, namely a and b, to denote them, and we write T for a first-order type built on them:

$$T ::= a \mid T \rightarrow T \mid \ldots$$

We refer to a as a "decoded" type variable and to T as a "decoded" type. In the following, we write ϕ for a partial mapping of the type variables α to decoded types T. The application of ϕ to a type τ produces a decoded type T.

We wish to define when a constraint C may produce a value V, where V denotes a value of the meta-language. (In §5, the values that we build are OCaml representations of System F terms.) We do not wish to fix the syntax of values, as it is under the user's control. We assume that it includes: (a) tuples of arbitrary arity, (b) decoded type schemes, and (c) a way of binding a vector $\vec{a}$ in a value V:

$$V ::= (V, \ldots, V) \mid \forall \vec{a}.T \mid \Lambda \vec{a}.V \mid \ldots$$

In order to define when a constraint C may produce a value V, we need a judgement of at least two arguments, namely C and V. In order to indicate which term variables x and which decoded type variables a are in scope, we add a third argument, namely an environment E, whose structure is as follows:

$$E ::= \emptyset \mid E, x : \forall \vec{a}.T \mid E, a$$

(This is essentially an ML type environment.)

Finally, to keep track of the values assigned to unification variables, we add a fourth and last argument, namely a substitution ϕ. Thus, we define a judgement of the following form:

$$E; \phi \vdash C \rightsquigarrow V$$

$$E; \phi \vdash \mathsf{true} \rightsquigarrow () \qquad \frac{E; \phi \vdash C_1 \rightsquigarrow V_1 \qquad E; \phi \vdash C_2 \rightsquigarrow V_2}{E; \phi \vdash C_1 \wedge C_2 \rightsquigarrow (V_1, V_2)}$$

$$\frac{\phi(\tau_1) = \phi(\tau_2)}{E; \phi \vdash \tau_1 = \tau_2 \rightsquigarrow ()} \qquad \frac{\begin{array}{c} E \vdash T \text{ ok} \\ E; \phi[\alpha \mapsto T] \vdash C \rightsquigarrow V \end{array}}{E; \phi \vdash \exists \alpha . C \rightsquigarrow (T, V)}$$

$$\frac{E(x) = \forall \vec{a} . T \qquad \phi(\tau) = [\vec{T} / \vec{a}] T}{E; \phi \vdash x \; \tau \rightsquigarrow \vec{T}} \qquad \frac{E; \phi \vdash C \rightsquigarrow V}{E; \phi \vdash \mathit{map} \; f \; C \rightsquigarrow f \; V}$$

$$\frac{\begin{array}{c} E, \vec{b} \vdash T \text{ ok} \qquad \vec{b} \subseteq \vec{a} \\ E, \vec{a}; \phi[\alpha \mapsto T] \vdash C_1 \rightsquigarrow V_1 \\ E, x : \forall \vec{b} . T; \phi \vdash C_2 \rightsquigarrow V_2 \end{array}}{E; \phi \vdash \mathsf{let} \; x = \lambda \alpha . C_1 \; \mathsf{in} \; C_2 \rightsquigarrow (\forall \vec{b} . T, \Lambda \vec{a} . V_1, V_2)}$$

Figure 9. Semantics of constraints with a value

where the free term variables of C are in the domain of E and the free type variables of C are in the domain of ϕ. This judgement means that *in the context E, the constraint C is satisfied by ϕ and produces the value V*.

The definition of this judgement appears in Figure 9.

The meaning of truth and conjunction is straightforward. The constraint $\mathtt{true}$ produces the empty tuple $()$, while a conjunction $C_1 \wedge C_2$ produces a pair (V_1, V_2), as announced earlier.

Quite obviously, an equation $\tau_1 = \tau_2$ is satisfied by ϕ only if the decoded types $\phi(\tau_1)$ and $\phi(\tau_2)$ are equal. Such an equation produces the empty tuple $()$.

The rule for existential quantification states that the constraint $\exists\alpha.C$ is satisfied iff there exists an assignment of α that satisfies C. More precisely, ϕ satisfies $\exists\alpha.C$ iff there exists a decoded type T such that $\phi[\alpha \mapsto T]$ satisfies C. This is a non-deterministic specification, not an executable algorithm, so the witness T is "magically" chosen. (The first premise requires the free type variables of T to be in the domain of E.) Finally, the rule states that if C produces the value V, then $\exists\alpha.C$ produces the pair (T, V). This means that the end user has access to the witness T.

An instantiation constraint $x\ \tau$ is satisfied by ϕ iff the decoded type $\phi(\tau)$ is an instance of the type scheme associated with x. The first premise looks up this type scheme, say $\forall\vec{a}.T$, in E. The second premise checks that the instance relation holds: that is, for some vector $\vec{T}$, the decoded types $\phi(\tau)$ and $[\vec{T}/\vec{a}]T$ are equal. The vector $\vec{T}$ is again "magically" chosen. Finally, this constraint produces the value $\vec{T}$. This means that the end user has access to the witnesses $\vec{T}$.

The constraint *map* f C is satisfied iff C is satisfied. If C produces V, then *map* f C produces $f\ V$. This allows the end user to transform, or post-process, the value produced by a constraint.

For the moment, let us read the rule that describes the constraint $\mathsf{let}\ x = \lambda\alpha.C_1\ \mathsf{in}\ C_2$ as if the vector $\vec{b}$ was equal to $\vec{a}$. We explain why they might differ in §B.

The rule's third premise requires that C_1 be satisfied by mapping α to T, in a context extended with a number of new type variables $\vec{a}$. This means that every instance of the type scheme $\forall\vec{a}.T$ satisfies the constraint abstraction $\lambda\alpha.C_1$. Again, $\vec{a}$ and T are "magically" chosen. The last premise requires that, under the assumption that x is associated with the type scheme $\forall\vec{a}.T$, the constraint C_2 be satisfied.

The conclusion states that, if C_1 and C_2 respectively produce the values V_1 and V_2, then the constraint $\mathsf{let}\ x = \lambda\alpha.C_1\ \mathsf{in}\ C_2$ produces the triple $(\forall\vec{a}.T, \Lambda\vec{a}.V_1, V_2)$. The first component of this triple is the type scheme that has been associated with x while examining C_2. The second component is V_1, in which the "new" type variables $\vec{a}$ have been made anonymous, so as ensure that "if $E; \phi \vdash C \rightsquigarrow V$ holds, then the free type variables of V are in the domain of E". The last component is just V_2.

Our proposed definition of the judgement $E; \phi \vdash C \rightsquigarrow V$ should be taken with a grain of salt, as we have not conducted any proofs about it. One might wish to prove that it is in agreement with the semantics of raw constraints, as defined by Rémy and the present author [19, p. 414].

The judgement $E; \phi \vdash C \rightsquigarrow V$ can be used to express the specification of the constraint solver. Let C be a closed constraint. The solver is correct: if the solver, applied to C, succeeds and produces a value V, then the judgement $\emptyset; \emptyset \vdash C \rightsquigarrow V$ holds, i.e., C is satisfiable and V can be viewed as a correct description of the solution. The solver is complete: if there exists a value V such that $\emptyset; \emptyset \vdash C \rightsquigarrow V$ holds, then the solver, applied to C, succeeds and produces a value V'[7].

B. On redundant quantifiers

B.1 The issue

The last rule of Figure 9, read in the special case where $\vec{b}$ is $\vec{a}$, states that such the constraint $\mathsf{let}\ x = \lambda\alpha.C_1\ \mathsf{in}\ C_2$ produces a triple of the form $(\Lambda\vec{a}.V_1, \forall\vec{a}.T, V_2)$. We pointed out earlier (§A) that abstracting the type variables $\vec{a}$ in the value V_1 is necessary, as all of these variables may appear in V_1. However, it could happen that some of these variables do not occur in the type T, which means that the type scheme $\forall\vec{a}.T$ exhibits redundant quantifiers.

In other words, simplifying the last rule of Figure 9 by forcing a coincidence between $\vec{b}$ and $\vec{a}$ would make good sense, but would lead to an inefficiency.

For instance, consider the ML term:

$$\mathsf{let}\ u = (\lambda f.())\ (\lambda x.x)\ \mathsf{in}\ \ldots$$

The left-hand side of the let construct applies a constant function, which always returns the unit value, to the identity function. Thus, intuitively, it seems that the type scheme assigned to the variable u should be just *unit*. However, if one constructs the contraint-with-a-value that describes this term (as per Figure 8) and if one applies the rules of Figure 9 in the most general manner possible, so as to determine what value this constraint produces, one finds that, at the let construct, one must introduce a type variable a, which stands for the type of x. In this case, the vector $\vec{a}$ consists of just a. The value V_1 is the explicitly-typed version of the function application, that is:

$$(\lambda f : a \to a.())\ (\lambda x : a.x)$$

The type T of this term is just *unit*. We see that the binder "Λa" in $\Lambda a.V_1$ is essential, since a occurs in V_1, whereas the binder "$\forall a$" in $\forall a.T$ is redundant, since a does not occur in T. The translation of our ML term in System F is as follows:

$$\mathsf{let}\ u = \Lambda a.(\lambda f : a \to a.())\ (\lambda x : a.x)\ \mathsf{in}\ \ldots$$

The type of u in System F is $\forall a.\mathit{unit}$ (which is not the same as *unit*). In the right-hand side $(\ldots)$, every use of u must be wrapped in a type application, which instantiates the quantifier a. But, one may ask, what will a be instantiated with? Well, naturally, with another type variable, which itself will later give rise to another redundant quantifier, and so on. Redundant quantifiers accumulate and multiply!

This is slightly unsatisfactory. In fact, formally, this may well violate our claim that the constraint solver has good complexity under the assumption that “type schemes have bounded size” [12]. Indeed, a plausible clarification of McAllester’s hypothesis is that “all type schemes ever inferred, once deprived of their redundant quantifiers, have bounded size”, and that does *not* imply that “all type schemes ever inferred, in the absence of redundant quantifier elimination, have bounded size”.

B.2 A solution

We address this issue by allowing the vectors $\vec{b}$ and $\vec{a}$ to differ in the last rule of Figure 9. In general, $\vec{b}$ is a subset of $\vec{a}$ (second premise) that the variables in $\vec{b}$ may occur in T while those in $\vec{a} \setminus \vec{b}$ definitely do not occur in T (first premise). All of the variables $\vec{a}$ are needed to express the solution of C_1 (third premise), hence all of them may appear in the value V_1. Thus, one must abstract over $\vec{a}$ in the value V_1. But the solver examines the constraint C_2 under the assumption that x has type scheme $\forall \vec{b}.T$, where the redundant quantifiers have been removed (last premise).

We can now explain in what way the Advisor’s informal code (§1) was misleading. In the Advisor’s discourse, $\forall \vec{\beta}.\theta$ is supposed to be a canonical form of a (raw) constraint abstraction, or in other words, a principal type scheme. Certainly it is permitted to

[7] We cannot require V' to be V, because the judgement $E; \phi \vdash C \rightsquigarrow V$ is non-deterministic: when E and C are fixed, there may be multiple choices of ϕ and V such that the judgement holds. In practice, a reasonable constraint solver always computes a most general solution ϕ and the value V that corresponds to it. One might wish to build this guarantee into the statement of completeness.

assume that it does not have any redundant quantifiers. So, β there corresponds to $\vec{b}$ here. When the Advisor suggested Λ-abstracting over $\vec{\beta}$, he was wrong. This is not enough: one must Λ-abstract over $\vec{a}$, or one ends up with dangling type variables.

Naturally, the potential mismatch between $\vec{a}$ and $\vec{b}$ means that one must be careful in the construction of an explicitly-typed term. When viewed as ML type schemes, $\forall\vec{a}.T$ and $\forall\vec{b}.T$ are usually considered equivalent; yet, when viewed as System F types, they most definitely are not.

For this reason, it does not make sense to translate the ML term "$\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$" to the System F term "$\mathsf{let}\ x = \Lambda\vec{a}.t_1'\ \mathsf{in}\ t_2'$". The subterm $\Lambda\vec{a}.t_1'$ has type $\forall\vec{a}.T$, but the subterm t_2' is constructed under the assumption that x has type $\forall\vec{b}.T$. Thus, one must adjust the type of x, by inserting an explicit coercion:

$$\mathsf{let}\ x = \Lambda\vec{a}.t_1'\ \mathsf{in}\ \mathsf{let}\ x = (x : \forall\vec{a}.T :> \forall\vec{b}.T)\ \mathsf{in}\ t_2'$$

This coercion is not a primitive construct in System F. It can be encoded via a suitable series of type abstractions and applications. The function *coerce* used at the end of Figure 8 (whose definition is omittted) performs this task. This function can be made to run in time linear in the size of $\vec{a}$ and can be made to produce no code at all if the lists $\vec{a}$ and $\vec{b}$ are equal.

The need to introduce a coercion may seem inelegant or curious. In fact, it is a phenomenon that becomes more plainly obvious as the source language grows. For instance, several real-world languages of the ML family have a construct that simultaneously performs pattern matching and generalization, such as "$\mathsf{let}\ (x, y) = t\ \mathsf{in}\ u$". It is clear that the quantifiers of the type scheme of x (resp. y) are in general a subset of the quantifiers that appear in the most general type scheme of the term t. Furthermore, upon closer investigation, one discovers that the type of the translated term t' is of the form $\forall\vec{\alpha}.(\tau_1 \times \tau_2)$, whereas deconstructing a pair in System F requires a term of type $(\forall\vec{\alpha}_1.\tau_1) \times (\forall\vec{\alpha}_2.\tau_2)$. Thus, a coercion is required in order to push the universal quantifiers into the pair and get rid, within each component, of the redundant quantifiers. In System F, such a coercion can be encoded, at the cost of an η-expansion. In an extension of System F with primitive erasable coercions, such as Cretin and Rémy's [1], this cost is avoided.