Hindley-Milner Elaboration in Applicative Style
François Pottier

To cite this version:

HAL Id: hal-01081233
https://inria.hal.science/hal-01081233
Submitted on 7 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Hindley-Milner Elaboration in Applicative Style

François Pottier
INRIA
Francois.Pottier@inria.fr

Abstract

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 suddenly entered the room. “I need your help,” she began. 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 suddenly entered the room. “I need your help,” she began. 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 suddenly entered the room. “I need your help,” she began.

“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 τ' 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:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM’14, September 1–6, 2014, Gothenburg, Sweden.
ACM 978-1-4503-2873-9/14/09/…$15.00.
http://dx.doi.org/10.1145/2628136.2628145
• If $t$ is $x$, and if the type scheme of $x$ is $\forall \beta. \theta$, then
  - for some vector $\vec{r}$, the types $\tau$ and $[\vec{r}/\beta] \theta$ should be equal, and $t'$ should be the type application $x \, \vec{r}$.

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

• If $t$ is $let \ x = t_1 \ in \ t_2$, then:
  - the constraint abstraction "$\lambda x. (t_1 \ has \ type \ \alpha)$" should have some canonical form $\forall \beta. \theta$,
  - assuming that the type scheme of $x$ is $\forall \beta. \theta$, the term $t_2$ should have type $\tau$,
  - and $t'$ should be let $x = \Lambda \beta.t_1 \ in \ t_2$.

"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 $\Lambda$-abstracting the type variables $\beta$ 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 $C ::= \text{true} \mid C \land C \mid \tau = \tau \mid \exists \alpha. C$. 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]:

$$\begin{align*}
\tau & ::= \alpha \mid \tau \rightarrow \tau \mid \ldots \\
C & ::= \text{true} \mid C \land C \mid \tau = \tau \mid \exists \alpha. C \\
&\mid \text{let } x = \lambda \alpha.C \text{ in } C \\
&\mid x \tau
\end{align*}$$

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 $\tau$. One way of defining or explaining the meaning of these constructs is to expand them away via the following substitution law:

$$\begin{align*}
\text{let } x = \lambda \alpha.C_1 \text{ in } C_2 & \equiv \exists \alpha.C_1 \land [\lambda \alpha.C_1/x]C_2
\end{align*}$$

That is, the let constraint on the left-hand side is equivalent to (a) requiring that there exist at least one value of $\alpha$ for which $C_1$ holds; and (b) replacing 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 annotated 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 $\tau$ to a constraint $[t : \tau]$, which informally means “$t$ has type $\tau$”. It can be defined as follows [19]:

$$\begin{align*}
[x : \tau] & = x \tau \\
[\lambda x.u : \tau] & = \exists \alpha_1 \alpha_2 \left( \begin{array}{l}
\tau = \alpha_1 \rightarrow \alpha_2 \land \\
\text{def } x = \alpha_1 \text{ in } [u : \alpha_2]
\end{array} \right) \\
[t_1, t_2 : \tau] & = \exists \alpha \left( [t_1 : \alpha \rightarrow \tau] \land [t_2 : \alpha] \right) \\
\text{let } x = t_1 \text{ in } t_2 & = \text{let } x = \lambda \alpha.t_1 : \alpha \text{ in } [t_2 : \tau]
\end{align*}$$

There, def $x = \tau$ in $c$ is a short-hand for let $x = \lambda \alpha.(x = \tau) \ in \ c$. 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 can 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 $\alpha$. This implies that a solved form of $[t : \alpha]$ cannot constrain any type variables other than $\alpha$. Such a solved form could be, for instance, $\alpha = \text{unit}$, 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. Furthermore, in this approach, the type inference and elaboration process is split
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 \((\exists \alpha. C_1) \land C_2\), for instance, the final value of \(\alpha\) 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 \(\alpha\) 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 \(\alpha\). 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 \(\tau\) to exist, subject to certain constraints, then goes on and uses \(\tau\) 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 \(\alpha\) of the meta-language\(^4\), we would like to be able to construct “\(\alpha\)-constraints”, that is, constraints which (once satisfied) produce a result of type \(\alpha\).

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 “\(\alpha\)-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 \(\alpha\). Our OCaml implementation (§4) is based on this representation.

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

\[
C ::= \begin{array}{l}
\text{true} \mid C \land C \mid \tau = \tau \mid \exists \alpha. C \\
\text{let } x = \lambda \alpha.C \text{ in } C \\
\text{map } f C
\end{array}
\]

This syntax is identical to that of raw constraints (§2), with one addition. A new construct appears: \(\text{map } f C\), where \(f\) is a metafunction. The intention is that this constraint is satisfied when \(C\) is satisfied, and if the constraint \(C\) produces some value \(V\), then \(\text{map } f C\) 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 \(C_1 \land C_2\) 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 \(C_1 \land C_2\) 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 \(\alpha\) 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”, which we write \(\tau\). 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 type scheme associated with \(x\) must be instantiated in order to obtain the type \(\tau\).

A constraint of the form \(let \ x = \lambda \alpha.C \text{ in } C_2\) produces a tuple of three values:

1. The canonical form of the constraint abstraction \(\lambda x. C_1\). 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 \(\forall \beta. T\).
2. A value of the form \(\vec{\alpha}. \vec{V}_1\), 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 “\(\vec{\alpha}\) 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 \(\exists \alpha. . . . \), then \(V_1\) is a pair \((T, . . .)\), where the decoded type \(T\) may have free decoded type variables. By introducing the binder “\(\vec{\alpha}\)”, the solver is telling the user that, at this particular place, the type variables \(\vec{\alpha}\) should be introduced. (In the OCaml code, the solver separately returns \(\vec{\alpha}\) 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 \(\vec{\alpha}\) and \(\vec{\beta}\). The short answer is, in general, \(\vec{\beta}\) is a subset of \(\vec{\alpha}\). 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
  val iter2: (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 iter2, 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
μa.t. This feature is required for two reasons: (a) the solver option-
ally 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
  end
  type ischeme
  type rawco =
    CTrue
    CConj of rawco × rawco
    CEq of variable × variable
    CExist of variable × rawco
    CInstance of tyvar × variable × variable list WriteOnceRef.t
    CDef of tyvar × variable × rawco
    CLet of (tyvar × variable × ischeme WriteOnceRef.t) list
       × rawco
       × rawco
       × variable list WriteOnceRef.t
  exception Unbound of tyvar
  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 references3.

3Instead 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 \texttt{solve} expects a closed constraint and determines whether it is satisfiable. The Boolean parameter indicates whether recursive types, in the style of \texttt{ocaml -rectypes}, are legal.

If the constraint is unsatisfiable, an exception is raised. The exception \texttt{Unify} \((v_1, v_2)\) means that the type variables \(v_1\) and \(v_2\) cannot be unified; the exception \texttt{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 \texttt{Unify} and \texttt{Cycle} or via the write-once references, consists of values of type \texttt{variable} and \texttt{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 \texttt{tyvar}, \(ty\), etc. These decoding functions are supposed to be used only after the constraint solving phase is finished.

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

The function \texttt{new_decoder} constructs a new type decoder, that is, a function of type \texttt{variable \rightarrow 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 \texttt{variable} to \(ty\)), so that the overall cost of decoding remains linear in the size of the graph.

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 \texttt{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 \(\alpha\) \(co\) of constraints that produce a value of type \(\alpha\); (b) a number of ways of constructing such constraints; and (c) a single function, \texttt{solve}, that solves and evaluates such a constraint and (if successful) produces a final result of type \(\alpha\).

The type \(\alpha\) \(co\) is internally defined as follows:

\[
\text{type } \alpha \text{ co } = \\
\text{rawco} \times (\text{env} \rightarrow \alpha)
\]

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

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

\[
\text{module Make}
\begin{align*}
&\text{ (}X : \text{TEVAR)} \\
&\text{ (}S : \text{STRUCTURE)} \\
&\text{ (}O : \text{OUTPUT with type } \alpha \text{ structure } = \alpha \text{ S.structure)}
\end{align*}
\]

\[
\begin{align*}
&\text{ sig } \\
&\text{ open X} \\
&\text{ open S} \\
&\text{ open O} \\
&\text{ type variable}
\end{align*}
\]

\[
\begin{align*}
&\text{ type } \alpha \text{ co} \\
&\text{ val pure: } \alpha \rightarrow \alpha \text{ co} \\
&\text{ val (^&): } \alpha \text{ co } \rightarrow \beta \text{ co } \rightarrow (\alpha \times \beta) \text{ co} \\
&\text{ val map: } (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta \text{ co} \\
&\text{ val (\rightarrow): variable } \rightarrow \text{ variable } \rightarrow \text{ unit co} \\
&\text{ val (\text{--}:): variable } \rightarrow \text{ variable structure } \rightarrow \text{ unit co} \\
&\text{ val exist: variable } \rightarrow \alpha \text{ co } \rightarrow (ty \times \alpha) \text{ co} \\
&\text{ val instance: tevar } \rightarrow \text{ variable } \rightarrow ty \text{ list co} \\
&\text{ val def: tevar } \rightarrow \text{ variable } \rightarrow \alpha \text{ co } \rightarrow \alpha \text{ co} \\
&\text{ val let: tevar } \rightarrow (\text{variable } \rightarrow \alpha \text{ co } \rightarrow \beta \text{ co } \rightarrow \\
&\text{(scheme } \times tyvar \text{ list } \times \alpha \times \beta) \text{ co}) \\
\end{align*}
\]

\[
\begin{align*}
&\text{ exception Unbound of tevar} \\
&\text{ exception Unify of ty } \times ty \\
&\text{ exception Cycle of ty} \\
&\text{ val solve: bool } \rightarrow \alpha \text{ co } \rightarrow \alpha \\
\end{align*}
\]

\text{ end}
\]

Figure 5. The solver’s high-level interface
If \( c \) is a constraint of type \( \alpha \ co \) and if the user-supplied function \( f \) maps \( \alpha \) to \( \beta \), then \( map \ f \ c \) is a constraint of type \( \beta \ co \). Its logical meaning is the same as that of \( c \).

### Let

\[
\text{let } map f (rc, k) = \text{rc}.
\]

### Fun

\[
\text{fun } env \to f (k \ env)
\]

Equipped with the combinators \( \text{pure} \), \( \wedge \), and \( \text{map} \), the type constructor \( \text{co} \) is an applicative functor. More specifically, it is an instance of McBride and Paterson’s type class \( \text{Monoidal} \) [13, §7]. Furthermore, the combinator \( \wedge \) is commutative, that is, it enjoys the following law:

\[
\alpha \wedge \beta \equiv \text{map} \ (\alpha \wedge \beta)
\]

where \( \text{swap} \ (a_2, a_1) \) is \( (a_1, a_2) \). This law holds because \( \text{CConj} \) 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 \( \text{co} \) is not a monad, as there is no sensible way of defining a \( \text{bind} \) operation of type \( \alpha \ co \rightarrow (\alpha \rightarrow \beta \ co) \rightarrow \beta \ co \). In an attempt to define \( \text{bind} \) \( (\alpha_1, \alpha_2) f_2 \), one would like to construct a raw conjunction \( \text{CConj} (\alpha_1, \alpha_2) \). In order to obtain \( \alpha_2 \), one must invoke \( f_2 \), and in order to do that, one needs a value of type \( \alpha_1 \), 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 \( \text{bind} \). This phenomenon, which was observed in Swierstra and Duponcheel’s LL1(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 \( \text{bind} \), we have \( \text{map} \). When one builds a constraint \( \text{map} \ f \ c \), 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 \( \text{map} \) with reversed argument order:

\[
\text{let } (\text{map} s) \ a \ f = \text{map} s f a
\]

### Combinators

---

---

### Combinator

\[
\text{let } \alpha \rightarrow \beta = \text{CEq} (\alpha, \beta).
\]

### Fun

\[
\text{fun } env \rightarrow ()
\]

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

\[
\text{let } (\alpha \rightarrow) v_1 v_2 = \text{CEq} (v_1, v_2).
\]

### Fun

\[
\text{fun } env \rightarrow()
\]

The constraint \( v_1 \rightarrow v_2 \) is also an equation, whose second member is a shallow type.

The constructor, \( \text{exist} \), builds an existentially quantified constraint \( \exists \alpha \ C \). Its argument is a user-defined function \( f \) which, once supplied with a fresh type variable \( \alpha \), must construct \( C \). It is defined as follows:

\[
\text{let } \text{exist } f = \\
\text{let } \nu v = \text{fresh } \text{None in } \\
\text{let } rc, k = f \ v \text{ in } \\
\text{CEnt} (v, rc), \\
\text{fun } env \rightarrow \\
\text{let } \text{decode } = \text{env in } \\
\text{(decode } v, k \text{ env})
\]

At constraint construction time, we create a fresh type variable \( v \) and pass it to the client by invoking \( f \). This produces a constraint, i.e., a pair of a raw constraint \( rc \) and a continuation \( k \). We can then construct the raw constraint \( \text{CEnt} (\nu, rc) \). 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 \( \text{exist } f \) has type \((\nu \times \alpha) \ co\); 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

\[
\text{let } \text{instance } x v = \\
\text{let } \text{witnesses } = \text{WriteOnceRef.create()} \text{ in } \\
\text{Clnt} (x, v, \text{witnesses}), \\
\text{fun } env \rightarrow \\
\text{let } \text{decode } = \text{env in } \\
\text{List.map decode (WriteOnceRef.get witnesses)}
\]

At constraint construction time, we create an empty write-once reference, \( \text{witnesses} \), and construct the raw constraint \( \text{Clnt} (x, v, \text{witnesses}) \), 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, \( \text{let} \), 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 \( \lambda x . c_1 \) (i.e., when applied to a fresh type variable \( x \), 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 \( \text{instance} \) above. As promised earlier (§3), this constraint produces the following results:

- A decoded type scheme, \( \forall \beta.T \), of type scheme. It can be viewed as the canonical form of the constraint abstraction \( \lambda x . c_1 \). This type scheme has been associated with \( x \) while solving \( c_2 \). We guarantee that \( \beta \) is a subset of \( \alpha \) (see §A and §B for details).
- A vector of decoded type variables \( \tilde{a} \), of type \( ty \ list \), and a value \( v_1 \), of type \( \alpha \), produced by \( c_1 \). The type variables \( \tilde{a} \) 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 “\( \tilde{a}.V_1 \)” in §3.
- A value \( v_2 \), produced by \( c_2 \), of type \( \beta \).

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

\[
\text{exception } \text{Unify of } O.ty \times O.ty \\
\text{exception } \text{Cycle of } O.ty
\]

and implement \( \text{solve} \) as follows:

\[
\text{let } \text{solve } \text{rectypes } (rc, k) = \\
\begin{align*}
\text{begin} & \text{try } \\
& \text{Lo.solve } \text{rectypes } rc \\
& \text{with } \\
& | Lo.\text{Unify } (v_1, v_2) \rightarrow \\
& \text{let } \text{decode } = \text{new_decoder } \text{true } \text{in } \\
& \text{raise } (\text{Unify } (\text{decode } v_1, \text{decode } v_2)) \\
& | Lo.\text{Cycle } v \rightarrow \\
& \text{let } \text{decode } = \text{new_decoder } \text{true } \text{in } \\
& \text{raise } (\text{Cycle } (\text{decode } v)) \\
\end{align*}
\]

The computation is in two phases. First, the low-level solver, \( \text{Lo.solve} \), is applied to the raw constraint \( rc \). Then, elaboration
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 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 α → variable → β co into one of type α → variable structure → β 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 \( \Lambda \)-abstracted in the term \( f \), as explained in the description of \( \text{let} \) (§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 \( \text{coerce } a \ b \ (F. \bar{\text{Var}} \ x) \). 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 \( \alpha \ co \) 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 \( \text{let} \) for defining a constraint abstraction with \( n \) entry points; this allows dealing with ML's "let \( p = t_1 \) in \( t_2 \)" syntax, 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 \( \text{Cycle} \) and \( \text{Unify} \). 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 \( \text{Unify} \) 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 let \( \alpha = T \) in \( t \), where \( \alpha \) 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 \( \lambda \)-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

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 $\alpha$ to denote a type variable. Such a variable may appear in a type $\tau$ 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:

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

We refer to $a$ as a “decoded” type variable and to $T$ as a “decoded” type. In the following, we write $\phi$ for a partial mapping of the type variables $\alpha$ to decoded types $T$. The application of $\phi$ to a type $\tau$ produces a decoded type $\phi(\tau)$. 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 $V$:

$$V ::= (V_1, \ldots, V_n) \mid \forall a.T \mid \Lambda a.V \mid \ldots$$

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 $\alpha$ are in scope, we add a third argument, namely an environment $E$, whose structure is as follows:

$$E ::= \emptyset \mid E, x : \forall a.T \mid E, \alpha$$

(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 $\phi$. Thus, we define a judgement of the following form:

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

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 $\phi$. This judgement means that in the context $E$, the constraint $C$ is satisfied by $\phi$ and produces the value $V$.

The definition of this judgement appears in Figure 9.

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

Quite obviously, an equation $\tau_1 = \tau_2$ is satisfied by $\phi$ 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 $\alpha$ that satisfies $C$. More precisely, $\phi$ 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 $\phi$ 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 $T$. This means that the end user has access to the witnesses $T$.

The constraint $\text{map } f \ C$ is satisfied iff $C$ is satisfied. If $C$ produces $V$, then $\text{map } f \ C$ 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 $let \ x = \lambda \alpha.C_1 \ in \ C_2$ as if the vector $\vec{b}$ was equal to $\vec{a}$. We explain why they might differ in §8.

The rule’s third premise requires that $C_1$ be satisfied by mapping $\alpha$ 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 $let \ x = \lambda \alpha.C_1 \ in \ C_2$ 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 C2. The second component is V1, in which the “new” type variables ⃗a have been made anonymous, so as to ensure that “if E; φ ⊢ C → V holds, then the free type variables of V are in the domain of E”. The last component is just V2.

Our proposed definition of the judgement E; φ ⊢ C → 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; φ ⊢ C → 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 θ; C ⊢ 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 θ; C ⊢ V holds, then the solver, applied to C, succeeds and produces a value V′.

B. On redundant quantifiers

B.1 The issue

The last rule of Figure 9, read in the special case where ⃗b is ⃗a, states that such the constraint let x = λa. C1 in C2 produces a triple of the form (λa. V1, ∀a. V1, V2). We pointed out earlier (§8A) that abstracting the type variables ⃗a in V1 is necessary, as all of these variables may appear in V1. However, it could happen that some of these variables do not occur in the type T, which means that the type scheme ∀a. V1 exhibits redundant quantifiers.

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

For instance, consider the ML term:

\[ \text{let } u = (\lambda f. (\lambda x.x)) \text{ in } \ldots \]

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 constraint-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 ⃗a consists of just a. The value V1 is the explicitly-typed version of the function application, that is:

\[ (\lambda f : a \rightarrow a.()) (\lambda x : a.x) \]

The type T of this term is just unit. We see that the binder “\(\lambda a\)” in \(\Lambda a. V_1\) is essential, since a occurs in V1, whereas the binder “\(\lambda a\)” 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:

\[ \text{let } u = \Lambda a. (\lambda f : a \rightarrow a.()) (\lambda x : a.x) \text{ in } \ldots \]

The type of u in System F is \(\forall a. \text{unit}\) (which is not the same as unit). In the right-hand side (.), 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 ⃗b and ⃗a to differ in the last rule of Figure 9. In general, ⃗b is a subset of ⃗a (second premise) that the variables in b may occur in T while those in ⃗a \setminus ⃗b definitely do not occur in T (first premise). All of the variables ⃗a are needed to express the solution of C1 (third premise), hence all of them may appear in the value V1. Thus, one must abstract over ⃗a in the value V1. But the solver examines the constraint C2 under the assumption that x has type scheme \(\forall b. T\), 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, \(\forall \beta. \theta\) 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 assume that it does not have any redundant quantifiers. So, \(\beta\) there corresponds to \(\tilde{b}\) here. When the Advisor suggested \(\lambda\)-abstracting over \(\beta\), he was wrong. This is not enough: one must \(\lambda\)-abstract over \(\tilde{a}\), or one ends up with dangling type variables.

Naturally, the potential mismatch between \(\tilde{a}\) and \(\tilde{b}\) means that one must be careful in the construction of an explicitly-typed term. When viewed as ML type schemes, \(\forall \tilde{a}. T\) and \(\forall b. T\) 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 “let x = t1 in t2” to the System F term “let x = \(\Lambda \tilde{a}. t'_1\) in t'_2”. The subterm \(\Lambda \tilde{a}. t'_1\) has type \(\forall \tilde{a}. T\), but the subterm t'_2 is constructed under the assumption that x has type \(\forall \tilde{b}. T\). Thus, one must adjust the type of x, by inserting an explicit coercion:

\[ \text{let x = } \Lambda \tilde{a}. t'_1 \text{ in } \{ x : \forall \tilde{a}. T : : } \forall \tilde{b}. T \text{ in t'}_2 \]

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 omitted) performs this task. This function can be made to run in time linear in the size of \(\tilde{a}\) and can be made to produce no code at all if the lists \(\tilde{a}\) and \(\tilde{b}\) 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 “let (x, y) = t 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 \(\forall \tilde{a}. (\tau_1 \times \tau_2)\), whereas deconstructing a pair in System F requires a term of type \((\forall \tilde{a}. \tau_1) \times (\forall \tilde{a}. \tau_2)\). 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 a \(\eta\)-expansion. In an extension of System F with primitive erasable coercions, such as Cretin and Rémy’s [1], this cost is avoided.