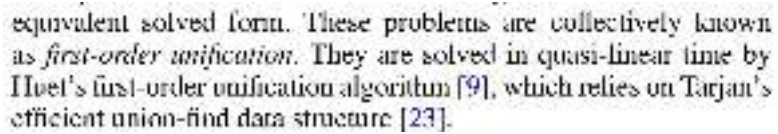
Hindley-Milner Elaboration in Applicative Style

Three processes

1. **Constraint generation**
   a. allocation of new unification variables
   b. unification constraints

2. **Constraint solving**
   a. most general unifier


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].

3. **Solidification/Elaboration**
   a. Annotate program with explicit type signatures and/or
   b. Translate to another language (e.g. Sys F)

Phase separation

- Practical implementations do all 3 at the same time
  a. Type-checking has to deal with quantifiers (implicit)

- (Some) theoretical presentations try to separate constraint solving
  a. That means, generate all the constraints first.
  b. All quantifiers are explicit.
  c. Then run a solver to see if a term is typable.
  d. Solver tightly coupled with the constraint language but not the object language.

Tight connection between 1. generation and 3. elaboration. So theoretical presentations skip elaboration or still look at solving.

Contraints-based type inference

- Constraint language

  C ::= true | C ∧ C | τ = τ | ∃ α.C
  - For the simply-typed lambda calculus.
  - (Works with HM if you solve at generalization points (let) the latest.)

For STLC this generates a unification constraint. τ has to be equal to the type assigned to x.

$$[\![x : \tau]\!] = x\ \tau$$

$$[\![\lambda x.u : \tau]\!] = \exists \alpha_1 \alpha_2 . \left(\begin{array}{l} \tau = \alpha_1 \to \alpha_2 \wedge \\ \textsf{def}\ x = \alpha_1\ \textsf{in}\ [\![u : \alpha_2]\!] \end{array} \right)$$

$$[\![t_1\ t_2 : \tau]\!] = \exists \alpha . ([\![t_1 : \alpha \to \tau]\!] \wedge [\![t_2 : \alpha]\!])$$

Contraints-based type inference

- Constraints abstractions
  C :: ... | let x = λα.C in C | x τ
  - Delays solving for generalization in HM

For HM this is a type-scheme instantiation. Exists + constraint.

$$[\![x : \tau]\!] = x\ \tau$$

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

$$[\![t_1\ t_2 : \tau]\!] = \exists \alpha . ([\![t_1 : \alpha \to \tau]\!] \wedge [\![t_2 : \alpha]\!])$$

$$[\![\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \tau]\!] = \mathsf{let}\ x = \lambda \alpha . [\![t_1 : \alpha]\!]\ \mathsf{in}\ [\![t_2 : \tau]\!]$$

Let generalization

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

Principal typing

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].

Elaboration!?

Elaboration via witnesses

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,

Side note: You kinda have to do this for the soundness proof of the algorithm, even if you don't consider elaboration.

Constraints with a value

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.

I think it's wrong to let the type constructor work on "any type [..] of the meta-language". You should consider a different category. More on that later.

Constraints with a value

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

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

Meaning

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.

Meaning

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

Meaning

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.

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

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

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

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

Figure 9. Semantics of constraints with a value

A very close and IMO better alternative is to interpret a constraint-with-a-value as a predicate transformer: Given a post-condition on the produced value the definition below essentially defines a pre-condition on the ground assignment ϕ.

This gives you essentially a program logic for verification of the type-inference algorithm.

Applicative interface

- Type constructor

The type $\alpha\ co$ is internally defined as follows:

type $\alpha\ co =$
$rawco \times (env \to \alpha)$

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

- Pure

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

let *pure* $a =$
$CTrue,$
fun $env \to a$

Applicative interface

- Map

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

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

- Monoidal composition

If c_1 and c_2 are constraints of types $\alpha\ co$ and $\beta\ co$, 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)
```

Applicative interface

- Unification constraints

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 → ()
```

- Existential quantification

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)
```

Elaboration

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

Low-level solver (out of scope)

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.

Not a monad

It is worth noting that *co* is not a monad, as there is no sensible way of defining a *bind* operation of type $\alpha\ co \rightarrow (\alpha \rightarrow \beta\ co) \rightarrow \beta\ co$. 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

Quantified Applicatives: API design for type-inference constraints

Olivier Martinot [1] Gabriel Scherer [2] Details
[1] Université Paris Diderot, Sorbonne Paris Cité, Paris, France
[2] PARTOUT - Automatisation et RepréSenTation: fOndation du calcUl et de la déducTion
LIX - Laboratoire d'informatique de l'École polytechnique [Palaiseau], Inria Saclay - Ile de France

1.3 Not a monad

Giving up on our page limit, let us explain why the `'a co` type of Inferno cannot be given the structure of a monad.

The witness `'a` is built with knowledge of the *global* solution to the inference constraints generated. Consider the applicative combinator `pair : 'a co -> 'b co -> ('a * 'b) co`; both arguments generate a part of the constraint, but the witnesses of `'a` and `'b` can only be computed once the constraints on *both* sides are solved. For example, when inferring the ML term `(x, x + 1)`, the type inferred for the first component of the pair is determined by unifications coming from the constraint of the second.

Implementing `bind : 'a co -> ('a -> 'b co) -> 'b co` would require building the witness for `'a` before the constraint arising from `'b co` is known; this cannot be done if `'a` requires the final solution.

For the abstractly inclined: internally `'a co` is defined as `raw_constraint * (solution -> 'a)` it is the composition of a "writer" monad that generate constraints and a "reader" monad consuming the solution. For the composition $W \circ R$ of two arbitrary monads to itself be a monad, a sufficient condition (Jones and Duponcheel, 1993) is that they commute: $(R \circ W) \to (W \circ R)$ – this suffices to define `join` : $(W \circ R \circ W \circ R) \to W \circ R$ from the `join` operation of W and R. Pushing the reader below the writer would require reading the solution before writing the constraint; this is not possible if we want the final solution.

Kind of a monad

I think it's wrong to look at a type constructor that works for any type. This limits everything to an applicative instead of a monad (in a different category).

```
Variable RawCo : Set.
Variable cand : RawCo -> RawCo -> RawCo.
Variable Env : Set.

Definition Enc (a : Set) : Set := Env -> a.
Definition Co (a : Set) : Set := RawCo * Enc a.

Definition bind {a b} (f : Enc a -> Co b) (C : Co a) : Co b :=
 let (rC1, ea) := C    in
 let (rC2, eb) := f ea in
 (cand rC1 rC2, eb).
```

Katamaran

A separation-logic based program verifier for SAIL

{PRE} CODE {POST}

CONSTRAINTS

Katamaran

- Constraint language https://github.com/skeuchel/katamaran/blob/export/src/MicroSail/Symbolic/Mutator.v#L871-L896
- Map and bind https://github.com/skeuchel/katamaran/blob/export/src/MicroSail/Symbolic/Mutator.v#L1084-L1118
- Weakest pre https://github.com/skeuchel/katamaran/blob/export/src/MicroSail/Symbolic/Mutator.v#L3621-L3642