Abstract

Lecture notes, April 14th, HOPL 2017. Don't mind the ornaments.

Soft Typing: Philosophy and Motivation

> "... the computer ... is willing to accept almost any sequence of instructions and make sense of them at its own level. That is the secret of the power, flexibility, and even reliability of computer hardware, and should therefore be cherished." — Tony Hoare [2]

🐍 Static type systems have well-known benefits.

1. A static type system can catch programming errors early (typos, logical mistakes), and exhaustively—by exploring all paths through the program.
2. Type signatures are a useful form of documentation.
3. Optimizing compilers can use type information to compile more efficient code.[1]
4. Designing types first can help guide the design of programs.

💣 Dynamic typing also has well-known benefits.

1. Unlike untyped languages, dynamically typed languages guarantee safety from memory errors (e.g. segfaults).
2. A dynamically typed language will run any syntactically correct program.
3. (consequence of point 2) Dynamically typed functions can be re-used on a larger class of arguments.
4. Dynamic typing is conceptually simpler than static typing; a programmer only needs to understand the runtime behavior of programs, not the behavior of the type checker.

⚔ **Note** to emphasize point 3 above, here is a program by Alan Mycroft that cannot be typed in the ML type system [3].

```
let rec f(x : structure) = case x of
  (basecase(y): ...
  |listcase(y): g(y, (hd, tl, null))
  |dlistcase(y): g(y, (dhd, dtl, dnull)))
```

[1] How useful is compiling with types? This was the subject of two HOPL talks in March, see: https://github.com/nuprl/hopl-s2017/blob/master/lecture_notes/2017-03-24.md and https://github.com/nuprl/hopl-s2017/blob/master/lecture_notes/2017-03-28.md.

```
and g(x : a, (xhd: a->b, xtl: a->a, xnull: a->bool)) =
  if xnull(x)
  then ()
  else (f(xhd x), g(xtl x, (xhd, xtl, xnull)))
```

Mycroft wrote this program while developing the ML compiler. **End Note.**

⚠ Why not combine all the benefits in a single system? The dream would be, an ideally-flexible type system that can type check any dynamically-typed program.

- Standard typecheckers try to prove that a program is "good", for some notion of "good" (think: *will not segfault at runtime*). The set of "good" programs is almost always recursively enumerable, therefore a *decidable* type checker **must reject some "good" programs**.
- Dynamically typed programs tend to rely heavily on untagged unions, recursive types, and structural subtyping. These features make type inference and type checking more difficult. In particular, Curry-style typing assumes that datatypes are disjoint.

  Philosophy: the types in a dynamically-typed program come from the *data* in that program. Functions rely heavily on structural subtyping, guided by dynamic type tests.

One approach to solving this problem is to design a new type system [3], in the hope that the system can accomodate enough programming idioms to be widely useful.

A second approach is soft typing. Given a type environment Γ and expression e, a soft type checker infers type information in e and rewrites the program to a semantically-equivalent and type-safe program e', which has static type τ.

$$\Gamma \vdash e \Rightarrow e' : \tau$$

Many more details follow.

Design Criteria for a Soft Type Checker

Fagan's dissertation outlines the design requirements for a soft type checker. The following list is adapted from Section 1.1 of the dissertation [1].

1. No syntactically correct programs are excluded (i.e. rejected by the type checker).
2. Run-time safety is assured by run-time checks if such checks cannot be safely eliminated.
3. The type checking process must be unobtrusive, where *unobtrusive* is characterized by two principles:

   Minimal text principle The type checking system should function in the absence of programmer-supplied type declarations.

Minimal failure principle The checker must pass a "large fraction" of dynamic programs that will not produce an execution error.

For example,

```
(+ 2 2)
;; no casts

(+ 2 (lambda (x) x))
;; add cast

(if #true 2 2)
;; well-typed

(if #true 2 (lambda (x) x))
;; well-typed, no casts

(define (flatten tree)
  (cond
   [(null? tree)
    '()]
   [(pair? tree)
    (append (flatten (first tree))
            (flatten (rest tree)))]
   [else ;; tree is a list
    (list tree)]))
;; well-typed, no casts
```

Any questions on the design criteria? These are important in subtle ways later.

Inferring Types from Untyped Code

Given the design criteria, the obvious challenge is how to infer descriptive types for annotation-free code. Hindley-Milner type inference works well for a similar problem in ML programs, so it is a natural place to start looking for a solution.

Hindley-Milner Type Inference

Types τ are disjoint, terms e are the lambda calculus. Type schemes σ quantify over type variables α.

$$\begin{array}{lcl} \tau & ::= & \alpha \mid \mathsf{Int} \mid \mathsf{Bool} \mid \tau \to \tau \\ e & ::= & x \mid \lambda\, x.\, e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\ \sigma & ::= & \tau \mid \forall\, \alpha.\, \sigma \\ \Gamma & ::= & \cdot \mid x{:}\sigma, \Gamma \\ S & ::= & \cdot \mid \alpha{:}\sigma, \Gamma \end{array}$$

Milner and Damas give syntax-directed type checking rules. Reading these rules bottom-up and using fresh type variables each time a new type is required yields an efficient inference algorithm (Algorithm W) parameterized by a solver.

Type Checking: $\boxed{\Gamma \vdash e : \tau}$ Type Inference: $\boxed{W(\Gamma, e) = (S, \tau)}$

Algorithm W has useful properties:

Theorem. *(soundness) If* $W(\Gamma, e) = (S, \tau)$*, then* $\Gamma \circ S \vdash e : \tau$

Theorem. *(completeness) If* $\Gamma \circ S_0 \vdash e : \tau_0$ *then* $W(\Gamma, e) = (S_1, \tau_1)$ *and there exists a substitution* S_2 *such that* $\Gamma \circ S_0 = \Gamma \circ S_1 \circ S_2$ *and* $\tau_0 = \tau_1 \circ S_2$ *modulo instantiation of quantified type variables in* τ_0

Theorem. *(principal types) If* $W(A, e) = (S, \tau)$ *then the generalization of* τ *under* $\Gamma \circ S$ *is a principal type of* e *under* $\Gamma \circ S$*.*

To summarize, the key ideas with Hindley-Milner are to generate *type equality constraints* from the syntax of the program, then *solve* the type constraints for a most general unifier (using Robinson Resolution).

Adapting Hindley-Milner

A non-starter idea is to directly apply the Hindley-Milner rules to dynamically typed programs. This is going to fail for many programs; Fagan gives a tautology checker as an example; the `flatten` program from above is another good example:

```
let rec tautology f =
  if f == true then true
  else if f == false then false
  else (tautology (f true)) and (tautology (f false))
```

At a minumum, the type system needs *recursive types* and *untagged unions*. The recursive types are not so difficult, but untagged unions mean that datatypes are no longer disjoint. In particular, non-disjointness means the type system needs a subtyping judgment; the rule for typing a function application must look like:

$$\frac{\Gamma \vdash f : \tau \to \tau' \qquad \Gamma \vdash e : \tau_e \qquad \tau_e \subseteq \tau}{\Gamma \vdash f\ e : \tau'}$$

Challenge 1: how to turn $\tau_e \subseteq \tau$ into a constraint? There are two "obvious" solutions (for some value of obvious).

- *(retained inequalities)* Do nothing! Keep $\tau_e \subseteq \tau$ as a constraint, and change the solver from Robinson resolution to something more sophisticated, that can handle subset constraints.

- *(convert to equalities)* Convert the inequality to an equality using the venerable technique of *slack variables*. In this case, $\tau_e \subseteq \tau$ generates the constraint $\tau_e \cup \alpha = \tau$.

☞ Fagan chooses to *convert* the inequalities.
Note: his choice is partially motivated by three perceived challenges with retained inequalities:

- the set of inequalities will grow quickly, may not be possible to simplify, and may require exponential time to solve;
- it is unclear how to convert an inconsistent set of constraints into a safety-ensuring runtime check;
- existing type systems with retained inequalities do not handle parametric polymorphism.

We will revisit these issues later, in the section on *Soft Typing with Conditional Types*. **End Note**
Challenge 2: the Hindley-Milner solver (Robinson resolution) finds most-general-unifiers only for terms in a *free* term algebra. But we have terms like $\tau_e \cup \alpha = \tau$ that use the union operator ($\cup$). Union is not free! It is associative, commutative, and idempotent (ACI).
What to do? We can start like computer science normally does, by asking mathematicians for help. And it turns out, general unification theory has results for ACI unification. Unfortunately ACI unificiation:

- produces multiple unifiers;
- does not support the circularity needed to infer recursive types;[2]
- does not offer any distributivity, but we need equations like $c\ \tau \cup c\ \tau' = c(\tau \cup \tau')$ (furthermore, Fagan notes that distributive, associative theories do not have unification methods).

What to do (2)? Fagan restricts the solution space to *discriminative* unions over rational regular trees.

- A *discriminative* union contains at most one occurrence of each type constructor. This solves the ACI issues because (AC) we can sort the constructors (I) and idempotence becomes a non-issue by definition.
- A *rational regular tree* is a possibly-infinite tree with a finite set of non-isomorphic subtrees. (Example: reduction graph for $(K\ I\ \Omega)$.)

Proposition. *Algorithm W can infer a substitution for two rational regular trees, if a substitution exists.*

Proposition. *There is a useful type system for dynamically typed programs whose types are isomorphic to rational regular trees.*

[2] This may have changed, but was true in 1990.

Diversion: The Rémy Encoding

Not going to prove those propositions, but want to give a general idea of how the encoding works. It is based on a technique by Didier Rémy for adding records with structural subtyping to ML.

Setup

Start with the ML language, want to add record types. A record type is a sequence of field labels l and types.

$$\begin{array}{rcl} e & ::= & \ldots \mid \{l = e, \ldots\} \\ \tau & ::= & \ldots \mid \{l : \tau, \ldots\} \end{array}$$

Also want structural subtyping, so if we have a function that extracts the value of the `left` label on a record, it works for records with more labels.

```
let get_left (x : {left : 'a}) : 'a =
  x.left

get_left {left = 1};;
get_left {left = 1, right = 2};;
```

Insight and Solution Sketch

Suppose the set l of labels has only 2 elements and records can only contain values of type Unit. Then we can encode all possible record types in a 4-element lattice, where each label is either present or absent in each type.

For example, $X =$ `xo` and $Y =$ `xx` are two possible record values. Their types are obvious.

One way to add subtyping would be to pretend that X and Y denote multiple values:

$$X = \texttt{oo} \mid \texttt{xo} Y = \texttt{oo} \mid \texttt{xo} \mid \texttt{ox} \mid \texttt{xx}$$

Then we can take the union or intersection of these "value sets". But we do not want multiple values; we want one value for X and one value for Y.

Rémy's solution is to encode the various types of X and Y with polymorphism. He does so in two steps:

1. Add *flags* to the labels in record types, indicating whether the label is present (+), absent (+), or unknown.
2. Introduce variables ϕ that range over the flags.

Using this encoding, the final types for X and Y are:

$$X : \{\phi_0 : \mathsf{Unit}, - : \mathsf{Unit}\} Y : \{\phi_1 : \mathsf{Unit}, \phi_2 : \mathsf{Unit}\}$$

The absent field in X has flag `-`. The present fields in X and Y have a flag variable, meaning they could be used or forgotten depending on the context. For example, the expression `if e then X else Y` has type $\{\phi : \mathsf{Unit}, \texttt{-} : \mathsf{Unit}\}$ by unifying ϕ_2 with `-`. Also, the type for `get_left` is $\{left : \texttt{+}, right : \phi\} \to \mathsf{Unit}$.

If you have more labels, just make longer types. That's the essence of the encoding.

> "Variants are to concrete data types what records are to labelled products." — Rémy

Adding structural subtyping for variants is straightforward; a variant type is the sum of all appropriately-flagged possibilities.

Rémy adds recursive types, records, and variants to the ML type system and proves the same *soundness*, *completeness*, and *principle types* properties as Milner. Clean and simple.

Fagan's Soft Type System

Back to soft typing, "the" type for expressions in a dynamically typed program is one large variant of all the possible base types.

Given the following grammar of types,

$$\tau \quad ::= \quad \mathsf{Int} \mid \mathsf{Bool} \mid \tau \to \tau$$

we can express:

- Integers : $[\mathsf{Int}+, \mathsf{Bool}-, \to -, \alpha_0, \alpha_1]$
- Booleans or Integers : $[\mathsf{Int}+, \mathsf{Bool}+, \to -, \alpha_0, \alpha_1]$
- Functions from integers to booleans : $[\mathsf{Int}-, \mathsf{Bool}-, \to +, [\mathsf{Int}+, \mathsf{Bool}-, \to -, \alpha_0, \alpha_1], [\mathsf{Int}-, \mathsf{Bool}+, \to -, \alpha_2, \alpha_3]]$

These are types written in Fagan's notation. The type variables cover "stucturally similar" positions. Wright makes further use of type variables, so we will use his system instead.

Wright's Soft Type System

Wright presents a soft type system for PureScheme.

Expressions, Operational Semantics, Stuck Programs

Terms e are lambda terms, and are made of values v, primitives c, base values b, and primitive functions p.

$$
\begin{array}{lcl}
e & ::= & v \mid (\mathsf{ap}\ e\ e) \mid (\mathsf{CHECK}-\mathsf{ap}\ e\ e) \mid (\mathsf{if}\ e\ e\ e) \mid (\mathsf{let}\ (x\ e)\ \mathsf{in}\ e) \\
v & ::= & c \mid x \mid (\lambda\, x.\, e) \\
c & ::= & b \mid p \\
b & ::= & \mathsf{integer} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \\
p & ::= & \mathsf{add1} \mid \mathsf{cons} \mid \mathsf{first} \mid \mathsf{second} \mid \mathsf{integer?} \mid \mathsf{CHECK\text{-}add1} \mid \ldots
\end{array}
$$

The unconventional thing about PureScheme is that applications are labeled as unsafe ($\mathsf{CHECK}-\mathsf{ap}$) or checked ($\mathsf{ap}$). Primitive operations also come in unsafe (e.g. $\mathsf{add1}$) and checked flavors ($\mathsf{CHECK\text{-}add1}$). If the checks are present, the operational semantics will raise a special error $\mathsf{checked}$ instead of undefined behavior.

$$
\begin{array}{lcll}
E[(\mathsf{ap}\ (\lambda\, x.\, e)\ v)] & \mapsto & E[e[x/v]] & \\
E[(\mathsf{CHECK}-\mathsf{ap}\ (\lambda\, x.\, e)\ v)] & \mapsto & E[e[x/v]] & \\
E[(\mathsf{let}\ x\ v\ \mathsf{in}\ e)] & \mapsto & E[e[x/v]] & \\
E[(\mathsf{if}\ v\ e_0\ e_1)] & \mapsto & E[e_0] & \text{when } v \neq \mathsf{false} \\
E[(\mathsf{if}\ \mathsf{false}\ e_0\ e_1)] & \mapsto & E[e_1] & \\
E[(\mathsf{ap}\ p\ v)] & \mapsto & E[v] & \text{when } v' = \delta(p, v) \\
E[(\mathsf{ap}\ p\ v)] & \mapsto & \mathsf{checked} & \text{when } \mathsf{checked} = \delta(p, v) \\
E[(\mathsf{CHECK}-\mathsf{ap}\ p\ v)] & \mapsto & E[v'] & \text{when } v' = \delta(p, v) \\
E[(\mathsf{CHECK}-\mathsf{ap}\ p\ v)] & \mapsto & \mathsf{checked} & \text{when } \mathsf{checked} = \delta(p, v) \\
E[(\mathsf{CHECK}-\mathsf{ap}\ b\ v)] & \mapsto & \mathsf{checked} &
\end{array}
$$

The δ metafunction what you would expect. Here are its cases for checked primitives:

$$
\begin{array}{lcll}
\delta(\mathsf{CHECK\text{-}p}, v) & = & v' & \text{when } v' = \delta(p, v) \\
\delta(\mathsf{CHECK\text{-}p}, v) & = & \mathsf{checked} & \text{when } \delta(p, v) \text{ is undefined}
\end{array}
$$

The purpose of showing these definitions was to define what it means for a program to be stuck. Stuck evaluation contexts have the following forms:

$$
\begin{array}{ll}
E[(\mathsf{ap}\ p\ v)] & \text{where } \delta(p, v) \text{ is undefined} \\
E[(\mathsf{CHECK}-\mathsf{ap}\ p\ v)] & \text{where } \delta(p, v) \text{ is undefined} \\
E[(\mathsf{ap}\ b\ v)] &
\end{array}
$$

Proposition. *(dynamic typing) terms e that always use $\mathsf{CHECK}-\mathsf{ap}$ and checked primitives never transition to a stuck state via the reflexive, transitive closure of $\mapsto$*

The (upcoming) soft type system will justify replacing some checked operations with unsafe ones.

Type System

First static types. Second, soft types.

The static type system will have the property SOUNDNESS.

The soft type system will have the property TODO.

Wright's grammar for static types τ is unconventional. To start, a type τ can be empty ($\emptyset$) or a type variable (α) or a recursive type ($\mu\,\alpha.\,\tau$). A type can also be a sequence of tagged (k), flagged (f), parameterized types, followed by a type variable or empty type. A tag is a type constructor; each constructor has a fixed arity. The grammar for flags is the same used by Rémy [4].

Type schemes Σ bind type variables. Substitutions S are used in unification.

$$\begin{array}{lcl} G0 & ::= & \alpha \mid \emptyset \\ G1 & ::= & (k^f\ \tau\ \ldots) \cup G1 \mid G0 \\ \tau & ::= & G0 \mid G1 \mid \mu\,\alpha.\,\tau \\ k & ::= & \mathsf{Int} \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Nil} \mid \rightarrow \\ f & ::= & + \mid \text{-} \mid \phi \\ \Sigma & ::= & \forall\,\nu.\,\Sigma \mid \tau \\ S & ::= & \cdot \mid \alpha = \Sigma, S \mid \phi = f, S \\ \nu & ::= & \alpha \mid \phi \end{array}$$

Note: the above grammar is incomplete, when we get to soft types we will need *absent* type and flag variables, along with absent types $\overline{\tau}$ and absent flags $\overline{f}$. Will clarify their purpose later, but for now the important point is that the full grammar for substitutions S maps absent variables to absent non-terminals:

$$S \quad ::= \quad \cdot \mid \alpha = \Sigma, S \mid \phi = f, S \mid \overline{\phi} = \overline{f}, S \mid \overline{\alpha} = \overline{\tau}, S$$

Examples:

- $4 : (\mathsf{Int}+) \cup \emptyset$
- $4 : (\mathsf{Int}+) \cup (\mathsf{True}\text{-}) \cup \emptyset$
- $\mathsf{true} : (\mathsf{True}+) \cup (\mathsf{False}+) \cup \emptyset$
- $\mathsf{integer?} : (\rightarrow +\alpha((\mathsf{True}+) \cup (\mathsf{False}+) \cup \emptyset)) \cup \emptyset$

(A little strange, but not as difficult to write as Fagan's.)

These types must also be discriminative; each constructor k can appear at most once in a union type. This means that type variables at the end of a union quantify *only* over missing constructors. Discriminativity also lets Wright reorder constructors in a union to a canonical order.

Example type scheme:

$$\forall\,\phi_0, \phi_1.\,(\mathsf{Int}\phi_0) \cup (\mathsf{Nil}\phi_1) \cup \emptyset$$

this scheme represents a lattice of four types, just like the Rémy-encoded type for records with two labels.

Types for primitive operations are next on the agenda. But you should keep in mind this general intuition:

- flag variables ϕ in a function domain allow subtyping at call sites

- type variables α in a function codomain let the function return a *supertype*

As a simple example illustrating the second point, consider this if-statement:

$$\mathsf{if}\ e\ 1\ \mathsf{false}$$

Its type should be the union of the types for the constants 1 and false. To make this union defined, their types are:

$$1 : \forall\,\alpha.\,(\mathsf{Int}+) \cup \alpha \mathsf{false} : \forall\,\alpha.\,(\mathsf{False}+) \cup \alpha$$

Hence the union is:

$$\forall\,\alpha.\,(\mathsf{Int}+) \cup (\mathsf{False}+) \cup \alpha$$

Static Primop Types
yolo
Static Type Checking
Soft Primop Types
Soft Type Checking
Correctness

Soft Typing for Scheme (R4RS)

Experience and Performance

Soft Typing with Conditional Types

Quasi-Static Typing

Global Tagging Optimization by Type Inference

Questions

In other words, these are things that do not fit in the real presentation, but are interesting and might come up as questions.

Q. Fundamental Theorem of Static Typing?
A. Fagan states and proves the "fundamental theorem", that a static type system must reject some meaningful programs [1]. Here are the relevant definitions from Fagan:

Let a *programming language* be a triple $\langle \mathsf{L}, \mathsf{V}, [\![\cdot]\!] \rangle$ where L is the syntax of the language, V are the value forms, and $[\![\cdot]\!]$ is a denotational semantics mapping syntax to values.

A programming language is *interesting* if (1) all terms have a value (2) the set V includes values denoting errors and non-terminating computations (3)

the language includes an if-statement (4) the set of terminating programs is recursively enumerable but not recursive.[3]

A *good program* written in a language $\langle \mathsf{L}, \mathsf{V}, [\![\cdot]\!] \rangle$ is a member of the set $\{e \mid e \in \mathsf{L} \wedge [\![e]\!] \neq \mathsf{wrong}\}$.

A *static type system* for a language $\langle \mathsf{L}, \mathsf{V}, [\![\cdot]\!] \rangle$ computes a recursive set $\mathsf{L}_W \subset \mathsf{L}$ such that for all $e \in \mathsf{L}$, the value $[\![e]\!]$ is not wrong.

Fagan then proves:

- **Lemma 1.1** *the set of good programs is not recursive*
- **Fundamental Theorem of Static Typing** *any set of well-typed programs* L_W *is a strict subset of the set of good programs*

The proofs are in Fagan's dissertation [1], but proving them is a nice exercise.

Q. What were Wright's extensions, how did they work?
A.

Appendix

References

[1] Mike Fagan. *Soft typing: An approach to type checking for dynamically typed languages*. PhD thesis, Rice University, 1991.

[2] C.A.R. Hoare. Hints on programming language design. In *DTIC document*, 1973.

[3] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *ICOP*, 1984.

[4] Didier Rémy. Typechecking records and variants in a natural extension of ml. In *POPL*, pages 77 – 88, 1989.

[3] A *recursively enumerable* set is a set that can be computed by a (partial) Turing machine. A *recursive* set can be computed by a total Turing machine. The difference is roughly "computable" vs. "decidable".