Disclaimer. These are preliminary note taken during and after Martin Steffen's presentation on Liquid Types at the reading club's meeting on Friday Nov. 9, 2018. Note that I am missing many of the things that Martin covered (and have added some things that were not covered). The goal was not accuracy or precision, but usefulness. Usefulness, however, is in the eye of the beholder. I find it useful to be collecting these notes and I hope they may be useful to some who read it. – Daniel F.

Type systems

The type system is usually given in operational semantics. The **progress and preservation** theorems are usually proved on the type systems' operational semantics. Note, *preservation* may also be referred to as *subject reduction.*

The operational semantic rules, however, can (often) be applied non-deterministically. In order to turn them into an algorithm, choices (restrictions) must be made.

It is important to prove a correspondence between the type system (spec) and the typing algorithm (implementation).

```
 Type system     ----------->   Typing algorithm
(specification)  <-----------   (implementation)
```

Parametric polymorphism

There are several flavors of polymorphism, for example, parametric, ad-hoc, and subtype polymorphisms. *Parametric polymorphism allows a single piece of code to be typed "generically," using variables in place of actual types, and then instantiate them with particular types as needed* (ToPL).

Starting with typed lambda-calculus, we can add parametric polymorphism in different ways. We will contrast the Hindley-Milner type system with System F.

- Typed lambda calculus, first order
- Hindley-Milner (HM) type system, "1.5" order
- System F, second order

Type inference is undecidable in System F. We can thing of Hindley-Milner as a pragmatic restriction to System F so that type inference is decidable and efficient.

Parametric polymorphism and compositionality. Note that parametric polymorphism leads to a certain flavor or compositionally in the type system. For example, say a program has two parts, `A` and `B`. We could show that part `A` and `B` work together by analyzing them together. Or, we could say that part `A` works

for all `X`, and that part `B` works on all `Y`. This makes the problem "harder" in the sense that proving "for all" is usually harder the specializing and proving for a particular instance. On the up-side, by making the problem harder, we can analyze `A` (and prove that it works for all `X`) in the absence of `B`. Similarly, we can analyze `B` without `A`.

Simply typed lambda calculus

Unification

Unification algorithm is due to Robinson, 1971 (robinson1971computational). Algorithm terminates and finds the **most general unifier**, if there exists one, else it returns a failure. The most general unifier is unique up to renaming.

Church vs. Curry style

Curry-style type systems: *we first define the terms, then define a semantics showing how they behave, then give a type system that reject some terms whose behavior we don't like. Semantics is prior to typing* (ToPL).

Church-style type systems: *define terms, then identify the well-typed terms, then give semantics just to these. Typing is prior to semantics* (ToPL).

Hindley-Milner (HM) type system

Allows for parametric polymorphism.

```
- fun id x = x;
val id = fn : 'a -> 'a
```

Parametric polymorphism is powerful. Imagine trying to implement `id` above in `C`, for example. Could write an `id` for each basic type, but wouldn't work on structs, for example. Could pass an address to '`id` and return the address, but would loose type information (address of what?).

HM is the type system for ML. We can thing of Hindley-Milner as a pragmatic restriction to System F so that type inference is decidable and efficient – type inference is undecidable in System F. HM restricts universal quantification to the top level, for example, the type of `id`

$$\lambda X.\ \ \lambda x : X.\ \ x \quad : \quad \forall X.\ X \to X$$

For an example of function not typable in HM, take the successor function for the Church numerals encoded in System F. Recall the Church numerals, $\lambda s.\ \lambda z.\ s(...s(z)...)$. They can be encoded in System F as:

$$\lambda X.\ \ \lambda s : X.\ \ \lambda z : X.\ \ s(...s(z)...z)) \quad : \quad \forall X.\ X \to X$$

Let $\mathsf{CNat} = \forall X.\ X \to X$. So far so good; the type of `CNat` has one top level quantifier. How about the type of `csucc`, the successor function?

$$\begin{aligned} \texttt{csucc} \ : \ & \mathsf{CNat} \to \mathsf{CNat} \\ & (\forall X.\ X \to X) \to (\forall X.\ X \to X) \end{aligned}$$

The type of `csucc` has a quantifier buried on the right-hand-size of an implication. This does not fit into HM.

Given the restriction that HM imposes (i.e. top level universal quantification), functions can be polymorphic, like `id`. However, to stay on the safe side, it is not allowed to pass polymorphic functions as arguments:

```
- map id;
stdIn:2.1-2.7 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 list -> ?.X1 list
```

Note that the following works because `id` is no longer polymorphic given the list of ints:

```
- map id [1,2,3];
val it = [1,2,3] : int list
```

System F

A new form of abstraction, $\lambda X.t$, whose parameter is a type, and a new form of applications, $t[T]$, in which the argument is a type expression (ToPL). On the liquid types paper, the syntax is slightly different: $[\Lambda\alpha]e$ and $[\tau]e$ (rondon2008liquid).

Sometimes called *second order lambda-calculus*, because it corresponds, via the Curry-Howard isomorphism, to second-order intuitionistic logic, which allows quantification not only over individuals [terms], but also over predicates [types] (ToPL).

Predicative vs. Impredicative systems. Something is impredicative if it invokes (mentions or quantifies over) the set being defined, or (more commonly) another set that contains the thing being defined (wiki).

The polymorphism of System F is often called impredicative. For example, in System F, the type variable X in the type $T = \forall X.\ X \to X$ ranges over all types, including T itself (ToPL).

With negation, it is easy to create paradoxes in impredicative systems; for example:

- the set of all sets that don't contain themselves (Russell's paradox)
- the barber shaves everyone who doesn't shave themselves (barber paradox)

Type systems don't typically have negation, and System F is consistent (no paradoxes). However, pitfalls still exist. Most notably, Wells showed in 1994 that type inference for System F is undecidable (wells1994typability).

It is undecidable whether, given a closed term `m` of the untyped lambda-calculus, there is some well-typed term `t` in System F such that `erase(t)=m`.

Dependent types

A mechanism for enhancing the expressivity of type systems. Such a system can express:

```
i :: { v : int | 1 <= v /\ v <= 99 }
```

which is the usual type of `in` together with a *refinement* stating that the run-time value of `i` is always an integer between `1` and `99` (rondon2008liquid).

Question (from Bjørnar). Consider the function `f x y = x * y` and the dependent type `f :: a -> b -> v : int | v <= 0`. One of `a` and `b` must be negative in order for the function to return a negative value (but not both `a` and `b`). How can we type this term? What would the most general type be?

1. `a <= 0` and `b >=0`, or
2. `a >= 0` and `b <= 0`

References

- (Robinson, 1971) Computational Logic: The Unification Computation, Machine Intelligence 6:63-72, Edinburgh University Press, 1971.
- rondon2008liquid
- ToPL
- wells1994typability