1.5.1 Local type inference (and bidirectional type systems) Pierce and Turner [222]

General overview

The paper is widely cited and influential, also and in particular in connection with type-based *synthesis* of programs. There are two ideas for which the paper is widely cited

1. *bidirectional* type checking, and
2. local type inference from the title

Local type inference seems also to be used in, for instance, Scala. It seems to me (as speculation) that both contributions are kind of independent and valuable in themselves. It could also be that both need to go hand in hand, so that "local type inference" makes only sense in connection with bidirectional formulation of a type system, but I discuss it as if it is independent.

Local type inference The core here is about type *inference,* i.e., type reconstruction or type synthesis. It is done here in connection with system F (more precisely system F with subtyping). It is well-known that type inference for F is *undecidable* (with or without subtyping) and the well-known HM type inference makes the following compromise: it restricts the polymoprhism to deal with *predicative* polymoprhism only, i.e., higher-order functions are ok, polymorphic functions are ok, but not polymorphic functions are arguments. That is captured by allowing only *prefix-quantified* types (also known as type-schemes).

The local type inference makes a different compromise but before we come to that, let's discuss more generally the problem of type inference in the face of polymorphism: polymorphism in programming languages means the fact that one program, when well-typed, can have more that one type (otherwise the typing discipline is called *monomorphic*). As far as type inference or type reconstruction is concerned, polymorphism is problematic insfar:

> if there is more than one type possible, *which one* should be inferred?

After all, type inference comes up with *one* answer only, if there is one, not enumerate all possible.[67] That is also algorithmically desirable: type systems are often (de-)compositional or syntax-directed, so they traverse the structure of the program. If the analysis of the sub-parts gives different possible solutions which need to be combined to different possible overall solutions, then that leads to a combinatorial explosion resp., when employing recursion, *backtracking.* So, it's favorable, if the algorithm can make a specific choice which is considered "best". The notion of "better" implies there is an *order* on the solution to compare their "quality", in the ideal case a form of lattice,[68] which allows to speak of *the* best solution, as opposed to "a" good one. Generally, there are *two* main ways to order types, which correspond to the two main flavors of polymorphism.[69] The first ordering is "matching-order" (say $\lesssim$): a specific type is a *substitution instance* of a more general type, having some of it's variables replaced by something more specific. This matching order (and unification, that goes with it) underlies the HM-style type inference. The alternative is "subset-based" generalization (which underlies subtype polymorphism or refinement or inclusion polymorphism ...) Local type inference here is *not based on the matching order* $\lesssim$, which also entails that there is no *unification* involved as in HM-style inference. Instead it is based on subtyping <:.

Coming back to the question of polymorphism: as mentioned, polymorphism means a piece of program may have *more* than one type. Basically, the trick of (non-ad-hoc) polymorphic type systems is to structure the "universe" of types in such a way, that there *is* actually one type that captures *all*

[67] We come back to that point later, however.

[68] Related is also the notion of Moore-family.

[69] According to the seminal classification by Cardelli and Wegner [53], there are 2 more forms (overloading and coercions), but both are "ad-hoc".

the types the piece of program can concretely have. That's either done by introducing type variables (and universal quantification perhaps), or else ordering the types via inclusion. It should also be noted: In case of unification based type inference, the unique best type the type inference is after and that captures all others (via substitution), is *the most general type.*[70] In contrast, for subtype-based polymorphism, the best type that captures all others (via subsumption) is *the most specific type.*

When talking about subtpying for functional types $T_1 \to T_2$, there is one problem namely the fact that *there might not be a best or smallest type.* The culprit is the fact that subtyping works contra-variantly on the domain-type T_1 (not covariantly, which would be fine).[71]

How does local type inference deal with that, and why is it called local? The perspective of local type inference can be best seen considering one specific situation, where a missing type has to be inferred. That's *argument type synthesis* and is coverered as one important inference situation in the first part of Pierce and Turner [222]. So, given a polymorphic function f applied to an argument e, what's the "missing" type argument, i.e., if one would use explicit syntax, where the type is given as explicit argument, what's the T in completing $f\ T$ to

$$f\ T\ e\ . \tag{1.30}$$

Seen differently: if f is of type $\forall X.T_1 \to T_2$, which T to choose to specialize the universal quantification to get the "best possible"

$$[T/X](T_1 \to T_2)$$

As mentioned, there might not be a best possible such choice for T, and the question is: what to do about it? The answer is two-fold: don't look for a globally best answer, look for the best answer in the given situation. This means, in a application situation $\ldots f\ e \ldots$, take the call-site into account. Unlike in HM-style type inference, howewer, the perspective is *local*: when analysing this particular call, take only *this* situation into account, not all. In HM-style type inference, of course also the local situations is taken into account, however, the analysis is global, as *unification* tries to reconcile all usages of a polymorphic function. That explains why it's called *local* type inference. Remains the question of how actually to take the local situation into account to determine the (locally) best choice for T. Actually, the answer is rather simple

Take the one that, for all possible choices of T gives the **minimal return type.**

As mentioned, for functional types $T_1 \to T_2$, it's in general not possible to minimize both at the same type, so the choice made here is to go for the strongest *postcondition.* Now, of course, it may be the case that T_2 is a functional type itself, and may therefore itself suffer from co/contra-variant behavior. In that case, there is simply no best choice, and local type inference gives up, i.e., fails.

A simplified[72] version of the type argument synthesis problem is given in the following "specification".

$$\frac{\begin{array}{c}\Gamma \vdash f : \forall \vec{X}.\vec{T} \to R \\ \Gamma \vdash \vec{e} : \vec{S} \qquad \vdash \vec{S} <: \theta\vec{T} \\ \forall \theta'\ (\vec{S} <: \theta'T \quad \text{implies} \quad \theta R\ <:\ \theta' R)\end{array}}{\Gamma\ \vdash f(\vec{e}) : \theta R}\ \textsc{App-InfSpec}$$

---

[70] Typically, universally quantified types are for (polymorphic) functions.

[71] No such contra-covariants issue exist in connection with the matching order. In connection with issue of contra-/covariance: it may be compared with the problem of finding the "best" pre/postcondition. For a given post-condition, there may be the best, i.e., weakest precondition, and conversely: for a given precondition, there exists a strongest post-condition. But there is not generally best pair of pre- and post-conditions. That is also related to the fact that finding loop-invariants is tricky ...

[72] Compared to [222], some extra premise forbidding $\vec{X}$ to be empty is left out. Also the "transformed" result of the type inference is left out, as it's obvious.

One has to find a substitution, i.e., a choice for the type variables $\vec{X}$), that 1) is correct wrt. the "input", i.e. $\vdash \vec{S} <: \theta\vec{T}$ and 2) from all those that satisfy that correctness condition, the one that *minimizes* the resulting return type θR. The above rule is the "specification" of the type argument synthesis problem. To turn it into an algorithm is actually pretty simple.

First one considers the requirement $\vdash \vec{S} <: \vec{T}$ as a *constraint*, describing all substitutions θ such that $\vdash \vec{S} <: \theta\vec{T}$. "Considering" it as constraint is not enough, one needs to "solve" the constraints as well. Actually, in a second step one needs to solve them in a way that minimizes θR, but that comes later. Before coming to solve them in a minimal way, we transform the constraint $\vdash \vec{S} <: \vec{T}$ in a way that makes the restrictions on the allowed substitutions "explicit". Turning implicit constraints (like $\vdash \vec{S} <: \vec{T}$ here) into an *explicit* equivalent form can be seen as *solving* the constraint set. The paper does not call it "solving", but that's how one can see it: given a set of constraints which is a set of conditions involving variables, solving constraints involves isolating the variables (here $\vec{X}$) and spelling out explicitly the conditions that they have to satisfy. For instance like $\vec{X} = \vec{\varphi}$, where in particular the $\vec{\varphi}$ should not mention $\vec{X}$, otherwise the constraints cannot be called "solved", as the variables are not "isolated". In the setting here, the solved form of the constraint system is not a set of equations, but a set of *intervals* of the form $T_1 <: X_i <: T_2$, spelling out for each variable X_i its allowed range.[73]

With the constraints $\vdash \vec{S} <: \vec{T}$ in solved interval form (say $C_{\vec{S}<:\vec{T}}$), the second part of the problem, minimizing θR, is pretty simple, as well: we need a substitution $\theta \vDash C_{\vec{S}<:\vec{T}}$ that makes θR minimal (if it exits). As it turns out: the problem is simple, basically because the chosen values for each X are *independent:* one can choose the best value for each variable indepdently of the other choices, and even more so, one can restrict oneself to either the lower bound or else the upper bound of the interval range of each variable. And the best of all: one does not have to try both so see which is better: if the variable occurs positively, pick the upper bound, if it occurs negatively, pick the lower bound, and that's it.[74]

Bidirectional type inference The presentation up-to here has tackled *only* the argument-type-synthesis part of type inference, and also the other part the paper is known for —bidirectional type checking— has not been mentioned so far. Actually, the general idea of bidirectional typing rules is rather simple. Derivation rules, like the one descrribing type inference problems, can be "read" in a goal-directed manner: in order to obtain the derivation in the conclusion of the logical rule, a procedure has to recursively solve the premises. That works smoothly (i.e., without guessing, backtracking, combinatorial explision), as long as the rules are *syntax-directed.* Syntax-dircted means, 1) the premise deals with *smaller* problems[75] and 2), all meta-variables mentioned in the premise are actually mentioned in the conclusion (in a way, there are no "free meta-variables" in the premise, all are "bound" by being mentioned in the conclusion). The latter point avoids *guessing.* As far as type inference is concerned, the prototypical situation is *abstraction*, whose type can be specified as

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e{:}\, T_1 \to T_2}$$

The question, whether the rule is syntax-directed or not depdends on what one considers as *input* and what as *output.* If it's seen as boolean procedure (type checking), then the rule is syntax-directed; in particular, T_1 is mentioned in the conclusion as the type of the arguments. For "type inference" one is interested in the problem $\Gamma \vdash e : ?$ and the part $T_1 \to T_2$ is intended as *output.* With that in mind, T_1 is not introduced in the conclusion, and in this interpetation the rule is not syntax-directed. In HM-style type infence (or other settings as well), they way to avoid guessing in a situation like that is to use a *fresh variable.*[76] Bidirectional checking takes a different route: it operates not with simply

[73] In the paper, turning the $\vdash \vec{S} <: \vec{T}$ into this explict interval form is not called solving them, the process is called constraint generation, and it's pretty easy actually here.

[74] There are corner cases where the variable occurs positively and negatively or not at all, but the idea is the same.

[75] In practice it mostly means: to obtain the type of a program, one has to determine the types of sub-expressions first.

[76] To be fresh basically avoids guessing, because any choice is as good as any other. So one can get away with a fresh-variable generator that determinstaclly gives the next unused one.

one jugement $\Gamma \vdash e : T$, but two: inference (or synthesis) on the one hand, and *checking* on the other, written

$$\Gamma \vdash e :\uparrow T \quad \text{and} \quad \Gamma \vdash e :\downarrow T$$

In the situation of abstraction of the form $\lambda x.e$ (with the type of x missing), that would be covered by a checking rule. It's called bidirectional checking, as, depdenting on the judgment, one can interpret it in such a way, that the type information "attached" to the nodes of the syntax tree propagates in a bottom-up as well as a top-down manner. That is in contrast to standard type inference system where the flow of information is bottom up only (or type checking, which can be interpeted as top-down).