Type inference The paper deals with the problem *type inference*, sometimes alternatively called *type reconstruction.* It does so in a rather complicated setting, as far as the types are concerned, in particular, covering *dependent types.* Type inference, in general is the following problem; given a program, figure out ("reconstruct" or "infer") type information. That is the *inverse problem* of type-directed program *synthesis*, where the missing program has to be found starting from the types as the programs specification[44]

The most well-known type inference algorithm is the one by Hindley-Milner-Damas (also knows as HM, or type inference for ML-style polymorphism). ML-style polymorphism or let-polymorphism refers to a form of polymorphism known as *parametric* polymorphism.[45] This form of type inference (based on term unification) is the bread-and-butter of type inference, it's almost synonymous with "standard type inference". It's therefore not surprising that many attempts have been made to *generalize* the underlying ideas (for instance to accommodate to another form of polymorphism like *subtype* polymorphism or to go make the types more expressive in other ways).[46] This paper here does both: covering *dependent* types + "subtyping". Subtyping in the context of the more expressive types becomes more of a flavor of "logical implication", and actually is not called *subtyping* here but *refinement.*

Dependent types, refinement types, and liquid types Refinement types are a form of dependent types (with "subtyping") and the *liquid* types and the corresponding type system is an well-known approach to allow HM-style type inference in a setting with such dependent types. BTW: Liquid is an abbreviation for *logically qualified types.* As indicated, type inference already in connection with parametric polymorphism and extensions is tricky business insofar one balances on the edge of undecidability. Liquid types are therefore a carefully tailor-made design of restrictions on dependent and refinement types[47] so to still remain within the bounds of decidable type inference.

The "carefully tailor-made restrictions" involve which form of logical specification or predicates to use (i.e., how to restrict the expressivity of the dependent types), where to use them, and also where allow *quantification.* The latter point seems in particular interesting: it's clear that quantifier-free specifications are less expressive and more managable than those that quantified ones; one often distinguishes between quantifier-free theories and solvers and those allowing quantification. Here, there's universal quantification is allowed (as inherited from the (predicative) let-polymorphism), the logical part, the *qualifiers* are *quantifier-free*! One may see it like that: the burden of dealing with universal quantification is relegated and dealt with by the HM-style type inference system. So when verifying the progam via liquid type inference, one uses classical HM inference to get *"theorems for free"*.[48] As mentioned, the rest of the specification, notably the "logical formulas" are quantifier-free. Moreover, they are basically a simple form of conjunctive constraints, forming the value domains at the *base* types (and more complex types are build up from there, without the possiblity of "refinement

[44]The synthesis problem, in that view, is reduced to type inhabitation problem and, in the spirit of the Curry-Howard isomorphism, as proof search. That is done for instance in the last paper Frankle et al. [110] in a similar setting.

[45]On the following, I use "polymorphism" to mean parametric polymorphism. when talking about subtype polymorphism, it's explictly stated. One way of seeing it is: polymorphism is reflected on the type level as universal quantification over type variables and the expressivity of the ML type system does not allow arbitrary universal quantification, but *prefix quantification, only.* It's thereby a fragment of the more general "polymorphic λ calculus" PLC also known as "System F". It's known that type inference for System F is undecidable, so the ML system seem to hit some sweet spot as it's useful in practice but still allows decidable type inference. As a further side remark: the quantification in System F corresponds to a second order quantification! The qantifier in a type $\forall\alpha.\tau$ ranges over *all* types, which is a second-order notion (like quantifying over sets of things). Even more: the type $\forall\alpha.\tau$ *is itself a type*, meaning that this form of quantifier antifies over "itself". This form of self-referentiality is known as *impredicativity* an is the souce of the strength of system F. ML-style polymorphism shies away from that: restricting to prefix quantification fundamentally reduces expressiveness. The form of polymorphism is also known as *predicative.*

[46]Another rather well-known general approach to find a workable marriage of HM-style polymorphism with a form of subtyping is known as HM(X). It's a "class" of type inference settings, like HM-modulo-X. How HM(X) and the paper here (and others about refinement type inferance) technically hang together is unclear to me.

[47]The problem of type *checking* in the presence of refinement types has been studied as well.

[48]The slogan "theorems-for-free" refers to a well-known paper by Wadler [293] elaborating on the fact that properties of universally qantified polymorphism functions apply to *all* specializations of the function.

forming" $\{v : \tau \mid \varphi(v)\}$ by "qualifying" allowed values v on *arbitrary* types τ.

Liquid type system specification and type inference algorithm The technical development of the type inference algorithm follows a route common to many, basically all more advanced type inference systems.

The starting point is the type system which one may call as the *specification* of the problem: it defines the type relation (typically in a rule-based manner) without bothering about the question, whether a corresponding type inference is possible, Often (as here), the specification is moderately straightforward. It deals with jugements often generally of a form like

$$\Gamma \vdash e : \tau$$

which is a triple of contexts (or type environments), programs or expression, and types. If one assume that all 3 parts are given and if one assumes type *checking* to be the problem of determining whether the question $\Gamma \vdash e : \tau$? is true or not, then type checking typically is straightforward. To say it differently, the rules directly specify a type ckecking algorithm, provided type checking is interpreted as the question as indicated.[49] Technically, the rules of the type system are (often) *syntax-directed*, here not quite due to subsumption.[50] Anyway, a *key* ingredient in turning the rules of the system into a (syntax-directed) algorithm (where τ is *not* part of the input) is the use of *type-level variabes* and (hand in hand with that) the use of corresponding *constraints* on said variables, That leads to a two-phase design. In the first phase, the "type checking", the syntax-directed rules traverse the syntax tree in a recursive way, i.e., in the standard manner of type system. Type-level variables are used since, while traversing the systax tree, certain facts cannot yet be determined and there determinization have therefore to be *postponed*: so while traversing in phase one, each time one some type or specification cannot yet be determined, one introduces a variable and remembers a corresponding constraint. Typically one uses *fresh* variables when encountering as "postponement" situations and the algorithm takes the form of

$$\Gamma \vdash e \rhd (\hat{\tau}, C) \tag{1.21}$$

where Γ, e is the *input* and the tuple $\hat{\tau}, C$ the output.[51] In the above "judgement", I use $\hat{\tau}$ to indicate, that now the types are often extended insofar in that one introduces *type-level* variables in order to express that certain pieces are still unknown and subject to constraints C.[52] In many presentations of the classical HM type inference (whose solution is known as algorithm W), the type inference does *not* operate with judgements of the form (1.21). Instead of makeing a two-phase approach —first harvest constraints, then solving them— many presentations solve the corresponding constraints *eagerly* on the fly (using term unification). That means, the algo operates on judgments

$$\Gamma \vdash e \rhd (\hat{\tau}, \theta)$$

instead, where θ is a substitution. More precisely, it's the most general unifier of the term unification constraints C in the judgement of (1.21).

[49]Note that the question $\Gamma \vdash e : \tau$? does not correspond to what one expects from a *type checker* as a part of a compiler, one often wants $\Gamma \vdash e :$?

[50]The paper [254] explicitly claims that the rules of the type system specification are syntax-directed. That claim, however, is *wrong* insofar that the subsumption rule LT-Sub is *not* syntax directed. Subsumption is actually the prototypical non-syntax directed rule in systems which cover "subtyping". That's even the case when we consider the type checking problem as $\Gamma \vdash e : \tau$?. As a side remark: in connection with parametric polymorphism, also the two rules in connection with $\forall\alpha.\tau$ (generalization and instantiation, $\forall$-introduction and $\forall$-elimination) may break syntax-directedness, provided we use a non-explict term-syntax, i.e., specializing a polymorphic function e is not written $[\tau]\ e$ as in this paper (or $e\ \tau$ more conventionally), and the formation of a polymorphic function is not explicitly done by writing $[\Lambda\alpha.]e$.

[51]The paper write the type inference in pseudo-code, not as rules. but that's a matter of presentation. Here I wrote $\rhd$ to separate input from output (instead of the ":" as separator).

[52]Note that in HM (but depending on what one takes as original starting point), the original types already had type-level variables (namely type variables α to express polymorphism. But as said, that depends a but what want takes as speficiation starting point there.

When going to more complex type inference territory (as here), the “one-phase-approach” that solve constraints on the fly often no longer works. In a way: standard type inference uses most general unifiers θ as the*solution* of the constraints during the algorithm. It specific to that setting that the *solutions* behave in a compositional or syntax-directedmanner: “give me the mgu’s of the subterms and I compute the mgu of the composed term”. In more complex settings, such “compositionality” of the *solutions* of the constraints in that form. Instead, one joins and collects the constraints as such (not their solutions) and relegates theirs solution to a second phase. This *frees* the “traversal” of the constrainsts, i.e., the strategy in which way they are treated and solve, from the transversal strategy of going through the syntax (which is typically fixed by the syntax directed typing rule). This decoupling is necessary in more complex setting as here. As simple example of the need of such a decouply are *flow type systems;* as additional information in the type system is flow information (as in data flow). Conventionally, solving them requires fix-point iteration. It may be intuitively plausible that this corresponds to “going in circles” in the syntax tree,[53] and that’s something one typically does not do in typing rules.

Now, as far as *type-level* variables for liquid typing go: here we are dealing with standard type variables α for polymorphic functions as well as *“liquid” variables* κ, representing unknown refinements (= specific dependent types), resp. refinements who will be soved later.

With all this machinery (type inference, HM etc), the core of the paper is actually not so suprising: the paper is very careful that the “logical part” (the dependent typing part in the form of the restructed refinement types, i.e., liquid types) is *decidable.* Concretely that means formulas similar what can be found in non-type based settings, when it comes to *invariant generation.* Note that In the context of recursion, the problem for refinement type inference correspond to *loop-invariant* generation. For example the restrinction deals with booleans and linear arithmethic, and conjunctions of constraints over such restricted formulas. Seen like that the set-up is also rather conventional. What is specific or challenging though might be the ability to deal with *higher-order* functions (and also to be able to tackle polymorphic functions to some extent).

[53]Maybe not in directly in the syntax tree but officially in the control-flow graph, but indiectly it means moving up and down iteratively in the syntax tree.