*>*

*>*

*> p.2, "In the early stages, Java < 1.5...". Milner introduced polymorphism in*

*> 1978, so it is somewhat curious to pick Java as a starting point.*

*> Furthermore, I have personally verified that Java type inference*

*> can take exponential time, so it is not the best example of type*

*> inference done right.*

*>*

*> p.3, "an accidentally misspelled variable name might cause a runtime error*

*> until the line of code is actually executed". Unclear. Missing negation?*

missing negation: "... does not cause a runtime error until ..."

*>*

*> p.4, [Pierce 2002] is not the original citation for simply-typed*

*> lambda-calculus. [Reynolds 1983] is not the original citation for System F,*

*> which I think was independently invented by Girard and Reynolds around*

*> 1972-1973. [Reynolds 1983] is not the original citation for*

*> implicit parametric polymorphism, which was invented by Milner*

*> and presented in his 1978 paper.*

*>*

*> p.13, "Hindley [Hindley 1969] and Milner [Milner 1978] independently*

*> discovered equivalent algorithms". I don't think that is quite accurate.*

*> Hindley studied ordinary lambda-terms (without a "let" construct) and*

*> proved that they have principal type schemes, whereas Milner added the*

*> polymorphic "let" construct and proved that, in its presence, programs*

*> still have principal type schemes.*

*>*

*> p.16, it would be desirable to clarify whether Theorems 2.1 and 2.2 are*

*> theorems that are proven in this thesis (using Abella), or quoted from*

*> the literature. I suppose the latter, since Algorithm W is not defined*

*> here.*

Added a note regarding this

*>*

*> Figure 2.5, it should be noted (in the text) that the problem of deciding*

*> subtyping in OL's type system can be reduced to first-order unification under*

*> a mixed prefix (that is, with an alternation of universal and existential*

*> quantifiers). As far as I know, this is folklore knowledge, and has been for a*

*> long time, perhaps as early as OL's original paper. The text on page 29 seems*

*> to suggest that DK are the inventors of the first decision algorithm for this*

*> subtyping relation; this is certainly not the case. Also, on page 32, "the*

*> ordered algorithmic context, inspired by Dunfield and Krishnaswami" can be*

*> understood as a mixed quantifier prefix; and a "worklist judgement" is just*

*> a unification constraint, beginning with a mixed quantifier prefix,*

*> followed with a conjunction of equations. A connection should be made*

*> with a constraint-based presentation of first-order unification under*

*> a mixed prefix.*

*>*

*> Section 2.2.3 is relatively unclear and could perhaps be improved. My personal*

*> suggestion would be to explain that both type inference for HM and type*

*> inference for OL can be reduced to solving a certain form of constraints,*

*> where constraints include equalities between types, conjunctions, ∀ and ∃*

*> quantifiers, and "let" constraints.*

*>*

*> p.21, DK's subtyping relation is exactly the same as OL's, right? This should*

*> be stated more clearly (and the explanation of the subtyping relation on p.22*

*> could be removed if it is redundant, or could be moved to the point where OL's*

*> subtyping relation is described).*

*>*

*> Figure 2.8, it is not entirely clear what the metavariable τ stands for,*

*> since there is only one category of types (declarative types, A, B).*

*> Please clarify.*

The syntactic category \tau is mentioned before in Figure 2.7.

*>*

*> p.22, "Allowing an arbitrary type (rather than a monotype) in rule ≤∀L is*

*> known to give rise to an undecidable subtyping relation [Tiuryn and Urzyczyn*

*> 1996]". The citation of Tiuryn and Urzyczyn here is relevant but is not quite*

*> accurate, as the subtyping relation considered by Tiuryn and Urzyczyn includes*

*> the rule Distrib, which allows exchanging ∀ and →, and is absent in OL's*

*> subtyping relation. An undecidability result in the absence of Distrib has*

*> been published by Chrzaszcz in MFCS (1998).*

[notes: the Distr rule is similar to deep-skolemisation]

Changed the citation to Chrzaszcz's

*>*

*> Figure 2.9, rule Decl→I⇒, it is not clear what σ and τ stand for, especially*

*> since the text says "this type system only synthesizes monotypes σ → τ", but*

*> the metavariable σ has been used earlier to denote a polytype. Please clarify.*

TODO change the notation a bit across these chapters;

but this syntax is indeed introduced in Figure 2.7 before.

*>*

*> p.25, "this type is equivalent to the original type with the constraint*

*> [...]". More accurately, it is not really an equivalence between types, but an*

*> equivalence between constrained type schemes.*

rephrased a bit, talk about constrained type schemes

*>*

*> In Section 2.4, it may be useful to announce what is the connection (if any)*

*> between DK and MLSub. Otherwise, the reader may wonder why MLSub is presented*

*> here.*

*>*

*> Figure 3.4, rule ≤ainstL, you might wish to explain why you introduce two*

*> auxiliary variables α1 and α2, and replace α with α1 → α2, instead of*

*> replacing α directly with A → B. I imagine that the purpose is to avoid the*

*> exponential explosion that could be caused by naive substitutions. So, it is a*

*> good thing to do, and makes your description closer to an actual efficient*

*> implementation. That said, your description still uses an eager substitution,*

*> so it would still be inefficient if it were implemented literally. (Its cost*

*> would not be exponential, but quadratic, I imagine, due to repeatedly applying*

*> substitutions to Ω.) An efficient, quasi-linear-time implementation would rely*

*> on a union-find data structure. Do you describe it somewhere? Do you give a*

*> complexity statement in the dissertation? (It would be worth discussing also*

*> whether the algorithm can be executed inside Abella. If that is the case, then*

*> the union-find algorithm would remain implicit, as it would be provided by*

*> Abella's execution engine, I suppose.)*

*>*

*> p.40, "Our development is only based on the reasoning logic of Abella, and*

*> does not make use of its specification logic." It would be desirable to recall*

*> what this distinction means.*

*>*

*> p.40, it would be worth explaining that Abella's ":=" apparently should be*

*> read as a Prolog ":-", that is, a right-to-left implication. (Correct?)*

*>*

*> p.40, it would be worth stressing that "Define env..." does not define "env"*

*> as a new type; instead it defines "env" as predicate on olists, and recognizes*

*> valid environments Ψ among them. (Correct?) It would be worth clearly stating*

*> that the type olist is a built-in type in Abella (correct?).*

*>*

*> p.41, "Exp" corresponds to Ω, right? This is not obvious at first.*

*>*

*> p.41, "exvar" is used in the definition of "subal", but hasn't been explained.*

*> If I understand correctly, whereas a context Ψ contains only "bound" items, a*

*> context Γ contains both "bound" and "exvar" items. (Correct?)*

*>*

*> p.41, "we would reduce significant lines of code if Abella provided more handy*

*> tactics". Did you consider extending Abella with a way of defining tactics?*

TODO Abella clarifications

*>*

*> Figure 4.1: the metavariable Ψ has been defined in Figure 3.1 already, with a*

*> different meaning.*

*>*

*> p.45, "In such a case, choosing ≤∀R first is always better." Indeed, that is*

*> fortunate. However, I seem to recall that, if one extends OL with existential*

*> types, then things become less clear-cut; there are situations where it is not*

*> clear which rule should be applied first. Have you looked into this?*

TODO think about this

*>*

*> p.46, "However, we argue that more specific rules are always the best*

*> choices". I suppose that this is proved as part of the completeness proof of*

*> the algorithmic system with respect to the declarative system?*

TODO clarify this

*>*

*> p.47-48, the need for scope markers could be better explained. To me, they*

*> look unnecessary: the point where α is bound is clearly indicated in the*

*> context, either by an entry of the form "α", or by an entry of the form "α =*

*> τ" (after α has been solved). My guess is that maybe DK's rules include a*

*> rule that discards entries of the form "α = τ", and this is why keeping an*

*> explicit marker is required. Am I right?*

TODO clarify this: I think this is true; TO BE DISCUSSED

*>*

*> p.48-49, giving precise counter-examples is good, but could you also give*

*> some intuition as to what the Parallel Admissibility lemma means and why*

*> it does not hold?*

TODO ask Ningning perhaps

*>*

*> p.50, "However they have separate phases for constraint generation and*

*> solving". Is this intended as a criticism? (The word "however" suggests so.)*

*> Constraint generation and constraint solving are separated because they can*

*> be separated, but it would of course be straightforward to describe them in*

*> a single unified system if desired, just by viewing the constraint generation*

*> rules and the constraint solving rules as a single system of rewriting rules.*

TODO good point, need to think

*>*

*> p.52, "Algorithmic judgment chain": I like this! You have a very nice syntax*

*> that allows representing a bidirectional type synthesis / type-checking*

*> problem, including its local term variables and (flexible / rigid) type*

*> variables, with precise scope information, and explaining how this problem is*

*> reduced, step by step, until it is solved. I think that this is clearly in the*

*> spirit of earlier purely constraint-based presentations of type inference such*

*> as EMLTI.*

The Essence of ML Type Inference

*>*

*> p.53, "continuation passing style": I am not sure that this terminology helps*

*> here. What does it refer to? Do you mean that the construct "e ⇒a ω" is in*

*> continuation-passing style? That's one way to think of it, but you could also*

*> write it in the form "let a = synthesized type of e in ω", and I am not sure*

*> that this form would evoke continuation-passing style.*

it is very similar, but not completely this case.

e1 e2 => \forall a. A, which cannot be encoded in types (not type schemes)

but it can be represented by another sort of variable, \epsilon,

to mimic the same strategy

*>*

*> By the way, I note that the construct "e ⇒a ω" corresponds rather closely to a*

*> "let" constraint in EMLTI: indeed, it means "synthesize a type for e and bind*

*> it to the type variable a in ω". EMLTI's "let" constraint "let x = λα.C1 in*

*> C2" allows expressing something very similar (although not quite identical),*

*> namely "synthesize the most general solution of the constraint C1 and bind it*

*> to the variable x in C2".*

*>*

*> Figure 4.5, rules 10 and 11. These rules are non-local, as they "fish" α out*

*> of the context and replace it with α1 and α2. In a more local style, one could*

*> locally introduce "∃α1, α2. (α = α1 → α2 ∧ ...)" and rely on separate*

*> rewriting rules to hoist out the existential quantifiers and the equation that*

*> describes α. (Things are done that way in EMLTI.)*

*>*

*> p.63, in constraint-based terminology, the "declarative transfer" relation of*

*> Figure 4.8 defines whether a ground constraint (one without existential*

*> variables) is true or false (with respect to the declarative type system).*

*> Thus, it defines the logical meaning of constraints. Right?*

I think that's true

*>*

*> p.65, "where the second measure is 2 for checking (1), 1 for inference (2) and*

*> 3 for application inference (3)": this is too complicated; I don't know what*

*> this sentence means.*

DONE rephrased a little bit

*>*

*> Corollary 4.10, it may be interesting to give an even simpler corollary where*

*> Γ is empty. Item 2 would then become (I think): "if e ⇐ A rewrites to · in the*

*> algorithmic system, then e ⇐ A holds in the declarative system". A similar*

*> remark concerns Corollary 4.13, where we would obtain the reverse implication.*

*> Thus, we would get a crisp final statement of equivalence between the two*

*> systems.*

TODO DISCUSS

*>*

*> p.76, "The premise shows that one judgment chain may relate to more than one*

*> elaboration elements, and that the outer judgment, being processed before*

*> inner ones, elaborates to the top element in the elaboration list". This text*

*> is somewhat unclear and could be clarified. So, if I understand correctly, the*

*> coercion f is a witness of A ≤ B, where A is the synthesized type of e, and*

*> the term t is the elaborated version of e, right? Overall, this makes sense,*

*> but I have a feeling that there must be a more readable (intuitive) way of*

*> presenting this, making it visually more obvious that the result of*

*> elaborating e is named t and that the result of elaborating A ≤ B is named f*

*> and that (in the end) the result of elaborating "e ⇒a a ≤ B" should be "f t".*

*>*

*> Although not directly relevant here, I would like to mention my ICFP 2014*

*> paper, which concerns elaboration in a Hindley-Milner setting, and tries*

*> to present it in a style that is as readable as possible.*

TO VIEW

*>*

*> p.78, "improving the rate of success guessing": is your goal really just to*

*> "improve the rate of success", or to design an algorithm that is sound and*

*> complete? Also, there are algorithms for solving (various kinds of) subtyping*

*> constraints that do not involve backtracking; could they be applicable here?*

*>*

*> p.78, "any type is more general than ⊤ and thus can be considered as an*

*> instance of type ⊤". Shouldn't this be the other way around? ⊤ is a supertype*

*> of every type.*

*>*

*> p.78, "An instance of ⊥ can be casted to a value of any type, which is usually*

*> impossible." Not clear what this means. (Also, "casted" should be "cast".)*

*> I suppose you mean that there is no value of type ⊥ so the cast instruction*

*> will never actually be executed. However, it is still possible (and useful)*

*> to use a cast from ⊥ to some other type.*

*>*

*> p.79, "Besides, subtyping comparison between constraint types is also not*

*> easy." Please clarify what you mean by "not easy". Is it costly? Undecidable?*

*> (The answer depends on what subtyping relation you are considering.)*

*>*

*> p.79, "principle" -> "principal"*

*>*

*> p.80, unification is the process of solving equalities, so "equality*

*> unification" is redundant.*

*>*

*> p.81, "If the judgment reduction succeeds, then we know that α := ⊥ is a*

*> possible solution." The question is, when can you declare success? Do you have*

*> to wait until the whole program has been type-checked? Or can you declare*

*> success earlier? In other words, when can you commit to a choice and forget*

*> about a potential backtracking point?*

wait until the whole process; not possible to forget a backtracking point,

as this is a global type inference algorithm

*>*

*> p.81, "completeness is impossible for any algorithm that reduces eagerly".*

*> Please clarify what you mean. Do you mean that a naïve procedure must examine*

*> an infinite tree of possibilities and therefore will sometimes diverge?*

Yes.

*>*

*> p.81, "complete in most cases" does not really make sense. An algorithm is*

*> complete, or it is not. One can, however, identify the places in the algorithm*

*> where completeness is lost; I suppose you mean that there are few such places.*

*>*

*> p.81, "we believe that this is a practical tradeoff". A tradeoff between what*

*> and what? Do you have any evidence to support the claim that this incomplete*

*> algorithm is practical? And what exactly do you mean by practical? Efficient*

*> enough? Able to accept most reasonable programs in some sense?*

Tradeoff between more complicated algorithms with incompleteness

No evidence yet.

A declarative specification that points this out would be much better.

But for this chapter, the algorithm is incomplete.

*>*

*> Figure 5.3, the new rule Decl⊤ looks highly suspicious. Does it really say*

*> that every (syntactically well-formed) expression has type ⊤? If so, then it*

*> seems unsound (at least under a call-by-value semantics), as the expression e*

*> could crash when evaluated. Do you have a call-by-name or call-by-need*

*> semantics in mind?*

! perhaps change the rule with:

P |- e   P |- e => A

**--------------------**

P |- e <= T

or

P |- e   P |- e <= A

**--------------------**

P |- e <= T

*>*

*> p.86, "we argue that comparing α with a function type that contains α itself*

*> is hardly useful in practice." This argument is quite surprising. One could*

*> argue that, once you have chosen a declarative type system that is equipped*

*> with a certain subtyping relation, that system becomes the specification of*

*> your type-checker, and you must adhere to it. If you don't think it is a good*

*> specification, then you should adopt some other specification; you should not*

*> keep this specification and propose an algorithm that is incomplete with*

*> respect to it. (As an anecdote, the constraint α ≤ 1 → α describes a function*

*> that is able to accept as many arguments as one wishes to pass to it. This is*

*> certainly unusual, but such a function can be defined in OCaml by "let rec f x*

*> = f", and it sometimes appears in the OCaml code that is produced by Coq's*

*> extraction process.)*

*>*

*> Figure 5.5, as far as I can see, the non-deterministic algorithm is described*

*> simply by a system of rewriting rules, where several rules are sometimes*

*> applicable. Thus, the choices that the algorithm must explore do not have a*

*> limited scope; all choice points must kept in memory, and the algorithm may*

*> have to backtrack arbitrarily far into the past. Is that the case?*

Yes, it is.

*>*

*> p.88, "A partial completeness theorem is shown under the rank-1 restriction".*

*> How can the algorithm be complete if it misses some solutions to the*

*> constraints, as described at the top of page 87? [OK, I see on page 93 that*

*> under the rank-1 hypothesis, you never encounter a goal of the form α ≤ β.]*

*>*

*> p.89, it is not clear what "its actual type" means. Do you mean the type that*

*> would be obtained in synthesis mode?*

*>*

*> p.90, "therefore Rule Decl⊤ is required". Well, not really. What is required*

*> is that \*if the expression e can be checked against some type A then\* it can*

*> be checked against the top type.*

Yes.

*>*

*> p.94, "our algorithm is complete". Considering that you have not proved the*

*> termination of our procedure, I don't think you should call it an algorithm.*

*>*

*> p.95, "The problem can be solved by introducing recursive types, so that*

*> recursive constraints can be solved". I don't understand. Without recursive*

*> types, the constraint already has an infinite number of solutions. Adding*

*> recursive types doesn't seem to simplify things at all; you still have an*

*> infinite number of finite solutions, plus new solutions, which involve*

*> recursive types.*

They might be a principal solution in addition to the existing system.

*>*

*> p.97, "When α is at the top of the worklist and thus going to be recycled, we*

*> need to further check if it can actually be solved, by ensuring that lower*

*> bounds are indeed subtypes of upper bounds: any type that is a subtype of α􏰤*

*> should be a subtype of any super type of α". What happens if α itself appears*

*> in one of the bounds?*

loop/cycle detection

*>*

*> p.98, "throughly"?*

*>*

*> p.101, "In those cases, type annotations may guide the type system".*

*> I don't understand. If the type system is restricted to predicative*

*> instantiations, then, even with the help of type annotations, an*

*> impredicative instantiation remains forbidden, doesn't it?*

This is true

*>*

*> p.101, "DK accepts even better type annotations, which also allow polymorphic*

*> types". I don't understand what this means.*

higher-ranked types

*>*

*> p.102, "Their subtyping relation is enriched with deep skolemisation". The*

*> terminology "deep skolemisation" seems to have been invented by Peyton Jones*

*> et al. (2007), but this subtyping rule is known as "distributivity" and goes*

*> back to Mitchell's presentation of System Fη (1984, "Coercion and type*

*> inference").*

indeed, similar

*>*

*> p.102, "its subtyping system is already undecidable [Tiuryn and Urzyczyn*

*> 1996]". Again, Tiuryn and Urzyczyn's result concerns System Fη, not*

*> System F. An undecidability result about System F's subtyping relation*

*> was proved by Chrzszcz ("Polymorphic subtyping withaout distributivity").*

*>*

*> p.111, "In a naive implementation, frequent iteration is required to look up*

*> the relative positions of variable declarations.". Rémy introduced the idea of*

*> "levels", that is, annotating variables with integers that indicate where they*

*> are bound in the context. This is described (not in great detail) in EMLTI,*

*> and also in a paper by Kiselyov (http://okmij.org/ftp/ML/generalization.html).*

True. use integers to track. but the problem is that insertions might change the numbering

*>*

*> p.112, "if we extend our type system by including recursive types, we would*

*> accept such judgment instead of rejecting it". As noted earlier, the constraint*

*> α ≤ A → B admits the solution α := ⊥, even if α occurs in A or B. Introducing*

*> recursive types only adds more solutions, but does not change the status of*

*> this constraint from unsatisfiable to satisfiable.*

*>*

*> The results by Su et al. ("The First-Order Theory of Subtyping Constraints")*

*> may have been relevant here.*