

Inférer un type sans exécuter

Hindley, Milner et ce que notre typeur promet

unification

polymorphisme de let

type principal

Comment un algorithme de 1978, qui tient en une page, devine le type de vos fonctions sans jamais les lancer, et pourquoi il refuse poliment d'en deviner davantage.

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Mini-typeur » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Le projet « mini-typeur » reprend le langage d'AP2 et lui ajoute un troisième moteur, après l'interpréteur et la machine à pile d'AP3 : un *typeur*, qui lit l'arbre d'un programme et annonce son type sans rien calculer. Les quatre carnets et le squelette vous font construire les règles sur `Float` et `Bool`, l'unification, le typage des fonctions, puis le polymorphisme de `let`. À la fin, `let id = fn(x) { x } let n = id(1.0) id(True)` est accepté, et `fn(f) { let n = f(1.0) f(True) }` est refusé.

Ce fascicule ne contient aucune solution du projet. Il raconte d'où viennent ces idées, ce qu'elles garantissent exactement, et ce qu'elles deviennent dans OCaml, Haskell, Rust, TypeScript ou dans le typeur qu'Elixir construit depuis 2024. Vous y trouverez un arbre de dérivation déroulé à la main, l'unification vue comme un graphe, une esquisse de la preuve de sûreté, et quelques limites que personne n'a réussi à contourner depuis quarante ans. Lisez-le un soir, après avoir fini le projet ou au milieu, quand une question vous démange.

2 Un type est une promesse

Commençons par le vocabulaire. Un *type* est une catégorie de valeurs : `Float` regroupe tous les nombres de notre langage, `Bool` ses deux booléens, et `fn(Float) -> Bool` les fonctions qui prennent un nombre et rendent un booléen. Écrire $e : \tau$, un *jugement de typage*, c'est faire une promesse sur l'expression e : quelle que soit la manière dont on l'exécute, si elle termine sans erreur, sa valeur appartient à τ . L'interpréteur d'AP2 constate une valeur après le calcul ; le typeur promet une catégorie avant.

Une promesse se tient à l'aide de règles. Le carnet 1 les présente sous forme de tableau, et c'est bien ainsi que les théoriciens les écrivent, à la notation près :

Expression	Condition sur les sous-expressions	Type
<code>8.0</code> , <code>True</code>	aucune	<code>Float</code> , <code>Bool</code>
<code>a + b</code> , <code>a - b</code> , <code>a * b</code> , <code>a / b</code>	<code>a : Float</code> et <code>b : Float</code>	<code>Float</code>
<code>a < b</code> , <code>a == b</code>	<code>a : Float</code> et <code>b : Float</code>	<code>Bool</code>
<code>case c { True -> a False -> b }</code>	<code>c : Bool</code> , puis <code>a</code> et <code>b</code> de même type	ce type commun
<code>fn(x) { corps }</code>	<code>corps : T</code> en supposant <code>x : S</code>	<code>fn(S) -> T</code>
<code>f(a)</code>	<code>f : fn(S) -> T</code> et <code>a : S</code>	<code>T</code>
<code>x</code>	<code>x : T</code> dans l'environnement	<code>T</code>

Les deux dernières lignes font intervenir l'*environnement*, le carnet des hypothèses en cours : les paramètres des fonctions englobantes et les noms liés par `let`, chacun avec son type. C'est le même rôle que l'environnement de l'interpréteur d'AP2, sauf qu'il contient des types et non des valeurs.

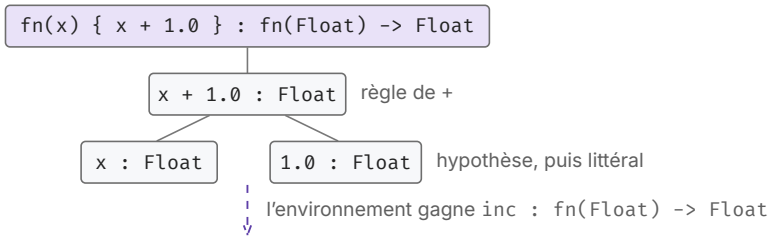
2.1 Un arbre de dérivation déroulé à la main

Appliquer les règles produit un *arbre de dérivation* : chaque nœud est un jugement, ses enfants sont les conditions que la règle exige, et les feuilles sont des littéraux ou des hypothèses lues dans l'environnement. Prenons ce programme de notre langage :

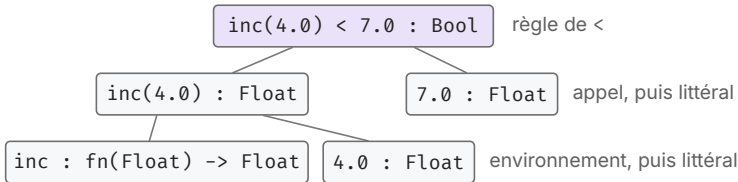
```
1 let inc = fn(x) { x + 1.0 }
2 inc(4.0) < 7.0
```

Pour typer la valeur du `let`, la règle des fonctions demande de supposer un type pour `x`, puis de typer le corps. Si nous supposons `x : Float`, la règle de `+` exige `x : Float`, ce que l'hypothèse donne, et `1.0 : Float`, ce que le littéral donne. Le corps a donc le type `Float` et la fonction le type `fn(Float) -> Float`. Le `let` range ensuite `inc` dans l'environnement avec ce type, et le corps du `let` se dérive de la même façon, comme illustré ci-dessous.

1. La valeur liée par `let`, en supposant $x : \text{Float}$



2. Le corps du `let`, avec `inc` dans l'environnement



Deux arbres de dérivation, le jugement à établir en haut et ses conditions en dessous. La flèche pointillée transporte le résultat du premier arbre dans l'environnement du second. Chaque trait est une règle du tableau.

Remarquez un détail gênant : nous avons *supposé* $x : \text{Float}$. Rien dans la règle ne dit comment choisir ce type ; nous l'avons deviné en lisant le corps. Si nous avions supposé $x : \text{Bool}$, la règle de `+` aurait échoué, et il n'existerait aucun arbre. Un typeur qui demande à l'utilisateur d'annoter chaque paramètre n'a pas à deviner : c'est le choix de Java ou de C. Tout l'intérêt de l'*inférence* est de remplacer cette devinette par un calcul, et c'est l'objet des deux sections suivantes.

SÛRETÉ : BIEN TYPÉ, DONC JAMAIS DE TYPE_MISMATCH

Si un programme possède un arbre de dérivation et si son évaluation par l'interpréteur d'AP2 termine, elle ne rend jamais `{:error, :type_mismatch}`. Pourquoi c'est vrai, en deux pas qu'ont formalisés Andrew Wright et Matthias Felleisen en 1994 : d'abord la *préservation*, chaque pas d'évaluation transforme une expression de type τ en une expression du même type τ , ce qui se vérifie règle par règle ; ensuite le *progrès*, une expression bien typée est soit une valeur, soit capable d'un pas de plus, car par ex. un `+` bien typé a forcément deux nombres sous la main. Un calcul qui conserve son type et ne se bloque jamais sur une opération absurde ne rencontre pas d'erreur de type.

3 Deviner sans deviner : les inconnues et l'unification

L'idée qui remplace la devinette est d'une grande simplicité : quand on ne connaît pas un type, on lui donne un nom. Le paramètre `x` reçoit une *variable de type*, une inconnue que le projet note `{:var, 0}` et affiche `t0`. On continue ensuite à appliquer les règles, et chaque règle qui exige quelque chose d'une inconnue produit une *contrainte*, une équation entre deux types. Dans `x + 1.0`, la règle de `+` exige `t0 = Float`. À la fin, il reste à résoudre toutes les équations ensemble.

3.1 Résoudre une équation entre types

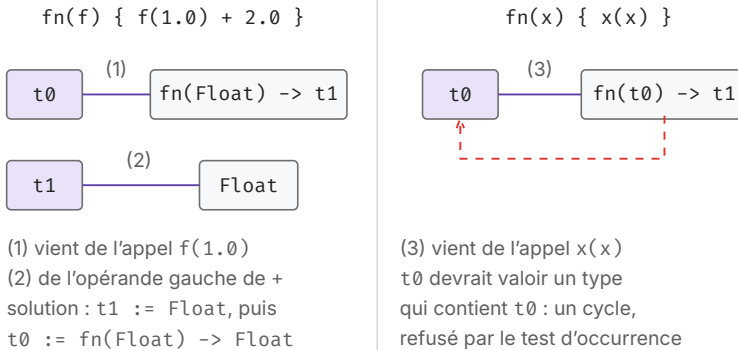
Résoudre ces équations s'appelle *unifier* : trouver une *substitution*, une table qui associe à chaque inconnue un type, telle qu'en remplaçant les inconnues des deux côtés d'une équation on obtienne deux types identiques. Le carnet 2 vous fait écrire cet algorithme ; sa logique tient en quatre cas, après avoir remplacé dans les deux types ce que la substitution sait déjà. Si les deux types sont identiques, il n'y a rien à apprendre. Si l'un des deux est une inconnue encore libre, on la lie à l'autre. Si ce sont deux types de fonctions de même arité, on unifie les arguments deux à deux, puis les résultats, en transmettant la substitution de paire en paire. Sinon, par ex. `Float` contre `Bool` ou `Float` contre une fonction, les deux types sont incompatibles et le programme est refusé.

Voyons l'algorithme travailler sur une fonction dont le paramètre est lui-même une fonction. Suivez les contraintes dans l'ordre où le parcours de l'arbre les rencontre :

```
1 fn(f) { f(1.0) + 2.0 }
```

Nœud visité	Contrainte produite	Substitution après
paramètre <code>f</code>	aucune ; <code>f : t0</code>	vide
appel <code>f(1.0)</code>	<code>t0 = fn(Float) -> t1</code> , <code>t1</code> frais	<code>t0 := fn(Float) -> t1</code>
opérande <code>f(1.0)</code> de <code>+</code>	<code>t1 = Float</code>	<code>t0 := fn(Float) -> t1</code> , <code>t1 := Float</code>
opérande <code>2.0</code> de <code>+</code>	<code>Float = Float</code>	inchangée
la fonction entière	aucune	type <code>fn(t0) -> Float</code>

Le type trouvé, $\text{fn}(t_0) \rightarrow \text{Float}$, se lit en appliquant la substitution jusqu'au bout : t_0 vaut $\text{fn}(\text{Float}) \rightarrow t_1$, et t_1 vaut Float , donc la fonction a le type $\text{fn}(\text{fn}(\text{Float}) \rightarrow \text{Float}) \rightarrow \text{Float}$. Personne n'a annoté f ; ce sont les usages de f dans le corps qui ont dit ce qu'il devait être.



À gauche, chaque équation relie deux types ; résoudre, c'est lier les inconnues jusqu'à ce que les deux bouts de chaque trait coïncident. À droite, l'équation de $x(x)$ demande à t_0 d'être égal à un type qui contient t_0 : le test d'occurrence refuse.

3.2 Le test d'occurrence

Le cas de droite mérite une pause. Dans $\text{fn}(x) \{ x(x) \}$, le paramètre x reçoit t_0 , et l'appel $x(x)$ exige $t_0 = \text{fn}(t_0) \rightarrow t_1$. Si l'on liait naïvement t_0 à ce type, on obtiendrait un type qui se contient lui-même, $\text{fn}(\text{fn}(\text{fn}(\dots) \rightarrow t_1) \rightarrow t_1) \rightarrow t_1$, sans fin. Aucun type fini ne satisfait l'équation. Avant de lier une inconnue, l'algorithme vérifie donc qu'elle n'apparaît pas dans le type en face : c'est le *test d'occurrence* (*occurs check*), et le projet rend alors `{:error, {:infinite_type, ...}}`.

PIÈGE

Quelle est la différence entre $t_0 = t_0$ et $t_0 = \text{fn}(t_0) \rightarrow t_1$? Dans les deux cas, t_0 apparaît à droite. Pourtant la première équation est triviale et la seconde impossible. Un typeur qui ferait le test d'occurrence avant de vérifier que les deux types sont identiques refuserait $t_0 = t_0$, et avec elle des programmes parfaitement corrects. L'ordre des quatre cas n'est pas une coquetterie.

Ce test a un prix : pour unifier, il faut parcourir des types, et un type peut grossir à chaque liaison. L'algorithme publié par John Alan Robinson en 1965, au cœur de sa méthode de *résolution* pour la démonstration automatique, est exponentiel dans le pire cas ; Michael Paterson et Mark Wegman ont montré en 1978 qu'on peut unifier

en temps linéaire en partageant les sous-termes au lieu de les recopier. Notre typeur, qui unifie des types de quelques dizaines de symboles, n'a pas ce souci.

4 Le type principal : Hindley, Milner et l'algorithme W

L'histoire de l'inférence se joue en trois temps. En 1969, le logicien J. Roger Hindley étudie la logique combinatoire, un calcul sans variables liées qui descend des travaux de Haskell Curry, et montre que tout terme typable y possède un *type principal* : un type dont tous les autres types possibles du terme sont des instances, c-à-d. obtenus en remplaçant ses variables par des types. Sa preuve utilise exactement l'unification de Robinson.

En 1978, Robin Milner, à Édimbourg, travaille sur LCF, un assistant de preuve, et sur son langage de commande, ML, « méta-langage ». Il a besoin d'un typage fort pour garantir que les scripts de preuve ne fabriquent pas de théorème faux, mais ne veut pas que chaque script soit encombré d'annotations. Il redécouvre le résultat de Hindley pour un vrai langage de programmation, avec `let`, et décrit dans « A Theory of Type Polymorphism in Programming » un algorithme, baptisé *W*, qui parcourt l'arbre en produisant des contraintes et les unifie au fil de l'eau. C'est l'algorithme que vous écrivez dans le carnet 3. Milner prouve que *W* est *sûr* : le type qu'il rend est bien un type du programme. En 1982, Luis Damas, son étudiant, et lui prouvent qu'il est aussi *complet* : si le programme a un type, *W* en trouve un, et celui qu'il trouve est principal. On parle depuis de système de types de Hindley–Milner, ou de typage « à la Damas–Milner ».

À RETENIR

Un *type principal* est le type le plus général d'une expression : tous ses autres types s'en déduisent par substitution. `fn(x) { x }` a pour type principal `fn(t0) -> t0` ; `fn(Float) -> Float` et `fn(Bool) -> Bool` en sont des instances. L'algorithme *W* rend toujours le type principal, et c'est ce qui autorise à ne jamais annoter : le typeur ne choisit jamais à votre place un type plus restrictif que nécessaire.

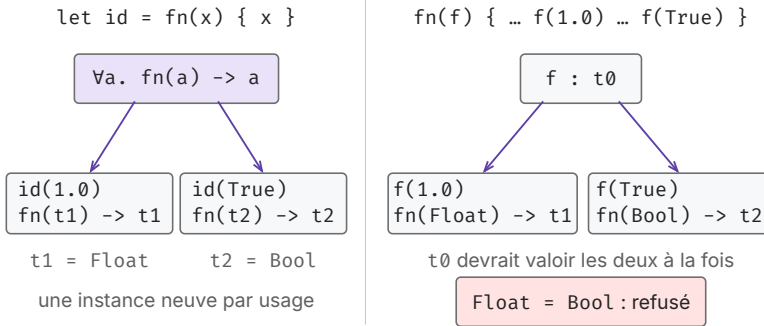
Ce résultat explique une propriété que vous observez dans le projet sans peut-être l'avoir nommée : le type affiché par `MiniTypeur.check/1` ne dépend pas de l'ordre

dans lequel vous avez traité les contraintes, à renommage des inconnues près. Deux parcours corrects donnent le même type principal.

5 Le polymorphisme de `let`

Le type principal `fn(t0) -> t0` dit que `fn(x) { x }` fonctionne pour n'importe quel `t0`. Encore faut-il pouvoir l'utiliser deux fois à deux types différents. Dans `let id = fn(x) { x } let n = id(1.0) id(True)`, si `id` garde l'inconnue `t0` partagée, le premier appel impose `t0 = Float` et le second `Float = Bool`. C'est le souci que le carnet 4 vous fait constater avant de le résoudre.

La solution de Milner est le *schéma de type* : au moment du `let`, on *généralise* le type de la valeur en marquant quantifiées, pour tout, les inconnues qui n'appartiennent qu'à elle ; `id` reçoit le schéma $\forall a. \text{fn}(a) \rightarrow a$. À chaque lecture du nom, on *instancie* le schéma en remplaçant ses variables quantifiées par des inconnues neuves, identiques à toutes leurs occurrences dans cette instance. Les deux appels reçoivent ainsi `fn(t1) -> t1` et `fn(t2) -> t2`, que rien n'oblige à coïncider.



À gauche, le `let` généralise `id` en un schéma, et chaque usage reçoit une instance neuve. À droite, `f` est un paramètre, rien n'est quantifié : son inconnue `t0` reste partagée entre les deux appels, et les contraintes se contredisent.

5.1 Pourquoi un paramètre reste monomorphe

Pourquoi ne pas généraliser aussi `f` dans `fn(f) { let n = f(1.0) f(True) }` ? Parce que `t0` n'appartient pas au corps : c'est l'appelant qui fournira `f`, et l'appelant fournira *une* fonction, d'un seul type. Généraliser `t0` reviendrait à promettre que n'importe quel argument convient à tous les types, promesse que l'appelant ne peut pas tenir. La règle est donc : on quantifie les inconnues libres du type de la valeur,

moins celles qui sont libres dans l'environnement, car ces dernières sont encore engagées ailleurs. C'est exactement la différence d'ensembles du carnet 4, et c'est pourquoi un alias `let y = x` d'un paramètre ne devient pas polymorphe non plus : `t0` est dans l'environnement, par `x`.

Cette asymétrie entre `let` et les paramètres n'est pas une paresse. Accepter qu'un paramètre soit polymorphe revient à inférer des types du système *F* de Jean-Yves Girard et John Reynolds, et Joe Wells a prouvé en 1999 que cette inférence est indécidable : aucun algorithme ne peut la faire dans tous les cas. Le polymorphisme restreint à `let`, dit de *rang 1*, est le compromis qui reste décidable tout en couvrant la plupart des programmes.

EN PRATIQUE

Pour tester votre généralisation, écrivez des programmes où la substitution a changé la forme de l'environnement avant le `let` : si `f : t0` et que l'on sait déjà `t0 = fn(Float) -> t1`, l'inconnue à protéger est `t1`. Calculer les variables libres sans résoudre d'abord les types de l'environnement généralise trop, et le typeur accepte un programme faux.

6 Ce que le typeur garantit, et ce qu'il ne garantit pas

L'article de Milner contient une phrase devenue slogan : *well-typed programs cannot go wrong*, les programmes bien typés ne peuvent pas « mal tourner ». Lue trop vite, elle promet trop. Dans l'article, « go wrong » est un terme technique : la sémantique de Milner donne une valeur spéciale, *wrong*, aux opérations absurdes, additionner un booléen, appeler un nombre. Le théorème dit qu'un programme bien typé ne s'évalue jamais en *wrong*. C'est notre propriété de sûreté, ni plus ni moins.

Deux choses restent donc hors de la promesse. Une division par zéro n'est pas *wrong* : c'est une erreur parfaitement définie, que la règle de `/` ne regarde pas, et `1.0 / 0.0` a le type `Float`. La terminaison n'est pas promise en général : un type dit ce que l'on obtiendra si le calcul finit. Notre langage est un cas particulier : sans `let` récursif, tout programme *bien typé* termine, c'est un théorème de normalisation, et le programme qui boucle en AP2, `fn(x) { x(x) }` appliqué à lui-même, est justement celui que le test d'occurrence refuse. Dès qu'un langage permet la récursion, en revanche, un programme bien typé peut boucler.

Inversement, le typeur peut refuser un programme qui s'exécuterait sans encombre : `case True { True -> 1.0 False -> True + 1.0 }` rend `1.0` en AP2, puisque la branche fausse n'est jamais évaluée, mais le typeur examine les deux branches et refuse. Il ne choisit pas un chemin ; il les promet tous.

Programme	Interpréteur AP2	Machine AP3	Typeur AP9
<code>True + 1.0</code>	erreur à l'évaluation	erreur au ADD	refusé avant
<code>case True { True -> 1.0 False -> True + 1.0 }</code>	<code>1.0</code>	<code>1.0</code>	refusé
<code>1.0 / 0.0</code>	erreur	erreur	Float
<code>fn(x) { x(x) }</code>	une fermeture	hors socle	refusé, type infini

6.1 Les compromis que l'on accepte en échange

Le système de Hindley–Milner est un point d'équilibre, et trois résultats en marquent les bords. D'abord, la *réursion polymorphe* : si une fonction récursive pouvait s'appeler elle-même à un type différent du sien, l'inférence deviendrait indécidable, comme l'a montré Fritz Henglein en 1993 ; les langages ML typent donc une fonction récursive de façon monomorphe dans son propre corps, sauf annotation explicite. Ensuite, la *restriction de valeur* : quand un langage ajoute des références modifiables, généraliser un `let` dont la valeur n'est pas syntaxiquement une valeur, par ex. le résultat d'un appel, rend le système faux, et Andrew Wright a proposé en 1995 la règle simple que suivent depuis Standard ML et, à quelques raffinements près, OCaml : ne généraliser que les valeurs. Enfin, le coût : Harry Mairson a prouvé en 1990 que l'inférence ML est exponentielle dans le pire cas, car un `let` imbriqué peut doubler la taille d'un type à chaque niveau ; à l'usage, les programmes réels ne le font jamais et l'inférence reste quasi linéaire.

7 Au-delà du projet

Les idées de ce fascicule vivent dans des langages que vous rencontrerez. OCaml, né à l'INRIA en 1996 dans la lignée de ML, et Haskell, dont le premier rapport date de 1990, utilisent tous deux Hindley–Milner comme socle. Dans un terminal OCaml, `let id x = x` affiche `val id : 'a -> 'a` : le `'a` est notre `τ₀` généralisé. Haskell y

ajoute les *classes de types* de Philip Wadler et Stephen Blott (1989), qui permettent à `+` de fonctionner sur plusieurs types numériques tout en restant inférable, là où notre langage a préféré interdire toute surcharge.

Rust, dont la version 1.0 date de 2015, fait un autre compromis : il infère les types à l'intérieur d'un corps de fonction, avec une unification de la même famille, mais exige d'écrire la signature de chaque fonction. L'inférence reste locale, les frontières sont documentées, et les *traits*, héritiers des classes de types, portent le polymorphisme. TypeScript, lancé par Microsoft en 2012, infère les types des variables à partir de leurs valeurs initiales et du contexte, mais ne généralise jamais un `let` ; ses fonctions génériques sont annotées. Chaque langage choisit où il devine et où il demande.

Elixir, enfin, construit un typeur sous vos yeux. Giuseppe Castagna, Guillaume Duboc et José Valim ont publié en 2023 « The Design Principles of the Elixir Type System », qui décrit un système de *types ensemblistes* : un type `y` est un ensemble de valeurs, et union, intersection et négation de types ont un sens, ce qui colle aux fonctions à clauses d'Elixir. Le typage `y` est *graduel*, avec un type `dynamic()` pour le code non encore typé. Il arrive par étapes : Elixir 1.17, en juin 2024, infère des types depuis les motifs et émet des avertissements sur les atomes et les maps ; 1.18, en décembre 2024, vérifie les appels de fonctions et infère les types de retour ; 1.19, en octobre 2025, type les fonctions anonymes et les protocoles. Avec Elixir 1.18.4, le script suivant compile, mais le compilateur prévient :

```
1 defmodule Essai do
2   def lance do
3     x = "quarante-deux"
4     x + 1
5   end
6 end
7 IO.puts("compilé")
```

Voici le début de l'avertissement, tel qu'il s'affiche, avant que le script n'écrive compilé :

```
1 warning: incompatible types given to Kernel.+/2:
2     x + 1
3 given types:
4     -binary()-, integer()
5 but expected one of:
6     #1
7     integer(), integer()
```

Les lignes vides ont été retirées ; le message continue avec les trois autres combinaisons acceptées et désigne la ligne fautive.

Aucune annotation n'a été écrite : le type de `x` vient de sa valeur, et la contrainte de `+` vient de la table des opérateurs, comme dans notre typeur. La différence est qu'Elixir, pour ne casser aucun programme existant, se contente d'avertir ; notre typeur refuse. Les signatures explicites sont annoncées pour des versions ultérieures.

8 Pour aller plus loin

A Theory of Type Polymorphism in Programming, Robin Milner (1978).

[https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)

L'article fondateur, en accès libre : l'algorithme W, la sémantique avec `wrong` et la preuve de sûreté. Les sections 1 à 3 se lisent avec le bagage du cours.

Principal Type-Schemes for Functional Programs, Luis Damas et Robin Milner (1982).

<https://doi.org/10.1145/582153.582176>

Six pages qui fixent la présentation moderne : les règles de typage, le schéma de type, et l'énoncé de complétude de W.

The Principal Type-Scheme of an Object in Combinatory Logic, J. Roger Hindley (1969).

<https://doi.org/10.2307/1995158>

Le résultat de 1969, pour voir que le type principal existait avant les langages de programmation qui l'exploitent.

A Machine-Oriented Logic Based on the Resolution Principle, John Alan Robinson (1965).

<https://doi.org/10.1145/321250.321253>

L'origine de l'unification, écrite pour démontrer des théorèmes automatiquement. La section sur l'algorithme d'unification est courte et se lit seule.

Type Inference with Polymorphic Recursion, Fritz Henglein (1993).

<https://doi.org/10.1145/169701.169692>

Pour comprendre la frontière : pourquoi laisser une fonction récursive changer de type la rend impossible à inférer.

Simple Imperative Polymorphism, Andrew Wright (1995).

<https://doi.org/10.1007/BF01018828>

La restriction de valeur expliquée par son auteur, avec les contre-exemples qui montrent pourquoi une généralisation naïve devient fausse dès qu'on ajoute des références.

The Design Principles of the Elixir Type System, Giuseppe Castagna, Guillaume Duboc et José Valim (2023).

<https://arxiv.org/abs/2306.06391>

Le plan du typeur d'Elixir : types ensemblistes, typage graduel, et pourquoi les clauses et les gardes d'Elixir demandent autre chose que Hindley–Milner.

Gradual set-theoretic types, documentation officielle d'Elixir.

<https://hexdocs.pm/elixir/gradual-set-theoretic-types.html>

L'état courant du typeur, mis à jour à chaque version : ce qui est inféré, ce qui est vérifié, et les avertissements que vous verrez dans vos projets.

9 Une question pour la route

Notre `let` n'est pas récursif : dans `let f = fn(n) { f(n - 1.0) }`, le `f` du corps est un nom inconnu. Supposez que l'on ajoute un `let rec`, qui rend `f` visible dans sa propre valeur. Quel type faut-il donner à `f` pendant que l'on type son corps, puisque l'on ne connaît pas encore le résultat ? Et pourquoi ce choix oblige-t-il `f` à rester monomorphe à l'intérieur de son propre corps, alors qu'il devient polymorphe juste après le `let rec` ? Si vous trouvez la règle, vous aurez retrouvé seul la raison du résultat de Henglein.

Pour toute question, vous pouvez me contacter par email, comme pour les autres séances.