Rapport de Prédoc III

Auto-élaboration avec types dépendants

Jean-Alexandre Barszcz

Résumé

L'inférence de types et les classes de types sont deux fonctionnalités importantes de certains langages de programmation statiquement typés, tant du point de vue de leur utilité, que de l'effort de développement qu'elles demandent.

Au lieu de les implanter dans l'interprète ou le compilateur d'un langage, on propose de les implanter par métaprogrammation, dans le langage développé lui-même. C'est ce qu'on appelle l'auto-élaboration. Procéder ainsi comporte des avantages qui rappellent ceux de l'auto-amorçage, en plus de faciliter celui-ci.

Le développement se fait dans le langage avec types dépendants Typer. Après la description de changements utiles à la métaprogrammation dans ce langage, on esquisse les deux bibliothèques d'élaboration. On propose également, comme contribution auxiliaire, une nouvelle sorte de macro où la portée est explicite, ce qui fournit une solution alternative aux macros hygiéniques. Finalement, on établit un plan pour le reste du projet de recherche.

1 Introduction

Beaucoup de langages de programmation sont constitués d'un langage noyau plus simple mais dans lequel les programmes seraient plus longs à écrire, et d'un langage plus ergonomique qui sert d'interface aux programmeurs. L'**élaboration** est la traduction du langage de suface au langage noyau. En Scheme, par exemple, l'élaboration fait l'expansion des macros, qui servent à définir des formes syntaxiques supplémentaires et donc étendre le langage et implanter des *Domain-Specific Languages* (DSLs). Dans les langages statiquement typés, l'élaboration sert à faire l'inférence de types et la résolution des arguments d'instance pour les classes de types. Pour certains assistants à la preuve basés sur les types dépendants, on y applique des tactiques de preuves, qui synthétisent le code réalisant des formules logiques à partir de leur contexte. Ainsi, cette phase constitue généralement une grande portion de l'implantation des langages statiquement typés. On en vient à définir l'auto-élaboration, un nouveau nom pour une vieille idée :

> L'**auto-élaboration** d'un langage permettant la métaprogrammation consiste à implanter des parties importantes de l'élaboration par métaprogrammation dans le même langage.

Même si cette définition est un peu floue, elle reste tout de même utile pour cerner certains avantages. L'auto-élaboration en soi hérite des bienfaits de l'auto-amorçage (*bootstrapping*), relatés à la section 1.2. Remarquons d'abord l'intérêt particulier de l'appliquer à un langage statiquement typé :

- L'élaboration y fait plus de choses, et représente un effort d'implantation plus important. Il est donc d'autant plus utile de transporter cet effort vers des bibliothèques écrites dans le langage lui-même.
- Le code qui servira à implanter les bibliothèques d'élaboration peut profiter des avantages du langage typé. Plus précisément, une difficulté courante en métaprogrammation est que les multiples niveaux de traduction peuvent rendre les erreurs plus difficiles à retracer. Le typage statique apporte certaines solutions à ce problème et encourage une métaprogrammation plus robuste.

1.1 Énoncé de thèse

Je montrerai comment on peut implanter, dans un langage avec des types dépendants, une élaboration extensible au point que d'importantes composantes habituelles de l'élaboration comme l'inférence de type et les classes de types deviennent des bibliothèques dans ce langage même.

1.2 L'auto-élaboration en relation à l'auto-amorçage

L'auto-amorçage d'un compilateur d'un langage de programmation – soit son implantation dans le même langage compilé – comporte plusieurs avantages :

1. Puisqu'un compilateur traduit généralement un langage d'un niveau d'abstraction plus élevé vers un autre d'un niveau plus bas, l'implantation dans ce même langage de niveau plus élevé en est facilitée.
2. Les développeurs du compilateur n'ont besoin de comprendre et manipuler qu'un seul langage.
3. Appliquer un compilateur à lui-même est un test important pour celui-ci, puisqu'il s'agit d'un programme non-trivial.
4. Il s'agit également d'un test utile pour évaluer le langage lui-même, qui doit alors être assez expressif et ergonomique pour permettre l'écriture de son compilateur.
5. Les améliorations au code généré par le compilateur s'appliquent également à son propre exécutable.

Tous ces avantages se transportent aussi à l'auto-élaboration :

1. L'implantation de fonctionnalités dans le système de métaprogrammation du langage est facilitée par rapport à une implantation équivalente dans un autre langage (ceci est également une condition).
2. Des développeurs peuvent comprendre et modifier une partie plus importante de l'implantation à l'aide d'un seul langage.
3. Les bibliothèques d'auto-élaboration constituent un test intéressant pour l'implantation du langage.
4. Elles offrent également un test utile pour évaluer à quel point le système de métaprogrammation est expressif et ergonomique.
5. Les améliorations au code généré par métaprogrammation s'appliquent également à une bibliothèque d'élaboration qui l'utiliserait.

En plus de ces avantages, l'auto-élaboration sert à simplifier le langage noyau, et elle rend donc son implantation plus portable et l'éventuel auto-amorçage d'autant plus facile.

1.3 Cadre

Typer (Monnier 2019) servira de base pour la conception. Il s'agit d'un langage fonctionnel avec types dépendants, et dans lequel on peut déjà employer une forme de métaprogrammation. L'implantation de Typer consiste principalement d'un interpréteur écrit en Ocaml, qui fait une élaboration dont une partie est déléguée à des macros. Par exemple, le sucre syntaxique `do` pour les monades, et l'implantation du filtrage (*pattern matching*) sont déjà réalisés par du code Typer. D'autres parties sont écrites en Ocaml, comme l'inférence à la Hindley-Milner, et les arguments d'instances pour classes de types. Ces derniers y ont d'ailleurs été développés pour le mémoire de maîtrise de l'auteur (Barszcz 2021).

1.4 Vue d'ensemble de l'auto-élaboration de Typer

L'auto-élaboration sous-entend une hiérarchie de langages. Ainsi, on séparera le langage noyau du langage d'inférence de Typer.

Le langage noyau est un sous-ensemble du langage d'inférence et emploiera une élaboration très simple (avec une inférence de types bidirectionnelle). Tous les arguments de types devront donc y être donnés explicitement.

Le langage d'inférence ajoutera à l'autre les éléments nécessaire à l'implantation de l'inférence par unification, soit les paramètres implicites et les métavariables. Il sera également conçu de façon à permettre des extensions qui s'intègrent à cette inférence de types, telles que les arguments d'instance servant aux classes de types.

Est-ce que la bibliothèque d'élaboration se chargeant de l'inférence sera elle même écrite dans le langage d'inférence ? Durant son développement, elle cohabitera avec l'implantation de l'inférence écrite en Ocaml et pourra donc elle même en profiter. On voudra éventuellement simplifier le code Ocaml et en retirer cette partie. On se retrouvera alors dans une situation commune pour l'auto-amorçage, où l'on pourra traduire la bibliothèque d'inférence vers le langage noyau (soit à la main ou en gardant une ancienne version élaborée, par exemple).

1.5 Structure de ce rapport

La prochaine section sert d'introduction à Typer et présente des changements préalables à l'écriture des bibliothèques d'élaboration. Ces dernières sont ébauchées à la section 3. La section 4 décrit une contribution auxiliaire. On passe en revue la littérature pertinente à la section 5. Finalement, on établit un plan de travail à la section 6.

2 Métaprogrammation dans le langage noyau

Actuellement, la métaprogrammation en Typer se fait à l'aide de macros semblables à celles de Scheme (utilisant `define-macro`). Ces macros agissent sur des S-expressions, soit essentiellement des arbres contenant des symboles et des littéraux tels que des nombres ou des chaînes de caractères.

Ainsi, les macros sont définies à l'aide de fonctions de type `List Sexp → IO Sexp`, qui sont souvent appellées *syntax transformers*. Ces fonctions prennent en entrée la liste des S-expressions correspondant aux paramètres de la macro et retournent une nouvelle S-expression dans une monade, reflétant le fait qu'elles peuvent avoir des effets de bord.

Malheureusement, les S-expressions sont assez loin de la représentation du langage d'inférence que la bibliothèque d'inférence devra manipuler. On choisit donc de remplacer les macros actuelles par des macros dites « sans résidus » qui produisent directement les λ-expressions du λ-calcul sous-jacent à Typer. Pour faire ce changement, on doit faire la réflexion de ce nouveau type de destination. Les deux sections suivantes décrivent ces étapes qui, sans être particulièrement innovatrices, sont préalables au reste du projet et servent d'introduction à la métaprogrammation en Typer.

2.1 Macros sans résidus

Herman et Wand distinguent les macros qu'ils appellent « d'ordre supérieur », qui peuvent produire du code qui contient d'autres appels de macros, de celles « de premier ordre » qui génèrent du code sans appels de macros résiduels (Herman and Wand 2008). Ganz *et al.* semblent utiliser une signification légèrement différente, indiquant que les macros d'ordre supérieur sont « paramétrées » par d'autres macros (Ganz, Sabry, and Taha 2001). On appellera les macros de premier ordre au sens de Herman et Wand « sans résidus ». Bien que les programmeurs de Scheme sont habitués à des macros **avec** résidus, qui paraîssent plus générales, les macros **sans** résidus sont de plus en plus communes pour les programmeurs des langages avec types dépendants.

Puisque la sortie de ces dernières est limitée aux formes syntaxiques de base, on peut éviter une analyse syntaxique redondante en produisant directement le code dans sa forme abstraite. Au lieu de macros récursives, on peut écrire des macros dont la fonction d'élaboration est une fonction récursive. Considérons l'exemple de la macro récursive de puissance.

```
1 (define-macro pow
2   (lambda (n x)
```

```
    (case n
      ((0) 1)
      ((1) x)
      (else `(* ,x (pow ,(- n 1) ,x))))))
```

Dans cet exemple Scheme, on utilise `define-macro` sans vérifier les entrées de la macro et sans considérer les questions d'hygiène. On remarque que celle-ci génère du code qui contient un nouvel appel à `pow` (qui est dans la quasi-citation `` ` ``, mais hors de l'anti-citation `,`). En sortie, on obtient donc une S-expression qui sera de nouveau analysée, et toute erreur sera signalée selon le nouveau code et déconnectée de ce que le programmeur aura écrit. En guise d'introduction à Typer, voici comment on traduirait cette macro avec résidus dans la version courante de Typer :

```
pow-trs : List Sexp -> IO Sexp;
pow-trs args =
  case args
  | cons sn (cons sl nil) => % Vérification qu'il y a 2 arguments
    (case Sexp_wrap sn       % Analyse du premier
     | integer n =>          % ... pour confirmer qu'il s'agit d'un entier
       IO_return (
         if (Integer_eq n 0) then Sexp_integer 1 else
         if (Integer_eq n 1) then sl else
         (quote (_*_ (uquote sl)
                     (pow (uquote (Sexp_integer (Integer_- n 1))) (uquote
                     ↪  sl))))
       )
     | _ => Sexp_error "Parsing error in `pow`")
  | _ => Sexp_error "Parsing error in `pow`";

pow = macro pow-trs;
```

Le fonctionnement de cette macro est pratiquement identique, bien que l'analyse des entrées est plus verbeuse (ce qui est dû, en partie, à la façon de réfléchir les `Sexp`, traitée à la prochaine section). En somme, on commence par définir la fonction d'élaboration qui n'est pas récursive, mais qui retourne un appel à la macro `pow` dans la S-expression générée. La macro `pow` est liée par la deuxième définition. On remarque que le bon fonctionnement de la macro dépend de la coopération entre ces deux définitions.

On voudra plutôt écrire une macro sans résidus, qui produit directement une `Lexp`, soit une λ-expression de Typer, en syntaxe abstraite (qu'on détaillera davantage à la prochaine section) :

```
pow-abs : Int → Lexp → Lexp;
pow-abs n l =
  if (Int_eq n 0) then Lexp_imm (Sexp_integer 1) else
  if (Int_eq n 1) then l else
    quote-lexp (_*_ (unquote l) (unquote pow-abs (n - 1) l));

pow-trs : Option Lexp → List Sexp → Elab Lexp;
pow-trs ot args =
  case args
  | cons (Sexp_integer n) (cons sl nil) ⇒
      do { l ← Elab_elab-expr ot sl;
           Elab_return (pow-abs (Integer→Int n) l)}
  | _ ⇒ Elab_error "Parsing error in `pow`";

pow = lmacro pow-trs;
```

Là, on sépare l'implantation de `pow` en trois déclarations : la dernière enregistre la macro `pow` et la lie à son implantation, soit la fonction d'élaboration (*syntax transformer*) `pow-trs`. Cette fonction d'élaboration est conceptuellement divisée en deux parties : `pow-trs` analyse (*parse*) les entrées, élaborant récursivement les sous-expressions et émettant des erreurs, si nécessaire, puis elle délègue le reste du travail à `pow-abs`, qui se charge de la logique applicative de la macro, soit la génération du code (une λ-expression) pour une puissance donnée. Pour distinguer les deux fonctions, on précisera que `pow-abs` est la fonction d'élaboration de syntaxe abstraite (*abstract syntax transformer*).

Le premier avantage de structurer les macros de cette façon est que chacune est responsable de son propre succès : si elles délèguent une partie de l'élaboration, par exemple, en appelant une autre fonction d'élaboration, elles ont encore l'occasion de gérer ou contextualiser l'erreur qui pourrait résulter de cet appel.

Le deuxième avantage est qu'on évite des analyses inutiles : au lieu de produire une S-expression qui sera de nouveau analysée (avec le potentiel d'erreur de cet aller-retour supplémentaire), la macro peut directement appeller les fonctions d'élaboration de syntaxe abstraite, comme dans l'exemple de `pow-abs`, qui s'appelle récursivement. Pour utiliser une syntaxe concrète, le programmeur devrait privilégier une quasi-citation vers la syntaxe abstraite plutôt que vers une S-expression, qui sera ainsi analysée au moment de l'élaboration de la macro plutôt qu'à son exécution.

Le troisième avantage est que les appels récursifs à l'élaborateur sont explicites (comme les appels à `Elab_elab-expr` ci-haut). En général, les fonc-

tions d'élaboration des expressions et déclarations sont d'abord définies pour les formes de base du langage et peuvent être étendues (paramétrées) par un ensemble de macros. Cet ensemble n'est connu qu'au lieu d'appel d'une macro, et pas encore à la définition de celle-ci. Les élaborations récursives sont donc particulières et, en suivant le principe que seules les choses qui sont simples dans l'esprit devraient être simples dans le code – et vice-versa – il est utile de les considérer de façon explicite. Notons qu'on pourrait toujours terminer la fonction d'élaboration par un appel à l'élaborateur pour avoir un comportement similaire à celui d'une macro récursive. On pourra donc écrire un code similaire à l'exemple Scheme précédent, démontrant la généralité de l'approche sans résidus avec une primitive d'élaboration adéquate. On remarque ainsi que même si cette conception est de premier ordre au sens de Herman et Wand, elle est d'ordre supérieur au sens de Ganz *et al.*, puisque chaque macro peut appeller l'élaborateur paramétré par d'autres macros. Cependant, cette puissance pose aussi un danger : celui d'une élaboration infinie. On peut vérifier la terminaison de chaque fonction d'élaboration séparément, mais ce n'est pas suffisant : un appel à l'élaborateur peut aussi être une source de récursion puisqu'il peut rappeller la même fonction d'élaboration. On ne considérera pas la terminaison des fonctions ni des macros plus en détail, mais on note tout de même que l'appel explicite à l'élaborateur est un pas dans la bonne direction. D'une part, il offre l'opportunité de se poser ces questions, et plus encore, il est compatible avec de solutions potentielles, comme celle de limiter les appels à `Elab_elab-expr` à des S-expressions « plus petites » que le paramètre de la macro.

Un quatrième avantage est qu'en séparant la validation du reste de l'implantation de la macro, on ouvre la possibilité de partiellement générer et réutiliser du code de validation. Cet aspect rappelle la validation des macros `syntax-parse` en Scheme (Culpepper and Felleisen 2010).

L'avantage final, et le plus important, est qu'on peut choisir le type de retour des macros pour qu'il soit plus évocateur. Historiquement, les systèmes de métaprogrammation on utilisé toute une hiérarchie de représentations des programmes, avec différents niveaux de garanties :

1. Les chaînes de caractères sont la représentation la moins structurée et n'offrent aucune garantie.

2. Les S-expressions expriment la structure hiérarchique de expressions, sans pour autant garantir que celle-ci sont bien formées. Cette représentation évite donc les plus simples erreurs de syntaxe (parenthésage).

3. Les arbres de syntaxe abstraite assurent que la syntaxe est respectée.

4. Les programmes bien typés peuvent être représentés par leur dérivations de type ou à l'aide d'une représentation intrinsèquement typée.

Alternativement, le résultat des macros peut être changé, non pas pour son niveau de garanties, mais comme dans Dolorem (Henniger and Amin 2023), où les macros retournent du code dans un langage plus près de la machine (plus bas niveau d'abstraction).

Dans un premier temps, par simplicité, on représente les programmes par leur arbre de syntaxe (niveau 3). La représentation de programmes bien typés a été étudiée ailleurs (ex. Devriese and Piessens 2013), et on estime qu'ils demanderaient trop de preuves pour être utilisables dans bien des cas. On considère donc qu'ils sont hors de la portée de ce travail. On explorera une solution intermédiaire, soit les programmes à bonne portée, comme contribution auxiliaire, à la section 4.

Notons que cette partie du travail, soit l'ajout de macros sans résidus, a déjà été accomplie en grande partie (suffisemment pour l'exemple `pow`[1]).

2.2 Implantation de la réflexion

Brazilay définit les systèmes réflexifs en requérant que ceux-cis puissent représenter leurs propres formes syntaxiques et avoir un moyen de les utiliser (Barzilay 2006). Dans leur étude sur la métaprogrammation, Lilis et Savidis qualifient cette forme de réflexion comme étant « structurelle » (Lilis and Savidis 2019). Puisqu'on a déjà donné un exemple d'utilisation des types `Sexp` et `Lexp` pour offir une interface de métaprogrammation, il reste à montrer comment on peut représenter ceux-ci en Typer afin d'implanter cette interface. Notons qu'il y a bien d'autres types correspondant à divers aspects de Typer. Par exemple, les grammaires sont réfléchies pour étendre la syntaxe concrète à partir de code Typer, et les contextes de typage et de définitions le sont pour permettre aux macros de les consulter.

Jusqu'à aujourd'hui, les concepts de Typer pouvaient être réfléchis à l'aide de constantes et de fonctions prédéfinies. Cependant, il paraît plus naturel de se servir des types inductifs de Typer comme équivalents aux types algébriques de Ocaml. De cette façon, on peut utiliser la forme `case` pour filter les données réfléchies. On a donc ajouté cette fonctionnalité à Typer. De plus, alors que la solution courante à ce problème se sert une paire de fonctions (`reify` / `reflect`) pour traduire l'arbre complet correspondant

1. À quelques détails près : on a pas encore de quasi-citation pour les `Lexp`, ni de *cross-stage persistence* pour `_*_`. L'exemple réel est similaire, mais ajoute un argument distrayant à `pow-abs` pour obtenir `_*_` dans le contexte de destination.

à une donnée du méta-langage vers le langage objet et vice-versa (ex. Korkut 2019 ; Brady 2021b), on adopte une approche paresseuse en adaptant `reflect` pour ne traduire que le niveau nécessaire à l'évaluation d'un `case`. L'amélioration de performance demeure à mesurer, mais constatons déjà que ce changement amoindrit les conséquences d'une réflexion imparfaite (ex. qui omet l'emplacement des expressions dans le code).

On conclut cette section en détaillant brièvement le λ-calcul de Typer pour cerner les différences entre ses différentes réalisations. La définition du type réfléchi apparaît au listage 1.

```
type Lexp
  | l_immediate (s : Sexp) % Strings, integers, floats ...
  | l_builtin (sym : Symbol) (t : Lexp)

  | l_var (vref : Vref)
  | l_lambda (k : ArgKind) (aname : Symbol) (to : Lexp) (body : Lexp)
  | l_call (f : Lexp) (args : List Arg)
  | l_arrow (k : ArgKind) (aname : Symbol) (from : Lexp) (to : Lexp)

  | l_sortlevel (sl : SortLevel)
  | l_let (defs : List Ldef) (body : Lexp)

  | l_inductive (name : Symbol) (ps : List Param) (ctors : List Ctor)
  | l_cons (t : Lexp) (lbl : Symbol)
  | l_proj (l : Lexp) (lbl : Symbol)
  | l_case (s : Lexp) % scrutinee
           (bt : Lexp) % return type of branches
           (bs : List Branch)
           (db : Option DefaultBranch);
```

Listing 1 : Définition du type inductif reflétant `Lexp` en Typer

Les deux premiers constructeurs servent aux valeurs littérales et aux primitives. Typer étant basé sur un λ-calcul, on ne s'étonnera pas d'y retrouver les variables, les fonction lambdas et les appels de fonction. `l_arrow` sert à représenter les flèches, soit le type des fonctions, `l_sortlevel` sert aux univers (les types des types), et `l_let` sert aux définitions (potentiellement récursives). Les quatres constructeurs restants servent à décrire les types inductifs et leurs formes d'introduction et d'élimination.

Les définitions des types réfléchis sont simplifiées par rapport à celles qu'on retrouve en Ocaml : les `Map` Ocaml sont traduites en liste d'association, des détails d'implantation comme les substitutions explicites sont cachés, et

on oublie l'emplacement du code source associé à chaque expression.

On n'entrera pas dans les détails des types auxiliaires, sauf `ArgKind`, qui sert à différencier les variétés de flèches et de lambdas du λ-calcul de Typer. Il s'agit d'une énumération qui distingue les flèches « normales » `→`, des flèches implicites `⇒` et des flèches effacées `≡>`. Les flèches implicites servent principalement à pouvoir omettre des arguments afin de les inférer, on pourrait donc vouloir les exclure du langage noyau. C'est, d'ailleurs, ce qui est fait dans une description du système de types de Typer (Monnier 2019). Au moins pour un premier prototype, on décide plutôt de conserver les paramètres implicites, afin de limiter l'écart entre les deux niveaux de langage et afin d'éviter l'annotation des flèches *a posteriori*. Même si la variété de flèche implicite fera partie du langage noyau, elles seront inopérantes sans unification et tous les arguments y seront fournis explicitement.

3 Bibliothèques d'élaboration

Maintenant que les bases de la métaprogrammation dans le langage noyau de Typer ont été expliquées, ont peut esquisser les deux bibliothèques d'élaboration principales visée dans ce travail.

3.1 Inférence

Alors que le langage noyau nécessite que tous les arguments de type soient donnés explicitement, le langage d'inférence permet d'omettre des informations qui seraient redondantes. Par exemple, l'inférence permet d'utiliser les constructeurs `nil : (t : Type) ⇒ List t` et `cons : (t : Type) ⇒ t → List t → List t` sans argument de type dans l'expression `(cons nil (cons (nil : List Int) nil))`. Les parties omises sont temporairement remplacées par des métavariables (préfixées par `?` en Typer) qui représentent des trous dans le code : `(cons (t :=?a) (nil (t :=?b)) (cons (t :=?c) (nil (t :=?d) : List Int) (nil (t :=?e))))`. L'inférence effectue l'unification pour déduire les types absent à partir de ceux qui sont connus. Par exemple, on peut déduire de l'appel ci-haut que `?a` et `List ?b` doivent correspondre à la même expression. C'est en arrivant aux arguments ultérieurs que le processus d'inférence déduit les λ-expression derrière ces métavariables.

La bibliothèque d'inférence doit ainsi commencer par définir les types correspondant au langage d'inférence. En particulier, `ILexp` correspondra essentiellement à `Lexp`, auquel on ajoutera les métavariables.

Une nouvelle monade d'élaboration avec inférence `IElab` pourra alors être implantée en Typer avec un nouveau type de macros témoignant du changement vers les `ILexp`. Ce changement se répercutera également aux primitives comme `Elab_elab-expr`.

Il faudra également prévoir un point d'entrée vers le langage d'inférence. Pour ce faire, on pourrait s'inspirer de Racket, où chaque module indique le langage dans lequel il est écrit (Tobin-Hochstadt et al. 2011).

L'unification se sert de la réduction du λ-calcul sous-jacent. Une difficulté consistera à faire la réflexion de la réduction des `Lexp` de façon à pouvoir l'utiliser avec `ILexp`.

3.2 Arguments d'instance

Les arguments d'instance (Devriese and Piessens 2011) servent à implanter les classes de types, un outil important à la modularité dans les langages de programmation fonctionnels statiquement typés et dans les assistants à la preuve (Ringer et al. 2019). Ils ont d'abord été introduits afin de faire la surcharge d'opérateurs (Wadler and Blott 1989). McBride a démontré leur utilité pour la recherche de preuves en Agda (McBride 2014).

En somme, il s'agit d'une façon d'instancier des arguments implicites en recherchant une variable du bon type dans le contexte de typage. Le fonctionnement sera similaire à ce que fait l'implantation Ocaml de Typer (Barszcz 2021).

On note que pour savoir quelles variables sont des instances, on devra ajouter cette nouvelle information à la monade d'élaboration. Cela peut se faire en définissant une nouvelle par dessus la dernière (possiblement avec un *monad transformer* (Liang, Hudak, and Jones 1995)). En Racket, ce sont les *syntax properties* ou *syntax parameters* qui servent à ajouter de l'information par la bande durant l'élaboration (Tobin-Hochstadt et al. 2011 ; Flatt and PLT 2023).

Un détail important est que le contexte d'élaboration peut contenir des métavariables qui ne sont pas encore instanciées durant l'inférence. Cela peut poser problème lorsqu'on essaye de déterminer si une instance (une variable du contexte) est adéquate ou non. La solution est de remettre la résolution jusqu'au moment où la métavariable est instanciée. L'interface de la monade d'inférence offrira donc un moyen de reporter certaines contraintes/problèmes d'élaboration. Klister est un langage qui est conçu pour que les macros puissent bloquer sur des métavariables et reprendre quand celles-ci sont instanciées (Barrett, Christiansen, and Gélineau 2020). D'ailleurs, ses auteurs espèrent aussi y implanter les classes de types comme

une bibliothèque. En Agda, les macros peuvent également bloquer, sauf qu'elles reprennent depuis leur début (The Agda Team 2021). La situation est similaire en Lean 4 (Ullrich and de Moura 2022).

4 Contribution auxiliaire : Macros à bonne portée

On décrit maintenant une contribution auxiliaire potentielle qui sera étudiée, et qui améliorerait la solution de l'auto-élaboration en Typer, mais qui n'est pas absolument nécessaire à son succès.

Les termes à bonne portée sont une représentation des λ-expressions utilisant le système de types du méta-langage pour garantir que les variables référencées par une expression correspondent effectivement à des variables de son contexte (Bird and Paterson 1999). Remarquons, au passage, qu'ils ont déjà été utilisés pour implanter, en Haskell, l'unification d'ordre supérieur pour faire l'inférence d'un langage avec des types dépendants (Kudasov 2022). On aimerait explorer la possibilité de les utiliser comme type de destination des macros.

En Typer, on peut se référer aux variables soit par leur nom, soit par leur indice de de Bruijn. Ainsi, on pourrait définir la portée par une liste de noms optionnels :

```
1 Scope : Type;
2 Scope = List (Option Symbol);
```

On pourrait ensuite ajuster les λ-expressions pour qu'elles aient cette portée comme indice de type. La signature suit, mais la définition demandera plus de travail, afin de refléter toute la structure de liaison du langage.

```
1 SLexp : Scope → Type;
```

Les macros pourraient insérer du code ouvert (c.à.d. avec des variables libres) à l'aide de fonctions d'élaboration de type `(s : Scope) ⇒ Option (SLexp s) → Sexp → Elab (SLexp s)`. Du côté des fonctions d'élaboration de syntaxe abstraite, les liaisons devront être explicites. Par exemple, `pow-abs` pourrait prendre le type `(s : Scope) ⇒ (« _*_ » in-scope s) ⇒ Int → SLexp s → SLexp s`, où la relation `in-scope` rend explicite le fait que le bout de code généré fait référence à `_*_`, et où le contexte des sous-expressions en paramètre est fixé.

Il s'agit d'une approche alternative à celle des macros hygiéniques, décrites en détail dans une étude récente (Clinger and Wand 2020). Les macros

hygiéniques ont été définies dans le contexte de Scheme par la *Strong Hygiene Condition* qui est formulée en fonction des liaisons et des références qui sont « insérées » par une macro. Cette définition s'appuie implicitement sur les macros avec résidus.

Notre alternative ressemble à certains systèmes mentionnés dans l'étude, où la structure de liaison de la macro est spécifiée d'emblée. En particulier, les *Staged Notational Definitions* étendent similairement des macros dans un langage à plusieurs phases (*multi-stage*) avec des annotations pour expliciter la structure de liaison (Taha and Johann 2003). Notre proposition est différente de deux façons :

- Elle utilise les types dépendants plutôt qu'une fonctionnalité *ad hoc*.
- L'hygiène n'est pas forcée. En effet, notre solution accomode également les macros non-hygiéniques (aussi dites « anaphoriques »).

5 Travaux connexes

Typed Racket (Tobin-Hochstadt et al. 2011) utilise des macros pour implanter un système de types pour Racket. Ce système inclut une inférence de types (Tobin-Hochstadt et al. 2023). Cependant, les types dépendants n'y sont pas considérés.

Turnstile+ (Chang et al. 2019) est un méta-DSL pour bâtir des langages avec types dépendants. Ce système est implanté en Racket. À titre d'exemple, ses auteurs ont implanté le langage Cur, et montrent comment son unification et son inférence peuvent être implantés à l'élaboration. Cependant, il ne s'agit pas d'une auto-élaboration, puisque leur `unify` est une macro Scheme, et leur langage d'inférence avec des paramètres implicite est implanté avec des règles du méta-DSL Turnstile+ : aucun des deux n'est fait dans Cur lui-même.

Autant pour Typed Racket que pour Turnstile+, on commence avec le langage noyau dynamiquement typé Racket. Dans les deux cas, donc, ces bibliothèques ne peuvent pas profiter des avantages du typage statique.

Le langage avec types dépendants Idris2 est auto-amorcé, et implante donc sa propre inférence, sauf qu'il ne le fait pas à travers son interface de métaprogrammation (Brady 2021a). Par ailleurs, c'est dans Idris qu'a d'abord été implantée la réflection de l'élaboration (Christiansen and Brady 2016 ; Christiansen 2016), qui est maintenant également utilisée dans d'autres langages similaires comme Agda. L'interface offerte dans ces systèmes ressemble à celle des macros qu'on développe au niveau du langage

d'inférence de Typer, sauf qu'on l'implante en Typer plutôt que de la réfléchir.

Le formalisme couramment utilisé pour expliquer l'auto-amorçage est celui des diagrammes en T, d'abord introduit par bratman (Bratman 1961), puis étendu pour inclure les interprètes et les machines en plus des compilateurs (Earley and Sturgis 1970). Lecarme *et al.* modélisent les processeurs de macros par des interpréteurs dans ce cadre, mais ça ne représente pas bien ce qui se passe en Scheme (Lecarme, Pellissier, and Thomas 1982).

6 Échéancier

On place maintenant les étapes du projet dans le temps. Le diagramme de Gantt de la figure 1 sert d'esquisse.

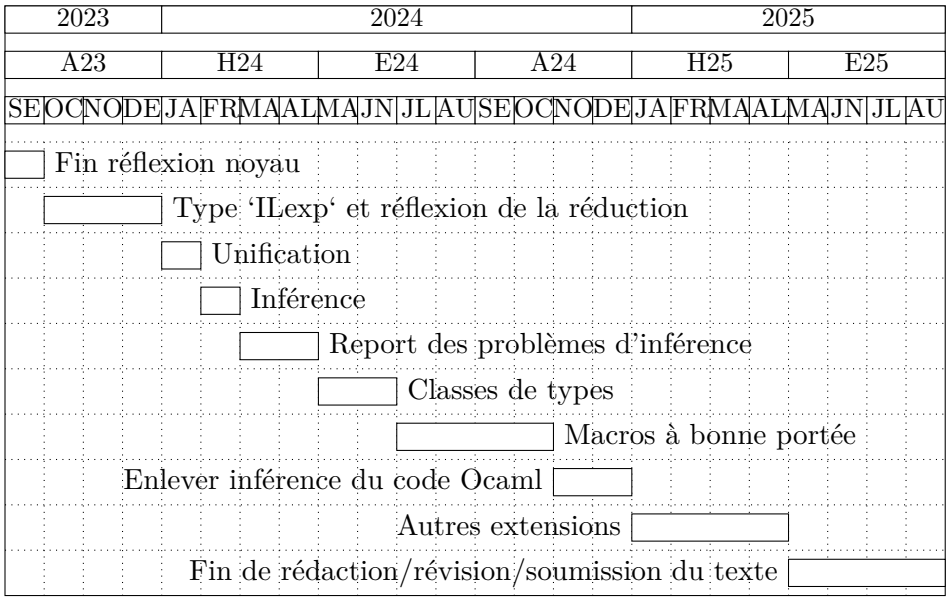

Fig. 1 : Diagramme de Gantt pour le reste du projet

Comme indiqué à la section 2, une bonne partie de l'interface de métaprogrammation du langage noyau a déjà été implantée. Puisque cette étape est préalable au reste, on la termine en premier. Les étapes suivantes suivent leur ordre logique, sauf pour la contribution auxiliaire qui pourra être développée en parallèle des bibliothèques d'élaboration. Il sera éventuellement

nécessaire d'adapter celles-ci, mais il semble préférable de ne pas retarder le développement principal. Un peu de temps est réservé dans l'échéancier pour explorer d'autres extensions d'élaboration qui pourront servir d'évaluation à notre système.

7 Conclusion

Ayant défini l'auto-élaboration et sa relation à l'auto-amorçage, on a fixé l'objectif de la thèse, soit d'implanter l'inférence de types et les arguments d'instances par des macros Typer plutôt que leur code Ocaml actuel. On a décrit ce qui a été fait dans cette direction – l'ajout de macros sans résidus par réflexion – et puis ébauché et planifié ce qu'il reste à travailler – l'interface et l'implantation des bibliothèques prévues.

Bibliographie

Barrett, Langston, David Thrane Christiansen, and Samuel Gélineau. 2020. "Predictable macros for hindley-milner (extended abstract)." http://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf.

Barszcz, Jean-Alexandre. 2021. "Typer a de la classe : Le polymorphisme ad hoc dans un langage avec des types dépendants et de la métaprogrammation." https://doi.org/1866/26064.

Barzilay, Eli. 2006. "Implementing direct reflection in nuprl."

Bird, Richard S., and Ross Paterson. 1999. "De bruijn notation as a nested datatype." *Journal of functional programming* 9 (1) : 77–91. https://doi.org/10.1017/S0956796899003366.

Brady, Edwin. 2021a. "Idris 2 : Quantitative Type Theory in Practice." In *35th european conference on object-oriented programming (ecoop 2021)*, edited by Anders Møller and Manu Sridharan, 194 :9 :1–9 :26. Leibniz international proceedings in informatics (lipics). Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9.

———. 2021b. "Idris 2 : Quantitative Type Theory in Practice (Artifact)." Edited by Edwin Brady. *Dagstuhl artifacts series* 7 (2) : 10 :1–10 :7. https://doi.org/10.4230/DARTS.7.2.10.

Bratman, Harvey. 1961. "A alternate form of the 'uncol diagram'." *Commun. acm* 4 (3) : 142. https://doi.org/10.1145/366199.366249.

Chang, Stephen, Michael Ballantyne, Milo Turner, and William J. Bowman. 2019. "Dependent type systems as macros." *Proc. acm program. lang.* 4 (POPL). https://doi.org/10.1145/3371071.

Christiansen, David, and Edwin Brady. 2016. "Elaborator Reflection : Extending Idris in Idris." In *Proceedings of the 21st acm sigplan international conference on functional programming*, 284–97. Icfp 2016. Nara, Japan : Association for Computing Machinery. https://doi.org/10.1145/2951913.2951932.

Christiansen, David Raymond. 2016. "Practical Reflection and Metaprogramming for Dependent Types." IT-Universitetet i København.

Clinger, William D., and Mitchell Wand. 2020. "Hygienic macro technology." *Proc. acm program. lang.* 4 (HOPL). https://doi.org/10.1145/3386330.

Culpepper, Ryan, and Matthias Felleisen. 2010. "Fortifying macros." In *Proceedings of the 15th acm sigplan international conference on functional programming*, 235–46. Icfp '10. Baltimore, Maryland, USA : ACM.

https://doi.org/10.1145/1863543.1863577.

Devriese, Dominique, and Frank Piessens. 2011. "On the bright side of type classes : Instance arguments in agda." *Sigplan not.* 46 (9) : 143–55. https://doi.org/10.1145/2034574.2034796.

———. 2013. "Typed syntactic meta-programming." In *Proceedings of the 18th acm sigplan international conference on functional programming*, 73–86. Icfp '13. Boston, Massachusetts, USA : Association for Computing Machinery. https://doi.org/10.1145/2500365.2500575.

Earley, Jay, and Howard Sturgis. 1970. "A formalism for translator interactions." *Commun. acm* 13 (10) : 607–17. https://doi.org/10.1145/355598.362740.

Flatt, Matthew, and PLT. 2023. "The racket reference v8.10." https://docs.racket-lang.org/reference/index.html.

Ganz, Steven E., Amr Sabry, and Walid Taha. 2001. "Macros as multi-stage computations : Type-safe, generative, binding macros in macroml." In *Proceedings of the sixth acm sigplan international conference on functional programming*, 74–85. Icfp '01. Florence, Italy : Association for Computing Machinery. https://doi.org/10.1145/507635.507646.

Henniger, Simon, and Nada Amin. 2023. "The Dolorem Pattern : Growing a Language Through Compile-Time Function Execution." In *37th european conference on object-oriented programming (ecoop 2023)*, edited by Karim Ali and Guido Salvaneschi, 263 :41 :1–41 :27. Leibniz international proceedings in informatics (lipics). Dagstuhl, Germany : Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2023.41.

Herman, David, and Mitchell Wand. 2008. "A theory of hygienic macros." In *Programming languages and systems*, edited by Sophia Drossopoulou, 48–62. Berlin, Heidelberg : Springer Berlin Heidelberg.

Korkut, Joomy. 2019. "Direct reflection for free !" https://www.cs.princeton.edu/~ckorkut/papers/icfp-src-19-reflection.pdf.

Kudasov, Nikolai. 2022. "Functional Pearl : Dependent type inference via free higher-order unification." *Arxiv e-prints*, April, arXiv :2204.05653. https://doi.org/10.48550/arXiv.2204.05653.

Lecarme, Olivier, Mireille Pellissier, and Marie-Claude Thomas. 1982. "Computer-aided production of language implementation systems : A review and classification." *Software : Practice and experience* 12 (9) : 785–824. https://doi.org/https://doi.org/10.1002/spe.4380120902.

Liang, Sheng, Paul Hudak, and Mark Jones. 1995. "Monad transformers and modular interpreters." In *Proceedings of the 22nd acm sigplan-sigact symposium on principles of programming languages*, 333–43. Popl '95. San

Francisco, California, USA : ACM. https://doi.org/10.1145/199448.199528.

Lilis, Yannis, and Anthony Savidis. 2019. “A survey of metaprogramming languages.” *Acm comput. surv.* 52 (6) : 113 :1–113 :39. https://doi.org/10.1145/3354584.

McBride, Conor Thomas. 2014. “How to keep your neighbours in order.” In *Proceedings of the 19th acm sigplan international conference on functional programming*, 297–309. Icfp ’14. Gothenburg, Sweden : Association for Computing Machinery. https://doi.org/10.1145/2628136.2628163.

Monnier, Stefan. 2019. “Typer : Ml boosted with type theory and scheme.” *Journées francophones des langages applicatifs*, 193–208. http://www.iro.umontreal.ca/~monnier/typer-jfla2019.pdf.

Ringer, Talia, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. “Qed at large : A survey of engineering of formally verified software.” *Foundations and trends® in programming languages* 5 (2-3) : 102–281. https://doi.org/10.1561/2500000045.

Taha, Walid, and Patricia Johann. 2003. “Staged notational definitions.” In *Generative programming and component engineering*, edited by Frank Pfenning and Yannis Smaragdakis, 97–116. Berlin, Heidelberg : Springer Berlin Heidelberg.

The Agda Team. 2021. “Agda User Manual : Release 2.6.1.3.” https://agda.readthedocs.io/en/v2.6.1.3/index.html.

Tobin-Hochstadt, Sam, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. “Languages as libraries.” In *Proceedings of the 32nd acm sigplan conference on programming language design and implementation*, 132–41. Pldi ’11. San Jose, California, USA : Association for Computing Machinery. https://doi.org/10.1145/1993498.1993514.

Tobin-Hochstadt, Sam, Vincent St-Amour, Eric Dobson, and AsumutitleThe Typed Racket Guide v8.10 Takikawa. 2023. https://docs.racket-lang.org/ts-guide/.

Ullrich, Sebastian, and Leonardo de Moura. 2022. “Beyond Notations : Hygienic Macro Expansion for Theorem Proving Languages.” *Logical Methods in Computer Science* Volume 18, Issue 2 (April). https://doi.org/10.46298/lmcs-18(2:1)2022.

Wadler, P., and S. Blott. 1989. “How to make ad-hoc polymorphism less ad hoc.” In *Proceedings of the 16th acm sigplan-sigact symposium on principles of programming languages*, 60–76. Popl ’89. Austin, Texas, USA : ACM. https://doi.org/10.1145/75277.75283.