Université de Montréal

Amélioration des messages d'erreurs Typer par Algorithme Génétique

par

Ismaïla Fall

Département de mathématiques et de statistique
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

April 28, 2023

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Amélioration des messages d'erreurs Typer par Algorithme Génétique

présenté par

Ismaïla Fall

a été évalué par un jury composé des personnes suivantes :

Jean-Yves Potvin

(président-rapporteur)

Stefan Monnier

(directeur de recherche)

Marc Feeley

(membre du jury)

Résumé

Un défi majeur pour les programmeurs, en particulier pour les novices, est de comprendre les messages d'erreurs émis par le compilateur. Nous nous intéresserons au problème d'affichage de bon message d'erreur de compilation. Dans certains langages, tels que Typer, la vérification du type des expressions est faite lors de la compilation; ce qui oblige le compilateur à déduire les types de certaines ou de toutes les expressions; mais aussi d'envisager la meilleure manière d'écrire le type (dans le langage source) dans un message d'erreur (ce qui est infaisable pour le moment dans Typer). Cependant l'interprétation du type des expressions faite par le compilateur est toujours différente de ce que l'utilisateur aimerait voir en cas d'erreur de compilation. En effet, lorsque le code source est converti en une représentation interne via une fonction complexe (appelée '`elaborate`'), il peut être difficile de trouver une correspondance entre le type "t_source" (type du code source) et le type "t_interne" (type de la représentation interne du code source) en cas d'erreur. Parfois, "t_source" peut ne pas être disponible ou même n'avoir jamais existé car "t_interne" a été créé de toute pièce par inférence de type. Il peut donc être difficile de trouver un "t_source" correspondant, d'autant plus qu'il doit être clair et compréhensible pour le programmeur. En d'autres termes, il n'existe pas d'algorithme déterministe permettant de trouver une représentation naturelle dans le code source correspondant à la représentation interne d'un type. D'où l'importance d'implémenter un système heuristique tel que les algorithmes génétiques ou les réseaux de neurones qui nous donne cette information, permettant ainsi une meilleure affichage du texte des messages d'erreurs. Nous avons donc décidé de travailler sur l'amélioration des messages d'erreur du compilateur Typer, dans sa phase de traduction du langage (interprétation et représentation des différentes expressions dans le langage source) en proposant une approche basée sur les algorithmes génétiques.

Mots Clés: Message d'erreur, Typer, Algorithme génétique, Compilateur

Abstract

A major challenge for programmers, especially for novices, is to understand the error messages issued by the compiler. We are interested in the problem of displaying correct compiler error messages. In some languages, such as Typer, the type checking of expressions is done at compile time; this forces the compiler to deduce the types of some or all expressions; but also to consider the best way to write the type (in the source language) in an error message (which is unfeasible for the moment in Typer). However, the interpretation of the type of expressions made by the compiler is always different from what the user would like to see in case of a compilation error. Indeed, when the source code is converted into an internal representation via a complex function (called '`elaborate`'), it can be difficult to find a correspondence between the type "t_source" (type of the source code) and the type "t_interne" (type of the internal representation of the source code) in case of error. Sometimes, "t_source" may not be available or even have never existed because "t_interne" was created from scratch by type inference. It can therefore be difficult to find a corresponding "source_t", especially since it must be clear and understandable for the programmer. In other words, there is no deterministic algorithm to find a natural representation in the source code corresponding to the internal representation of a type. Hence the importance of implementing a heuristic system such as genetic algorithms or neural networks that gives us this information; thus allowing a better display of the text of error messages. We therefore decided to work on the improvement of the error messages of the Typer compiler, in its language translation phase (interpretation and representation of the different expressions in the source language) by proposing an approach based on genetic algorithms.
Keywords: Error message, Typer, Genetic algorithm, Compiler

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et des abréviations

AG	Algorithme génétique, de l'anglais *genetic algorithm*
DSL	Langages spécifiques à un domaine, de l'anglais *Domain-Specific Languages*
EDSL	Langages spécifiques à un domaine intégrés, de l'anglais *Embedded Domain-Specific Languages*
OPG	Grammaire de Précédence d'Opérateurs, de l'anglais *Operator Precedence Grammar*
ML	Méta-langage, de l'anglais *Meta Language*

Remerciements

Je souhaite exprimer ma gratitude envers Dieu et plusieurs personnes qui ont contribué à ma réussite dans la réalisation de ce travail. Tout d'abord, je remercie Dieu pour Sa bénédiction et Sa guidance tout au long de ce parcours. Je suis également reconnaissant envers Cheikhouna Ahmadou Bamba wa Khadimou Rassoulihi pour Sa lumière et Sa sagesse qui m'ont inspiré tout au long de ma vie.

Un grand merci à mes chers parents pour leur amour, leur soutien, leur éducation et les sacrifices qu'ils ont faits pour moi. Je vous dédie ce travail en signe de mon amour et de ma gratitude. Je tiens également à remercier mes mamans, ma grand-mère, mes frères et soeurs, ainsi que toute ma famille pour leur amour et soutien.

Je suis reconnaissant envers mon directeur de recherche pour sa patience, son encadrement et ses précieux conseils tout au long de ce projet. Enfin, je remercie mes amis pour le soutien moral et leur encouragement tout au long de ce parcours.

Chapitre 1

Introduction

1.1. Contexte

Typer est un langage de programmation typé statiquement qui appartient à la famille ML. Son système de type est en grande partie inspiré du système de type Hindley-Milner [**12**]. Ce système de type, combiné avec les macros expansions forment la partie élaboration (compilateur) de Typer.

Le système de type Hindley-Milner permet à Typer de réduire la surcharge de typage explicite grâce à une certaine forme d'inférence de type. Cela augmente non seulement la productivité du programmeur, mais fournit également une sécurité de type au programmeur. Cependant, il n'est pas sans inconvénients. Le signalement des messages d'erreur de type devient plus difficile, car bien que le compilateur indique précisément l'endroit où l'incohérence a été détectée, celle-ci peut être très éloignée de la source de l'erreur. De plus, il arrive aussi que les types inférés ne correspondent pas aux types souhaités.

Typer peut être utilisé comme un outil pour écrire des preuves, où les types jouent le rôle de propositions et le code joue le rôle de preuve de ces propositions, conformément à la correspondance de Curry-Howard [**5**]. En effet, les types dans Typer peuvent être complexes et peuvent contenir du code, mais ce code n'est généralement pas exécuté. Il est vérifié par le compilateur en tant que preuve valide en demandant la vérification du type du code. Ainsi, les messages d'erreurs de typage dans Typer revêtent une importance particulière, car ils permettent de vérifier la validité des preuves et de faciliter le processus de débogage.

Malheureusement, dans l'état actuel, les messages d'erreurs dans Typer sont très difficiles à comprendre. C'est dans cette optique que nous avons été chargés de proposer un système

heuristique visant à améliorer les messages d'erreurs de type du compilateur Typer, afin de rendre leur compréhension plus accessible aux programmeurs qui interagissent avec le compilateur.

Dans le cadre de ce travail, nous ne nous concentrerons que sur les types simples, par opposition aux types dépendants où le type d'une valeur peut dépendre de la valeur elle-même ou d'une autre valeur. Nous avons choisi cette approche car les types simples sont suffisants pour illustrer les problèmes et concepts abordés. Toutefois, cela ne constitue pas une limitation de notre travail, car Typer offre également la possibilité d'utiliser les types dépendants pour des cas plus avancés de vérification de types et de preuves formelles.

1.2. Problématique

Les problèmes liés au compilateur Typer sont nombreux et variés. Parmi ces problèmes, on retrouve principalement l'incompréhension des messages d'erreurs de type. Pour mieux illustrer ce fait, prenons l'exemple de la définition du type des listes chaînées:

```
type List (a : Type)
    | nil
    | cons (hd : a) (tl : List a);
```

Dans cet extrait, nous avons une fonction '`List`' qui prend en argument '`a`' qui est le type des éléments de la liste, et renvoie un nouveau type. Ce "nouveau" type est décrit comme étant formé soit de la constante *nil*, c'est-à-dire la liste vide; soit d'une valeur de la forme '`cons hd tl`', où *cons* est un tag qui permet de reconnaître cette valeur, **hd** est la valeur du premier élément de la liste, et **tl** est une référence qui pointe vers le reste de la liste.
Prenons maintenant l'exemple de la déclaration de variable suivante:

```
x = cons 1 nil;
```

Nous avons dans '`x`' une liste qui contient juste un entier (le nombre 1). '`x`' est de type '`List Int`', qui est la représentation que le programmeur aimerait voir dans Typer, mais au contraire, il aura à la place un type qui est plutôt représenté par une syntaxe comme suit :

```
typecons (nil)(cons (hd : ##Int)(tl : List ##Int))
```

Cette représentation difficile à comprendre pour le programmeur est le résultat de l'inférence de type, mais aussi de l'expansion de la macro "type" utilisée dans la définition de '`List`' pendant l'élaboration du code source. En réalité, lors de l'expansion de macro, la définition de type '`List`' est transformée en une définition de fonction dépendante de type, avec une signature de type '`Type -> Type`', qui prend un argument de type '`a`'. Ainsi, nous obtenons

```
List : Type -> Type;
List (a : Type) = typecons (nil) (cons hd : a) (tl : List a);
```

Fig. 1.1. Définition complète du type des listes chaînées

```
typecons (nil)(cons (hd : ##Int)(tl : List ##Int))
```

Fig. 1.2. Représentation du type des listes chaînées

la définition complète du type des listes chaînées, qui est illustrée dans la figure 1.1. Dans cette définition, le type '`List`' prend un argument de type '`a`' et renvoie un type '`Type`', représentant une liste paramétrée par le type '`a`'.

Lorsque la déclaration de variable '`x = cons 1 nil`' est analysée par l'inférence de type, elle déduit que '`1`' a le type '`##Int`' (où simplement '`Int`' grâce à une définition préalable de '`Int`' comme étant équivalent à '`##Int`' dans le contexte initial). Par conséquent, '`cons 1 nil`' a non seulement le type '`List Int`', mais aussi le type '`List ##Int`'.

Notons également que la définition complète du type des listes chaînées (figure 1.1) peut être "reduite" lors de l'inférence de type en évaluant l'appel de fonction '`List`' et en substituant '`##Int`' dans son corps, ce qui donne précisément la représentation illustrée dans la figure 1.2.

Il est important de noter que le résultat de l'élaboration diffère toujours du code source d'origine, car il est créé de toute pièce par le compilateur en déduisant les types des éléments du programme en se basant sur les règles de typage du langage et des informations disponibles dans le code lui-même. Au pire la bonne représentation qu'on aimerait utiliser (comme dans notre exemple '`List Int`') n'est souvent présente nulle part dans le code source. Par conséquent, la question est de savoir comment transformer la représentation du type des listes chaînées (figure 1.2) en une représentation similaire à celle de '`List Int`' ?

Dans l'exemple de '`List Int`', il convient de souligner que '`List`' est une fonction au niveau des types. Lorsque nous utilisons la représentation du type des listes chaînées, nous obtenons la valeur renvoyée par cet appel de fonction. Cette valeur peut également être reproduite en effectuant simplement un autre appel à la fonction '`List`' avec les mêmes arguments, c'est-à-dire '`List Int`'.

Prenons un autre exemple pour illustrer ce point :

```
Vector : Type -> Nat -> Type;
Vector t n
    = if n == 0 then Unit
      else Pair t (Vector t (n - 1));
```

Ici, nous avons une fonction '`Vector`' qui prend deux arguments : '`t`', qui représente le type des éléments, et '`n`', qui représente la longueur de la liste. Si, '`n`' est égal à 0, alors le type renvoyé est '`Unit`', qui représente une liste vide. Sinon, le type renvoyé est une paire ('`Pair`') composée d'un élément de type '`t`' et d'une liste de type '`Vector t (n - 1)`'.
Ainsi, lorsque nous analysons l'expression suivante :

```
Pair Int (Pair Int Unit)
```

L'expression '`Pair Int (Pair Int Unit)`' peut être interprétée comme étant équivalente au type de l'expression '`Vector Int 2`', qui représente une liste contenant deux valeurs de type '`Int`'.

Ces exemples mettent en évidence la possibilité que les types générés par le compilateur Typer soient des valeurs produites par des appels de fonctions au niveau des types. Par conséquent, pour comprendre et simplifier une représentation complexe, il est important de reconnaître que le type complexe en question peut potentiellement résulter d'un ou de plusieurs appels de fonction, tel que '`List Int`' ou '`Vector t 2`'. Il est essentiel de comprendre que ces valeurs de type peuvent être reproduites simplement en utilisant des appels de fonction similaires avec les arguments appropriés. De plus, notez également que dans ce contexte, il n'est pas nécessaire de trouver un appel de fonction original spécifique, mais plutôt de créer un appel de fonction qui correspond à la structure et aux schémas du type recherché.

Il convient également de noter que les deux exemples donnés ci-dessus ne sont que quelques-uns parmi de nombreux autres problèmes liés au signalement des messages d'erreurs de type dans le compilateur Typer.

1.3. Contributions

Rappelons que lors de l'inférence de type, il est possible que des macros soient étendues, entraînant ainsi l'évaluation d'une partie du code source. De plus, des substitutions peuvent également avoir lieu, par exemple, en remplaçant '`t`' par '`Int`' dans l'expression '`List t`'. Cependant, il n'est pas toujours possible de retracer les étapes d'évaluation à partir du résultat obtenu jusqu'au code source initial, y compris la décomposition des macros en code

source généré.

En réalité, le problème auquel nous sommes confrontés est que nous avons '`lexp = elab (sexp)`', où nous connaissons '`lexp`' et nous cherchons à retrouver '`sexp`'. Cependant, '`elab`' peut effectuer virtuellement n'importe quelle opération, ce qui rend la récupération du code source à partir du résultat de l'inférence de type complexe difficile, voire impossible, dans certains cas.
Nos contributions consistent donc à :

- Améliorer la manière dont l'information d'origine (c'est-à-dire le code source de l'utilisateur) est préservée. Pour cela, nous jugeons important de conserver directement dans le code élaboré le code source à partir duquel il a été généré, plutôt que de simplement stocker des informations de localisation ("ligne+colonne"). Étant donné que cette information n'est pas directement disponible dans le nœud d'appel de fonction, nous l'avons ajoutée à ce dernier. Ces changements commencent principalement à partir de l'arbre abstrait du compilateur, afin de permettre une propagation dans presque toutes les phases du processus de compilation.

- Implémenter un système qui devine de manière heuristique le code "source" ou propose un code "en sortie" aussi proche que possible de ce que le programmeur aurait pu écrire dans le code "source". Ce système se divise en outre en plusieurs composantes notamment:

  - Système de vérification : indique si le code qu'on a deviné est valide et fidèle dans le contexte où l'erreur est signalée. Il s'agit d'une étape essentielle pour s'assurer que le code imprimé est cohérent avec le code source d'origine et que les informations fournies dans le message d'erreur sont claires, précises et pertinentes.

  - Système de traçage : vise à incorporer directement dans le code généré des informations sur le code source d'origine, permettant ainsi de retracer le cheminement du code généré jusqu'au code source d'origine. Cela dépend, entre autres, des modifications apportées au compilateur. Pour un code en sortie, il est généralement impossible d'effectuer l'opération inverse d'inférence de type pour retrouver directement le code source d'origine, mais ce système peut être utilisé pour approximer cette représentation en fonction des informations disponibles dans le code généré.

  - Système de sauvegarde : à l'aide d'une table de hachage, nous stockons en mémoire les résultats d'élaboration de chaque type pour une utilisation ultérieure.

– Système d'algorithme génétique : utilisé de manière systématique pour améliorer les résultats, même lorsque les autres systèmes fournissent des résultats valides selon notre système de vérification. L'étape génétique consiste à générer des candidats aléatoires en complément des solutions proposées par les autres systèmes, dans le but d'obtenir une représentation plus pertinente et de meilleure qualité du code source.

Le système heuristique devrait permettre au compilateur Typer de fournir des messages d'erreurs plus clairs et précis en proposant une solution alternative identique ou aussi proche que possible de ce que le programmeur aurait pu écrire dans le code "source".

1.4. Structure du mémoire

Dans le but de mettre en avant le travail réalisé, nous structurerons ce mémoire comme suit:

(1) **État de l'art** : Dans ce chapitre, il s'agit de faire une synthèse des travaux liés à notre projet d'étude, afin de dresser un panorama des recherches existantes dans le domaine.

(2) **Généralités** : Pour ce chapitre, nous présenterons plus précisément Typer et son processus de compilation. Ensuite, nous aborderons en détail les algorithmes génétiques, qui constituent la méthode principale de notre système heuristique.

(3) **Implémentation de la solution** : Ici, il sera question d'exposer la mise en oeuvre de la solution, en commençant par exposer l'idée de base sur laquelle repose notre système heuristique. Nous détaillerons ensuite les étapes de l'implémentation des différentes approches.

(4) **Évaluation des résultats** : Pour ce chapitre, nous procéderons à une évaluation des résultats obtenus à partir de notre solution, avant de finir par une analyse des performances de notre système heuristique.

(5) **Conclusion** : Et pour terminer par ce chapitre, nous présenterons nos travaux ainsi que les perspectives de travaux futurs.

Chapitre 2

État de l'art

Ce chapitre met l'accent sur l'état de l'art des erreurs de compilation, en synthétisant quelques approches qui ont été utilisées pour rendre les messages d'erreur plus compréhensibles. Les approches relèvent des domaines des macros et de l'inférence de type. Nous présenterons également brièvement un travail en rapport avec la personnalisation des messages d'erreurs. Enfin, nous ferons une brève présentation d'autres travaux en rapport avec les algorithmes génétiques.

2.1. Macros

Un langage de programmation avec un système de type statique assure au programmeur l'exactitude de son programme en permettant le rejet par le compilateur des programmes mal typés. Cependant, l'écriture de programmes typés de manière dépendante nécessite une expertise importante. Pour réduire cette charge d'implémentation des langages à types dépendants, une approche possible serait l'utilisation des macros. En effet, elles permettent au programmeur d'étendre facilement le langage avec de nouveaux types d'expressions (abstractions ou extensions du langage), faisant abstraction des modèles syntaxiques récurrents. Cependant, l'utilisation de macros peut poser des problèmes de débogage aux programmeurs, quel que soit leur niveau d'expérience, ainsi que des problèmes de typage en générant du code non conforme aux règles du langage de programmation. Cependant, dans le cas des langages fonctionnels, ces problèmes peuvent être plus importants, car ces langages ont souvent des systèmes de typage très stricts pour garantir la sécurité des programmes. Les macros peuvent ainsi potentiellement violer ces règles de typage et rendre le débogage plus complexe.
Voici un exemple concret d'une erreur de compilation dans LaTeX qui donne un message d'erreur peu intuitif pour l'utilisateur:

```
\begin{displaymath}
    2 + 2 = 5   \cite{LargeIntegers}
\end{displaymath}
```

L'erreur générée par LaTeX serait quelque chose comme :

```
!LaTeX Error : Command \unshape invalid in math mode.

See the LaTeX manual or LaTeX Companion for explanation.
Type H<return> for immediate help.

...

l.13      2 + 2 = 5 \cite{LargeIntegers}
```

Pourtant, il n'y a pas de commande *\unshape* dans le code de l'utilisateur, ce qui suggère que cette erreur est probablement liée à une macro personnalisée utilisée dans le document LaTeX. L'utilisateur peut ne pas connaître l'existence de cette macro ou comprendre son fonctionnement.
Cet exemple met en évidence la manière dont les erreurs de compilation liées aux macros peuvent être déroutantes et peu claires pour les utilisateurs, soulignant ainsi la nécessite d'améliorer la compréhensibilité des messages d'erreur, notamment lorsqu'ils sont liés à l'utilisation de macros personnalisées.

Les macros sont également utilisées dans la gestion des tâches de méta-programmation et dans certains compilateurs spécifiques, comme l'illustrent les travaux de Herman et Meunier [**19**] qui ont utilisé les macros pour améliorer l'analyse statique des programmes Scheme. De même Owens et al. [**30**] ont ajouté une bibliothèque de générateurs d'analyseurs syntaxiques à Scheme en utilisant des macros.

Des recherches ont été entreprises dans la spécification des macros, dont une approche a commencé avec la vérification statique de la structure syntaxique, initiée par Culpepper et Felleisen [**10**]. Cette approche a depuis évolué pour englober l'expansion hygiénique des macros (Herman et Wand [**20**]), qui garantit que les noms des variables générées par la macro ne vont plus entrer en conflit avec ceux du code appelant.

Culpepper et Felleisen [**11**] ont décrit un système de macros qui permet de créer des macros robustes à partir de spécifications étonnamment simples. Ce système permet de détecter les

erreurs de manière précoce (lorsque l'on dispose du code source), c'est-à-dire avant l'expansion de macro. Dans leur travail, les macros utilisent des annotations pour choisir le modèle approprié pendant la validation syntaxique. Chaque annotation conserve ces capacités de rapport d'erreurs pendant le processus d'analyse. Cela permet de corriger les erreurs avant que le code soit compilé, ce qui peut faire gagner du temps et réduire les risques d'erreurs. De plus, cela permet de faire le suivi et le classement des erreurs. Si le processus de mise en correspondance est rejeté, l'algorithme de correspondance choisira l'erreur la plus appropriée à signaler, puis décrira l'erreur en identifiant le terme défectueux.

2.2. Inférence de type

Le problème que rencontrent de nombreux programmeurs survient lorsque leur programme est rejeté par le compilateur, et le processus de déblocage peut être fastidieux. Certaines erreurs ne sont signalées que tardivement, ce qui rend la localisation de l'erreur souvent inexacte. Dans le pire des cas, le compilateur peut renvoyer non pas une, mais plusieurs positions d'erreur possibles. C'est dans cette optique que plusieurs techniques ont été développées pour générer des messages d'erreurs compréhensibles pour le programmeur dans les langages typés.
Supposons que nous avons le code suivant en Ocaml :

```
let x = 42
let y = String.concat "Number of ducks: " x
```

Dans cet exemple, nous tentons de concaténer une chaîne de caractères avec un entier '`x`'. Si l'inférence de type commence par la ligne '`let x = 42`', elle déduira que '`x`' est de type '`Int`'. Lorsque la ligne suivante sera examinée, une erreur de type sera détectée, car '`String.concat`' attend une liste de chaînes de caractères ('`string list`') et non un entier ('`Int`'). Cependant, si l'inférence de type examine d'abord la ligne '`String.concat`', elle déduira que '`x`' doit avoir le type '`string list`'. Ainsi, une erreur sera signalée à la première ligne ('`let x = 42`'), '`42`' n'est pas une liste de chaînes de caractères et le message peut indiquer '`42 n'est pas une string list`'. Dans les deux cas, l'erreur de type sera détectée à la compilation, mais l'ordre dans lequel les lignes sont examinées peut influencer la manière dont l'erreur est signalée.

L'une des premières propositions a été présentée dans un article publié en 1987 par Wand [**22**]. L'idée était de générer une explication probable en conservant une trace des raisons des déductions de type lors de l'inférence de type.

L'article de Chitil [**9**] propose une approche combinée visant à améliorer la compréhension et la résolution des erreurs de type. Cette approche repose sur deux composantes principales: l'explication compositionnelle des types et le débogage algorithmique des erreurs de type. L'explication compositionnelle des types fournit des explications claires sur la structure et la relation des types. Elle permet aux développeurs de mieux comprendre les erreurs de types en offrant des explications détaillées sur les raisons de ces erreurs. Le débogage algorithmique des erreurs de type propose des algorithmes et des techniques pour localiser et résoudre automatiquement ces erreurs. Ces techniques permettent d'automatiser une partie du processus de débogage, en identifiant les erreurs de type spécifiques et en proposant des solutions potentielles pour les résoudre. En combinant ces deux approches, les développeurs bénéficient d'explications détaillées sur les erreurs de type tout en ayant accès à des méthodes de débogage automatisées, améliorant ainsi l'efficacité du processus de développement et la qualité du code.

Le travail de Beaven et Stansifer [**6**] propose une approche de débogage des erreurs de types basée sur l'inférence de type. Leur contribution réside dans une modification spécifique de l'algorithme d'unification utilisé lors de l'inférence de type. Cette modification permet d'enregistrer les substitutions effectuées pendant le processus d'inférence de type, ce qui facilite l'analyse et la compréhension des erreurs de type. Cette approche utilise deux fonctions principales, la fonction '`Why`' et la fonction '`How`'. La fonction '`Why`' se sert des informations de l'arbre de syntaxe pour répondre à la question de savoir pourquoi cette expression a reçu le type qui lui a été attribué. La fonction '`How`', d'autre part, utilise les liaisons entre les différentes parties de l'expression pour tenter de répondre à la question de savoir pourquoi une variable de type s'est vu attribuer un certain type.

Le concept du découpage d'erreur [**16**] introduit par Haack et Well a été l'une des premières méthodes de rapport d'erreurs basées sur des contraintes. Il consiste à signaler plusieurs emplacements d'erreur; parce que le véritable emplacement d'une erreur peut dépendre de la sémantique souhaitée par le programmeur. Cette même idée est reprise par Chen et Erwig [**8**] à l'aide d'une technique connue sous le nom de *typage contrefactuel*. L'idée est d'effectuer une vérification de type après avoir examiné les expressions atomiques avant l'exécution du programme.

Plusieurs méthodes ont été développées pour améliorer la vérification de type, en plus de l'algorithme W [**26**], dans le domaine de la compréhension des erreurs de type. Parmi ces méthodes, on retrouve l'algorithme M [**23**], qui est similaire à l'algorithme W mais transfère

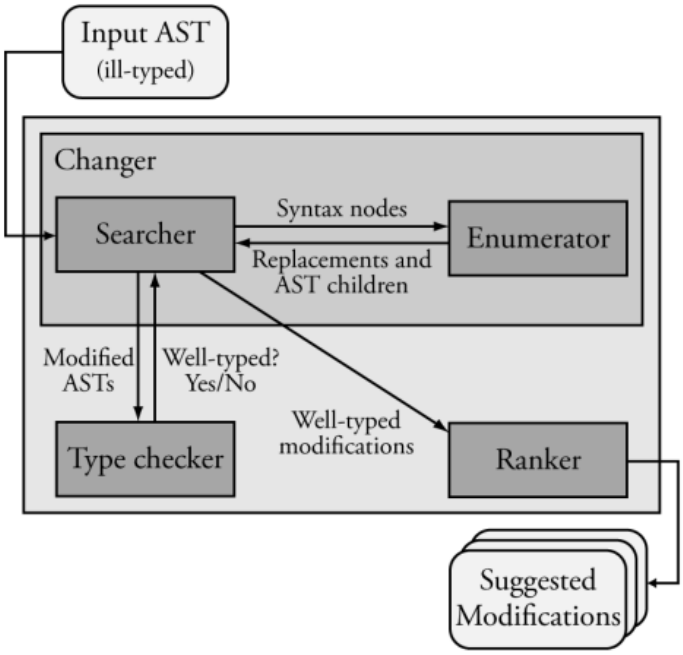

Fig. 2.1. Architecture Compilateur (extrait Lerner et al. 2007)

plus d'informations de type vers le bas lors de la traversée de l'expression, ce qui lui permet de trouver des conflits de type plus tôt que W. On retrouve également l'algorithme G [**27**], qui est une extension de M et utilise un système de sous-typage pour permettre la déduction de types plus expressifs. Toutefois, ces méthodes souffrent en principe d'un problème de biais lié à l'ordre dans lequel les substitutions sont construites et raffinées. Une autre façon de contourner ce problème est d'afficher tous les emplacements à partir desquels les erreurs peuvent provenir. Cette approche, appelée découpage, introduite par Haack et Well et également utilisée par Chen et Erwig, a été présentée au paragraphe précédent.

L'article de Lerner et al [**24**] aborde les problèmes liés à la génération de messages d'erreur de type dans les systèmes de typage avancés qui utilisent l'inférence de type pour réduire la charge liée à la spécification explicite des types. Les auteurs proposent une nouvelle approche de construction des compilateurs qui utilise un oracle pour rechercher des programmes similaires susceptible d'effectuer une vérification de type. Cette approche vise à améliorer la précision des messages d'erreur de type en fournissant des informations plus spécifiques, plutôt que de se contenter des messages d'erreur imprécis du compilateur. La figure 2.1 illustre l'architecture du compilateur proposée par Lerner et al [**24**].

Cette architecture comprend plusieurs composants. Le premier composant, appelé "changer" est responsable des modifications apportées au programme en remplaçant certaines expressions par d'autres. Ensuite, le vérificateur de type existant est utilisé pour vérifier si les

programmes modifiés sont corrects au niveau du typage. Les résultats de cette vérification sont transmis au composant "ranker", qui classe les programmes modifiés en fonction de certains critères. Enfin, les messages d'erreur de type améliorés sont générés en mettant en évidence les modifications spécifiques effectuées.

Dans sa thèse de doctorat [**18**], B.J. Heeren utilise des heuristiques pour suggérer des changements de programme en plusieurs étapes. Parmi ces changements, on retrouve la collection et l'ordonnancement des contraintes de type, ainsi que la résolution de ces contraintes pour décider des types d'expressions.

Une des solutions mises en oeuvre pour résoudre le problème des messages d'erreur de type dans le système Hindley-Milner est la méthode de Wand [**36**]. Cette méthode est basée sur le principe de programmation fonctionnelle selon lequel les fonctions sont des expressions qui retournent une valeur et ne modifient pas l'état global du programme. Elle permet de garder une trace de chaque variable de type dont le fragment de programme force son instanciation, ce qui permet d'améliorer la précision des messages d'erreur de type. La reconstruction de l'algorithme W avec la méthode de Wand conduit à une inférence de type plus précise.

Toujours en ce qui concerne les messages d'erreur de type, nous pouvons citer le travail de Zang et Myers [**37**]. Ils ont mis au point une méthode pour détecter et localiser les erreurs de types dans les programmes Haskell. Leur approche utilise un système de type extensible et un algorithme de résolution de contraintes pour inférer les types des expressions. En comparant les types inférés avec les types attendus, ils peuvent identifier les erreurs de types potentielles. Cette méthode s'est avérée efficace pour réduire le temps de débogage et pour améliorer la qualité du code, mais elle souligne également l'importance de l'analyse statique pour détecter les erreurs avant l'exécution du programme.

2.3. Personnalisation des messages d'erreur de type

Les travaux de recherche cités abordent de manières diverses et complémentaires la résolution des erreurs de type dans les langages fonctionnels. Toutefois, une caractéristique commune à ces travaux est qu'ils considèrent que les types “en soi” sont compréhensibles, et que le problème est de choisir les types à imprimer et les numéros de ligne/colonne à leur associer. En revanche, notre travail se différencie en ne se préoccupant pas des numéros de ligne/colonne, mais plutôt en se concentrant sur la personnalisation des messages d'erreur de type en conservant l'information d'origine pendant le processus d'inférence de type ou

en la stockant pendant l'élaboration des types. Cette approche novatrice, appliquée dans le contexte des langages spécifiques à un domaine intégré peut offrir des solutions nouvelles et complémentaires pour améliorer la compréhension et la résolution des erreurs de type. Il est important de noter que notre travail ne vise pas à proposer une nouvelle méthode d'inférence de type ou de résolution des erreurs de type, mais plutôt à résoudre un autre problème qui n'est pas souvent rencontré dans de nombreux langages en raison de leur système de types trop simples.

Dans sa thèse "*Type Error Customization for Embedded Domain-Specific Languages*", Serrano [**35**] décrit une méthode de personnalisation des messages d'erreur de type pour les langages spécifiques à un domaine (DSL). Lorsqu'ils sont incorporés dans des langages de programmation, en particulier des langages fortement typés tels-que Haskell, ils sont appelés *DSL intégrés* (EDSL). Ces EDSL sont limités à un domaine spécifique et permettent de réutiliser des fonctionnalités comme la génération de code, l'optimisation et la vérification de type pour créer de nouveaux DSL.
L'auteur aborde également la question du coût de leur utilisation en cas d'erreur. En effet, les DSL intégrés sont considérés comme des bibliothèques par le compilateur, de sorte que le message d'erreur résultant de leur utilisation ne reflètent pas souvent leur domaine d'application. Parmi ces messages d'erreurs, nous pouvons distinguer :

- Messages d'erreurs de type générique : ils sont souvent peu utiles pour résoudre les problèmes de typage dans les DSL intégrés, car les types utilisés dans ces DSL intégrés sont souvent spécifiques au domaine. Cela peut rendre les messages d'erreur de typage générique difficile à comprendre pour les programmeurs, rendant ainsi la résolution des problèmes de typage plus complexe.
  Supposons que nous avons un DSL intégré pour la manipulation de listes d'entiers dans Ocaml, avec une fonction pour calculer la somme d'une liste d'entiers. Ci-dessous un exemple de code erroné dans ce DSL:

  ```
  let myList = [1; 2 ; "3"; 4]
  let somList = List.sum myList
  ```

  Ici, nous avons une liste d'entiers '`myList`' qui contient une chaîne de caractères "3". Cette liste viole les contraintes de typage du DSL intégré qui exige une liste d'entiers. Lorsque nous essayons d'appeler la fonction '`List.sum`' pour calculer la somme des entiers dans '`myList`', le EDSL générera probablement un message d'erreur de type générique comme celui-ci :

```
Error : This expression has type string but an expression was expected
of type int
```

Ce message d'erreur générique peut être moins utile pour résoudre le problème dans le contexte du ESDL, car il ne fournit pas d'informations spécifiques au domaine sur la nature de l'erreur. Les types de données dans le EDSL sont spécifiques au domaine (c'est-à-dire les listes d'entiers), et le message d'erreur générique ne donne pas d'indications sur la cause exacte de l'erreur dans ce contexte.
Au contraire, un message d'erreur plus spécifique au domaine, tel que

```
Erreur de typage:
L'élément "3" de la liste doit être un entier, mais il a été trouvé une
chaîne de caractères.
```

- Messages d'erreurs de typage inadapté : ils résultent de l'utilisation de DSL intégrés. Ces messages sont signalés dans un langage différent de celui utilisé par les programmeurs, qui les rendent peu clairs ou incompréhensibles pour ces derniers. Supposons que nous ayons un DSL intégré pour la manipulation d'images :

```
img = load_image (image.jpg)
filter = "gausian" # Erreur de typage : mauvaise valeur attribuée à filter
result = apply_filter (img, filter) # Erreur de typage : mauvais argument
                                                          passé à apply_filter
save_image (result, "img_filter.jpg")
```

Dans cet exemple, le programmeur a fait une faute de frappe en écrivant "gausian" au lieu de "gaussian" dans le nom du filtre, ce qui pourrait entraîner une erreur de typage à l'utilisation de la fonction '`apply_filter`'. Le message d'erreur généré pourrait ressembler à ceci :

```
Erreur de typage : Type de données incompatible
```

Le message d'erreur de typage généré par défaut est générique et ne fournit pas suffisamment d'informations pour comprendre la nature exacte de l'erreur. Dans ce cas, il ne mentionne pas spécifiquement la variable ou la fonction concernée, la valeur

erronée attribuée à la variable '`filter`' ou le mauvais argument passé à la fonction '`apply_filter`'. Cela rend difficile la résolution du problème pour le programmeur. Un exemple d'erreur de typage personnalisé que le programmeur aurait aimé voir serait :

```
La valeur attribuée à 'filter' doit être une chaîne de caractères
représentant le type de filtre (ex: 'gaussian', 'median')
la fonction 'apply_filter' attend un filtre valide en tant que
deuxième argument. Vérifiez la valeur attribuée à 'filter'.
```

- Messages d'erreurs de typage incomplets: ils peuvent être peu précis et difficiles à localiser, rendant ainsi la tâche de les trouver et de les corriger plus difficile pour les programmeurs.
  Par exemple, dans le langage de programmation Coq, qui utilise des arguments implicites, les messages d'erreur de typage peuvent être incomplets. Par défaut, Coq n'affiche pas les arguments implicites dans les messages d'erreurs, ce qui peut entraîner des messages peu clairs. Par exemple, un message d'erreur pourrait être :

  ```
  Cette expression a le type 'Foo x' alors que le contexte attend
  une expression de type 'Foo x'
  ```

  Où le problème réel est lié à des arguments implicites différents entre les occurrences de 'Foo x'. En réalité, les deux occurrences de '`Foo x`' n'ont pas les mêmes arguments implicites, mais cela n'est pas indiqué clairement dans le message d'erreur. Dans un tel cas, la nature exacte de l'erreur et sa localisation peuvent être difficiles à déterminer, rendant la correction du code plus compliquée pour les programmeurs. Il est donc important que les messages d'erreur de typage soient complets, y compris les informations sur les arguments implicites, pour faciliter la détection et la résolution des problèmes de typage dans le code source.

Pour résoudre les problèmes mentionnés ci-dessus, l'auteur propose une approche en trois étapes. Ces trois étapes sont les suivantes :

(1) Définir les règles de typage spécifiques au domaine (DSL) : cette étape permet de définir les règles, les syntaxes et les définitions de types ainsi que les opérations disponibles pour manipuler les éléments du domaine ciblé par le DSL.

(2) Implémenter le système de typage personnalisé : cette étape consiste à créer et à mettre en oeuvre le système de typage adapté à un DSL particulier. Elle implique

d'établir les contraintes et les normes régissant les types de données et les opérations permises dans ce langage. Après cette phase vient la phase l'implémentation du système de typage, qui englobe la création du code et des mécanismes nécessaires pour vérifier et garantir que les programmes écrits dans le DSL sont de types corrects, conformément aux règles de typage définies dans la phase précédente. Elle est absolument nécessaire pour utiliser efficacement le DSL et s'assurer que les programmes écrits dans ce langage sont de type corrects.

(3) Fournir des suggestions de correction : enfin, cette étape consiste à présenter de manière claire et compréhensible au programmeur les erreurs de type. Cela peut inclure l'ajout de détails tels que l'emplacement exact de l'erreur, ainsi que des conseils sur la manière de corriger l'erreur et la règle de validation qui a été violée.

Ces trois étapes permettent de concevoir un système DSL intégré qui aide les programmeurs à comprendre plus facilement et à résoudre rapidement les messages d'erreur de type.

Parmi les travaux présentés précédemment, celui-ci se rapproche le plus de notre travail, car il se concentre sur le choix de la représentation textuelle des types eux-mêmes. Avec les EDSL, les erreurs se produisent souvent au niveau interne (dans un langage peu connu des programmeurs). Ainsi, l'auteur vise à transformer cette représentation incomprise des programmeurs en un langage compréhensible. Cela correspond exactement à ce que nous cherchons à faire dans Typer. En effet, les messages d'erreurs Typer se produisent pendant l'inférence de type, alors la représentation du type à l'origine de l'erreur n'est pas connue de l'utilisateur. C'est pourquoi nous cherchons à faire correspondre ces types en une représentation connue de l'utilisateur.

2.4. Algorithme Génétique

Faisant partie de la famille des algorithmes évolutionnistes, les algorithmes génétiques sont apparus pour la première fois en 1975 dans *Adaptation in Natural and Artificial Systems* [**21**]. Ils sont utilisés dans des domaines tels que l'apprentissage (prédiction, robotique, ...); la programmation automatique (programmes Lisp [**14**] , ...) en raison de leur réputation de fournir de meilleurs résultats que les réseaux de neurones dans des cas de problèmes très complexes. Nous allons présenter quelques travaux qui tentent d'utiliser les algorithmes génétiques pour proposer une solution à un problème donné.

Nous pouvons citer le mémoire de Ahumada Pardo en 2015 [**31**] qui utilise une approche évolutionniste dans le but d'automatiser le processus de génération des requêtes SQL valides

en se basant sur des exemples d'entrée et de sortie fournis par l'utilisateur.

D'autres travaux allant dans ce sens sont ceux de Arcuri et Fraser [**1**] ainsi que Sayyad et al. [**2**]. Dans leur étude sur la configuration des paramètres dans l'apprentissage automatique et l'estimation de l'effort logiciel, un algorithme génétique a été utilisé pour le réglage des paramètres. Les résultats de ces études montrent que le réglage des paramètres a un impact significatif sur la performance des algorithmes, et aussi que le sur-ajustement du réglage des paramètres constitue une limitation importante des études empiriques en génie logiciel.

L'article de Saber et al. [**33**] présente une approche basée sur les algorithmes évolutionnaires pour résoudre le problème de sélection de caractéristiques dans les grandes lignes de produits logiciels. Il met en évidence l'importance de la réparation, c'est-à-dire la capacité à corriger et améliorer les solutions non valides ou incomplètes générées par les algorithmes évolutionnistes. L'objectif est de trouver des solutions efficaces pour sélectionner les caractéristiques dans ces logiciels, en prenant en compte plusieurs objectifs simultanément. Pour relever ce défi, les auteurs proposent des techniques spécifiques de réparation adaptées aux contraintes et objectifs des lignes de produits logiciels. En les combinant avec les algorithmes évolutionnistes, il est possible d'obtenir des résultats significatifs en termes de sélection de caractéristiques dans les lignes de produits logiciels.

Dans l'article [**4**], les auteurs proposent une approche basée sur les algorithmes génétiques pour la combinaison optimale de drapeaux de compilation afin de générer un code exécutable efficace en termes de temps. Les drapeaux de compilation font référence aux différentes options disponibles dans le compilateur GCC (GNU Compiler Collection) pour contrôler les optimisations appliquées lors de la compilation. L'objectif est de générer un code exécutable efficace en sélectionnant les bons drapeaux de compilation.

Les travaux cités utilisent des algorithmes génétiques pour essayer de trouver un morceau de code dont le comportement satisfait au mieux certaines contraintes. En revanche, dans ce travail, nous ne cherchons pas à trouver un code qui se comporte d'une certaine manière, mais simplement un code dont la forme satisfait une certaine contrainte. Ainsi, notre critère d'optimisation diffère du leur et se concentre sur la forme du code généré.

Chapitre 3

Langages et Techniques utilisées

Nous venons de voir quelques notions essentielles pour comprendre notre sujet. Nous avons constaté qu'au cours des cinquante dernières années, les débogages des erreurs de type se sont présentées sous de nombreuses formes. Nous avons jugé utile cette exploration dans la mesure où il nous a permis de prendre connaissance des techniques et méthodes existantes pour améliorer les messages d'erreurs des compilateurs ML.
Dans ce chapitre, nous présenterons Typer en décrivant son compilateur ainsi que son système d'inférence de type. Nous aborderons également les algorithmes génétiques utilisés dans le cadre de notre projet.

3.1. Typer

Typer est une combinaison de langages tels que Lisp, Coq et ML, dont elle tire le meilleur de chacun. En effet, sa structure syntaxique est très similaire à celle de Lisp, son système de type est aussi puissant que celui de Coq; et on le compte parmi les membres de la famille des langages ML (Méta-Langage). Dans l'un de ses articles [**29**], *Monnier* parle de la conception de Typer en ces termes: "Sa conception est généralement conservatrice en ce sens qu'elle utilise principalement des solutions existantes, mais essaie de les rationaliser et de les combiner de manière à simplifier, espérons-le, le système global tout en le rendant plus flexible en même temps". En effet, l'idée derrière la conception de Typer est d'avoir un langage ML aussi élégant que le langage Scheme (de son nom d'origine Schemer) avec un noyau primitif où plusieurs langages peuvent être définis sous forme de bibliothèques superposées.

En plus de partager une grande partie syntaxique, les langages ML ont également en commun plusieurs autres traits fondamentaux. Dans le cas de Typer, on peut constater que :

```
type lexp =
  | Var of vref
  | Lambda of arg_kind * vname * ltype * lexp
  | Call of lexp * (arg_kind * lexp) list
  ...
```

Fig. 3.1. Simplification de l'arbre syntaxique abstrait du compilateur en conservant Var, Lambda et Call

- Son système d'inférence de types et l'expansion de macros est similaire à celui utilisé par Template Haskell.
- Elle combine également un mélange de type statique et de macros de style Lisp, à l'instar du langage Typed Racket.
- Son système de types dépendants est similaire à celui de langages tels qu'Idris, F-Stars et Zombie.

Pottier [**32**] définit l'élaboration comme un processus par lequel le compilateur transforme le code source écrit par le programmeur, en complétant les éléments omis par le programmeur pour créer une représentation explicite du programme. Dans le cas de Typer, l'élaboration consiste à traduire une S-expression (pour '`sexp`') en une lambda-expression (pour '`lexp`'). Ce processus se déroule en plusieurs étapes :

- Terminer l'analyse syntaxique
- Expansion des macros
- Inférence et vérification des types

Pour une meilleure illustration du processus de l'élaboration, prenons une définition basique du type '`lexp`' (figure 3.1).

La définition simplifiée du type '`lexp`' que nous avons présentée dans la figure 3.1 inclut des constructeurs '`Var`', '`Lambda`' et '`Call`'. Ces constructeurs représentent respectivement :

- Lambda : '`arg_kind`' (le type d'argument de la fonction, par exemple, par valeur, par référence), '`vname`' (le nom de la variable de la fonction), '`ltype`' (le type de retour de la fonction) et '`lexp`' (le corps de la fonction).
- Call : '`lexp`' (l'expression de la fonction appelée), et une liste de paires ('`arg_kind * lexp`') qui représente les arguments passés à la fonction et leur type d'argument respectif.

```
type sexp =
    | Node of sexp * sexp list
    | Symbol of symbol
    | Integer of location * integer
    | String of location * string
    ...
```

Fig. 3.2. Définition du type sexp

- Var : '`vref`' (la référence à la variable dans le langage source).

Les autres constructeurs de l'arbre syntaxique abstrait, tels que '`Imm`', '`Builtin`', '`Proj`', '`Let`', '`Arrow`', '`Inductive`', '`Cons`', '`Case`', etc, n'ont pas été inclus dans cette version simplifiée parce qu'ils ne sont pas directement pertinents pour l'explication en cours. Cependant, une version plus détaillée de l'arbre syntaxique abstrait sera présentée dans la section 4.3.

Bien que l'élaboration soit la dernière phase du front-end de Typer qui peut signaler une erreur liée au code fourni, certaines erreurs peuvent également être détectées lors de la phase de vérification des types qui la suit.

3.1.1. Analyse syntaxique

Typer tire son inspiration syntaxique de langages tels que ML et Lisp, bien que sa syntaxe ressemble davantage à celle d'autres langages comme Haskell ou Ocaml. Lorsqu'un programme est soumis à l'analyse syntaxique de Typer, la forme résultante est une liste d'expressions symboliques, également appelée S-expression ou '`sexp`'. Dans la figure 3.2, nous avons la définition du type sexp:

Il est important de noter qu'à ce niveau, il n'y a pas de notion de sémantique. La phase de génération de l'arbre de symboles ne permet pas au code résultant d'indiquer ce qu'est une fonction, un appel de fonction, une macro ou une définition de type. Tout cela sera déterminé ultérieurement dans le processus d'élaboration du code Typer.

L'analyse syntaxique de Typer est également assez simpliste et se base sur l'utilisation de la Grammaire de Précédence d'Opérateurs (OPG, Operator Precedence Grammar), une classe de grammaire relativement restrictive. Cette approche suffit toutefois à introduire des opérateurs infixes et mixfixes.
Reprenons l'exemple du type '`List`' défini dans le première chapitre.

```
type List (a : Type)
  | nil
  | cons (hd : a)(tl : List a);
```

Dans cet extrait de code, '`type`' est utilisé comme mot-clé préfixe, tandis que '`|`' et '`:`' sont déclarés comme mots-clés infixes. Le résultat de l'élaboration de ce type donne le même résultat que si nous avions écrit :

```
type_ (_|_ (List (_:_ a Type))
  nil
  (cons (_:_ hd a) (_:_ tl (List a)));
```

À l'opposé de Lisp, les parenthèses ne sont pas obligatoires dans Typer, leur utilité est donc de regrouper les différents éléments.

3.1.2. Macro-expansions

L'expansion de macro est l'étape par laquelle les appels de macro sont remplacés par le résultat de leur évaluation. Elle doit se produire avant la vérification des types par le compilateur, afin que celui-ci dispose de toutes les informations nécessaires pour déduire le type des sous-expressions et expressions. Dans Typer, les macros sont des fonctions qui prennent en argument une liste de '`sexp`' et retourne une '`sexp`'. Toutefois, lors d'un appel de type *(f args)*, nous nous basons sur le type de f pour déterminer s'il s'agit d'un appel de fonction ou d'un appel de macro. Pour un appel de macro, l'expression f aura le type `Macro` défini ci-dessous :

```
type Macro
   | macro (List Sexp -> IO Sexp);
```

Comme expliqué ci-dessus, bien que le début du nœud soit de type `Macro`, cela ne garantit pas la validité de l'appel de macro. Ainsi, pour vérifier le code source, nous pouvons utiliser le format suivant :

```
(if x then mymacro else yourmacro) 42
```

Ici, '`(if x then mymacro else yourmacro)`' a le type '`Macro`', ce qui en fait un appel de macro valide. Cependant, si la valeur de '`x`' n'est pas connue lors de l'expansion des macros, l'appel est invalide, car on ne sait pas quelle macro appeler. Lorsque '`x`' ne peut pas être évalué à une valeur pendant l'expansion des macros, il est difficile de déterminer quelle macro doit être utilisée, ce qui peut entraîner des erreurs lors de la compilation. Il est donc important de s'assurer que toutes les valeurs nécessaires à l'expansion des macros sont connues au moment de la compilation pour garantir des appels de macro valides.

L'inconvénient de cette approche est que nous devons connaître le type d'en-tête du nœud afin de reconnaître l'appel de macro, ce qui revient à développer toutes les macros dans une seule phase afin de deviner le type. De plus, l'expansion de macro et l'inférence de type doivent être imbriqués car le type sera déduit après expansion de la macro. Cela peut être un inconvénient sérieux, mais l'avantage d'effectuer une expansion de macro dans la phase d'inférence de type est que la macro a accès au contexte de type complet.

3.1.3. Inférence de type

Certains types omis par le programmeur sont connus par le compilateur pendant l'élaboration. Cette étape est réalisée par deux fonctions de récursivité mutuelles, ce qui permet une approche bidirectionnelle de vérification de type :

- Infer : prend en argument une '`sexp`' et son environnement de typage, puis retourne deux '`lexps`'. La première '`lexp`' représente la forme élaborée de la '`sexp`', tandis que la deuxième '`lexp`' représente son type.

- Check : prend en argument une '`sexp`', une '`lexp`' représentant le type attendu, ainsi que son contexte d'élaboration. Elle renvoie une '`lexp`' représentant la '`sexp`' après élaboration. '`check`' effectue une vérification pour s'assurer que le type de la '`sexp`' élaborée correspond au type attendu, en utilisant la fonction de validation appropriée en fonction du type d'expression.

3.2. Algorithme génétique

Inspirés de l'évolution naturelle des espèces, ces algorithmes ont suscité l'intérêt de plusieurs chercheurs, notamment Holland [**17**] qui a développé les principes fondamentaux; suivi de Goldberg en 1989 [**15**] qui développe une application informatique de recherche stochastique pour les problèmes d'optimisation, ainsi que dans la théorie des jeux répétés par Axelord [**13**] et dynamiques par Ozyildirim et Alemdar [**39**].

3.2.1. Principe de base

Les algorithmes génétiques utilisent des approches d'optimisation inspirées de la génétique et de l'évolution de la nature. Ils peuvent être appliqués à différents problèmes, mais reposent sur des éléments communs, notamment :

(1) Choisir le codage des éléments : une fonction qui permet de modéliser les données du problème réel en données pouvant être utilisées par les algorithmes génétiques.

(2) Génération de la population initiale : la génération de la population initiale est importante car elle constitue le point de départ de l'algorithme et son choix conditionne la rapidité et l'optimalité de la solution finale.

(3) Fonction de performance : elle permet d'évaluer chaque solution de la population et de les comparer entre elles.

(4) Opérations génétiques : elles permettent de générer de nouvelles solutions dans l'espoir d'améliorer la fonction objectif (fonction de performance).

- Opérateur de croisement : il commence par choisir des parents dans la population pour construire un nouvel individu. Ensuite, il reconstruit les gènes de cet individu en manipulant les structures des chromosomes des parents.
- Opérateur de mutation : il a pour but d'explorer l'espace des solutions afin d'éviter d'avoir des populations uniformes.

(5) Remplacement de la population : après l'application des opérations génétiques, une nouvelle génération de solutions est formée.

(6) Paramètres de dimensionnement : ils comprennent la taille de la population, le nombre total de générations à obtenir avant d'arrêter, et les probabilités d'application de chaque opération génétique.

3.2.2. Fonctionnement

Le comportement de l'algorithme génétique commence par une sélection généralement aléatoire d'une population de premières solutions potentielles. La performance relative de chaque individu (solution) est évaluée par une fonction de performance, souvent appelée fonction de *fitness*, qui permet de déterminer sa qualité. La génération de nouvelles populations, c'est-à-dire de nouvelles solutions potentielles, est basée sur des opérateurs évolutifs tels que la sélection, le croisement et la mutation. Ce cycle d'évolution se répète depuis le début jusqu'à ce que la condition d'arrêt soit atteinte, qui peut être définie par le nombre de générations ou par la propriété de la solution obtenue.

La figure 3.3 illustre le principe de fonctionnement des algorithmes génétiques, ce qui facilite la compréhension du processus.

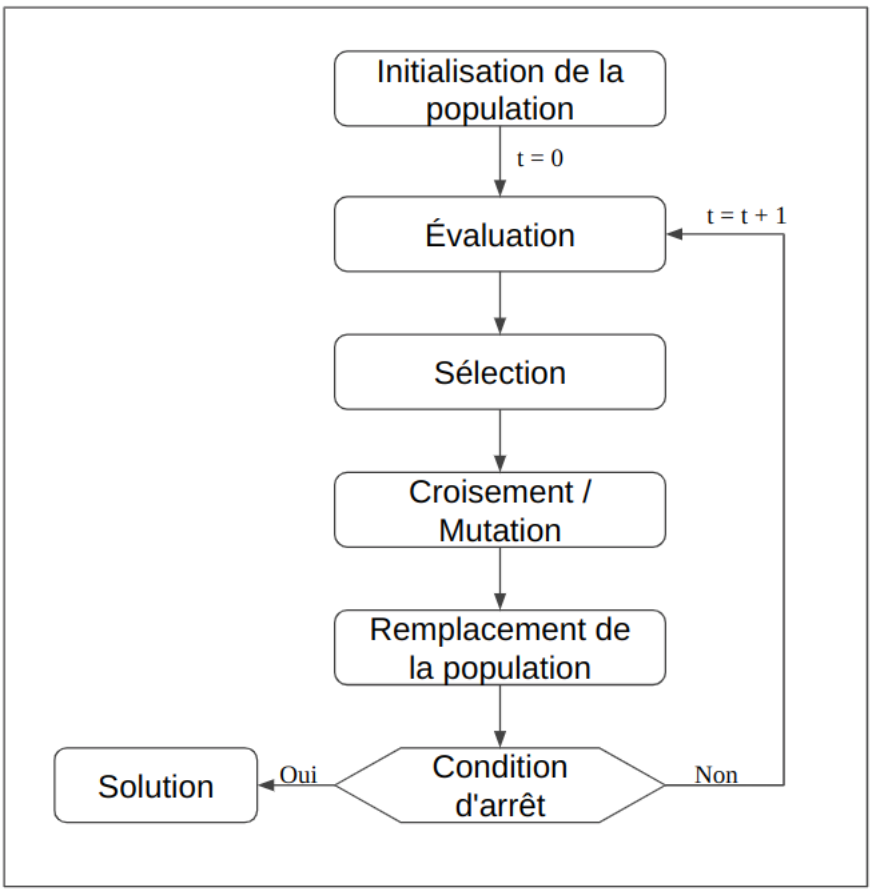

Fig. 3.3. Principe de fonctionnement d'un algorithme génétique

Avant de parler des caractéristiques d'un algorithme génétique, faisons une présentation de sa forme classique :

(1) Initialisation de la population P

(2) Évaluer P

(3) Tant que (Non Convergence) faire :

- P_1 = Sélectionner les meilleurs éléments de P
- P_2 = Appliquer Opérateur de Croisement sur P_1
- P_3 = Appliquer Opérateur de Mutation sur P_2
- P = Remplacer l'ancien population par la nouvelle
- Évaluer à nouveau P

Fin Tant que

3.2.3. Caractéristiques

Les algorithmes génétiques fournissent des approximations dans un délai raisonnable à des problèmes pour lesquels il n'existe pas de solution calculable analytiquement ou algorithmiquement. Le mécanisme d'évolution et de sélection reste le même, seules trois fonctionnalités changent dépendemment du problème à résoudre : le codage des données du problème à résoudre; l'espace de recherche et la fonction de performance qui permet dévaluer l'adéquation d'une solution à un problème et de déterminer sa pertinence.

3.2.3.1. Codage. Le choix du codage des données dépend de la spécificité du problème à résoudre et il influe fortement sur l'efficacité de l'algorithme génétique. Habituellement, sans les algorithmes génétiques, le codage est présumé être fait avec une séquence sur un alphabet donné. Cependant, dans notre cas, nous utilisons un codage sous programmation génétique d'arbre pour représenter les chromosomes, ce qui rend la notion de croisement plus complexe que dans le cas d'un codage sur une séquence avec un alphabet.

3.2.3.2. Fonction de performance. Également appelée fonction d'adaptation, d'objectif ou encore fitness, la fonction de performance offre la possibilité de comparer les individus entre eux en attribuant une valeur de performance à chaque individu. Cela permet à l'algorithme de déterminer quels individus doivent être remplacés ou sélectionnés pour la reproduction. Étant unique à chaque problème traité, il est crucial de bien choisir la fonction de performance. En effet, elle est à la base des mécanismes impliqués dans la conservation des individus adaptés et dans l'élimination progressive des individus peu performants. Fréquemment utilisée, il est souhaitable d'implémenter la fonction de manière rapide et efficace en termes de calcul.

3.2.4. Opérateurs génétiques

La reproduction est le processus de construction d'une nouvelle population à partir d'une population existante à l'instant "t". Ce processus est réalisé à l'aide des opérations de sélection, de croisement et/ou de mutation, qui agissent conformément à des normes spécifiques.

3.2.4.1. Sélection. L'opération de sélection est le processus par lequel les individus les plus performants sont choisis en fonction de leur valeur de fitness, c'est-à-dire leur score selon la fonction de performance. Plus un individu a une valeur de fitness élevée, plus il a de chances d'être sélectionné. Il existe plusieurs principes de sélection dans la littérature, tels que :

- *Sélection par rang* : Chaque individu se voit attribuer un rang en fonction de sa fonction de performance, et le rang 1 est attribué à l'individu avec la valeur la plus faible si le classement est effectué en ordre croissant.

- *Sélection proportionnelle* : Étant l'une des méthodes les plus couramment utilisées dans la littérature [**25**], la sélection proportionnelle à la fitness attribue à chaque individu de la population une probabilité p_i proportionnelle à sa valeur de fitness f_i selon la fonction de performance.

- *Sélection aléatoire* : Elle se fait de manière aléatoire, uniforme et sans aucun prise en compte de la fonction de performance. Les individus de la population ont la même probabilité d'être sélectionnés, ce qui rend généralement lente la convergence de l'algorithme génétique.

- *Sélection par tournoi* : On divise souvent la population en plusieurs groupes et seules la meilleure dans chaque groupe est choisie.

3.2.4.2. Croisement. Le premier opérateur du module de reproduction est le croisement. Après la sélection, deux individus de la population (les parents) échangent des gènes entre eux pour donner naissance à deux descendants (les enfants) possédant des caractéristiques héritées des parents.
Le but principal du croisement est de combiner différents schémas génétiques pour former de meilleures populations capables de conduire à la convergence de l'algorithme génétique. Il existe différentes techniques de croisement applicables sur des représentations binaires ou réelles, parmi lesquelles nous pouvons citer:

- Croisement à un point : sélectionner au hasard, pour chaque couple, un point de croisement, puis combiner la partie de gauche du premier parent à la partie de droite du deuxième (vice-versa) pour constituer la nouvelle génération.

- Croisement à multiple points : un échange de gènes entre les parents est effectué sur plusieurs points de coupure. Du fait que ses résultats sont très prometteurs, elle est largement utilisée dans diverses applications [**38**].

- Croisement uniforme : dans ce cas, un masque binaire de la taille des chromosomes est généré aléatoirement pour chaque parent. Le gène du premier enfant est une copie du gène du parent 1 si la valeur du bit de masque est 1 et du parent 2 si la valeur du bit de masque est 0. L'opération inverse est effectuée sur l'enfant 2.

3.2.4.3. Mutation. Longtemps sous-estimées dans le passé, plusieurs études [**28**] ont été menées sur le rôle de l'opérateur de mutation dans les algorithmes génétiques. Elle sert d'opérateur de base et est complémentaire à l'opérateur de croisement, apportant une valeur ajoutée en termes de diversité lors de l'évolution de la population. L'opérateur de mutation consiste à changer ou permuter des valeurs d'un individu afin d'améliorer ses caractéristiques. Les propriétés de convergence de l'AG dépendent largement de cet opérateur génétique. En 1994, Cerf [**7**] a démontré théoriquement que l'algorithme génétique converge en probabilité grâce à l'utilisation de l'opérateur de mutation, même sans recourir à l'opérateur de croisement.

Chapitre 4

Implémentation de la solution

Au chapitre précèdent, nous avons dressé un panorama du langage Typer en nous concentrant particulièrement sur son compilateur. Nous avons également présenté les concepts fondamentaux des algorithmes génétiques.
À ce stade du mémoire, nous entamons la phase de mise en oeuvre de la théorie proposée dans la section 1.3. Notre objectif est d'améliorer la manière dont les types sont imprimés dans les messages d'erreurs du compilateur Typer.
Dans ce chapitre, nous présenterons les différentes approches que nous avons jugées utiles pour améliorer les erreurs de types dans les messages d'erreurs. Pour ce faire, nous commencerons par décrire l'idée générale de notre approche. Enfin, nous présenterons la mise en oeuvre réalisée pour atteindre notre objectif.

4.1. Idée de base

Comme mentionné dans la section 1.3, le problème spécifique auquel nous faisons face est la recherche de l'expression '`sexp`' à partir de l'expression typée '`lexp`' dans le contexte de '`lexp = elab (sexp)`', où la fonction '`elab`' peut avoir une large variété de comportements. Nous sommes d'avis qu'il est possible de concevoir un modèle reposant exclusivement sur un algorithme génétique pour retrouver cette '`sexp`'. Cependant, il convient de noter que cela peut avoir un impact sur le temps d'exécution du modèle en raison de la complexité algorithmique associée à la représentation choisie. De plus, en raison du manque de travail pour améliorer les messages d'erreur des compilateurs ML à l'aide d'algorithmes génétiques, il est difficile de garantir à l'avance le succès de leur utilisation. Par conséquent, nous pensons qu'il est judicieux de mener deux expériences rapides avant d'essayer un algorithme génétique.

Dans la section 3.1 du chapitre précèdent, nous avons vu qu'une '`lexp`' est généralement composée d'un champs '`location`'. Comme première expérience, nous envisageons de remplacer ce champ par un autre champ de type '`sexp`', qui correspondra au résultat de la représentation d'origine du code source de l'utilisateur, après analyse. En toute logique, nous voulons faire l'opération inverse du résultat de l'inférence de type pour trouver l'information d'origine du code source du programmeur. Pour ce faire, nous aurons besoin de modifier le compilateur pour préserver la '`sexp`' d'origine obtenue à partir du code source. Dans le cas où cette '`sexp`' ne correspond pas à la '`sexp`' idéale, nous allons utiliser une table de hachage globale pour trouver cette '`sexp`'. Sur la base de notre intuition, nous nous attendons à ce que ces deux méthodes puissent donner des résultats concluants pour certains types.

L'exemple de '`List Int`' (section 1.2) n'est pas complexe en soi, mais il illustre un cas où la substitution du champs '`location`' par une '`sexp`' basée sur la représentation d'origine du code source ne sera pas suffisante pour trouver la solution, car l'information '`List Int`' n'est pas présente dans le code source de l'utilisateur. Cela montre que l'utilisation de la première approche ne sera pas une solution appropriée pour tous les cas. L'utilisation d'une table de hachage globale pour rechercher la '`sexp`' idéale peut être utile, mais cela ne garantit pas le succès dans tous les cas non plus. C'est pourquoi, nous envisageons également d'incorporer un algorithme génétique pour explorer différents solutions possibles et évoluer vers une solution valide, en manipulant génétiquement les structures de données de l'expression. Nous allons combiner ces différents approches pour améliorer nos chances de trouver une solution valide.

4.2. Modifications du compilateur

Dans la présentation du compilateur Typer, au chapitre 3, nous avons vu que l'élaboration d'une '`sexp`' en une '`lexp`' contient souvent comme information un champ '`location`'. L'idée derrière ce champ était d'indiquer plus ou moins la position relative de l'objet dans le code source. Cependant, cette information n'est pas d'une grande aide lors du débogage des messages d'erreur, car elle peut être obtenue à partir d'une '`sexp`' avec la fonction '`sexp_location`'. Ainsi, il est possible de remplacer la '`location`' par la '`sexp`' qui est à cette '`location`', sans perdre d'information.

Nous allons substituer l'information actuellement contenue dans le champs '`location`' par une nouvelle information appelée '`sinfo`', qui sera dérivée du type '`sexp`'. Cette approche nous permettra de propager l'information d'origine (c'est-à-dire le code source du programmeur) de manière différente, en utilisant le champ '`sinfo`' à la place de '`location`' dans l'arbre de syntaxe abstraite du compilateur Typer.

```
type ltype = lexp
    and lexp =
        | Imm of sexp
        | Builtin of symbol * ltype
        | Var of vref
        | Proj of sinfo * lexp * label
        | Let of sinfo * (vname * lexp * ltype) list * lexp
        | Arrow of sinfo * arg_kind * vname * ltype * ltype
        | Lambda of arg_kind * vname * ltype * lexp
        | Call of sinfo * lexp * (arg_kind * lexp) list
        | Inductive of sinfo
                    * (arg_kind * vname * ltype)
                    * ((arg_kind * vname * ltype) list) SMap.t
        | Cons of lexp * symbol
        | Case of sinfo * lexp
                * ltype
                * (sinfo * (arg_kind * vname) list * lexp) SMap.t
                * (vname * lexp) option
        ...
```

Fig. 4.1. Arbre syntaxique abstrait du compilateur Lexp mis à jour

```
    type sinfo = sexp
```

En effectuant ce changement, nous en avons profité pour apporter quelques ajustements nécessaires.

4.3. Vue 360 de l'arbre syntaxique abstrait du compilateur

En tenant compte des dernières modifications apportées, nous allons présenter à nouveau la structure de données pour l'arbre de syntaxe abstraite du compilateur Typer, qui est représentée dans la figure 4.1.

En détails, nous avons :

- `Imm` : correspond à une valeur présente littéralement dans le code source de Typer (sexp, entier ou chaîne de caractères).
- `Builtin` : représente les constantes (de types, valeurs ou fonctions) implémentées dans le code Ocaml.
- `Var` : correspond aux variables.

- `Proj` : est un cas particulier de '`Case`'. Elle permet un accès direct à un champ d'un tuple lorsque ce dernier ne possède qu'un seul constructeur.
- `Let(`*`SI, DECLS, BODY`*`)` : représente une expression qui introduit une liste de déclarations locales *DECLS*, qui peuvent ensuite être utilisées dans *BODY*. Les déclarations peuvent être mutuellement récursives, ce qui signifie qu'elles peuvent se référencer les unes les autres dans leur définition.
- `Arrow et Lambda` : représentent les fonctions et leur type de flèche, où chaque argument possède un `arg_kind` indiquant s'il s'agit d'un argument normal, implicite ou effacé, ainsi qu'un nom et un type.
- `Call(`*`SI, FUN, ARGS`*`)` : représente un appel à la fonction *FUN* avec arguments *ARGS*.
- `Inductive(`*`SI, CONSTRS`*`)` : représente un type algébrique, également connu sous le nom de "type inductif", composé des constructeurs *CONSTRS*. Il correspond au '`typecons`' montré dans l'introduction.
- `Cons` : les valeurs de constructeurs font référence à un type spécifique ainsi qu'au nom d'un de ses constructeurs.
- `Case` : les expressions de filtrage sont des outils en programmation qui permettent de destructurer des types inductifs. Elles sont constituées d'une expression à filtre, d'un type de retour, d'une table de branche et éventuellement d'une branche par défaut. La table de branche associe chaque constructeur d'un type inductif à une liste de variables à lier et un corps de branche qui spécifie l'action à effectuer sur ces variables.

4.4. Jeu de données

Dans la jargon des algorithmes génétiques, on parle de jeu de données (ou ensemble de données) en utilisant les termes population, individu, chromosome et gène. Donc une population est un groupe d'individus, chaque individu (solution) est représenté par un chromosome avec une séquence de gènes dont les valeurs sont choisies aléatoirement.

Pour nos tests, nous utilisons les types des éléments, fonctions et modules prédéfinis du langage comme jeu de données. Quelques exemples de notre jeu de données sont présentés dans la figure 4.2.

Notre jeu de données est constitué de '`lexps`', qui sont des expressions dans le langage Typer. Comme illustré dans la figure 4.2, ces '`lexps`' sont représentées en '`sexp`' afin d'être

```
(ℓ : TypeLevel) ≡> (a : (##Type_ ℓ)) -> (##Type_ ℓ)
Type ?ℓ -> Type ?ℓ


(ℓ : TypeLevel) ≡> (a : (##Type_ ℓ)) ≡> (a -> Option (_ := ℓ) a)
?a -> Option ?a


(ℓ : TypeLevel) ≡> (a : (##Type_ ℓ)) ≡> Option (_ := ℓ) a
Option ?a

(ℓ : TypeLevel) ≡> (a : (##Type_ ℓ)) ≡> (xs : (List (_ := ℓ) a))
                                              -> (Option (_ := ℓ) a)
( xs : (List ?a)) -> Option ?a
```

Fig. 4.2. Quelques éléments du jeu de données

utilisées comme données d'entrée pour notre algorithme génétique. Pour chaque exemple, nous avons la représentation de la '`sexp`' idéale qu'on aimerait obtenir. La figure 4.3 donne la représentation basique sous forme d'arbre du dernier exemple de la figure 4.2.

En analysant les éléments du jeu de données, nous distinguons deux types d'arguments : les arguments formels et les arguments actuels. En tant qu'arguments actuels, nous avons 'ℓ' qui suit '`Option`' dans la deuxième ligne, dont la présence n'est pas obligatoire. En ce qui concerne les arguments formels, nous pouvons citer le '`(xs : List (_ := ℓ) a)`' dans le dernier exemple de la figure 4.2 et les arguments implicites ('⇒' ou '≡>') qui sont déduits lors de la compilation, de sorte que les programmeurs peuvent les omettre sans obtenir d'erreur de compilation.

Le jeu de données que nous utilisons comprend 100 éléments. Le tableau 4.1 présente les caractéristiques des types les plus complexes de notre jeu de données, notamment leur nom, leur taille ainsi que les problèmes associés à chaque type. Parmi ces problèmes, nous identifions principalement :

- Problème 1 : mauvais placement de certaines flèches implicites.
  Prenons l'exemple du type de '`K`' :

  ```
  (a : Type) ≡> a -> (b : Type) ≡> b -> a
  ```

  Si nous représentons cette expression sans les flèches implicites, nous obtenons :

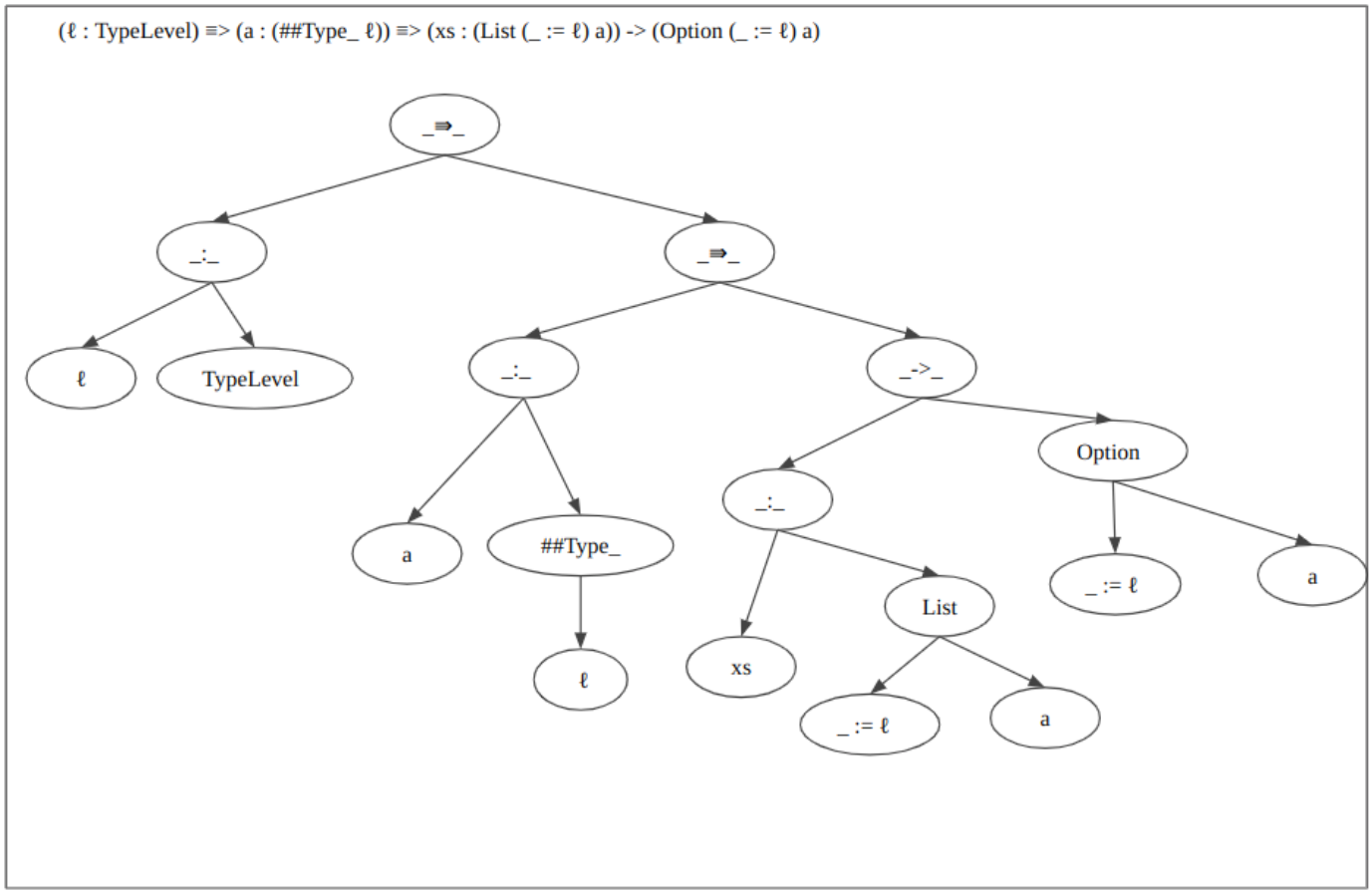

Fig. 4.3. Représentation sous forme d'arbre de la sexp de base de la fonction `List_head1`

```
?a -> ?b -> ?a
```

Cette représentation peut conduire à une interprétation incorrecte de la forme originale de l'expression. Ce que Typer va interpréter comme :

```
(a : Type) ⇒ (b : Type) ⇒ a -> b -> a
```

Cependant, cette interprétation est incorrecte. Lorsque nous utilisons cette représentation sans flèches implicites dans notre système de vérification de type, l'expression générée sera rejetée, car elle ne correspond pas à la structure valide de l'expression d'origine. Il est donc important de noter qu'il n'est pas possible d'éliminer toutes les flèches implicites tout en garantissant la validité de l'expression générée par rapport au vérificateur de type.

- Problème 2 : importation excessive de certains modules.
  Certaines représentations de types nécessitent l'utilisation d'éléments provenant

Tableau 4.1. Caractéristiques des types les plus complexes du jeu de données

Nom de fonction	Nombre de nœuds	Problèmes
List_mapi	25	Problème 1
K	18	Problème 1
Decidable	11	Problème 1
vnil	24	Problème 1
vcons	36	Problème 1
fold	12	Problème 1
poly-lits	49	Problème 1 & 2
fromInteger	12	Problème 1 & 2
monads	161	Problème 1 & 2
pure	19	Problème 1 & 2
IO_monad	15	Problème 1 & 2
Option_monad	17	Problème 1 & 2
list	650	Problème 1 & 2
do-lib	113	Problème 1 & 2
tuple-lib	102	Problème 1 & 2

d'autres modules de type tels que '`poly-lits`', '`fromInteger`', '`monads`', '`pure`', '`IO_monad`', '`Option_monad`', '`list`', '`do-lib`' et '`tuple-lib`'. Cependant, notre système actuel ne permet pas de récupérer ces éléments en utilisant la notation "Nom_du_module.Nom_element". Par conséquent, afin d'utiliser ces éléments, nous sommes contraints d'importer l'ensemble du module contenant l'élément dans la définition du type. Cela entraîne une augmentation significative de la taille de certains types complexes.

Par exemple, dans le cas du type de '`IO_monad`' qui utilise l'élément '`monad`' provenant du module '`Monad`', notre système actuel nécessite d'importer tout le module '`Monad`', même si nous n'avons besoin que de l'unique élément '`monad`'.

Une analyse plus détaillée de ces problèmes sera effectuée dans la section 5.1.2.2, où nous examinons les résultats de notre système heuristique.

4.5. Implémentation de la solution

Dans cette section, nous décrivons le schéma d'implémentation de notre système heuristique. Nous commençons par présenter notre première méthode dans la section suivante.

4.5.1. lexp_sinfo

Dans la section 4.4, nous avons pu observer une vue de 360 degrés de l'arbre de synthèse du compilateur Typer mise à jour. On remarque que les '`lexps`' élaborées contiennent une information appelée '`sinfo`'.

Lorsqu'on cherche la '`sexp`' idéale pour une '`lexp`' donnée, il est essentiel de prendre en compte la '`sinfo`' incluse dans cette dernière. Si la '`sinfo`' contenue dans la '`lexp`' est acceptée par notre système de vérification de type (section 1.3), alors elle est considérée comme une bonne candidate, et nous la retournons directement à l'utilisateur. Dans le cas contraire, nous tentons de trouver cette '`sexp`' idéale en utilisant une autre approche présentée dans la section suivante.

4.5.2. H_Table

L'idée de base est de maintenir une table qui agit comme un cache de résultats pour les '`lexps`' élaborées et leur '`sexps`' correspondantes. À chaque utilisation de la fonction '`elaborate`' pour élaborer une '`sexp`', le résultat obtenu, c'est-à-dire la '`lexp`' résultante, est enregistré dans cette table de hachage.

Il convient de souligner que différentes '`sexps`' peuvent renvoyer la même '`lexp`' élaborée. Ainsi, notre table de hachage garde, pour chaque '`lexp`' élaborée, l'ensemble des '`sexps`' qui lui correspondent. Le choix d'une '`sexp`' spécifique est effectué de manière aléatoire. Par conséquent, la relation entre les '`lexps`' élaborées et les '`sexps`' correspondantes est préservée dans les deux sens.

Rappelons également que les approches décrites précédemment peuvent ne pas fonctionner dans tous les cas. En effet, ces approches sont souvent efficaces pour les '`lexps`' qui ont été élaborées à partir des '`sexps`' explicitements définies dans le code source. Cependant, les '`lexps`' qui nous intéressent peuvent souvent être générées par l'inférence de types, sans être explicitement définies dans le code source. Cela peut rendre ces approches moins efficaces car les informations nécessaires pour résoudre les types peuvent ne pas être disponibles.

Si toutefois, après ces deux approches, nous ne parvenons toujours pas à obtenir une solution valide acceptée par notre système de vérification de types, nous utilisons comme dernière approche un algorithme génétique pour tenter de trouver une solution valide similaire ou

proche de l'idéal.

4.5.3. Algorithme Génétique

Dans la deuxième partie du chapitre précédent (3.2), nous avons présenté brièvement le fonctionnement des algorithmes génétiques. Maintenant, nous allons appliquer ces règles pour créer notre algorithme génétique. Nous commençons naturellement par la première phase.

(1) *Initialisation de la population*

Utilisée une seule fois en démarrant le processus de l'AG, l'initialisation de la population en est du moins la phase la plus importante, car elle influence non seulement la qualité de la solution finale, mais aussi le nombre d'itérations nécessaires pour obtenir une solution satisfaisante [**34**]. Cette phase doit se faire de manière aléatoire avec une taille suffisamment grande pour explorer l'espace de recherche dans un temps raisonnable.

En ce qui concerne la taille de la population, si elle est trop petite, l'espace d'état peut ne pas être exploré de manière adéquate, ce qui peut entraîner une convergence prématurée de la population vers un minimum local. D'autre part, une taille trop grande peut entraîner une convergence lente de l'algorithme et affecter ses performances. Dans cette optique, Alander [**3**] suggère une taille de population comprise entre ℓ et 2ℓ avec ℓ la taille du chromosome.

La figure 4.4 illustre une représentation sous forme d'arbre de deux individus générée à partir de la '`sexp`' de la figure 4.3 :

Une fois la population initiale créée, nous pouvons passer à l'étape suivante, la sélection des individus.

Comme ce projet est exploratoire, nous avons choisi d'utiliser une taille de population fixe dans notre approche.

(2) *Sélection*

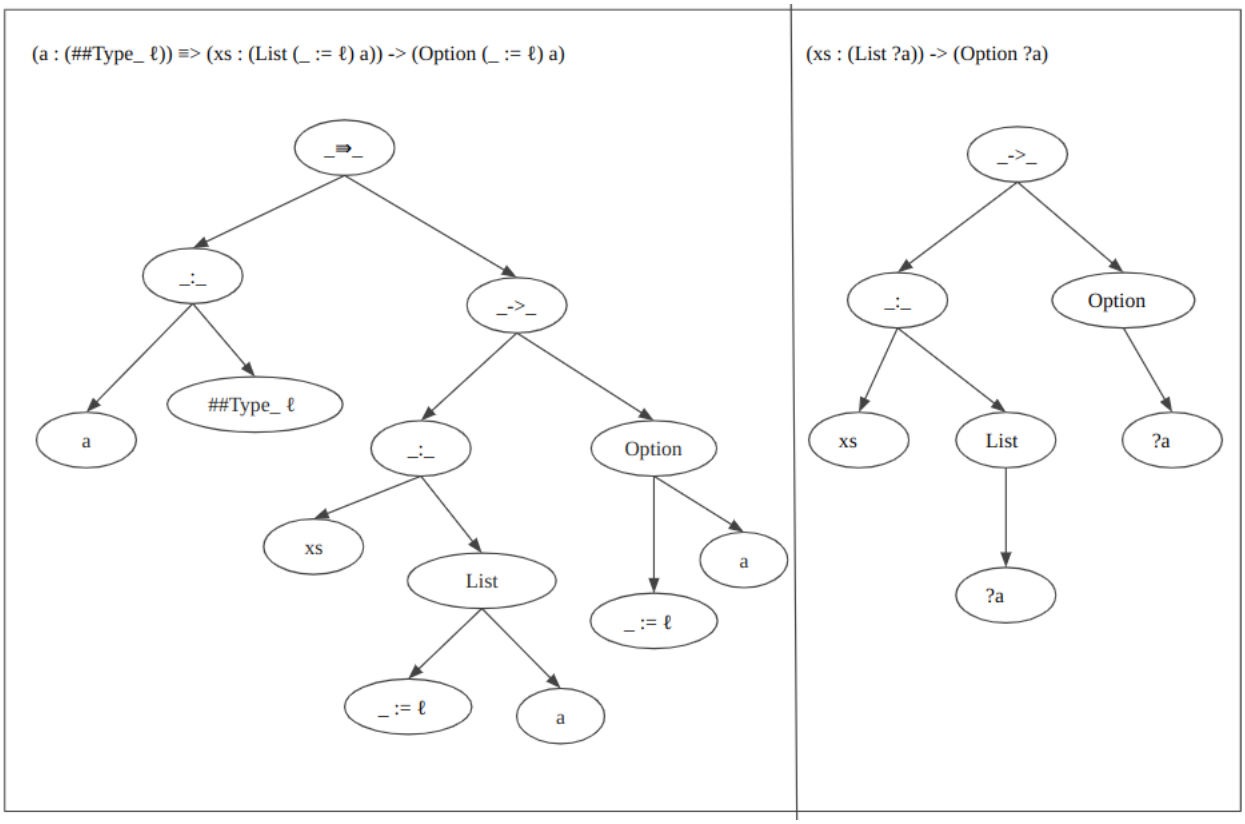

Fig. 4.4. Quelques éléments de la population représentés sous forme d'arbre

Étant donné que l'initialisation de la population est effectuée de manière aléatoire, la sélection a pour objectif d'évaluer et de choisir les individus les plus performants dans la population.

Il existe plusieurs méthodes de sélection (voir section 3.2.4.1), mais pour ce projet, nous avons opté en premier lieu pour une méthode de sélection proche de celle utilisée dans la sélection proportionnelle. Dans cette méthode, nous évaluons la performance de chaque individu à l'aide de valeurs numériques entières plutôt que des probabilités, contrairement à la méthode de sélection proportionnelle. Pour ce faire, notre fonction de performance combine deux systèmes pour attribuer une valeur à chaque individu:

(a) Un système de vérification de type qui consiste à vérifier que le type de l'expression générée (représentée sous forme de '`sexp`') est cohérent avec le type attendu.

(b) Une métrique de la taille qui mesure la taille de la '`sexp`' en comptant le nombre de nœuds dans l'arbre syntaxique de la '`sexp`'.

En combinant ces deux éléments, chaque individu de la population se voit attribuer une valeur comme score de *fitness*. Si un individu est valide selon notre système de

vérification de type, sa taille est utilisée comme score de *fitness*, mais contrairement à la logique habituelle, plus la taille est grande, plus la *fitness* est petite. En revanche, si l'individu est rejeté, nous lui attribuons un score de *fitness* arbitraire. Après la phase de sélection, si certains individus sont valides selon le système de vérification de type, nous allons les croiser entre eux et avec d'autres individus rejetés par le système de vérification de type pour essayer d'obtenir des individus adaptés.

Si le système de vérification de type rejette tous les individus de la population, ils auront tous la même valeur de performance. Dans ce cas, nous allons choisir la méthode de sélection aléatoire.

Une fois que l'étape de sélection est terminée, nous passons à la prochaine étape, le croisement des individus sélectionnés.

(3) *Croisement*

Un des principaux opérateurs utilisés dans les algorithmes génétiques, est une synthèse des candidats parents ayant survécu à l'étape de sélection, en vue de créer de nouveaux descendants qui constitueront la population de la prochaine génération. Lors du croisement les parents sont pris par paires et leurs formes physiques sont échangées dans un certain ordre afin d'obtenir la solution souhaitée.

Contrairement aux méthodes de croisements classiques qui renvoient généralement deux enfants, notre méthode ne génère qu'un seul enfant, mais elle est similaire au croisement de type uniforme en raison de la nature de nos gènes. En effet, nos gènes sont constitués d'un ou de plusieurs '`sexps`', chaque '`sexp`' est un nœud composé d'un symbole tel qu'une flèche normale ($\rightarrow$) ou implicites ($\Rightarrow$ et $\equiv>$). Dans la section 4.4, nous avons constaté que la taille de nos gènes varie, ce qui rend impossible la génération d'un masque binaire pour le croisement. À chaque niveau de l'arbre, nous effectuons plutôt un choix aléatoire entre le nœud du premier et le nœud du deuxième parent. Cependant, cette approche présente l'inconvénient majeur de limiter le croisement aux nœuds du même niveau. De plus, elle ne garantit pas nécessairement la production d'expressions valides.

Pour illustrer ce processus, prenons deux parents comme indiqué dans la figure 4.5. Chaque parent est représenté par une '`sexp`', qui est un arbre syntaxique composé de nœuds symboliques représentant différents éléments du langage. Nous utilisons

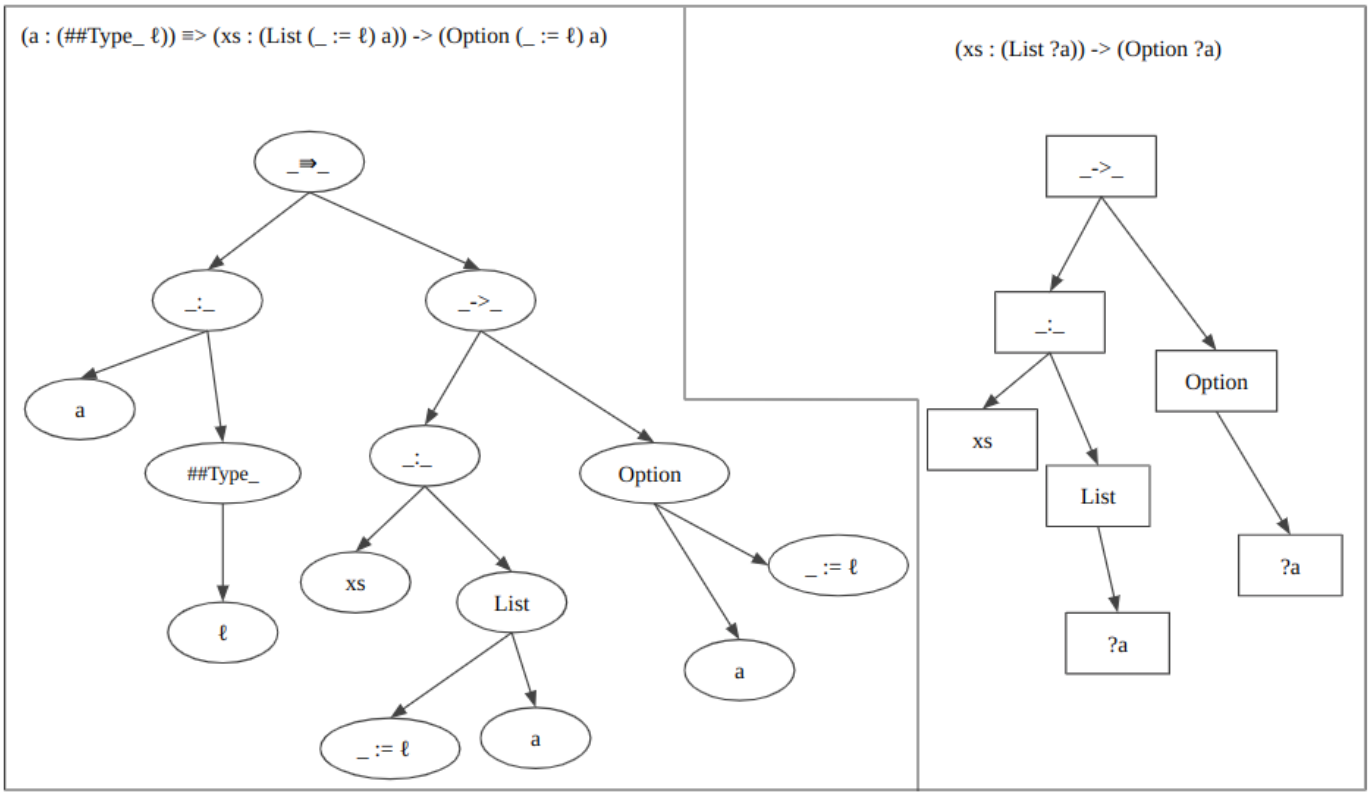

Fig. 4.5. Les parents du croisement

des ovales pour représenter l'un des parents et les rectangles pour représenter l'autre parent. Cette convention visuelle nous permet de différencier visuellement les deux parents pendant le processus de croisement.

La figure 4.6 présente deux exemples d'enfants possibles résultant du croisement des parents de la figure 4.5.

Chaque enfant de la figure 4.6 ne montre qu'un seul changement afin d'illustrer une partie spécifique du processus de croisement. Toutefois, il est important de souligner que cela ne signifie pas que les croisements se limitent à un seul changement. Dans notre approche, les croisements peuvent potentiellement impliquer plusieurs changements, mais nous avons choisi de présenter des exemples simples pour des raisons de clarté et de concision. Cette limitation de l'exemple ne doit pas être interprétée comme une restriction générale sur le nombre de changements pouvant être effectués lors des croisements.

Après la phase de croisement, il y a généralement une phase de mutation dans les algorithmes génétiques. Toutefois, dans le cadre de ce projet exploratoire, nous ne

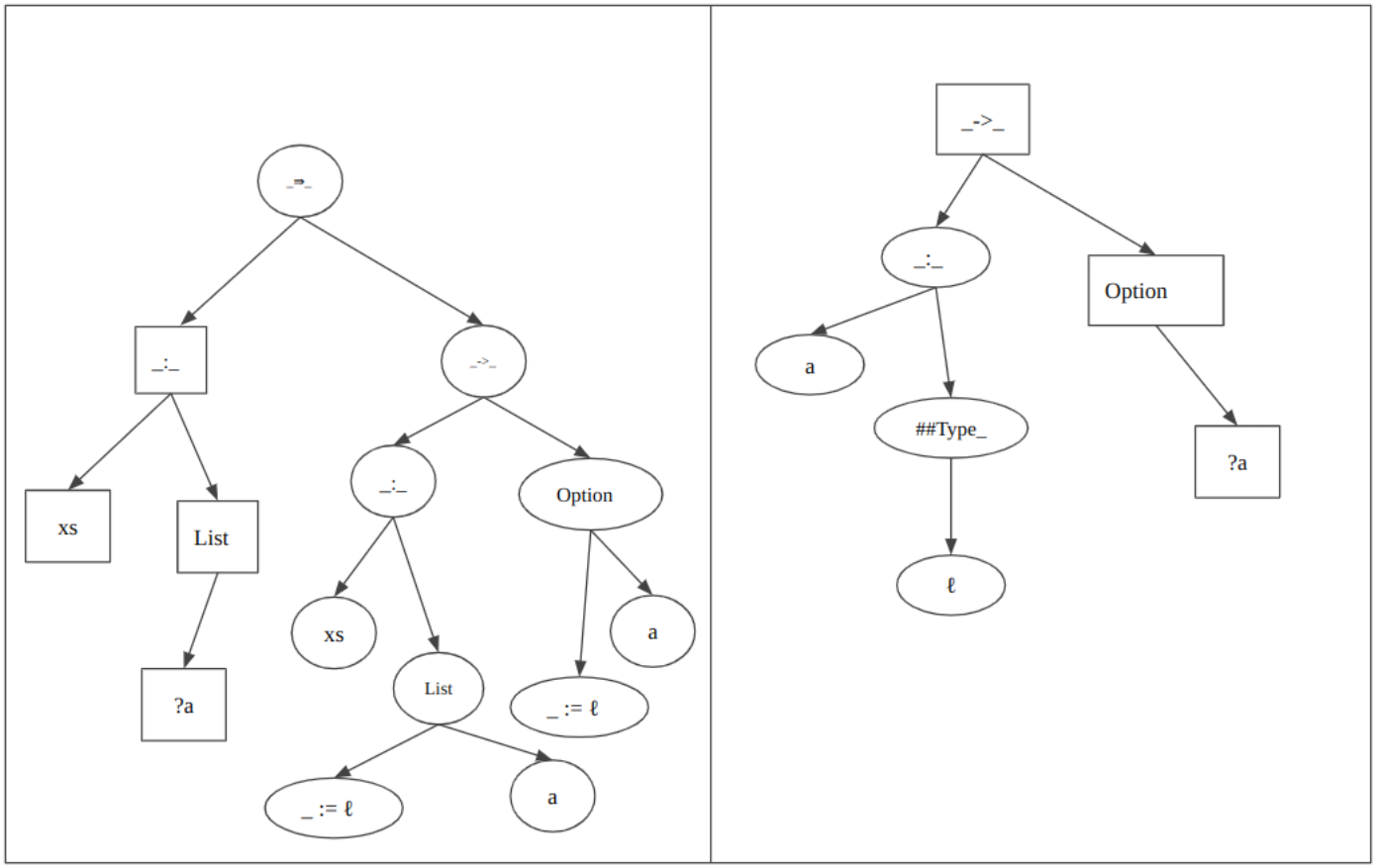

Fig. 4.6. Les individus issus du croisement

jugeons pas nécessaire d'implémenter la phase de mutation. Nous nous concentrons donc uniquement sur la phase de croisement.

4.5.4. Modèle fiable

Afin de garantir au moins la génération d'une expression valide, indépendamment de la complexité du type donné, nous avons mis en place un modèle fiable. Ce modèle utilise la même implémentation que celle employée pour générer des solutions aléatoires dans notre algorithme génétique, sauf qu'il est configuré de manière à faire le choix le plus fiable à chaque étape. Ainsi, à chaque nœud de la solution, des choix pertinents sont effectués pour assurer la cohérence et la validité de l'expression générée. Grâce à cette approche, nous sommes en mesure de garantir qu'au moins une solution valide sera obtenue par notre système heuristique.

4.5.5. Modèle optimiste

En parallèle du modèle fiable, nous avons également mis en place un modèle optimiste qui utilise la même implémentation que celle employée pour générer des solutions aléatoires dans notre algorithme génétique, mais avec les choix les plus optimistes. Ce modèle vise à maximiser les performances en recherchant des expressions plus lisibles, tout en étant plus permissif quant à leur validité. Contrairement au modèle fiable qui se concentre strictement sur la validité des expressions, le modèle optimiste propose certaines expressions qui pourraient ne pas être valides, mais qui sont néanmoins plus compréhensibles pour les utilisateurs.

Nous avons conçu notre système heuristique de manière à pouvoir le paramétrer selon les besoins. Ainsi, nous pouvons le forcer à renvoyer une solution qui fait toujours les choix les plus optimistes, privilégiant la lisibilité des expressions, ou à toujours faire les choix les plus fiables, garantissant la validité des expressions. En utilisant cette flexibilité, nous extrayons deux candidats spéciaux : le candidat optimiste et le candidat fiable. Le candidat optimiste offre des expressions plus compréhensibles pour les utilisateurs, tandis que le candidat fiable garantit la validité des expressions générées.

Chapitre 5

Évaluation

Le chapitre précèdent, consacré à la mise en place de la solution, nous a permis de présenter l'idée de base de notre solution. Nous avons ensuite détaillé l'implémentation de la solution, en commençant par les modifications apportées au compilateur; suivi de la présentation de nos trois méthodes ainsi que leurs interactions. Nous avons également passé en revue la stratégie de collecte de données.
Ce chapitre se concentre sur l'évaluation de notre modèle et est divisé en deux sections. Dans un premier temps nous allons évaluer l'aspect général de notre travail. Et enfin, nous présenterons quelques repères mettant en avant les performances du système heuristique.

5.1. Évaluation des Résultats

Dans cette section, il sera question d'identifier les forces et les faiblesses de notre solution, mais aussi de trouver la configuration optimale des différents paramètres du système heuristique, avant de finir avec l'analyse des résultats.

5.1.1. Tests des modèles

Avant de passer en phase de test proprement dire, nous devons d'abord choisir les critères d'évaluation de nos modèles. Pour ce faire, nous avons sélectionné deux métriques d'évaluation couramment utilisées, à savoir l'*accuracy* et la *precision*. L'*accuracy* mesure le pourcentage de résultats valides et la *precision* mesure le pourcentage de résultats valides qui sont réellement corrects.

Bien que ces métriques soient généralement utilisées pour évaluer la performance des modèles de classification en comparant leurs prédictions avec les valeurs réelles de l'ensemble de test, leur applicabilité dans le contexte de typage d'expression peut ne pas être évidente.

Toutefois, pour un type (‘`lexp`’) donné, notre système peut suggérer plusieurs solutions (‘`sexps`’) valides. Il est donc crucial de distinguer ces solutions afin de retourner la représentation la plus lisible au programmeur. Malheureusement, dans le contexte de ce projet, nous ne disposons pas d'un jeu de données de taille conséquente (voir section 4.4).

Cependant, les concepts sous-jacents à ces métriques restent pertinents et seront pris en compte pour évaluer la performance de notre système heuristique.

Nous débutons nos tests en utilisant la métrique *accuracy*. Elle mesure uniquement la validité, c'est-à-dire la proportion d'observations qui sont correctement prédites comme ayant un type valide. En effet, nous sommes confrontés à deux problèmes dans le contexte du typage des expressions. D'une part, les ‘`sexps`’ naïves générées par les modèles peuvent souvent être incompréhensibles pour l'utilisateur. D'autre part, ces ‘`sexps`’ peuvent ne pas toujours être valides. Par conséquent, l'*accuracy* ne mesure pas la lisibilité ou la qualité des types prédits, mais seulement leur validité.

L'*accuracy* est définie comme suit :

$$Accuracy = \frac{Nombre\ d'observations\ correctement\ prédites\ ayant\ un\ type\ valide}{Nombre\ total\ d'observations} \quad (5.1.1)$$

Nous allons évaluer la performance de chaque modèle en utilisant ses propres solutions, à l'exception du modèle AG pour lequel nous tiendrons compte des solutions prédites par les autres modèles pour générer la population initiale.

5.1.1.1. Modèle lexp_sinfo. Pour rappel, dans ce modèle, nous tentons de récupérer la ‘`sinfo`’ incorporée directement dans la ‘`lexp`’ élaborée.

L'*accuracy* score pour ce modèle est de zéro (‘`0 %`’). Cela peut s'expliquer par le fait que la ‘`sinfo`’ de la fonction `mapcar` (plus connue sous le nom de `map` dans de nombreux langages de programmation) renvoie souvent le code source du `mapcar` lui-même plutôt que le code source de son type (qui a été inféré et donc n'existe plus). La raison pour cela est que le code préserve les ‘`sinfos`’ pour chaque ‘`lexp`’, où la ‘`sinfo`’ d'une ‘`lexp`’ correspond à la ‘`sexp`’ qui en est l'origine ou la cause. Ainsi, pour le type inféré d'une fonction, c'est en effet le code de la fonction qui est utilisé comme indication.

Cependant, il convient de préciser que d'un côté, nous avons la ‘`sinfo`’ qui se trouve à la racine de la ‘`lexp`’, qui correspond à la “cause” de cette expression. De l'autre, nous avons

les '`sinfos`' qui se trouvent dans les sous-nœuds internes de la '`lexp`'. Ainsi, il est possible que la '`sinfo`' renvoyée par une '`lexp`' donnée soit incorrecte, mais cela ne signifie pas nécessairement que toutes les '`sinfos`' trouvées dans les sous-nœuds internes sont également incorrects. En conséquence, il est toujours utile l'inclure ces '`sinfos`' dans le modèle final du système heuristique.

5.1.1.2. Modèle H_Table. Ce modèle stocke les résultats de toutes les élaborations effectuées par le compilateur. Cela nous permet de garder en mémoire toutes les '`sexps`' qui peuvent conduire à une ou plusieurs '`lexps`'. Pour une '`lexp`' donnée qui existe dans la table de hachage, à chaque pas de nœud nous retournons aléatoirement une '`sexp`' de notre table de hachage. Ce modèle, nous donne un *accuracy* score de '`92 %`'.

Nous avons constaté que les '`sexps`' générées par le modèle `H_Table` présentent des similitudes en termes de représentation avec les '`sexps`' du modèle fiable.

Dans la section suivante, nous aborderons plus en détail le modèle fiable.

5.1.1.3. Modèle fiable. Pour une lexp donnée, ce modèle est censé renvoyer une solution valide quelle que soit la complexité du type. Les solutions de ce modèle préservent les arguments implicites dans les nœuds. De plus, les flèches (normales, implicites ou effaçables) et les types prédéfinis comme ('`##Typer_`', '`##TypeLevel_z`') sont affichés dans un format spécifique; toujours préfixés des caractères '`##`'.

L'utilisation du préfixe '`##`' est une convention spécifique à Typer pour distinguer les identifiants prédéfinis des identifiants définis par l'utilisateur. Par exemple, considérons le code suivant :

```
Type-entier = Int;
Int = "I never talk";
String.length (Int)
```

La réponse renvoyée par l'évaluation de la dernière ligne pourrait être :

```
12 : ##Int
```

Dans cette réponse,'`##`' est utilisé pour représenter le type défini '`Int`'. En effet, à cet endroit, '`Int`' n'est plus le nom du type des nombres entiers, donc '`12 : Int`' serait incorrect. Ce serait correct par contre de dire '`12 : Type-entier`' vu que '`Type-entier`' à cet endroit est "un" nom valide (pas le seul) pour désigner le type des nombres entiers.

Ainsi, le résultat de l'expression '`String.length (Int)`' serait de longueur 12 et de type '`##Int`'.
Il est imprtant de noter que '`##Int`' est un nom spécial qui ne peut pas être redéfini. Par conséquent, '`12 : ##Int`' est toujours valide.

Avec ce modèle, nous obtenons un *accuracy* score de '`100 %`', ce qui indique que toutes les solutions proposées par ce modèle sont acceptées par notre système de vérification de type.

5.1.1.4. Modèle optimiste. Avec cette solution, nous donnons à l'utilisateur une représentation simplifiée du type des expressions. En effet, tous les arguments implicites sont enlevés, quel que soit le nœud d'appartenance. Contrairement à la solution fiable qui utilise le préfixe '`##`' pour représenter les identifiants prédéfinis, dans cette solution, les variables prédéfinies et les flèches normales sont sous leur forme simple, sans le préfixe '`##`'. Cette approche permet de raccourcir les expressions dans le code source, en utilisant des méta-variables. Par exemple, l'expression '$(a : Type) \Rightarrow a \rightarrow List\ a$' peut être raccourcie en écrivant '$?a \rightarrow List\ ?a$'. Les méta-variables sont des noms temporaires qui sont remplacés par leur forme plus explicite pendant l'élaboration du code, ce qui facilite la lisibilité du code tout en conservant la validité des expressions générées.

Cependant, il est important de noter que cette solution peut être moins fiable en termes de validité des types, car elle simplifie les types en enlevant les arguments implicites, ce qui peut entraîner des résultats incorrects dans certains cas. Cette problématique correspond au "problème 1" que nous avons identifié dans la section 4.4 du chapitre précèdent.

L'*accuracy* score pour ce modèle est égal à '`84 %`'.

Pour mieux illustrer la différence entre les solutions proposées par le modèle fiable et celles proposées par le modèle optimiste, nous présentons dans la figure 5.1 des exemples de types avec leur représentation en utilisant les deux modèles.
Sans entrer dans les détails, ces exemples mettent en évidence la différence de lisibilité entre les deux approches. Chaque exemple est composé de deux expressions : la première expression représente la solution obtenue avec le modèle fiable, tandis que la seconde expression représente la solution obtenue avec le modèle optimiste.

```
(##_≡>_ (_:_ ℓ ##TypeLevel)(##_≡>_ (_:_ a (##Type_ ℓ))
                                   (##_->_ (List (_:=_ _ ℓ) a) Bool)))
(_->_ (List ?a) Bool)


(##_≡>_ (_:_ ℓ ##TypeLevel)(##_≡>_ (_:_ τ (##Type_ ℓ))
                                          (##_->_ (_:_ x τ) τ)))
(_->_ (x : ?τ) ?τ)


(##_->_ Sexp (##_->_ Sexp (##_->_ Sexp Sexp)))
(_->_ Sexp (_->_ Sexp Sexp))


(##_≡>_ (_:_ ℓ ##TypeLevel)(##_≡>_ (_:_ a (##Type_ ℓ))
                         (##_->_ a (##_≡>_ (_:_ b Type) (##_->_ b a)))))
(_->_ ?a (_->_ ?b ?a))
```

Fig. 5.1. Différences entre les solutions fiables et optimistes

Notez que la compréhension détaillée des exemples n'est pas nécessaire, ils sont utilisés uniquement à des fins de comparaison de lisibilité entre les deux modèles.

5.1.1.5. Modèle algorithme génétique. Il s'agit d'un algorithme génétique qui vise à suggérer une '`sexp`' correcte étant donné une '`lexp`'. L'algorithme génétique génère des solutions aléatoires, où les dés sont jetés à chaque sous-nœud interne de l'arbre de la '`lexp`' pour sélectionner parmi les différentes options disponibles.
Les solutions générées par le modèle AG sont souvent une combinaison des modèles mentionnés précédemment.

Pendant la phase d'initialisation de la population, les solutions du modèle optimiste ont été ajoutées avant l'application des opérations génétiques pour proposer une solution finale, ce qui nous permet d'obtenir un '`accuracy`' score de '`98 %`'. Nous avons ensuite inclus les solutions du modèle fiable pour augmenter ce score à '`100 %`'. Ainsi, l'*accuracy* score obtenu à partir de ce modèle est '`100 %`'.

Avec ce modèle, notre objectif principal n'est plus seulement d'obtenir une solution valide, mais aussi de produire une solution lisible pour l'utilisateur. C'est pourquoi à chaque initialisation de la population, nous nous assurons de toujours inclure les solutions des autres modèles : la solution obtenue avec le modèle `lexp_sinfo` , la solution retournée par

le modèle `H_Table`, la solution retournée par le modèle `fiable` et enfin la solution retournée par le modèle `optimiste`. Quelle que soit la taille de la population, les autres solutions sont générées de manière aléatoire.

Dans la section suivante, nous allons manipuler les paramètres du système heuristique pour identifier la combinaison optimale et idéale de paramètres qui donnent de bien meilleurs résultats.

5.1.2. Ajustement des paramètres

Les mesures précédentes montrent qu'avec notre jeu de données actuel, le système est capable de donner un type valide pour chaque élément du jeu de données. Notre modèle obtient maintenant le score *accuracy* maximum, ce n'est plus pertinent de faire notre ajustement sur la base de cette métrique.

Par conséquent, nous évaluons le modèle final à l'aide du score de *precision*. Le score de *précision* est le rapport entre les types parfaits (types correctement représentés) et les types corrects (valides). En d'autres termes, la métrique *precision*, nous permet d'évaluer dans quelle mesure le modèle renvoie le bon type avec la meilleure représentation possible.

$$Precision = \frac{Nombre\ d'observations\ correctement\ représentés}{Nombre\ total\ d'observations\ valides} \tag{5.1.2}$$

Parmi les paramètres du système heuristique figurent la taille de la population, le nombre total de générations ou critères d'arrêt ainsi que plusieurs probabilités. Il y a la probabilité de retourner la solution du modèle `lexp_sinfo`, la probabilité de retourner la solution du modèle `H_Table`, la probabilité de retourner un choix `fiable` et la probabilité de retourner un choix `optimiste`.

En raison de la diversité des combinaisons possibles de paramètres, nous avons jugé qu'il n'était pas nécessaire, à ce stade de l'exploration, de tester toutes les combinaisons possibles de paramètres.

5.1.2.1. Les probabilités. En cherchant les meilleurs paramètres de probabilités, nous pourrions potentiellement diversifier la population et ainsi accroître le nombre d'éléments qui peuvent être de bons candidats pour un type donné. C'est pourquoi, nous commençons par chercher les probabilités les plus favorables.

- *Pro_sLs* : probabilité de retourner la solution du modèle `lexp_sinfo`.
- *Pro_sH* : probabilité d'utiliser le modèle H-table. Cette probabilité s'applique uniquement si le modèle `lexp_sinfo` n'a pas été choisi au préalable
- *Pro_sF* : probabilité de retourner un choix fiable pour un nœud donné de l'arbre de recherche. Elle indique simplement que la solution proposée pour ce nœud répond aux critères établis par le modèle fiable. Cette probabilité est applicable uniquement si aucun des modèles précédents n'a été choisi. Le choix de `Pro_sF` à chaque nœud de l'arbre de recherche permettra d'obtenir une solution fiable.
- *Pro_sOp* : probabilité de retourner un choix optimiste; elle n'affecte pas automatiquement les nœuds futures dans l'arbre de recherche. Elle indique simplement que la solution proposée pour ce nœud répond aux critères établis par le modèle optimiste. Cette probabilité est applicable uniquement si aucun des modèles précédents n'a été choisi. Le choix de `Pro_sOp` à chaque nœud de l'arbre de recherche permettra d'obtenir une solution optimiste.
  Si ni le choix fiable ni le choix optimiste n'est pris pour un nœud donné, le système renvoie simplement la forme de base du nœud, c'est-à-dire la forme initiale du nœud avant l'application des règles de transformations.

Lors de la phase d'initialisation de la population, ces probabilités sont utilisées pour introduire de la diversité dans les solutions générées. Lorsqu'une solution aléatoire est générée pour un type donné, les dés sont jetés à chaque nœud de l'arbre (en commençant par la racine), et en fonction du résultat obtenu, un choix est effectué parmi les différents modèles disponibles. Si le choix aléatoire favorise l'utilisation du modèle `lexp_sinfo` (contrôlé par la probabilité `Pro_sLs`) ou du modèle `H_Table` (contrôlé par la probabilité `Pro_sH`), la récursion pour la génération de cette solution s'arrête au nœud courant. En d'autres termes, la solution proposée à ce niveau sera basée sur le modèle `lexp_sinfo` ou le modèle `H_Table`. En revanche, si le modèle `fiable` (contrôlé par la probabilité `Pro_sF`) ou le modèle `optimiste` (contrôlé par la probabilité `Pro_sOp`) sont choisies, la récursion se poursuit dans les sous-nœuds de l'arbre (enfants). Il est important de noter que ces probabilités ne contrôlent pas le choix fiable/optimiste pour l'ensemble du sous-arbre, mais plutôt pour chaque nœud individuellement. Ainsi, la génération de la solution peut continuer en prenant en compte les critères établis par ces modèles.

Notez également que la probabilité d'utiliser le choix fiable ou optimiste au niveau N de l'arbre est calculée en soustrayant de 1 la proportion respective de `Pro_sF` ou `Pro_sOp`, multipliée par la probabilité combinée de choisir `Pro_sH` et `Pro_sLs` à chaque niveau de

Tableau 5.1. Score de *precision* des différentes probabilités avec population=150 et nombre de générations=50

Pro_sF; Pro_sOp \ Pro_sH	1/3	1/4	1/5	1/6	1/7
1/2 ; 1/2	**97**	95	96	96	94
1/3 ; 1/3	**95**	91	94	93	92
1/4 ; 1/4	**95**	91	92	**95**	92
1/5 ; 1/5	95	**96**	91	94	93
1/6 ; 1/6	**96**	94	95	95	93
1/7 ; 1/7	93	95	94	**96**	95

l'arbre (c'est-à-dire `(Pro_sH * Pro_sLs)` élevée à la puissance N. Par conséquent, pour conserver la diversité dans les choix de modèles et éviter de s'appuyer exclusivement sur le choix du modèle `lexp_sinfo` ou `H_Table`, il est recommandé que les probabilités `Pro_sLs` et `Pro_sH` soient relativement petites. Cela permet de garantir une exploration plus équilibrée des différentes options et de maintenir une certaine variété dans la génération des solutions.

Dans les manipulations suivantes, nous allons procéder en deux étapes distinctes. Dans la première étape, nous allons changer les probabilités de l'AG en gardant les autres paramètres tels que la taille de la population et le nombre de générations fixes. Dans la deuxième étape, nous allons changer la taille de la population et le nombre de générations tout en gardant les probabilités de l'AG fixes.

En ce qui concerne les probabilités, nous allons fixer une taille de population de 150 et 50 générations. De plus, compte tenu, des tests faits dans la section "Tests des modèles", la probabilité d'utilisation du modèle `lexp_sinfo`('`Pro_sLs` ') sera fixée à une valeur proche de zéro, soit environ '`0.01 %`'. Le regroupement des probabilités par couple (`Pro_sF`, `Pro_sOp`) est arbitraire et n'a pas d'impact significatif dans le résultat. Seules leurs valeurs influencent le score de *precision*, tandis que la façon dont elles sont regroupées n'affecte pas le résultat final.

Compte tenu des tests du tableau 5.1, nous constatons que certains éléments (types) du jeu de données génèrent plusieurs solutions valides, dont une solution fiable, une solution optimiste et d'autres solutions aléatoires. Pour choisir la meilleure solution parmi celles-ci, nous optons pour celle qui présente la plus petite métrique de taille. Cela nous permet d'évaluer la qualité (lisibilité) des différentes solutions proposées et de sélectionner celle qui convient le mieux. En effet, la taille de la solution est un indicateur important pour estimer

Tableau 5.2. Score de *precision* en ajustant la taille de la population et le nombre de générations

Nombre de générations \ Taille de la population	50	100	150	200
25	92	96	97	**100**
50	92	97	97	**100**
75	97	98	**100**	**100**
100	99	**100**	**100**	**100**

sa complexité et sa facilité de compréhension par les utilisateurs.

En se référant au tableau 5.1, on peut remarquer que les probabilités `Pro_sF`, `Pro_sOp`, et `Pro_sH` obtiennent un score de précision plus élevé (97 %), avec des valeurs respectives de 1/2, 1/2, et 1/3. Pour rappel, le score de *precision*, nous permet d'évaluer dans quelle mesure notre modèle trouve la meilleure représentation possible pour un type. Par conséquent, dans les prochains tests, nous utiliserons ces valeurs afin d'améliorer les performances du modèle.
Même si nous combinons les cinq modèles, il est important de souligner que la meilleure solution retournée par certains éléments du jeu de données n'est pas nécessairement "idéale" ni même "bonne".

5.1.2.2. Taille de la population et nombre de générations. Maintenant que nous avons identifié les paramètres optimaux pour nos probabilités, nous allons les utiliser dans les opérations suivantes où nous ferons varier la taille de la population et le nombre de générations de notre algorithme. Les paramètres utilisés sont 1/2, 1/2 et 1/3 pour les probabilités respectives de `Pro_sF`, `Pro_sOp` et `Pro_sH`.
Rappelons que la probabilité d'utilisation du modèle `lexp_sinfo` est toujours fixée à '`0.01`'.

Le tableau 5.2 démontre qu'il est possible d'obtenir un score de *precision* élevé même avec une petite taille population. Comme mentionné dans la section 4.5.3, il existe une règle empirique suggérant que la taille de la population devrait être proportionnelle à la taille du chromosome. Toutefois, étant donné que nous travaillons principalement avec des chromosomes de petite taille, une taille de population fixe de 100 a été suffisante pour obtenir des résultats satisfaisants.

Grâce à une taille de population de 200 individus, nous obtenons déjà un score de *precision* de '`100%`', ce qui est remarquable. Étant donné que l'augmentation de la taille de la

population peut avoir un impact significatif sur le temps d'exécution, nous avons pris la décision, de ne pas dépasser cette valeur dans nos tests.

Au cours de nos tests, nous avons utilisé des probabilités de 1/2, 1/2 et 1/3 pour `Pro_sF`, `Pro_sOp` et `Pro_sH` respectivement. Nous avons obtenu un score de *precision* de '`97%`' en utilisant une taille de population de 100 et 50 générations, ainsi qu'une taille de population de 150 et 50 générations. Bien que certaines combinaisons de paramètres aient conduit à des scores de *precision* de '`100%`', nous avons opté pour une taille de population 100 et 50 générations pour nos tests ultérieurs. Cette décision a été guidée par plusieurs facteurs. Tout d'abord, en utilisant une taille de population de 100 et 50 générations, nous avons pu générer suffisamment de diversité dans les solutions proposées. Cela a permis d'explorer efficacement l'espace de recherche et d'obtenir des résultats satisfaisants termes de *precision*. De plus, nous avons également pris en compte le temps d'exécution nécessaire pour obtenir ces résultats. L'augmentation de la taille de la population peut entraîner une augmentation significative du temps de calcul. En limitant la taille de la population à 100 avec 50 générations, nous avons pu maintenir des temps d'exécution raisonnables tout en conservant un score élevé de *precision* dans les performances.

Quant à la variation de la taille de la population en fonction de la taille de l'expression régulière (l'élément de notre jeu de données), nous avons choisi de ne pas la faire pour éviter une complexité supplémentaire et un temps de calcul considérablement accru. En outre, nous avons constaté que des tailles de populations fixes fonctionnaient bien pour la plupart des types d'expressions régulières que nous avons testés, ce qui nous a permis de nous concentrer davantage sur l'optimisation des autres paramètres de l'algorithme.

5.1.3. Analyse des résultats

Dans cette section, nous nous concentrons sur l'analyse des résultats obtenus après réglage des paramètres du système heuristique.

Une solution valide renvoyée par le système heuristique est considérée comme étant la meilleure représentation que si elle est identique à la solution optimiste ou si elle possède la métrique de taille la plus petite. Les résultats des tests présentés dans la section 5.1.1.4 ont montré que le système heuristique avait un *accuracy* de '`84 %`' lorsqu'il était utilisé avec uniquement les solutions optimistes. C'est pourquoi nous avons principalement concentré notre analyse sur les types de solutions qui n'étaient pas classés comme optimistes, car ces

Tableau 5.3. Les caractéristiques causant l'échec du modèle optimiste

Nom des fonctions \ Caractéristiques	symboles ##	flèches implicites	arguments actuels implicites
List_mapi	-	-	-
K	-	-	-
Decidable	-	-	-
vnil	-	-	-
vcons	-	-	-
fold	-	-	-
poly-lits	-	-	-
fromInteger	-	-	-
monads	x	x	x
bind	x	x	-
pure	x	-	-
IO_monad	-	-	-
Option_monad	x	x	x
list	x	x	x
do-lib	-	-	-
tuple-lib	-	-	-

derniers représentaient les '16%' des cas les plus complexes et intéressants à étudier.

Pour les échecs avec la solution optimiste, nous utilisons les types des fonctions suivantes: '`List_mapi`', '`K`', '`Decidable`', '`vnil`', '`vcons`', '`fold`', '`poly-lits`', '`fromInteger`', '`monads`', '`bind`', '`pure`', '`IO_monad`', '`Option_monad`', '`list`', '`do-lib`', '`tuple-lib`'. Alors pour tester ces types avec le système heuristique, nous allons utiliser différentes caractéristiques de comparaison, dans le tableau 5.3. En ce qui concerne la colonne *flèches implicites*, nous regardons uniquement les flèches implicites situées entre deux flèches explicites ("problème 1" de la section 4.4).

Pour certains types, il est nécessaire d'écrire explicitement les flèches implicites afin d'assurer une représentation correcte. Ci-dessous, nous présentons des exemples de types :

```
Bool -> (a : Type) ≡> a -> a -> a
(a : Type) ≡> a -> (b : Type) ≡> b -> a
```

Si, dans la ligne **2**, nous remplaçons l'argument formel '`b`' par une méta-variable '`?b`', le compilateur interprétera cette méta-variable comme un argument actuel déclaré avant la première flèche explicite. Par conséquent, l'expression de type sera modifiée comme suit:

```
(a : Type) ≡> (b : Type) ≡> a -> b ≡> b -> a
```

Cette modification entraîne une différence significative avec la représentation d'origine, rendant ainsi le résultat invalide. C'est pourquoi il est obligatoire d'avoir une flèche explicite entre ces deux flèches implicites. Nous pouvons également observer la même situation dans la ligne **1**, où l'argument actuel '`(a : Type)`' est présent entre deux flèches normales.

Dans la section 1.1 de l'introduction, nous avons parlé de la complexité de certains types dans Typer. Cette complexité est particulièrement visible dans des types tels que '`monads`', '`list`', '`bind`', '`pure`' et '`Option_monad`'. Pour ces types, aucune solution autre que la solution insatisfaite (illisible) n'est acceptée par le compilateur actuellement.
Dans ces cas, il peut être nécessaire d'utiliser la notation $< module > . < elem >$ pour se référer aux éléments d'un module dans le code ("problème 2" de la section 4.4). Cependant, la structure de l'arbre d'analyse (lexp) ne contient pas de référence au nom du module. Au lieu de cela, elle fait référence directement aux définitions qui le composent. Par exemple, si nous prenons le module suivant:

```
MyMod = module
          type IntList
             | mynil
             | mycons Int IntList;
              ...
        end
```

Le type de `MyMod.mynil` ressemblera à :

```
let IntList = typecons mynil (mycons Int IntList)
in IntList
```

Dans ce cas, notre système est incapable de découvrir qu'il est nécessaire d'utiliser la notation '`MyMod.mynil`' pour accéder à l'élément `mynil` du module `MyMod`, plutôt que de simplement utiliser `IntList`. À moins qu'une telle correspondance ne soit déjà présente dans la table de hachage, résultant d'un code antérieur.

En augmentant la taille de la population et le nombre de générations, le nombre de solutions valides potentielles peut certainement augmenter. Cependant, cela ne garantit pas nécessairement l'amélioration de la meilleure représentation de la solution retournée par le système heuristique. En d'autres termes, bien que le nombre de solutions potentielles augmente avec la taille de la population et le nombre de générations, cela ne garantit pas que la meilleure solution sera améliorée.

5.2. Performances

Équipé d'un ordinateur portable exécutant le système d'exploitation '`Ubuntu 18.04`', nous pouvons maintenant analyser l'impact de notre système heuristique sur les performances du compilateur Typer. Voici les données techniques de la machine :

```
ubuntu@ubuntu-Lenovo-V15-IIL:~$ sudo hwinfo --short
cpu:                 Intel® Core i5-1035G1 CPU @ 1.00GHz × 8
Intel RAM memory:                                    19,3 GiB
```

La compilation peut être influencée par de nombreux paramètres, ce qui nécessite de trouver les paramètres optimaux pour mesurer les performances du système heuristique. Ainsi, nous allons ajuster plusieurs paramètres, notamment la taille de la population, le nombre de générations ainsi que les autres probabilités de notre système heuristique. Nous allons toutefois nous concentrer sur les performances des types complexes, étant donné que ceux-ci sont les seuls à requérir l'utilisation de l'algorithme génétique dans notre approche.

5.2.1. Analyse

La performance du système heuristique varie significativement en fonction des paramètres considérés. Alors pour mesurer le temps d'exécution du système heuristique pour retrouver une solution valide pour un type donné, nous allons effectuer 50 exécutions consécutives du système heuristique avec les mêmes paramètres. Ces paramètres incluent les probabilités du système heuristique (avec `Pro_sLs`=0.01, `Pro_sH`= 1/3, `Pro_sF`= 1/2 et `Pro_sOp`= 1/2), une population de 100 individus et 50 générations. Pour faciliter l'analyse, nous allons créer une représentation graphique des mesures (figure 5.2).

En utilisant le coefficient de variation pour analyser les performances des différents types, nous avons fixé un temps d'exécution maximal d'**une seconde**. Le coefficient de variation est calculé en divisant l'écart-type des temps d'exécution par la moyenne des temps d'exécution. Cette mesure nous permet de quantifier la dispersion des temps d'exécution par rapport à leur moyenne. La figure 5.2 met en évidence que le type '`fromInteger`', présente une dispersion plus importante des données, illustrée par un écart type maximal. Cette observation peut s'expliquer par le fait que ce type est relativement petit en taille mais possède une structure complexe, avec des arguments assez variés. En revanche, pour les autres types, le coefficient de variation est relativement faible, indiquant ainsi une plus grande stabilité des mesures de performances pour ces types.

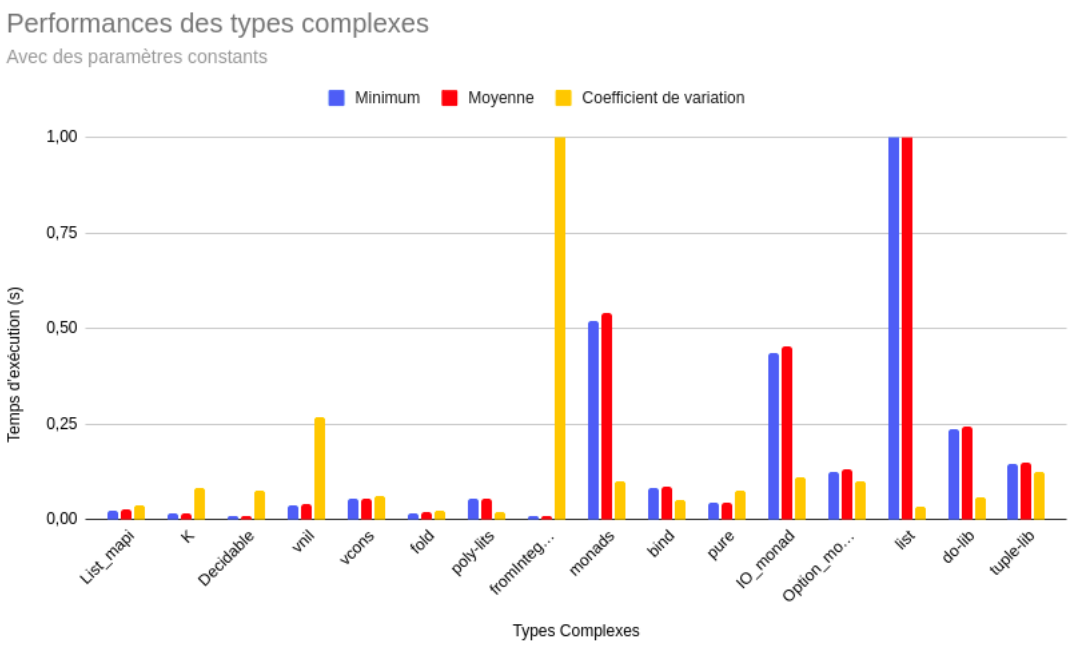

Fig. 5.2. Performances des types complexes avec des paramètres constants

Pour étudier l'impact de chaque paramètre sur les performances du système heuristique, nous allons modifier un paramètre à la fois, tout en maintenant les autres constants. Nous supposons que la variation du nombre de générations donnera des résultats similaires à la variation de la taille de la population (temps d'exécution augmentant de manière linéaire). Par conséquent, nous avons décidé de nous concentrer principalement sur la variation des probabilités du système heuristique et de la taille de la population.

5.2.2. Les probabilités du système heuristique

Afin d'évaluer le temps d'exécution de notre système heuristique, nous utiliserons une population de taille fixe, soit 100 individus. Nous effectuerons des tests en variant ensemble les valeurs des probabilités `Pro_sH`, `Pro_sOp` et `Pro_sF`, et en gardant `Pro_sLs` fixée à '`0.01%`'. Le choix de maintenir la même valeur de probabilité est arbitraire en raison de contrainte de temps. Cette approche nous permettra de comparer l'efficacité du système heuristique dans des conditions variées et de déterminer l'impact des différentes probabilités sur les performances globales du système.

Pour la figure 5.3, nous avons suivi la démarche suivante : pour chaque type complexe, nous avons sélectionné la probabilité qui présentait le temps d'exécution maximal et l'avons utilisée comme référence. Ensuite, nous avons tracé des histogrammes pour représenter les

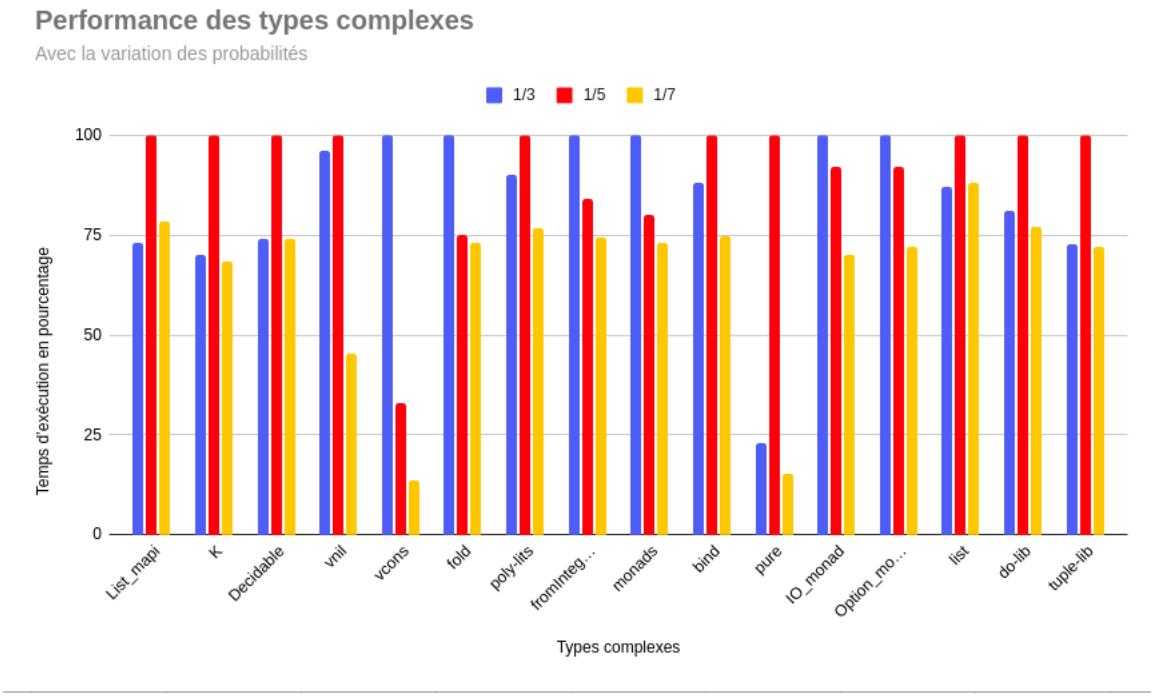

Fig. 5.3. Performance des types complexes avec variation des probabilités

temps d'exécution des autres probabilités par rapport à cette référence.

En examinant la figure 5.3, nous pouvons observer que la variation du temps n'est pas uniforme entre les différents types de données. Pour le type `pure`, le temps d'exécution avec les probabilités 1/3 et 1/7 est négligeable par rapport au temps d'exécution de la probabilité 1/5. Cette différence peut être attribuée au fait que les probabilités 1/3 et 1/7 ne favorisent peut-être pas la représentation des arguments implicites dans le code généré. Cependant, dans l'ensemble, une analyse plus approfondie du temps d'exécution indique que la probabilité de 1/7 est généralement associée à des temps d'exécution plus courts parmi les trois probabilités considérées.

5.2.3. Taille de la population

Nous allons mesurer le temps d'exécution de notre système heuristique avec les probabilités `Pro_sLs`=0.01, `Pro_sH`= 1/3, `Pro_sF`= 1/2 et `Pro_sOp`= 1/2. Nous prévoyons que le temps d'exécution augmentera de manière linéaire avec la taille de la population.

Dans la figure 5.4, nous avons tronqué le graphe en fixant une limite de temps d'exécution maximale d'`une seconde` pour une meilleure visualisation des résultats. Nous constatons que le temps d'exécution des types complexes autres que '`monads`' et '`list`' est insignifiant

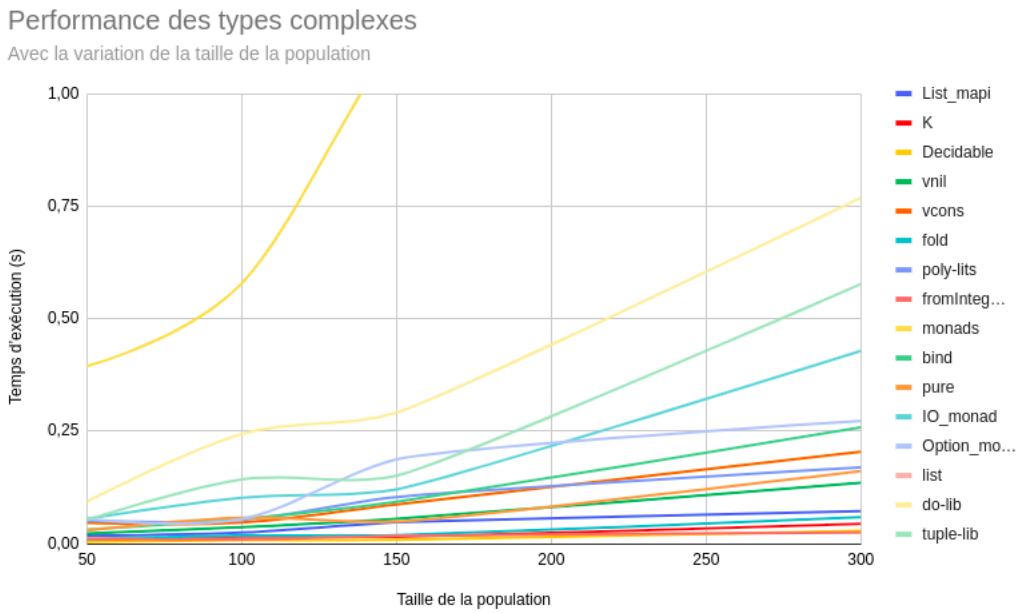

Fig. 5.4. Performance des types complexes avec variation de la taille de la population

en raison de leur petite taille, tandis que pour ces deux types, le temps d'exécution est particulièrement plus élevé et croît de manière linéaire avec la taille de la population. En effet, une analyse plus approfondie révèle que leur temps d'exécution est proportionnel à la taille de l'entrée, ce qui confirme notre hypothèse de linéarité. La courbe du temps d'exécution de '`list`' n'est pas visible car elle se situe en haut des autres courbes.

Toutefois, les résultats indiquent que le temps d'exécution est davantage influencé par la taille de la population que par les probabilités de notre système heuristique. Ceci suggère que la taille de la population est un facteur critique pour la performance de l'algorithme.

Chapitre 6

Conclusion

Ce travail s'est concentré sur "l'amélioration des messages d'erreurs Typer par algorithme génétique". Notre contribution principale a été la mise en place d'un système heuristique permettant au compilateur Typer de présenter les types d'une manière plus compréhensible pour les programmeurs, améliorant ainsi l'affichage des messages d'erreurs de compilation. Pour y arriver, nous avons dû apporter quelques modifications au compilateur Typer afin de préserver l'information d'origine.

Notre système heuristique repose sur une combinaison de trois approches différentes, dont les algorithmes génétiques sont le point clé. En général, le temps d'exécution, est acceptable, même sans optimisation particulières, souvent de l'ordre de 0.1 seconde. Cependant, dans certains cas, le temps d'exécution peuvent être un peu long que souhaité, atteignant parfois plusieurs secondes, surtout lorsque la complexité du type est élevée.

En termes de qualité des types suggérés, en général, ils sont lisibles et compréhensibles pour les programmeurs. Cependant, dans certains cas plus complexes, les types suggérés peuvent être un peu moins évidents à comprendre, mais restent valides.

Malgré des résultats de performance prometteurs, le système heuristique est incapable pour le moment de donner une solution autre que la solution insatisfaisante (illisible) pour certains types complexes de Typer, tels que `list`, `monads`, `IO_monad`, `Option_monad` et `list` de notre jeu de données.

Il convient de rappeler que l'objectif de ce mémoire était principalement exploratoire. Ainsi, nous terminons ce travail en identifiant ce que nous pensons être les prochaines étapes pour améliorer les messages d'erreurs du compilateur Typer.

Pour améliorer le modèle `lexp_sinfo`, nous envisageons de faire une distinction entre nos termes pendant l'élaboration. En effet, il a été observé qu'en pratique la relation entre une '`sexp`' et son équivalent de code '`lexp`' peut être soit une élaboration, soit une inférence. Autrement dit, soit la '`lexp`' représente la même chose que la '`sexp`', soit elle représente son type. Il serait donc utile de prendre en compte cette distinction pour savoir quand il est pertinent d'utiliser la '`sinfo`' (dérivée de la '`sexp`'). En prenant en compte cette distinction, il serait possible d'améliorer les analyses réalisées en fonction de l'utilisation ou non de la '`sinfo`'.

Pour résoudre les problèmes liés aux champs de module, il pourrait être utile d'offrir aux librairies (comme celle qui définit les modules de Typer) une manière d'ajouter des "règles de réécriture" à notre code. Ces règles de réécriture permettraient de transformer des types complexes en expressions plus simples et plus utilisables dans le contexte spécifique de la librairie.

L'opérateur de croisement pourrait être modifié de façon à ce qu'il ne génère que des solutions réalisables avec des scores de performance toujours meilleurs que celle des solutions de la génération précédente.

Références bibliographiques

[1] T. Menzies A. S. Sayyad, K. Goseva-Popstojanova et H. Ammar : On parameter tuning in search based software engineering: A replicated empirical study. *In* Myra B. Cohen et Mel Ó Cinnéide, éditeurs : *In Proceedings of the International Workshop on Replication in Empirical Software Engineering Research*, volume 6956, pages 84–90. IEEE Computer Society, October 2013.

[2] T. Menzies A. S. Sayyad, K. Goseva-Popstojanova et H. Ammar. : On parameter tuning in search based software engineering: A replicated empirical study. pages 84–90, October 2013.

[3] Jarmo T Alander : On optimal population size of genetic algorithms. *In CompEuro 1992 Proceedings computer systems and software engineering*, pages 65–70. IEEE, 1992.

[4] Prathibha A. Ballal, H. Sarojadevi et Harsha P S : Compiler optimization: A genetic algorithm approach. *International Journal of Computer Applications*, 112(10):9–13, February 2015.

[5] Jean-Alexandre Barszcz : Typer a de la classe, le polymorphisme ad hoc dans un langage avec des types dépendants et de la métaprogrammation. Mémoire de D.E.A., Université de Montréal, Mai 2021.

[6] Mike Beaven et Ryan Stansifer : Explaining type errors in polymorphic languages. *Association for Computing Machinery (ACM) Letters Programming Languages Systems*, 2(1–4):17–30, mar 1993.

[7] Raphaël Cerf : *Une théorie asymptotique des algorithmes génétiques*. Thèse de doctorat, 1994.

[8] Sheng Chen et Martin Erwig : Counter-factual typing for debugging type errors. *In* Suresh Jagannathan et Peter Sewell, éditeurs : *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 583–594. ACM, 2014.

[9] Olaf Chitil : Compositional explanation of types and algorithmic debugging of type errors. *In* Benjamin C. Pierce, éditeur : *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, pages 193–204. ACM, 2001.

[10] Ryan Culpepper et Matthias Felleisen : Taming macros. *In* Gabor Karsai et Eelco Visser, éditeurs : *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 de *Lecture Notes in Computer Science*, pages 225–243. Springer, 2004.

[11] Ryan Culpepper et Matthias Felleisen : Fortifying macros. *In* Paul Hudak et Stephanie Weirich, éditeurs : *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 235–246. ACM, 2010.

[12] Pierre Delaunay : Implémentation d'un langage fonctionnel orienté vers la méta programmation. Mémoire de D.E.A., Mars 2017.

[13] Robert E. Dorsey et Walter J. Mayer : Genetic algorithms for estimation problems with multiple optima, nondifferentiability, and other irregular features. *Journal of Business & Economic Statistics*, 13(1):53–66, 1995.

[14] Oussama El Gerari : *Contribution à l'amélioration des techniques de la programmation génétique*. Theses, Université du Littoral Côte d'Opale, décembre 2011.

[15] David E. Goldberg : *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[16] Christian Haack et Joe B. Wells : Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[17] Frederick Hayes-Roth : Review of "adaptation in natural and artificial systems by john h. holland", the u. of michigan press, 1975. *stands for Special Interest Group on Artificial Intelligence (ACM)*, 53:15, 1975.

[18] Bastiaan Heeren : *Top quality type error Messages*. Thèse de doctorat, Utrecht University, Netherlands, 2005.

[19] David Herman et Philippe Meunier : Improving the static analysis of embedded languages via partial evaluation. *In* Chris Okasaki et Kathleen Fisher, éditeurs : *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, pages 16–27. ACM, 2004.

[20] David Herman et Mitchell Wand : A theory of hygienic macros. *In* Sophia Drossopoulou, éditeur : *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 de *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

[21] John H. Holland : *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[22] Patrik Jansson et Johan Jeuring : Polyp—a polytypic programming language extension. *In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 470–482, New York, NY, USA, 1997. Association for Computing Machinery (ACM).

[23] Oukseh Lee et Kwangkeun Yi : Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages Systems*, 20(4):707–723, 1998.

[24] Benjamin S. Lerner, Matthew Flower, Dan Grossman et Craig Chambers : Searching for type-error messages. *SIGPLAN Notices*, 42(6):425–434, jun 2007.

[25] Kim-Fung Man, Wallace Kit-Sang Tang et Sam Kwong : Genetic algorithms: concepts and applications [in engineering design]. *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Industrial Electronics*, 43(5):519–534, 1996.

[26] B. J. McAdam : *Repairing type errors in functional programs*. Thèse de doctorat, 2002.

[27] Ropas Memo, Oukseh Lee et K. Yi Lee : A generalized let-polymorphic type inference algorithm. *Journal of KIISE:Software and Applications*, 28(1):73–89, 2000.

[28] Melanie Mitchell : *An introduction to genetic algorithms*. Massachusetts Institute of Technology (MIT) Press, 1998.

[29] Stefan Monnier : Typer : Ml boosted with type theory and scheme. *In Journées Francophones des Langages Applicatifs*, page pages 193–208, 2019.

[30] Scott Owens, M. Flatt, O. Shivers et Benjamin McMullan : Lexer and parser generators in scheme. *In Proceedings of the Scheme Workshop*. Scheme workshop, 2004.

[31] Dania Isabel Ahumada Pardo : Una aproximación evolucionista para la generación automática de sentencias sql a partir de ejemplos. Mémoire de D.E.A., Université de Montréal, Mars 2015.

[32] François Pottier : Hindley-Milner elaboration in applicative style: functional pearl. *In* Johan Jeuring et Manuel M. T. Chakravarty, éditeurs : *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 203–212. ACM, 2014.

[33] Takfarinas Saber, David Brevet, Goetz Botterweck et Anthony Ventresque : Reparation in evolutionary algorithms for multi-objective feature selection in large software product lines. *SN Computer Science*, 2(3), mar 2021.

[34] Sadiq M. Sait et Habib Youssef : *Iterative computer algorithms with applications in engineering - solving combinatorial optimization problems*. Institute of Electrical and Electronics Engineers (IEEE), 1999.

[35] Alejandro Serrano : Type error customization for embedded domain-specific languages. *In Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 109–176, Utrecht, Pays-Bas, January 2018. Association for Computing Machinery (ACM), University Utrecht.

[36] Mitchell Wand : Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[37] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis et Simon Peyton-Jones : Diagnosing type errors with class. *ACM Special Interest Group on Programming Languages*, 50(6):12–21, jun 2015.

[38] Ágoston E. Eiben, Robert Hinterding et Zbigniew Michalewicz : Parameter control in evolutionary algorithms. *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[39] S. Özyildirim : Three-country trade relations: A discrete dynamic game approach. *Computers & Mathematics with Applications*, 32(5):43–56, 1996.