Synthèse d’article:

Data Flow Refinement Type Inference

M2 STL

Edwin Ansari
Lounes Brahimi

09/11/2021

Introduction:

Les types de raffinement permettent une vérification légère des programmes fonctionnels. Cependant Les méthodes actuelles d'inférence de type statique utilisent les clauses de horn qui ont des limitations à cause de leurs abstractions. L'article propose un système de type paramétré par un domaine abstrait choisi et le niveau souhaité de sensibilité au contexte. Il est basé sur l'interprétation abstraite et permet de dériver des algorithmes d'inférence de type.
Le système proposé est complet, efficace et corrige les lacunes présentes dans les systèmes de raffinement de type existant. Ce dernier a été implémenté en OCaml et est nommé $Drift$, des expérimentations démontrent que l'outil propose une bien meilleure solution.

Types de raffinement :

Les types de raffinement sont des types de base enrichis par des prédicats, qui permettent de spécifier et de vérifier automatiquement les propriétés sémantiques du code.
Par exemple :
Le type de l'opérateur de division indique qu'étant donné deux entiers, on retourne un entier.

$$div :: int \rightarrow int \rightarrow int$$

En utilisant les prédicats on peut raffiner le type ci-dessus de manière que le diviseur ne soit jamais égale à 0, cela nous permet plus tard de vérifier statiquement l'absence de cette erreur.

$$div :: int \rightarrow \{v: int \mid v \neq 0\}$$

Cependant une telle vérification exige que l'expression du diviseur soit aussi contrainte par les types de raffinement appropriés.

Problèmes d'inférence de types:

Le travail porte sur des systèmes de types qui améliorent les systèmes classiques, tels que Hindley-milner, cela en infèrent des invariants globaux de programmes d'ordre supérieur en utilisant la notion de raffinements sur les types de base classiques.

Considérons la fonction de Fibonacci:

$$let\ rec\ fib\ x\ =\ if\ x\ >=\ 2\ then\ fib(x\ -\ 1)\ +\ fib(x\ -\ 2)\ else\ 1$$

L'algorithme d'inférence de type de raffinement typique fonctionne comme suit:

1) Une inférence de type Hindley-Milner est effectué pour déduire la forme de base du type, le type inféré pour la fonction est: $int \rightarrow int$
2) Le type est raffiné en introduisant des prédicats $\varphi_1$et φ_2 :

   $x : \{v: int \mid \varphi_1(v)\} \rightarrow \{v: int \mid \varphi_2(v, x)\}$

3) L'algorithme dérive ensuite des clauses de Horn comme :
   $\varphi_1(x) \wedge x \geq 2 \wedge \nu = x - 1 \Rightarrow \varphi_2(\nu)$

Cette dernière signifie que si le prédicat φ_1 est supérieur à 2 pour une entrée donnée x alors il est aussi vrai pour l'entrée (x-1), cela est déduit de l'appel récursif sur la branche "then". Ainsi il dérive un ensemble de prédicats monomial et infer une assignation pour chaque φ_i satisfaisant toutes les clauses. Dans le cas de fibonacci et en prenant pour choix $Q = \{0 \leq \nu,\ 0 > \nu,\ \nu < 2,\ \nu \geq 2\}$ le type final déduit pour la fonction fib sera:
$x : \{v: int \mid 0 \leq \nu\} \rightarrow \{v: int \mid 0 \leq v\}$

Supposons que nous voulions prouver que la fonction de fibonacci est croissante en utilisant le prédicat Q. Donc on souhaite montrer que $\varphi_2(v, x)$ implique x ≤ v alors que ce dernier n'est pas inductif.
Il nous faut un domaine plus riche, on peut donc étendre l'ensemble des prédicats en permettant de comparer une variable entière à 1 et à x en ajoutant x ≤ v et 1 ≤ v.

Pour conclure, les types liquides extraient les prédicats de la syntaxe du programme et permettent aux utilisateurs de les spécifier, mais connaître à l'avance l'ensemble des prédicats nécessaires est vraiment difficile. Le problème se pose si nous avions besoin d'un domaine infini de raffinements, alors il serait impossible d'obtenir l'inférence de type sur les domaines avec les algorithmes classiques.

Considérons le programme suivant :

```
1  let apply f x = f x and g y = 2 * y and h y = -2 * y
2  let main z =
3    let v = if 0 <= z then (apply_i g)_j z else (apply_k h)_ℓ z in
4    assert (0 <= v)
```

La fonction g multiplie l'entier d'entrée par 2 et la fonction h le multiplie par -2, $apply$ prend en entrée une fonction et l'applique à x.
La fonction $main$ applique g à un entier non négatif et la fonction h à une entrée négative, retournant ainsi toujours un nombre positif.
Quel que soit le domaine des raffinements de type, nous ne parviendrons pas à vérifier que v est positif. Le types de g et h sont obscurcient par f, de même pour les z positifs et négatifs par x.
L'inférence de apply ne fait pas de distinction entre les deux fonctions g et h car elle n'est pas consciente que g doit être appliqué aux entiers positifs et h doit être appliqué aux entiers négatifs. L'inférence conclut donc que l'entrée f d'apply peut être appelée et produire n'importe quel entier, ce qui n'est pas suffisant pour prouver que le résultat retourné est toujours positif.

$$apply : (x: \{v : int \mid true\} \to \{v : int \mid true\}) \to \{v : int \mid true\}$$

$Drift$ contourne ce problème en appelant la fonction apply selon différents contextes d'appel, si nous définissons ce contexte d'appel comme des emplacements où l'appel à $apply$ se produit (dans cet exemple i et k) comme on peut le voir sur la fonction, l'inférence de $apply$ peut associer g à des entrées positives et h à des entrées négatives.

$$i :: (x : \{v : int \mid v >= 0\} \to \{v : int \mid v >= x\}) \to \{v : int \mid v >= 0\} \; and$$
$$k :: (x : \{v : int \mid v < 0\} \to \{v : int \mid v > x\}) \to \{v : int \mid v > 0\}$$

Contrairement aux algorithmes classiques on peut maintenant prouver que le résultat retourné est toujours positif.

Le principal défi et problème est de construire systématiquement un raffinement de flux de données, des algorithmes d'inférence de type qui se terminent, prennent en charge les domaines infinis et finis de raffinements de type, prennent en charge la sensibilité au

contenu et n'exigent pas de l'utilisateur de créer ou de fournir le typage, règles et l'algorithme d'inférence de type.

Système de Type de Raffinement de Flux de Données Paramétrique:

Le système de type prend deux paramètres : un ensemble fini de piles abstraites symbolisé par **Ŝ** et un treillis complet de types de raffinement de base $\langle R^t , \sqsubseteq b , \bot b , \top b , \sqcap b , \sqcup b \rangle$.

Langage : Un programme est une expression fermée, le langage définit formellement les variables $x \in$ Var, les constantes $c \in$ Cons , les applications de fonctions et les lambda abstractions.

$$e \in Exp :: = \ c \mid x \mid e_1 \ e_2 \mid \lambda x. e$$

Note : Une expression "**e"** est dite fermée si toutes les occurrences de variables qu'elle contient sont liées dans des lambda abstractions.

Annotation et emplacement : Pour toute expression **e**, chacune de ses sous-expressions est annotée de façon unique qui fait référence à un emplacement spécifique dans l'ensemble **Loc**. Par exemple on peut nommer les emplacements d'une expression par $a,\ b\ et\ l$: $(e_a\ e_b)_l$.

Note : Var $\subseteq Loc$.

Types : Un raffinement de type de base **b** $\in R_t$ est équipé d'une portée implicite $X \subseteq$ **Var** , qui signifie que **b** représente une relation entre une valeur primitive tel qu'un entier et d'autres valeurs liées aux variables dans **X**. Pour $X \subseteq$ **Var** , R^t_x représente l'ensemble de tous les types de raffinement de base de la portée de X.

Prenant par exemple: R^{lia}_x **::=** $\bot\ b$ **|** $\top b$**| {**v **: int |** $\phi\ (X)$**}**

L'un des types de bases dans R^{lia}_x est **{**v **: int |** $x \leq v \wedge$ **1 ≤** v**}** ici, $\phi\ (X)$ qui est une contrainte linéaire convexe sur des variables entières,

Les types de raffinement de base sont étendus aux types de raffinement de flux de données comme suit :

$$t \in \mathcal{V}^{\mathrm{t}}_X ::= \bot^{\mathrm{t}} \mid \top^{\mathrm{t}} \mid b \mid x : \boldsymbol{t} \qquad b \in \mathcal{R}^{\mathrm{t}}_X \qquad x : \boldsymbol{t} \in \mathcal{T}^{\mathrm{t}}_X \stackrel{\mathrm{def}}{=} \Sigma x \in Var \setminus X. \hat{\mathcal{S}} \to \mathcal{V}^{\mathrm{t}}_X \times \mathcal{V}^{\mathrm{t}}_{X \cup \{x\}}$$

1. le type $\perp_t$ signifie “inaccessible”.
2. $\top_t$ représente une erreur de type.
3. Un type de raffinement de base **b**.
4. **x : t** est la capture d’un type de fonction dépendante distincte x **:** $t_i \rightarrow t_o$ par une pile abstraite **Ŝ**, cette dernière représente les appels concrets.

Exemple: Pour illustrer l’efficacité de cette méthode reprenant la fonction $apply$ déjà vu, on a expliqué que le typage le plus précis qu’on pouvait inféré est:

f **: (**y **: int →** **int)** **→** x **: int →** **int .**

Sachant que f fait référence à la fonction $apply$, y à la fonction g et h. On définit l’ensemble des piles abstraites **Ŝ =** Loc **∪ {**ϵ**}** . On utilise les éléments de ce dernier afin de distinguer le type de fonction en entrée à l’aide des annotations d’emplacements où les fonctions représentées sont appelées. Enfin, avec V^t, R^{lia}_x et $\hat{S}^1$ on obtient un typage plus précis :

f **: [**$i \triangleright y$ **: [**$j \triangleright$ **{**v **| 0 ≤** v**}** **→ {**v **|** v **= 2**y**}]→** x **: [**$j \triangleright$ **{**v **| 0 ≤** v**}** **→ {**v **|** v **= 2**x**}],**
$k \triangleright y$ **: [ℓ ▷ {**v **| 0 >** v**}** **→ {**v **|** v **= −2**y**}]** **→** x **: [ℓ ▷ {**v **| 0 >** v**}** **→ {**v **|** v **= −2**x**}] .**

Note : Pour **t =** x **: t** et t **(Ŝ) =** $\langle t_i , t_o \rangle$ On dit que t a été appelé en **Ŝ** et on écrit x **: [Ŝ ▷** $t_i \rightarrow t_o$ **].**

Environnements de typage et opérations sur les types :
Dans un premier temps l’article introduit quelques opérations pour construire et manipuler des types :

1. Le type de raffinement de base qui fait abstraction d'une seule constante, $c \in$ **Cons** est noté $[\nu = c]^t$, exemple pour $c \in$ **Z,** alors $[\nu = c]^t$ **= {**v**:int |** v **=** c**}**.
2. Étant donné un type t sur la portée **X** et un type t' sur la portée $X' \subseteq X$**\{**x**}** , on note t **[**$x \leftarrow t'$**]** le type obtenu en étendant t à x avec l'information fournie par le type t'. Exemple pour **b = {**v **|** ϕ **(**X**)}** et **b ′ = {**v **|** ϕ **′ (**X **′)}**, alors **b[**$x \leftarrow$ **b′] = {**v**|** ϕ **(**X**)** $\wedge$ ϕ **′ (**X **′)** **[**x**/**v**]}** .
3. Soit une variable $x \in X$ et $t \in V^t_x$, l'opération $t\ [\nu = c]^t$ étend t avec la valeur liée à x, exemple pour **b = {**v **|** ϕ **(**X**)}** alors **b** $[\nu = c]^t$ **= {**v **|** ϕ **(**X**)** $\wedge$ ν **=** x**}** .

Un environnement de typage avec une portée X est une fonction Γ^t tel que t **[**$x \leftarrow t'$**]** est représenter par l'opération t **[**Γ^t**]** qui étend t selon les contraintes sur les variables dans X.

Note : Les piles abstraites sont équipées d'une opération de concaténation $\hat{:}$ **:** Loc **×** $\hat{\mathbf{S}}$ **→** $\hat{\mathbf{S}}$ qui ajoute un emplacement de site d'appel i, on le note $i \hat{:} \hat{\mathbf{S}}$.

Règles de typage :

Les règles de typage ont la forme Γ^t**,** $\hat{\mathbf{S}} \vdash e$ **:** t et se basent sur la relation de sous-typage t **<:** t'.

Règles de sous-typage de raffinement de flux de données :

$$\text{S-BOT}\ \frac{t \neq \top^t}{\bot^t <: t} \qquad \text{S-BASE}\ \frac{b_1 \sqsubseteq^b b_2}{b_1 <: b_2} \qquad \text{S-FUN}\ \frac{\boldsymbol{t}_1(\hat{S}) = \langle t_{i1}, t_{o1}\rangle \qquad \boldsymbol{t}_2(\hat{S}) = \langle t_{i2}, t_{o2}\rangle \qquad t_{i2} <: t_{i1} \qquad t_{o1}[x \leftarrow t_{i2}] <: t_{o2}[x \leftarrow t_{i2}]}{x : \boldsymbol{t}_1 <: x : \boldsymbol{t}_2}\ \forall \hat{S} \in \hat{\mathcal{S}}$$

Fig. 1. Data flow refinement subtyping rules

1. La règle **S-BOT** déclare que $\bot_t$ est un sous-type de tous les autres types à l'exception de $\top_t$.
2. **S-BASE** définit le sous-typage sur les types de raffinement de base, qui fait simplement référence à l'ordre partiel $\sqsubseteq b$ sur R_t.
3. La règle **S-FUN** rappelle la règle familière de sous-typage contravariant pour les types de fonction, à l'exception qu'elle utilise toutes les entrées présente des piles abstraites.

Règles de typage de raffinement de flux de données :

$$\text{T-VAR}\ \frac{\Gamma^t(x)[v{=}x]^t[\Gamma^t] <: t[v{=}x]^t[\Gamma^t]}{\Gamma^t, \hat{S} \vdash x : t} \qquad \text{T-APP}\ \frac{\Gamma^t, \hat{S} \vdash e_i : t_i \qquad \Gamma^t, \hat{S} \vdash e_j : t_j \qquad t_i <: x : [i \hat{:} \hat{S} \triangleleft t_j \rightarrow t]}{\Gamma^t, \hat{S} \vdash e_i\, e_j : t}$$

$$\text{T-CONST}\ \frac{[v{=}c]^t[\Gamma^t] <: t}{\Gamma^t, \hat{S} \vdash c : t} \qquad \text{T-ABS}\ \frac{\Gamma_i^t = \Gamma^t.x : t_x \qquad \Gamma_i^t, \hat{S}' \vdash e_i : t_i \qquad x : [\hat{S}' \triangleleft t_x \rightarrow t_i] <: t|_{\hat{S}'}}{\Gamma^t, \hat{S} \vdash \lambda x.\, e_i : t}\ \forall \hat{S}' \in t$$

1. La règle **T-CONST** indique que $[\nu = c]^t$ étendu à Γ^t doit être un sous-type de **t**, ainsi, le sous-typage peut être défini sans suivre les environnements de typage explicites.
2. La règle **T-VAR**, ressemblant à T-CONST, le type Γ^t**(x)** lié à x doit être un sous-type de t étendu avec l'égalité ν **=** x et les contraintes environnementales.
3. La règle **T-APP** pour le typage des applications $e_i\, e_j$, le type t_i de e_i doit être un sous-type du type de fonction x **: [**$i \,\hat{:}\, \hat{\mathbf{S}}$ **:** $t_j \rightarrow t$**]** où t_j est le type de e_j et t est le type du résultat de l'application. L'extension de la pile des appels abstraits $i \,\hat{:}\, \hat{\mathbf{S}}$ force t_i à avoir un type d'entrée correct.
4. La règle **T-ABS** à la même définitions classique pour un typage d'une lambda abstraction à l'exception qu'elle fait la vérification pour toutes les piles abstraites **Ŝ′** auquel t a été appelé.

Note: Toutes les variables libres de **e** sont dans Γ^t.

Sémantique Des Flux De Données :

L'objectif de cette section est d'introduire une nouvelle sémantique de type de raffinement de flux de données.

Domaines sémantiques :
Chaque étape d'exécution est modélisée en tant que nœud $n \in$ **N.**

$$n \in \mathcal{N} \stackrel{\text{def}}{=} \mathcal{N}_e \cup \mathcal{N}_x \qquad \mathcal{N}_e \stackrel{\text{def}}{=} Loc \times \mathcal{E} \qquad \mathcal{N}_x \stackrel{\text{def}}{=} Var \times \mathcal{E} \times \mathcal{S}$$
$$S \in \mathcal{S} \stackrel{\text{def}}{=} Loc^* \qquad E \in \mathcal{E} \stackrel{\text{def}}{=} Var \rightharpoonup_{fin} \mathcal{N}_x \qquad M \in \mathcal{M} \stackrel{\text{def}}{=} \mathcal{N} \rightarrow \mathcal{V}$$
$$v \in \mathcal{V} ::= \bot \mid \top \mid c \mid \boldsymbol{v} \qquad \boldsymbol{v} \in \mathcal{T} \stackrel{\text{def}}{=} \mathcal{S} \rightarrow \mathcal{V} \times \mathcal{V}$$

1. Les nœuds d'expressions N_e représentent des points d'évaluation d'une expression à un endroit donné dans l'environnement courant.
2. Les nœuds de variables N_x qui représentent les points d'exécution de la liaison d'une valeur à une variable d'entrée. Les nœuds de variables sont modélisés en utilisant l'environnement actuel, la variable d'entrée elle-même et la pile d'appels pour représenter de façon unique l'évaluation actuelle du corps de la fonction.
3. Les piles **S** sont modélisées comme des séquences d'emplacements d'appel induisant l'appel en cours.
4. Les environnements mappent les variables aux nœuds de variables correspondants.

Le produit final de la sémantique est une map d'exécutions allant des annotations des locations aux valeurs de flux de données.

Note: Pour tout nœud n, on note $loc(n)$ la localisation de n et $env(n)$ son environnement. Si "n" est un nœud variable alors son composant de pile est noté par $stack(n)$. Un couple $\langle e, E \rangle$ est dit bien formé si $E(x)$ est défini pour tout
variable x qui apparaît libre dans e.

Valeurs et map d'exécution:
Il existe quatre types flux de données $v \in \mathbf{V}$:

1. Toute constante est aussi une valeur.
2. La valeur $\bot$ correspond à l'inaccessibilité d'un nœud.
3. La valeur $\top$ modélise les erreurs d'exécution.
4. Les fonctions sont représenté par des tableaux, tel qu'une table $\boldsymbol{v}$ conserve un couple de valeurs d'entrée et sortie (S) = $\langle v_i, v_o \rangle$ pour chaque pile de sites d'appel.

La sémantique du flux de données calcule des tableaux d'exécution qui lient les nœuds aux valeurs.

On prend l'exemple de la fonction suivante:

```
1  let id x = x_k in
2  let u = (id_q 1_f)_g in
3  (id_a 2_b)_c
```

La figure suivante montre l'expression correspondante dans le langage simple définie précédemment avec chaque sous-expression annotée avec son emplacement unique :

$$((\lambda \mathrm{id}.\,(\lambda \mathrm{u}.\,(\mathrm{id}_a\ 2_b)_c)_d\ (\mathrm{id}_q\ 1_f)_g)_h\ (\lambda x.\,x_k)_o)_p$$

La map d'exécution du programme obtenue est :

$$h \mapsto [h \triangleleft [q \triangleleft 1 \to 1,\ a \triangleleft 2 \to 2] \to 2]$$
$$\mathrm{id}, o \mapsto [q \triangleleft 1 \to 1,\ a \triangleleft 2 \to 2] \quad d \mapsto [d \triangleleft 1 \to 2]$$
$$q \mapsto [q \triangleleft 1 \to 1] \quad a \mapsto [a \triangleleft 2 \to 2]$$
$$\mathrm{u}, f, g, \mathrm{x}^q, k^q \mapsto 1 \quad b, c, d, h, \mathrm{x}^a, k^a, p \mapsto 2$$

L'entrée cumulée b, c, d, h, xa , ka , $p \mapsto\to$ 2 montre que le programme retourne 2.

Sémantique concrète:
Dans cette section l'article définit formellement la sémantique du flux de données d'un programme d'ordre supérieur comme le plus petit point fixe d'un transformateur concret.

Transformateur concret :
Le transformateur concret pour notre sémantique est une étape de fonction qui évalue une expression dans l'environnement actuel, la pile et la map d'exécution, produisant une map d'exécution mise à jour.

$$\mathrm{step} : Exp \to \mathcal{E} \times \mathcal{S} \to \mathcal{M} \to \mathcal{V} \times \mathcal{M}$$

Le transformateur est purement structurel et défini sur une fonction de propagation nommée "prop".
La sémantique concrète actuelle prend le programme d'entrée et met à jour la map initialement vide comme un point fixe jusqu'à ce qu'il n'y ait plus de mises à jour, c'est-à-dire qu'il n'y ait plus de flux de données.
Les transformateurs d'états primitifs et les opérations de composition sont :

$$!n \stackrel{\text{def}}{=} \Lambda M.\, \langle M(n), M \rangle$$
$$n := v \stackrel{\text{def}}{=} \Lambda M.\, \mathbf{let}\, v' = M(n) \sqcup v\, \mathbf{in}\, \mathbf{if}\, safe(v')\, \mathbf{then}\, \langle v', M[n \mapsto v'] \rangle\, \mathbf{else}\, \langle \top, M_\top \rangle$$
$$\mathbf{env}(E) \stackrel{\text{def}}{=} \Lambda M.\, \langle M \circ E, M \rangle$$
$$\mathbf{assert}(P) \stackrel{\text{def}}{=} \Lambda M.\, \mathbf{if}\, P\, \mathbf{then}\, \langle \bot, M \rangle\, \mathbf{else}\, \langle \top, M_\top \rangle$$
$$\mathbf{for}\, v\, \mathbf{do}\, F \stackrel{\text{def}}{=} \Lambda M.\, \dot{\bigsqcup}_{S \in v} F(S)(M)$$
$$\mathbf{return}\, v \stackrel{\text{def}}{=} \Lambda M.\, \langle v, M \rangle$$
$$\mathbf{bind}(F, G) \stackrel{\text{def}}{=} \Lambda M.\, \mathbf{let}\, \langle u, M' \rangle = F(M)\, \mathbf{in}\, \mathbf{if}\, u = \top\, \mathbf{then}\, \langle \top, M_\top \rangle\, \mathbf{else}\, G(u)(M')$$

1. Le transformateur ! n lit la valeur du nœud n dans la map d'exécution actuelle M et renvoie cette valeur avec la map inchangée.

2. Le transformateur n **:=** v met à jour le nœud n dans la map M en faisant la jointure de la valeur actuelle de n avec v. Enfin, il retourne la nouvelle valeur obtenue et la map mise à jour.
3. L'opération bind(F,G) définit la composition des calculs avec l'état F et G, tel que $F \in \mathrm{M} \to \alpha \times \mathrm{M}$ et $G \in \alpha \to \mathrm{M} \to \beta \times \mathrm{M}$ pour certains α et β.

Définition des steps :
La définition de step[|e|](E, S) est donnée en utilisant l'induction sur la structure de e :

$$\begin{array}{l}\mathsf{step}[\![c_\ell]\!](E,S) \stackrel{\text{def}}{=} \\ \quad \textbf{do}\ n = \ell\diamond E\,;\ v' \leftarrow n := c\,;\ \textbf{return}\ v'\end{array}$$

1. Pour une constante c, on affecte au nœud actuel n**,** le lien entre sa valeur actuelle M**(**n**)** et la valeur c.

$$\begin{array}{l}\mathsf{step}[\![x_\ell]\!](E,S) \stackrel{\text{def}}{=} \\ \quad \textbf{do}\ n = \ell\diamond E\,;\ v \leftarrow !n\,;\ n_x = E(x)\,;\ \Gamma \leftarrow \textbf{env}(E) \\ \qquad v' \leftarrow n_x,\ n := \Gamma(x) \ltimes v \\ \quad \textbf{return}\ v'\end{array}$$

2. Le cas des variables implémente la propagation du flux de données entre le nœud variable n_x , la liaison x et le nœud d'expression actuel n où x est utilisé.

   La fonction de prograpagation $\ltimes$ est la suivante :

$$\begin{array}{l}v_1 \ltimes v_2 \stackrel{\text{def}}{=} \\ \quad \textbf{let}\ v' = \Lambda S. \\ \quad\ \textbf{if}\ S \notin v_2\ \textbf{then}\ \langle v_1(S), v_2(S)\rangle\ \textbf{else} \\ \quad\ \textbf{let}\ \langle v_{1i}, v_{1o}\rangle = v_1(S);\ \langle v_{2i}, v_{2o}\rangle = v_2(S) \\ \qquad \langle v'_{2i}, v'_{1i}\rangle = v_{2i} \ltimes v_{1i};\ \langle v'_{1o}, v'_{2o}\rangle = v_{1o} \ltimes v_{2o} \\ \quad\ \textbf{in}\ (\langle v'_{1i}, v'_{1o}\rangle, \langle v'_{2i}, v'_{2o}\rangle) \\ \quad \textbf{in}\ \langle \Lambda S.\ \pi_1(v'(S)), \Lambda S.\ \pi_2(v'(S))\rangle\end{array}$$

$$v \ltimes \bot \stackrel{\text{def}}{=} \langle v, v_\bot\rangle$$

$$v \ltimes \top \stackrel{\text{def}}{=} \langle \top, \top\rangle$$

$$v_1 \ltimes v_2 \stackrel{\text{def}}{=} \langle v_1, v_1 \sqcup v_2\rangle \quad \textbf{(otherwise)}$$

La fonction de propagation est appliquée entre deux valeurs à des nœuds d'exécution successifs. Les constantes sont propagées vers l'avant à l'aide de jointures. En pratique, le v_2 renvoie la constante à l'étape d'exécution suivante. Les tableaux sont les cas les plus intéressants. Pour chaque pile d'appels, la fonction propage d'abord les valeurs des entrées dans le sens inverse. La propagation des valeurs de sortie va dans l'autre sens. Les valeurs d'entrée et de sortie mises à jour sont finalement stockées dans les tables.

Note : La fonction de propagation est monotone et croissante.

$$
\begin{array}{l}
\mathsf{step}[\![(e_i\ e_j)_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\ n = \ell\diamond E\,;\ n_i = i\diamond E\,;\ n_j = j\diamond E\,;\ v \leftarrow !n \\
\qquad v_i \leftarrow \mathsf{step}[\![e_i]\!](E,S)\ \textbf{if}\ v_i \neq \bot \\
\qquad \textbf{assert}(v_i \in \mathcal{T}) \\
\qquad v_j \leftarrow \mathsf{step}[\![e_j]\!](E,S) \\
\qquad v_i',\ [i\cdot S \triangleleft v_j' \rightarrow v'] = v_i \ltimes [i\cdot S \triangleleft v_j \rightarrow v] \\
\qquad v'' \leftarrow n_i,\ n_j,\ n := v_i',\ v_j',\ v' \\
\quad \textbf{return}\ v''
\end{array}
$$

3. L'application de fonction accompagnée de l'environnement actuel **E**, la pile d'appels **S** et la map d'exécution **M**. Ainsi, e_1 est d'abord évalué et on obtient une map mise à jour ainsi que la table v_i qui stocke l'exécution du nœud e_i. Les mêmes opérations sont effectuées sur e_2**.** Ensuite, des informations sont échangées entre l'argument v_i' et la valeur d'entrée stockée dans la table pour la pile d'appels. Des échanges similaires se produisent entre la sortie de la table pour cette pile d'appels et le résultat de l'application de la fonction. Enfin, les valeurs mises à jour sont stockées et on obtient une nouvelle map.

Les algorithmes des steps pour l'abstraction abstraite et body sont les suivants :

$$
\begin{array}{l}
\mathsf{step}[\![(\lambda x.e_i)_\ell]\!](E,S) \stackrel{\text{def}}{=} \\
\quad \textbf{do}\ n = \ell\diamond E\,;\ v \leftarrow n := v_\bot \\
\qquad v' \leftarrow \textbf{for}\ v\ \textbf{do}\ \mathsf{body}(x, e_i, E, v) \\
\qquad v'' \leftarrow n := v' \\
\quad \textbf{return}\ v''
\end{array}
$$

$$
\begin{array}{l}
\mathsf{body}(x, e_i, E, v)(S') \stackrel{\text{def}}{=} \\
\quad \textbf{do}\ n_x = x\diamond E\diamond S'\,;\ E_i = E.x : n_x\,;\ n_i = i\diamond E_i \\
\qquad v_x \leftarrow !n_x \\
\qquad v_i \leftarrow \mathsf{step}[\![e_i]\!](E_i, S') \\
\qquad [S' \triangleleft v_x' \rightarrow v_i'], v' = [S' \triangleleft v_x \rightarrow v_i] \ltimes v|_{S'} \\
\qquad n_x,\ n_i := v_x',\ v_i' \\
\quad \textbf{return}\ v'
\end{array}
$$

La sémantique $S[\![e]\!]$ d'un programme e est définit comme le plus petit point fixe de step sur le treillis complet des maps d'exécution :

$$
\mathrm{S}[\![e]\!] \stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq}_{M_\bot}\, \Lambda M.\ \textbf{let}\ \langle \_, M' \rangle = \mathsf{step}[\![e]\!](\epsilon,\epsilon)(M)\ \textbf{in}\ M'
$$

Sémantique abstraite intermédiaire :
Ensuite, L'article présente deux sémantiques abstraites qui représentent des étapes d'abstraction cruciales lors du calcul de notre système de types de raffinement de flux de données.
Prenons pour exemple un programme d'ordre supérieur qu'on nommera programme1 :

```
let h g = g^a 1 in
let g1 x = 0 in
let g2 x = x in
(h^c g1)/(h^e g2)
```

1. **Sémantique relationnelle :**
   La sémantique relationnelle remplace les valeurs concrètes par des relations modélisant la façon d'en les valeurs concrètes se rapportent à leur environnement.

   En appliquant la sémantique relationnelle sur le programme1 on obtient :
   $x\colon [c \mapsto \langle y\colon [a\,.\ c \mapsto \langle \{(v\colon 1)\},\ \{(y\colon 1, v\colon 0)\}\rangle],\ \{(x\colon F, v\colon 0)\}\rangle$
   $\quad e \mapsto \langle y\colon [a\,.\ e \mapsto \langle \{(v\colon 1)\},\ \{(y\colon 1, v\colon 1)\}\rangle],\ \{(x\colon F, v\colon 1)\}\rangle]$

   considérons le tableau pour h, tout d'abord, nous introduisons une variable de dépendance x pour la table afin que nous puissions lier les sorties de fonction aux entrées. Regardons maintenant l'entrée pour la pile d'appels du singleton c, l'entrée est une table imbriquée que nous décrivons brièvement, la sortie est une relation avec la valeur dénotait par v. L'entrée x est liée à cette dernière sortie est une fonction, notée ici avec une majuscule F.
2. **Sémantique réduite :**
   La sémantique réduite abstrait davantage les piles concrètes avec une tâche abstraite finie fournie par l'utilisateur, elle réduit ainsi les tableaux.

   Supposons que nous voulions faire une analyse insensible au contexte similaire aux types liquid sur le programme1 on obtient :

   $x\colon \langle y\colon\ \langle \{(v\colon 1)\},\ \{(y\colon 1, v\colon 0), (y\colon 1, v\colon 1)\}\rangle,\ \{(x\colon F, v\colon 0), (x\colon F, v\colon 1)\ \}\rangle$

   La sortie de h est une relation et elle encode l'information selon laquelle la valeur de sortie peut être soit 0 ou 1. L'entrée de h est un tableau imbriqué réduit qui a 1 en entrée et à la fois 0 et 1 en sortie.

Raffinement du flux de données paramétriques type sémantique :

La sémantique des types de raffinement paramétrique abstrait les relations concrètes avec les types de base et le domaine de raffinement fourni par l'utilisateur.

Implémentation et évaluation :

DRIFT a été testé sur 250 programmes en utilisant des benchmarks de références qui impliquent à la fois des programmes de premier ordre et d'ordre supérieur. Le but est de mesurer à quel point des domaines abstraits plus riches et une sensibilité au contexte peuvent aider, ainsi que de le comparer aux outils de pointe actuels.
Le framework a été instancié avec les domaines “octagon” et “polyhedral” sans contexte et avec une sensibilité, il a aussi utilisé testé avec polka. Il a aussi été comparé avec les outils de pointe actuels. Enfin ils ont déduit qu'il peut vérifier plus de programmes.

Bench-mark cat.	DRIFT				R_TYPE				DSOLVE				MOCHI			
	succ	full	avg	med	succ	full	avg	med	succ	full	avg	med	succ	full	avg	med
FO (73)	59(5)	138.33	1.89	0.26	44	5.78(15)	0.08	0.06	49	16.27	0.22	0.12	**62**	332.39(9)	4.55	21.50
HO (62)	**60**	316.18	5.10	1.77	49	9.19(5)	0.15	0.03	41	9.76	0.16	0.24	58	276.37(4)	4.46	15.18
T (80)	79	1497.28(1)	18.72	0.00	73	13.18	0.16	0.04	30	26.26	0.33	0.45	**80**	41.56	0.52	0.11
A (13)	**11**	45.94	3.53	3.70	-	-	-	-	8	7.15	0.55	0.77	9(1)	2.44	0.19	0.10
L (20)	**18(2)**	26.43	1.32	1.23	-	-	-	-	8	6.10	0.30	0.26	17	455.28(3)	22.76	19.81
E (17)	**17**	32.71	1.92	0.39	**17**	1.22	0.07	0.05	14	8.22	0.48	0.17	14	95.95(3)	5.64	43.41

Conclusion:

L’article décrit un nouveau framework nommé DRIFT, qui permet la construction systématique d'algorithmes sophistiqués d'inférence de type de raffinement de flux de données, testé en comparaison avec les systèmes de pointe actuels, ce dernier se révèle être plus précis et résout de nouveaux problèmes de typage.