Théorie, conception et réalisation d'un langage de programmation adapté à XML

Alain Frisch

Soutenance de thèse - 13/12/2004
Directeur: Giuseppe Castagna

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

1. . . . XML.

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

1. . . . XML.
2. . . . langage de programmation adapté . . .

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

1. . . . XML.
2. . . . langage de programmation adapté . . .
3. . . . conception . . .

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

1. . . . XML.
2. . . . langage de programmation adapté . . .
3. . . . conception . . .
4. Théorie . . .

Plan

Théorie, conception, et réalisation
d'un langage de programmation adapté à XML

1. . . . XML.
2. . . . langage de programmation adapté . . .
3. . . . conception . . .
4. Théorie . . .
5. . . . réalisation . . .

Plan

XML et schémas

XML

- Un langage de balises.
- Pour représenter des données de nature arborescente.
- Représentation indépendante de l'application.

Schémas

- Chaque application définit des contraintes sur les documents XML qu'elle peut manipuler :
  - balises (nom, structure d'imbrication),
  - attributs (présence, valeurs autorisées),
  - texte (présence).
- Un tel ensemble de contraintes est un schéma XML.
- Il existe des langages pour écrire des schémas de manière formelle : DTD, XML-Schema, Relax-NG.

XML : schémas

Transformation XML

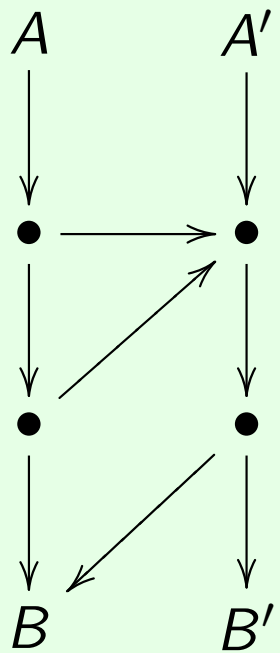
$$\text{Schéma A} \xrightarrow{\text{Programme T}} \text{Schéma B}$$

Garanties

- Le programme T s'engage à fournir un document XML de schéma B si on lui donne un document XML de schéma A.
- Peut-on le croire ?

XML : schémas

Application complexe



Toutes les opérations effectuées sur les documents à l'intérieur de l'application sont-elles légales ?

Opérations illégales

- Consulter un attribut qui peut ne pas exister.
- Supprimer un élément qui doit être présent.

XML : schémas et types

- Typage dans les langages de programmation : interdire des opérations illégales
  (`1 + "Hello"`, `customer.shutdown()`).
- Il est naturel de voir les schémas XML comme des types de données, à l'intérieur des applications.

XML		langages
schémas	$\longleftrightarrow$	types
documents	$\longleftrightarrow$	valeurs

XDuce (Hosoya, Vouillon, Pierce)

Pour le programmeur

- Un langage pour la transformation de documents XML.
- Types $\simeq$ DTD. Valeurs = arbres XML.
- Style fonctionnel. Fonctions récursives et filtrage.

Théorie

- Types = automates d'arbres $\rightsquigarrow$ langages réguliers d'arbres.
- Sous-typage = inclusion des langages.

Regular Expression Types for XML (ICFP 2000).
Regular Expression Pattern Matching for XML (POPL 2001).

Sous-typage

Sous-typage dans XDuce

- Définition sémantique du sous-typage. Pourtant :
  - Système de types : dépend du sous-typage.
  - Sémantique du langage : dirigée par les types (filtrage).

Paradoxe ?

Sous-typage

Sous-typage dans XDuce

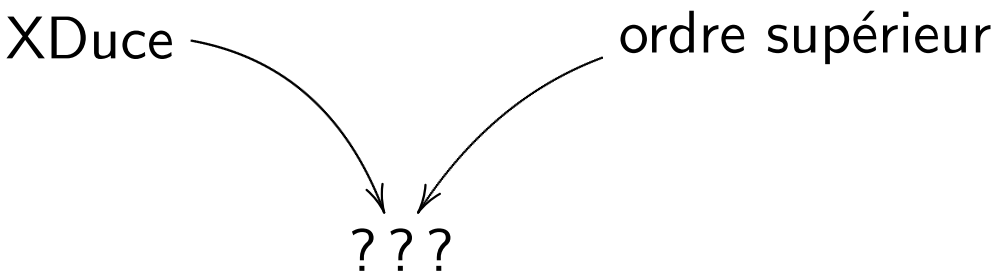
- Définition sémantique du sous-typage. Pourtant :
  - Système de types : dépend du sous-typage.
  - Sémantique du langage : dirigée par les types (filtrage).

Paradoxe ?

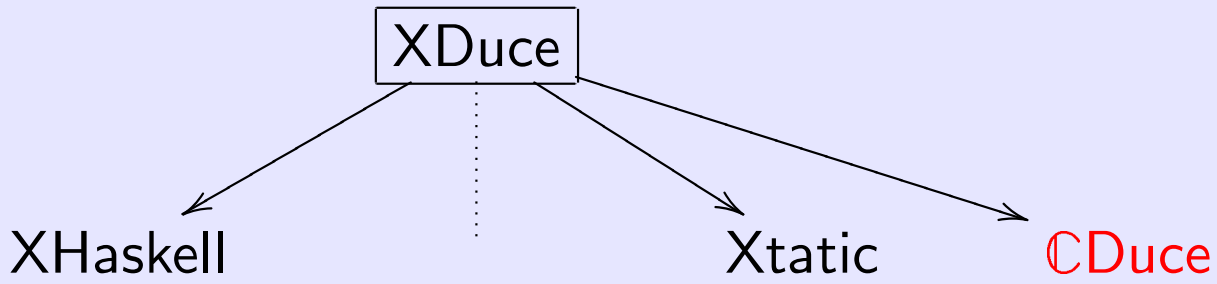
Non !

- La sémantique des types ne dépend pas du langage.

XDuce + ordre supérieur



XDuce : descendance



- XHaskell : types XDuce dans Haskell (sous-typage encodé dans les *type classes*) ; pas de filtrage XML, pas de mélange entre XML et fonctions, pas de sous-typage sémantique.
  (Zhuo Ming Lu, Sulzmann.)
- Xtatic : types XDuce dans C$^{\#}$ (sous-typage par nom pour les objets) ; filtrage limité. Perte du coté structurel des types.
  (Pierce, Schmitt, Gapeyev, Levin.)

Objectif initial de la thèse

Étendre l'approche ensembliste de XDuce à un calcul d'ordre supérieur, qui permet de mélanger librement fonctions et documents XML, dans un cadre structurel.

Coté langage

- XDuce est un langage fonctionnel, mais sans ordre supérieur : ça manque !
- Plus spécialement pour XML : transformations paramétrables de manière modulaire, systèmes de *templates* de première classe, ...

Coté λ-calcul

Prospectif

Objectif initial de la thèse

Étendre l'approche ensembliste de XDuce à un calcul d'ordre supérieur, qui permet de mélanger librement fonctions et documents XML, dans un cadre structurel.

Coté langage

Coté λ-calcul

- Défi intellectuel : sous-typage ensembliste avec types récursifs, combinaisons booléennes, constructeurs de types.
- Sous-typage ensembliste ($t \leq s \iff [\![t]\!] \subseteq [\![s]\!]$) :
  - Facile à expliquer. Exhiber valeur pour illustrer $t \not\leq s$.
  - Définition conceptuellement simple, indépendante d'un algorithme ou d'un système axiomatique complexe.

Prospectif

Objectif initial de la thèse

Étendre l'approche ensembliste de XDuce à un calcul d'ordre supérieur, qui permet de mélanger librement fonctions et documents XML, dans un cadre structurel.

Coté langage

Coté λ-calcul

Prospectif

- Interaction données/comportements sur le web (formulaires,javascript,applets,servlets,...)

Plan

Aperçu de ℂDuce

Types XML

```
type Bib = [ Book* ]
type Book = <book>[ Title Subtitle? Author+ ]
type Title = <title>[ PCDATA ]
type Subtitle = <title>[ PCDATA ]
type Author = <author>[ PCDATA ]
```

Fonctions et filtrage

```
let title(Book -> String)  <book>[ <title>x _* ] -> x
```

Aperçu de ℂDuce

Types XML

```
type Bib = [ Book* ]
type Book = <book>[ Title Subtitle? Author+ ]
type Title = <title>[ PCDATA ]
type Subtitle = <title>[ PCDATA ]
type Author = <author>[ PCDATA ]
```

Fonctions et filtrage

```
let title(Book -> String)  <book>[ <title>x _* ] -> x

let author(Book -> [ Author+ ]) <book>[ (x::Author | _)* ] -> x
```

Aperçu de ℂDuce

Ordre supérieur

```
type FBook = Book -> String
type ABook = <book print=FBook>[ Title Subtitle? Author+ ]
type ABib = [ ABook* ]
```

Remarque: $ABook \leq Book \quad ABib \leq Bib$

Aperçu de ℂDuce

Ordre supérieur

```
type FBook = Book -> String
type ABook = <book print=FBook>[ Title Subtitle? Author+ ]
type ABib = [ ABook* ]
```

Remarque: $ABook \leq Book \quad ABib \leq Bib$

```
let set(<book>c : Book)(f : FBook) : ABook = <book print=f>c
```

Aperçu de ℂDuce

Ordre supérieur

```
type FBook = Book -> String
type ABook = <book print=FBook>[ Title Subtitle? Author+ ]
type ABib = [ ABook* ]
```

Remarque: $ABook \leq Book \quad ABib \leq Bib$

```
let set(<book>c : Book)(f : FBook) : ABook = <book print=f>c

let prepare(b : Bib) : ABib = map b with x -> set x title
```

Aperçu de ℂDuce

Ordre supérieur

```
type FBook = Book -> String
type ABook = <book print=FBook>[ Title Subtitle? Author+ ]
type ABib = [ ABook* ]
```

Remarque: $ABook \leq Book \quad ABib \leq Bib$

```
let set(<book>c : Book)(f : FBook) : ABook = <book print=f>c

let prepare(b : Bib) : ABib = map b with x -> set x title

type Ul = <ul>[ Li+ ]
type Li = <li>[ PCDATA ]

let display (ABib -> Ul; ABook -> Li)
 | <book print=f>_ & x -> <li>(f x)
 | [] -> raise "Empty bibliography"
 | p -> <ul>(map p with z -> display z)
```

Aperçu de ℂDuce

Ordre supérieur

```
let change(p : Book -> Bool)(f : FBook)(b : ABib) : ABib =
 map b with x -> if (p x) then set x f else x
```

Aperçu de ℂDuce

Ordre supérieur

```
let change(p : Book -> Bool)(f : FBook)(b : ABib) : ABib =
 map b with x -> if (p x) then set x f else x

type HasSub = <_>[ _* Subtitle _* ]

let change_if_sub =
 change (fun (Book -> Bool) HasSub -> `true | _ -> `false)

let subtitle_first (Bib -> Bib; ABib -> ABib)
 [ (x::HasSub | y::_)* ] -> x @ y
```

Plan

XDuce revisité

Encodages des valeurs et des types XML

- Séquences.
  $[\mathtt{v}_1 \ldots \mathtt{v}_n] \equiv (\mathtt{v}_1, (\ldots, (\mathtt{v}_n, \text{`}\mathtt{nil}) \ldots))$
  $[\mathtt{t}_1 * \mathtt{t}_2?] \equiv \mu\alpha.(\mathtt{t}_1 \times \alpha) \vee (\mathtt{t}_2 \times \text{`}\mathtt{nil}) \vee \text{`}\mathtt{nil}$
- Éléments XML.
  $<\mathtt{tag}> [\mathtt{v}_1 \ldots \mathtt{v}_n] \equiv (\mathtt{tag}, [\mathtt{v}_1, \ldots, \mathtt{v}_n])$

Des types bien connus

Types : produits, atomiques, réunions, récursifs.

Vers ℂDuce

Encodages des valeurs et des types XML

- Séquences.
  $[\mathtt{v}_1 \ldots \mathtt{v}_n] \equiv (\mathtt{v}_1, (\ldots, (\mathtt{v}_n, \text{`}\mathtt{nil}) \ldots))$
  $[\mathtt{t}_1 * \mathtt{t}_2?] \equiv \mu\alpha.(\mathtt{t}_1 \times \alpha) \vee (\mathtt{t}_2 \times \text{`}\mathtt{nil}) \vee \text{`}\mathtt{nil}$
- Éléments XML.
  $<\mathtt{tag}> [\mathtt{v}_1 \ldots \mathtt{v}_n] \equiv (\mathtt{tag}, [\mathtt{v}_1, \ldots, \mathtt{v}_n])$

- Constructeurs : produit, flèche, enregistrement.
- Connecteurs booléens : réunion, intersection, différence.
- Types récursifs.

Algèbre de types

Types mal formés : à éviter

$t = \mu\alpha.\alpha$

$t = \mu\alpha.\alpha \vee \alpha$

$t = \mu\alpha.\neg\alpha$

Stratification

$\mathbb{T} = (T, \tau)$ avec $\tau : T \longrightarrow F(T)$

$C \in F(T) := b \mid t \times t \mid t \rightarrow t \mid C \vee C \mid C \wedge C \mid \neg C \mid \mathbb{0} \mid \mathbb{1}$

Deux conditions :

- Décomposition finie des types
- Existence des types récursifs

Algèbre de types : stratification

Tautologies booléennes

- Liberté pour quotienter modulo tautologies booléennes.
- $C \vee C \simeq C$
  $C \wedge \neg C \simeq \mathbb{0}$

Partage

- Liberté pour partager ou non types modulo déroulement.
- $\mu\alpha.\ (\alpha \times \alpha) \vee t \stackrel{?}{=} \mu\alpha.\ t \vee (\alpha \times (\mu\beta.\beta \times \alpha \vee t) \vee \alpha \times \alpha)$

Algèbre de types

Résultats

- Cadre catégorique : souplesse d'implémentation ; naturalité des algorithmes.
- Constructions de plusieurs algèbres, avec partage minimal, maximal, intermédiaire, et caractérisations.
- Représentation des combinaisons booléennes par arbres de décision ternaire.

Sous-typage

Définition ensembliste

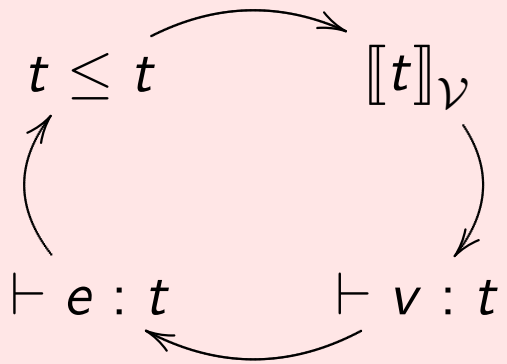
$t \leq s \iff [\![t]\!]_\nu \subseteq [\![s]\!]_\nu \quad \text{où} \quad [\![t]\!]_\nu = \{v \mid \vdash v : t\}$

Sous-typage

Définition ensembliste

$$t \leq s \iff [\![t]\!]_\mathcal{V} \subseteq [\![s]\!]_\mathcal{V} \quad \text{où} \quad [\![t]\!]_\mathcal{V} = \{v \mid \vdash v : t\}$$

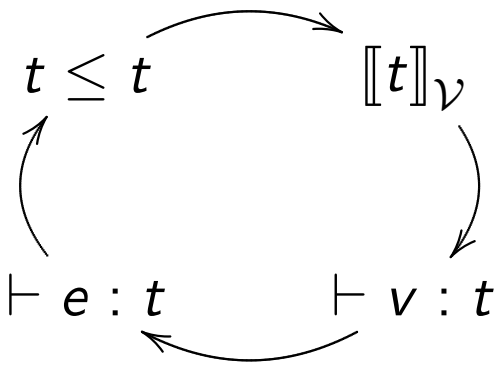
Circularité !



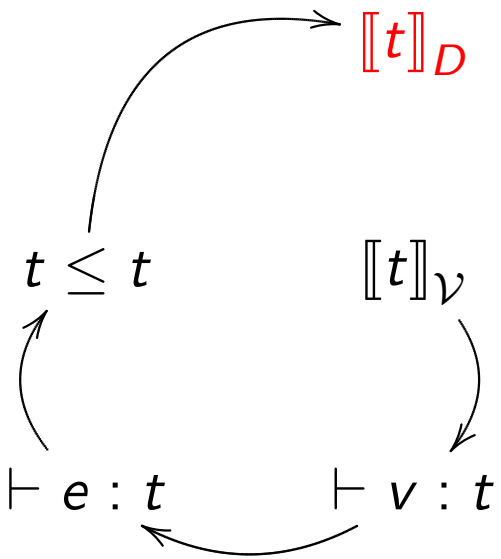
Circularité : comment la dépasser ?

Circularité : comment la dépasser ?

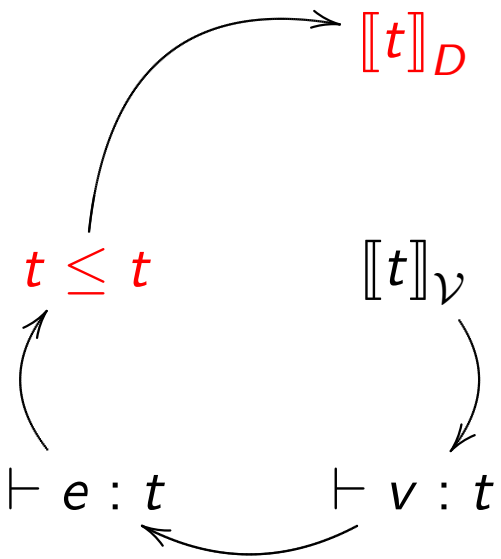
Amorçage



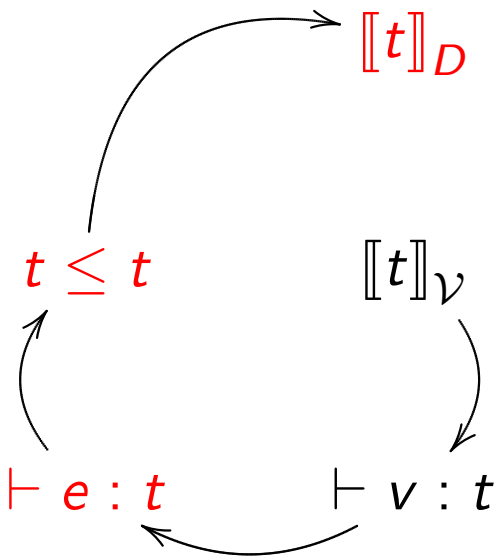
Amorçage



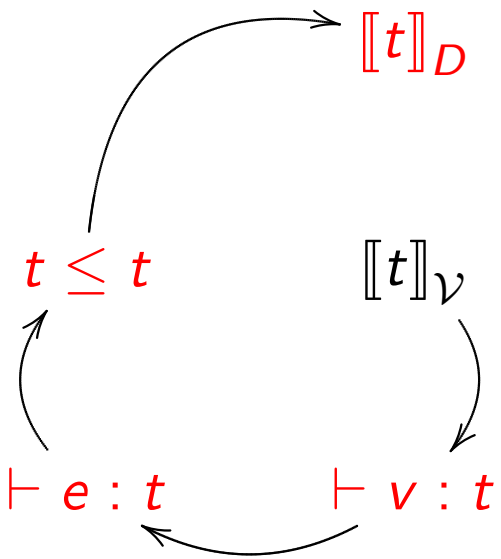
Amorçage



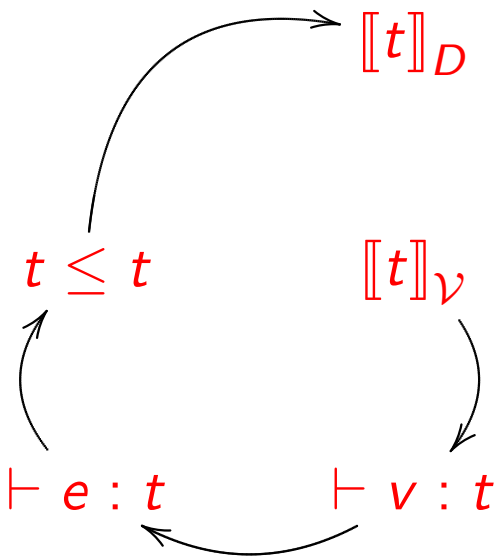
Amorçage



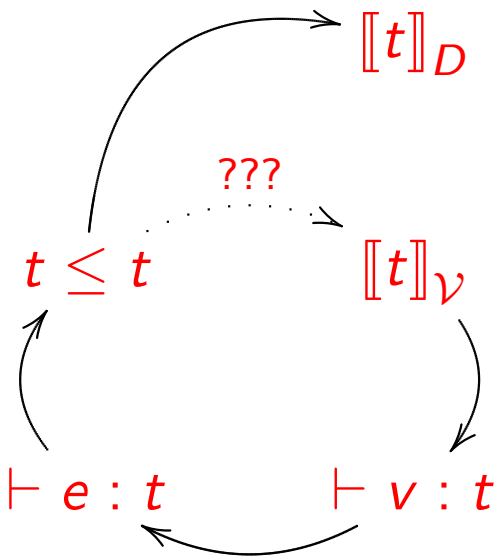
Amorçage



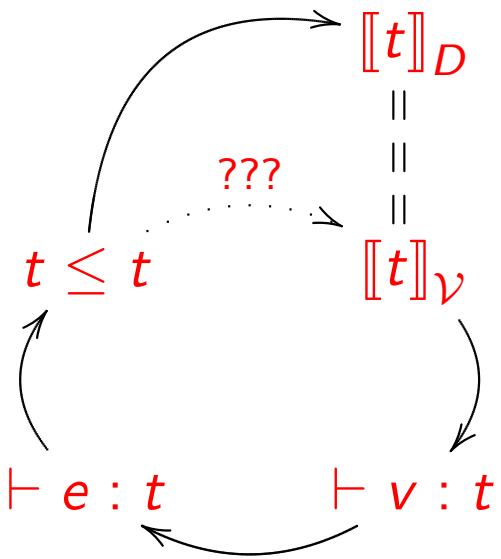
Amorçage



Amorçage



Amorçage



Modèles

Interprétation ensembliste : $[\![\_]\!] : T \to \mathcal{P}(D)$

$$
\begin{array}{lcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\
[\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\neg t]\!] & = & D \setminus [\![t]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & D
\end{array}
$$

Modèles

Interprétation ensembliste : $[\![\_]\!] : T \to \mathcal{P}(D)$

$$
\begin{array}{lcl}
[\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] \\
[\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] \\
[\![\neg t]\!] & = & D \setminus [\![t]\!] \\
[\![\mathbb{0}]\!] & = & \emptyset \\
[\![\mathbb{1}]\!] & = & D
\end{array}
$$

Constructeurs : $\mathbb{E}[\![\_]\!] : T \to \mathcal{P}(\mathbb{E}D)$

$$\mathbb{E}D = \mathcal{C} + D \times D + D^D$$

$$
\begin{array}{lcll}
\mathbb{E}[\![t_1 \times t_2]\!] & = & [\![t_1]\!] \times [\![t_2]\!] & \subseteq \quad D \times D \\
\mathbb{E}[\![t_1 \to t_2]\!] & = & [\![t_2]\!]^{[\![t_1]\!]} & \subseteq \quad D^D \\
\mathbb{E}[\![t_1 \vee t_2]\!] & = & \mathbb{E}[\![t_1]\!] \cup \mathbb{E}[\![t_2]\!] \ldots &
\end{array}
$$

Modèles

Peut-on avoir $D = \mathbb{E}D = \mathcal{C} + D \times D + D^D$?

Modèles

Peut-on avoir $D = \mathbb{E}D = \mathcal{C} + D \times D + D^D$?

Non !

- Pour une simple raison de cardinalité.
- On veut *in fine* que l'ensemble des valeurs soit un modèle.

Modèles

Peut-on avoir $D = \mathbb{E}D = \mathcal{C} + D \times D + D^D$?

Non !

- Pour une simple raison de cardinalité.
- On veut *in fine* que l'ensemble des valeurs soit un modèle.

Faisons « comme si »

Définition : $(D, [\![\_]\!])$ est un modèle si :

$$\forall t, s.\ [\![t]\!] \subseteq [\![s]\!] \iff \mathbb{E}[\![t]\!] \subseteq \mathbb{E}[\![s]\!]$$

(ou : $\forall t.\ [\![t]\!] = \emptyset \iff \mathbb{E}[\![t]\!] = \emptyset$)

Modèles

Résultats

- Caractérisation axiomatique des modèles en terme de simulation (notion coinductive).
- Existence d'un modèle universel $[\![\_]\!]^0$, qui induit la plus grande relation de sous-typage.
- Existence de modèles non-universels (plus difficile !).
  $t = \mu\alpha.(\mathbb{0}\to\mathbb{0})\setminus(\alpha\to\mathbb{0}) \qquad [\![t]\!] = \emptyset$?

Raisonnements dans les modèles

- Propriétés simples : transitivité de $\leq$, variance des constructeurs, tautologies booléennes, égalités ensemblistes :
  $(t_1 \times s_1) \wedge (t_2 \times s_2) \simeq (t_1 \wedge t_2) \times (s_1 \wedge s_2)$
  $(t \to s_1) \wedge (t \to s_2) \simeq t \to (s_1 \wedge s_2)$
  $(t_1 \to s) \wedge (t_2 \to s) \simeq (t_1 \vee t_2) \to s$
- Très utile pour préparer l'étude algorithmique du sous-typage et pour mener l'étude méta-théorique du système de types !

Sous-typage : décomposition

$$\bigwedge_{i\in I} t_i \times s_i \leq \bigvee_{i\in J} t_i \times s_i$$

$$\iff \forall J' \subseteq J.\ \left(\bigwedge_{i\in I} t_i \leq \bigvee_{i\in J'} t_i\right) \text{ ou } \left(\bigwedge_{i\in I} s_i \leq \bigvee_{i\in J\setminus J'} s_i\right)$$

$$\bigwedge_{i\in I} t_i \to s_i \leq \bigvee_{i\in J} t_i \to s_i$$

$$\iff \exists j \in J.\forall I' \subseteq I.\ \left(t_j \leq \bigvee_{i\in I'} t_i\right) \text{ ou } \left(I' \neq I \text{ et } \bigwedge_{i\in I\setminus I'} s_i \leq s_j\right)$$

Coller au langage

$$(t_1 \to s_1) \wedge (t_2 \to s_2) \overset{?}{\leq} (t_1 \vee t_2) \to (s_1 \wedge s_2)$$

$$\texttt{int} \to \texttt{int} \overset{?}{\leq} \mathbb{1} \to \mathbb{1}$$

Résultat : variantes du système

- Avec/sans fonctions surchargées.
- Avec/sans erreur de type à l'application.

Plan

Noyau fonctionnel, système de types

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}$$

$$\frac{}{\Gamma \vdash c : b_c} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \frac{\Gamma \vdash e_1 : t \to s \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\, e_2 : s}$$

$$\frac{\forall i = 1..n.\ (x : t_i), \Gamma \vdash e : s_i}{\Gamma \vdash \lambda^{t_1 \to s_1; \ldots; t_n \to s_n} x.e : \bigwedge_{i=1..n} (t_i \to s_i)}$$

Noyau fonctionnel, système de types

$$\frac{\Gamma \vdash e : t_0 \quad (x : t_0 \wedge t), \Gamma \vdash e_1 : s_1 \quad (x : t_0 \setminus t), \Gamma \vdash e_2 : s_2}{\Gamma \vdash (x = e \in t ? e_1 | e_2) : s_1 \vee s_2}$$

Système de types

Résultats

- Sûreté pour une sémantique à petits pas.
- On obtient un modèle en prenant $[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$. Il est équivalent au modèle d'amorce (pour n'importe quel choix du modèle d'amorce). La boucle est bouclée !
- Exactitude locale du typage de l'application et du filtrage.
- Pas de type « principal », mais algorithme d'inférence.
- Itérateurs génériques, typage par déroulement des types.

Plan

Calcul du sous-typage

Pour un modèle universel, le prédicat unaire $P(t) \equiv [\![t]\!] \neq \emptyset$ est un prédicat inductif (mais pas sur la structure de t qui est cyclique !).

Modularisation

1. Reformuler $P(t)$ comme plus petit point fixe d'un opérateur monotone (contraintes booléennes).
2. Calculer ce plus petit point fixe.

Résultats

1. Divers optimisations ($\Leftarrow$ raisonnements ensemblistes).
2. Solveur *top-down* sans *back-tracking*, avec implémentation légère.

Compilation du filtrage

Un exemple

```
type A = <a>[ A* ]
type B = <b>[ B* ]

fun (A|B -> Int) A -> 0 | B -> 1
≃
fun (A|B -> Int) <a>_ -> 0 | _ -> 1
```

- Tirer partie des informations de types statiques
- Autoriser un style plus déclaratif ($\rightsquigarrow$ robuste)

Résultat

- Cadre pour raisonner sur les stratégies de compilation.
- Critère de correction des optimisations.
- Heuristique de compilation.

Principales contributions

Aspects sémantiques

- Extension de XDuce avec l'ordre supérieur, en préservant l'approche ensembliste.
- Calcul de fonctions surchargées. Variante sans surcharge.
- Formulation originale pour les types et motifs récursifs.
- Filtrage : motifs intersection, capture plus puissante, typage exact.
- Enregistrements extensibles et sous-typage ensembliste.

Principales contributions

Aspects algorithmiques

- Compilation optimisée du filtrage.
- Modularisation de l'algorithme de sous-typage et calcul efficace.
- Implémentation efficace : représentation des valeurs.

Implémentation de ℂDuce

- $\simeq$ 20 000 lignes de code (OCaml).
- Distribution publique depuis juin 2003.
- Implémentation assez efficace (ordre de grandeur des moteurs XSLT et XQuery pour transformations/requêtes équivalentes).
- Prise en compte de tout XML, Unicode, Namespaces. Traduction DTD $\rightsquigarrow$ types, et partiellement pour XML Schema.
- Quelques dizaines d'utilisateurs.
- Utilisations : sites web statiques, dynamiques (y compris en production), enseignement.
- Système d'interface typé avec OCaml :
  - Utiliser des bibliothèques OCaml existantes.
  - Appeler ℂDuce depuis OCaml (projets mixtes).

Thèses autour de ℂDuce

- Analyse de sécurité (M. Burelle)
- Langage de requêtes (C. Miachon)
- Itérateurs, chemins, . . . (K. Nguyen)

Travaux futurs

Sémantique, typage

- Polymorphisme paramétrique.
- Extension de ML avec types et opérations XML.
- Inférence des types de fonctions.
- Logique plus puissante (ex : contraintes d'arité, pointeurs ID/IDREF).

Travaux futurs

Algorithmique, compilation

- Représentation des valeurs dirigée par le type et le contexte d'utilisation.
- Implémentation du partage optimal des types récursifs modulo déroulement + tautologies booléennes.
- Modèle de coût pour la compilation du filtrage.

Merci !

http://www.cduce.org/