ℂDuce

Un langage fonctionnel pour manipuler des documents XML

Alain Frisch

INRIA Rocquencourt (projet Cristal)

Séminaire IRIT - 16/12/2004

Plan

Plan

XML

XML

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

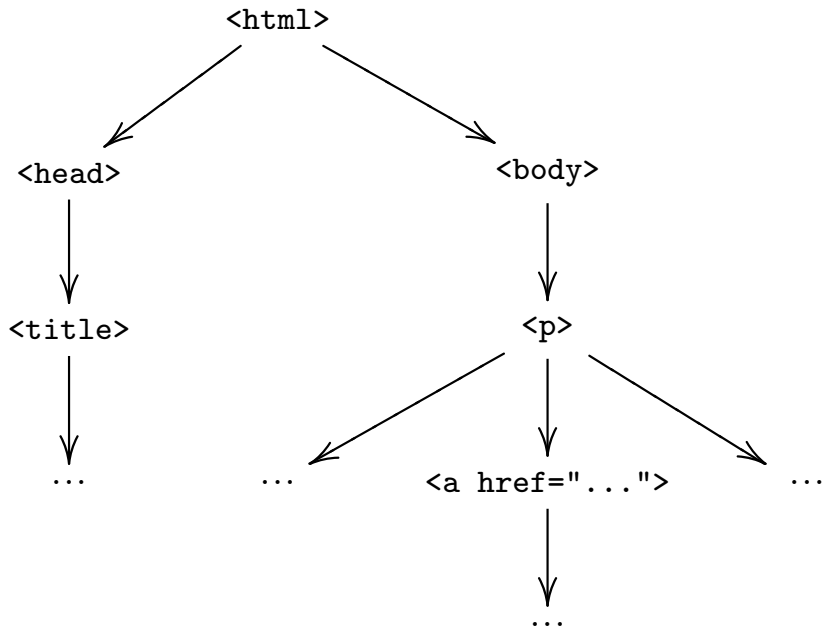
Un exemple

```
<html>
    <head>
        <title>CDuce</title>
    </head>
    <body>
        <p>Please visit the <a href="http://www.cduce.org">CDuce</a> home page.</p>
    </body>
</html>
```

XML

Exemple

```
<html>
 <head>
  <title>...</title>
 </head>
 <body>
  <p>...<a href="...">...</a>...</p>
 </body>
</html>
```



XML

Utilisations

- Documents : web, bureautique, documentation.
- Format d'échange entre applications de gestion.
- Bases de données.
- Fichiers de configuration, de description d'interface.
- Services web.

XML : schémas

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 de données XML.
- Il existe des langages pour écrire des schémas de manière formelle : DTD, XML-Schema, Relax-NG.

XML : schémas

DTD simplifiée de HTML

```
<!ELEMENT html (head, body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT body (p | ul | table)*>
<!ELEMENT p (#PCDATA | a)*>
<!ELEMENT a (#PCDATA)>
<!ATTLIST a
  href        CDATA          #IMPLIED
>
<!ELEMENT ul (li)+>
<!ELEMENT li (p | ul | table)*>
```

XML : schémas

À quoi servent les schémas ?

- Documentation formelle sur les contraintes en entrée ou les garanties en sortie des applications.
- Outils de validations génériques, indépendants des applications.
- Éditeurs XML dirigés par les schémas.

Plan

Intégrer données et applications : ça ne va pas de soi !

BD relationnelles

- Modèle de données assez simple
- Langage SQL : faiblement typé
- Intégration pauvre dans langages de programmation
- (corollaires en terme de sûreté, sécurité, . . .)

BD objets

. . .

Développer avec XML

Niveau 0 : représentation textuelle des documents XML

AWK, sed, Perl

Niveau 1 : vue abstraite

SAX, DOM

Niveau 2 : petits langages spécifiques pour XML, sans types

XSLT, XPath

Niveau 3 : types XML pris avec sérieux

- Data-binding
- Nouveaux langages

Schémas et applications

Transformation XML

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 ?

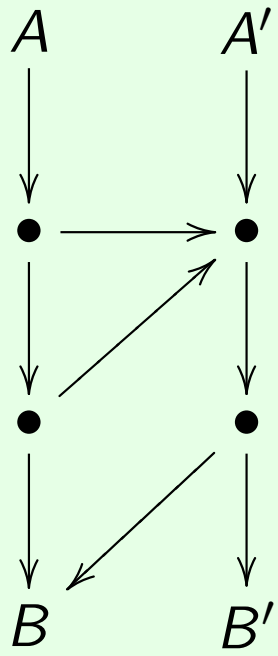
XML : schémas

Sûreté des applications

- Tests
- Spécifications et preuves formelles
- Analyses statiques sur langages existants
- Nouveaux langages

XML : schémas

Application complexe



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

Opérations illégales

- Extraire l'attribut `href` d'un élément `<title>`.
- Supprimer un élément `<li>`.

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

XML : schémas et types

Point de vue des applications

- XML = format de données externes.
  - Les documents XML sont des versions sérialisées des structures de données internes de l'application.
  - Types $\rightsquigarrow$ schémas.
- XML = format d'échange standardisé.
  - Travailler sur des schémas/données déjà définis.
  - Schémas $\rightsquigarrow$ types.

Types et programmation

- Types = représentation concrète des données (C).
- Types = formules logiques (λ-calcul).
- Types = actions possibles (Java).
- Types = structure des données (ML).

- Pour éviter certaines erreurs.
- Pour compiler (efficacement).
- Pour structurer le code.
- Comme documentation. Pour faciliter la maintenance.

Data-binding

Traduire les schémas en des types ?

- Schémas XML : grande souplesse.
- Types de données des langages : plus rigides, moins précis.

<!ELEMENT ul (li)+>

- Un élément <ul> : séquence non vide d'éléments <li>.
- Produire un élément vide : illégal.
- Extraire le premier, ou le dernier sous-élément : légal.
- Oublier la contrainte « non vide ».
- Travailler avec plusieurs schémas qui définissent des contraintes différentes pour le même élément <ul>.

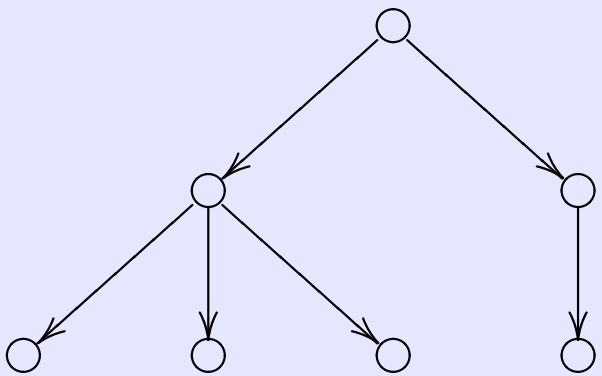
Plan

Automates d'arbres

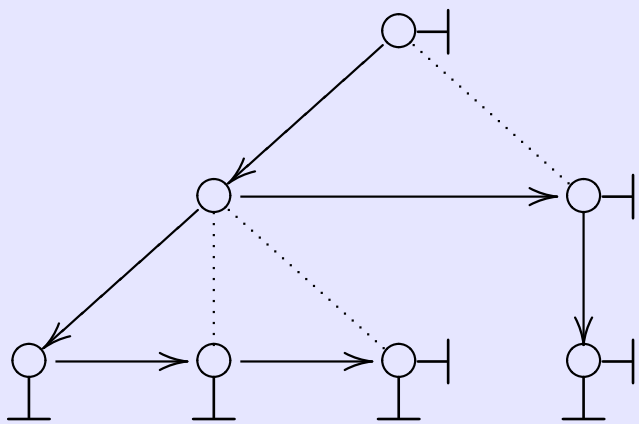
- Généralisation des automates finis.
- Un automate d'arbre définit un ensemble (dit régulier) d'arbres.
- Bonnes propriétés : singletons, clôture par combinaisons booléennes, décidabilité de l'inclusion.
- Dans le cadre de XML, on peut voir les automates comme un formalisme de schémas.
- DTD, XML-Schema $+/-$ automates.

Encodage en arbres binaires

XML



Arbre binaire



Automates d'abres et langages XML

Makoto Murata a proposé d'utiliser les automates dans le cadre de SGML dès 1994.

XDuce (Hosoya, Vouillon, Pierce)

- Types = automates d'arbres. Valeurs = arbres XML.
- Sous-typage = inclusion des langages. (Problème exponentiel, algorithme efficace en pratique, linéaire pour les DTD.)
- Types purement structurels $\neq$ types nominaux.
- Fertile descendance (XQuery, Xtatic, ℂDuce, XHaskell, . . .)

Regular Expression Types for XML (ICFP 2000).
Regular Expression Pattern Matching for XML (POPL 2001).
Hosoya, Vouillon, Pierce.

Types réguliers

$$\vdash v : t$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

Types réguliers

$$\vdash v : t$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day=String>[
    <invited>[ <title>[ PCDATA ]
               <author>[ PCDATA ] ]
    <talk>[ <title>[ PCDATA ]
            <author>[ PCDATA ]
            <author>[ PCDATA ]
            <author>[ PCDATA ] ] ] ]
```

Types réguliers

$$\vdash v : t$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day=String>[
    <invited>[ Title Author ]
    <talk>[ Title Author Author Author  ] ] ]
```

```
type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Types réguliers

$$\vdash v : t$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day=String>[
    <invited>[ Title Author+ ]
    <talk>[ Title Author+ ] ] ]
```

```
type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Types réguliers

$$\vdash v : t$$

```
v ==
  <program>[
    <date day="monday">[
      <invited>[ <title>[ 'Conservation of information' ]
                 <author>[ 'Thomas Knight, Jr.' ] ]
      <talk>[ <title>[ 'Scripting the type-inference process' ]
              <author>[ 'Bastiaan Heeren' ]
              <author>[ 'Jurriaan Hage' ]
              <author>[ 'Doaitse Swierstra' ] ] ] ]
t ==
  Program
```

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Langages fonctionels, typage

Paradigme fonctionnel

Un programme est une expression à évaluer. Le résultat d'une expression est une valeur.

Typage

$\Gamma \vdash e : t$

e : expression à typer

Γ : environnement (types des variables libres)

t : type possible pour e

Exemples

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : t}$$

$$\frac{\Gamma \vdash e_1 : t \to s \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : s}$$

Sous-typage

Subsomption

$t \leq s$ signifie que toute valeur de type t est aussi valeur de type s. Il est alors légitime d'utiliser une expression de type t dans un contexte qui attend une expression de type s.

$$\frac{\Gamma \vdash e : t \quad t \leq s}{\Gamma \vdash e : s}$$

Exemple

Si on dispose d'une fonction de type $s \rightarrow s'$, on peut l'appeler avec un argument de type t si $t \leq s$.

Plan

Transducteurs d'abres

- Ex : k-pebble transducers (Milo, Suciu, Vianu).
- Modèles de calculs formels, avec propriétés de décidabilité pour type-checking inverse.
- Algorithmes de décision couteux, pas implémentés.
- On peut envisager de construire des langages qui utilisent des transducteurs soit comme langage cible, soit comme outil d'analyse.

Motifs réguliers (XDuce, ℂDuce, ...)

- Généralisation du pattern-matching de ML.
- Types XML (expressions régulières) + variables de capture.
- Deux visions : transducteurs rudimentaires, ou automates enrichis par de la capture.
- Récursion en largeur et en profondeur.
- Mini langage de requêtes.
- Motifs ambigus : desambiguation (*first match*, *longest match*), non determinisme, rejet (détection statique), sémantique *all match*.

Motifs et typage

- Le système de types garantit l'exhaustivité et infère le type (exact) des variables de capture.
- Détection des branches/motifs inutilisés.

Filtrage : un petit goût de ML

```
match e with <date day=d>_ -> d

type E = <add>[Int Int] | <sub>[Int Int]
fun eval (E -> Int)
 | <add>[ x y ] -> x + y
 | <sub>[ x y ] -> x - y
```

Les motifs sont des "types avec des variables de capture"

Filtrage : au delà de ML

- Dispatch sur le type :

```
match e with
 | x & Int -> ...
 | x & Char -> ...

let doc =
  match (load_xml "doc.xml") with
    | x & DocType -> x
    | _ -> raise "Invalid input !";;
```

Filtrage : au delà de ML

- Expressions régulières et capture :

```
fun (Invited|Talk -> [Author+])  <_>[ Title x::Author* ] -> x

fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
  [ (i::Invited | t::Talk | _)* ] -> (i,t)

fun parse_email (String -> (String,String))
 | [ local::_* '@' domain::_* ] -> (local,domain)
 | _ -> raise "Invalid email address"
```

Filtres (Hosoya)

Idée

- Un filtrage a la forme $p_1 \longrightarrow e_1 \mid \ldots \mid p_n \longrightarrow e_n$.
- On peut voir $\mid$ comme l'aternation des expressions régulières
- . . . et généraliser le filtrage aux autres opérateurs : séquence, étoile, . . .

Exemple

```
((<author>x -> x @ ", ")* (<author>x -> x @ "et "))? (<author>x -> x)
```

Prolongement

Unifier filtrage, XPath, itérateurs.

Plan

Templates

- Arbres partiels avec des trous nommés.
- On peut brancher un arbre partiel dans un autre.
- Construire progressivement des arbres de manière descendante.
- Au cœur du langage Xact (ex : `<bigwig>`, Jwig).
- Il est nécessaire de donner un type aux trous. En fait, analyse par interprétation abstraite (inférence de flot).

Templates

- On peut naturellement voir un template comme une fonction de première classe.

$$\frac{\Gamma \vdash e_1 : \{x_1 : t_1; \alpha\} \rightarrow t \quad \Gamma \vdash e_2 : \{\beta\} \rightarrow t_1}{\Gamma \vdash \mathtt{plug}_{x_1}\, e_1\, e_2 : \{\alpha; \beta\} \rightarrow t}$$

- Templates et motifs sont des notions duales.
- Langage avec motifs de première classe ?

Plan

ℂDuce

Langage :

- orienté XML ;
- centré sur les types ;
- basé sur le filtrage par expression régulières ;
- avec des caractéristiques généralistes ;
- (assez) efficace.

Extension de XDuce avec ordre supérieur, fonctions surchargées, . . .

ℂDuce

Scénarios d'utilisation

- Petits "adaptateurs" entre applications XML.
- Applications plus grosses.
- Applications web, web services.
- Couche d'E/S XML pour des applications OCaml.

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).

Types

Les types sont partout dans CDuce :

- Validation statique
  - Ex. : est-ce que ça produit toujours du XHTML valide ?
- Semantique dirigée par les types
  - Dispatch dynamique
  - Fonctions surchargées
- Compilation dirigée par les types
  - Optimisations rendues possibles par les types statiques
  - Évite des opérations inutiles/redondantes à l'exécution
  - Autorise un style plus déclaratif

Fonctions

- Surchargées, de première classe, sous-typage, partage de nom, partage de code . . .

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]

let patch_program (p:[Program], f:(Invited -> Invited) & (Talk -> Talk)):[Program] =
  xtransform p with (Invited | Talk) & x -> [ (f x) ]

let first_author ([Program] -> [Program]; Invited -> Invited; Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t a _* ] -> <invited>[ t a ]
| <talk>[ t a _* ] -> <talk>[ t a ]

(* On peut remplacer les deux dernières branches par:
  <(k)>[ t a _* ] -> <(k)>[ t a ]
*)
```

Un autre exemple

```
type Person = FPerson | MPerson
type FPerson = <person gender="F">[ Name Children ]
type MPerson = <person gender="M">[ Name Children ]
type Children = <children>[Person*]
type Name = <name>[ PCDATA ]

type Man = <man name=String>[ Sons Daughters ]
type Woman = <woman name=String>[ Sons Daughters ]
type Sons = <sons>[ Man* ]
type Daughters = <daughters>[ Woman* ]

let split (MPerson -> Man ; FPerson -> Woman)
  <person gender=g>[ <name>n <children>[(mc::MPerson | fc::FPerson)*] ] ->
     let tag = match g with "F" -> `woman | "M" -> `man in
     let s = map mc with x -> split x in
     let d = map fc with x -> split x in
     <(tag) name=n>[ <sons>s  <daughters>d ]
```

Expressivité du filtrage

```
...
  <person gender=g>[...] ->
     let tag = match g with "F" -> `woman | "M" -> `man in
...
```

peut être remplacé par :

```
  <person gender=("F" & (tag := `woman) | (tag := `man))>[...] ->
```

ou par :

```
  <person>[...] & (FPerson & (tag := `woman) | MPerson & (tag := `man)) ->
```

(Plus robuste aux changements de représentation !)

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
```

Erreurs de types précises

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]

let x : Talk = <talk>[ <author>[ 'Alain Frisch' ] <title>[ 'CDuce' ] ]

⇝
let x : Talk = <talk>[ <author>[ 'Alain Frisch' ] <title>[ 'CDuce' ] ]
This expression should have type:
`title
but its inferred type is:
`author
which is not a subtype, as shown by the sample:
`author
```

Erreurs de types précises

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]

fun mk_talk(s : String) : Talk = <talk>[ <title>s ]

⇝
fun mk_talk(s : String) : Talk = <talk>[ <title>s ]
This expression should have type:
[ Author+ ]
but its inferred type is:
[  ]
which is not a subtype, as shown by the sample:
[  ]
```

Erreurs de types précises

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]
type Invited = <invited>[ Title Author+ ]
type Day = <date>[ Invited? Talk+ ]

fun (Day -> [Talk+]) <date>[ Invited? x::_*] -> x
fun (Day -> [Talk+]) <date>[ _? x::_*] -> x

⇝
fun (Day -> [Talk+]) <date>[ _? x::_*] -> x
This expression should have type:
[ Talk+ ]
but its inferred type is:
[ Talk* ]
which is not a subtype, as shown by the sample:
[  ]
```

Erreurs de types précises

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]

fun (p :[Program]):[Program] = xtransform p with Invited -> []
fun (p :[Program]):[Program] = xtransform p with <invited>c -> [<talk>c]

fun (p :[Program]):[Program] = xtransform p with Talk -> []

⇝
fun (p :[Program]):[Program] = xtransform p with Talk -> []
This expression should have type:
[ Program ]
but its inferred type is:
[ <program>[ <date day = String>[ Invited? ]* ] ]
which is not a subtype, as shown by the sample:
[ <program>[ <date day = "">[  ] ] ]
```

Plan

Principales contributions

Aspects sémantiques

- Extension de XDuce avec l'ordre supérieur, en préservant l'approche ensembliste.
- Calcul de fonctions surchargées.
- 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.

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)

Plan

Interface avec OCaml

Appeler OCaml depuis CDuce

- Réutiliser bibliothèques (y compris bindings avec C).
- Structures de données et algorithmes complexes.
- Passer fonctions CDuce comme arguments.
- Instancier fonctions polymorphes avec types CDuce.

Utiliser une unité CDuce comme un module OCaml

- ℂDuce comme couche E/S pour applis OCaml.
- Présentation XHTML de résultats, CGI, sérialisation des données internes, gestion des fichiers de config.

Interface avec OCaml

Objectifs

On veut :

- Préserver sûreté du typage.
- Éviter d'écrire fonctions de coercions.

Approche retenue

Une fonction de traduction dans un seul sens : OCaml $\Longrightarrow$ CDuce.
Pour un type OCaml t, déduire un type CDuce $\hat{t}$ et deux fonctions de coercion $t \leftrightarrow \hat{t}$.

Un mapping CDuce $\Longrightarrow$ OCaml ne peut pas être compatible avec le sous-typage (sauf un mapping constant) !

Traduction des types

t	$\hat{t}$
`int`	`min_int` $--$ `max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\hat{t_1}, \hat{t_2})$
$t_1 \rightarrow t_2$	$\hat{t_1} \rightarrow \hat{t_2}$
A_1 of $t_1 \mid \ldots \mid A_n$ of t_n	$(`A_1, \hat{t_1}) \mid \ldots \mid (`A_n, \hat{t_n})$
$[`A_1$ of $t_1 \mid \ldots \mid `A_n$ of $t_n]$	$(`A_1, \hat{t_1}) \mid \ldots \mid (`A_n, \hat{t_n})$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{\|l_1 = \hat{t_1}; \ldots; l_n = \hat{t_n}\|\}$
t `list`	$[\hat{t}*]$
t `ref`	$\{\|get = [] \rightarrow \hat{t}; set = \hat{t} \rightarrow []\|\}$
t (*abstrait*)	$!t$
`CDuce.Value.t`	`Any`
`Big_int.big_int`	`Int`

Appeler OCaml depuis CDuce

- Soit v une valeur OCaml de type t.
- Pour l'utiliser depuis CDuce : appliquer la coercion $t \longrightarrow \hat{t}$.

```
let home = Sys.getenv "HOME"    (* Latin1 *)
let exists = Sys.file_exists   (* Latin1 -> Bool *)
let listmap = List.map { Int Int } (* (Int -> Int) -> [Int*] -> [Int*] *)
let lst = listmap (fun (x : Int) : Int = x * 2) [ 10 20 30 ]  (* [Int*] *)
```

Caractéristiques OCaml non gérées

- Objets (faisable)
- Tableaux (facile)
- Types récursifs non réguliers (impossible)
- Types abstraits paramétrés (faisable)

Appeler OCaml depuis CDuce

- L'instanciation des variables de type OCaml par des types ℂDuce est-elle sûre ?
- Oui !
  Si $f : \forall\alpha.\tau[\alpha]$, alors $f : \tau[X]$ pour n'importe quel ensemble X de valeurs.

Appeler CDuce depuis OCaml

- CDuce produit des unités de compilation OCaml (l'interface `.cmi` est donnée par le programmeur).
- `ocamlc -c a.mli` ⇝ `a.cmi`
- `cduce --compile a.cd` ⇝ `a.cdo`
- `ocamlc -c -impl a.cdo -pp cdo2ml` ⇝ `a.cmo`
- `ocamlc -o prog ... a.cmo ...` ⇝ `prog`

a.cd

```
let aux ((Int,Int) -> Int)
 | (x, 0 | 1) -> x
 | (x, n) -> aux (x * n, n - 1)

let fact (x : Int) : Int = aux (1, x)
```

a.mli

```
val fact: Big_int.big_int -> Big_int.big_int
```

Autres caractéristiques

- Généralistes : enregistrements, tuples, entiers, exceptions, références, ...
- Chaînes + expressions régulières (types, motifs)
- Connecteurs booléens (types, motifs)
- Autres itérateurs
- Fragment de XPath, `select..from..where...`

Merci !

http://www.cduce.org/