ℂDuce: un aperçu

Alain Frisch

INRIA Rocquencourt

15 octobre 2004
GT Cristal

XML

XML dans la Vraie Vie ™

- Documents, bureautique, web.
- Format d'échange entre applications.
- Web-services.
- Stockage, bases de données.
- Fichiers de configuration, XUL, ...

Comment accompagner XML ?

Point du vue des langages de programmation...

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

. . .

Spécificités de XML

- Notion de typage bien présente: DTD, XML Schema.
- Couplage « lâche » entre documents et types.
- Les documents sont plus importants que les types !

- Poids du passé, choix techniques douteux.
- Le contenu d'un document est une notion mal définie :-(

« *The essence of XML:*

- *The problem it solves is not hard.*
- *It doesn't solve it very well.* »

P. Wadler (*The Essence of XML* - POPL 2003)

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: XQuery, XDuce, CDuce, Xtatic, Xen.

Data-binding

Principe

Utiliser les types d'un langage hôte pour représenter les types XML.

Avantage

On reste dans le cadre d'un langage bien maîtrisé.

Problèmes

- On perd la souplesse des types XML (expressions régulières).
- On se repose sur un système de types trop grossier.
- Problème de robustesse: évolution des schémas XML.
- *Impedance mismatch* entre modèle XML et modèle OO.

CDuce

Langage:

- orienté XML ;
- centré sur les types ;
- avec des caractéristiques généralistes ;
- (vaguement) efficace.

CDuce

- Outil d'expérimentation (valider la théorie, essayer algorithmes, optimisations, . . .).
- Néanmoins à peu près utilisable (et utilisé).

Scénarios d'utilisation

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

Orienté XML, centré sur les données

- Litteraux XML dans la syntaxe.
- Fragments XML: citoyens de première classe, pas enfouis dans des objets.

```
<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

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

XML avec des types

$$\vdash v : t$$

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' ] ] ] ]
```

XML avec des types

$$\vdash v : t$$

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 ] ] ] ]
```

XML avec des types

$$\vdash v : t$$

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 ]
```

XML avec des types

$$\vdash v : t$$

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 ]
```

XML avec des types

$$\vdash v : t$$

```
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 ]
```

Filtrage

- Opération de base du langage.
- Le système de type 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"
```

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 !)

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 = "">[  ] ] ]
```

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...`

Algèbres de termes cycliques

- Les types et les motifs sont des objets cycliques (graphes/automates) sans lieur.
- On veut pouvoir les manipuler comme des termes (constructeurs/destructeurs) d'une algèbre, et laisser les questions de partage et de représentation à l'implémentation.
- On veut faire passer de la théorie des automates pour de la théorie des types !

- Petit cadre catégorique pour faire ça proprement.

Algèbres de termes cycliques

- Soit $F : Set \longrightarrow Set$ un foncteur « signature ». Une F-coalgèbre est un couple (T, τ) avec T ensemble, $\tau : T \longrightarrow FT$.
- Coalgèbre régulière: tout élément de T est dans une sous-coalèbre finie.
  - $\Rightarrow$ terminaison des algorithmes.
- Coalgèbre récursive: toute extension finie se retracte (par un morphisme) dans T.
  - $\Rightarrow$ existence de types récursifs introduits par des systèmes d'équations.
- Remarque: pas de propriété universelle (pas d'unicité des solutions $\Rightarrow$ liberté laissé à l'implémentation pour avoir plus ou moins de partage).
- Constructions: partage optimal, partage minimal, partage modulo renommage des lieurs.

Types

Foncteur: $F = \mathcal{B} \circ S$

$a \in SX$	$:=$	b	type de base
	$\mid$	$x \times x$	constructeur produit
	$\mid$	$x \to x$	constructeur flèche
	$\mid$	$\ldots$	(XML, enregistrements)
$t \in \mathcal{B}A$	$:=$	a	atome
	$\mid$	$t \vee t \mid t \wedge t \mid \neg t$	connecteurs booléens
	$\mid$	$\mathbb{0} \mid \mathbb{1}$	types vide, plein

- Garantit bonne formation de la récursion (exclure $\mu\alpha.\ \alpha \vee \alpha$).
- En fait, égalité modulo tautologies booléennes pour $\mathcal{B}$; par exemple, formes normales disjonctives:
  $\mathcal{B}A = \mathcal{P}_f(\mathcal{P}_f(A) \times \mathcal{P}_f(A))$

Types

- Les types décrivent des ensembles de valeurs.
- Sous-typage sémantique (ensembliste):

$$t \leq s \iff [\![t]\!] \subseteq [\![s]\!]$$

où

$$[\![t]\!] = \{v \mid \vdash v : t\}$$

Problème: définition circulaire entre sous-typage et typage!

- Le sous-typage dépend du typage des valeurs.
- Le typage des valeurs dépend du typage des expressions (λ-abstractions !).
- Le typage des expressions dépend du sous-typage (règle de subsomption !).

$\rightsquigarrow$ développement d'une méthode d'amorçage

Amorçage

- On considère des interprétations booléennes de l'algèbre de types $[\![\_]\!] : T \to \mathcal{P}(D)$ ($[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$, ...).
- Soit $[\![\_]\!]$ une telle interprétation. On pose:

$$\begin{aligned}
&\mathbb{E}D = \mathcal{C} + D \times D + D^D \\
&\mathbb{E}[\![_]\!] : T \to \mathcal{P}(\mathbb{E}D) \\
&\mathbb{E}[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \\
&\mathbb{E}[\![t_1 \to t_2]\!] = \{f : D \to D \mid f([\![t_1]\!]) \subseteq [\![t_2]\!]\}
\end{aligned}$$

- « Moralement » on veut $D = \mathbb{E}D$. C'est impossible !
- On dit que $[\![\_]\!]$ est un modèle si:

$$\forall t.\ [\![t]\!] = \emptyset \iff \mathbb{E}[\![t]\!] = \emptyset$$

ou encore:

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

Amorçage

- Chaque modèle induit une relation de sous-typage: $t \leq s \iff [\![t]\!] \subseteq [\![s]\!]$.
- Toute la théorie qui suit ne dépend pas du choix du modèle d'amorce.
- On peut construire un modèle universel (qui induit la plus grande relation $\leq$), des modèles non universels (pas facile !).

Raisonnements dans les modèles

- Propriétés simples: transitivité de $\leq$, variance des constructeurs, tautologies booléennes, égalités ensemblistes $(t_1 \times s_1) \wedge (t_2 \times s_2) \simeq (t_1 \wedge t_2) \times (s_1 \wedge s_2)$.
- Très utile pour préparer l'étude algorithmique du sous-typage et pour mener l'étude méta-théorique du système de types !

Règles de sous-typage

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

$$\Longleftrightarrow$$

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

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

$$\Longleftrightarrow$$

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

Typage de l'application fonctionnelle

Exemple avancé de raisonnement ensembliste.

$$\frac{}{\mathbf{app} : ((t \to s) \times t) \to s}\ (\mathbf{app}\to)$$

$$\frac{\mathbf{app} : t_1 \to s_1 \quad \mathbf{app} : t_2 \to s_2}{\mathbf{app} : (t_1 \wedge t_2) \to (s_1 \wedge s_2)}\ (\mathbf{app}\wedge)$$

$$\frac{\mathbf{app} : t_1 \to s_1 \quad \mathbf{app} : t_2 \to s_2}{\mathbf{app} : (t_1 \vee t_2) \to (s_1 \vee s_2)}\ (\mathbf{app}\vee)$$

$$\frac{\mathbf{app} : t' \to s' \quad t \leq t' \quad s' \leq s}{\mathbf{app} : t \to s}\ (\mathbf{app} \leq)$$

Thm: si $t \simeq \bigvee_{i=1..n} t_1^i \times t_2^i$, alors: $(\mathbf{app} : t \to s) \iff \forall i.\ t_1^i \leq t_2^i \to s$

Méthode: passer par une interprétation ensembliste de l'application dans un modèle D' déduit de D ; $\mathbf{app}([\![t]\!]') \subseteq [\![s]\!]'$.

Calcul, système de types

$$\frac{}{\Gamma \vdash c : b_c} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e : t \quad o : t \to s}{\Gamma \vdash o(e) : s}$$

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2} \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}$$

Calcul, système de types

$$t = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j) \not\simeq \mathbb{0}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e : t_0 \leq \lbag p_1 \rbag \vee \lbag p_2 \rbag \\ ((t_0 \wedge \lbag p_1 \rbag)/p_1), \Gamma \vdash e_1 : s_1 \\ ((t_0 \setminus \lbag p_2 \rbag)/p_2), \Gamma \vdash e_2 : s_2\end{array}}{\Gamma \vdash \texttt{match } e \texttt{ with } p_1 \to e_1 \mid p_2 \to e_2 : s_1 \vee s_2}$$

($\lbag p \rbag, (t/p)$ sont définis de manière sémantique)

Résultats

- On obtient un modèle en prenant $[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$. Il est équivalent au modèle d'amorce !
- Sûreté pour une sémantique à petits pas.
- Pas de type « principal », mais l'ensemble des types d'une expression est un filtre ; algorithme d'inférence.
- Implémentation de l'algorithme de typage avec propagation bidirectionnelle des contraintes pour localiser erreurs.
- Exactitude du typage de l'application:

$$\mathbf{app} : t \rightarrow s \iff \forall v \in [\![t]\!]_{\mathcal{V}}.\mathbf{app}(v) \overset{*}{\rightsquigarrow} v' \Rightarrow v' \in [\![s]\!]_{\mathcal{V}}$$

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 de l'algorithme de sous-typage.
- Utiliser des raisonnements sémantiques pour déduire une définition de P comme solution minimale d'un système de contraintes booléennes $\{\Phi(t) \Rightarrow P(t) \mid t \in T\}$ où $\Phi(t)$ est une formule booléenne positive construite sur des $P(t')$.
  - Plusieurs versions possibles.
  - Heuristiques pour traiter cas usuels.
- Utiliser un solveur léger de contraintes booléennes.

Solveur avec backtracking

```
eval(t, N, P) :=
 if t ∈ P then (1, N, P)
 else if t ∈ N then (0, N, P)
 else match eval(Φ(t), N ∪ {t}, P) with
      | (0, N′, P′) → (0, N′, P′)
      | (1, N′, P′) → (1, N, P′ ∪ {t})
eval(φ1 ∧ φ2, N, P) :=
 match eval(φ1, N, P) with
      | (0, N′, P′) → (0, N′, P′)
      | (1, N′, P′) → eval(φ2, N′, P′)
eval(φ1 ∨ φ2, N, P) :=
 match eval(φ1, N, P) with
      | (0, N′, P′) → eval(φ2, N′, P′)
      | (1, N′, P′) → (1, N′, P′)
eval(0, N, P) := (0, N, P)
eval(1, N, P) := (1, N, P)
```

On peut implémenter N et P avec une structure persistente (table de hash) + une pile pour backtracker N !

Solveur sans backtracking

Idées:

- Garder les dépendences entre assertions en cours de vérification et leurs conséquences.
- Retarder autant que possible le déroulement des formules booléennes.
- Utiliser la structure du treillis $\{0, 1\}$ (hauteur 2) pour simplifier !

Compilation du filtrage

Implementation du filtrage

- $\rightsquigarrow$ Sans backtracking
- $\rightsquigarrow$ Utilise les informations de types statiques
- $\rightsquigarrow$ Factorisation des captures
- $\rightsquigarrow$ Autorise un style plus déclaratif ($\rightsquigarrow$ robuste)

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

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

Représentation des valeurs

- Concaténation en temps constant (représentation symbolique), avec applatissement à la demande.
- Représentation compacte des chaînes de caractères.
- Hash-consing des tags XML.

XML

- Gérer les pointeurs dans les documents (aka ID/IDREF), et inter-documents (XLink/XPointer).
- Typer l'opération « père » (le type parle du contexte, pas seulement du sous-arbre).
- Intéractions avec le typage par nom (approche de XML Schema / XQuery / XSLT).

Compilation

- Variantes d'heuristiques pour la compilation du filtrage (moins spécialiser pour avoir code plus compact).
- Techniques de fusing/déroulement/spécialisation pour profiter autant que possible de l'implémentation efficace du filtrage.
- Résolution statique de la surcharge à l'appel des fonctions.
- Représentation des valeurs.

Polymorphisme paramétrique

- On a: polymorphisme par sous-typage et par surcharge.
- On veut: opérations génériques.
- 2 pistes:
  - Polymorphisme paramétrique: ajoute de variables dans l'algèbre de types ; quantification universelle.
  - Itérateurs récursifs génériques dont le typage est repoussé à l'application: généralisation des itérateurs prédéfinis, du filtrage, encodage de XPath, tout ça avec un typage très précis. Typage = dérouler l'itérateur et le type de l'argument pour les synchroniser...

Rapprocher ℂDuce et ML

- Objectif: extension conservative (ou pas) de ML avec un type XML, sous-typage sémantique, et filtrage XML (annotations explicites !).
- En fait, étendre ML avec: - une algèbre de types externes pour laquelle on dispose de $\leq$ (implicite) et $\vee$; - des constructions avec typage bottom-up.
- Refinement types, inférence locale, HM(X), ...
- Pour aller plus loin: mélanger plus en profondeur les algèbres de types (variables ML dans le type XML) ...

Travaux en cours autour de ℂDuce

- Sous-typage sémantique des canaux du π-calcul (R. De Nicola, D. Varacca, G. Castagna)
- Securité & analyse de flot d'information (M. Burelle, G. Castagna)
- Langage de requête CQL (C. Miachon, V. Benzaken)
- Intégration XPath / filtrage (K. Nguyen, G. Castagna, A. Frisch)
- Error mining dans ℂDuce (D. Colazzo, G. Castagna, A. Frisch)
- BD XML native (V. Benzaken, I. Manolescu)
- Polymorphisme paramétrique (H. Hosoya, G. Castagna, A. Frisch)

Merci !

http://www.cduce.org/