Langages de programmation (INF4083)
Langages fonctionnels: Kernel/CORE

Wojciech Fraczak

15 mars 2011

Introduction

- *Kernel* — une notation pour la description de systèmes informatiques — mon projet de recherche !

Introduction

- *Kernel* — une notation pour la description de systèmes informatiques — mon projet de recherche !
- *Kernel* est un langage de programmation qui peut être compilé en code machine, et un langage de spécification pour la modélisation et la vérification.

Introduction

- *Kernel* — une notation pour la description de systèmes informatiques — mon projet de recherche !
- *Kernel* est un langage de programmation qui peut être compilé en code machine, et un langage de spécification pour la modélisation et la vérification.
- Les traits uniques de *Kernel* sont :
  - une sémantique en termes de *relations* (pas fonctions totales !) ;
  - *composition* à la place de l'*application* ;
  - modèle (machine virtuelle) en termes de systèmes FIFO ;
  - pas de librairies prédéfinies et pas de "built-in types".

Motivation

1. Mon expérience à Solidum/IDT Canada : PDL/PTL (Packet Description/Transformation Language)
   - pour la programmation d'un automate à pile, un CPU/NPU

Motivation

1. Mon expérience à Solidum/IDT Canada : PDL/PTL (Packet Description/Transformation Language)
   - pour la programmation d'un automate à pile, un CPU/NPU
2. Edgewater : RTEdge — langage temps réel pour faciliter la vérification des contraintes temporelles

Motivation

1. Mon expérience à Solidum/IDT Canada : PDL/PTL (Packet Description/Transformation Language)
   - pour la programmation d'un automate à pile, un CPU/NPU
2. Edgewater : RTEdge — langage temps réel pour faciliter la vérification des contraintes temporelles

- Dans les deux cas des nouveaux langages étaient nécessaires même si des milliers de langages de programmation existent déjà...

Motivation

1. Mon expérience à Solidum/IDT Canada : PDL/PTL (Packet Description/Transformation Language)
   - pour la programmation d'un automate à pile, un CPU/NPU
2. Edgewater : RTEdge — langage temps réel pour faciliter la vérification des contraintes temporelles

- Dans les deux cas des nouveaux langages étaient nécessaires même si des milliers de langages de programmation existent déjà...
- *Kernel* se donne comme objectif d'être une base (noyau) qui puisse être facilement étendue pour satisfaire les exigences de PTL, RTEdge, et autres langages dédiés.

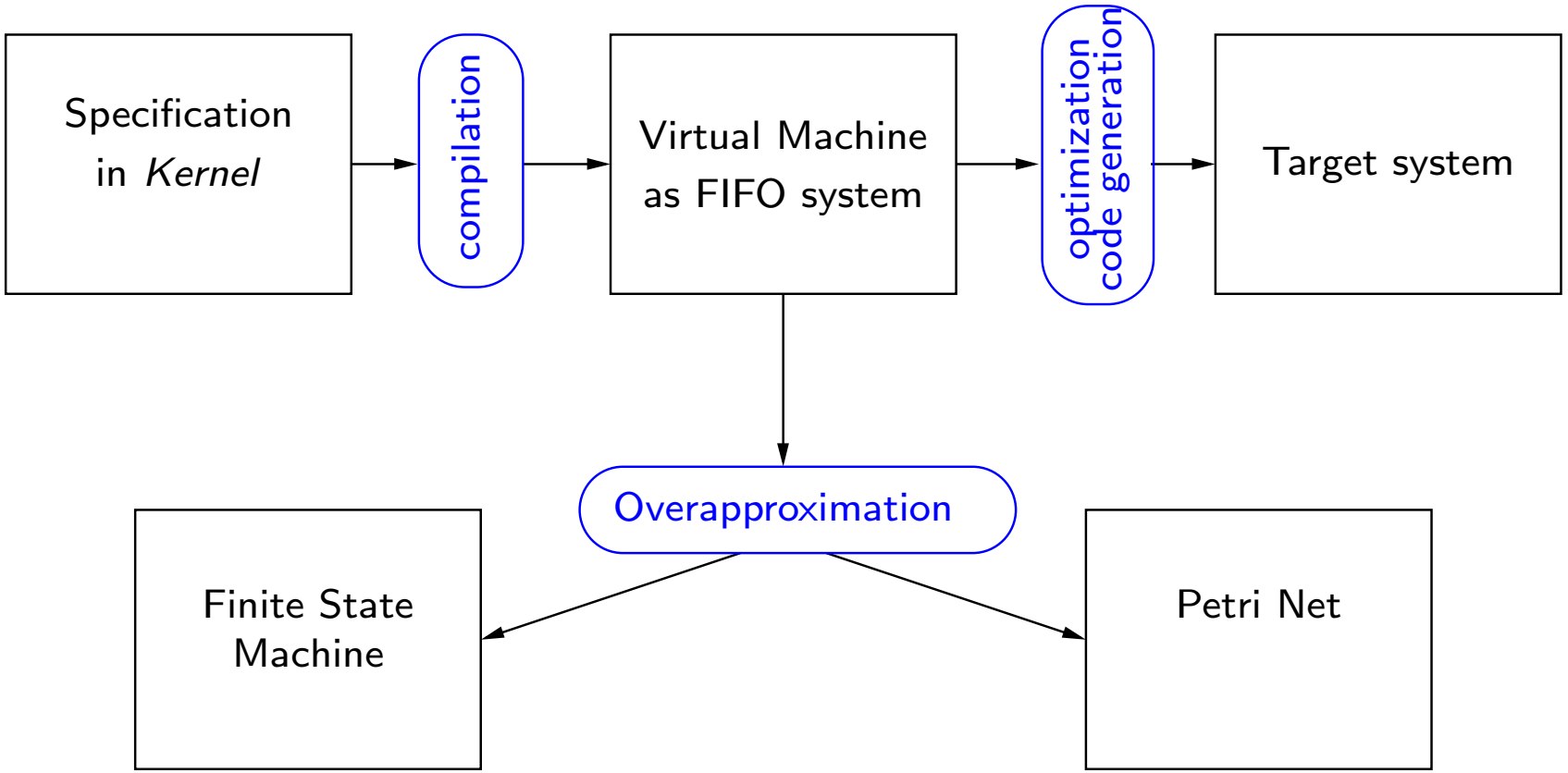
À quoi ça sert ...

- *Kernel* n'est pas un langage de programmation d'usage générale pour les professionnels de l'informatique...
  ...même si l'on pouvait l'utiliser pour développer des cites WEB ou pour des GUIs, la notation *Kernel* n'était pas conçue pour ça.

À quoi ça sert ...

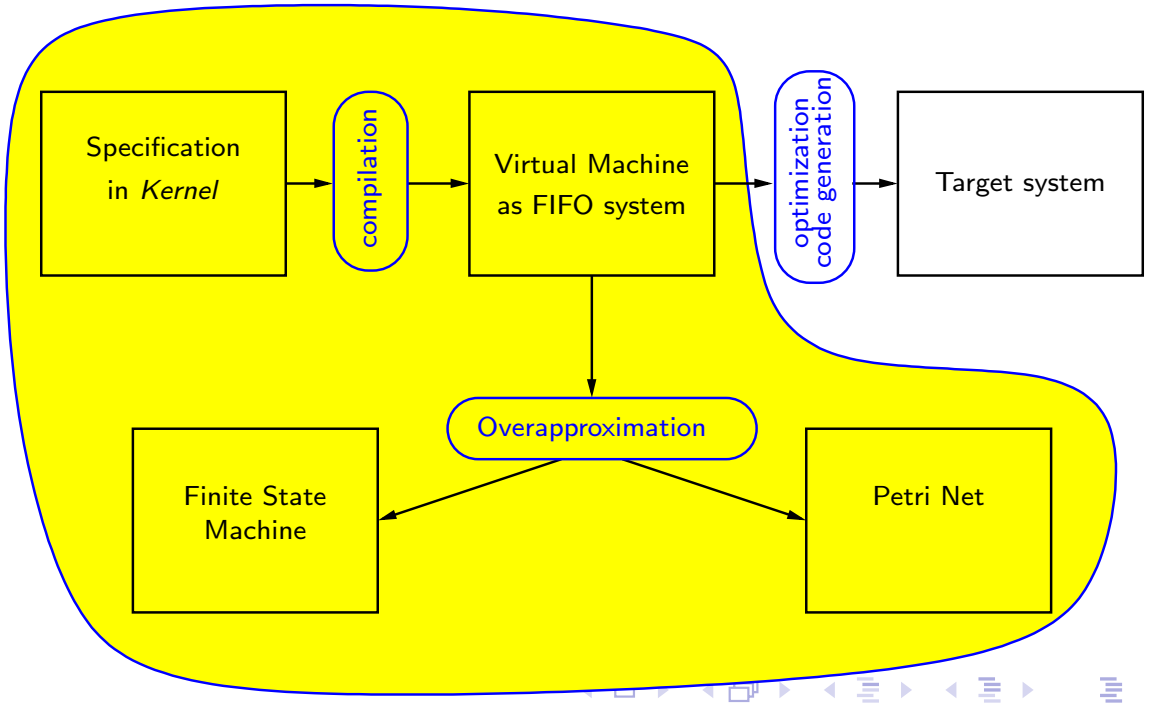
- *Kernel* n'est pas un langage de programmation d'usage générale pour les professionnels de l'informatique...
  ...même si l'on pouvait l'utiliser pour développer des cites WEB ou pour des GUIs, la notation *Kernel* n'était pas conçue pour ça.
- On peut être intéressé par *Kernel* si on cherche à développer des programmes avec des preuves que :
  - le programme utilise des ressources d'une manière contrôlée,
  - le programme est correct.

Survole de l'approche



Plan du cours

1. Introduction à la notation *Kernel*
2. Compilation vers le système FIFO
3. Vérification par *over-approximations*



Types de données :

En BNF :

```
Type_def     ::=  "<type>" Type_Name "<is>" Type ";"
Type         ::=  Record_Type | Variant_Type | Type_Name
Record_Type  ::=  "{" Label Type "," ... "}"
Variant_Type ::=  "[" Label Type "," ... "]"
Label        ::=  "'" Identifier
Type_Name    ::=  Identifier
```

Par exemple :

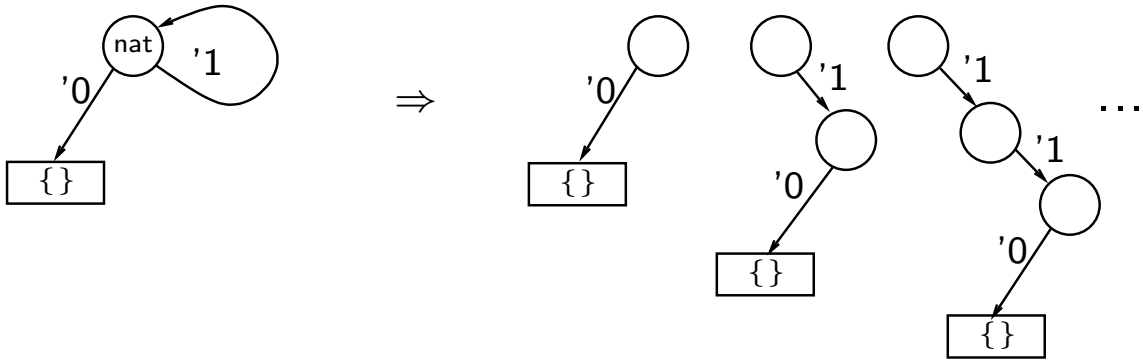
```
<type> UNIT <is> {};
<type> BOOL <is> [ 'true {}, 'false {} ];
```

Par convention, on peut éliminer les accolades {}. Par exemple :

```
<type> BOOL <is> [ 'true, 'false ];
```

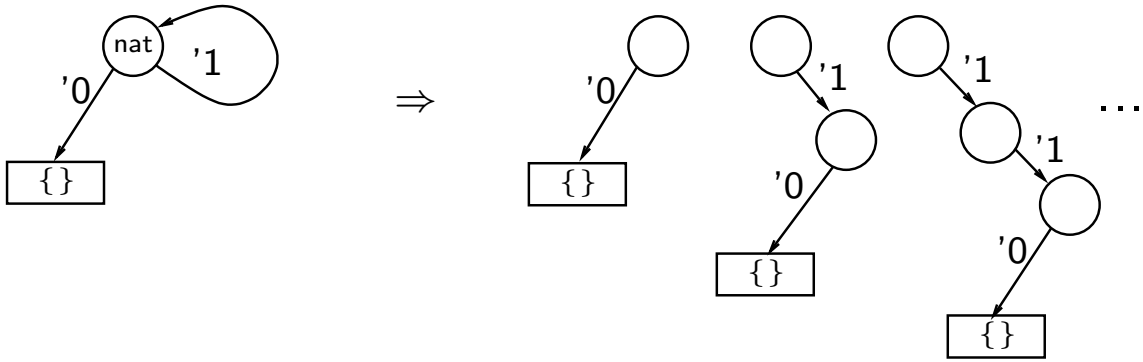
Types de données, valeurs et patterns

```
<type> nat <is> ['0 {}, '1 nat ] ;
```



Types de données, valeurs et patterns

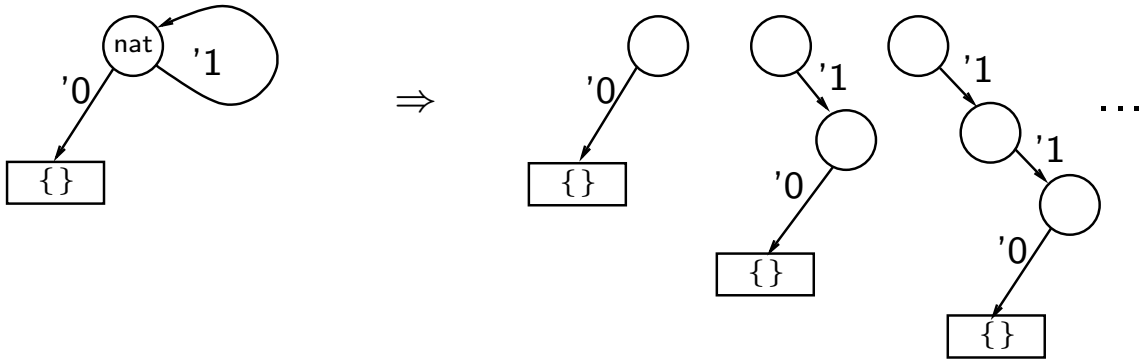
```
<type> nat <is> ['0 {}, '1 nat ] ;
```



Valeurs possibles : `'0`, `'1'0`, `'1'1'0`, etc.

Types de données, valeurs et patterns

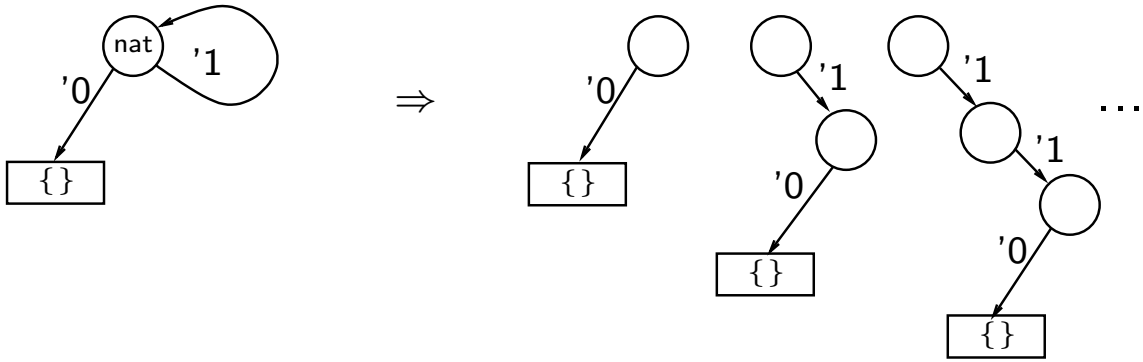
```
<type> nat <is> ['0 {}, '1 nat ] ;
```



Valeurs possibles : `'0`, `'1'0`, `'1'1'0`, etc., sont des arbres !

Types de données, valeurs et patterns

```
<type> nat <is> ['0 {}, '1 nat ] ;
```



Valeurs possibles : `'0`, `'1'0`, `'1'1'0`, etc., sont des arbres !

Dans *Kernel*, un « type » est :

- un ensemble de valeurs (arbres),
- un ensemble de *constructeurs* (attributs), et
- un ensemble de *patterns* (la partie initiale de la valeur, e.g., « `'1 ...` »).

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs :

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`,

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`,

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`,

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.
- Pattern « `2bits` » (ou `{'b0 bit, 'b1 bit}`, etc)

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.
- Pattern « `2bits` » (ou `{'b0 bit, 'b1 bit}`, etc) dénote toutes les valeurs de ce type !

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.
- Pattern « `2bits` » (ou `{'b0 bit, 'b1 bit}`, etc) dénote toutes les valeurs de ce type !
- Pattern « `{'b0 '0, 'b1 bit}` »

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

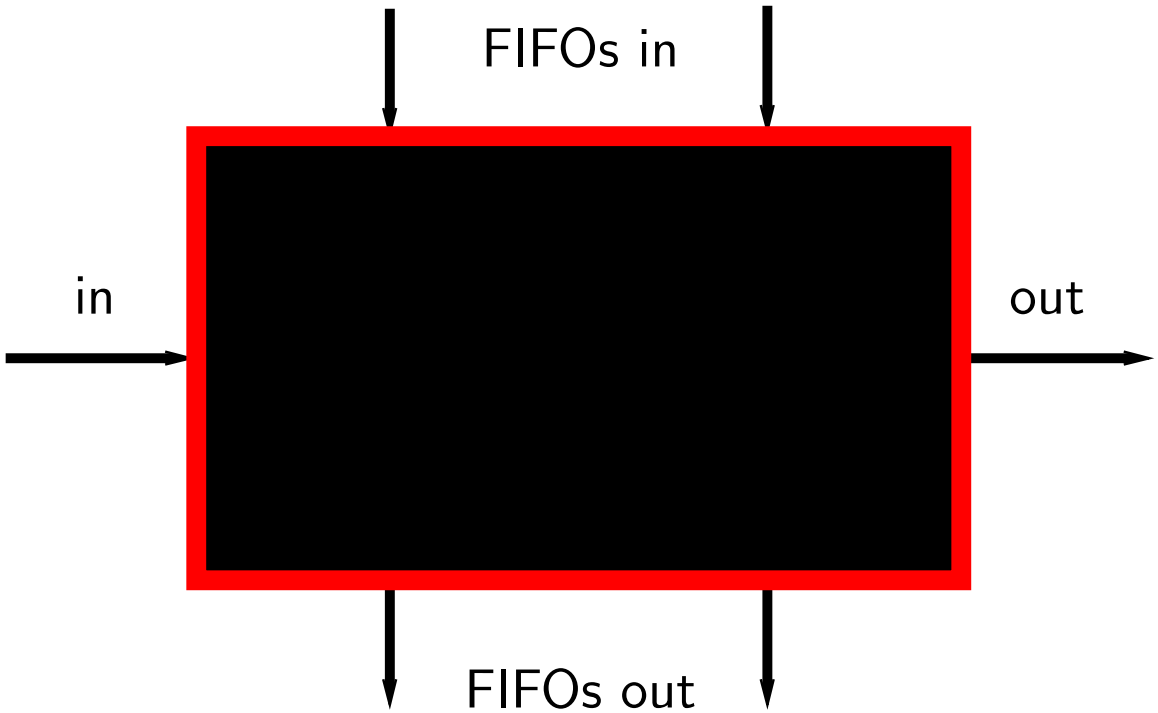
- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.
- Pattern « `2bits` » (ou `{'b0 bit, 'b1 bit}`, etc) dénote toutes les valeurs de ce type !
- Pattern « `{'b0 '0, 'b1 bit}` », dénote `{'b0 '0, 'b1 '0}` et `{'b0 '0, 'b1 '1}`.

Types de données, valeurs et patterns : un exemple

```
<type> bit <is> ['0, '1];
<type> 2bits <is> {'b0 bit, 'b1 bit};
<type> 2bits-bis <is> {'b1 ['0,'1], 'b0 ['1,'0]};
```

- Valeurs : `{'b0 '0, 'b1 '0}`, `{'b0 '0, 'b1 '1}`, `{'b0 '1, 'b1 '0}`, `{'b1 '1, 'b0 '1}`.
- Pattern « `2bits` » (ou `{'b0 bit, 'b1 bit}`, etc) dénote toutes les valeurs de ce type !
- Pattern « `{'b0 '0, 'b1 bit}` », dénote `{'b0 '0, 'b1 '0}` et `{'b0 '0, 'b1 '1}`.
- Si le type de pattern est non ambiguë, on peut utiliser « `...` », e.g. : « `{'b0 '0, 'b1 ...}` », ou même « `{'b0 '0, ...}` ».

Les composantes de *Kernel* sont des *relations*



Chaque composante (non-récursive) peut être transformée en un automate fini (à plusieurs bandes).

Intuition

- Produit (parallélisme) : « `{'one R1, 'two R2}` »

  Si `R1:({} -> T1)` *et* `R2:({} -> T2)`, *alors*
  `{'one R1, 'two R2}:({} -> {'one T1, 'two T2})`.

- Union (concurrence) : « `[R1, R2]` »

  Les relations `R1`, `R2`, *et* `[R1, R2]` *sont tous de même type.*

- Abstraction : « `( $x Pattern -> R )` »

  Si `R:({} -> T1)` *et* `Pattern:T2`, *alors*
  `($x Pattern -> R):(T2 -> T1)`.

- Composition (piping) : « `R1 :: R2` »

  Si `R1 : (T1 -> T3)` *et* `R2 : (T3 -> T2)`, *alors*
  `R1::R2 : (T1 -> T2)`.

Syntaxe

En BNF :

```
Rel_def     ::= "(" Type "->" Type ")" Rel_Name "=" Expr ";"
              | "(" "{}" "->" Type ")" Rel_Name "=" "<input>" ";"
              | "(" Type "->" "{}" ")" Rel_Name "=" "<output>" ";"
Expr        ::= Rel_Name | "$" Var_Name | "@" Field_Name
              | Label Expr  |  Expr "." Label  |  Expr "::" Expr
              | "(" "$" Var_Name Pattern "->" Expr ")"
              | "{" Label Expr "," ... Serialize "}"
              | "[" Expr "," ... Priority "]"
Pattern     ::= "..." | Type | Label Pattern
              | "{" Label Pattern "," ... "}"
Serialize   ::= | "<serialize>" Rel_Name Serialize
Priority    ::= | "<unknown>" | "<longest>"  Rel_Name Priority
              | "<shortest>" Rel_Name Priority
```

Syntaxe par exemple

Syntaxe par exemple

```
<type> bool <is> ['true {}, 'false {}];

({} -> bool) t = 'true {};

(bool -> bool) not =
      [ ($x 'true -> 'false {}),
        ($x 'false -> 'true {}) ];

({'x bool, 'y bool} -> bool ) or =
   [ ($x {'x 'true,...} -> 'true {}),
     ($x {'x 'false,...} -> $x.y) ];

({} -> bool) main = {'x t, 'y t :: not} :: or;
```

Syntaxe par exemple (notation simplifiée)

Syntaxe par exemple (notation simplifiée)

```
<type> bool <is> ['true, 'false];

bool t = 'true;

(bool -> bool) not =
      [ ('true -> 'false),
        ('false -> 'true) ];

({'x bool, 'y bool} -> bool ) or =
   [ ({'x 'true,...} -> 'true),
     ({'x 'false,...} -> $.y) ];

bool main = {'x t, 'y t :: not} :: or;
```

Syntaxe par exemple (notation simplifiée)

```
<type> bool <is> ['true, 'false];  -- ['true {}, 'false {}];

bool t = 'true;                    -- ({} -> bool) t = 'true {};

(bool -> bool) not =
      [ ('true -> 'false),         -- ($x 'true -> 'false {})
        ('false -> 'true) ];       -- ($x 'false -> 'true {})

({'x bool, 'y bool} -> bool ) or =
   [ ({'x 'true,...} -> 'true),    -- ($x {'x 'true,...} -> 'true {})
     ({'x 'false,...} -> $.y) ];   -- ($x {'x 'false,...} -> $x.y)

bool main = {'x t, 'y t :: not} :: or;  -- ({} -> bool) main = ...
```

Construire l'automate

Chaque construction de *Kernel* est une opération sur les *relations*.

- **Pattern**

Construire l'automate

Chaque construction de *Kernel* est une opération sur les *relations*.

- **Pattern**
  - À chaque *pattern* correspond un langage régulier d'arbres
  - Les langages réguliers d'arbres forme une algèbre de Boole : union, intersection, et complément des langages réguliers d'arbres restent réguliers.

Construire l'automate

Chaque construction de *Kernel* est une opération sur les *relations*.

- **Pattern**
  - À chaque *pattern* correspond un langage régulier d'arbres
  - Les langages réguliers d'arbres forme une algèbre de Boole : union, intersection, et complément des langages réguliers d'arbres restent réguliers.
- **Produit**, **Union**, et **Composition**

Construire l'automate

Chaque construction de *Kernel* est une opération sur les *relations*.

- **Pattern**
  - À chaque *pattern* correspond un langage régulier d'arbres
  - Les langages réguliers d'arbres forme une algèbre de Boole : union, intersection, et complément des langages réguliers d'arbres restent réguliers.
- **Produit**, **Union**, et **Composition** sont des variations de compositions des relations.

Construire l'automate

Chaque construction de *Kernel* est une opération sur les *relations*.

- **Pattern**
  - À chaque *pattern* correspond un langage régulier d'arbres
  - Les langages réguliers d'arbres forme une algèbre de Boole : union, intersection, et complément des langages réguliers d'arbres restent réguliers.
- **Produit**, **Union**, et **Composition** sont des variations de compositions des relations.
  Ils correspondent aux opérations sur des automates.
- ...

Un exemple...

- Pattern {'x ..., 'y '0} de type {'x nat, 'y nat} :

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

Un exemple...

- Pattern {'x ..., 'y '0} de type {'x nat, 'y nat} :

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

- Projection $.x :

$$(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)$$

Un exemple...

- Pattern $\{$`'x ..., 'y '0`$\}$ de type $\{$`'x nat, 'y nat`$\}$:

$$(x/x)(1/1)^*(0/0)(y/y)(0/0)$$

- Projection `$.x` :

$$(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)$$

- Abstraction :

$$\underbrace{\Big(\underbrace{\{\texttt{'x ..., 'y '0}\}}_{(x/x)(1/1)^*(0/0)(y/y)(0/0)} \rightarrow \underbrace{\texttt{\$.x}}_{(x/)(1/1)^*(0/0)(y/)(1/)^*(0/)}\Big)}_{(x/)(1/1)^*(0/0)(y/)(0/)}$$

Systèmes FIFO

- Un système FIFO est un réseau d'automates finis connectés par canaux unidirectionnels point-à-point de type FIFO (First-In-First-Out).

Systèmes FIFO

- Un système FIFO est un réseau d'automates finis connectés par canaux unidirectionnels point-à-point de type FIFO (First-In-First-Out).
- Les avantages d'une telle représentation :
  - simple et facile à réaliser

Systèmes FIFO

- Un système FIFO est un réseau d'automates finis connectés par canaux unidirectionnels point-à-point de type FIFO (First-In-First-Out).
- Les avantages d'une telle représentation :
  - simple et facile à réaliser
  - admet une réalisation distribuée

Systèmes FIFO

- Un système FIFO est un réseau d'automates finis connectés par canaux unidirectionnels point-à-point de type FIFO (First-In-First-Out).
- Les avantages d'une telle représentation :
  - simple et facile à réaliser
  - admet une réalisation distribuée
  - facilite les techniques formelles de vérification via "over-approximation"

Construction de systèmes FIFO

- La finitude d'automates n'est pas toujours préservée, par exemple, à cause de la récursion

Construction de systèmes FIFO

- La finitude d'automates n'est pas toujours préservée, par exemple, à cause de la récursion
- On peut réduire l'augmentation importante de la taille en construisant un système FIFO à la place d'un seul automate

Construction de systèmes FIFO

- La finitude d'automates n'est pas toujours préservée, par exemple, à cause de la récursion
- On peut réduire l'augmentation importante de la taille en construisant un système FIFO à la place d'un seul automate
- Le défi est la construction d'un "bon" système FIFO

Construction de systèmes FIFO

- La finitude d'automates n'est pas toujours préservée, par exemple, à cause de la récursion
- On peut réduire l'augmentation importante de la taille en construisant un système FIFO à la place d'un seul automate
- Le défi est la construction d'un "bon" système FIFO

plusieurs techniques peuvent être envisagées pour la construction d'un système FIFO à partir d'un programme *Kernel* ... ça reste un de mes projets de recherche ...

Sur-approximation (over-approximation)

- En théorie tout système FIFO (avec la taille de canaux finie) peut-être vérifié en utilisant SPIN/Promela ...

Sur-approximation (over-approximation)

- En théorie tout système FIFO (avec la taille de canaux finie) peut-être vérifié en utilisant SPIN/Promela ... — mais déjà, la vérification si système FIFO est fini est indécidable

Sur-approximation (over-approximation)

- En théorie tout système FIFO (avec la taille de canaux finie) peut-être vérifié en utilisant SPIN/Promela ... — mais déjà, la vérification si système FIFO est fini est indécidable
- Sur-approximation (Over-approximation) :
  - Une sur-approximation de système S consiste à ajouter des "exécutions" au système pour créer sa "sur-approximation" S' plus simple pour analyser

Sur-approximation (over-approximation)

- En théorie tout système FIFO (avec la taille de canaux finie) peut-être vérifié en utilisant SPIN/Promela ... — mais déjà, la vérification si système FIFO est fini est indécidable
- Sur-approximation (Over-approximation) :
  - Une sur-approximation de système S consiste à ajouter des "exécutions" au système pour créer sa "sur-approximation" S' plus simple pour analyser
  - pour prouver que toute exécution de S a une propriété ϕ, on prouve que toute exécution de S' vérifie ϕ

Sur-approximation (over-approximation)

- En théorie tout système FIFO (avec la taille de canaux finie) peut-être vérifié en utilisant SPIN/Promela ... — mais déjà, la vérification si système FIFO est fini est indécidable
- Sur-approximation (Over-approximation) :
  - Une sur-approximation de système S consiste à ajouter des "exécutions" au système pour créer sa "sur-approximation" S' plus simple pour analyser
  - pour prouver que toute exécution de S a une propriété ϕ, on prouve que toute exécution de S' vérifie ϕ
  - Par exemple, l'over-approximation peut être utilisée pour borner le "WCRT" par son "deadline" (la date d'échéance).

Fin

Merci

Fin

Merci

the end