Fermetures et Modules dans les langages Concurrents avec Contraintes fondés sur la Logique Linéaire

Rémy Haemmerlé

INRIA – Projet CONTRAINTES

Thèse dirigée par François Fages

17 janvier 2008

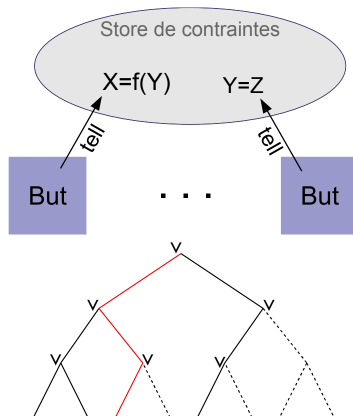
INRIA

Motivations : les Modèles d'Exécution de la Programmation Par Contraintes

La Programmation Logique (Prolog)

PL = programmation à l'aide de clauses logiques [Kowalski 74][Colmerauer]

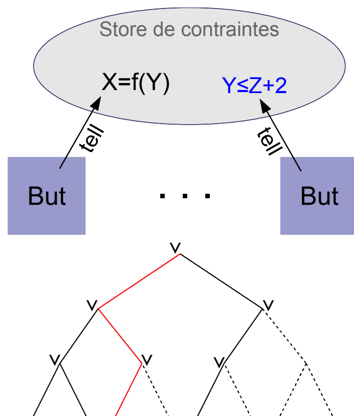
- manipulation de variables sur le domaine de Herbrand (unification de termes)
- programmes réversibles (entrées = sorties)
- recherche non-déterministe
- évolution monotone du store



La Programmation Logiques avec Contraintes

PLC = PL + contraintes [Jaffar-Lassez POPL'87]

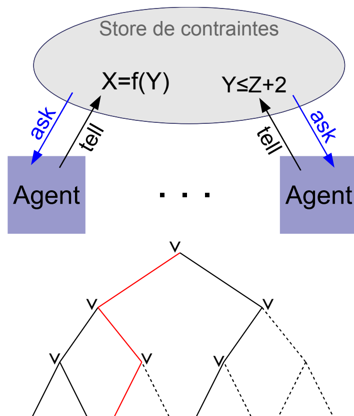
- données définies sur un domaine arbitraire
- manipulation de variables mathématiques (contraintes quelconques)
- résolution de problèmes combinatoires (planification, ordonnancement, sudoku) [Van Hentenryck 89 MIT Press]
- peu de contrôle sur l'exécution



Les langages Concurrents avec Contraintes

CC = PLC + concurrence [Saraswat 89 PhD]

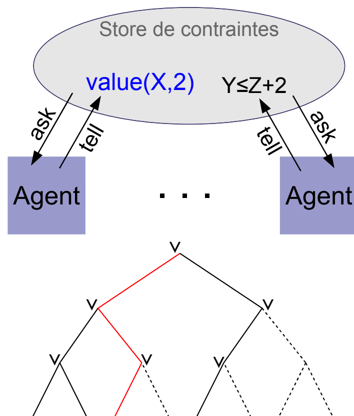
- synchronisation par implication de contraintes
- sémantique en logique linéaire [Lincoln & Saraswat 91]
- les variables jouent le rôle des canaux du π-calcul
- internalisation des mécanismes de :
  - coroutines
  - propagation de contraintes
- évolution monotone du store



Les langages Concurrents avec Contraintes fondés sur la Logique Linéaire

LCC = CC + Logique Linéaire
[Saraswat & Lincoln 91] [Fages, Ruet & Soliman LICS'98]

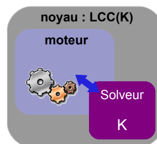
- système de contraintes de logique linéaire
- évolution non-monotone du store
  [Best, de Boer & Palamidessi 97]
- modélisation des changements d'états
- internalisation des algorithmes de résolution de contraintes



Le Projet SiLCC : vers un Système de PPC Extensible

SiLCC =

- un langage noyau :
  - suffisamment expressif
  - le plus petit possible
- solveurs de contraintes
- bibliothèques à usage générique
- implémentation bootstrappée (première fois depuis Prolog)



⇨ système extensible de PPC

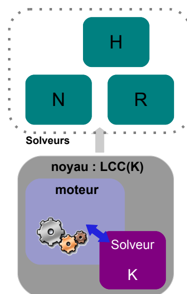
☞ nécessité d'un système de modules puissant pour LCC

INRIA

Le Projet SiLCC : vers un Système de PPC Extensible

SiLCC =

- un langage noyau :
    - suffisamment expressif
    - le plus petit possible
- solveurs de contraintes
- bibliothèques à usage générique
- implémentation bootstrappée (première fois depuis Prolog)



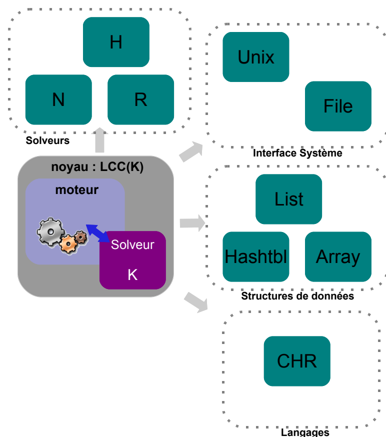
⇨ système extensible de PPC

☞ nécessité d'un système de modules puissant pour LCC

Le Projet SiLCC : vers un Système de PPC Extensible

SiLCC =

- un langage noyau :
  - suffisamment expressif
  - le plus petit possible
- solveurs de contraintes
- bibliothèques à usage générique
- implémentation bootstrappée (première fois depuis Prolog)



⇨ système extensible de PPC

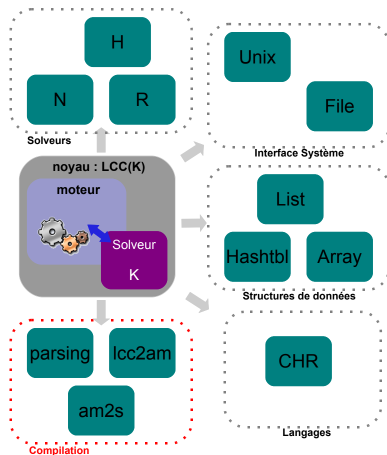
☞ nécessité d'un système de modules puissant pour LCC

INRIA

Le Projet SiLCC : vers un Système de PPC Extensible

SiLCC =

- un langage noyau :
  - suffisamment expressif
  - le plus petit possible
- solveurs de contraintes
- bibliothèques à usage générique
- implémentation bootstrappée (première fois depuis Prolog)



⇨ système extensible de PPC

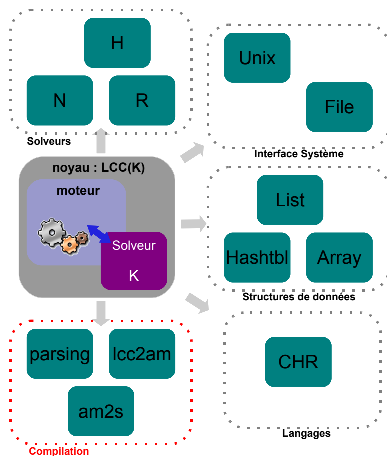
☞ nécessité d'un système de modules puissant pour LCC

INRIA

Le Projet SiLCC : vers un Système de PPC Extensible

SiLCC =

- un langage noyau :
  - suffisamment expressif
  - le plus petit possible
- solveurs de contraintes
- bibliothèques à usage générique
- implémentation bootstrappée (première fois depuis Prolog)



⇨ système extensible de PPC

☞ nécessité d'un système de modules puissant pour LCC

Deux approches pour la conception des systèmes de modules

Première approche :

Modules vus comme **indépendants** du langage hôte
"Modular" module systems [Leroy 00 JFP]

- Avantages :
  - adoption d'un système de modules bien établi
  - réutilisation d'outils théoriques et pratiques déjà développés
- Inconvénients :
  - redondance des concepts de programmation (fermetures/foncteurs, portée/modules et objets)
  - interférence des modules avec le langage hôte : (ex : meta-appel call de Prolog [Haemmerlé & Fages ICLP'06])
  - les modules n'héritent pas de la sémantique formelle du langage hôte

Deux approches pour la conception des systèmes modules

Deuxième approche :

Internalisation du système de modules dans le langage hôte

- Avantages :
  - unicité des concepts de programmation
  - sémantique formelle modulaire héritée du langage hôte
- Limite :
  - le langage hôte doit posséder les mécanismes essentiels à un système de modules :
    - Paramétrisation du code
    - Protection du code

Paramétrisation du Code

Paramétrisation du code = composition de modules

☞ permet la conception de bibliothèques génériques

Exemples de mécanismes de paramétrisation :

- objets (Java, ocaml, Oz/Mozart ...)
- fermetures et foncteurs (SML, ocaml, Oz/Mozart ...)

⇨ internalisation dans LCC de fermetures aussi puissantes que celles du λ-calcul

INRIA

Paramétrisation du Code

Paramétrisation du code = composition de modules

☞ permet la conception de bibliothèques génériques

Exemples de mécanismes de paramétrisation :

- objets (Java, ocaml, Oz/Mozart ...)
- fermetures et foncteurs (SML, ocaml, Oz/Mozart ...)

⇨ internalisation dans LCC de fermetures aussi puissantes que celles du λ-calcul

Paramétrisation du Code

Paramétrisation du code = composition de modules

☞ permet la conception de bibliothèques génériques

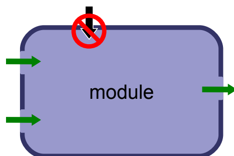
Exemples de mécanismes de paramétrisation :

- objets (Java, ocaml, Oz/Mozart ...)
- fermetures et foncteurs (SML, ocaml, Oz/Mozart ...)

⇨ internalisation dans LCC de fermetures aussi puissantes que celles du λ-calcul

Protection du Code

Protection du code =
impossibilité d'entrer dans un module autrement que par l'entrée



☞ : garantit l'utilisation correcte des modules

habituellement obtenue par **masquage** (signature, opérateur de masquage...)

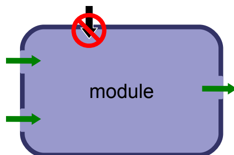
: l'opérateur de masquage des variables dans CC n'est pas suffisant

⇨ : restriction syntaxique pour rendre l'opérateur efficace

INRIA

Protection du Code

Protection du code =
impossibilité d'entrer dans un module autrement que par l'entrée



☞ : garantit l'utilisation correcte des modules

habituellement obtenue par **masquage** (signature, opérateur de masquage...)

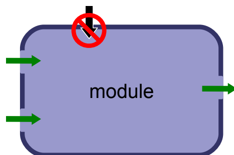
: l'opérateur de masquage des variables dans CC n'est pas suffisant

⇨ : restriction syntaxique pour rendre l'opérateur efficace

INRIA

Protection du Code

Protection du code =
impossibilité d'entrer dans un module autrement que par l'entrée



☞ : garantit l'utilisation correcte des modules

habituellement obtenue par **masquage** (signature, opérateur de masquage...)

: l'opérateur de masquage des variables dans CC n'est pas suffisant

⇨ : restriction syntaxique pour rendre l'opérateur efficace

Plan

1. Motivations
2. Les langages LCC avec asks persistants
3. Les Fermetures : un mécanisme de Paramétrisation du code
4. Les Modules Syntaxiques : un mécanisme de Protection du code
5. Conclusions et Perspectives

INRIA

Les Langages Concurrents avec Contraintes fondés sur la Logique Linéaire avec Asks Persistants

INRIA

La Logique Linéaire, la logique des agents CC classiques

La sémantique logique consiste à :

- interpréter les **agents** CC comme des **formules logiques**
- identifier des **transitions** CC avec des **déductions logiques**

l'affaiblissement et la contraction de LC peuvent oublier ou dupliquer des agents

⇨ La Logique Linéaire [Girard 87] apparaît comme *la* logique des agents *CC* classiques [Saraswat & Lincoln 91]

La logique linéaire est une logique de **ressources** :

- **L'implication** ($\multimap$) : "ressources consommables"

  $!(€ \multimap ☕)$ $\qquad\qquad$ $€,\ € \multimap ☕ \vdash ☕$

- **La conjonction** ($\otimes$) : "ressources énumérables"

  $!((€ \otimes €) \multimap ☕)$ $\qquad\qquad$ $€,\ € \otimes € \multimap ☕ \nvdash ☕$
  $€,\ €,\ €,\ € \otimes € \multimap ☕ \vdash ☕ \otimes €$

- **La modalité** ($!$) : "ressources illimitées"

  $!(\$ \multimap !☕)$ $\qquad\qquad$ $\$,\ \$ \multimap !☕ \vdash ☕$
  $\$,\ \$ \multimap !☕ \vdash ☕ \otimes \cdots \otimes ☕$

La Logique Linéaire, la logique des agents CC classiques

La sémantique logique consiste à :

- interpréter les **agents** CC comme des **formules logiques**
- identifier des **transitions** CC avec des **déductions logiques**

l'affaiblissement et la contraction de LC peuvent oublier ou dupliquer des agents

⇨ La Logique Linéaire [Girard 87] apparaît comme *la* logique des agents *CC* classiques [Saraswat & Lincoln 91]

La logique linéaire est une logique de **ressources** :

- **L'implication** ($\multimap$) : “ressources consommables”

  $!(\euro \multimap \coffeecup)$ $\qquad$ $\euro, \euro \multimap \coffeecup \vdash \coffeecup$

- **La conjonction** ($\otimes$) : “ressources énumérables”

  $!((\euro \otimes \euro) \multimap \coffeecup)$

  $\euro, \euro \otimes \euro \multimap \coffeecup \nvdash \coffeecup$

  $\euro, \euro, \euro, \euro \otimes \euro \multimap \coffeecup \vdash \coffeecup \otimes \euro$

- **La modalité** ($!$) : “ressources illimitées”

  $!(\$ \multimap !\coffeecup)$

  $\$, \$ \multimap !\coffeecup \vdash \coffeecup$

  $\$, \$ \multimap !\coffeecup \vdash \coffeecup \otimes \cdots \otimes \coffeecup$

La Logique Linéaire, la logique des agents CC classiques

La sémantique logique consiste à :

- interpréter les **agents** CC comme des **formules logiques**
- identifier des **transitions** CC avec des **déductions logiques**

l'affaiblissement et la contraction de LC peuvent oublier ou dupliquer des agents

⇨ La Logique Linéaire [Girard 87] apparaît comme *la* logique des agents *CC* classiques [Saraswat & Lincoln 91]

La logique linéaire est une logique de **ressources** :

- **L'implication** ($\multimap$) : "ressources consommables"

  $!(€ \multimap ☕)$ $\qquad$ $€,\ € \multimap ☕ \vdash ☕$

- **La conjonction** ($\otimes$) : "ressources énumérables"

  $!((€ \otimes €) \multimap ☕)$

  $€,\ € \otimes € \multimap ☕ \nvdash ☕$

  $€,\ €,\ €,\ € \otimes € \multimap ☕ \vdash ☕ \otimes €$

- **La modalité** ($!$) : "ressources illimitées"

  $!(\$ \multimap\ !☕)$

  $\$,\ \$ \multimap\ !☕ \vdash ☕$

  $\$,\ \$ \multimap\ !☕ \vdash ☕ \otimes \cdots \otimes ☕$

Un système de contraintes fondé sur la Logique Linéaire

Définition (Système de contraintes)

paire $(\mathcal{C}, \vdash_{\mathcal{C}})$ telle que :

- $\mathcal{C}$ est un ensemble de formules (les contraintes)
- $\vdash_{\mathcal{C}}$ est un système de déduction (le comportement formel d'un solveur)

En pratique $\mathcal{C}$ sera divisé en deux sous-ensembles

contraintes classiques $(\mathcal{D})$	***tokens* linéaires** $(\mathcal{P})$
contraintes de logique classique encodées grâce au (!)	contraintes de logique linéaire sans axiomes
☞ programmation par contraintes	☞ assignation impérative

INRIA

Syntaxe des Langages LCC

☞ internalisation des déclarations [Saraswat et al. POPL'91] comme des asks persistants

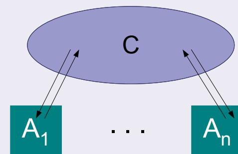
Syntaxe

$$A ::= c \mid A \| A \mid \exists x.A \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$$

Définition (configuration)

triplet de la forme $\langle X; c; A_1, \ldots, A_n \rangle$ où :

- X est l'ensemble des variables masquées
- c est une contrainte représentant le *store*,
- $A_1, \ldots, A_n$ est un multi-ensemble d'agents



Définition (Congruence structurelle)

$\langle X; c; A \| B, \Gamma \rangle \equiv \langle X; c; A, B, \Gamma \rangle$ $\quad$ $\langle X; c; \exists x.A, \Gamma \rangle \equiv \langle X; c; \exists y.A[x \backslash y], \Gamma \rangle$ si y frais

Sémantique Opérationnelle (1/2)

Tell

$$\frac{}{\langle X;\ c;\ d, \Gamma\rangle \longrightarrow \langle X;\ c \otimes d;\ \Gamma\rangle}$$

Masquage

$$\frac{y \notin \mathcal{V}(c, \Gamma)}{\langle X;\ c;\ \exists y.A, \Gamma\rangle \longrightarrow \langle X \uplus \{y\};\ c;\ A, \Gamma\rangle}$$

Ask

$$\frac{c \vdash_{\mathcal{C}} d[\vec{y}\backslash\vec{t}] \otimes e}{\langle X;\ c;\ \forall\vec{y}(d \rightarrow A), \Gamma\rangle \longrightarrow \langle X;\ e;\ A[\vec{y}\backslash\vec{t}], \Gamma\rangle}$$

Ask Persistant

$$\frac{c \vdash_{\mathcal{C}} d[\vec{y}\backslash\vec{t}] \otimes e}{\langle X;\ c;\ \forall\vec{y}(d \Rightarrow A), \Gamma\rangle \longrightarrow \langle X;\ e;\ A[\vec{y}\backslash\vec{t}], \forall\vec{y}(d \Rightarrow A), \Gamma\rangle}$$

INRIA

Sémantique Opérationnelle (2/2)

Définition (Observables)

- $d \in \mathcal{C}$ est une contrainte accessible pour un agent A si :
  - $\langle \emptyset; \mathbf{1}; A \rangle \xrightarrow{*} \langle X; c; \Gamma \rangle$
  - $\exists X.c \vdash_{\mathcal{C}} d \otimes \top$
- $d \in \mathcal{C}$ est un succès pour un un agent A si :
  - $\langle \emptyset; \mathbf{1}; A \rangle \xrightarrow{*} \langle X; c; \Gamma \rangle \not\longrightarrow$
  - $\exists X.c \vdash_{\mathcal{C}} d$
  - Γ est un multi-ensemble de asks persistants

Définition (Sémantique Opérationnelle)

- $\mathcal{O}^{ca}(A)$ est l'ensemble des contraintes accessibles pour un agent A.
- $\mathcal{O}^{succ}(A)$ est l'ensemble des succès pour un agent A.

INRIA

Sémantique Logique

☞ Adaptation des résultats de [Fages, Ruet & Soliman 01] aux asks persistants

Une traduction simple en logique linéaire

$$c^\dagger = c \qquad (\exists x.A)^\dagger = \exists x.A^\dagger \qquad (A \parallel B)^\dagger = A^\dagger \otimes B^\dagger$$
$$(\forall \bar{x}(c \rightarrow A))^\dagger = \forall \bar{x}(c \multimap A^\dagger) \qquad (\forall \bar{x}(c \Rightarrow A))^\dagger = !\forall \bar{x}(c^\dagger \multimap A^\dagger)$$

Théorème (Correction)

Si $(X; c; \Gamma) \xrightarrow{*} (Y; d; \Delta)$ *alors* $\exists X.(c^\dagger \otimes \Gamma^\dagger) \vdash_{\mathcal{C}} \exists Y.(d^\dagger \otimes \Delta^\dagger)$

Théorème (Complétude des contraintes accessibles)

$$\mathcal{O}^{ca}(A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{\mathcal{C}} c \otimes \top\}$$

☞ autre théorème de complétude pour les succès.

INRIA

Les Fermetures : un mécanisme de Paramétrisation du code

Les Fermetures, des constructions de Première Classe

Fermeture

(portion de code + environnement partiel) manipulés comme une donnée.

Abstraction

$\forall X(apply(C,X) \Rightarrow (X \leq Max))$ représente l'abstraction $\lambda X.(X \leq Max)$

Application

$apply(C,Y)$ représente l'application CY

Itérateur sur les structures de données

$\forall C(forall(C,[]) \Rightarrow 1) \parallel$
$\forall C,T,H(forall(C,[H|T]) \Rightarrow apply(C,H) \parallel forall(C,T)) \parallel$

$\exists C.(\forall X(apply(C,X) \Rightarrow (Min \leq X \otimes X \leq Max)) \parallel forall(C,L))$

INRIA

Les Fermetures, des constructions de Première Classe

Fermeture

(portion de code + environnement partiel) manipulés comme une donnée.

Abstraction

$\forall X(apply(C,X) \Rightarrow (X \leq Max))$ représente l'abstraction $\lambda X.(X \leq Max)$

Application

$apply(C,Y)$ représente l'application CY

Itérateur sur les structures de données

$\forall C(forall(C,[]) \Rightarrow 1) \parallel$
$\forall C,T,H(forall(C,[H|T]) \Rightarrow apply(C,H) \parallel forall(C,T)) \parallel$

$\exists C.(\forall X(apply(C,X) \Rightarrow (Min \leq X \otimes X \leq Max)) \parallel forall(C,L))$

Les Fermetures, des constructions de Première Classe

Fermeture

(portion de code + environnement partiel) manipulés comme une donnée.

Abstraction

$\forall X(apply(C,X) \Rightarrow (X \leq Max))$ représente l'abstraction $\lambda X.(X \leq Max)$

Application

$apply(C,Y)$ représente l'application CY

Itérateur sur les structures de données

$\forall C(forall(C,[]) \Rightarrow \mathbf{1}) \parallel$
$\forall C,T,H(forall(C,[H|T]) \Rightarrow apply(C,H) \parallel forall(C,T)) \parallel$

$\exists C.(\forall X(apply(C,X) \Rightarrow (Min \leq X \otimes X \leq Max)) \parallel forall(C,L))$

INRIA

Le λ-calcul faible avec partage

Syntaxe

$$M ::= x|\lambda x.M|MM$$

Sémantique (λ-calcul faible)

β-**réduction :** $(\lambda x.M)N \rightarrow_\beta M[x\backslash N]$

appel gauche : $\dfrac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}$ **appel droit :** $\dfrac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'}$

λ-calcul faible avec partage

Les arguments ne sont évalués qu'une fois :

exemple : Si $M \rightarrow_{\beta_p} N$ alors $(\lambda x.xx)M \rightarrow_{\beta_p} MM \rightarrow_{\beta_p} NN$

INRIA

Le λ-calcul faible avec partage

Syntaxe

$$M ::= x \mid \lambda x.M \mid MM$$

Sémantique (λ-calcul faible)

β-**réduction :** $(\lambda x.M)N \rightarrow_\beta M[x \backslash N]$

appel gauche : $\dfrac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}$ **appel droit :** $\dfrac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'}$

λ-calcul faible avec partage

Les arguments ne sont évalués qu'une fois :

exemple : Si $M \rightarrow_{\beta_p} N$ alors $(\lambda x.xx)M \rightarrow_{\beta_p} MM \rightarrow_{\beta_p} NN$

INRIA

Un codage simple du λ-calcul en LCC

Codage des λ-termes par une fonction qui associe à une variable un agent LCC

$$
\begin{array}{lcl}
[\![x]\!](y) & = & (x = y) \\
[\![\lambda x.M]\!](y) & = & \forall xz(apply(y,x,z) \Rightarrow [\![M]\!](z)) \\
[\![MN]\!](y) & = & \exists zz'.\,(apply(z,z',y) \parallel [\![M]\!](z) \parallel [\![N]\!](z'))
\end{array}
$$

Théorème (Correction)

Si $M \rightarrow^{*}_{\beta_p} N$ *alors il existe* κ *tq* $\langle \emptyset; \mathbf{1}; [\![M]\!](y) \rangle \xrightarrow{*} \kappa$ *et* $\mathcal{O}^{ca}(\kappa) = \mathcal{O}^{ca}([\![N]\!](y))$

Théorème (Complétude)

Si $\langle \emptyset; \mathbf{1}; [\![M]\!](y) \rangle \xrightarrow{*} \kappa$ *alors il existe* N *tq* $M \rightarrow^{*}_{\beta_p} N$ *et* $\mathcal{O}^{ca}(\kappa) = \mathcal{O}^{ca}([\![N]\!](y))$

☞ LCC possède la puissance des fermetures fonctionnelles

INRIA

Un codage simple du λ-calcul en LCC

Codage des λ-termes par une fonction qui associe à une variable un agent LCC

$$\begin{array}{rcl} [\![x]\!](y) & = & (x = y) \\ [\![\lambda x.M]\!](y) & = & \forall xz(apply(y,x,z) \Rightarrow [\![M]\!](z)) \\ [\![MN]\!](y) & = & \exists zz'.\,(apply(z,z',y) \parallel [\![M]\!](z) \parallel [\![N]\!](z')) \end{array}$$

Théorème (Correction)

Si $M \rightarrow^*_{\beta_p} N$ *alors il existe* κ *tq* $\langle \emptyset; \mathbf{1}; [\![M]\!](y) \rangle \xrightarrow{*} \kappa$ *et* $\mathcal{O}^{ca}(\kappa) = \mathcal{O}^{ca}([\![N]\!](y))$

Théorème (Complétude)

Si $\langle \emptyset; \mathbf{1}; [\![M]\!](y) \rangle \xrightarrow{*} \kappa$ *alors il existe* N *tq* $M \rightarrow^*_{\beta_p} N$ *et* $\mathcal{O}^{ca}(\kappa) = \mathcal{O}^{ca}([\![N]\!](y))$

☞ LCC possède la puissance des fermetures fonctionnelles

INRIA

Remarques sur le codage du λ-calcul en LCC

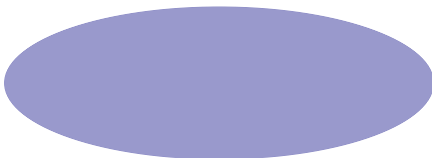
- la reduction **ask** de LCC correspond à la β-reduction
- codage complètement **compositionnel**
  (à la différence des codages du λ-calcul en CC [Laneve & Montanari 92])
- il peut être rendu plus déterministe grâce aux tokens :
  - stratégie "**appel par valeur**" (cf. SML, ocaml)
  - vraie stratégie **paresseuse** (cf. Haskell)
- codage est très **simple**
  (à la différence des codages du λ-calcul en π-calcul synchrone [Milner 90])

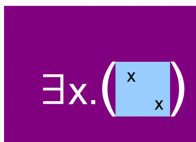
INRIA

Les Modules Syntaxiques : un mécanisme de Protection du code

Introduction au problème de Masquage dans LCC

L'opérateur $(\exists)$ de LCC masque partiellement

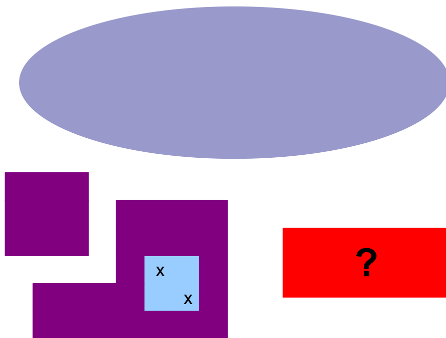




☞Le $(\forall)$ permet de contourner le masquage des variables imposé par le $(\exists)$

Introduction au problème de Masquage dans LCC

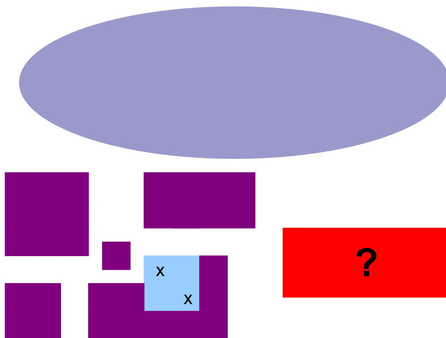
L'opérateur ($\exists$) de LCC masque partiellement



☞Le ($\forall$) permet de contourner le masquage des variables imposé par le ($\exists$)

Introduction au problème de Masquage dans LCC

L'opérateur (∃) de LCC masque partiellement

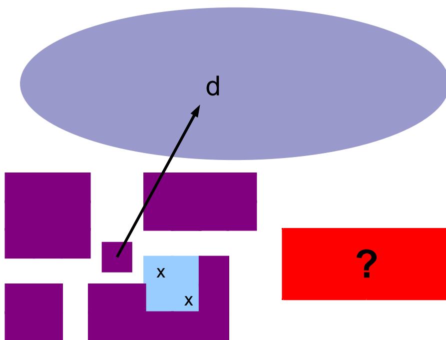


☞Le (∀) permet de contourner le masquage des variables imposé par le (∃)

INRIA

Introduction au problème de Masquage dans LCC

L'opérateur (∃) de LCC masque partiellement

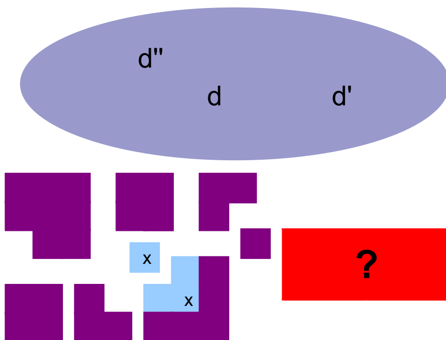


☞Le (∀) permet de contourner le masquage des variables imposé par le (∃)

INRIA

Introduction au problème de Masquage dans LCC

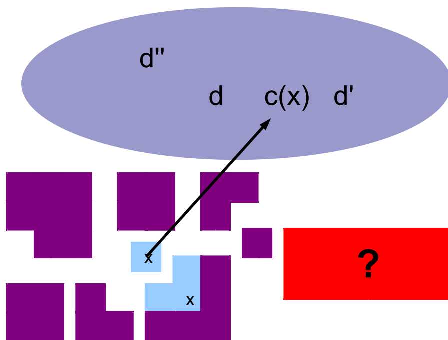
L'opérateur (∃) de LCC masque partiellement



☞Le (∀) permet de contourner le masquage des variables imposé par le (∃)

Introduction au problème de Masquage dans LCC

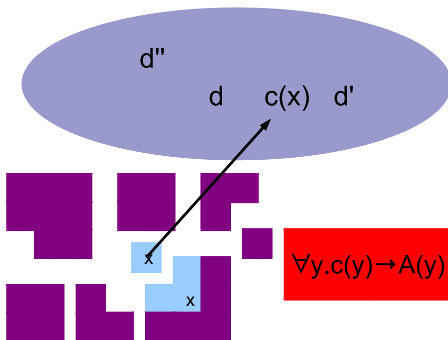
L'opérateur (∃) de LCC masque partiellement



☞Le (∀) permet de contourner le masquage des variables imposé par le (∃)

Introduction au problème de Masquage dans LCC

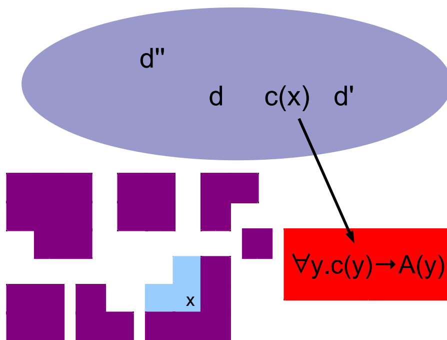
L'opérateur ($\exists$) de LCC masque partiellement



☞Le ($\forall$) permet de contourner le masquage des variables imposé par le ($\exists$)

Introduction au problème de Masquage dans LCC

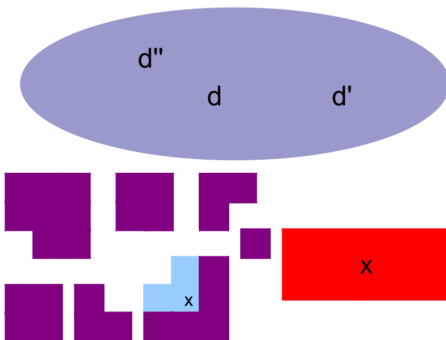
L'opérateur $(\exists)$ de LCC masque partiellement



☞Le $(\forall)$ permet de contourner le masquage des variables imposé par le $(\exists)$

Introduction au problème de Masquage dans LCC

L'opérateur $(\exists)$ de LCC masque partiellement



Le $(\forall)$ permet de contourner le masquage des variables imposé par le $(\exists)$

Des Modules Syntaxiques pour LCC

Syntaxe

$$A ::= x\{A\} \mid x:p \mid d \mid A \| A \mid \exists x.A \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$$

- $x\{A\}$ dénote la création d'un module x contenant l'agent A
- $x:p$ dénote l'ajout du token p dans le module x
- la pose d'une contrainte classique d sera toujours globale

Traduction en LCC : A^x est l'interprétation de A dans le module x

$$p(\vec{t})^x = \dot{p}(x, \vec{t}) \qquad (y\{A\})^x = A^y \qquad (y:p)^x = p^y \qquad d(\vec{t})^x = d(\vec{t})$$

$$(\exists y.A)^x = \exists y.A^x \qquad (c_1 \otimes c_2)^x = c_1^x \otimes c_2^x \qquad (A \| B)^x = A^x \| B^x$$

$$(\forall \vec{y}(c \rightarrow A))^x = \forall \vec{y}(c^x \rightarrow A^x) \qquad (\forall \vec{y}(c \Rightarrow A))^x = \forall \vec{y}(c^x \Rightarrow A^x)$$

avec $x \notin \mathcal{V}(y, \vec{y})$, d une contrainte classique, p un token

☞ le premier argument des tokens n'est jamais quantifié par $\forall$

INRIA

Des Modules Syntaxiques pour LCC

Syntaxe

$A ::= x\{A\} \mid x : p \mid d \mid A \| A \mid \exists x.A \mid \forall \vec{x}(c \rightarrow A) \mid \forall \vec{x}(c \Rightarrow A)$

- $x\{A\}$ dénote la création d'un module x contenant l'agent A
- $x : p$ dénote l'ajout du token p dans le module x
- la pose d'une contrainte classique d sera toujours globale

Traduction en LCC : A^x est l'interprétation de A dans le module x

$$p(\vec{t})^x = \dot{p}(x, \vec{t}) \qquad (y\{A\})^x = A^y \qquad (y : p)^x = p^y \qquad d(\vec{t})^x = d(\vec{t})$$

$$(\exists y.A)^x = \exists y.A^x \qquad (c_1 \otimes c_2)^x = c_1^x \otimes c_2^x \qquad (A \| B)^x = A^x \| B^x$$

$$(\forall \vec{y}(c \rightarrow A))^x = \forall \vec{y}(c^x \rightarrow A^x) \qquad (\forall \vec{y}(c \Rightarrow A))^x = \forall \vec{y}(c^x \Rightarrow A^x)$$

avec $x \notin \mathcal{V}(y, \vec{y})$, d une contrainte classique, p un token

☞ le premier argument des tokens n'est jamais quantifié par $\forall$

INRIA

Examples

l'inversion de liste dans le module *List* en LCC modulaire

$$
\begin{array}{l}
List\{\exists Impl.(\\
\quad \forall XY.rev(X,Y) \Rightarrow Impl : rev(X,[],Y) \parallel \\
\quad Impl\{ \\
\qquad \forall X.rev([],X,X) \Rightarrow \mathbf{1} \parallel \\
\qquad \forall XYZT.rev([X|Y],Z,T) \Rightarrow Impl : rev(Y,[X|Z],T). \\
\quad \} \\
)\}
\end{array}
$$

traduction LCC non-modulaire

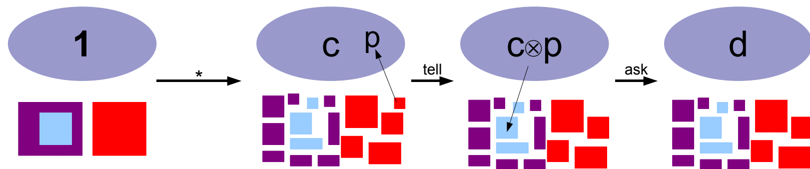
$$
\begin{array}{l}
\exists Impl.(\\
\quad \forall XY.\dot{rev}(List,X,Y) \Rightarrow \dot{rev}(Impl,X,[],Y) \parallel \\
\quad \forall X.\dot{rev}(Impl,[],X,X) \Rightarrow \mathbf{1} \parallel \\
\quad \forall XYZT.\dot{rev}(Impl,[X|Y],Z,T) \Rightarrow \dot{rev}(Impl,Y,[X|Z],T). \\
)
\end{array}
$$

☞ $rev/3$ est protégé dans le sous-module *Impl*

Protection du Code (1/2)

est **protégé** dans :

si pour tout et toute dérivation de la forme

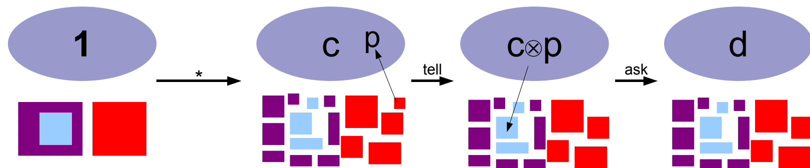


☞ ne peut réveiller les *asks* de que par l'intermédiaire de

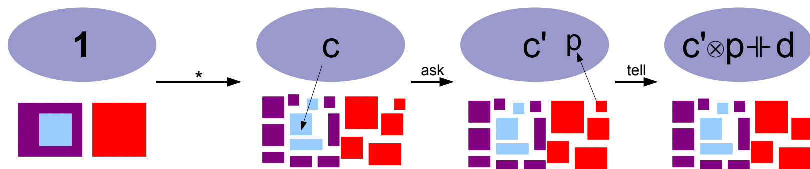
Protection du Code (1/2)

■ est **protégé** dans ■ :

si pour tout ■ et toute dérivation de la forme



on a

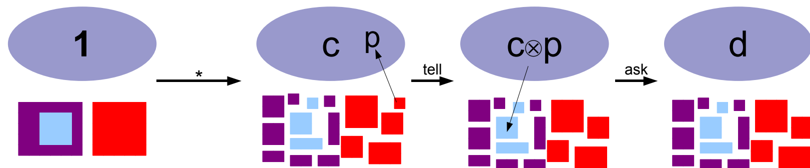


☞ ■ ne peut réveiller les *asks* de ■ que par l'intermédiaire de ■

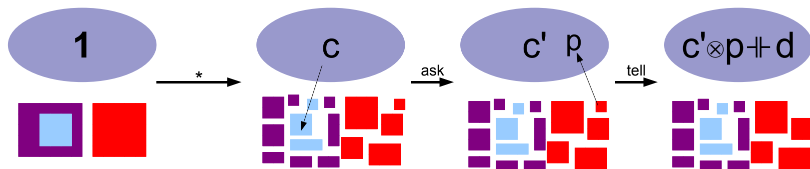
Protection du Code (1/2)

est **protégé** dans :

si pour tout et toute dérivation de la forme



on a



☞ ne peut réveiller les *asks* de que par l'intermédiaire de

Protection du Code (2/2)

Théorème (Protection du code)

Soit A *et* B *deux agents modulaires et* x *une variable.*

Si :

- A *n'a pas de module interne*
- x *n'apparaît que dans des tells de la forme* $x : p$ *avec* $x \notin \mathcal{V}(p)$

alors A *est protégé dans* $\exists x.(x\{A\} \parallel B)$

mCLP : Un Système de Modules Puissant pour la PLC

Syntaxe

```
G ::=  module(T,E){D} | T:p(S1,...,Sn) | p(S1,...,Sn) |
       c(S1,...,Sn) | G,G | G;G
D ::=  p(S1,...,Sn) :- G.D | p(S1,...,Sn).D | :- G.D | ε
```

Interprétation en LCC modulaire

$[\![\mathtt{module(S,E)\{D\}}]\!]^{T} = \mathtt{S}\{[\![D]\!]^{S}_{E}\}$ $\qquad [\![C]\!]^{T} = (C)$ $\qquad [\![G_1, G_2]\!]^{T} = [\![G_1]\!]^{T} \parallel [\![G_2]\!]^{T}$

$[\![P]\!]^{T} = T:P$ $\qquad [\![S:P]\!]^{T} = S:P$ $\qquad [\![G_1 ; G_2]\!]^{T} = [\![G_1]\!]^{T} + [\![G_2]\!]^{T}$

$$[\![p(\vec{t}) \text{ :- } G.D]\!]^{T}_{E} = \forall \vec{X}(p(\vec{X}) \Rightarrow \exists \bar{Y}.[\![\vec{X} = \vec{t}, G]\!]^{S}) \parallel [\![D]\!]^{T}_{E}$$

$$[\![p(\vec{t}).D]\!]^{T}_{E} = \forall \vec{X}(p(\vec{X}) \Rightarrow \exists \bar{Y}.[\![\vec{X} = \vec{t}]\!]^{S}) \parallel [\![D]\!]^{T}_{E}$$

$$[\![\text{ :- } G.D]\!]^{T}_{E} = \exists \bar{Y}.[\![G]\!]^{S} \parallel [\![D]\!]^{T}_{E}$$

où $\vec{X}$ est un ensemble de variables fraîches et $\bar{Y} = \mathcal{V}(\vec{t}, G) \setminus E$.

☞ implémenté dans un prototype de compilateur mCLP(H) vers C

INRIA

Protection du Code en mCLP

```
:- module(sort, [Impl]){

   sort(List, SortedList) :- Impl:qs(List, SortedList).

   :- module(Impl, []){
      qs([],[]).
      qs([H|Tail], Sorted) :- split(H, T, Small, Big),
         qs(Small, SortedSmall), qs(Big, SortedBig),
         append(SortedSmall, [X|SortedBig], Sorted).

      split(X, [],[],[]).
      split(X, [H|T], [H|Small], Big) :- math:(X>Y),
         split(X, T, Small, Big).
      split(X, [H|T], Small, [Y|Big]) :- math:(X=<Y),
         split(X, T, Small, Big).
   }.
}.
```

Fermetures en mCLP

```
:- module(iterator, []){

   forall([], _).
   forall([H|T], C) :- C:apply(H), forall(T, C).

   exists([H|T], C) :- C:apply(H).
   exists([H|T], C) :- exists(T, C).
}.

:-module(fd, []){

   domain(Vars, Min, Max):-
      module(Cl , [Min, Max]){
         apply(X) :- Min=<X , X=<Max.
      },
      iterator:forall(Vars, Cl).
}.
```

INRIA

Modules Paramétriques en mCLP

```
:- module(sort, []){

   generic_sort(Sort, Order) :- module(Sort, [Order,Impl]){

      sort(List, SortedList) :- Impl:qs(List, SortedList).

      :- module(Impl, [Order]){
         qs([],[]).
         qs([H|Tail], Sorted) :- split(H, T, Small, Big),
            qs(Small, SortedSmall), qs(Big, SortedBig),
            append(SortedSmall, [X|SortedBig], Sorted).

         split(X, [],[],[]).
         split(X, [H|T], [H|Small], Big) :- Order:(X>Y),
            split(X, T, Small, Big).
         split(X, [H|T], Small, [Y|Big]) :- Order:(X=<Y),
            split(X, T, Small, Big).
      }.
   }.
}.
```

Vers l'implémentation effective de SiLCC

- **lpsolve-inc** : extension incrémentale de lpsolve (simplexe) :
  ⇨ solveur de contraintes sur les nombres réels
  `http://contraintes.inria.fr/~haemmerl/lpsolve-inc/`

- **GNU Prolog RH** : extension de gprolog avec variables attribuées :
  `http://contraintes.inria.fr/~haemmerl/gprolog-rh/`

- Extension modulaire de GNU Prolog RH [Haemmerlé & Fages ICLP'06] : (modules statiques + protection du code + fermetures ad-hoc)
  - Développement d'un parser dynamique [de Rauglaudre]
  - Portage de CHR [Bouissou 04 master]

- **mCLP** : modules référencés par des variables (compilation vers C) :
  `http://contraintes.inria.fr/~haemmerl/pub/mclp.tgz`
  [Haemmerlé, Fages & Soliman FSTTCS'07]

INRIA

Conclusions & Perspectives

INRIA

Conclusions

- L'ajout de asks persistants à LCC permet :
  - la gestion des fermetures et de modules paramétriques
  - l'internalisation d'un système de modules avec protection du code
- Cette approche pour la conception de modules est mono-paradigme et premier ordre
- Les modules ont une sémantique logique simple en Logique Linéaire
- La protection du code a été définie par une propriété de commutation dan les dérivations LCC

Perspectives

- implémenter effectivement SiLCC
- proposer un système de type flexible pour les agents protégés
- définition de protection du code dans les languages concurrents
- porter notre système de module pour d'autres langages (CHR, Ciao ...)

Merci