Des Programmes Impératifs vers la Logique Équationnelle pour la Vérification

Olivier Ponsini

sous la direction de Carine Fédèle et Emmanuel Kounalis

24 novembre 2005

Université Nice Sophia Antipolis

i3s sophia antipolis

CNRS Centre National de la Recherche Scientifique

Contexte

- Omniprésence de l'informatique
- Besoin de fiabilité logicielle
  - spécification des programmes
  - correction des programmes
- Réalité industrielle
  - intérêt pour les langages impératifs
  - tests
  - méthodes formelles

Exemple

```
int somme_n(int n) {
  int s = 0;

  while(n > 0) {
    s = s + n;
    n = n - 1;
  }
  return s;
}
```

```
int somme_n2(int n) {
  int s = 0;

  while(n > 0) {
    n = n - 1;
    s = s + n;
  }
  return s;
}
```

Propriété : $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Approches Existantes

Méthodes Formelles

- Approche opérationnelle
- Approche axiomatique
- Approche dénotationnelle
- Approche algébrique

Exemples de méthodes

- Logique de Floyd-Hoare
- Algebraic Denotational Semantics (ADS)

Approches Existantes

Méthodes Formelles

- Approche opérationnelle
- Approche axiomatique
- Approche dénotationnelle
- Approche algébrique

Exemples de méthodes

- Logique de Floyd-Hoare
  - approche axiomatique
  - annotation des programmes
  - règles logiques pour raisonner
- Algebraic Denotational Semantics (ADS)

Approche Axiomatique

Logique de Floyd-Hoare [Floyd 67, Hoare 69]

$\{N \geq 0\}$

```
int somme_n(int n) {
  int s = 0;
  while(n > 0) {
    s = s + n;
    n = n - 1;
  }
  return s;
}
```

$\{somme\_n(N) = \mathrm{sum}(N)\}$

Approche Axiomatique

Logique de Floyd-Hoare [Floyd 67, Hoare 69]

$\{N \geq 0\}$

```
int somme_n(int n) {
  int s = 0;
  while(n > 0) {
    s = s + n;
    n = n - 1;
  }
  return s;
}
```

$\{somme\_n(N) = \mathrm{sum}(N)\}$

$$\frac{\{C \wedge I\}\mathtt{E}\{I\}}{\{I\}\mathtt{while(C)\ E}\{\neg C \wedge I\}}$$

Approche Axiomatique

Logique de Floyd-Hoare [Floyd 67, Hoare 69]

$\{N \geq 0\}$
```
int somme_n(int n) {
  int s = 0;
```
$\{I = (n \geq 0 \wedge \mathrm{sum}(N) = \mathrm{sum}(n) + s)\}$
```
  while(n > 0) {
```
$\{n > 0 \wedge I\}$
```
    s = s + n;
    n = n - 1;
```
$\{I\}$
```
  }
```
$\{\neg(n > 0) \wedge I\}$
```
  return s;
}
```
$\{somme\_n(N) = \mathrm{sum}(N)\}$

$$\frac{\{C \wedge I\}\mathtt{E}\{I\}}{\{I\}\mathtt{while(C)\ E}\{\neg C \wedge I\}}$$

Approches Existantes

Méthodes Formelles

- Approche opérationnelle
- Approche axiomatique
- Approche dénotationnelle
- Approche algébrique

Exemples de méthodes

- Logique de Floyd-Hoare
- Algebraic Denotational Semantics (ADS)
  - approche algébrique
  - logique équationnelle
  - modélisation abstraite de la mémoire

Approche Algébrique

ADS/OBJ3 [Goguen et Malcolm 96]

Définition de l'invariant

$$inv(M) \quad = \quad \text{sum}(n) \text{ is } \text{sum}(M[['N]]) + M[['S]] \text{ and } M[['N]] \geq 0$$

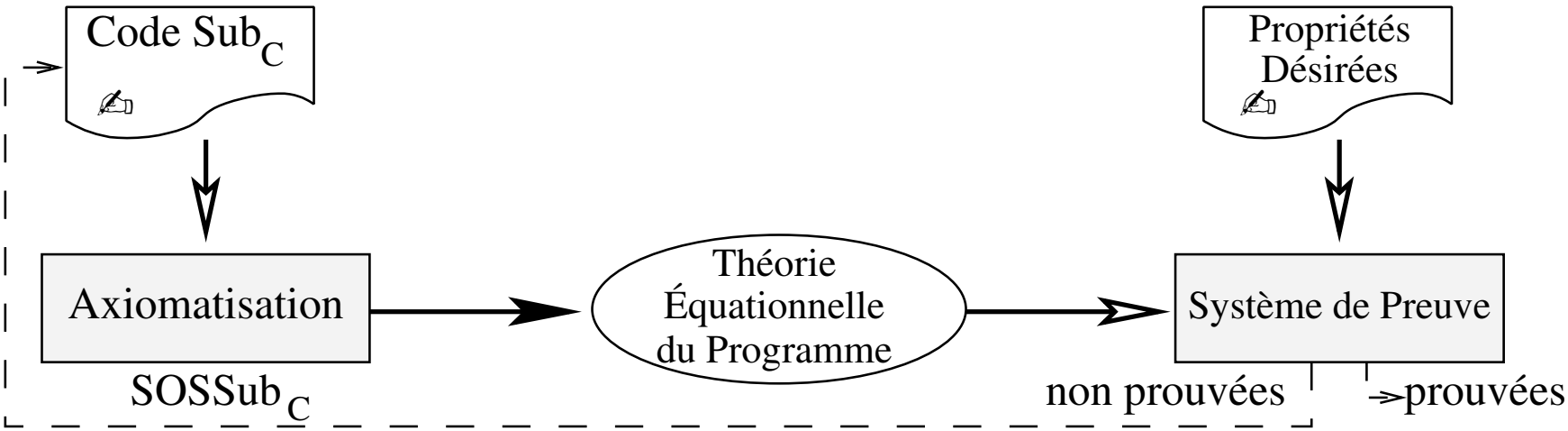
Préservation de l'invariant

$$m[['N]] > 0 = true \wedge inv(m) \stackrel{?}{\Rightarrow} inv(m\,;\,'S := 'S + 'N\,;\,'N := 'N - 1) = true$$

Notre Approche

Motivation et aperçu

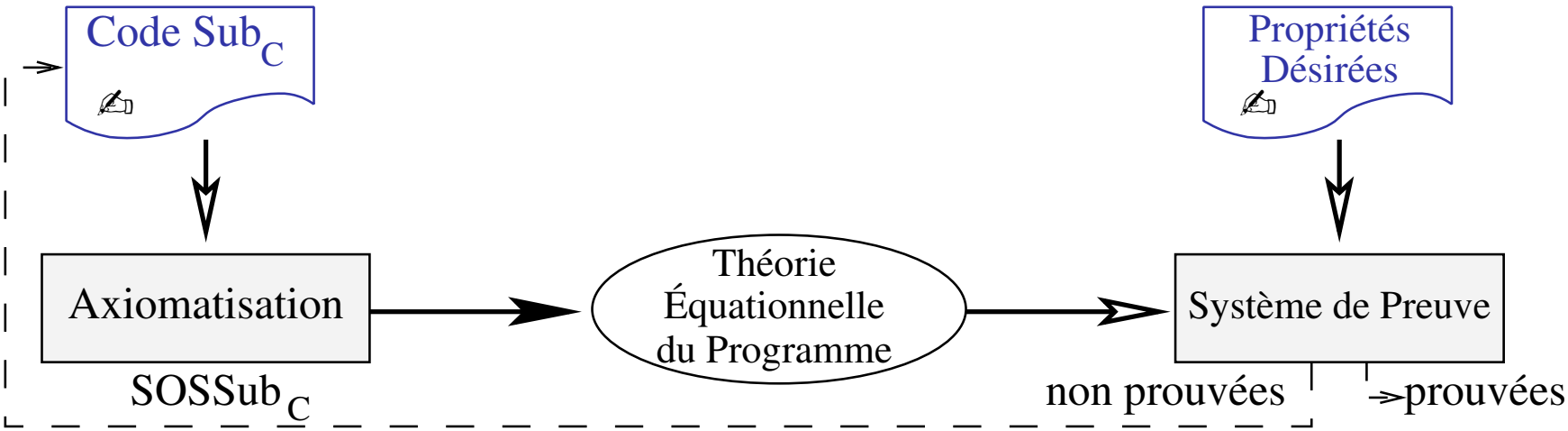
- Exprimer programmes et propriétés dans la logique équationnelle
  - la sémantique des programmes
  - sans modèle de la mémoire



Notre Approche

Motivation et aperçu

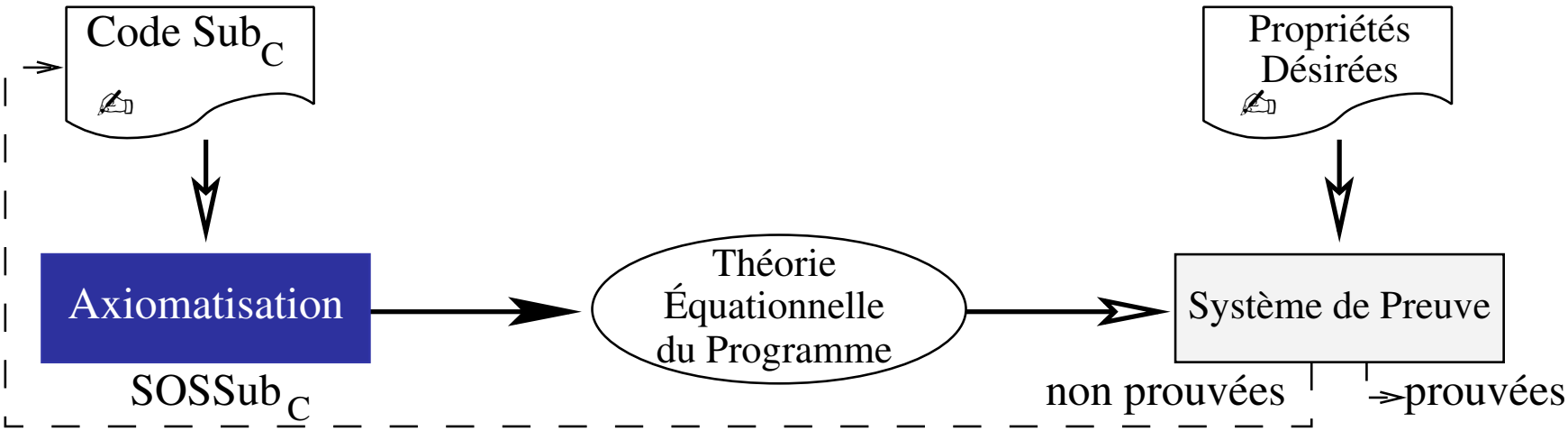
- Exprimer programmes et propriétés dans la logique équationnelle
  - la sémantique des programmes
  - sans modèle de la mémoire



Notre Approche

Motivation et aperçu

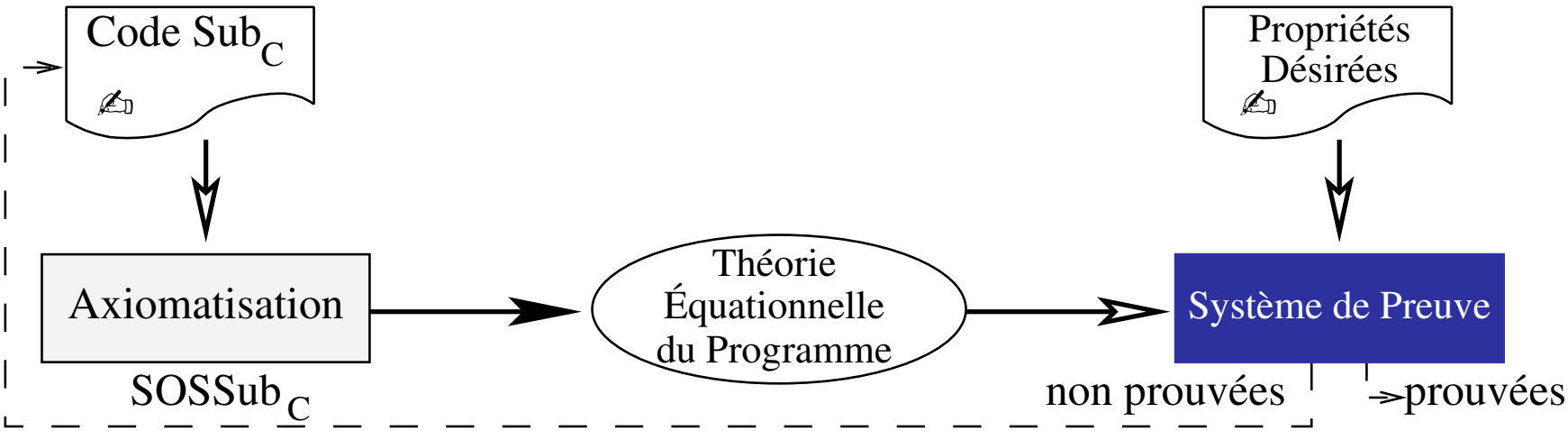
- Exprimer programmes et propriétés dans la logique équationnelle
  - la sémantique des programmes
  - sans modèle de la mémoire



Notre Approche

Motivation et aperçu

- Exprimer programmes et propriétés dans la logique équationnelle
  - la sémantique des programmes
  - sans modèle de la mémoire



Pourquoi la Logique Équationnelle ?

- Simplicité du formalisme
  - syntaxe concise
  - sémantique claire
- Puissance
- Capacité à être automatisée
  - algorithmes efficaces
  - nombreux outils existants
- Largement répandue

Processus de Preuve

Exemple

Équations du programme

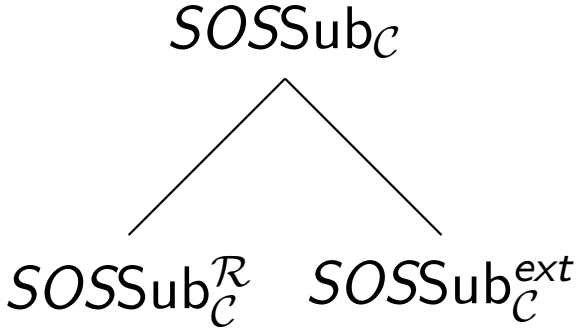
$$\begin{aligned} somme\_n(n) &= loop(n, 0) \\ \neg(n > 0) &\Rightarrow loop(n, s) = s \\ n > 0 &\Rightarrow loop(n, s) = loop(n - 1, s + n) \end{aligned}$$

Propriété du programme à vérifier

$$n \geq 0 \Rightarrow somme\_n(n) = n * (n + 1)/2$$

Notre Contribution

- Conception et développement du système *SOS*Sub$_{\mathcal{C}}$
  - traduction des programmes en équations
- Deux déclinaisons du système



Plan

Introduction

Notions préliminaires
- Logique équationnelle
- Systèmes de Réécriture
- Langage Sub$_{\mathcal{C}}$

Langage Sub$_{\mathcal{C}}$ sans référence

Langage Sub$_{\mathcal{C}}$ avec référence

Conclusion

Logique Équationnelle

- Sous-ensemble de la logique du premier ordre
- Substitution de termes égaux
- Formules de la logique :

$$e_1 \wedge e_2 \wedge \cdots \wedge e_n \Rightarrow e_0$$

avec $\forall 0 \leq i \leq n,\ e_i \equiv t_i = u_i$

Exemples

$$(x.y).z = x.(y.z) \qquad z \geq 0 = true \Rightarrow abs(z) = z$$

$$x.e = x \qquad z < 0 = true \Rightarrow abs(z) = -z$$

$$x.x^{-1} = e$$

Systèmes de Réécriture

- Ensemble d'équations orientées en règles
- Processus de réécriture : application des règles
- Propriétés
  - terminaison
  - confluence

Systèmes de Réécriture

- Ensemble d'équations orientées en règles
- Processus de réécriture : application des règles
- Propriétés
  - terminaison
  - confluence

Exemples

$$(x.y).z = x.(y.z) \qquad z \geq 0 = true \Rightarrow abs(z) = z$$

$$x.e = x \qquad z < 0 = true \Rightarrow abs(z) = -z$$

$$x.x^{-1} = e$$

Systèmes de Réécriture

- Ensemble d'équations orientées en règles
- Processus de réécriture : application des règles
- Propriétés
  - terminaison
  - confluence

Exemples

$$(x.y).z \longrightarrow x.(y.z) \qquad z \geq 0 = true \Rightarrow abs(z) \longrightarrow z$$

$$x.e \longrightarrow x \qquad z < 0 = true \Rightarrow abs(z) \longrightarrow -z$$

$$x.x^{-1} \longrightarrow e$$

Langage Sub$_\mathcal{C}$

- Langage impératif typé : entiers et listes
- Inspiré du langage C
- Constructions disponibles :
  - affectation
  - séquence
  - conditionnelle
  - itération
  - sous-programme

Langage Sub$_{\mathcal{C}}$

- Opérateurs disponibles :
  - opérateurs usuels
    - arithmétiques
    - logiques
    - relationnels
  - opérateurs sur les listes
    - `NULL`
    - `element(L)`
    - `next(L)`
    - `add(elt, L)`

Plan

Introduction

Notions préliminaires

Langage Sub$_{\mathcal{C}}$ sans référence
- Le problème
- Principes
- Exemple

Langage Sub$_{\mathcal{C}}$ avec référence

Conclusion

Problème

Axiomatisation

Entrée : Un programme Sub$_{\mathcal{C}}$ P

Sortie : Un ensemble d'équations qui expriment la sémantique du programme P

- Utiliser le code source non annoté
- Traduire des programmes incomplets
- Abstraire les programmes par des fonctions de transfert des entrées vers les sorties
- Ne pas expliciter la mémoire dans les équations

Principes de Notre Solution

Définir la sémantique du langage dans le but d'obtenir celle du programme

- Environnement d'un programme = ensemble d'équations
- Décrire la sémantique des constructions par des modifications dans l'environnement d'un programme
- Évaluer symboliquement les programmes avec cette sémantique

Principes de Notre Solution

Définir la sémantique du langage dans le but d'obtenir celle du programme

- Environnement d'un programme = ensemble d'équations
- Décrire la sémantique des constructions par des modifications dans l'environnement d'un programme
- Évaluer symboliquement les programmes avec cette sémantique

Pierre angulaire de la traduction

- Système de réécriture

Détails de l'Axiomatisation

- La définition de la sémantique est donnée par les règles d'un système de réécriture
  - définition formelle
  - définition exécutable

Règles pour l'affectation

$$Comp(Assign(var, exp), EmptyEnv) \longrightarrow Env((var, exp), EmptyEnv)$$

$$Comp(Assign(var, exp), Env(pair, env)) \longrightarrow UpdateEnv((var, exp), Env(pair, env))$$

Détails de l'Axiomatisation

- Les trois étapes de l'axiomatisation d'un programme
  1. le code source est transformé, syntaxiquement, en un terme
  2. le terme est réécrit en l'environnement final
  3. l'environnement est formulé en équations

Intuition de la Sémantique

- Affectation
- Conditionnelle
- Itération

Intuition de la Sémantique

- Affectation

  `x=2 ;`
- Conditionnelle
- Itération

Intuition de la Sémantique

- Affectation
  `x=2 ;` représentée par $(x, 2)$
- Conditionnelle
- Itération

Intuition de la Sémantique

- Affectation
  `x=2` ; représentée par $(x, 2)$ signifie $x = 2$
- Conditionnelle
- Itération

Intuition de la Sémantique

- Affectation
- Conditionnelle
  - partage les sous-programmes en chemins d'exécution
  - introduit une équation conditionnelle
- Itération

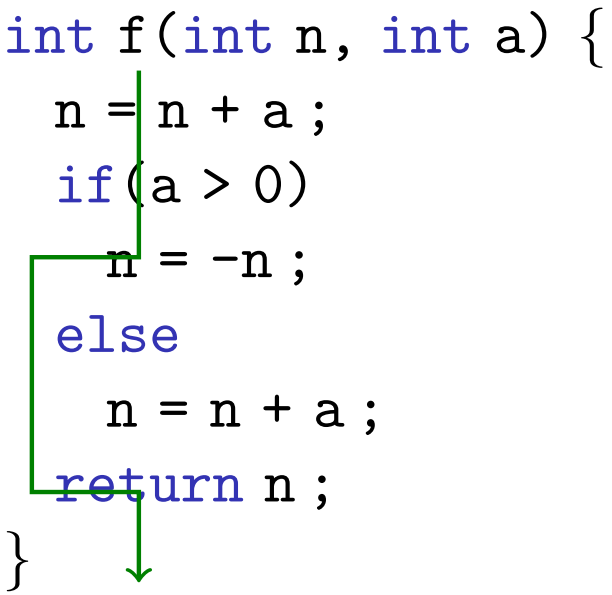
Intuition de la Sémantique

Conditionnelle

```
int f(int n, int a) {
  n = n + a ;
  if(a > 0)
    n = -n ;
  else
    n = n + a ;
  return n ;
}
```

Intuition de la Sémantique

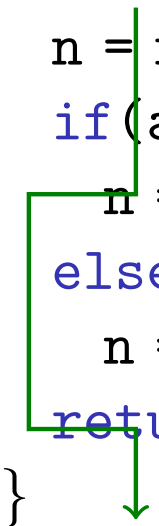
Conditionnelle



Intuition de la Sémantique

Conditionnelle

```
int f(int n, int a) {
  n = n + a ;
  if(a > 0)
    n = -n ;
  else
    n = n + a ;
  return n ;
}
```

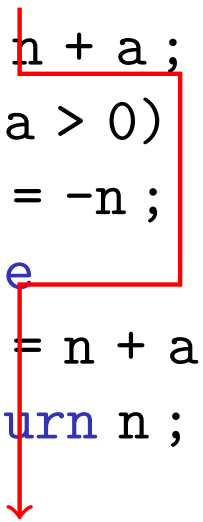


$a > 0 = true \Rightarrow f(n, a) =$ -(n+a)

Intuition de la Sémantique

Conditionnelle

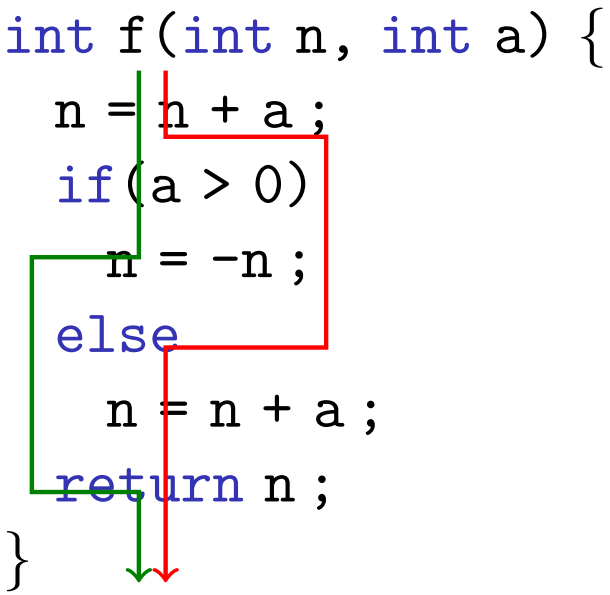
```
int f(int n, int a) {
  n = n + a ;
  if(a > 0)
    n = -n ;
  else
    n = n + a ;
  return n ;
}
```



$$\neg(a > 0) = true \Rightarrow f(n, a) = (n + a + a)$$

Intuition de la Sémantique

Conditionnelle



$$a > 0 = \mathit{true} \Rightarrow f(n, a) = \text{-(n+a)}$$
$$\neg(a > 0) = \mathit{true} \Rightarrow f(n, a) = (n + a + a)$$

$$\mathit{Choice}(\mathit{Branch}(a > 0, (n, -(n + a))),$$
$$\mathit{Branch}(\neg(a > 0), (n, n + a + a)))$$

Intuition de la Sémantique

- Affectation
- Conditionnelle
- Itération
  - fonction récursive terminale

Intuition de la Sémantique

Itération

```
int somme_n(int n) {
  int s = 0;

  while(n > 0) {
    s = s + n;
    n = n - 1;
  }

  return s;
}
```

Intuition de la Sémantique

Itération

```
int somme_n(int n) {
 int s = 0;

 while(n > 0) {
   s = s + n;
   n = n - 1;
 }

 return s;
}
```

```
int loop_n(int n, int s) {
  if(n > 0) {
    s = s + n;
    n = n - 1;
    return loop_n(n, s);
  }
  else return n;
}
```

Intuition de la Sémantique

Itération

```
int somme_n(int n) {
 int s = 0;

 while(n > 0) {
   s = s + n;
   n = n - 1;
 }

 return s;
}
```

```
int loop_s(int n, int s) {
  if(n > 0) {
    s = s + n;
    n = n - 1;
    return loop_s(n, s);
  }
  else return s;
}
```

Intuition de la Sémantique

Itération

```
int somme_n(int n) {
 int s = 0;

 s' = s;
 n' = n;
 s = loop_s(n', s');
 n = loop_n(n', s');

 return s;
}
```

```
int loop_s(int n, int s) {
  if(n > 0) {
    s = s + n;
    n = n - 1;
    return loop_s(n, s);
  }
  else return s;
}
```

Intuition de la Sémantique

Itération

```
while(n > 0) {
  s = s + n;
  n = n - 1;
}
```

- Environnement

$$\mathit{WhileClosure}(n > 0, \{(s, s + n) \cdot (n, n - 1)\})$$

- Équations

$$n > 0 = \mathit{true} \Rightarrow \mathit{loop}_{\mathsf{s}}(n, s) = \mathit{loop}_{\mathsf{s}}(n - 1, s + n)$$
$$\neg(n > 0) = \mathit{true} \Rightarrow \mathit{loop}_{\mathsf{s}}(n, s) = s$$

Exemple

Inversion des éléments d'une liste

```
list inverseSubC(list L) {
  list r;
  while(L != NULL) {
    r = add(element(L), r);
    L = next(L);
  }
  return r;
}
```

$$inverseSubC(L) = loop(L, NULL)$$
$$L = NULL \Rightarrow loop(L, r) = r$$
$$\neg(L = NULL) = true \Rightarrow loop(L, r) = loop(next(L), add(element(L), r))$$

Plan

Introduction

Notions préliminaires

Langage Sub$_{\mathcal{C}}$ sans référence

Langage Sub$_{\mathcal{C}}$ avec référence
- Le problème
- Principes
- Exemple

Conclusion

Problème

Motivation

```
list inverseSubC(list L) {
  list r;

  while(L != NULL) {
    r = add(element(L), r);
    L = next(L);
  }
  return r;
}
```

Problème

Motivation

```
list inverseSubC(list L) {
  list r;

  while(L != NULL) {
    r = add(element(L), r);
    L = next(L);
  }
  return r;
}
```

```
void inverse(LIST L) {
  LIST pre, p;

  while(L != NULL) {
    p = L;
    L = L->next;
    p->next = pre;
    pre = p;
  }
  L = pre;
}
```

Langage Sub$_{\mathcal{C}}^{ext}$

- Une extension du langage Sub$_{\mathcal{C}}$
  - listes mutables
    - `NULL`, `element(L)`, `next(L)`, `add(elt, L)`
    - `L->element`
    - `L->next`
  - appel par référence
    - déclaré par &

Axiomatisation

Entrée : Un programme $\text{Sub}_{\mathcal{C}}^{ext}$ P

Sortie : Un ensemble d'équations qui expriment la sémantique du programme P

- Difficultés liées aux extensions
  - modélisation des listes mutables
  - sémantique des opérations de mutation
  - alias

Modélisation des Listes Mutables

- Étendre la notion d'environnement
  - mémoire dynamique pour les maillons des listes
  - notion de référence
- Structures de données non bornées
  - utiliser la récursivité dans la logique équationnelle
    - pas de représentation en extension des listes
    - analyse de fragments de code sans itération

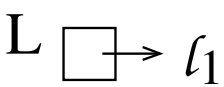
Opérateurs sur les listes

- Expression simple des opérations de mutation dans la modélisation choisie
- Cas des listes dont la structure est inconnue
  - « Matérialiser » les éléments accédés par le programme
  - Schéma de nommage des éléments des listes

Opérateurs sur les listes

- Expression simple des opérations de mutation dans la modélisation choisie
- Cas des listes dont la structure est inconnue
  - « Matérialiser » les éléments accédés par le programme
  - Schéma de nommage des éléments des listes

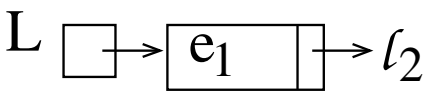
Matérialisation



Opérateurs sur les listes

- Expression simple des opérations de mutation dans la modélisation choisie
- Cas des listes dont la structure est inconnue
  - « Matérialiser » les éléments accédés par le programme
  - Schéma de nommage des éléments des listes

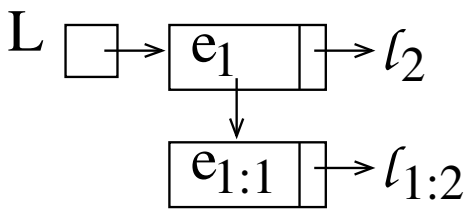
Matérialisation



Opérateurs sur les listes

- Expression simple des opérations de mutation dans la modélisation choisie
- Cas des listes dont la structure est inconnue
  - « Matérialiser » les éléments accédés par le programme
  - Schéma de nommage des éléments des listes

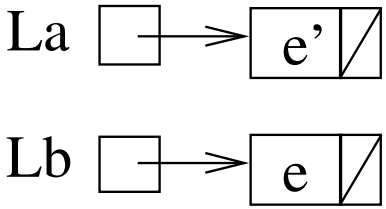
Matérialisation

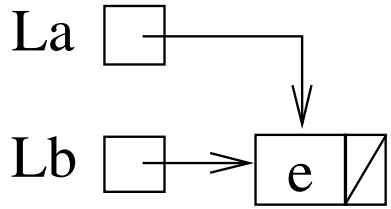


Alias

- Deux noms désignant la même zone de mémoire
  - Influence sur le comportement d'un programme
  - Analyse statique indécidable [Landi 92]

Influence des alias






```
La->element = 1 ;
Lb->element = 2 ;
i = element(La) ;
/* i = 1 */
```

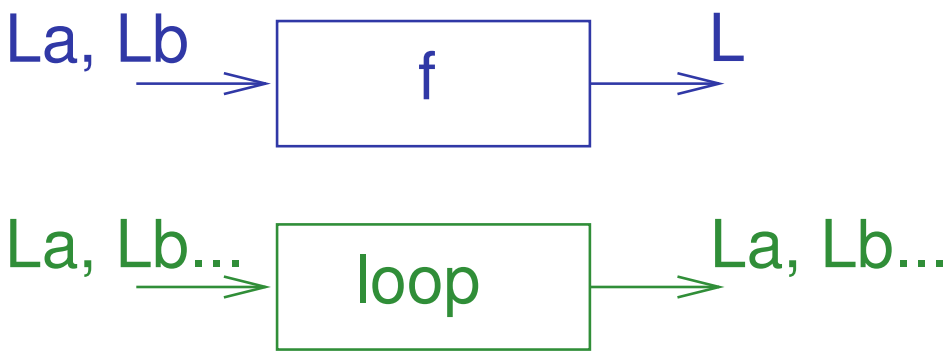
```
La->element = 1 ;
Lb->element = 2 ;
i = element(La) ;
/* i = 2 */
```

Prérequis

- Nécessité de prérequis d'axiomatisation
  - cause : perte des relations d'alias
  - but : garantir que la méconnaissance des alias n'entraîne pas d'ambiguïté

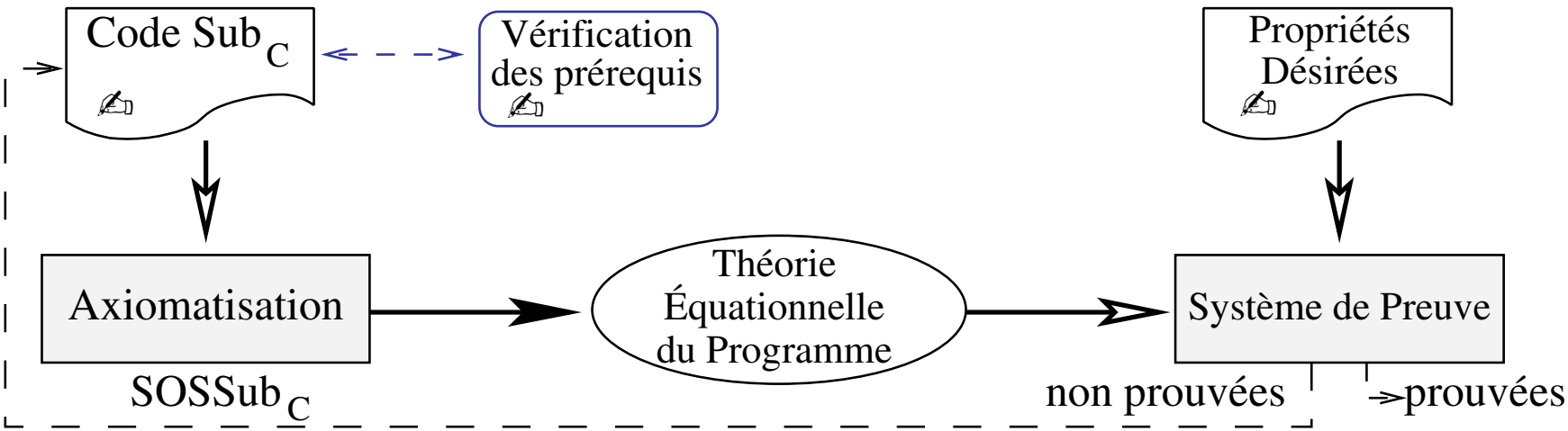
Lieu des pertes des relations d'alias

```
list f(list La, list Lb) {
  ...
  while(cond)
   { ... }
  ... return L;
}
```



Prérequis

- Processus de preuve remanié pour Sub$_{\mathcal{C}}^{ext}$



Exemple

```
void inverse(list &L) {
  list pre, p;
  while(L != NULL) {
    p = L; L = next(L);
    p->next = pre; pre = p;
  }
  L = pre;
}
```

$$
\begin{aligned}
\mathit{inverse}(L) &= \mathit{loop}(L, \mathit{NULL}, \mathit{NULL}) \\
\mathit{loop}(e_1 \cdot l_2, p, \mathit{pre}) &= \mathit{loop}(l_2, \mathit{NULL}, e_1 \cdot \mathit{pre}) \\
\mathit{loop}(\mathit{NULL}, p, \mathit{pre}) &= \mathit{pre}
\end{aligned}
$$

Conclusion

- Traduction des programmes impératifs dans la logique équationnelle
  - richesse du langage de programmation et simplicité de la logique équationnelle
  - sans expliciter la mémoire dans les équations
  - degré d'automatisation
- Conception et développement de deux versions de la traduction
  - formalisation par un système de réécriture pour Sub$_{\mathcal{C}}$
  - énoncé de prérequis à l'axiomatisation en présence de références pour Sub$_{\mathcal{C}}^{ext}$
- Validation de l'approche par des expérimentations sur machine

Limitations

- Propriétés sur les valeurs des entrées et sorties des programmes
- Prérequis
  - restriction sur la classe des programmes axiomatisables
  - vérification non automatique

Perspectives

- Identifier des classes de programmes pour lesquelles les prérequis sont décidables
- Intégrer dans le langage les contraintes sur la formation des alias
- Étendre le langage à des structures de données plus élaborées