

Vérification de Propriétés de Programmes Impératifs

une introduction

Olivier PONSINI

Université de Nice-Sophia Antipolis — Master 1

14 décembre 2005

Plan

Vérification

Logique de Floyd-Hoare

Logique équationnelle

SOSSub \mathcal{C}

Contexte et Motivation

- ▶ Omniprésence de l'informatique
- ▶ Besoin de fiabilité logicielle



Contexte et Motivation

- ▶ Omniprésence de l'informatique
- ▶ Besoin de fiabilité logicielle
 - ▶ tests



Contexte et Motivation

- ▶ Omniprésence de l'informatique
- ▶ Besoin de fiabilité logicielle
 - ▶ tests
 - ▶ montrer la présence d'erreurs, pas leur absence !



Contexte et Motivation

- ▶ Omniprésence de l'informatique
- ▶ Besoin de fiabilité logicielle
 - ▶ tests
 - ▶ montrer la présence d'erreurs, pas leur absence !
 - ▶ méthodes formelles
 - ▶ langages, techniques et outils mathématiques pour la spécification et la vérification des systèmes



Contexte et Motivation

Réalité industrielle

Difficile diffusion des méthodes formelles

- ▶ **Coût financier** de mise en œuvre
 - ▶ temps
 - ▶ expertise
- ▶ **Vide juridique** quant à la responsabilité des développeurs
- ▶ **En train d'évoluer**
 - ▶ pertes financières dues à des erreurs
 - ▶ normes (Critères Communs)

Le Problème de la Correction des Programmes

Définition

Le programme satisfait-il ses spécifications ?

- ▶ Une (petite) partie du problème de la fiabilité des systèmes
- ▶ La partie sans doute la mieux comprise pour l'instant

Spécifier

Pourquoi spécifier formellement ?

- ▶ Décrire une infinité de comportements
 - ▶ révéler ambiguïtés, incohérences, oublis
 - ▶ communiquer efficacement
 - ▶ documenter (guide le développement, facilite la maintenance et la réutilisation)
- ▶ Automatiser les traitements ultérieurs
 - ▶ prototype (si spécification exécutable)
 - ▶ vérification

Spécifier

Limites

- ▶ Correction de la spécification
 - ▶ essence du problème informelle
 - ▶ transition informel/formel invérifiable de façon absolue
- ▶ Complétude de la spécification
 - ▶ **abstraction** mathématique du monde réel
 - ▶ compilateur ? matériel ?
 - ▶ **assomptions** irréductibles sur la modélisation du monde physique

Spécifier

Limites

Ne pas avoir une confiance démesurée dans les méthodes formelles

- ▶ Correct, mais toujours sous certaines hypothèses
- ▶ Choix du niveau acceptable de modélisation et de formalisation

Spécifier

Malgré tout !

Le Processus de spécification formel est vertueux

- ▶ Améliore la compréhension du problème réel
- ▶ Seule méthode à admettre un traitement rigoureux
- ▶ Possibilité d'automatisation
 - ▶ spécification manuelle par nature
 - ▶ vérification
 - ▶ indécidable en général
 - ▶ automatique pour des cas particuliers (démonstrateur de théorèmes)
 - ▶ au moins vérifier mécaniquement la preuve de la vérification (assistant de preuves)

Approches Existantes

- ▶ Synthèse de programme
 - ▶ programmes corrects par construction
 - ▶ spécification exécutable
 - ▶ logiques constructives (preuve = programme)
 - ▶ génération de code (raffinements successifs)
- ▶ Vérification de modèle
 - ▶ modélisation **finie** du système (automate)
 - ▶ parcours exhaustif **automatique** des états du modèle
 - ▶ explosion de l'espace des états

Approches Existantes

- ▶ Vérification par démonstration
 - ▶ raisonner dans des systèmes de calcul permettant de traiter un nombre infini d'états
 - ▶ semi-automatique
 - ▶ adaptée à la vérification de propriétés
 - ▶ adaptée aux programmes séquentiels manipulant des données complexes
 - ▶ adaptée à la vérification de portions de code critiques

Exemple

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

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

Comment raisonner sur les programmes ?

- ▶ Donner un sens au langage de programmation
- ▶ Définir la **sémantique** d'un langage formel = lui donner une signification mathématique
- ▶ Rôles
 - ▶ comprendre les programmes
 - ▶ tester et vérifier les programmes
 - ▶ écrire et comprendre des spécifications
 - ▶ écrire des compilateurs
 - ▶ classer les langages, en concevoir de nouveaux

Sémantiques

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

Exemples de méthodes

- ▶ Logique de Floyd-Hoare
- ▶ SOSSub \mathcal{C}

Plan

Vérification

Logique de Floyd-Hoare

Définition

Correction

Exemple

Logique équationnelle

SOSSub \mathcal{C}

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

- ▶ Approche axiomatique
- ▶ Annotation des programmes
- ▶ Règles logiques pour raisonner

Triplets de Hoare

$$\{P\}I\{Q\}$$

I Une instruction

P L'antécédent (pré-condition)

Q Le conséquent (post-condition)

Définition

Si la propriété P est vraie pour les valeurs des variables du programme avant l'exécution de I , et si l'exécution termine, alors la propriété Q est vraie après l'exécution de I .

Correction Partielle et Totale

- ▶ Partielle : $\models_{par} \{P\}I\{Q\}$, ssi pour tous états σ, σ'
 - ▶ si $\sigma \models P$ et $(I, \sigma) \rightarrow \sigma'$ alors $\sigma' \models Q$
- ▶ Totale : $\models_{tot} \{P\}I\{Q\}$, ssi pour tout état σ
 - ▶ si $\sigma \models P$ alors I **termine** et
 - ▶ pour tout état σ' , si $(I, \sigma) \rightarrow \sigma'$ alors $\sigma' \models Q$

Calcul de Hoare

► Axiome

$$\{P[E/x]\}_{x=E}\{P\}$$

► Règles d'inférence

► séquence

$$\frac{\{P\}I_1\{P'\} \quad \{P'\}I_2\{Q\}}{\{P\}I_1 ; I_2\{Q\}}$$

► conditionnelle

$$\frac{\{C \wedge P\}I_1\{Q\} \quad \{\neg C \wedge P\}I_2\{Q\}}{\{P\}\text{if}(C) I_1 \text{ else } I_2\{Q\}}$$

Calcul de Hoare

- ▶ boucle

$$\frac{\{C \wedge P\} I \{P\}}{\{P\} \text{while}(C) I \{\neg C \wedge P\}}$$
- ▶ conséquence

$$\frac{P \Rightarrow P' \quad \{P'\} I \{Q'\} \quad Q' \Rightarrow Q}{\{P\} I \{Q\}}$$

Correction du calcul de Hoare

Théorème

Si $\vdash \{P\}I\{Q\}$ alors $\models_{par} \{P\}I\{Q\}$

Exemple

```
 $\{N \geq 0\}$   
int somme_n(int n) {  
    int s = 0;  
    while(n > 0) {  
        s = s + n;  
        n = n - 1;  
    }  
    return s;  
}  
 $\{somme\_n(N) = \frac{N(N+1)}{2}\}$ 
```

Exemple

$\{N \geq 0\}$

```
int somme_n(int n) {
```

```
  int s = 0;
```

$\{I = (n \geq 0 \wedge \text{somme}(N) = \text{somme}(n) + s)\}$

```
  while(n > 0) {
```

```
    s = s + n;
```

```
    n = n - 1;
```

```
  }
```

```
  return s;
```

```
}
```

$\{\text{somme}_n(N) = \frac{N(N+1)}{2}\}$

$\{C \wedge I\} E \{I\}$

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

Exemple

```
 $\{N \geq 0\}$   
int somme_n(int n) {  
   $\{n = N \wedge N \geq 0\}$   
  int s = 0;  
   $\{I = (n \geq 0 \wedge \text{somme}(N) = \text{somme}(n) + s)\}$   
  while(n > 0) {  
    s = s + n;  
    n = n - 1;  
  }  
  return s;  
}  
 $\{\text{somme\_n}(N) = \frac{N(N+1)}{2}\}$ 
```

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

Exemple

```
{N ≥ 0}
int somme_n(int n) {
  {n = N ∧ N ≥ 0}
  int s = 0;
  {n = N ∧ N ≥ 0 ∧ s = 0}
  {I = (n ≥ 0 ∧ somme(N) = somme(n) + s)} {C ∧ I}E{I}
  while(n > 0) {
    s = s + n;
    n = n - 1;
  }
  return s;
}
{somme_n(N) =  $\frac{N(N+1)}{2}$ }
```

$$\frac{\{I\} \text{while}(C) E\{\neg C \wedge I\}}{\{I = (n \geq 0 \wedge \text{somme}(N) = \text{somme}(n) + s)\} \{C \wedge I\} E\{I\}}$$

Exemple

```
 $\{N \geq 0\}$   
int somme_n(int n) {  
  int s = 0;  
   $\{I = (n \geq 0 \wedge \text{somme}(N) = \text{somme}(n) + s)\}$   
  while(n > 0) {  
    s = s + n;  
    n = n - 1;  
  }  
  return s;  
}  
 $\{\text{somme}_n(N) = \frac{N(N+1)}{2}\}$ 
```

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

Exemple

```

{N ≥ 0}
int somme_n(int n) {
    int s = 0;
    {I = (n ≥ 0 ∧ somme(N) = somme(n) + s)}
    while(n > 0) {
        {n > 0 ∧ I}
        s = s + n;
        n = n - 1;
        {I}
    }
    return s;
}
{somme_n(N) =  $\frac{N(N+1)}{2}$ }
    
```

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

Exemple

```

{N ≥ 0}
int somme_n(int n) {
    int s = 0;
    while(n > 0) {
        {n > 0 ∧ I}
        s = s + n;
        n = n - 1;
        {I = (n ≥ 0 ∧  $\frac{N(N+1)}{2} = \frac{n(n+1)}{2} + s$ )}
    }
    return s;
}
{somme_n(N) =  $\frac{N(N+1)}{2}$ }
    
```

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

Exemple

```
 $\{N \geq 0\}$   
int somme_n(int n) {  
  int s = 0;  
  while(n > 0) {  
     $\{n > 0 \wedge I\}$   
    s = s + n;  
     $\{n - 1 \geq 0 \wedge \frac{N(N+1)}{2} = \frac{(n-1)n}{2} + s\}$   
    n = n - 1;  
     $\{I = (n \geq 0 \wedge \frac{N(N+1)}{2} = \frac{n(n+1)}{2} + s)\}$   
  }  
  return s;  
}  
 $\{somme\_n(N) = \frac{N(N+1)}{2}\}$ 
```

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

Exemple

```

{N ≥ 0}
int somme_n(int n) {
  int s = 0;
  while(n > 0) {
    {n > 0 ∧ I}
    {n ≥ 1 ∧  $\frac{N(N+1)}{2} = \frac{(n-1)n}{2} + s + n$ }
    s = s + n;
    n = n - 1;
    {I = (n ≥ 0 ∧  $\frac{N(N+1)}{2} = \frac{n(n+1)}{2} + s$ )}
  }
  return s;
}
{somme_n(N) =  $\frac{N(N+1)}{2}$ }

```

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

Exemple

```

{N ≥ 0}
int somme_n(int n) {
    int s = 0;
    {I = (n ≥ 0 ∧ somme(N) = somme(n) + s)}
    while(n > 0) {
        s = s + n;
        n = n - 1;
    }
    {I}
    }
    {¬(n > 0) ∧ I}
    return s;
}
{somme_n(N) =  $\frac{N(N+1)}{2}$ }
    
```

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

Plan

Vérification

Logique de Floyd-Hoare

Logique équationnelle

Définition

Exemples

Systèmes de Réécriture

SOSSub \mathcal{C}

Logique Équationnelle

Langage

- ▶ Sous-ensemble de la logique du premier ordre
- ▶ Substitution de termes égaux
- ▶ Langage
 - ▶ un ensemble \mathcal{F} de **symboles de fonction** ($f, g \dots$) munis d'une **arité**
 - ▶ un ensemble \mathcal{V} de **variables** ($x, y \dots$)
 - ▶ l'ensemble \mathcal{T} des **termes** sur \mathcal{F} et \mathcal{V} , défini récursivement :
 - ▶ $x \in \mathcal{V}$ est un terme
 - ▶ soient $t_1, \dots, t_n \in \mathcal{T}$, $f \in \mathcal{F}$ d'arité n , $f(t_1, \dots, t_n) \in \mathcal{T}$

Logique Équationnelle

Langage

- Formules de la logique équationnelle conditionnelle :

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

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

Exemples

$$(x.y).z = x.(y.z)$$

$$x.e = x$$

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

$$z \geq 0 = \text{true} \Rightarrow \text{abs}(z) = z$$

$$z < 0 = \text{true} \Rightarrow \text{abs}(z) = -z$$

Logique Équationnelle

Axiomes

$$t = u \wedge t' = u' \Rightarrow t = u$$

$$t = t$$

$$t = v \wedge u = v \Rightarrow t = u$$

$$t_1 = u_1 \wedge \dots \wedge t_n = u_n \Rightarrow f(t_1, \dots, t_n) = f(u_1, \dots, u_n)$$

pour tout f d'arité n

Logique Équationnelle

Règles d'inférence

$$\frac{F \Rightarrow t' = u' \wedge E \wedge t' = u' \Rightarrow t = u}{E \wedge F \Rightarrow t = u}$$

$$\frac{E \Rightarrow t = u}{\sigma(E) \Rightarrow \sigma(t) = \sigma(u)}$$

pour toute substitution σ

Avec $E = \{t_1 = u_1 \wedge \dots \wedge t_n = u_n\} (n \geq 0)$,
 $\sigma(E) = \{\sigma(t_1) = \sigma(u_1) \wedge \dots \wedge \sigma(t_n) = \sigma(u_n)\}$,
 $F = \{t'_1 = u'_1 \wedge \dots \wedge t'_m = u'_m\} (m \geq 0)$.

Exemples

$$x + 0 = x$$

$$x + s(y) = s(x + y)$$

Montrer que $1 + 2 = 3$

$$a = b$$

$$f(x) = g(x)$$

Montrer que $g(b) = f(a)$

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

$$(x.y).z = x.(y.z)$$

$$x.e = x$$

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

$$z \geq 0 = true \Rightarrow abs(z) = z$$

$$z < 0 = true \Rightarrow abs(z) = -z$$

Systèmes de Réécriture

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

Exemples

$$(x.y).z \rightarrow x.(y.z)$$

$$x.e \rightarrow x$$

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

$$z \geq 0 = \text{true} \Rightarrow \text{abs}(z) \rightarrow z$$

$$z < 0 = \text{true} \Rightarrow \text{abs}(z) \rightarrow -z$$

Exemples (bis)

$$x + 0 \rightarrow x$$

$$x + s(y) \rightarrow s(x + y)$$

Montrer que $1 + 2 = 3$

$$a \rightarrow b$$

$$f(x) \rightarrow g(x)$$

Montrer que $g(b) = f(a)$

Plan

Vérification

Logique de Floyd-Hoare

Logique équationnelle

SOSSub \mathcal{C}

Motivation et aperçu de l'approche

Langage Sub \mathcal{C}

Le problème

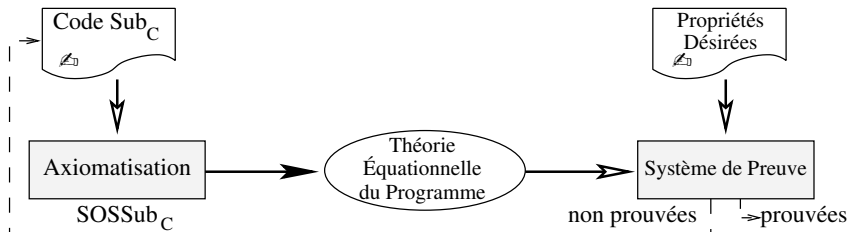
Principes

Exemples

SOSSub_C

Motivation et aperçu

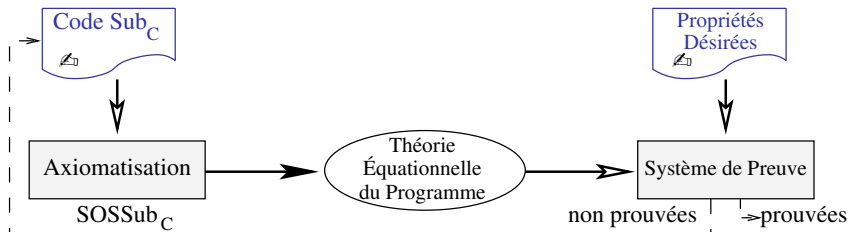
- ▶ Exprimer programmes et propriétés dans la logique équationnelle
 - ▶ approche algébrique
 - ▶ la sémantique des programmes
 - ▶ sans modèle de la mémoire



SOSSub_C

Motivation et aperçu

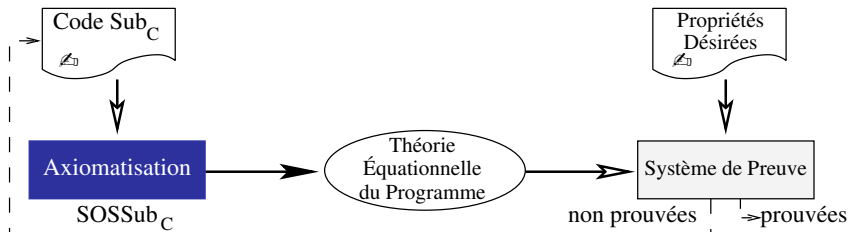
- ▶ Exprimer programmes et propriétés dans la logique équationnelle
 - ▶ approche algébrique
 - ▶ la sémantique des programmes
 - ▶ sans modèle de la mémoire



SOSSub_C

Motivation et aperçu

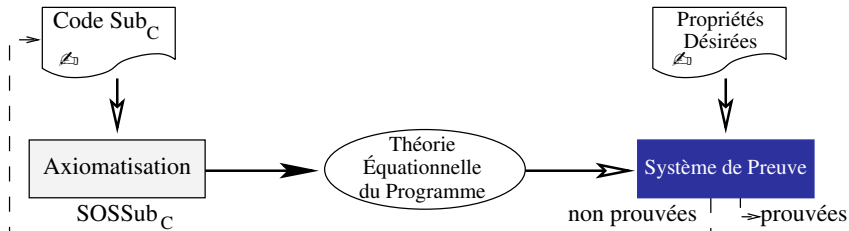
- ▶ Exprimer programmes et propriétés dans la logique équationnelle
 - ▶ approche algébrique
 - ▶ la sémantique des programmes
 - ▶ sans modèle de la mémoire



SOSSub_C

Motivation et aperçu

- ▶ Exprimer programmes et propriétés dans la logique équationnelle
 - ▶ approche algébrique
 - ▶ 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

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

Propriété du programme à vérifier

$$n \geq 0 \Rightarrow \text{somme_n}(n) = n * (n + 1) / 2$$

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)`

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

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

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

$$\text{Comp}(\text{Assign}(\text{var}, \text{exp}), \text{EmptyEnv}) \rightarrow \text{Env}((\text{var}, \text{exp}), \text{EmptyEnv})$$

$$\text{Comp}(\text{Assign}(\text{var}, \text{exp}), \text{Env}(\text{pair}, \text{env})) \rightarrow \text{UpdateEnv}((\text{var}, \text{exp}), \text{Env}(\text{pair}, \text{env}))$$

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

$x=2 ;$

Intuition de la Sémantique

- ▶ Affectation
 $x=2$; représentée par $(x, 2)$

Intuition de la Sémantique

- ▶ Affectation

$x=2$; représentée par $(x, 2)$ signifie $x = 2$

Intuition de la Sémantique

- ▶ Conditionnelle
 - ▶ partage les sous-programmes en chemins d'exécution
 - ▶ introduit une équation 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 ;  
}
```


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

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

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

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

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

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

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

Intuition de la Sémantique

- ▶ 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

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

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

Intuition de la Sémantique

Itération

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

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


Intuition de la Sémantique

Itération

```
s' = s ;  
n' = n ;  
s = loops(n', s') ;  
n = loopn(n', s') ;
```

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

Intuition de la Sémantique

Itération

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

► Environnement

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

► Équations

$$n > 0 = true \Rightarrow loop_s(n, s) = loop_s(n - 1, s + n)$$
$$\neg(n > 0) = true \Rightarrow loop_s(n, s) = s$$

SOSSub \mathcal{C} - Somme des n premiers entiers

Étape 1

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

SOSSub \mathcal{C} - Somme des n premiers entiers

Étape 1

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

$\{Assign(s, 0) \cdot$
 $While(n > 0, \{Assign(s, s + n) \cdot$
 $Assign(n, n - 1)\}) \cdot$
 $Assign(somme_n, s)\}$

SOSSub_C - Somme des n premiers entiers

Étape 2

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

$\{Assign(s, 0) \cdot$
 $While(n > 0, \{Assign(s, s + n) \cdot$
 $Assign(n, n - 1)\}) \cdot$
 $Assign(somme_n, s)\}$

$\{WhileClosure(n > 0, \{(n, n - 1) \cdot$
 $(s, s + n)\}) \cdot$
 $(n, loop_n(n_{eff}, 0)) \cdot$
 $(s, loop_s(n_{eff}, 0)) \cdot$
 $(somme_n, loop_s(n_{eff}, 0))\}$

SOSSub \mathcal{C} - Somme des n premiers entiers

Étape 3

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

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

$$\begin{aligned} &\{ \text{Assign}(s, 0) \cdot \\ &\quad \text{While}(n > 0, \{ \text{Assign}(s, s + n) \cdot \\ &\quad\quad \text{Assign}(n, n - 1) \}) \cdot \\ &\quad \text{Assign}(\text{somme_n}, s) \} \end{aligned}$$

$$\begin{aligned} &\{ \text{WhileClosure}(n > 0, \{ (n, n - 1) \cdot \\ &\quad (s, s + n) \}) \cdot \\ &\quad (n, \text{loop}_n(n_{\text{eff}}, 0)) \cdot \\ &\quad (s, \text{loop}_s(n_{\text{eff}}, 0)) \cdot \\ &\quad (\text{somme_n}, \text{loop}_s(n_{\text{eff}}, 0)) \} \end{aligned}$$

SOSSub \mathcal{C} - Somme des n premiers entiers

Axiomatisation

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

$$\text{somme_n}(n) = \text{loop}(n, 0)$$

$$\neg(n > 0) \Rightarrow \text{loop}(n, s) = s$$

$$n > 0 \Rightarrow \text{loop}(n, s) = \text{loop}(n - 1, s + n)$$

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;  
}
```


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;  
}
```

$$\begin{aligned} & \text{inverseSubC}(L) = \text{loop}(L, \text{NULL}) \\ & L = \text{NULL} \Rightarrow \text{loop}(L, r) = r \\ & \neg(L = \text{NULL}) = \text{true} \Rightarrow \text{loop}(L, r) = \\ & \qquad \text{loop}(\text{next}(L), \text{add}(\text{element}(L), r)) \end{aligned}$$