Sémantiques mécanisées, deuxième cours

Traduttore, traditore: vérification formelle d'un compilateur

Xavier Leroy

2019-12-12

Collège de France, chaire de sciences du logiciel

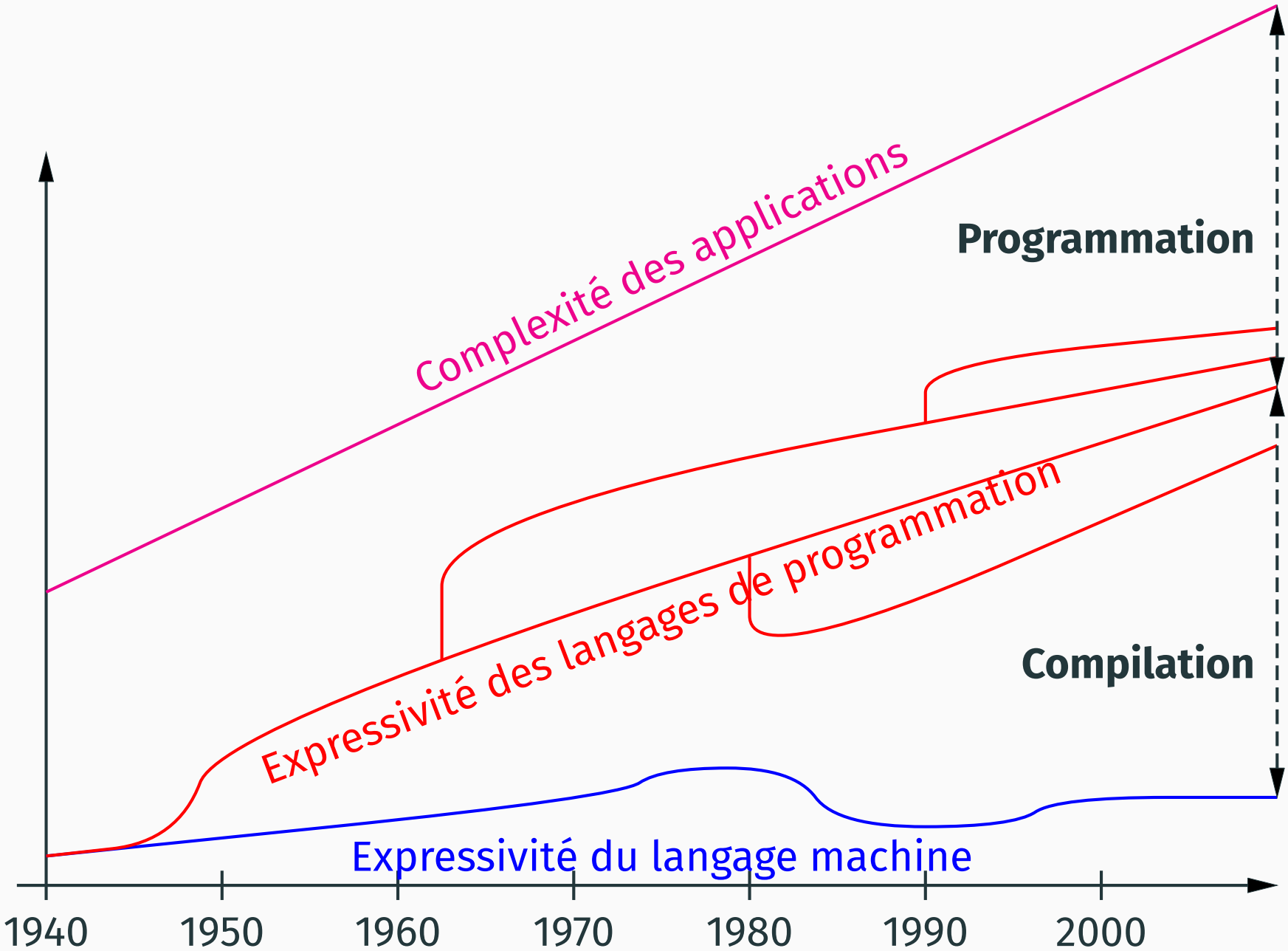
La compilation

En général : toute traduction automatique d'un langage informatique vers un autre.

Plus précisément : une traduction automatique

- d'un langage source utilisable par les programmeurs
- vers un langage machine exécutable par l'ordinateur
- avec un souci d'efficacité : rapidité d'exécution, faible taille de code, économie d'énergie.

Une vision historique de la compilation



Les premiers compilateurs

1953 Les *autocoders* A-0, A-1, A-2 (G. Hopper, Rand Remington)
≪I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic ; they could not do programs≫

1957 Le *translator* Fortran 1 (J. Backus et al, IBM)
Premier compilateur incluant des optimisations de boucles.

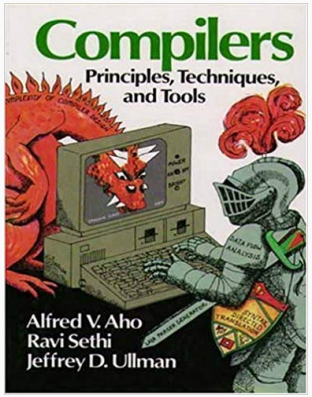
1960 Premier compilateur Algol 60 (E. Dijkstra, U. Amsterdam)
Utilisation d'une pile pour implémenter la récursion et l'appel par nom.

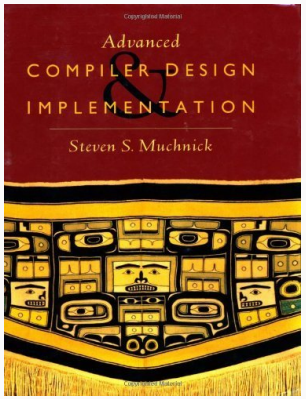
1962 Premier compilateur Lisp (T. Hart et M. Levin, MIT)
Avc auto-amorçage (*bootstrap*).

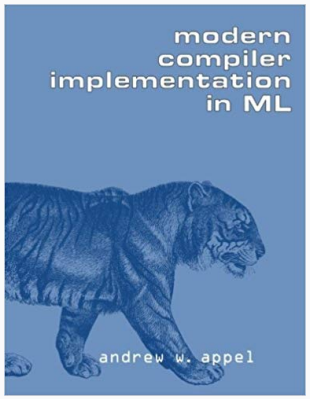
Quelques grandes tendances en compilation

1970's	Génération automatique d'analyseurs syntaxiques (p.ex. `lex` et `yacc`).
1980's	L'approche RISC : allocation de registres, ordonnancement d'instructions (*scheduling*).
1990's	La forme intermédiaire SSA (*Static Single Assignment*).
2000's	Compilation dynamique optimisée de langages de script (moteurs JavaScript).

La compilation aujourd'hui







Un domaine mature de l'informatique.

Vaste corpus d'algorithmes de génération de code et d'optimisation des performances.

De nombreux compilateurs (libres ou commerciaux) qui mettent en œuvre des transformations subtiles.

Un exemple de compilation optimisante

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{i<n} a_i b_i$$

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

Compilé avec un bon compilateur, puis décompilé à la main en C.

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

```
L17: f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
     f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
     f12 = a[4]; f16 = f18 * f16;
     f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
     f11 += f17; r1 += 4; f10 += f15;
     f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
     f1 += f16; dp += f19; b += 4;
     if (r1 < r2) goto L17;
```

```
double dotproduct(int n, double a[], double b[]) {
     dp = 0.0;
     if (n <= 0) goto L5;
     r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
     if (r2 > n || r2 <= 0) goto L19;
     prefetch(a[16]); prefetch(b[16]);
     if (4 >= r2) goto L14;
     prefetch(a[20]); prefetch(b[20]);
     f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
     r1 = 8; if (8 >= r2) goto L16;









L16: f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
     f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
     f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
     a += 4; b += 4; f14 = a[8]; f15 = b[8];
     f11 += f22; f1 += f21; dp += f24;
L18: f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
     f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
     a += 4; f28 = f29 * f28; b += 4;
     f10 += f14; f11 += f12; f1 += f26;
     dp += f28;  dp += f1; dp += f10; dp += f11;
     if (r1 >= n) goto L5;
L19: f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
     dp += f18;
     if (r1 < n) goto L19;
L5:  return dp;
L14: f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}
```

Le risque de mécompilation

Mécompilation, n.f. : acte de produire du code exécutable faux à partir d'un programme source correct.

> *We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables.*
>
> *E. Eide & J. Regehr, EMSOFT 2008*

> *To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input.*
>
> *X. Yang, Y. Chen, E. Eide & J. Regehr, PLDI 2011*

La vérification formelle de compilateurs

Les compilateurs sont des programmes complexes, mais ont une spécification ≪de bout en bout≫ assez simple :

> *Le code produit doit s'exécuter comme prescrit par la sémantique du programme source.*

Cette spécification devient mathématiquement précise dès que l'on dispose de sémantiques formelles pour les langages source et machine.

Dès lors on peut envisager une vérification formelle du compilateur.

Une idée ancienne…

John McCarthy
James Painter[1]

CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS[2]

1. **Introduction.** This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

Mathematical Aspects of Computer Science, 1967

Une idée ancienne…

3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

Abstract

We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

Machine Intelligence (7), 1972.

Une idée ancienne…

Même les scripts de preuve sont familiers !

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,|swfse e|:MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e,|swfse e|:|swft(compe e)≡TT,
     ∀e,|swfse e|:(count(compe e)=0)≡TT;

TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfs_efun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE,,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES typ_ e=_E;
     TRY 1 SUBST ,FCOMPE;
     SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
     INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
      TRY 1 CONJ;
       TRY 1 SIMPL;
        TRY 1 USE COUNT1;
         TRY 1;
         APPL ,INDHYP+2,arg1of e;
         LABEL CARG1;
         SIMPL-;QED;
         TRY 2 USE COUNT1;
          TRY 1;
```

Objectif du cours

Dans ce cours, nous allons mener à bien le programme de Milner et Weyrauch : vérifier formellement (en Coq) un compilateur non optimisant pour un langage impératif simple (IMP).

Nous allons dégager des approches qui s'étendent jusqu'à la vérification de compilateurs pour de ≪vrais≫ langages (CompCert, CakeML).

Le cours suivant abordera les optimisations et leur vérification.

La machine virtuelle IMP

Les machines virtuelles

Produire du code machine pour des microprocesseurs existants (x86, ARM, …) est délicat.

De nombreux compilateurs (Java, C#, …) utilisent une machine virtuelle comme étape intermédiaire entre le langage source et le code machine.

Comme les machines réelles, les machines virtuelles exécutent des suites d'instructions simples : pas d'expressions complexes, pas de structures de contrôle.

Les instructions de la machine virtuelle ne sont pas directement exécutables par le matériel, mais sont choisies pour ≪coller≫ aux opérations de base du langage source.

La machine virtuelle IMP

Quatre composantes :

- Le code C : une liste d'instructions.
- Le pointeur de code pc : un entier qui donne la position dans C de l'instruction en cours d'exécution.
- L'état mémoire s : il associe à chaque variable sa valeur
- La pile σ : une liste de valeurs entières
  (sert à stocker temporairement des résultats intermédiaires)

(Inspiré par les anciennes calculatrices HP et par la Java Virtual Machine.)

Le jeu d'instructions

$i ::=$	$\texttt{Iconst}(n)$	empile l'entier n
$\mid$	$\texttt{Ivar}(x)$	empile la valeur de x
$\mid$	$\texttt{Isetvar}(x)$	dépile une valeur et l'affecte à x
$\mid$	$\texttt{Iadd}$	dépile deux valeurs, empile leur somme
$\mid$	$\texttt{Iopp}$	dépile une valeur, empile son opposé
$\mid$	$\texttt{Ibranch}(\delta)$	branchement inconditionnel
$\mid$	$\texttt{Ibeq}(\delta_1, \delta_0)$	dépile deux valeurs, saute δ_1 si $=$, δ_0 si $\neq$
$\mid$	$\texttt{Ible}(\delta_1, \delta_0)$	dépile deux valeurs, saute δ_1 si $\leq$, δ_0 si $>$
$\mid$	$\texttt{Ihalt}$	fin de l'exécution

Toutes les instructions incrémentent *pc* de 1, sauf les instructions de branchement, qui l'incrémentent de $1 + \delta$.

(δ est un déplacement relatif à l'instruction qui suit.)

Exemple

			1		
pile	ε	12	12	13	ε
état	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 12$	$x \mapsto 13$
p.c.	0	1	2	3	4
code	`Ivar(`x`);`	`Iconst(1);`	`Iadd;`	`Isetvar(`x`);`	`Ibranch(`-5`)`

Sémantique de la machine

Définie en style opérationnel par une relation de transition qui représente l'exécution d'une instruction.

L'instruction exécutée est celle qui est dans le code C à la position *pc*.

```
Definition code := list instruction.
Definition stack := list Z.
Definition config : Type := (Z * stack * store)%type.

Inductive transition (C: code): config -> config -> Prop :=
  ...
```

(Voir le fichier Coq `Compil.v`.)

Exécution d'un programme par la machine

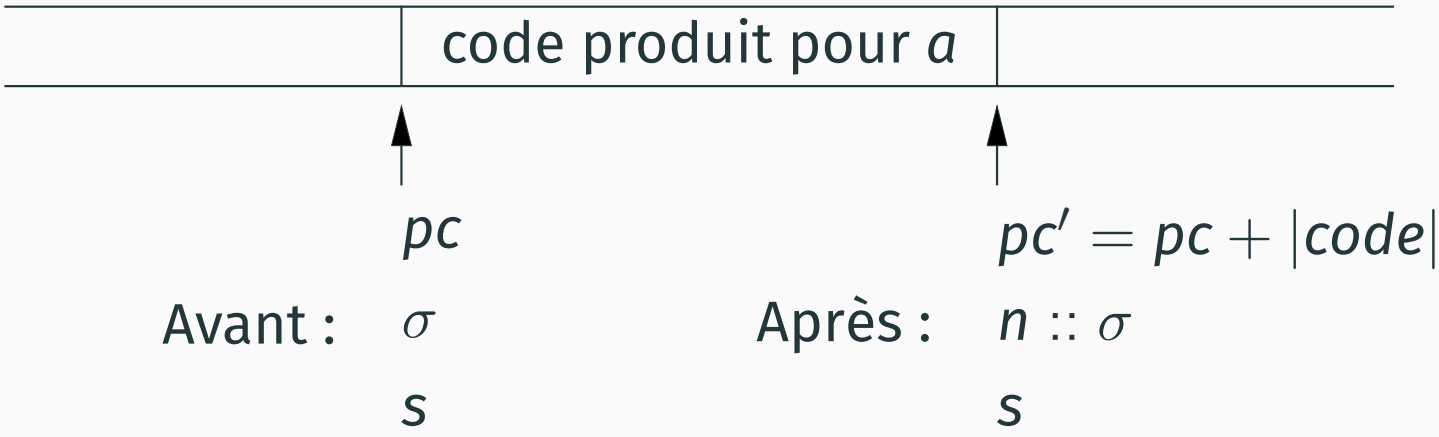
En formant des suites de transitions :

- Configuration initiale :
  $pc = 0$, état mémoire initial, pile vide.
- Configuration finale :
  pc pointe sur une instruction `Ihalt`, pile vide.

Le compilateur

Compilation des expressions arithmétiques

Le contrat : si a s'évalue en la valeur n dans l'état s,

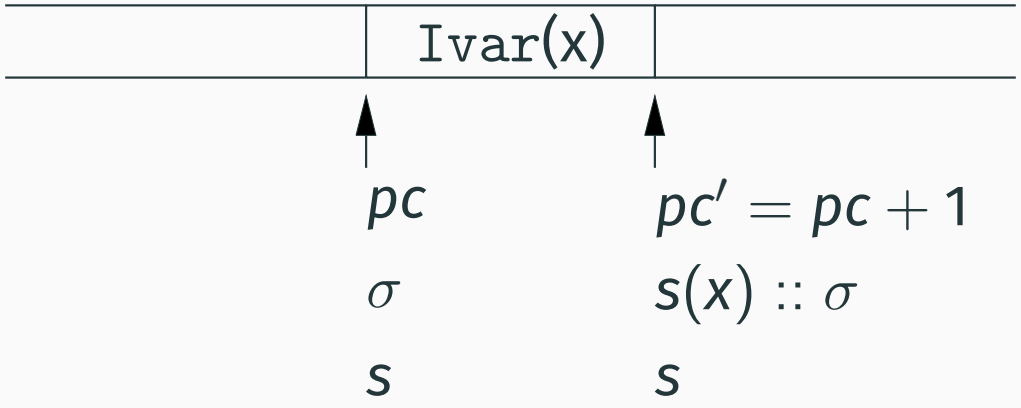


La compilation est la traduction en ≪notation polonaise inverse≫.

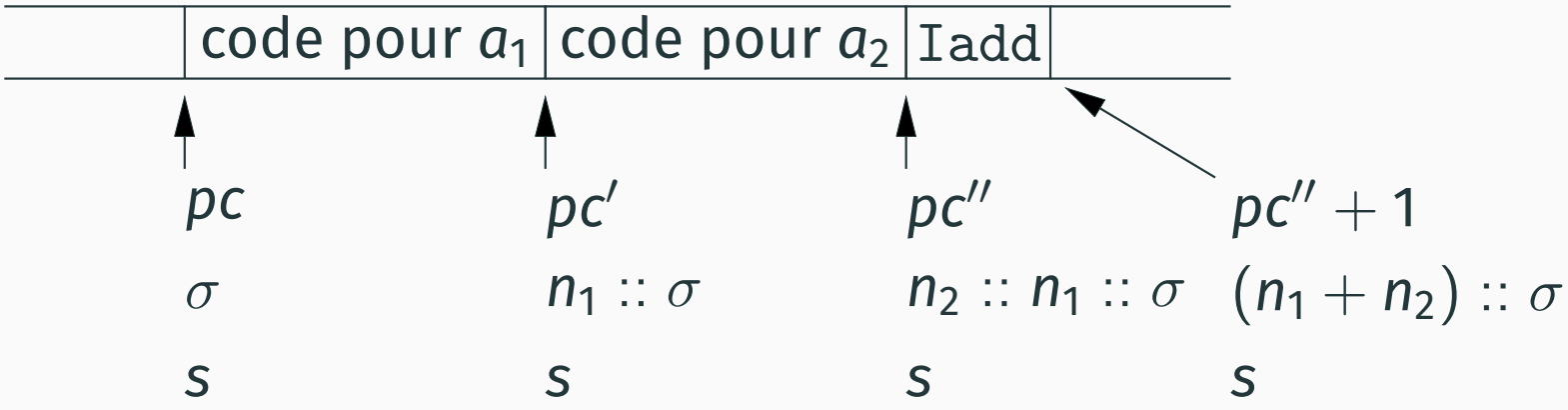
(Fonction Coq : `compile_aexp`)

Compilation des expressions arithmétiques

Un cas de base : si $a = x$,



Un cas de décomposition récursive : si $a = a_1 + a_2$,

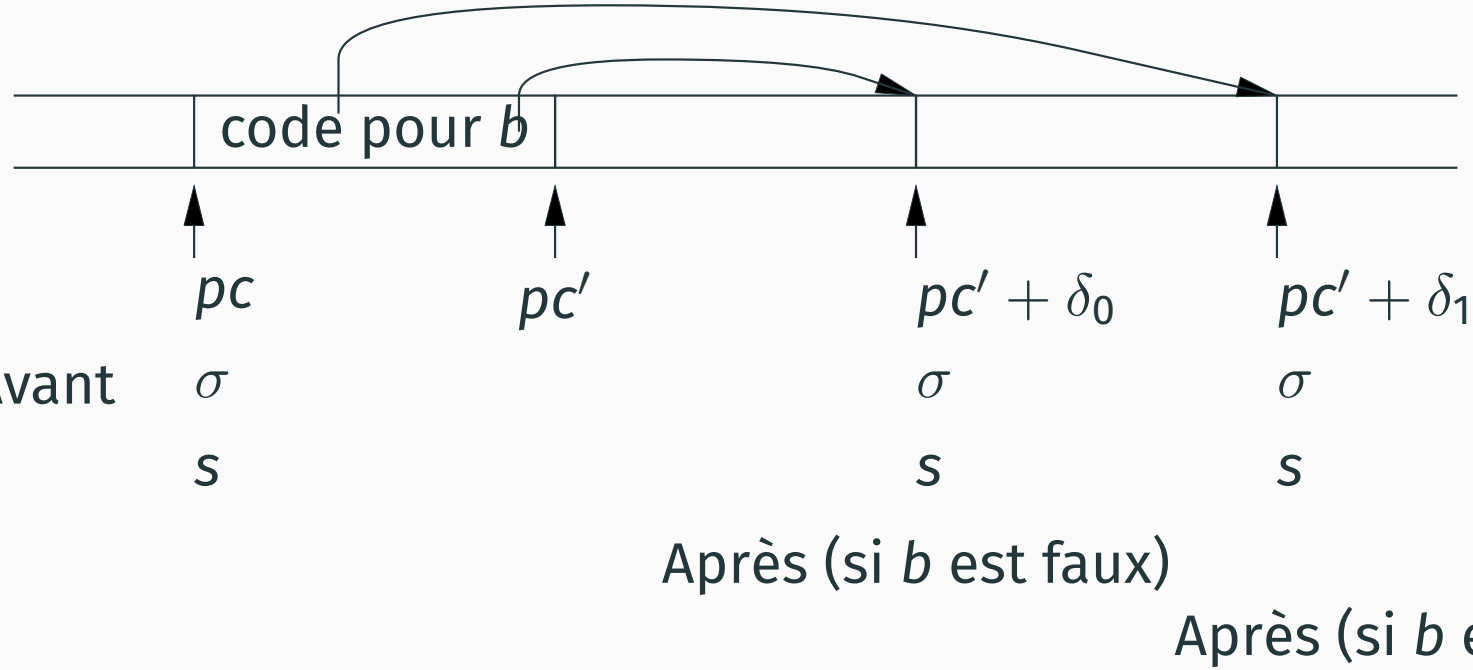


Compilation des expressions booléennes

Contrat : `compile_bexp` b δ_1 δ_0 doit

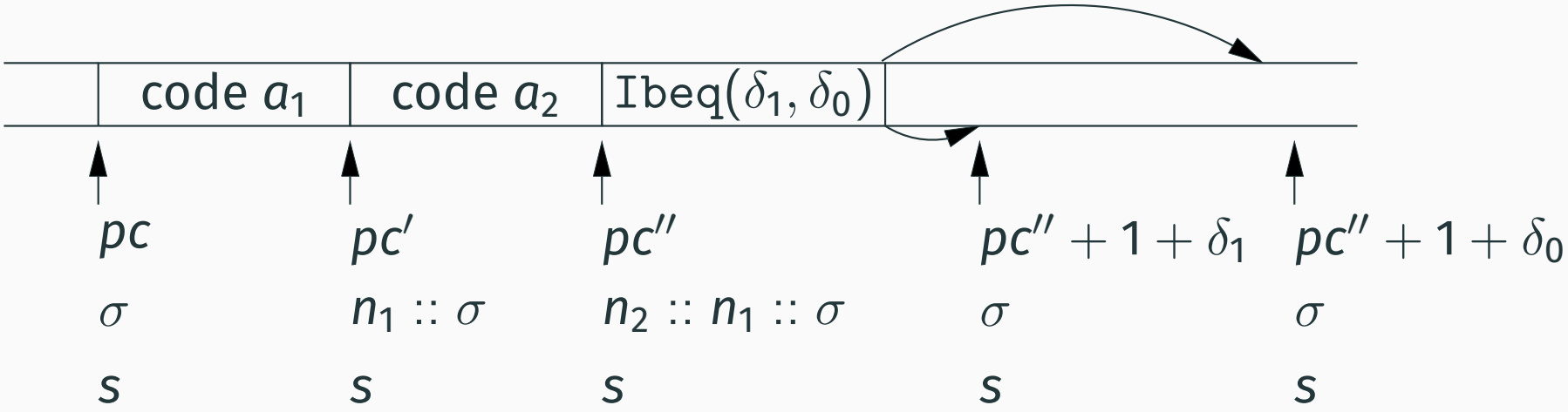
sauter δ_1 instructions si b s'évalue à vrai

sauter δ_0 instructions si b s'évalue à faux.



Compilation des expressions booléennes

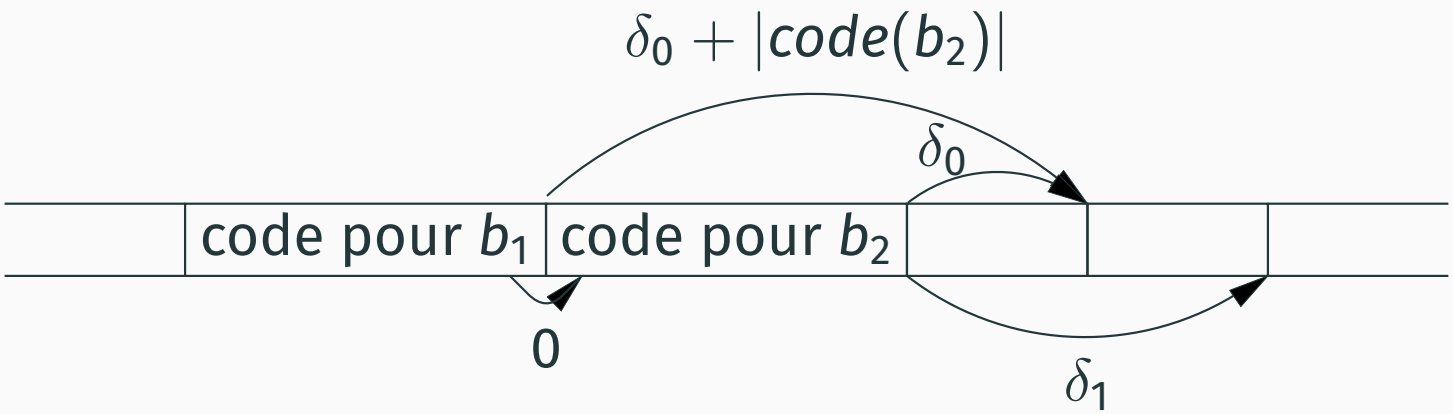
Un cas de base : $b = (a_1 = a_2)$



Court-circuiter les expressions «and»

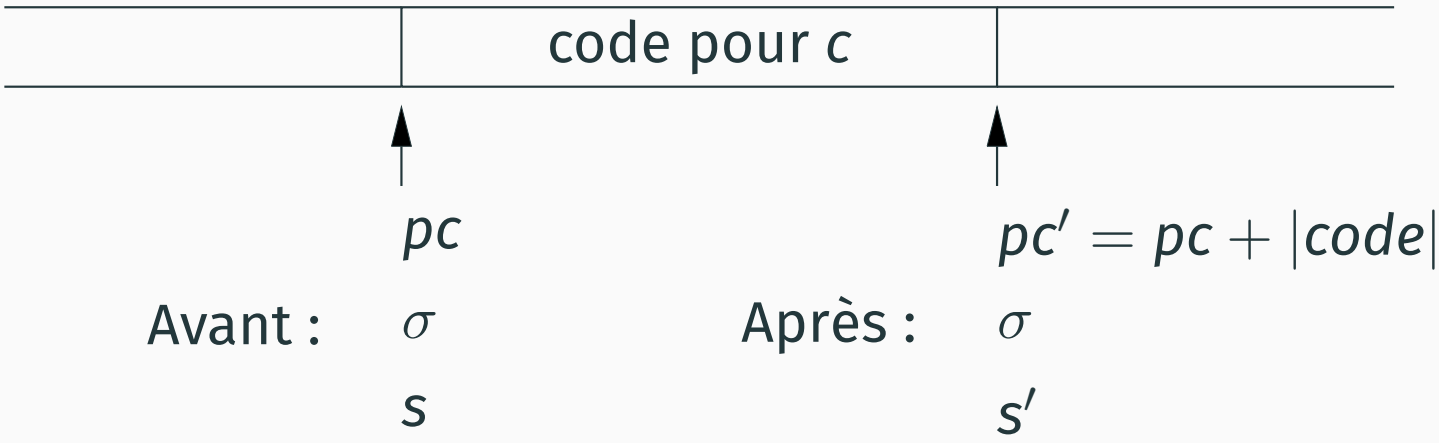
Si b_1 s'évalue à faux, b_1 `and` b_2 s'évalue aussi à faux :
pas besoin d'évaluer b_2 !

Donc, si b_1 est faux, le code compilé pour b_1 `and` b_2 devrait sauter par-dessus le code pour b_2 et se brancher directement à la destination attendue.



Compilation des commandes

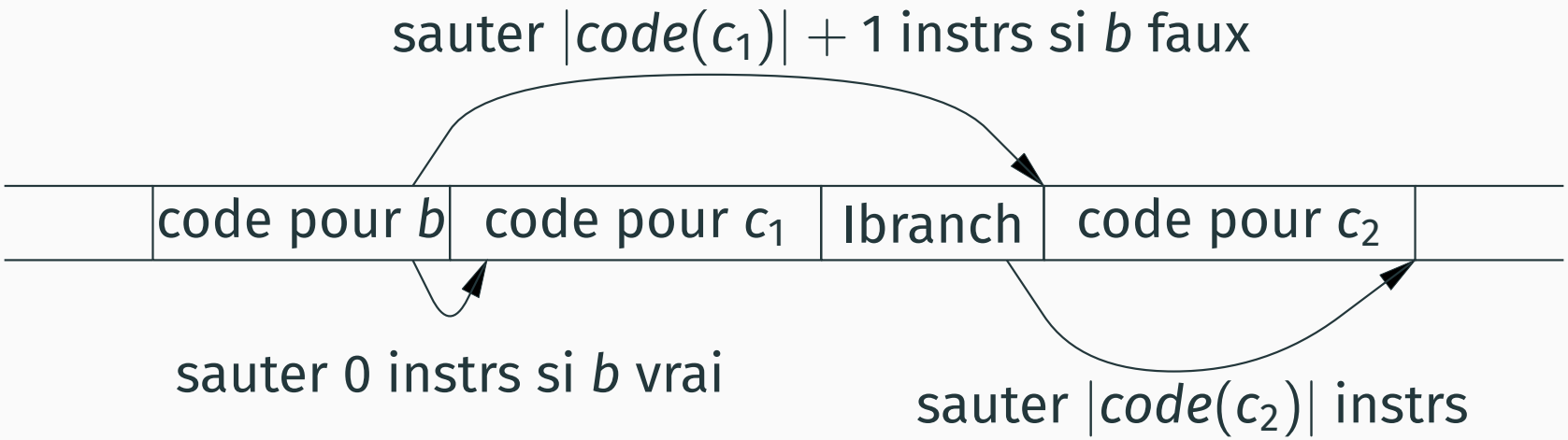
Contrat : si la commande c, démarrée dans l'état mémoire initial s, termine dans l'état final s',



(Fonction Coq : `compile_com`)

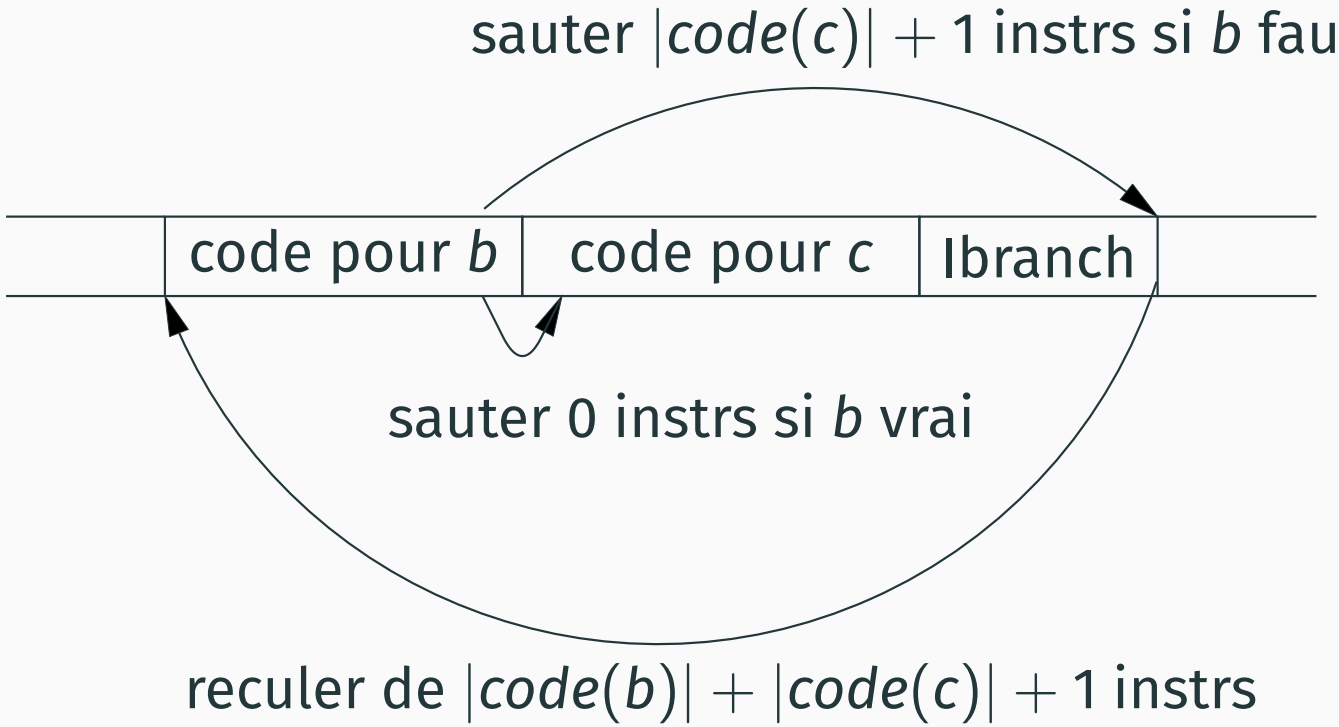
Les mystérieux branchements

Code produit pour `IFTHENELSE` b c_1 c_2 :



Les mystérieux branchements

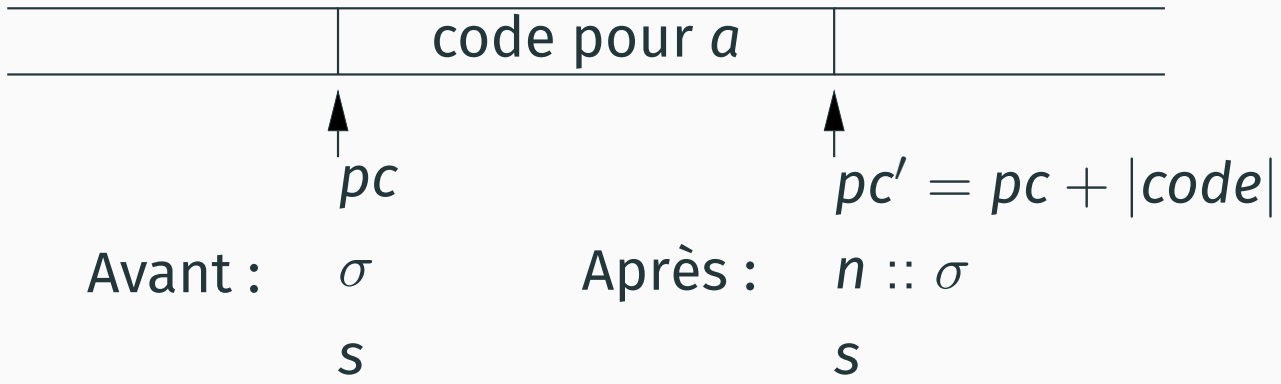
Code produit pour `WHILE` b c :



Premiers résultats de correction

Premières vérifications

Le ≪contrat≫ pour les expressions arithmétiques : si a s'évalue en n dans l'état s,



Un énoncé formel plausible pour ce ≪contrat≫ :

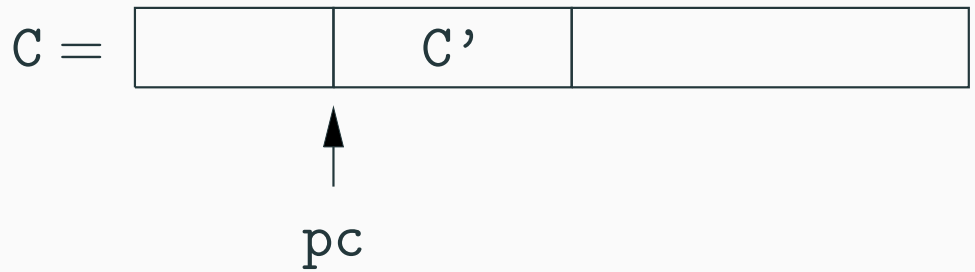
```
Lemma compile_aexp_correct:
  forall s a pc σ,
  transitions (compile_aexp a)
       (0, σ, s)
       (codelen (compile_aexp a), aeval a s :: σ, s).
```

Vérifier la compilation des expressions

Cet énoncé n'est pas démontrable par récurrence sur la structure de a. Il faut le généraliser pour que

- le `pc` initial n'est pas forcément `0`;
- le code `compile_aexp` a apparaît à l'intérieur d'un code C plus grand, à la position `pc`.

Pour ce faire, on définit le prédicat `code_at C pc C'` qui est vrai dans la situation suivante :



Vérifier la compilation des expressions

```
Lemma compile_aexp_correct:
  forall C s a pc σ,
  code_at C pc (compile_aexp a) ->
  transitions C
       (pc, σ, s)
       (pc + codelen (compile_aexp a), aeval a s :: σ, s).
```

Démonstration : une récurrence sur la structure de a.
(C'est la démonstration de McCarthy et Painter, 1967 !)

Les cas de base sont triviaux :

- $a = n$: exécution d'une seule transition `Iconst`
- $a = x$: exécution d'une seule transition `Ivar`(x).

Un cas de récurrence

Prenons $a = a_1 + a_2$ et supposons

$$\texttt{code\_at}\ C\ pc\ (code(a_1)\ \texttt{++}\ code(a_2)\ \texttt{++}\ \texttt{Iadd} :: nil)$$

On construit une suite de transitions :

$$(pc, \sigma, s)$$

$$\downarrow *\quad \text{hyp. de rec. sur } a_1$$

$$(pc + |code(a_1)|, \texttt{aeval}\ a_1\ s :: \sigma, s)$$

$$\downarrow *\quad \text{hyp. de rec. sur } a_2$$

$$(pc + |code(a_1)| + |code(a_2)|, \texttt{aeval}\ a_2\ s :: \texttt{aeval}\ a_1\ s :: \sigma, s)$$

$$\downarrow \quad \text{transition } \texttt{Iadd}$$

$$(pc + |code(a_1)| + |code(a_2)| + 1, (\texttt{aeval}\ a_1\ s + \texttt{aeval}\ a_2\ s) :: \sigma, s)$$

Vérifier la compilation des expressions

Même démarche pour les expressions booléennes : le ≪contrat≫, une fois formalisé et généralisé, est

```
Lemma compile_bexp_correct:
  forall C s b d1 d0 pc σ,
  code_at C pc (compile_bexp b d1 d0) ->
  transitions C
       (pc, σ, s)
       (pc + codelen (compile_bexp b d1 d0)
           + (if beval b s then d1 else d0), σ, s).
```

La démonstration est une récurrence sur la structure de `b`.

Vérifier la compilation des commandes

```
Lemma compile_com_correct_terminating:
  forall s c s',
  cexec s c s' ->
  forall C pc σ,
  code_at C pc (compile_com c) ->
  transitions C
      (pc, σ, s)
      (pc + codelen (compile_com c), σ, s').
```

Une récurrence sur la structure de `c` échoue dans le cas `WHILE`.
Une récurrence sur la dérivation du prédicat `cexec s c s'` fonctionne parfaitement.

Point d'étape

En combinant les résultats précédents et en définissant

```
compile_program c = compile_command c ++ Ihalt :: nil
```

on obtient un joli théorème :

```
Theorem compile_program_correct_terminating:
  forall s c s',
  cexec s c s' ->
  machine_terminates (compile_program c) s s'.
```

Cela suffit-il à conclure que notre compilateur est correct ?

Avons-nous pensé à tout ?

```
Theorem compile_program_correct_terminating:
  forall s c s',
  cexec s c s' ->
  machine_terminates (compile_program c) s s'.
```

Et si le code machine engendré s'arrête sur un état différent de `s'` ? ou qu'il boucle ? ou qu'il bloque en erreur ?

Impossible ! parce que la machine est déterministe : chaque code machine a au plus un comportement (terminer sur s, diverger, ou ≪planter≫).

Avons-nous pensé à tout ?

```
Theorem compile_program_correct_terminating:
  forall s c s',
  cexec s c s' ->
  machine_terminates (compile_program c) s s'.
```

Et si le programme `c` démarré dans l'état `s` diverge au lieu de terminer ? Que fait le code machine engendré dans ce cas ?

Exemple

On «optimise» `while true do skip` en `skip`.
C'est douteux ; pourtant, le théorème n'est pas invalidé !

Il nous faut une vérification plus poussée qui montre également la préservation de la non-terminaison.

Diagrammes de simulation

Sémantiques à transitions

Définies par une relation $a \to a'$ de transition ou de réduction.

Aussi appelées sémantiques opérationnelles à ≪petits pas≫.

Exemples :

- la sémantique à réduction d'IMP
- la sémantique à transitions de la machine virtuelle
- le lambda-calcul $M \to_\beta M'$
- les calculs de processus $P \xrightarrow{\alpha} P'$

Sémantiques à transitions

Les sémantiques à transition définissent les comportements possibles d'un programme en termes de suites de transitions :

- Terminaison : suite finie de transitions vers une configuration finale.

$$a \to a_1 \to \cdots \to a_n \in \mathit{Fin}$$

- Divergence : suite infinie de transitions

$$a \to a_1 \to \cdots \to a_n \to \cdots$$

- Partir en erreur : suite finie de transitions vers une configuration qui est bloquée et n'est pas finale.

$$a \to a_1 \to \cdots \to a_n \not\to \text{ avec } a_n \notin \mathit{Fin}$$

Diagrammes de simulation

Supposons que le programme source S et le code compilé C ont des sémantiques à transition.

Montrons que chaque transition dans l'exécution de S

- est simulée par des transition dans l'exécution de C
- tout en préservant une relation entre les configurations de S et celles de C.

Diagrammes de simulation synchrones

Chaque transition du source est simulée par exactement une transition du code compilé.

(En noir : les hypothèses ; en rouge : les conclusions.)

Diagrammes de simulation synchrones

On montre que les configurations initiales sont reliées :

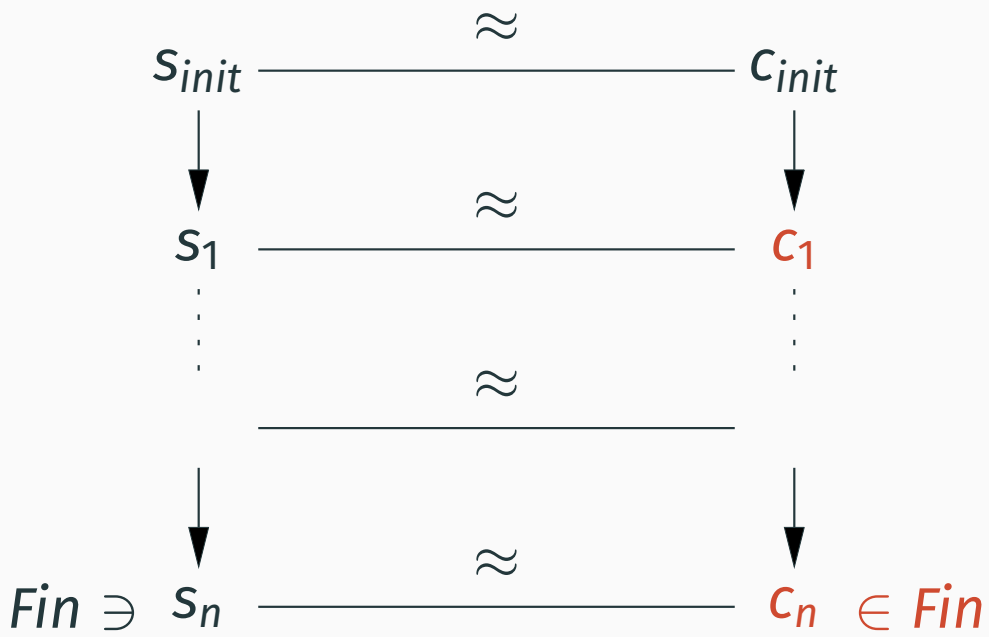
$$s_{init} \approx c_{init}$$

On montre que les configurations finales sont reliées :

$$s \approx c \ \wedge \ s \in \mathit{Fin} \implies c \in \mathit{Fin}$$

Diagrammes de simulation synchrones

Il s'ensuit que si S termine, C termine aussi :

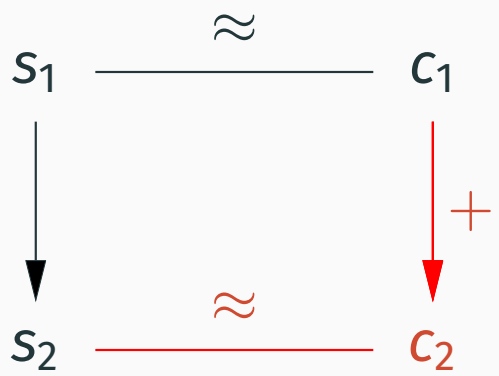


De même, si on a une infinité de transitions issues de s_{init}, on a une infinité de transitions issues de c_{init}. Donc, si S diverge, C diverge aussi.

Diagrammes de simulation «plus»

Parfois, chaque transition dans le programme source est simulée par une ou plusieurs transitions dans le code compilé.

(Exemple : le code engendré pour $x := a$ contient plusieurs instructions machine.)

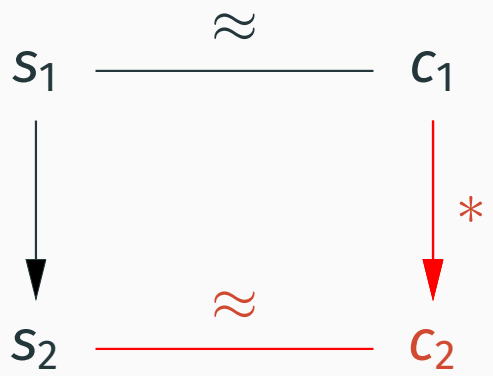


Là encore, la terminaison et la divergence sont préservées.

Diagrammes de simulation «étoile» (incorrect!)

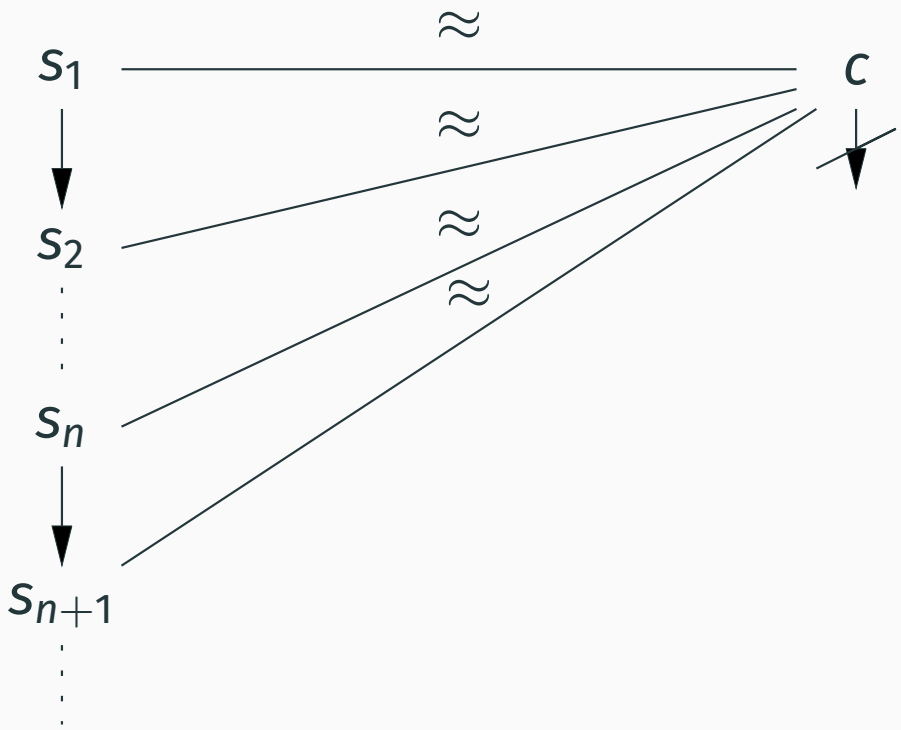
Parfois, chaque transition dans le programme source est simulée par zéro, une ou plusieurs transitions dans le code compilé.

Exemple : la réduction $(\texttt{SKIP}; c)/s \to c/s$ correspond à zéro transitions de la machine. On dit qu'il y a «bégaiement».



Les exécutions qui terminent sont préservées ;
les exécutions qui divergent ne le sont pas nécessairement !

Le problème du bégaiement infini

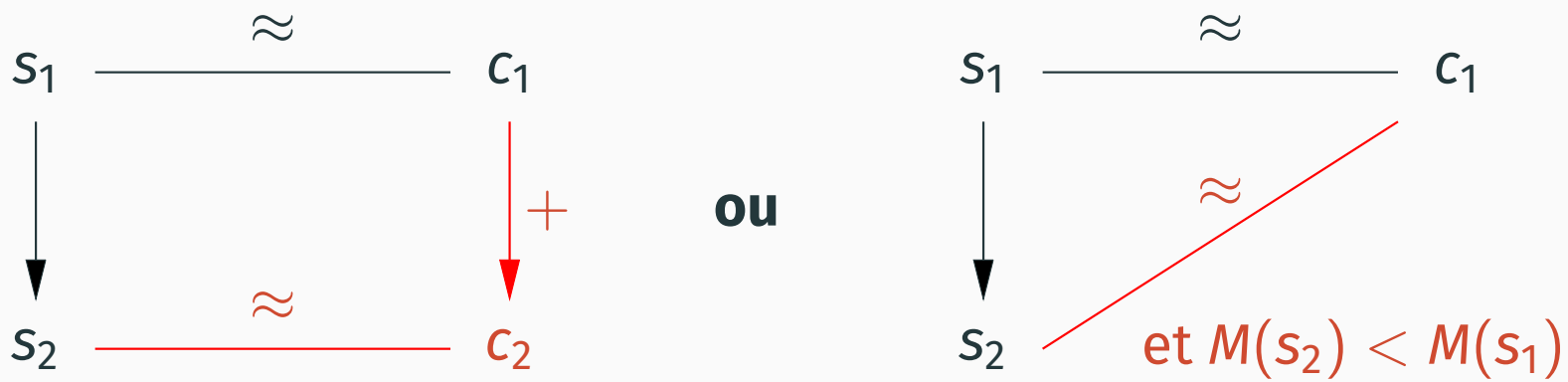


Ici, le programme source diverge mais le code compilé peut terminer, normalement ou en erreur.

C'est le signe d'une optimisation incorrecte de programmes qui divergent, p.ex. «optimiser» `while true do skip` en `skip`.

Diagrammes de simulation «étoile» (corrigé)

Trouver une mesure $M(s)$: `nat` pour les configurations source qui décroît strictement lors des étapes de bégaiement.



Si s termine, c termine aussi (comme précédemment).

Si s diverge, il doit effectuer une infinité d'étapes non-bégayantes, donc c effectue une infinité de transitions.

(Remarque : on peut utiliser tout ordre bien fondé sur s).

Vers une simulation pour la compilation de IMP

Essayons de démontrer un diagramme de simulation entre une commande IMP et son code machine compilé.

La sémantique à réductions d'IMP soulève deux difficultés :

- comment relier la commande IMP et le code machine ;
- trouver la mesure qui évite le bégaiement infini.

≪Génération spontanée≫ de commandes

En sémantique naturelle, dans la dérivation de $c/s \Downarrow s'$, toutes les commandes qui apparaissent sont des sous-termes de c.

Ce n'est pas le cas en sémantique à réductions : des commandes apparaissent lors de la réduction qui ne sont pas des sous-termes du programme c initial :

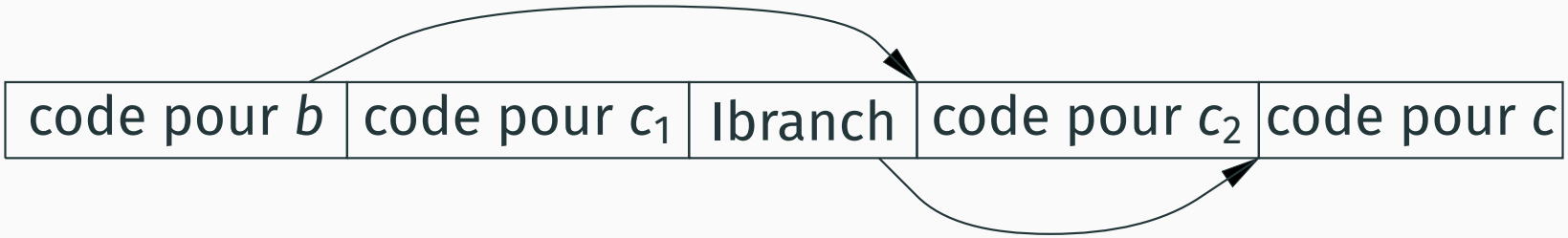
$$(\texttt{while}\ b\ \texttt{do}\ c)/s \to (c; \texttt{while}\ b\ \texttt{do}\ c)/s \quad \text{si } [\![s]\!]\, b = \texttt{true}$$

$$((\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c)/s \to (c_1; c)/s \quad \text{si } [\![b]\!]\, s = \texttt{true}$$

Retrouver les commandes dans le code compilé

Le code compilé pour le programme initial ne change pas pendant l'exécution. On n'y retrouve pas le code pour les commandes ≪spontanément engendrées≫ pendant les réductions.

Code compilé pour $(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2); c$:



Ce code ne contient pas le code compilé pour $c_1; c$, à savoir :

La mesure anti-bégaiement

Les étapes de réduction qui ≪bégayent≫, c.à.d. correspondent à zéro transitions de la machine, sont, entre autres,

$$(\texttt{skip}; c)/s \to c/s$$

$$(\texttt{if true then } c_1 \texttt{ else } c_2)/s \to c_1/s$$

$$(\texttt{while true do } c)/s \to (c; \texttt{while true do } c)/s$$

Il faut donc trouver une mesure M telle que

$$M(\texttt{skip}; c) > M(c)$$

$$M(\texttt{while true do } c) > M(c; \texttt{while true do } c)$$

C'est impossible ! Regardez $M(\texttt{while true do skip})$…

Marquage des séquences

On peut contourner le problème en marquant spécialement $;^\dagger$ les séquences produites par la réduction des boucles :

$$
\begin{aligned}
(\texttt{while}\ b\ \texttt{do}\ c)/s &\to \texttt{skip}/s && \text{si } [\![b]\!]\, s = \texttt{false} \\
(\texttt{while}\ b\ \texttt{do}\ c)/s &\to (c;^\dagger\ \texttt{while}\ b\ \texttt{do}\ c)/s && \text{si } [\![s]\!]\, b = \texttt{true} \\
(c_1;^\dagger\ c_2)/s &\to (c_1';^\dagger\ c_2)/s' && \text{si } c_1/s \to c_1'/s' \\
(\texttt{skip};^\dagger\ c_2)/s &\to c_2/s
\end{aligned}
$$

La réduction $(\texttt{skip};^\dagger\ c_2)/s \to c_2/s$ n'est pas bégayante, car elle correspond à l'instruction `Ibranch` qui redémarre la boucle. On peut donc prendre $M(c_1;^\dagger\ c_2) = M(c_1)$ et satisfaire les contraintes sur M.

Un changement d'approche

Signal d'alarme : on se prépare à modifier la syntaxe du langage juste pour ≪faire passer≫ une démonstration d'un compilateur…

Approche plus saine : sans changer la syntaxe d'IMP, trouvons une autre sémantique :

- de type opérationnel ≪à petits pas≫, afin de pouvoir raisonner par diagrammes de simulation ;
- qui ne pose pas de problèmes de ≪génération spontanée≫ de commandes, ni de contrôle du bégaiement.

La sémantique à continuations

Une sémantique à petits pas avec des continuations

Au lieu de réécrire des programmes complets c

$$c/s \to c'/s'$$

on va réécrire des commandes en cours d'examen c et leurs continuations k :

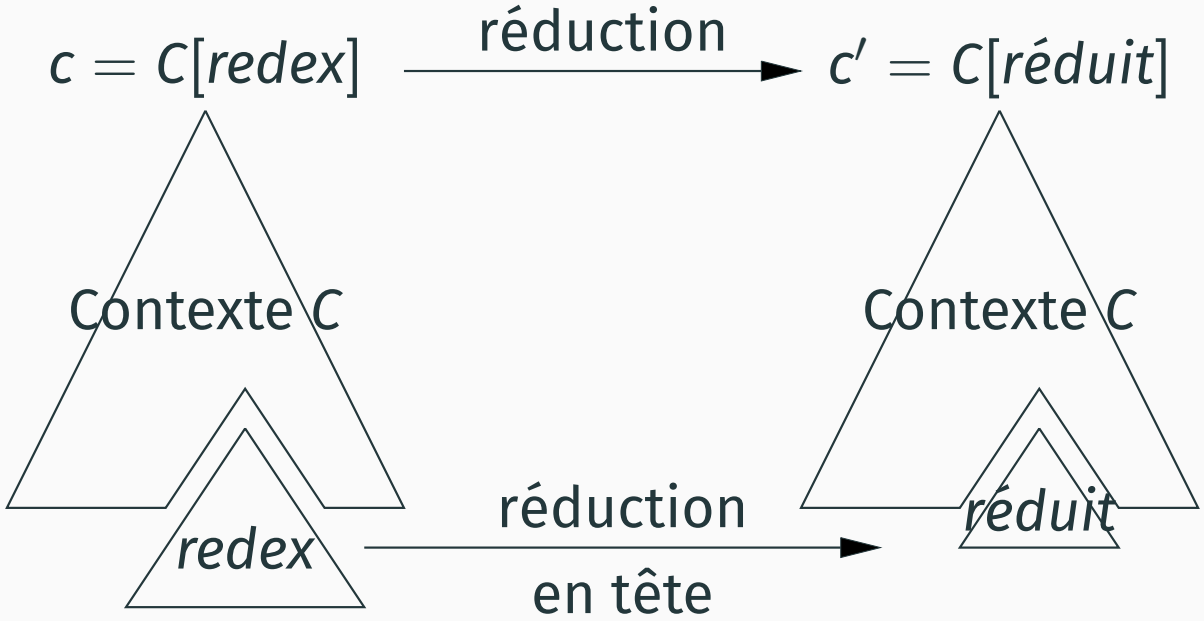
$$c/k/s \to c'/k'/s'$$

(Idée reprise de A. W. Appel et S. Blazy, *Separation Logic for Small-Step Cminor*, 2007.)

(Proche de la focalisation en théorie de la démonstration.)

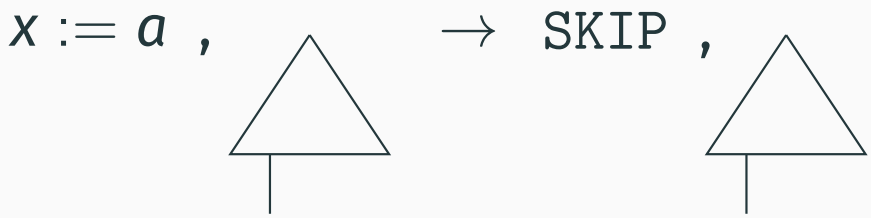
La sémantique à réduction habituelle

Réécrit le programme entier, même si une seule sous-commande change (le *redex*).

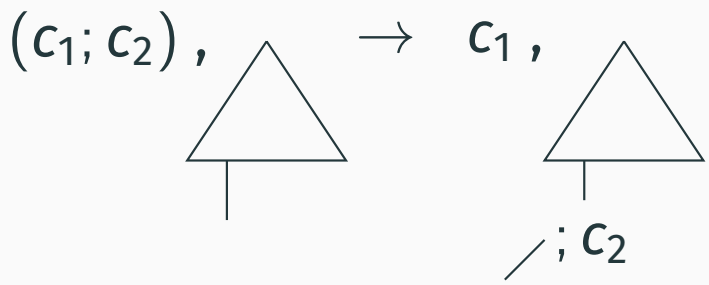


Focaliser la sémantique à réductions

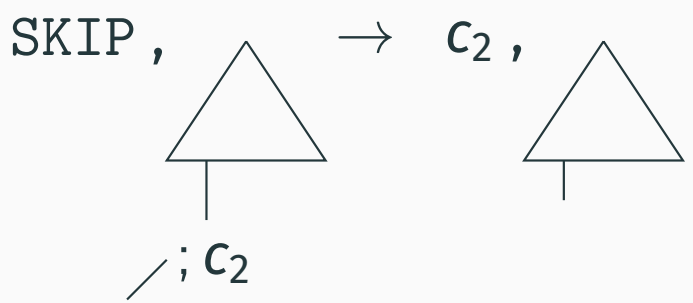
Réécrire des paires (sous-commande, contexte où elle apparaît).



La sous-commande n'est pas toujours le redex ! On ajoute des règles explicites de focalisation et de reprise pour déplacer des termes entre sous-commande et contexte.



Focalisation à gauche d'une séquence

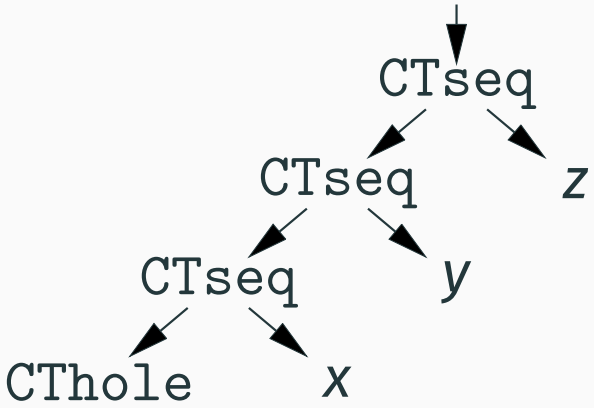


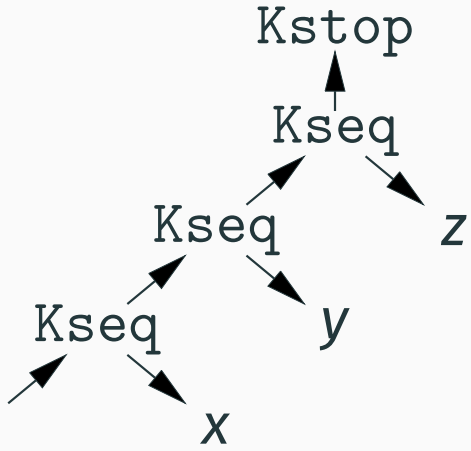
Reprise d'une séquence

Représenter les contextes «la tête en bas»

```
Inductive ctx :=
  | CThole: ctx
  | CTseq: com -> ctx -> ctx.
```

```
Inductive cont :=
  | Kstop: cont
  | Kseq: com -> cont -> cont.
```





`CTseq` (`CTseq` (`CTseq` `CThole` *x*) *y*) *z*

`Kseq` *x* (`Kseq` *y* (`Kseq` *z* `Kstop`))

Contexte la tête en bas ≈ continuation.

(«Ensuite, faire *x*, puis faire *y*, puis faire *z*, puis s'arrêter.»)

Règles de transition

$$
\begin{array}{rcll}
x := a/k/s & \to & \texttt{skip}/k/s[x \leftarrow \texttt{aeval}\ a\ s] & \\
(c_1; c_2)/k/s & \to & c_1/\texttt{Kseq}\ c_2\ k/s & \\
\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2/k/s & \to & c_1/k/s & \text{si } \texttt{beval}\ \ b\ s = \texttt{true} \\
\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2/k/s & \to & c_2/k/s & \text{si } \texttt{beval}\ \ b\ s = \texttt{false} \\
\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}/k/s & \to & c/\texttt{Kwhile}\ b\ c\ k/s & \\
 & & \text{si } \texttt{beval}\ \ b\ s = \texttt{true} & \\
\texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}/k/s & \to & \texttt{skip}/c/k & \text{si } \texttt{beval}\ \ b\ s = \texttt{false} \\
\texttt{skip}/\texttt{Kseq}\ c\ k/s & \to & c/k/s & \\
\texttt{skip}/\texttt{Kwhile}\ b\ c\ k/s & \to & \texttt{while}\ b\ \texttt{do}\ c\ \texttt{done}/k/s &
\end{array}
$$

Note : pas de génération spontanée de commandes !

Correction complète du compilateur

Un diagramme de simulation

Nous allons enfin construire un diagramme de simulation des transitions de la sémantique à continuations d'IMP par des transitions de la machine.

Cela montrera la préservation des évaluations qui terminent (déjà démontré) et des évaluations qui divergent (nouveau !).

La machine étant déterministe, on obtient la bisimulation entre le programme source et son code compilé.

Deux points délicats :

1. Empêcher le bégaiement infini
2. Relier la commande-continuation courante c, k avec le code compilé C (qui est fixé pendant l'exécution).

La mesure anti-bégaiement

Les étapes de réduction qui bégaient sont, principalement :

$$(c_1; c_2)/k/s \to c_1/\texttt{Kseq}\ c_2\ k/s$$
$$\texttt{skip}/\texttt{Kseq}\ c\ k/s \to c/k/s$$
$$(\texttt{if true then}\ c_1\ \texttt{else}\ c_2)/k/s \to c_1/k/s$$
$$(\texttt{while true do}\ c)/k/s \to c/\texttt{Kwhile true}\ c\ k/s$$

Comme précédemment, mesurer c ne mène à rien. Il faut mesurer des paires (c, k).

La mesure anti-bégaiement

Après essais et erreurs, la mesure suivante fonctionne :

$$M(c, k) = \|c\| + M(k)$$

avec

$$M(\texttt{Kskip}) = 0 \quad M(\texttt{Kseq}\, c\, k) = \|c\| + M(k) \quad M(\texttt{Kwhile}\, b\, c\, k) = M(k)$$

En effet, elle vérifie

$$\begin{aligned} M((c_1; c_2), k) &= M(c_1, \texttt{Kseq}\ c_2\ k) + 1 \\ M(\texttt{SKIP}, \texttt{Kseq}\ c\ k) &= M(c, k) + 1 \\ M(\texttt{IFTHENELSE}\ b\ c_1\ c_2, k) &> M(c_1, k) \\ M(\texttt{WHILE}\ b\ c, k) &= M(c, \texttt{Kwhile}\ b\ c\ k) + 1 \end{aligned}$$

Relier commande, continuation, et code compilé

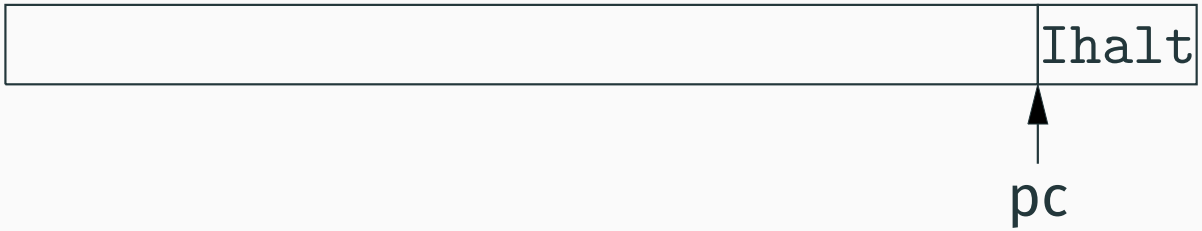
Pour une commande c : `code_at C pc (compile_com c).`

Pour une paire (commande c, continuation k) :

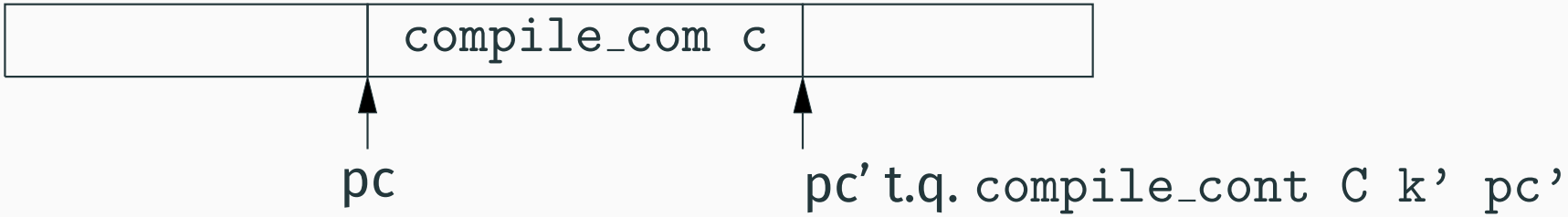
Relier continuation et code compilé

Un prédicat `compile_cont` C k pc, signifiant «il existe un chemin dans le code c qui part de pc, finit sur une instruction `Ihalt`, et exécute les calculs en attente décrits par k».

Cas de base $k = $ `Kstop` :

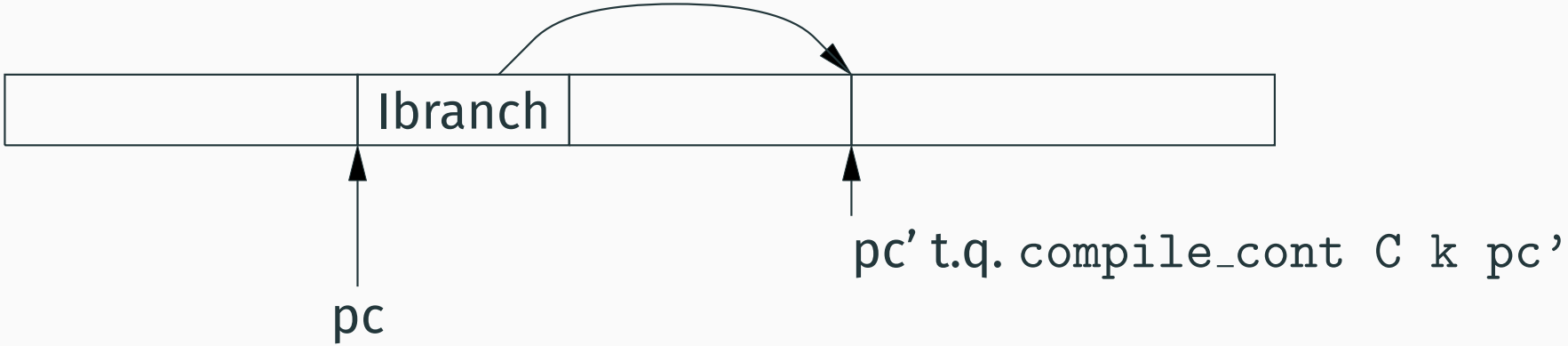


Cas de la séquence $k = $ `Kseq` c k' :

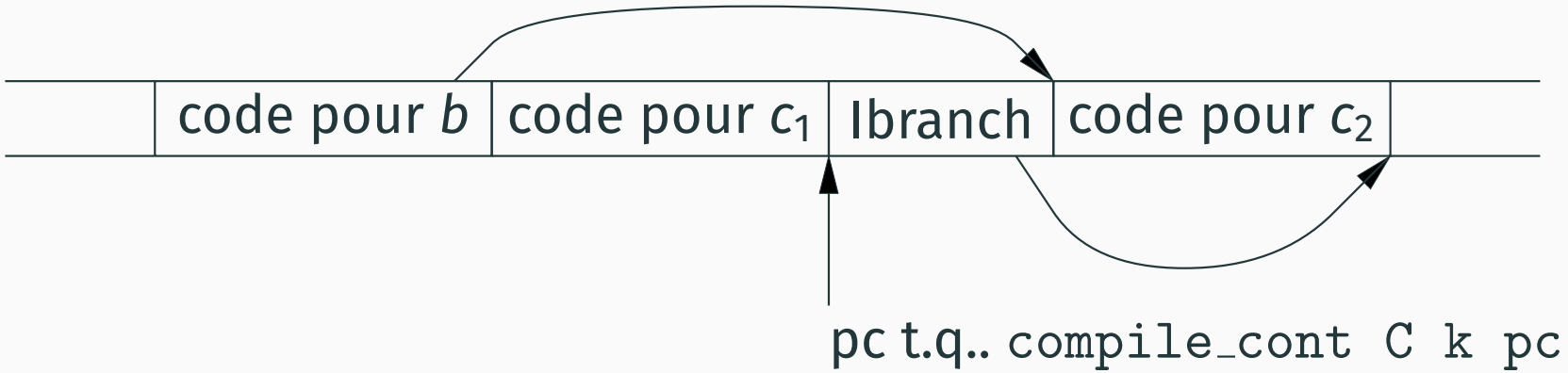


Relier continuation et code compilé

Un cas «non structurel» nous permet d'insérer des branchements à volonté :



Permet de gérer les continuation produites par `if` b `then` c_1 `else` c_2 :

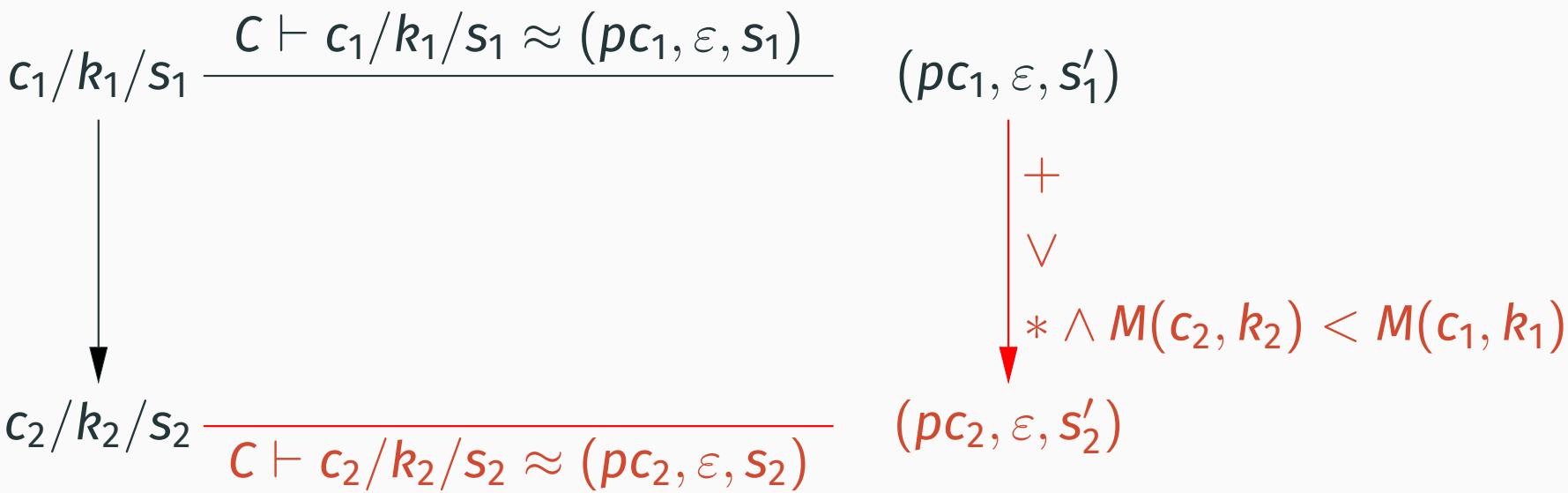


La relation de simulation

Une configuration du programme source (c, k, s) est reliée à une configuration de la machine $C, (pc, \sigma, s')$ ssi :

- les états mémoire sont identiques : $s' = s$
- la pile est vide : $\sigma = \varepsilon$
- C contient le code compilé pour c à la position initiale pc
- C contient du code correspondant à k à partir de la position $pc + |code(c)|$.

Le diagramme de simulation



Démonstration : grosse analyse de cas sur la transition de gauche.

En conclusion

En conséquence de ce diagramme, nous obtenons :

- Une autre démonstration de la correction du compilateur pour les programmes qui terminent :
  si $c/\texttt{Kstop}/s \xrightarrow{*} \texttt{SKIP}/\texttt{Kstop}/s'$
  alors $\texttt{machine\_terminates}\ (\texttt{compile\_program}\ c)\ s\ s'$
- Une démonstration de la correction du compilateur pour les programmes qui divergent :
  si $c/\texttt{Kstop}/s$ se réduit infiniment,
  alors $\texttt{machine\_diverges}\ (\texttt{compile\_program}\ c)\ s$

Mission accomplie !

Point d’étape

Point d'étape

Nous avons illustré sur un compilateur non optimisant pour un petit langage quelques approches qui ≪passent à l'échelle≫ de compilateurs vérifiés plus ambitieux, notamment CompCert :

- La génération de code par récursion sur la syntaxe abstraite.
- Les sémantiques naturelles pour les premières explorations.
- Les diagrammes de simulation entre deux sémantiques à transition pour la démonstration finale.
- Les sémantiques à continuation pour les langages à contrôle structuré.

Bibliographie

Bibliographie

Un livre de classe sur la compilation :

- A. W. Appel, *Modern Compiler Implementation in Java / ML / C*, Cambridge University Press, 1998.

Une autre vérification d'un compilateur pour IMP :

- T. Nipkow, G. Klein, *Concrete Semantics*, Springer, 2014, chapitre 8.

Deux vérifications formelles de compilateurs réalistes :

- X. Leroy, *Formal verification of a realistic compiler*, Comm. ACM 52(7), 2009.
- R. Kumar, M. O. Myreen, M. Norrish, S. Owens, *CakeML : A Verified Implementation of ML*, POPL 2014.