Types dépendants : tout un programme !

Pierre-Évariste Dagand – Sorbonne Université - CNRS - Inria

Ce document récapitule les constructions présentées lors du séminaire du 28 Novembre 2018. On les trouvera par ailleurs sous la forme d'un dépôt Git[1].

[1] https://github.com/pedagand/compilateur-CdF/commits/master

Version simplement typée

Dans la version suivante, nous écrivons une version simplement typée du compilateur, telle qu'on aurait pu l'écrire en OCaml ou en Haskell. On découvre ainsi l'environnement de programmation dans un contexte familier.

```
open import Lib

-- ================================================================
-- Types dépendants : tout un programme !
--
-- Séminaire du 28 Nov. 2018
-- Pierre-Évariste Dagand -- Sorbonne Université - CNRS - Inria
-- https://bit.ly/2QqmiNC
-- https://www.lip6.fr/Pierre-Evariste.Dagand/stuffs/college-2018/seminaire.zip
-- ================================================================

-- Inspiré de
--     "Correctness of a compiler for arithmetic expressions"
--         McCarthy, J. & Painter, J.A. (1967)

-- "This paper contains a proof of the correctness of a simple
-- compiling algorithm for compiling arithmetic expressions into
-- machine language."

-- "Note that there are no jump instructions. Needless to say, this is
-- a severe restriction on the generality of our results which we
-- shall overcome in future work."

-- --------------------------------
-- Expressions
-- --------------------------------

{-
  v ::=                     (valeurs)
      | true | false        (booléens)
      | 0 | 1 | ...         (entiers naturel)
-}

data valeur : Set where
  estNat  : (n : ℕ)    → valeur
  estBool : (b : Bool) → valeur

{-
  b, e₁, e₂ ::=             (expressions)
              | v           (valeur)
              | plus e₁ e₂  (addition)
              | ifte b e₁ e₂ (conditionnelle)
-}

data exp : Set where
  val  : (v : valeur)                  → exp
  plus : (e₁ : exp)(e₂ : exp)          → exp
  ifte : (b : exp)(e₁ : exp)(e₂ : exp) → exp

ex₁ : exp
ex₁ = plus (val (estNat 1)) (val (estNat 1))

ex₂ : exp
ex₂ = plus (val (estNat 2))
           (ifte (val (estBool true))
                 (val (estNat 3))
                 (plus (val (estNat 1)) (val (estNat 7))))

-- --------------------------------
-- Machine à pile
-- --------------------------------

{-
  π ::=                     (pile)
      | ε                   (pile vide)
      | v • π               (valeur sur pile)
-}

data pile : Set where
  ε   :                          pile
  _•_ : (v : valeur)(π : pile) → pile
```

```
infixr 5 _•_

{-
  c₁, c₂ ::=                (machine à pile)
           | SKIP
           | PUSH v         (valeur sur pile)
           | ADD            (addition sur pile)
           | IFTE c₁ c₂     (conditionnelle sur pile)
           | c₁ # c₂        (séquence)
-}

data code : Set where
  SKIP :                          code
  PUSH : (v : valeur)           → code
  ADD  :                          code
  IFTE : (c₁ : code)(c₂ : code) → code
  _#_  : (c₁ : code)(c₂ : code) → code

infixr 20 _#_

ex₃ : code
ex₃ = PUSH (estNat 1) #
      PUSH (estNat 1) #
      ADD

ex₄ : code
ex₄ = PUSH (estNat 2) #
      PUSH (estBool true) #
      IFTE (PUSH (estNat 3))
           (PUSH (estNat 1) # PUSH (estNat 7) # ADD) #
      ADD

-- --------------------------------
-- Compilateur
-- --------------------------------

compile : exp → code
compile (val v)        = PUSH v
compile (plus e₁ e₂)   = compile e₂ #
                         compile e₁ #
                         ADD
compile (ifte b e₁ e₂) = compile b #
                         IFTE (compile e₁)
                              (compile e₂)
```

Sémantique & manque de dépendance

On souhaite prouver la correction de la première version : il faut donc définir la *sémantique* des langages source et cible. La version suivante introduit la fonction `semE` qui calcule la dénotation des expressions dans Agda. Cette fonction s'avère être trop compliquée à écrire.

```
open import Lib

-- ================================================================
-- Types dépendants : tout un programme !
--
-- Séminaire du 28 Nov. 2018
-- Pierre-Évariste Dagand -- Sorbonne Université - CNRS - Inria
-- https://bit.ly/2QqmiNC
-- https://www.lip6.fr/Pierre-Evariste.Dagand/stuffs/college-2018/seminaire.zip
-- ================================================================

-- Inspiré de
--     "Correctness of a compiler for arithmetic expressions"
--         McCarthy, J. & Painter, J.A. (1967)

-- "This paper contains a proof of the correctness of a simple
-- compiling algorithm for compiling arithmetic expressions into
-- machine language."

-- "Note that there are no jump instructions. Needless to say, this is
-- a severe restriction on the generality of our results which we
-- shall overcome in future work."

-- --------------------------------
-- Expressions
-- --------------------------------

{-
  v ::=                     (valeurs)
      | true | false        (booléens)
      | 0 | 1 | ...         (entiers naturel)
-}

data valeur : Set where
  estNat  : (n : ℕ)    → valeur
  estBool : (b : Bool) → valeur

{-
  b, e₁, e₂ ::=             (expressions)
              | v           (valeur)
              | plus e₁ e₂  (addition)
              | ifte b e₁ e₂ (conditionnelle)
-}

data exp : Set where
  val  : (v : valeur)                  → exp
  plus : (e₁ : exp)(e₂ : exp)          → exp
  ifte : (b : exp)(e₁ : exp)(e₂ : exp) → exp

ex₁ : exp
ex₁ = plus (val (estNat 1)) (val (estNat 1))

ex₂ : exp
ex₂ = plus (val (estNat 2))
           (ifte (val (estBool true))
                 (val (estNat 3))
                 (plus (val (estNat 1)) (val (estNat 7))))

semE : exp → Option valeur
semE (val v)                          = OK v
semE (plus e₁ e₂)
  with semE e₁ | semE e₂
... | OK (estNat n₁) | OK (estNat n₂) = OK (estNat (n₁ + n₂))
... | OK (estBool b) | _              = KO
... | OK (estNat n₁) | OK (estBool b) = KO
... | OK v           | KO             = KO
... | KO             | _              = KO
semE (ifte b e₁ e₂)
  with semE b | semE e₁ | semE e₂
... | OK (estBool v) | v₁ | v₂        = if v then v₁ else v₂
... | _ | _ | _                      = KO
```

```
test-ex₁ : semE ex₁ ≡ OK (estNat 2)
test-ex₁ = refl

-- "The above proposition is occasionally useful."
--         Whitehead & Russell (PM, vol.II, p.362)

test-ex₂ : semE ex₂ ≡ OK (estNat 5)
test-ex₂ = refl

-- --------------------------------
-- Machine à pile
-- --------------------------------

{-
  π ::=                       (pile)
      | ε                     (pile vide)
      | v • π                 (valeur sur pile)
-}

data pile : Set where
  ε   :                             pile
  _•_ : (v : valeur)(π : pile) → pile

infixr 5 _•_

{-
  c₁, c₂ ::=                  (machine à pile)
           | SKIP
           | PUSH v           (valeur sur pile)
           | ADD              (addition sur pile)
           | IFTE c₁ c₂       (conditionnelle sur pile)
           | c₁ # c₂          (séquence)
-}

data code : Set where
  SKIP :                              code
  PUSH : (v : valeur)               → code
  ADD  :                              code
  IFTE : (c₁ : code)(c₂ : code) → code
  _#_  : (c₁ : code)(c₂ : code) → code

infixr 20 _#_

ex₃ : code
ex₃ = PUSH (estNat 1) #
      PUSH (estNat 1) #
      ADD

ex₄ : code
ex₄ = PUSH (estNat 2) #
      PUSH (estBool true) #
      IFTE (PUSH (estNat 3))
           (PUSH (estNat 1) # PUSH (estNat 7) # ADD) #
      ADD

-- --------------------------------
-- Compilateur
-- --------------------------------

compile : exp → code
compile (val v)        = PUSH v
compile (plus e₁ e₂)   = compile e₂ #
                         compile e₁ #
                         ADD
compile (ifte b e₁ e₂) = compile b #
                         IFTE (compile e₁)
                              (compile e₂)
```

Sémantique & dépendance

Ayant identifié la nécessité de typer les expressions afin d'en simplifier la sémantique, la version suivante introduit le typage au niveau des expressions, des piles et du code machine. Cela a pour conséquence de rendre aisé la définition de la sémantique `semE` des expressions et de la sémantique `semC` du code machine. On peut alors énoncer et prouver le lemme de correction souhaité.

```
open import Lib

-- ================================================================
-- Types dépendants : tout un programme !
--
-- Séminaire du 28 Nov. 2018
-- Pierre-Évariste Dagand -- Sorbonne Université - CNRS - Inria
-- https://bit.ly/2QqmiNC
-- https://www.lip6.fr/Pierre-Evariste.Dagand/stuffs/college-2018/seminaire.zip
-- ================================================================

-- Inspiré de
--     "Correctness of a compiler for arithmetic expressions"
--         McCarthy, J. & Painter, J.A. (1967)

-- "This paper contains a proof of the correctness of a simple
-- compiling algorithm for compiling arithmetic expressions into
-- machine language."

-- "Note that there are no jump instructions. Needless to say, this is
-- a severe restriction on the generality of our results which we
-- shall overcome in future work."

-- --------------------------------
-- Expressions
-- --------------------------------

{-
  v ::=                     (valeurs)
      | true | false        (booléens)
      | 0 | 1 | ...         (entiers naturel)
-}

data type : Set where
   nat bool : type

variable T : type

valeur : type → Set
valeur nat  = ℕ
valeur bool = Bool

{-
  b, e₁, e₂ ::=             (expressions)
              | v           (valeur)
              | plus e₁ e₂  (addition)
              | ifte b e₁ e₂ (conditionnelle)
-}

data exp : type → Set where
  val  : (v : valeur T)                → exp T
  plus : (e₁ : exp nat)(e₂ : exp nat)  → exp nat
  ifte : (b : exp bool)(e₁ e₂ : exp T) → exp T

ex₁ : exp nat
ex₁ = plus (val 1) (val 1)

ex₂ : exp nat
ex₂ = plus (val 2)
           (ifte (val true)
                 (val 3)
                 (plus (val 1) (val 7)))

semE : exp T → valeur T
semE (val v)        = v
semE (plus e₁ e₂)   = semE e₁ + semE e₂
semE (ifte b e₁ e₂) = if semE b then semE e₁ else semE e₂

test-ex₁ : semE ex₁ ≡ 2
test-ex₁ = refl
```

```
-- "The above proposition is occasionally useful."
--         Whitehead & Russell (PM, vol.II, p.362)

test-ex₂ : semE ex₂ ≡ 5
test-ex₂ = refl

-- --------------------------------
-- Machine à pile
-- --------------------------------

{-
  π ::=                       (pile)
      | ε                     (pile vide)
      | v • π                 (valeur sur pile)
-}

type-pile : Set
type-pile = List type

variable σ σ₁ σ₂ σ₃ : type-pile

data pile : type-pile → Set where
  ε   : pile []
  _•_ : valeur T → pile σ → pile (T :: σ)

infixr 5 _•_

tete : pile (T :: σ) → valeur T
tete (t • _) = t

queue : pile (T :: σ) → pile σ
queue (_ • s) = s

{-
  c₁, c₂ ::=                  (machine à pile)
           | SKIP
           | PUSH v           (valeur sur pile)
           | ADD              (addition sur pile)
           | IFTE c₁ c₂       (conditionnelle sur pile)
           | c₁ # c₂          (séquence)
-}

data code : type-pile → type-pile → Set where
  SKIP :                                        code σ σ
  PUSH : (v : valeur T)                       → code σ (T :: σ)
  ADD  :                                        code (nat :: nat :: σ) (nat :: σ)
  IFTE : (c₁ c₂ : code σ₁ σ₂)                 → code (bool :: σ₁) σ₂
  _#_  : (c₁ : code σ₁ σ₂)(c₂ : code σ₂ σ₃) → code σ₁ σ₃

infixr 20 _#_

ex₃ : code [] (nat :: [])
ex₃ = PUSH 1 #
      PUSH 1 #
      ADD

ex₄ : code [] (nat :: [])
ex₄ = PUSH 2 #
      PUSH true #
      IFTE (PUSH 3)
           (PUSH 1 # PUSH 7 # ADD) #
      ADD

semC : code σ₁ σ₂ → pile σ₁ → pile σ₂
semC SKIP π                   = π
semC (PUSH v) π               = v • π
semC ADD (m • n • π)          = m + n • π
semC (IFTE c₁ c₂) (true • π)  = semC c₁ π
semC (IFTE c₁ c₂) (false • π) = semC c₂ π
semC (c₁ # c₂) π              = semC c₂ (semC c₁ π)

-- --------------------------------
```

```
-- Compilateur
-- -------------------------------

compile : exp T → code σ (T :: σ)
compile (val v)        = PUSH v
compile (plus e₁ e₂)   = compile e₂ #
                         compile e₁ #
                         ADD
compile (ifte b e₁ e₂) = compile b #
                         IFTE (compile e₁)
                              (compile e₂)

correction : (e : exp T)(π : pile σ) → semC (compile e) π ≡ semE e • π
correction (val v) π                 = refl
correction (plus e₁ e₂) π
  rewrite correction e₂ π
        | correction e₁ (semE e₂ • π) = refl
correction (ifte b e₁ e₂) π
  rewrite correction b π
  with semE b
... | true                           = correction e₁ π
... | false                          = correction e₂ π
```

Perspectives de recherche

Nous nous sommes contenté d'exploiter les types dépendants pour garantir le typage des expressions. Ainsi, nous avons pu écrire le compilateur de telle sorte qu'il produise du code bien typé à partir d'expressions bien typées. Une extension naturelle serait d'indexer le code machine par sa sémantique, afin que le compilateur puisse garantir que le code produit respecte la sémantique des expressions **par construction**. Comme on le voit dans la version suivante, cela ne se fait pas sans heurt. Souffrons-nous de la limitation de nos outils ? Ou bien sommes-nous en train d'utiliser des types *trop* dépendants ?

```
open import Lib

-- ================================================================
-- Types dépendants : tout un programme !
--
-- Séminaire du 28 Nov. 2018
-- Pierre-Évariste Dagand -- Sorbonne Université - CNRS - Inria
-- https://bit.ly/2QqmiNC
-- https://www.lip6.fr/Pierre-Evariste.Dagand/stuffs/college-2018/seminaire.zip
-- ================================================================

-- Inspiré de
--     "Correctness of a compiler for arithmetic expressions"
--         McCarthy, J. & Painter, J.A. (1967)

-- "This paper contains a proof of the correctness of a simple
-- compiling algorithm for compiling arithmetic expressions into
-- machine language."

-- "Note that there are no jump instructions. Needless to say, this is
-- a severe restriction on the generality of our results which we
-- shall overcome in future work."

-- --------------------------------
-- Expressions
-- --------------------------------

{-
  v ::=                     (valeurs)
      | true | false        (booléens)
      | 0 | 1 | ...         (entiers naturel)
-}

data type : Set where
   nat bool : type

variable T : type

valeur : type → Set
valeur nat  = ℕ
valeur bool = Bool

variable m n : valeur nat
variable b : valeur bool

{-
  b, e₁, e₂ ::=             (expressions)
              | v           (valeur)
              | plus e₁ e₂  (addition)
              | ifte b e₁ e₂ (conditionnelle)
-}

data exp : type → Set where
  val  : (v : valeur T)                → exp T
  plus : (e₁ : exp nat)(e₂ : exp nat)  → exp nat
  ifte : (b : exp bool)(e₁ e₂ : exp T) → exp T

ex₁ : exp nat
ex₁ = plus (val 1) (val 1)

ex₂ : exp nat
ex₂ = plus (val 2)
           (ifte (val true)
                 (val 3)
                 (plus (val 1) (val 7)))

semE : exp T → valeur T
semE (val v)        = v
semE (plus e₁ e₂)   = semE e₁ + semE e₂
semE (ifte b e₁ e₂) = if semE b then semE e₁ else semE e₂
```

```
test-ex₁ : semE ex₁ ≡ 2
test-ex₁ = refl

-- "The above proposition is occasionally useful."
--         Whitehead & Russell (PM, vol.II, p.362)

test-ex₂ : semE ex₂ ≡ 5
test-ex₂ = refl

-- --------------------------------
-- Machine à pile
-- --------------------------------

{-
  π ::=                       (pile)
      | ε                     (pile vide)
      | v • π                 (valeur sur pile)
-}

type-pile : Set
type-pile = List type

variable σ σ₁ σ₂ σ₃ : type-pile

data pile : type-pile → Set where
  ε   : pile []
  _•_ : valeur T → pile σ → pile (T :: σ)

infixr 5 _•_

variable π π₁ π₂ π₃ : pile _

tete : pile (T :: σ) → valeur T
tete (t • _) = t

queue : pile (T :: σ) → pile σ
queue (_ • s) = s

{-
  c₁, c₂ ::=                  (machine à pile)
           | SKIP
           | PUSH v           (valeur sur pile)
           | ADD              (addition sur pile)
           | IFTE c₁ c₂       (conditionnelle sur pile)
           | c₁ # c₂          (séquence)
-}

data code : type-pile → type-pile → Set where
  SKIP :                                        code σ σ
  PUSH : (v : valeur T)                       → code σ (T :: σ)
  ADD  :                                        code (nat :: nat :: σ) (nat :: σ)
  IFTE : (c₁ c₂ : code σ₁ σ₂)                 → code (bool :: σ₁) σ₂
  _#_  : (c₁ : code σ₁ σ₂)(c₂ : code σ₂ σ₃) → code σ₁ σ₃

infixr 20 _#_

ex₃ : code [] (nat :: [])
ex₃ = PUSH 1 #
      PUSH 1 #
      ADD

ex₄ : code [] (nat :: [])
ex₄ = PUSH 2 #
      PUSH true #
      IFTE (PUSH 3)
           (PUSH 1 # PUSH 7 # ADD) #
      ADD

semC : code σ₁ σ₂ → pile σ₁ → pile σ₂
semC SKIP π                  = π
semC (PUSH v) π              = v • π
semC ADD (m • n • π)         = m + n • π
```

```
semC (IFTE c₁ c₂) (true • π)  = semC c₁ π
semC (IFTE c₁ c₂) (false • π) = semC c₂ π
semC (c₁ # c₂) π              = semC c₂ (semC c₁ π)

data code-semC : (σ₁ σ₂ : type-pile) → pile σ₁ → pile σ₂ → Set where
  SKIP : code-semC σ σ π π
  PUSH : (v : valeur T) →
         code-semC σ (T :: σ) π (v • π)
  ADD  : code-semC (nat :: nat :: σ) (nat :: σ) (m • n • π) (m + n • π)
  IFTE : (c₁ : code-semC σ₁ σ₂ π π₁) →
         (c₂ : code-semC σ₁ σ₂ π π₂) →
         code-semC (bool :: σ₁) σ₂ (b • π) (if b then π₁ else π₂)
  _#_ : (c₁ : code-semC σ₁ σ₂ π₁ π₂)
        (c₂ : code-semC σ₂ σ₃ π₂ π₃) →
        code-semC σ₁ σ₃ π₁ π₃

oubli : (π₁ : pile σ₁) → code-semC σ₁ σ₂ π₁ π₂ → code σ₁ σ₂
oubli π₁ SKIP         = SKIP
oubli π₁ (PUSH v)     = PUSH v
oubli π₁ ADD          = ADD
oubli π₁ (IFTE c₁ c₂) = IFTE (oubli _ c₁) (oubli _ c₂)
oubli π₁ (c₁ # c₂)    = oubli _ c₁ # oubli _ c₂

valide : (c : code-semC σ₁ σ₂ π₁ π₂) → semC (oubli π₁ c) π₁ ≡ π₂
valide SKIP                     = refl
valide (PUSH v)                 = refl
valide ADD                      = refl
valide (IFTE {b = true} c₁ c₂)  = valide c₁
valide (IFTE {b = false} c₁ c₂) = valide c₂
valide (c₁ # c₂)
  rewrite valide c₁ | valide c₂  = refl

-- --------------------------------
-- Compilateur
-- --------------------------------

rustine : (b : Bool) {v₁ v₂ : valeur T} →
     if b then (v₁ • π) else (v₂ • π) ≡ (if b then v₁ else v₂) • π
rustine false = refl
rustine true  = refl

compile : (e : exp T) → code-semC σ (T :: σ) π (semE e • π)
compile (val v)        = PUSH v
compile (plus e₁ e₂)   = compile e₂ #
                         compile e₁ #
                         ADD
compile (ifte b e₁ e₂) = compile b #
                         subst (code-semC _ _ _ ) (rustine _)
                               (IFTE (compile e₁)
                                     (compile e₂))

compile' : (π : pile σ) → exp T → code σ (T :: σ)
compile' π e = oubli π (compile e)

correction : (e : exp T)(π : pile σ) → semC (compile' π e) π ≡ semE e • π
correction e π = valide (compile e)
```