digiteo

Recherche en sciences & technologies de l'information

FORUM 2011
Ecole Polytechnique
18 octobre 2011

Nicolas Pouillard, thèse encadrée par François Pottier

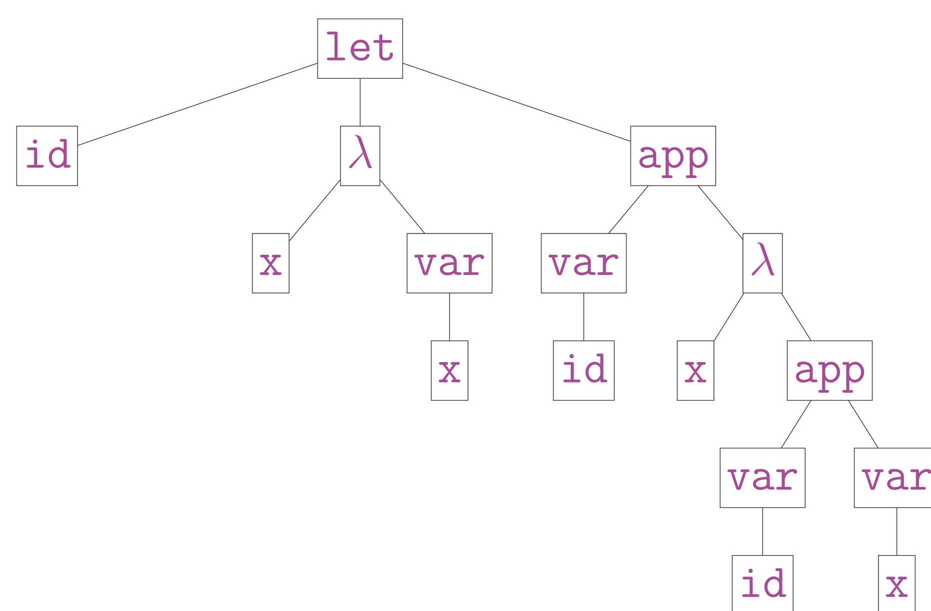
Nº 56 Vers des langages plus expressifs et plus sûrs pour la méta-programmation

RÉSUMÉ

Lorsqu'on écrit un programme `A` chargé de construire un programme `B`, on doit non seulement vérifier que `A` est bien construit (par exemple, toutes les variables sont bien déclarées), ce qui est facile, mais aussi se convaincre que `B` (qui n'existe pas encore!) satisfera cette même propriété, ce qui est beaucoup plus difficile. Dans le cadre de ce projet, nous avons proposé de nouvelles techniques, basées sur le typage statique, pour garantir cette propriété de sûreté.

CONTEXTE

```
let id = λ x → x in
id (λ x → id x)
```



INTERFACE

On offre au programmeur l'interface suivante pour programmer avec les structures contenant des variables et des lieurs. AGDA est le langage utilisé pour ce système:

```
record NomPa : Set_1 where
  field
    Binder  : Set
    zero^B  : Binder
    suc^B   : Binder → Binder

    World : Set
    ∅     : World
    _◁_   : Binder → World → World

    Name    : World → Set
    name^B  : ∀ {α} (b : Binder) → Name (b ◁ α)

    _==^N_     : ∀ {α} (x y : Name α) → Bool
    export^N?  : ∀ {α} (b : Binder) → Name (b ◁ α) →? Name α

    _⊆_        : World → World → Set
    ...          -- omission des témoins d'inclusion
    coerce^N   : ∀ {α β} → (α ⊆ β) → (Name α → Name β)

  _ ^B : ℕ → Binder              _ ^N : ∀ {α} (n : ℕ) → Name (n ^B ◁ α)
  zero ^B = zero^B               n ^N = name^B (n ^B)
  suc n ^B = suc^B (n ^B)
```

EXEMPLES

Le programmeur peut ensuite définir son langage de la manière suivante :

```
data Tm α : Set where
  V    : (x : Name α) → Tm α
  _·_  : (t : Tm α) (u : Tm α) → Tm α
  ƛ    : (b : Binder) (t : Tm (b ◁ α)) → Tm α
  Let  : (b : Binder) (t : Tm α) (u : Tm (b ◁ α)) → Tm α

-- λ f → λ x → f x
apTm : Tm ∅
apTm = ƛ (0 ^B) (ƛ (1 ^B) (V (coerce^N ... (0 ^N)) · V (1 ^N)))
-- λ x → f x
non-closed^D : Tm (1 ^B ◁ ∅)
non-closed^D = ƛ (0 ^B) (V (coerce^N ... (1 ^N)) · V (0 ^N))

rm : ∀ {α} b → List (Name (b ◁ α)) → List (Name α)
rm b [] = []
rm b (x :: xs) with  export^N? b x
...                  | nothing  = rm b xs
...                  | just x′  = x′ :: rm b xs

fv : ∀ {α} → Tm α → List (Name α)
fv (V x)          = [ x ]
fv (fct · arg)    = fv fct ++ fv arg
fv (ƛ b t)        = rm b (fv t)
fv (Let b t u)    = fv t ++ rm b (fv u)
```

SÛRETÉ

La sûreté de l'interface est démontrée avec une technique de *relations logiques* :

```
  e   :   τ          -- pour chaque programme bien typé
⟦ e ⟧ : ⟦ τ ⟧ e e    -- un théorème gratuit

-- Relations indicées par les types
⟦ℕ⟧              x_1 x_2 = x_1 ≡ x_2
⟦Set⟧            A_1 A_2 = A_1 → A_2 → Set
(A_r ⟦→⟧ B_r)    f_1 f_2 = ∀ {x_1 x_2} (x_r : A_r x_1 x_2)
                           → B_r (f_1 x_1) (f_2 x_2)
⟦World⟧          α_1 α_2 = (ℛ : Name α_1 → Name α_2 → Set)
                          × ... -- ℛ préservant les égalités de noms
⟦Name⟧ (ℛ , _) x_1 x_2 = ℛ x_1 x_2

f       : ∀ {α} → Name α → Bool
f_r     : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Name⟧ α_r ⟦→⟧ ⟦Bool⟧) f f
f-const : ∀ x_1 x_2 → f x_1 ≡ f x_2

Ren        : (α β : World) → Set
⟨_⟩        : ∀ {α β} → Ren α β → Tm α → Tm β
f          : ∀ {α} → Tm α → Tm α
f_r        : (∀⟨ α_r : ⟦World⟧ ⟩⟦→⟧ ⟦Tm⟧ α_r ⟦→⟧ ⟦Tm⟧ α_r) f f
f-comm-ren : ∀ {α β} (Φ : Ren α β) → ⟨ Φ ⟩ ∘ f ≗ f ∘ ⟨ Φ ⟩
```

îledeFrance

Inria — INVENTEURS DU MONDE NUMÉRIQUE

Contacts : nicolas.pouillard@inria.fr et francois.pottier@inria.fr