Termination with Dependent Types

Guillaume Genestier

Monday, July 1st, 2019

Dedukti is a type-checker for the $\lambda\Pi$-calculus modulo rewriting.

Example of dependent type

```
def F : Nat -> TYPE
[]  F 0      --> Nat
[n] F (s n) --> Nat -> F n
```

$\texttt{F n} = \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \cdots \rightarrow \texttt{Nat}$ with n arrows.

Example of rewriting rules

```
def sum : (n: Nat) -> F n
[]  sum 0           --> 0
[]  sum (s 0)       --> λx, x
[n] sum (s (s n))   --> λx y, sum (s n) (plus x y)
```

Example : sum 5 1 2 3 4 5 $\longrightarrow^*$ 1+2+3+4+5 $\longrightarrow^*$ 15

Typing Rules

Abstraction:

$$\frac{\Gamma \vdash A : \mathrm{Type} \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

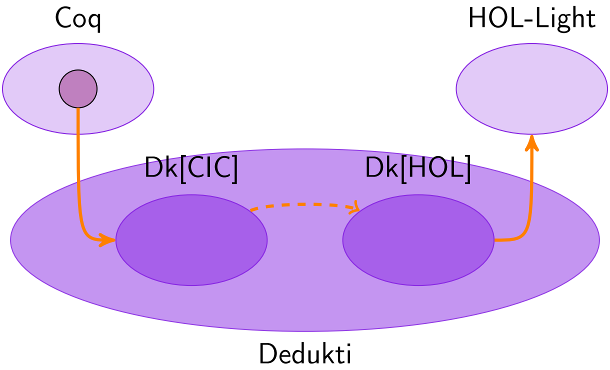
Application:

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B\,[x \mapsto u]}$$

Conversion:

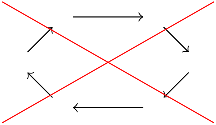
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \leftrightarrow_{\beta\mathcal{R}} B}{\Gamma \vdash t : B}$$

Dedukti is well-suited for interoperability [Thiré18]

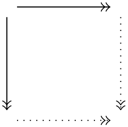


Expected Properties of Rewriting

- Termination: There is no infinite sequence of reduction starting from a well-typed term;



- Typing preservation (*Subject reduction*): If a term is well-typed, its reducts have the same type;
- Confluence: Two reducts of a term have a common reduct.



- Lambda abstraction,

  ```
  [F] ∂ (λ x, ln (F x)) --> λ x, (∂ F x) / (F x)
  ```

- Variable and partial application,

  ```
  [f,x,l] map f (Cons x l) --> Cons (f x) (map f l)
  ```

- Type level rewrite rules,
- Influence of dependency in types.

```
(; Type of the code of simple types ;)
typ : Type.
arrow : typ -> typ -> typ.

(; Decoding function ;)
T : typ -> Type.

(; Constructions of the lambda calculus ;)
lambda : (a : typ) -> (b : typ) -> (T a -> T b) -> T (arrow a b).
def appli : (a: typ) -> (b: typ) -> T (arrow a b) -> T a -> T b.

(; Beta rule ;)
[a,b,f,x] appli a b (lambda _ _ f) x --> f x.
```

Difficulties with higher-order rewriting and dependent types

- Lambda abstraction,

  `[F] ∂ (λ x, ln (F x)) --> λ x, (∂ F x) / (F x)`

- Variable and partial application,

  `[f,x,l] map f (Cons x l) --> Cons (f x) (map f l)`

- Type level rewrite rules,
- Influence of dependency in types.

```
(; No more codes ;)



(; Type of all terms ;)
T : Type.

(; Constructions of the lambda calculus ;)
lambda : (T -> T) -> T.
def appli  : T -> T -> T.

(; Beta rule ;)
[f,x] appli (lambda f) x --> f x.
```

Dependency Pairs

Definition (Dependency Pairs)

A rule $f\ \bar{l} \to r$ gives rise to the *dependency pairs* $f\ \bar{l} > g\ \bar{m}$ where:

- g is (partially) defined by rewriting,
- $g\ \bar{m}$ is a maximally applied subterm of r.

Theorem (Arts and Giesl, 2000)

First order:
$\to_{\mathcal{R}}$ *terminates iff there is no* $f\ \bar{t} > g\ \bar{u} \to^*_{arg} g\ \bar{u}' > \dots$

Higher-Order

Static and dynamic definition: [Blanqui06][Kusakari, Sakai 07][Kop, van Raamsdonk 12][Kop, Fuhs 19]

Example

```
def plus : Nat -> Nat -> Nat.
set infix "+" := plus.
[q]   0     + q     --> q.
[p,q] (S p) + q     --> S (p + q). (1)
[p,q] p     + (S q) --> S (p + q). (2)

def append: (p: Nat) -> List p ->
            (q: Nat) -> List q -> List (p + q).
[q,m]       append _ nil          q m --> m.
[x,p,l,q,m] append _ (cons x p l) q m -->
             cons x (p + q) (append p l q m). (3)
```

```
(1)              (S p) + q     > p + q
(2)              p     + (S q) > p + q
(3) append _ (cons x p l) q m > append p l q m
(3) append _ (cons x p l) q m > p + q
```