F-safe : un langage fonctionnel terminant

Frédéric Peschanski

7 novembre 2011

Résumé

Ce document présente les spécifications du langage de modélisation F-safe, un langage fonctionnel terminant (ou normalisant).

1 Présentation du langage F-safe

Dans cette section est présentée la syntaxe du langage F-safe au travers d'un certain nombre d'exemples. Une syntaxe plus formelle est également spécifiée.

1.1 Types inductifs et fonctions récursives

La première étape du cycle de modélisation en F-safe consiste principalement en la définition de *types inductifs*.

Les entiers de Peano (ou entiers naturels) représentent l'exemple emblématique des types inductifs. Une définition possible en F-safe est la suivante :

```
type nat = Zero | Succ(n:nat)
```

Les entiers naturels de type `nat` sont donc soit le zéro, dénoté `Zero`, soit un le successeur `Succ` d'un entier naturel `n`.

Les identificateurs `Zero` et `Succ` sont des *constructeurs*, que l'on peut donc utiliser pour construire des données de type `nat`. Par exemple, le zéro est construit par l'expression `Zero`. L'entier 3 est construit par `Succ(Succ(Succ(Zero)))`.

On peut ensuite définir des fonctions récursives permettant d'effectuer des calculs arithmétiques sur les entiers naturels.

Par exemple, nous pouvons définir une fonction d'addition de la façon suivante :

```
def add(n:nat, m:nat) : nat = case n {
  | Zero => m
  | Succ(p:nat) => Succ(add(p,m))
}
```

Le calcul de l'addition de 2 et 3 se déroule ainsi :

```
add(2,3) ==> Succ(add(Succ(Zero),3))
         ==> Succ(Succ(add(Zero,3)))
         ==> Succ(Succ(3))
         ==> Succ(4)
         ==> 5  ou (Succ(Succ(Succ(Succ(Succ(Zero))))))
```

Ce style de modélisation ressemble de très près au style de programmation dans un langage fonctionnel comme OCaml. On remarque cependant que les types numériques comme `int` ne sont pas présents en F-safe. En modélisation les entiers relatifs nécessitent une définition inductive et non extensionnelle (cf. Coq, bibliothèque Zarith).

La définition de fonction ci-dessus est en fait un raccourcis syntaxique pour la définition suivante :

```
def add:nat*nat->nat = fun(n:nat,m:nat):nat =>
  case n {
    | Zero => m
    | Succ(p:nat) => Succ(add(p,m))
  }
```

On peut également définir des fonctions mutuellement récursives, sans fioritures syntaxiques :

```
def is_even:nat->bool,is_odd:nat->bool =
  let(even:nat->bool=fun(n:nat) : bool =>
        case n {
         | Zero => True
         | Succ(m:nat) => odd(m)
        },
      odd:nat->bool=fun(n:nat) : bool =>
        case n {
         | Zero => False
         | Succ(Zero) => True
         | Succ(m:nat) => even(m)
        }) { even,odd }
  }
```

Remarque : le type `bool` peut être défini par une simple énumération : `type bool = True | False`

En F-safe chaque donnée d'un type inductif correspond à un arbre dont le type définit la structure, ou de façon équivalente à un terme dont le type définit la grammaire.

Prenons par exemple un type inductif permettant de construire des arbres binaires dont les noeuds contiennent des entiers :

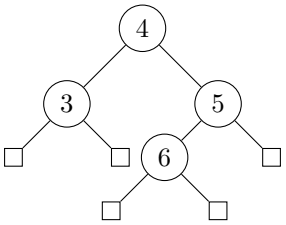
```
type bintree = Empty | Node(val:nat,left:bintree,right:bintree)
```

La définition d'un type inductif sous forme d'un « générateur d'arbres » permet la définition « automatique » d'opérations élémentaires.

Construction explicite de données Voici un exemple d'arbre binaire :

```
Node(4,Node(3,Empty,Empty),
       Node(5,Node(6,Empty,Empty),
              Empty))
```

La représentation arborescente peut être visualisée de la façon suivante :



Déconstruction par filtrage de motif Le filtrage de motif (ou *pattern matching*) peut être employé pour décomposer une donnée. Voici par exemple un prédicat permettant de tester si un arbre binaire est une feuille :

```
def is_leaf(t:bintree) : bool = case t {
  | Node(_:nat,Empty,Empty) => True
  | _:bintree => False
}
```

Remarques :

1. le motif `_:T` permet de filtrer une expression quelconque.
2. le filtrage doit être exhaustif, les fonctions en F-safe doivent être totales

Egalité structurelle On peut comparer deux termes quelconques avec l'opérateur = qui correspond à un isomorphismes d'arbres. L'égalité est définie automatiquement, mais on peut retrouver simplement sa définition récursive, par exemple pour le type nat :

```
def eq_nat(n:nat,m:nat) : bool = case n,m {
  | Zero,Zero => True
  | Succ(n':nat),Succ(m':nat) => eq_nat(n',m')
  | _:nat,_:nat => False
}
```

En comparaison avec les langages de programmation fonctionnelle usuels, les types fonctionnels ne peuvent apparaître dans les types inductifs, notamment parce que l'égalité n'est pas décidable sur les fonctions. En terme de modélisation, cette contrainte n'est pas rédhibitoire.

Taille d'un terme La taille d'un terme `t` de type inductif quelconque peut être obtenue par l'opérateur |t|. Cette taille pourra jouer un rôle important dans la problématique de terminaison. La taille est définie automatiquement, mais voici une définition explicite pour les arbres binaires :

```
def bintree_size(t:bintree) : nat = case t {
  | Empty => Zero
  | Node(_:nat,l:bintree,r:bintree) => Succ(add(bintree_size(l),bintree_size(r)))
}
```

Remarque : la taille d'un terme représente la définition implicite de mesure pour les termes, mesure pouvant être utilisée par l'algorithme de terminaison.

Relation d'ordre Deux termes t_1 et t_2 d'un même type T sont naturellement liés dans une relation d'ordre de type lexicographique, par les comparaisons $t_1 < t_2, t_1 \leq t_2, t_1 > t_2, t_1 \geq t_2$ ainsi que la fonction prédéfinie T_ord. La définition de l'ordre lexicographique - automatique - peut être reconstruite, par exemple sur les arbres binaires, de la façon suivante :

```
def bintree_ord(t1:binttree,t2:bintree) : ord = case t1,t2 {
  | Empty,Empty => Equal
  | Node(v1:nat,l1:bintree,r1:bintree),
    Node(v2:nat,l2:binree,r2:bintree) =>
        case nat_ord(v1,v2) {
          | Lower => Lower
          | Greater => Greater
          | Equal => case bintree_ord(l1,l2) {
             | Lower => Lower
             | Greater => Greater
             | Equal => bintree_ord(r1,r2)
            }
          }
  }
```

Remarque : on suppose `type ord = Lower | Equal | Greater`

La relation d'ordre, dans son interprétation stricte, représente une information cruciale pour l'étude de terminaison des fonctions récursives.

Entrées/sorties On dispose de fonctionnalités permettant les entrées/sorties de termes types, par exemple au format XML.

1.2 Types polymorphes

Le langage F-safe permet le polymorphisme paramétrique comparable à celui proposé en ML.

Un exemple simple concerne le type option permettant un encodage simple de nombreuses fonctions partielles, dans un langage n'acceptant que des fonctions totales :

```
type option[T] = None | Some(v:T)
```

Voici un exemple de fonction partielle sur les entiers naturels :

```
def pred(n:nat) : option[nat] = case n {
  | Zero => None[nat]
  | Succ(n':nat) => Some[nat](n')
}
```

Remarque : lors de l'utilisation d'un constructeur de type polymorphe, on passe au constructeur les valeurs des paramètres, par exemple `None[nat]` pour le constructeur `None` du type `option[nat]` [1].

Un exemple emblématique de type polymorphe et inductif concerne les listes fonctionnelles :

```
type list[T] = Nil | Cons(hd:T,tl:list[T])
```

Voici par exemple la fonction d'application `map` :

```
def map[T,U](f:T->U,l:list[T]):list[U] = case l {
   | Nil[T] => Nil[U]
   | Cons[T](e:T,l':list[T]) => Cons[U](f(e),map[T,U](f,l'))
}
```

Remarque : F-safe accepte donc les fonctions d'ordre supérieur, à condition bien sûr que la terminaison en soit détectée.

Voici un exemple d'utilisation de la fonction `map` :

```
map[nat,bool](is_even,Cons[nat](1,Cons[nat](2,Cons[nat](3,Cons[nat](4,Nil[nat])))))
==> Cons[bool](False,Cons[bool](True,Cons[bool](False,Cons[bool](True,Nil[bool]))))
```

1.3 Type mutuellement inductifs

Le langage F-safe permet d'encoder des types mutuellement inductif, dont voici un exemple :

```
type even = Zero | ENext(n:odd)
and odd = ONext(m:even)
```

Ainsi `ONext(ENext(ONext(Zero)))` est de type `odd` (impaire) alors que `ENext(ONext(Zero))` est de type `even` (pair).

Un autre exemple de type mutuellement inductif, cette fois-ci polymorphe, correspond aux arbres n-aires :

```
type ntree[T]= Node(e:T,f:forest[T])
and forest[T]= FNil | FCons(n:ntree[T],f:forest[T])
```

Bien sûr, ce type peut-être défini de façon plus générale de la façon suivante :

```
type ntree[T] = Node(e:T,f:list[ntree[T]])
```

1.4 Types applicatifs

Remarque : il s'agit d'une extension expérimentale du noyau de F-safe. Les types applicatifs peuvent être construits avec les constructeurs précédents, mais une construction native est beaucoup plus efficace en pratique.

Le langage F-safe définit un constructeur de type applicatif `Map[T->U]` où `T` et `U` sont des expressions de types.

Une contrainte importante concerne l'injectivité de la relation : pour un élément m de type `M=Map[T->U]` alors $\forall v, v' \in T,\ v = v' \implies m(v) = m(v')$

Les opérations élémentaires pour les éléments de type `M=Map[T->U]` sont les mêmes que pour un type « standard », avec quelques différences et ajouts :

– la construction : `{(a1,b1), ...,(aN,bN) }[T->U]` où chaque `aI` est de type `T` et chaque `bI` est de type `U`
– l'accès : `m(a1)` a pour valeur `b1` et `m(b)` a pour valeur `undefined` si `b` n'appartient pas à `m`
– la déconstruction inductive par motif de filtrage, avec les motifs :

1. Les paramètres de constructeur de type peuvent être inférés automatiquement, mais pour simplifier la définition du vérificateur de type (qui est considéré comme séparé de l'inférence), ces paramètres sont indiqués explicitement.

  - {}[T->U] pour la relation vide et
  - {(x :T,y :U),m' :Map[T->U]} avec `x`, `y` et `m'` des variables, pour la sélection d'une association arbitraire `x,y`.
- les opérations ensemblistes usuelles : appartement (`in`) union (`+`), intersection (`^`), différence (`-`)

On peut considérer en guise d'exemple l'implémentation d'un type ensembliste. Soit `T` un type quelconque, on définit `Set[T]=Map[T->unit]` avec `type unit = Unit`

On peut par exemple écrire un test d'appartenance, dans une version naïve :

```
def in_naive[T](e:T,s:Set[T]) : bool = case s {
  | {}[T->unit] => False
  | {(f:T,_:unit),s':Set[T]} => if e=f then True else in_naive[T](e,s')
}
```

Remarque : la construction `if C then E1 else E2` peut s'encoder en `case C { | True => E1 | False => E2 }`

Cette version de l'appartenance est de complexité linéaire, une version efficace peut être écrite de la façon suivante :

```
def in_fast[T](e:T,s:Set[T]) : bool = case s(e) {
  | undefined => False
  | _:unit => True
}
```

De la même manière, on pourra définir les types multi-ensembles `MSet[T]=Map[T->nat]`.

2 Syntaxe formelle

2.1 Spécifications de type

Les spécifications de types inductifs suivent le format général suivant :

$\bigwedge_i T_i\,[\alpha_{i,1},\ldots,\alpha_{i,n}] = \bigoplus_j C_{i,j} \bigotimes_k l_{i,j,k} : U_{i,j,k}$ où

- T_i est un identificateur unique de type
- $\alpha_{i,1},\ldots,\alpha_{i,n}$ sont les paramètres (variables de types) pour T_i
- $C_{i,j}$ est un nom de constructeur de type
- $l_{i,j,k}$ est un label
- $U_{i,j,k}$ est une expression de type

Les expressions de types sont :

- soit des occurrences de variables paramétriques α
- soit des applications $T[U_1,\ldots,U_n]$ où T est un identificateur de type et les $U_1,\ldots,U_n$ (avec $n \geq 0$) sont des expressions de type
- soit une application $Map[T \to U]$ où T et U sont des expressions de type.

Un environnement de définitions de type Δ est une liste ordonnée de spécifications. L'environnement vide est noté $\emptyset$. L'extension d'un environnement Δ par une spécification de type S, ce que l'on note Δ, S est bien formée si et seulement si :

- $S = \bigwedge_i T_i\,[\alpha_{i,1},\ldots,\alpha_{i,n}] = \bigoplus_j C_{i,j} \bigotimes_k l_{i,j,k} : U_{i,j,k}$
- tous les T_i sont distincts, et différents de tous les noms de types déjà définis dans Δ.
- toutes les variables paramétriques α sont distinctes.
- toutes les variables paramétriques α sont instanciées au moins une fois (il existe au moins une occurrence de la variable dans sa portée).
- tous les constructeurs $C_{i,j}$ sont distincts dans T_i (dans un autre type T' on peut utiliser la notation $T' : C_{i,j}$ pour désambiguer).
- tous les labels $l_{i,j,k}$ sont distincts pour tout k.
- $U_{i,j,k}$ est une expression de type avec variable(s), de la forme $V[W_1,\ldots,W_m]$ avec :
  - V est une variable paramétrique α et $m = 0$
  - V est un nom de type dans S, d'arité m, et tel que chaque W_l est soit :
    - une occurrence de variable paramétrique α
    - une expression de type avec variable(s) de Δ mais sans occurrence de V.
  - V est un nom de type dans Δ, d'arité m, et tel que chaque W_l est une expression de type avec variable(s)
- Tout type T_i possède un chemin (imbrication de constructeurs) vers un constructeur terminal (sans paramètre inductif).

Pour illustrer les contraintes (garantissant l'existence d'un plus petit point fixe), voici quelques exemples incorrects :

```
type bad1 = Bad1 and bad1 = Bad2
==> Erreur : identificateur de type bad1 dupliqué
.
type bad2 = Bad2
type bad2 = Bad2'
==> Erreur : type bad2 déjà défini
.
type bad3[T,T] = Bad3(e:T,f:T)
==> Erreur : paramètre T dupliqué
.
type bad4[T] = Bad4
==> Erreur : variable T non-instanciée
.
type bad5 = Bad5 | Bad5(n:nat)
==> Erreur : constructeur Bad5 dupliqué
type notbad5 = Bad5
type notbad5' = Bad5
Bad5 ==>  type notbad5'
Bad5:notbad5 ==> type notbad5
.
type bad6 = Bad6(n:nat,n:bool)
==> Erreur : label n dupliqué dans le constructeur Bad6
.
type bad7 = Bad7(n:bad7)
==> Erreur : type bad7 mal fondé (pas de chemin vers un terminal)
type notbad7 = Notbad7(n:notbad8) and notbad8 = NotBad8(n:notbad7) | Leaf8
.
type bad8[T] = Bad8(n:bad8[bad8[nat]])
==> Erreur : occurrence (directe) du type bad8 dans une application du type bad8
type bad9[T] = Bad9(n:bad9[bad10[T]]) and bad10[T] = Bad10(e:bad9[T])
==> Erreur : occurrence (indirecte) du type bad9 dans une application de type bad9
type notbad9[T] = Notbad9(n:list[notbad9[T]]) | Leaf9
```

2.2 Expressions et programmes

Une expression de F-safe est soit :
- une occurrence de variable `x`
- une constante `C[P1,...,PM](v1,...,vN)` avec `C` un constructeur de type, `P1,...,PM` des expressions de type et `v1,...,vN` des expressions
- une définition locale `let(x1:T1=e1,...,xN:TN=eN) { f }` avec :
  - `x1, ... , xN` des identificateurs de variables
  - `T1,...,TN` des expressions de type
  - `e1,...,eN` des expressions
  - `f` une expression
- un filtrage de motif : `case e1,...,eN { | patt1 => f1 | ... | pattM => fM }` avec :
  - `e1,...,eN` : des expressions
  - `patt1,...,pattM` : des motifs de filtrage
  - `f1,...,fM` : des expressions
- une fonction anonyme : `fun[P1,...,PM](x1:T1,...,xN:TN):V => e` avec :
  - `P1,...,PM` : des variables paramétriques (pour les fonctions polymorphes)
  - `x1,...,xN` : des identificateurs de variables
  - `T1,...,TN` : des expressions de type
  - `V` : une expression de type
  - `e` : une expression
- une application : `f[T1,...,TM](e1,...,eN)` avec :
  - `f` est une expression

- `T1,...,TM` sont des expressions de type
- `e1,...,eN` sont des expressions

Les programme F-safe sont formés d'une suite de :
- définitions de variables `def x1:T1,...,xN:TN = e1,...,eN` avec :
  - `x1,...,xN` : des identificateurs de variable
  - `T1,...,TN` : des expressions de type
  - `e1,...,eN` : des expressions
- expressions

Les motifs de filtrages sont de la forme `p1,...,pN` où chaque `pI` est une expression de motif, au choix :
- une variable `x:T` où `x` est un identificateur de variable et `T` une expression de type
- une variable anonyme `_:T` où `T` est une expression de type
- une constante `C[P1,...,PM](e1,...,eN)` avec `C` un constructeur de type, `P1,...,PM` des expressions de type et `e1,...,eN` des expressions de motif

3 Règles de typage

Dans un environnement de typage Δ et un environnement de type δ, une expression e possède le type T si on peut démontrer $\Delta \bullet \delta \vdash e : T$ par les règles qui suivent.

3.1 Constantes

$$\frac{\begin{array}{l}\Delta = \Delta', \bigwedge_i T_i\,[\alpha_{i,1}, \ldots, \alpha_{i,n}] = \bigoplus_j C_{i,j} \bigotimes_k l_{i,j,k} : U_{i,j,k} \\ \exists i,\ T = T_i[V_1, \ldots, V_n] \\ \Delta \bullet \delta \vdash v_{i,j,k} : U_{i,j,k}\,\{V_1/\alpha_{i,1}, \ldots, V_n/\alpha_{i,n}\}\end{array}}{\Delta \bullet \delta \vdash C_{i,j}\,[V_1, \ldots, V_n]\,(v_{i,j,1}, \ldots, v_{i,j,m}) : T}\ (const)$$

3.2 Variables

$$\frac{}{\Delta \bullet \delta, x : T \vdash x : T}\ (var)$$

3.3 Fonctions anonymes

$$\frac{\Delta \bullet \delta, x_1 : T_1, \ldots, x_n : T_n \vdash e : U}{\Delta \bullet \delta \vdash \text{fun}\,[\alpha_1, \ldots, \alpha_m]\,(x_1 : T_1, \ldots, x_n : T_n) : U \Rightarrow e : [\alpha_1, \ldots, \alpha_m]\,T_1 \times \ldots \times T_n \to U}\ (fun)$$

3.4 Applications

$$\frac{\begin{array}{l}\Delta \bullet \delta \vdash f : [\alpha_1, \ldots, \alpha_m]T_1 \times \ldots \times T_n \to U \\ T'_1 = T_1\,\{V_1/\alpha_1, \ldots, V_m/\alpha_m\} \quad \ldots \quad T'_n = T_n\,\{V_1/\alpha_1, \ldots, V_m/\alpha_m\} \\ U' = U\,\{V_1/\alpha_1, \ldots, V_m/\alpha_m\} \\ \Delta \bullet \delta \vdash e_1 : T'_1 \quad \ldots \quad \Delta \bullet \delta \vdash e_n : T'_n\end{array}}{\Delta \bullet \delta \vdash f[V_1, \ldots V_m](e_1, \ldots, e_n) : U'}\ (app)$$

3.5 Variables lexicales

$$\frac{\begin{array}{l}\delta' = \delta, x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n \\ \Delta \bullet \delta' \vdash e_1 : T_1 \quad \ldots \quad \Delta \bullet \delta' \vdash e_n : T_n \\ \Delta \bullet \delta' \vdash f : U\end{array}}{\Delta \bullet \delta \vdash \text{let}(x_1 : T_1 := e_1, \ldots, \ldots x_n : T_n := e_n)\,\{f\} : U\Delta \vdash}\ (let)$$

3.6 Filtrage

$$\frac{\begin{array}{l}\Delta \bullet \delta \vdash e_1 : T_1 \quad \ldots \quad \Delta \bullet \delta \vdash e_r : T_r \\ \Delta \bullet \delta \vdash p_1^1 : T_1 \quad \ldots \quad \Delta \bullet \delta \vdash p_r^1 : T_r \\ \ldots \\ \Delta \bullet \delta \vdash p_1^m : T_1 \quad \ldots \quad \Delta \bullet \delta \vdash p_r^m : T_r \\ \Delta \bullet \delta \cup \text{vars}(p_1^1, \ldots, p_r^1) \vdash f_1 : U \quad \ldots \quad \Delta \bullet \delta \cup \text{vars}(p_1^m, \ldots, p_r^m) \vdash f_m : U\end{array}}{\Delta \bullet \delta \vdash \text{case } e_1, \ldots, e_r \left\{ \mid p_1^1, \ldots, p_r^1 \Rightarrow f_1 \mid \ldots \mid p_1^m, \ldots, p_r^m \Rightarrow f_m \right\} : U} \ (case)$$

avec $\text{vars}(p_1, \ldots, p_n) = vars(p_1) \cup \ldots \cup \text{vars}(p_n)$ et :
- $\text{vars}(x : T) = \{x : T\}$
- $\text{vars}(\_ : T) = \emptyset$
- $\text{vars}(C(p'_1, \ldots, p'_m)) = vars(p'_1) \cup \ldots \cup \text{vars}(p'_n)$

3.7 Définitions globales

$$\frac{\Delta \bullet \delta \vdash e_1 : T_1 \quad \ldots \quad \Delta \bullet \delta \vdash e_n : T_n}{\Delta \bullet \delta \vdash \text{def } x_1 : T_1, \ldots, \ldots x_n : T_n = e_1, \ldots, e_n : unit} \ (def)$$

4 Sémantique opérationnelle

On définit un environnement global Γ et un environnement local γ contenant des liaisons entre des variables et des valeurs. Les liaisons dans Γ sont statiques, et les liaisons dans γ sont lexicales.

Une réduction globale $\Gamma \models p \to \Gamma' \models p'$ correspond à un pas d'exécution au top-niveau.

Une réduction locale $\Gamma \bullet \gamma \vdash p \to p'$ correspond à un pas d'évaluation au niveau local dans un programme. Les règles sont les suivantes :

4.1 Programmes

$$\frac{\Gamma \models p_1 \to \Gamma' \models v_1}{\Gamma \models p_1\ p_2 \to \Gamma' \models v_1\ p_2} \ (left)$$

$$\frac{\Gamma \models p_2 \to \Gamma' \models v_2}{\Gamma \models v_1\ p_2 \to \Gamma' \models v_2} \ (right)$$

$$\frac{\Gamma \bullet \emptyset \vdash e \to v}{\Gamma \models e \to \Gamma \models v} \ (expr)$$

4.2 Définitions globales

$$\frac{\begin{array}{l}\Gamma \models e_1 \to \Gamma \models v_1 \quad \ldots \quad \Gamma \models e_n \to \Gamma \models v_n \\ x_1, \ldots, x_n \notin \Gamma\end{array}}{\Gamma \models \text{def } x_1 : T_1, \ldots, x_n : T_n = e_1, \ldots, e_n \to \Gamma, x_1 = v_1, \ldots, x_n = v_n \models Unit} \ (def)$$

4.3 Définitions locales

$$\frac{\Gamma \bullet \delta, x_1 = e_1, \ldots, x_n = e_n \vdash f \to v}{\Gamma \bullet \gamma \vdash \text{let}(x_1 : T_1 := e_1, \ldots, \ldots x_n : T_n := e_n)\,\{f\} \to v} \ (let)$$

4.4 Variables

$$\frac{x \notin \gamma \quad \Gamma \bullet \gamma \vdash e \to v}{\Gamma, x = e \bullet \gamma \vdash x : T \to v} \ (gvar)$$

$$\frac{\Gamma \bullet \gamma \vdash e \to v}{\Gamma \bullet \gamma, x = e \vdash x : T \to v} \ (lvar)$$

4.5 Constantes

$$\frac{\begin{array}{c}\Gamma \bullet \gamma \vdash e_1 \to v_1 \\ \dots \\ \Gamma \bullet \gamma \vdash e_m \to v_m\end{array}}{\Gamma \bullet \gamma \vdash C\,[V_1, \dots, V_n]\,(e_1, \dots, e_m) \to C\,[V_1, \dots, V_n]\,(v_1, \dots, v_m)} \ (const)$$

4.6 Applications

$$\frac{\begin{array}{l}\Gamma \bullet \gamma \vdash e_1 \to v_1 \\ \dots \\ \Gamma \bullet \gamma \vdash e_n \to v_n \\ \Gamma \bullet \gamma \vdash f \to \mathrm{fun}[\alpha_1, \dots, \alpha_m](x_1 : T_1, \dots, x_n : T_n) \Rightarrow e \\ \Gamma \bullet \gamma, x_1 = v_1, \dots, x_n = v_n \vdash e\,\{V_1/\alpha_1, \dots, V_m/\alpha_m\} \to v\end{array}}{\Gamma \bullet \gamma \vdash f[V_1, \dots V_m](e_1, \dots, e_n) \to v} \ (app)$$

4.7 Filtrage

$$\frac{\begin{array}{l}\Gamma \bullet \gamma \vdash e_1 \to v_1 \dots \Gamma \bullet \gamma \vdash e_r \to v_r \\ \exists i,\ match_{\Gamma \bullet \gamma}(p_1^i, \dots, p_r^i \Leftrightarrow v_1, \dots, v_r) = \gamma' \neq \bot \\ \forall j < i,\ match_{\Gamma \bullet \gamma}(p_1^j, \dots, p_r^j \Leftrightarrow v_1, \dots, v_r) = \bot \\ \Gamma \bullet \gamma \cup \gamma' \vdash f_i \to v\end{array}}{\Gamma \bullet \gamma \vdash \mathrm{case}\, e_1, \dots, e_r \left\{ \mid p_1^1, \dots, p_r^1 \Rightarrow f_1 \mid \dots \mid p_1^m, \dots, p_r^m \Rightarrow f_m \right\} \to v} \ (match)$$

avec :
- $match_{\Gamma \bullet \gamma}(p_1, \dots, p_n \Leftrightarrow v_1, \dots, v_n) = \bigcup_{i=1}^{n} match_{\Gamma \bullet \gamma}(p_i \Leftrightarrow v_i)$ avec $E \cup \bot = \bot$
- $match_{\Gamma \bullet \gamma}(C\,[V_1, \dots, V_m]\,(p_1, \dots, p_n) \Leftrightarrow C\,[V_1, \dots, V_m]\,(w_1, \dots, w_n)) = \bigcup_{i=1}^{n} match_{\Gamma \bullet \gamma}(p_i, w_i)$
- $match_{\Gamma \bullet \gamma}(x : T \Leftrightarrow w) = \{x = w\}$
- $match_{\Gamma \bullet \gamma}(\_ : T \Leftrightarrow w) = \emptyset$
- $match_{\Gamma \bullet \gamma}(p, v) = \bot$ sinon