# PURe CAMILA: A System for Software Development using Formal Methods

Alexandra Mendes
João Ferreira

Departamento de Informática
Universidade do Minho
http://haskell.di.uminho.pt

15 de Março de 2005

# Conteúdo



# Motivações

- pôr em prática conhecimentos adquiridos:
  - CAMILA
  - VDM e IFAD VDM-SL Toolbox
  - Haskell

  no desenho e construção de uma ferramenta de especificação formal;

# Objectivos

## Objectivos iniciais

- estudo de conceitos associados a ferramentas de especificação formal, nomeadamente:
  - invariantes de tipos de dados;
  - funções parciais (pré-condições);
  - estado;
  - persistência;
- implementação desses conceitos em Haskell;
- sistema modular;

# Invariantes sobre tipos de dados

## Um invariante sobre um tipo de dados...

- restringe valores desse tipo a um sub-conjunto dos valores originais;
- é definido à custa de um predicado;
- é utilizado para definir propriedades sobre esse tipo;

## Invariantes em VDM-SL

```
Stack = seq of int
inv s == forall i in set (elems s) & i mod 2 <> 0
```

# Invariantes sobre tipos de dados

## Um invariante sobre um tipo de dados...

- restringe valores desse tipo a um sub-conjunto dos valores originais;
- é definido à custa de um predicado;
- é utilizado para definir propriedades sobre esse tipo;

## Invariantes em VDM-SL

```
Stack = seq of int
inv s == forall i in set (elems s) & i mod 2 <> 0
```

# Invariantes em Haskell

## Haskell

- Associar invariante *inv* a um tipo de dados $T$:
  - Sistema de classes do Haskell - forma natural;

# Invariantes em Haskell

## Classe CData (Constrained Data)[Nec05]

```
class CData a where
 inv ::  a -> Bool
 inv a = True
 inv' ::  a -> CamilaError a
 inv' a = if (inv a) then return a
                     else fail "Invariant violation"
```

## Classe CData (Constrained Data)

- Definir um invariante para um tipo é declará-lo como instância da classe CData e definir o predicado (inv)!

# Invariantes em Haskell

## Classe CData (Constrained Data)[Nec05]

```
class CData a where
 inv ::  a -> Bool
 inv a = True
 inv' ::  a -> CamilaError a
 inv' a = if (inv a) then return a
                     else fail "Invariant violation"
```

## Classe CData (Constrained Data)

- Definir um invariante para um tipo é declará-lo como instância da classe CData e definir o predicado (inv)!

# Invariantes em Haskell - Exemplos

## Stack de ímpares

```
instance CData Stack where
 inv s = filter odd s == s
```

Como e quando verificar o invariante de um determinado tipo?

## Função PUSH

```
push ::  Int -> Stack -> CamilaError Stack
push n s = inv' (n:s)
```

# Invariantes em Haskell - Exemplos

## Stack de ímpares

```
instance CData Stack where
 inv s = filter odd s == s
```

Como e quando verificar o invariante de um determinado tipo?

## Função PUSH

```
push ::  Int -> Stack -> CamilaError Stack
push n s = inv' (n:s)
```

# Funções Parciais

## Uma função parcial...

- não está definida para todos os elementos do seu domínio;
- tem associada uma pré-condição, que determina para que valores a função está definida;

## Funções parciais em VDM-SL

```
top:  Stack -> int
top(s) == hd s
pre not empty(s);
```

# Funções Parciais

## Uma função parcial...

- não está definida para todos os elementos do seu domínio;
- tem associada uma pré-condição, que determina para que valores a função está definida;

## Funções parciais em VDM-SL

```
top:  Stack -> int
top(s) == hd s
pre not empty(s);
```

# Funções Parciais em Haskell

## Haskell

- uma pré-condição pode ser vista como um **invariante** do domínio da função;
- aproveitando o mecanismo dos invariantes de tipos, surge uma questão: como associar a um tipo invariantes diferentes?

# Funções Parciais em Haskell

## Classe Partial

```
class Partial a where
 pre ::  a -> Bool
 pre a = True
 pre' ::  a -> CamilaError a
 pre' a = if (pre a) then return a
                     else fail "Pre-condition violated"
```

## Classe Partial

- Para definir uma pré-condição é necessário:
  - Criar um tipo de dados local "igual" ao tipo do domínio da função;
  - Definir esse tipo local como instância da classe Partial.

# Funções Parciais em Haskell

## Classe Partial

```
class Partial a where
 pre ::  a -> Bool
 pre a = True
 pre' ::  a -> CamilaError a
 pre' a = if (pre a) then return a
                     else fail "Pre-condition violated"
```

## Classe Partial

- Para definir uma pré-condição é necessário:
  - Criar um tipo de dados local "igual" ao tipo do domínio da função;
  - Definir esse tipo local como instância da classe Partial.

# Funções Parciais em Haskell - Exemplos

## Função top - tipo de dados local

```
data Stack_top = Stack_top Stack deriving Show
```

## Função top - definição da pré-condição

```
instance Partial Stack_top where
 pre (Stack_top s) = not $ s == []
```

## Função top - definição

```
top ::  Stack -> CamilaError Int
top s = do pre' (Stack_top s)
           return (head s)
```

# Funções Parciais em Haskell - Exemplos

## Função top - tipo de dados local

```
data Stack_top = Stack_top Stack deriving Show
```

## Função top - definição da pré-condição

```
instance Partial Stack_top where
 pre (Stack_top s) = not $ s == []
```

## Função top - definição

```
top ::  Stack -> CamilaError Int
top s = do pre' (Stack_top s)
           return (head s)
```

# Funções Parciais em Haskell - Exemplos

## Função top - tipo de dados local

```
data Stack_top = Stack_top Stack deriving Show
```

## Função top - definição da pré-condição

```
instance Partial Stack_top where
 pre (Stack_top s) = not $ s == []
```

## Função top - definição

```
top ::  Stack -> CamilaError Int
top s = do pre' (Stack_top s)
           return (head s)
```

# Estado

## Estado

- Normalmente, especificações em VDM-SL são constituídas por funções e tipos, juntamente com estado e operações sobre esse estado;
- O Haskell é uma linguagem funcional pura: utilizámos a mónade de estado;

## Mónades

- Os invariantes de tipos e as funções parciais utilizam a mónade de erro;
- É necessário "juntar" a mónade de estado com a mónade de erro: utilizámos *monad transformers*;

# Estado

## Estado

- Normalmente, especificações em VDM-SL são constituídas por funções e tipos, juntamente com estado e operações sobre esse estado;
- O Haskell é uma linguagem funcional pura: utilizámos a mónade de estado;

## Mónades

- Os invariantes de tipos e as funções parciais utilizam a mónade de erro;
- É necessário "juntar" a mónade de estado com a mónade de erro: utilizámos *monad transformers*;

# Estado - implementação

## Tipo de dados

```
type ST ivars a = ErrorT CamilaError'
                         (StateT ivars IO) a
```

- *ErrorT* e *StateT* estão definidos nas bibliotecas do GHC;

# Estado - exemplos

## Folder - variáveis de instância

```
data FolderIVars = FolderIVars {
   leftPages ::  Stack,
   rightPages ::  Stack } deriving Show
```

## Folder - estado inicial

```
initialFolder = FolderIVars {
   leftPages = [],
   rightPages = filter odd [1..4] }
```

# Estado - exemplos

## Folder - variáveis de instância

```
data FolderIVars = FolderIVars {
   leftPages ::  Stack,
   rightPages ::  Stack } deriving Show
```

## Folder - estado inicial

```
initialFolder = FolderIVars {
   leftPages = [],
   rightPages = filter odd [1..4] }
```

# Estado - exemplos

## Folder - ler uma página

```
opread ::  ST FolderIVars ()
opread = do lpages <- gets leftPages
            rpages <- gets rightPages
            rpages2 <- liftE $ pop rpages
            lpages2 <- liftE $ push (head rpages)
                                    lpages
            modify(\s -> s { leftPages = lpages2,
                             rightPages = rpages2 })
```

# Persistência

## Introdução

- **Objectivo:** manter o estado de computação para computação;
- Queremos criar instâncias (estados) e executar operações nessas instâncias, sem perder o estado;
- Interpretador baseado em hs-plugins;

## Estado das instâncias

- É necessário "guardar" as instâncias criadas;
- O estado das instâncias é um FiniteMap:

```
instances :: FiniteMap String Dynamic
```

# Persistência

## Introdução

- **Objectivo:** manter o estado de computação para computação;
- Queremos criar instâncias (estados) e executar operações nessas instâncias, sem perder o estado;
- Interpretador baseado em hs-plugins;

## Estado das instâncias

- É necessário "guardar" as instâncias criadas;
- O estado das instâncias é um FiniteMap:

```
instances ::  FiniteMap String Dynamic
```

# Persistência

## Operações

Executar uma operação numa instância corresponde a:

- obter a representação "dinâmica" da instância;
- obter o verdadeiro estado (*fromDyn*);
- executar a operação, tendo como estado inicial o estado obtido no ponto anterior;
- actualizar estado global.

# Conclusões

## Conclusões

- Especificações sem estado são quase traduzidas directamente;
- Especificações com estado são mais trabalhosas, mas sistemáticas;
- As implementações poderão ser melhoradas, mas acreditamos que os ingredientes estão presentes;

## Limitações

- Verificação implícita *versus* Verificação explícita;
- A definição de pré-condições involve a criação de muito código - melhores alternativas?
- Pode ser complicado prototipar certas especificações;

# Conclusões

## Conclusões

- Especificações sem estado são quase traduzidas directamente;
- Especificações com estado são mais trabalhosas, mas sistemáticas;
- As implementações poderão ser melhoradas, mas acreditamos que os ingredientes estão presentes;

## Limitações

- Verificação implícita *versus* Verificação explícita;
- A definição de pré-condições involve a criação de muito código - melhores alternativas?
- Pode ser complicado prototipar certas especificações;

# Trabalho Futuro

## Trabalho Futuro

- Definir estratégias finais para implementação dos conceitos apresentados, para:
  - Criar *frontends* para converter VDM-SL para Haskell;
  - Criar um interpretador "a sério";

# Discussão

Questões?


C. Necco.
Polytypic data processing.
Master's thesis, Facultad de Cs. Físico Matemáticas y Naturales,University of San Luis, Argentina, 2005. (Submitted.).