Exploring the 2LT Desing Space

Hugo Pacheco
hpacheco@di.uminho.pt

Orientadores
Manuel Alcino Cunha
José Nuno Oliveira

Departamento de Informática
Universidade do Minho, Braga, Portugal

11 de Outubro de 2007

Introdução

A transformação de um tipo de dados determina transformações associadas para as instâncias de dados relativas a esse tipo.

2LT

transformação a dois níveis: de tipos e valores associados

1LT

simplificação de expressões em expressões semanticamente equivalentes

Cenário de aplicação: cálculo de funções *point-free*

Lei natural para a função identidade (*nat*-Id):

$$
\begin{aligned}
id \circ f &= f \\
f \circ id &= f
\end{aligned}
$$

As estratégias de reescrita são definidas através da composição de combinadores como:

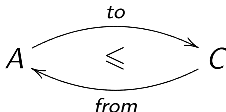
$$
\begin{aligned}
try\ r &= r \mid\mid nop \\
many\ r &= try\ (r \gg many\ r) \\
&\ldots
\end{aligned}
$$

Cenário de aplicação: mapeamento de dados entre diferentes linguagens (XML $\leftrightarrow$ SQL)

Refinamento de Dados

expressa a relação entre especificações abstractas de dados e implementações concretas

Uma 2LT é uma transformação bidireccional em que a função de *representação to* é injectiva e total e a *abstracção from* é sobrejectiva e possivelmente parcial:



O "pacote 2LT" fornece interfaces e permite a automatização e mecanização do processo de reescrita.

2LT - Sistema de Reescrita

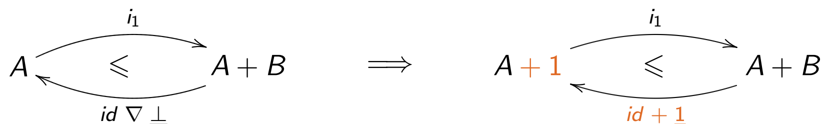
Podemos combinar 2LTs em estratégias de reescrita:

$$A \underset{from}{\overset{to}{\rightleftarrows}} B \underset{from'}{\overset{to'}{\rightleftarrows}} C \qquad (\leqslant)$$

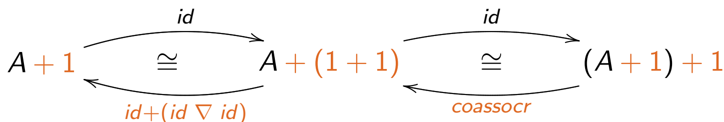
$$\begin{aligned} &try\ r = r \ \|\ nop \\ &many\ r = try\ (r \ \rhd\ many\ r) \\ &\ldots \end{aligned}$$

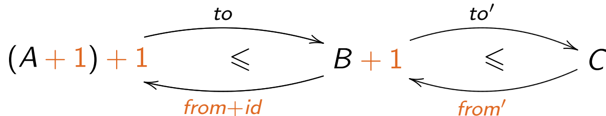
2LT Parcial

A "parcialização" de uma 2LT consiste em adicionar uma alternativa (de insucesso) à função possivelmente parcial *from*, convertendo-a numa função total.
Um exemplo típico é o refinamento *addalt*:



No entanto, a complexidade da transformação inversa *from* aumenta na versão parcializada:





Um tipo recursivo T é representado como o ponto fixo do seu functor base F_T, para os functores:

$$
\begin{array}{lll}
\textbf{newtype}\ \mathit{Id}\ a & = \mathit{Ident}\ \{\mathit{unIdent} :: a\} & \textbf{deriving}\ \mathit{Eq} \\
\textbf{newtype}\ K\ b\ a & = \mathit{Const}\{\mathit{unConst} :: b\} & \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ (g \oplus h)\ a & = \mathit{Inl}\ \ (g\ a) \mid \mathit{Inr}\ (h\ a) & \textbf{deriving}\ \mathit{Eq} \\
\textbf{data}\ (g \otimes h)\ a & = \mathit{Pair}\ (g\ a)\ (h\ a) & \textbf{deriving}\ \mathit{Eq}
\end{array}
$$

Considere definições recursivas para listas e números naturais $\mathbb{N}_0^+$:

$$
\begin{array}{ll}
\textbf{data}\ [a] & = \mathit{Nil} \mid \mathit{Cons}\ a\ [a] \\
F_{[a]} & = K\ () \oplus (K\ a \otimes \mathit{Id}) \\
\textbf{data}\ \mathit{Nat} & = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat} \\
F_{\mathit{Nat}} & = K\ () \oplus \mathit{Id}
\end{array}
$$

1LT c/ Tipos Recursivos

Problema: manter representações *point* − *free* simultâneas para tipos e functores

$$\begin{array}{l} \pi_1 : (a, b) \rightarrow c \\ \pi_1 : (K\ a \otimes K\ b)\ c \rightarrow a \\ \pi_1 : (Id \otimes Id)\ a \rightarrow a \\ \quad \ldots \end{array}$$

Três implementações em Haskell:

Functores adicionar o functor de composição $\odot$ para representar functores aplicados a somas/produtos de tipos:

$$(f \odot (K\ a \otimes K\ b))\ c \cong f\ (a, b)$$

Replicação replicar combinadores para tipos e functores

Classe *Rep* classe Haskell que suporta a conversão implícita de functores em tipos equivalentes:

$$(Id \otimes K\ b)\ a \rightarrow (a, b)$$

Consideramos agora a definição de uma DSL de modelação de sistemas de reescrita para termos genéricos, suportando a conversão nativa entre tipos e functores.
O sistema 1LT pode ser representado como uma instância da nova linguagem:

```
module PF where

%sorts{PF} -- type declaration

out  :: PF
(.)  :: PF -> PF -> PF
cata  :: Type -> PF -> PF
fmap  :: Fctr -> PF -> PF

nat_Id  : id . f -> f
        : f . id -> f

cata_Def : cata t@f g -> g . fmap_KDef (fmap f (cata t g)) . out t

fmap_KDef : fmap (K a) f -> id
          : fmap (f:+:g) h -> fmap_KDef (fmap f h) -|- fmap_KDef (fmap g h)
          : fmap (f:*:g) h -> fmap_KDef (fmap f h) >< fmap_KDef (fmap g h)
```

Se suportássemos polimorfismo

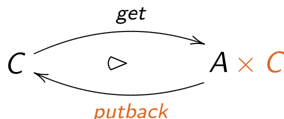
```
(.) :: PF (b -> c) -> PF (a -> b) -> PF (a -> c)
```

“Lentes“

View-update

um modelo concreto é abstraído numa vista

Uma “lente“ define uma transformação bidireccional com noção de estado:



Para ser considerada bem-comportada, uma “lente“ tem que satisfazer as propriedades de:

aceitabilidade $\forall c\ .\ \in C\ .\ \exists\ a\ \in A\ .\ get\ (putback\ (a, c)) = a$

estabilidade $\forall c\ \in C\ .\ putback\ (get\ c, c) = c$

Conclusões

Neste estágio, propusemos três sistemas de reescrita para: 2LT parcial; 1LT com tipos recursivos; e “lentes“.
Como trabalho futuro, existem algumas questões que pretendemos ultrapassar:

- adicionar invariantes de tipos no 2LT;
- melhorar a eficiência e suportar tipos polimórficos no nosso sistema de reescrita genérico;
- implementar “lentes“ para tipos recursivos.

Hiperligações

- relatório completo: `http://Haskell.di.uminho.pt/2ltdocs/estagio/estagio.pdf` (inglês)
- código fonte: repositórios subversion 2lt e rhs em `http://Haskell.di.uminho.pt/svn`