Set Functions for Functional Logic Programming

Michael Hanus

Christian-Albrechts-Universität Kiel

KPS 2009

Joint work with Sergio Antoy, Portland State University

Motivation

Functional Logic Programming

- deterministic (functional) computations (e.g., I/O)
- non-deterministic (logic) computations (e.g., search for solutions)
- desired feature: reason about various solutions (e.g., select best one)
- necessary: encapsulate non-deterministic computations
- non-trivial challenge (many proposals over a decade)

Motivation

Functional Logic Programming

- deterministic (functional) computations (e.g., I/O)
- non-deterministic (logic) computations (e.g., search for solutions)
- desired feature: reason about various solutions (e.g., select best one)
- necessary: encapsulate non-deterministic computations
- non-trivial challenge (many proposals over a decade)

Non-determinism

- expressions may evaluate to more than one value, e.g.,

```
coin = 0 ? 1
```

- predefined operator "`?`" yields either one of its arguments
- non-determinism simplifies modeling and solving problems in many domains

Functional Logic Programming: Example

Flight data as non-deterministic operation

```
flight = (LH469, Portland, Frankfurt, 10:.15)
       ? (NWA92, Portland, Amsterdam, 10:.00)
       ? (LH10,  Frankfurt,Hamburg,    1:.00)
       ? (KL1783,Amsterdam,Hamburg,    1:.52)
       ...
```

Functional Logic Programming: Example

Flight data as non-deterministic operation

```
flight = (LH469, Portland, Frankfurt, 10:.15)
       ? (NWA92, Portland, Amsterdam, 10:.00)
       ? (LH10,  Frankfurt,Hamburg,    1:.00)
       ? (KL1783,Amsterdam,Hamburg,    1:.52)
       ...
```

Computation of an itinerary: non-stop

```
itinerary orig dest
   | flight =:= (num,orig,dest,len)
   = [num]                              where num,len free
```

Functional Logic Programming: Example

Flight data as non-deterministic operation

```
flight = (LH469, Portland, Frankfurt, 10:.15)
       ? (NWA92, Portland, Amsterdam, 10:.00)
       ? (LH10,  Frankfurt,Hamburg,    1:.00)
       ? (KL1783,Amsterdam,Hamburg,    1:.52)
       ...
```

Computation of an itinerary: non-stop or one-stop

```
itinerary orig dest
   | flight =:= (num,orig,dest,len)
   = [num]                              where num,len free
itinerary orig dest
   | flight =:= (num1,orig,stop,len1)
   & flight =:= (num2,stop,dest,len2)
   = [num1,num2]        where num1,len1,num2,len2,stop free
```

Functional Logic Programming: Example

Flight data as non-deterministic operation

```
flight = (LH469, Portland, Frankfurt, 10:.15)
       ? (NWA92, Portland, Amsterdam, 10:.00)
       ? (LH10,  Frankfurt,Hamburg,    1:.00)
       ? (KL1783,Amsterdam,Hamburg,    1:.52)
       ...
```

Computation of an itinerary: non-stop or one-stop

```
itinerary orig dest
   | flight =:= (num,orig,dest,len)
   = [num]                              where num,len free
itinerary orig dest
   | flight =:= (num1,orig,stop,len1)
   & flight =:= (num2,stop,dest,len2)
   = [num1,num2]        where num1,len1,num2,len2,stop free
```

Problem: itinerary with shortest air time? $\leadsto$ reason about *all* values

Encapsulating Search

Encapsulate non-deterministic computations

Standard approach: provide "set of values" operation

$$\mathcal{S}(e) = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\} \quad \approx \text{set of all the values derivable from } e$$

Encapsulating Search

Encapsulate non-deterministic computations

Standard approach: provide "set of values" operation

$$\mathcal{S}(e) = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\} \quad \approx \text{set of all the values derivable from } e$$

```
coin = 0 ? 1
```

$\mathcal{S}$(`coin`) $\xrightarrow{*}$ `???`

Encapsulating Search

Encapsulate non-deterministic computations

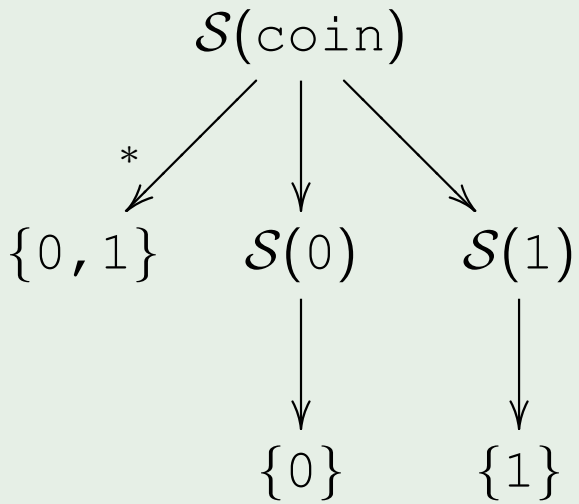
Standard approach: provide "set of values" operation

$$\mathcal{S}(e) = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\} \quad \approx \text{set of all the values derivable from } e$$

```
coin = 0 ? 1
```

$\mathcal{S}$(`coin`) $\xrightarrow{*}$ `???`

Result depends on evaluation order!



Encapsulating Search

Encapsulate non-deterministic computations

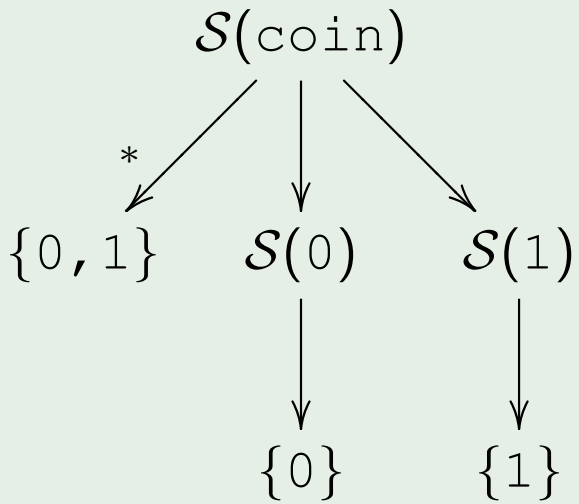
Standard approach: provide "set of values" operation

$$\mathcal{S}(e) = \{v \mid e \xrightarrow{*} v, v \text{ is a value}\} \quad \approx \text{set of all the values derivable from } e$$

```
coin = 0 ? 1
```

$\mathcal{S}$(`coin`) $\xrightarrow{*}$ `???`

Result depends on evaluation order!



- declarative languages: order independence
- don't reason about individual computation steps

Set Functions

Idea

Associate to any operation f a new operation f_S (set function)

- f_S computes set of all values computed by f
- for all values $\bar{c}$: $f_S\ \bar{c} \approx$ set of all values of $f\ \bar{c}$

Set Functions

Idea

Associate to any operation f a new operation f_S (set function)

- f_S computes set of all values computed by f
- for all values $\bar{c}$: $f_S\ \bar{c} \approx$ set of all values of $f\ \bar{c}$
- capture non-determinism of f
- exclude non-determinism originating from arguments

Set Functions

Idea

Associate to any operation f a new operation f_S (set function)

- f_S computes set of all values computed by f
- for all values $\bar{c}$: $f_S\ \bar{c} \approx$ set of all values of $f\ \bar{c}$
- capture non-determinism of f
- exclude non-determinism originating from arguments

`coin = 0 ? 1` $\leadsto$ $\texttt{coin}_S = \{0,1\}$

`id x = x` $\leadsto$ $\texttt{id}_S\ \texttt{v} = \{\texttt{v}\}$ for all values `v`

Set Functions

Idea

Associate to any operation f a new operation f_S (set function)

- f_S computes set of all values computed by f
- for all values $\bar{c}$: $f_S\ \bar{c} \approx$ set of all values of $f\ \bar{c}$
- capture non-determinism of f
- exclude non-determinism originating from arguments

```
coin = 0 ? 1      ⇝ coin_S = {0,1}
id x = x          ⇝ id_S v = {v}     for all values v

bigCoin = 2 ? 4
f x = coin + x
```

Set Functions

Idea

Associate to any operation f a new operation $f_{\mathcal{S}}$ (set function)

- $f_{\mathcal{S}}$ computes set of all values computed by f
- for all values $\bar{c}$: $f_{\mathcal{S}}\ \bar{c} \approx$ set of all values of $f\ \bar{c}$
- capture non-determinism of f
- exclude non-determinism originating from arguments

```
coin = 0 ? 1      ⇝ coin_S = {0,1}
id x = x          ⇝ id_S v = {v}     for all values v

bigCoin = 2 ? 4
f x = coin + x
S(f bigCoin)  ⇝  {2,3,4,5}
```

Set Functions

Idea

Associate to any operation f a new operation $f_{\mathcal{S}}$ (set function)

- $f_{\mathcal{S}}$ computes set of all values computed by f
- for all values $\bar{c}$: $f_{\mathcal{S}}\ \bar{c}\ \approx$ set of all values of $f\ \bar{c}$
- capture non-determinism of f
- exclude non-determinism originating from arguments

```
coin = 0 ? 1      ⇝ coin_S = {0,1}
id x = x          ⇝ id_S v = {v}     for all values v

bigCoin = 2 ? 4
f x = coin + x
```

$\mathcal{S}$(`f bigCoin`) ⇝ `{2,3,4,5}`

`f`$_{\mathcal{S}}$ `bigCoin` ⇝ `{2,3}` or `{4,5}`

Set Functions: Properties

Determinism of set functions

If f is an operation of a program, then f_S is a deterministic operation.

Set Functions: Properties

Determinism of set functions

If f is an operation of a program, then f_S is a deterministic operation.

Reminder: order dependence of "set of values"

$$\mathcal{S}(\texttt{coin}) \xrightarrow{*} \{\texttt{0},\texttt{1}\} \text{ or } \{\texttt{0}\} \text{ or } \{\texttt{1}\}$$

Set Functions: Properties

Determinism of set functions

If f is an operation of a program, then f_S is a deterministic operation.

Reminder: order dependence of "set of values"

$$\mathcal{S}(\texttt{coin}) \xrightarrow{*} \{0,1\} \text{ or } \{0\} \text{ or } \{1\}$$

Order independence of set functions

"Over-evaluating" the argument of a set function when evaluating some expression does not affect the result of the set function.

Programming with Set Functions

n-queens puzzle

Place n queens on an $n \times n$ board without capturing:

- represent placement by a permutation (row of each queen)
- choose a *safe* permutation

n-queens puzzle

Place n queens on an $n \times n$ board without capturing:

- represent placement by a permutation (row of each queen)
- choose a *safe* permutation

A permutation is not safe if some queens are in the same diagonal:

```
unsafe (_++[x]++y++[z]++_)
                    = abs (x-z) =:= length y + 1
```

Programming with Set Functions

n-queens puzzle

Place n queens on an $n \times n$ board without capturing:

- represent placement by a permutation (row of each queen)
- choose a *safe* permutation

A permutation is not safe if some queens are in the same diagonal:

```
unsafe (_++[x]++y++[z]++_)
                        = abs (x-z) =:= length y + 1

queens n | isEmpty (unsafe_S p) = p
         where p = permute [1..n]
```

Note: use of set function is important here
(all occurrences of `p` must denote the *same* permutation!)

Computing shortest paths

Consider `itinerary` program
an itinerary `it` is the shortest one if there is no itinerary shorter than `it`:

```
shortestItin s e
  | isEmpty (shorterItinThan_S (duration it))
  = it
  where it = itinerary s e

        shorterItinThan itduration
           | duration its < itduration
           = its
           where its = itinerary s e
```

Notes:

- distinction between different sets of values is essential
- direct executable specification

Computing shortest paths

Alternative (more efficient) definition:

compute set of all itineraries and select shortest in this set

(`minSet`: computes minimal element of a set w.r.t. total ordering)

```
shortestItin s e = minSet shorter (itinerary_S s e)
   where
     shorter it1 it2 = duration it1 <= duration it2
```

Related Work

General problem: `let x =` e' `in` $\mathcal{S}(e[\mathtt{x}])$

Share non-determinism of e' inside/outside the capsule?

Related Work

General problem: `let x =` e' `in` $\mathcal{S}(e[\mathtt{x}])$

Share non-determinism of e' inside/outside the capsule?

[H./Steiner'98] `try` operator to define search strategies, sharing not considered

[Lux'99] weak encapsulation, sharing preserved, encapsulation depend on syntactic structure of expressions

[Braßel/H./Huch'04] strong encapsulation, no sharing, search in I/O monad

[Braßel/Huch'07] strong encapsulation, demand-driven implementation, no sharing (unintended results)

[Antoy/Braßel'07] `getAllValues` primitive, no sharing, no levels of non-determinism

[López-Fraguas/Sánchez-Hernández'04] computing with failures, check emptiness of all values, no reasoning about various values

Conclusions

Set functions

- associate to each function a *set function*
- set function encapsulate non-determinism caused by function definition
- non-determinism of arguments not encapsulated
- natural programming style
- first provable order-independent approach to encapsulation
- lazy implementation possible by *fingerprints*
- solution to a long-standing problem:
  new standard for encapsulation in functional logic languages?

Conclusions

Set functions

- associate to each function a *set function*
- set function encapsulate non-determinism caused by function definition
- non-determinism of arguments not encapsulated
- natural programming style
- first provable order-independent approach to encapsulation
- lazy implementation possible by *fingerprints*
- solution to a long-standing problem:
  new standard for encapsulation in functional logic languages?

Future aspects

- Implementation of set functions in current FLP systems
- Parallel evaluation of set functions (“or-parallelism”) [Reck/Fischer]