Set Functions for FLP

Sergio Antoy
Portland State University

PPDP'09 – Coimbra, Portugal, Sept 7–9, 2009

Joint work with **Michael Hanus**, CAU Kiel

Partial support by DFG Ha 2457/5-2 and DAAD D/06/29439 and D/08/11852

Introduction

- Non-determinism is a major feature of Functional Logic Programming.
- A functional logic program is non-deterministic when some expression evaluates to *distinct* values, e.g., in Curry:

```
coin = 0 ? 1
```

- The predefined operator `?` yields either one of its arguments.
- Non-determinism simplifies modeling and solving problems in many domains, e.g., modeling a set of flights:

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

Get one

Non-deterministic functions are used in two ways: either get *one* value or get *all* the values satisfying some conditions.

Example: find a non-stop or one-stop flight from Portland to Hamburg.

```
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
```

Get all

Example: find a non-stop or one-stop flight from Portland to Hamburg with shortest time in the air.

- Must compute the *set* of flights from Portland to Hamburg ...
- to find a minimal element according to some criterion.
- The language provides a set type and a primitive.
- The primitive computes the set of values of some expression.
- The set type has operations for finding a minimal element.

Get all

Example: find a non-stop or one-stop flight from Portland to Hamburg with shortest time in the air.

- Must compute the *set* of flights from Portland to Hamburg ...
- to find a minimal element according to some criterion.
- The language provides a set type and a primitive.
- The primitive computes the set of values of some expression.
- The set type has operations for finding a minimal element.
- Unfortunately, the *order of evaluation* affects the result.

Unfortunately

Suppose that $\mathcal{S}(e)$ computes the set of all the values of e.

Recall that `coin = 0 ? 1`.

What is the value of $\mathcal{S}(\texttt{coin})$?

Unfortunately

Suppose that $\mathcal{S}(e)$ computes the set of all the values of e.

Recall that `coin = 0 ? 1`.

What is the value of $\mathcal{S}(\texttt{coin})$?

It depends on the order of evaluation!

Unfortunately

Suppose that $\mathcal{S}(e)$ computes the set of all the values of e.

Recall that `coin = 0 ? 1`.

What is the value of $\mathcal{S}$(`coin`)?

It depends on the order of evaluation!

Case 1: apply $\mathcal{S}$ *before* evaluating `coin`. Result: $\{$`0,1`$\}$

Case 2: apply $\mathcal{S}$ *after* evaluating `coin`. Result: $\{$`0`$\}$ `?` $\{$`1`$\}$

Unfortunately

Suppose that $\mathcal{S}(e)$ computes the set of all the values of e.

Recall that `coin = 0 ? 1`.

What is the value of $\mathcal{S}$(`coin`)?

It depends on the order of evaluation!

Case 1: apply $\mathcal{S}$ *before* evaluating `coin`. Result: {`0`,`1`}

Case 2: apply $\mathcal{S}$ *after* evaluating `coin`. Result: {`0`} `?` {`1`}

There are two problems with $\mathcal{S}$: consistency and semantics.

Non right-linear rules *(sharing)* make $\mathcal{S}$ inconsistent.

The Idea

Get rid of $\mathcal{S}$.

Every function f, implicitly defines a function $f_{\mathcal{S}}$ as follows:

> For each tuple of argument *values* $\bar{c}$,
> $f_{\mathcal{S}}\ \bar{c}$ is the set of all the values of $f\ \bar{c}$.

The Idea

Get rid of $\mathcal{S}$.

Every function f, implicitly defines a function $f_{\mathcal{S}}$ as follows:

> For each tuple of argument *values* $\bar{c}$,
> $f_{\mathcal{S}}\ \bar{c}$ is the set of all the values of $f\bar{c}$.

Examples:

`coin = 0 ? 1`	$\texttt{coin}_{\mathcal{S}}$ `=` $\{\texttt{0,1}\}$
`id x = x`	$\texttt{id}_{\mathcal{S}}$ `x =` $\{\texttt{x}\}$

The Idea

Get rid of $\mathcal{S}$.

Every function f, implicitly defines a function $f_{\mathcal{S}}$ as follows:

> For each tuple of argument *values* $\bar{c}$,
> $f_{\mathcal{S}}\ \bar{c}$ is the set of all the values of $f\bar{c}$.

Examples:

`coin = 0 ? 1`	$\texttt{coin}_{\mathcal{S}}$ `= {0,1}`
`id x = x`	$\texttt{id}_{\mathcal{S}}$ `x = {x}`

Given:

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

The value of $\texttt{f}_{\mathcal{S}}\,\texttt{bigCoin}$ is `{2,3} ? {4,5}`,
whereas the value of $\mathcal{S}(\texttt{f bigCoin})$ is `{2,3,4,5}`.

Properties

- Results are *independent* of the order of evaluation.

  must define the class of programs
  and the notion of independent steps.

Properties

- Results are *independent* of the order of evaluation.

  must define the class of programs
  and the notion of independent steps.

- $f_{\mathcal{S}}$ is deterministic for any f:

  non-determinism of arguments is irrelevant.

Properties

- Results are *independent* of the order of evaluation.

  must define the class of programs
  and the notion of independent steps.

- $f_{\mathcal{S}}$ is deterministic for any f:

  non-determinism of arguments is irrelevant.

- Can still compute $\mathcal{S}(e)$ for any compile-time e:

  as $e_{\mathcal{S}}$.

Programming

The usual n-queens puzzle

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

% queens x and y capture each other
unsafe (_++[x]++y++[z]++_)
   = abs (x-z) =:= length y + 1
```

Testing the safety with $\mathcal{S}$(`unsafe p`)
would produce an *unintended* result.

The non-determinism of `permute` must be excluded
from the non-determinism of `unsafe`.

Set functions are the *intended* semantics.

Implementation

- Exists only on paper, but proved correct.
- The evaluation of $f_{\mathcal{S}}$ is lazy and complete.
- $f_{\mathcal{S}}$ is not actually coded or implemented. Rather, the values of $f\,\bar{t}$ provide $f_{\mathcal{S}}\,\bar{t}$.
- The computations of $f\,\bar{t}$ must distinguish between steps of $\bar{t}$ and steps of f.
- The non-deterministic steps of $\bar{t}$ contribute different values of $f_{\mathcal{S}}\,\bar{t}$.
- The non-deterministic steps of f contribute different elements in a value of $f_{\mathcal{S}}\,\bar{t}$.

Related work

- "Set of values" is a primitive in both Curry and Toy
- Sharing makes order of evaluation uncontrollable [Braßel et al.]
- Weak encapulation (preserve sharing) in MCC [Lux]
- Strong encapsulation (sever sharing) in KICS [Braßel et al.]
- Formalizes order independence, discovers levels [Antoy et al.]
- Constructive negation [Lopez-Fraguas et al.]

Conclusion

- New approach to non-deterministic computations
- Turns away from “set of values” primitive
- Introduces function sets
- Separates levels of non-determinism
- Proves order independence
- Is natural for non-trivial problems
- Proposes provably correct implementation

The End