An Implementation of Narrowing Strategies

Frank Steiner

CAU Kiel

Joint work with

Sergio Antoy *Bart Massey* *Michael Hanus*

Portland State University *CAU Kiel*

Introduction

Modern functional logic languages often based on narrowing

Two implementation principles:

- Prolog code
- abstract machines

Drawbacks:

- restricted machine language
- different language for machine and implementation
- depth-first search $\rightsquigarrow$ incompleteness

  Exception: *Curry2Java* – complete but inefficient

A new Implementation Approach

We present a new back end:

- translation into an imperative language (Java)
- same language for machine and its implementation $\leadsto$ easy to maintain
- breadth-first search $\leadsto$ completeness
- generic:
  - several frontend languages
  - evaluation strategy is not fixed
  - covers constructor-based left-linear TRS (Antoy 2001)

Functional Logic Languages

Integration of functional and logic languages:

- from FP: equations, higher-order, (laziness) ...
- from LP: logic variables, search, constraints ...
- functions working an algebraic data types

```
data Nat = Z | S Nat

add Z y     = y
add (S x) y = S (add x y)
```

Operational semantics:

- Narrowing
  - sound and complete
  - search through logic variables
  - optimal strategies (needed narrowing)
- Residuation
  - suspend on free variables
  - incomplete

Modern languages: Curry, Toy, Escher, Mercury, Oz, ...

A Needed Narrowing Example

Case-expressions encode strategy:

```
add x y = case x of Z   → y                    add Z y     = y
                    S n → S (add n y)          add (S x) y = S (add x y)
```

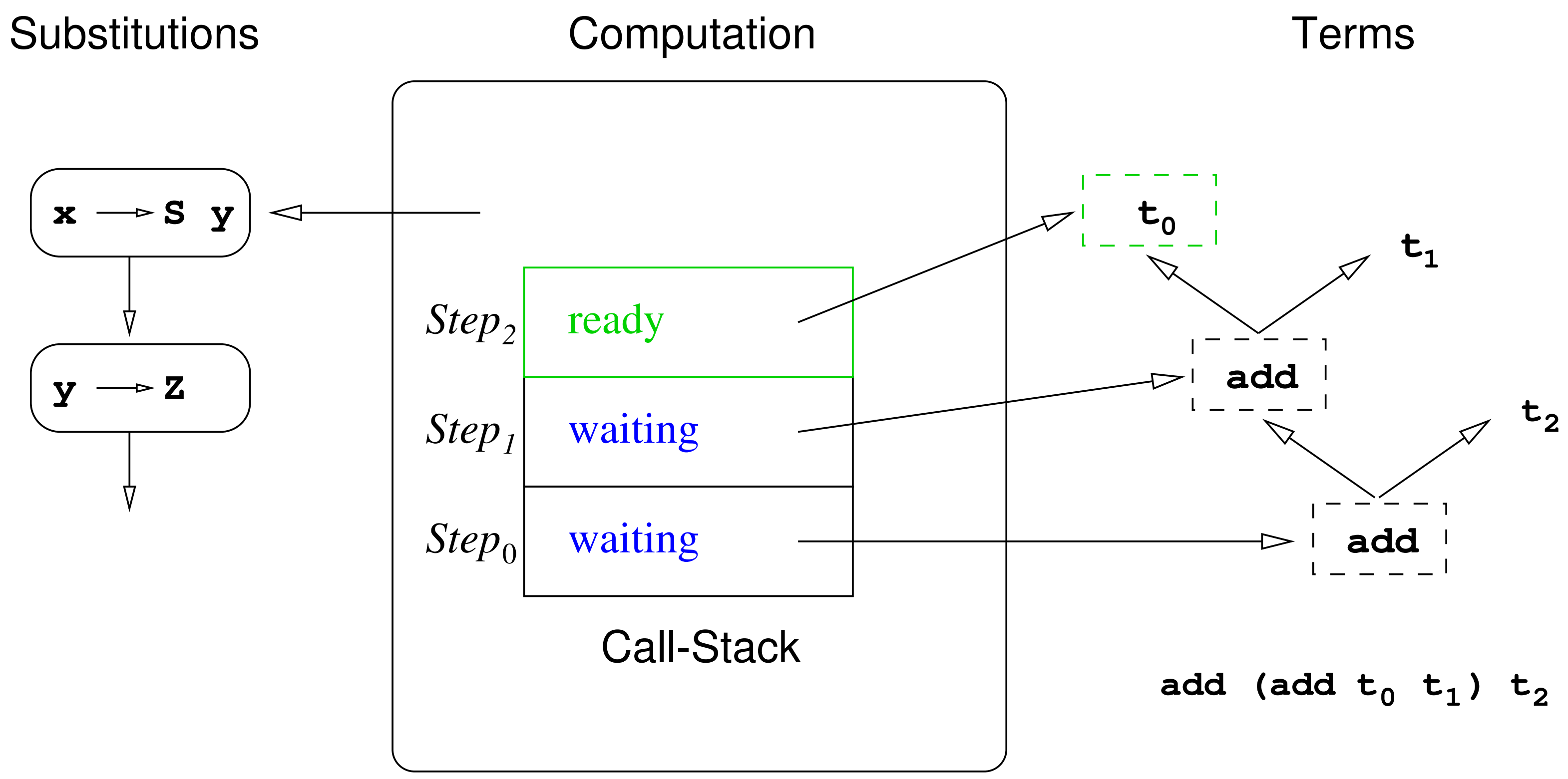
1. `add Z` $t \rightsquigarrow t$ *Pattern Matching*

2. `add x` $t \rightsquigarrow \{x{=}Z\}$ `add Z` t | $\{x{=}S\ n\}$ `add (S n)` t *Narrowing*

3. `add (add Z` t_1`)` $t_2 \rightsquigarrow$ `add` $t_1\ t_2$ *Evaluate Subterm*

Implementation of Narrowing

Term:	head symbol (*expand* method), arguments	Java objects
Reduction step:	replace term with reduct	
Computation:	organize chain of steps	
Substitutions:	incrementally	



Term Reduction

Head symbol contains expansion code:

```
add x y = case x of Z    → y
                    S n  → S (add n y)
```

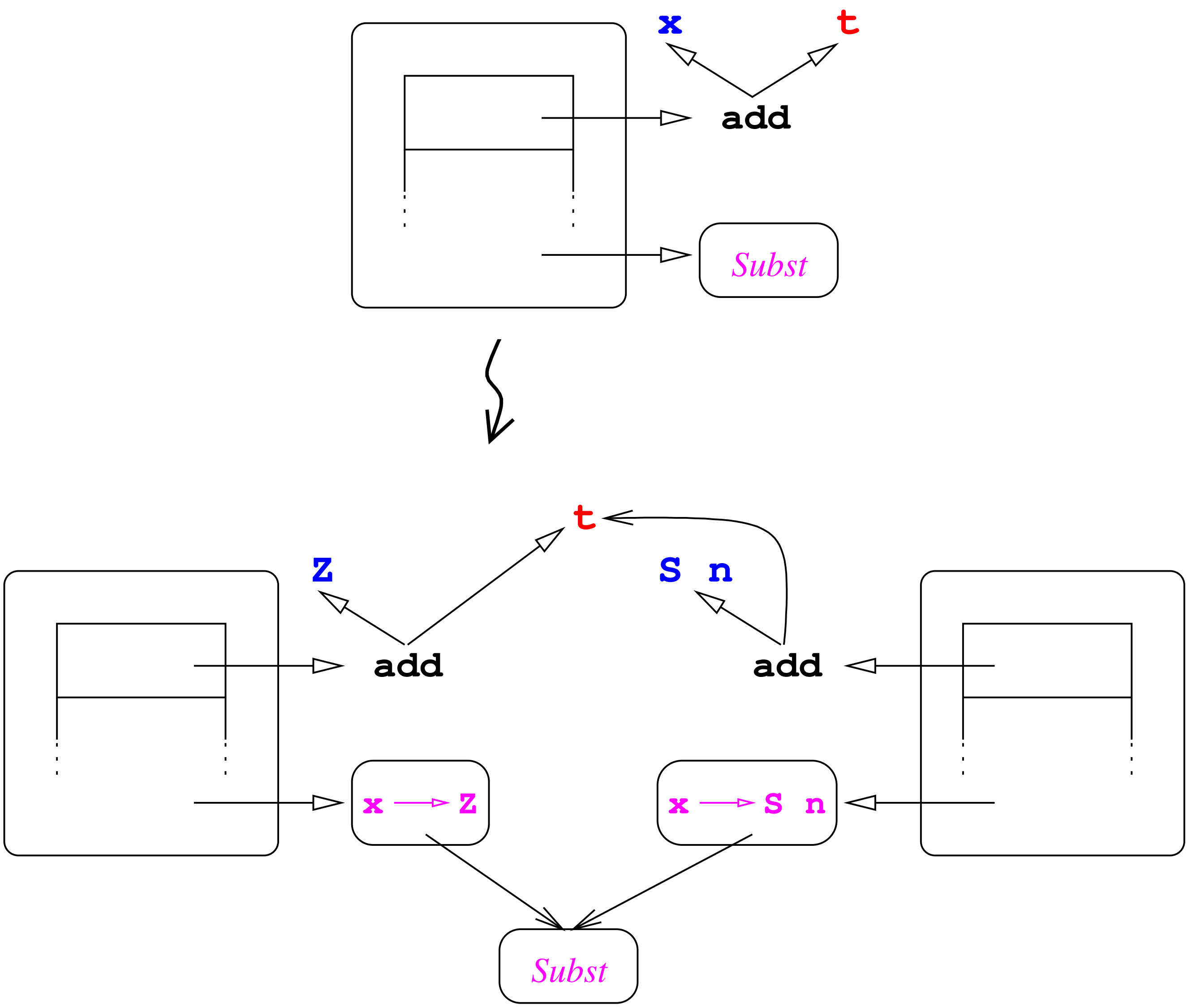
Java method of symbol add (simplified):

```
expand (Computation comp) {
    term = comp.getTerm();
    x = term.getArg(0); y = term.getArg(1);
    switch on x

    case constructor:                          Pattern Matching
    switch on constructor symbol
       case Z: term.update(y);
       case S: n = x.getArg(0);
                   term.update(S (add n y));

    case variable:                             Narrowing
       x.bindTo(Z);       comp.spawn(add Z y);
       x.bindTo(S newV); comp.spawn(add (S newV) y);
       comp.kill();

    case operation:                            Evaluate subterm
       comp.pushOnStack(x);
    }
```

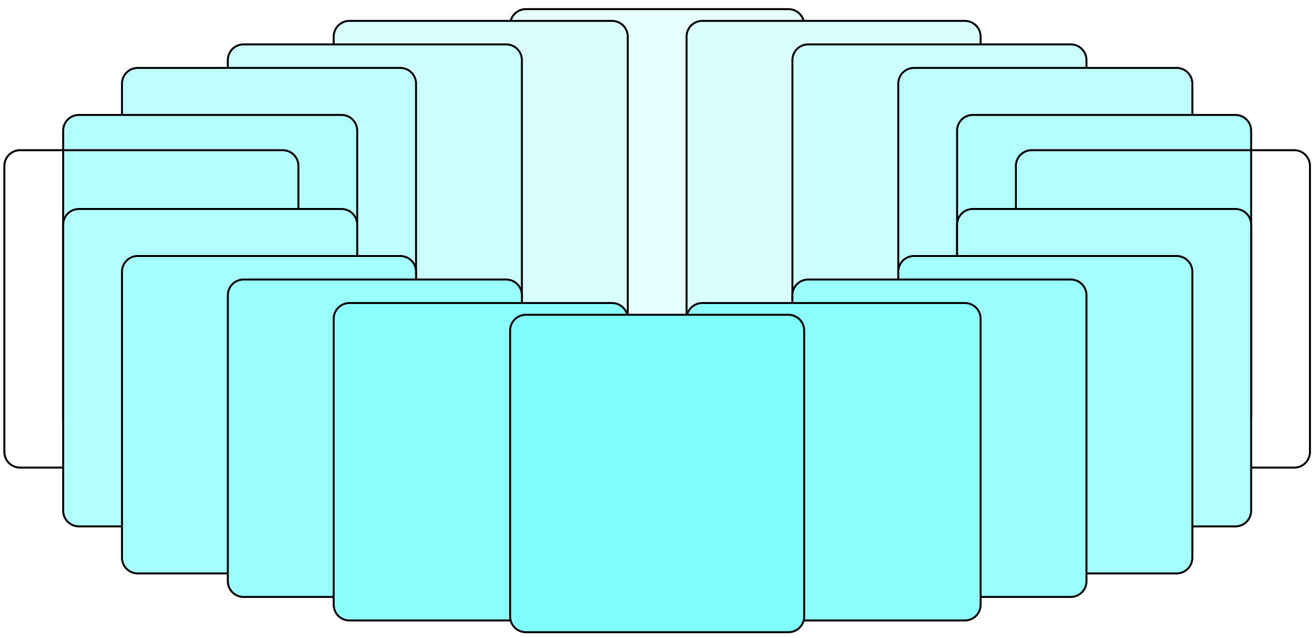
Nondeterminism and Completeness

Narrowing splits a computation:

`add` $\mathtt{x}$ t $\rightsquigarrow$ $\{x{=}Z\}$ `add Z` t `|` $\{x{=}S\ n\}$ `add (S n)` t



$\Rightarrow$ computations share terms/substitutions



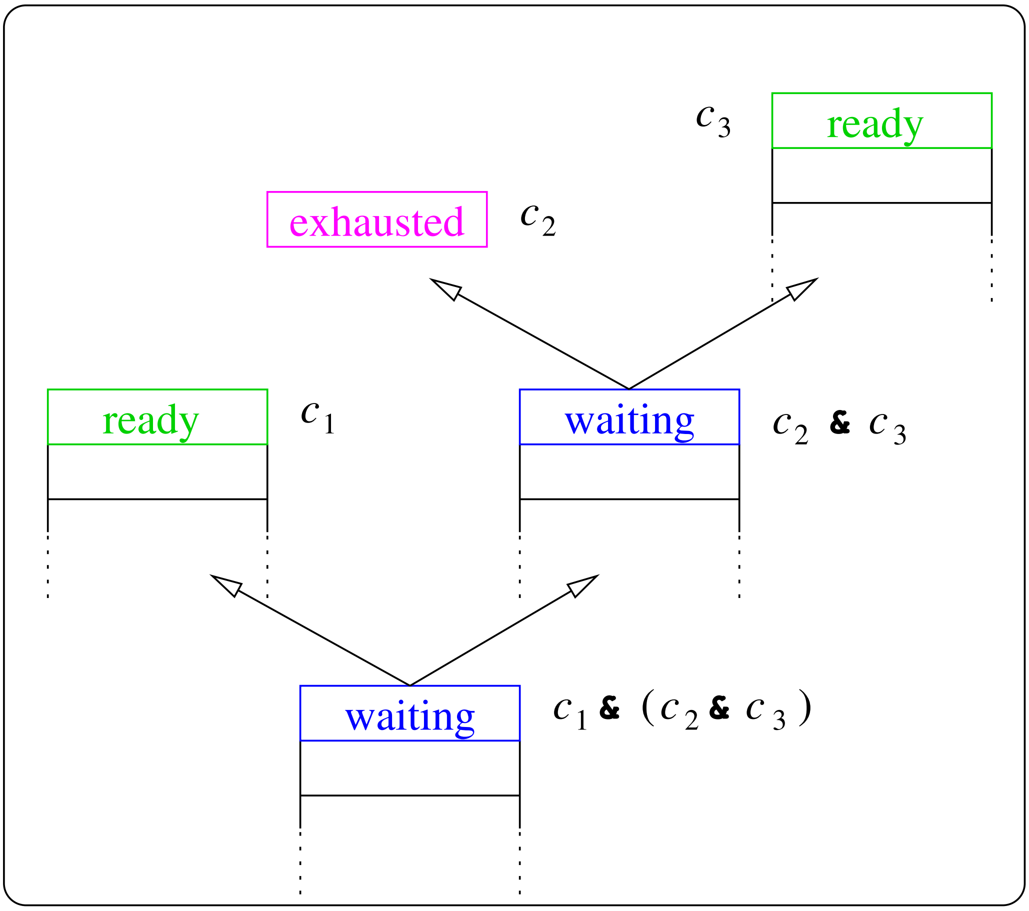
Completeness: Ring of computations (Java queue)

Concurrency

Parallel-and:

- concurrency *within one* computation
- c_1 & c_2: evaluate two constraints concurrently
- & can reduce if one c_i fails or both succeed

$\rightsquigarrow$ Call-tree:



- step waits on several children
- computation reduces leaves concurrently
- children wake up parent step

Residuation

Functional logic languages like Oz or Escher are based on *residuation*:

- suspend on unbound variables
- useful for constraint solving, process synchronisation
- incomplete strategy
- integration of narrowing and residuation (Curry, Hanus '97)

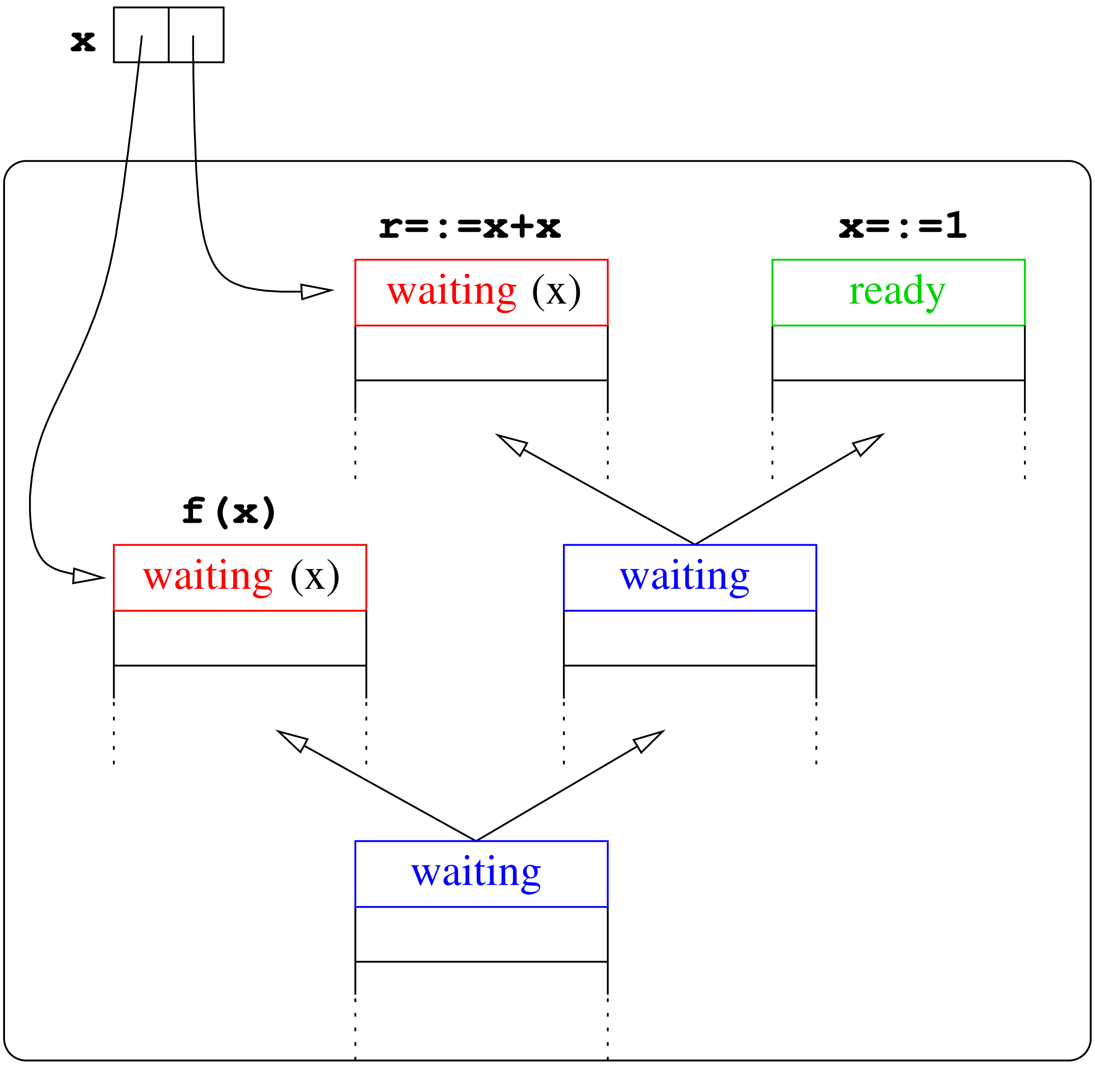
Example

```
r =:= x+x  &  x =:= 1
```

- code for `+` not known ⇒ suspend `x+x`
- `x=:=1` binds `x` ⇒ `x+x` wakes up

Implementation of Residuation

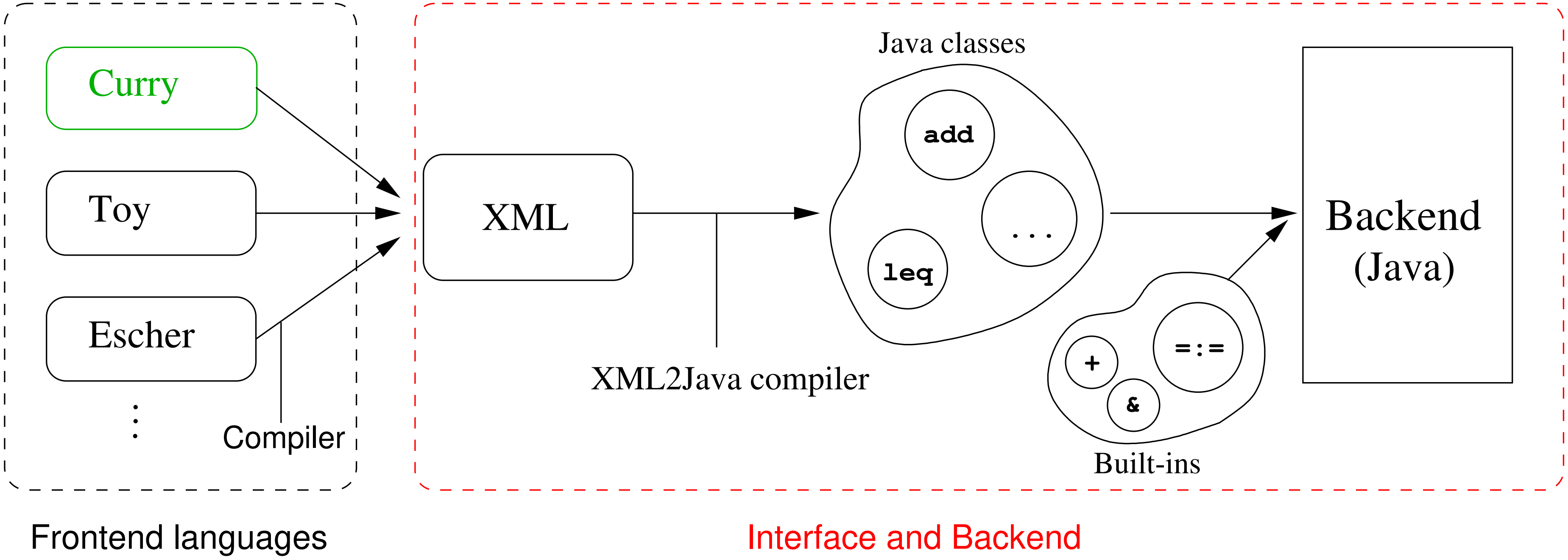
Extension: Steps flounder ($\widehat{=}$ wait on *variables*)



- residuating leaves are ignored
- variable stores floundering steps
- on bind: Just set state back to ready!

  $\rightsquigarrow$ no conceptual difference

A Generic Back End



- several frontend languages
- common interface: intermediate XML representation (case-expressions)
- frontend compilers fix strategy
- XML2Java compiler (written in Curry): one Java class per function

How We Compete

First runtime results (times in seconds):

	our back end	Curry2Java	MCC	Curry2Prolog	Jinni
$\texttt{rev}_{400}$	0.69	2.6 (≈ 4)			
$\texttt{rev}_{1200}$	5.5	N/A	0.69 (≈ -8)	0.68 (≈ -8)	45.9
$\texttt{add}_{300}$	2.1	16.2 (≈ 8)	0.12 (≈ -20)	0.09 (≈ -20)	2.5
engine	Java	AM in Java	AM in C	SICStus Prolog	Java
complete	$\surd$	$\surd$	$\otimes$	$\otimes$	$\otimes$

MCC:	Münster Curry Compiler (Lux)
Curry2Prolog:	$\in$ PAKCS (Portland Aachen Kiel Curry System) (Hanus, Antoy, . . .)
Curry2Java:	$\in$ PAKCS (Sadre)
Jinni:	Implementation of Prolog (Tarau)

$\Rightarrow$ fastest *complete* implementation of (needed) narrowing so far

Future Work

- add more built-ins (I/O, floats, ...)
- encapsulated search
- search strategies on top
- add external function interface
- connect other languages than Curry
- conceptual performance tweaks
- implementation in C

Conclusions

New approach for implementing FL languages:

- compilation to imperative language is feasible
- simple concept to combine narrowing, residuation and concurrency
- completeness
- generic back end for several languages

Easy to understand and maintain $\Rightarrow$ starting point for developers?

Backend: `http://nmind.cs.pdx.edu` (GPL)

XML: `http://www.informatik.uni-kiel.de/~curry/flat/`