CO-FINANCED by EUROPEAN COMMISSION

ICT for EU-India Cross Cultural Dissemination

An offline approach to narrowing driven partial evaluation

J. Guadalupe Ramos

DSIC, Technical University of Valencia
guadalupe@dsic.upv.es
www.dsic.upv.es/~guadalupe
(joint work with **Josep Silva** and **Germán Vidal**)

1) Domain specific languages
2) An example of a domain specific language
3) Partial Evaluation
4) Narrowing driven partial evaluation (NPE)
5) A new offline approach to NPE
6) Conclusion and future work

Domain Specific Languages

They are programming languages tailored for a specific domain

Domain Specific Language (DSL) e.g., latex, html, VHDL, etc.

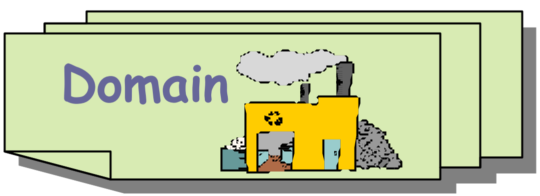
A DSL is at higher level than a conventional high level language

Advantages:

Reduced programming effort

- Applications with fewer lines of code
- Programs easier to reason about and maintain

Can be used by **non-expert programmers**



DSLs are a convenient technology both for the domain users, since they can easily **learn to programming real software applications** and for the DSL designer, in order **to teach the use of a new language**

Domain Specific Embedded Languages

But creating new languages is expensive (lexer, parser, and tools)

alternative

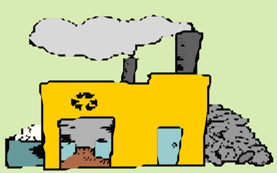
Domain Specific Embedded Language (DSEL)

A DSEL is implemented as a library in a "host" language

Host language with convenient features

- ☑ Higher order functions
- ☑ Syntax extension mechanisms
- ☑ Flexible/extensible type system
- ☑ Laziness

Domain



In this way, language tools are reused

The host language: Curry

- Curry does a strict distinction between (data) constructors and operations or defined functions on these data

- A Curry program consists of a set of type and function declarations

Curry built-in types (**Int, Bool, Char**, ...)

Data type declarations:

$$\texttt{data}\ T\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ C_1\ \tau_{11}\ \ldots\ \tau_{1n_1}\ |\ \cdots\ |\ C_k\ \tau_{k1}\ \ldots\ \tau_{kn_k}$$

```
data Boolean  = True | False
data Tree Int = Leaf Int | Node (Tree Int) Int (Tree Int)
```

Type synonym declarations:

$$\texttt{type}\ T\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ \tau$$

```
type Name = [Char]
type List a = [a]
```

The host language: Curry

A function is defined by a type declaration (which can be omitted)

$$f \; :: \; \tau_1 \; \text{->} \; \tau_2 \; \text{->} \; \cdots \; \text{->} \; \tau_n \; \text{->} \; \tau$$

followed by a list of defining equations $f \; t_1 \ldots \; t_n = e$ e.g.

```
append   []    y = y
append (x:xs) y = x : app xs y
```

functions

Higher order features:

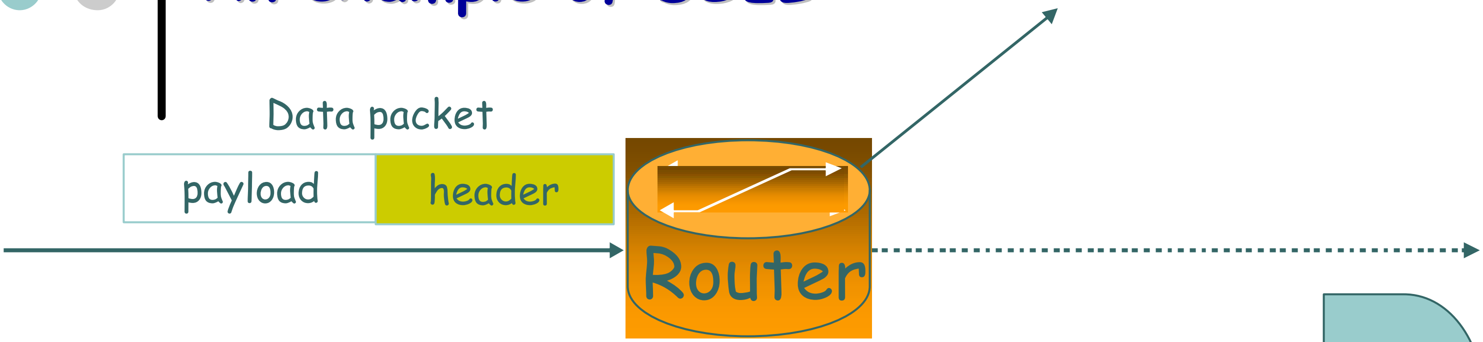
```
map f []     = []
map f (x:xs) = f x : map f xs
```

e.g., given `inc x = x + 1`

we use `map inc [4,9]`

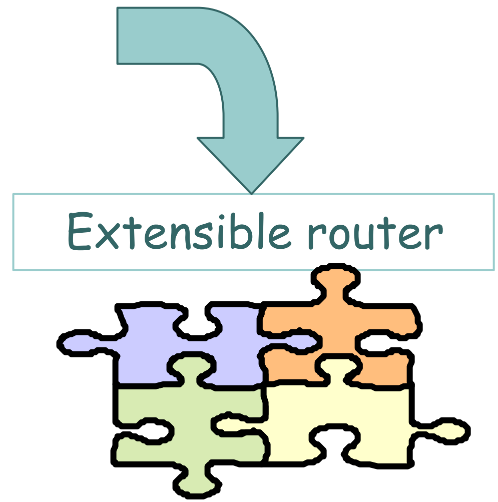
And it produces `[5,10]`

An example of DSEL



o A router is a special device that connects two or more networks and forward data packets between them

o Due to growing of networks (and Internet) there is a trend to extend the set of functions that routers should support (with run-time customization capabilities), giving rise to extensible routers

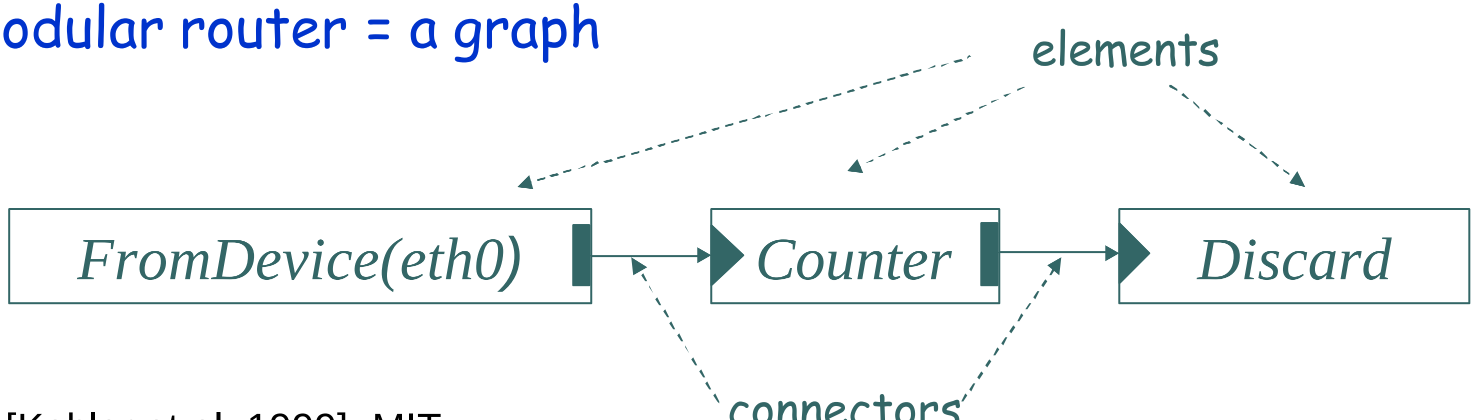


- Security
- Policies
- QoS
- Addresses
- Evolution

An example of DSEL

- Among extensible routers, Click is distinguished
- In Click, each functional aspect of a router is encapsulated in an element (an instance of a C++ class)
- A Click router is based on composing many elements to produce a system that implements the desired behavior

A modular router = a graph



- **Click** [Kohler et al. 1999], MIT

Rose: an example of DSEL for Router specification

In Rose, packet streams are:

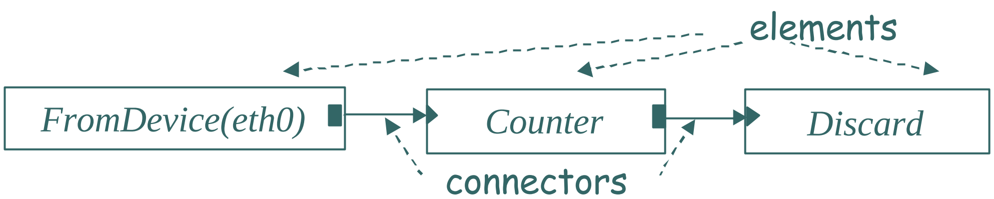
```
type Packet = [Int]
type Stream = [Packet]
```

We follow Click style,

i.e., router = a set of elements joined by connectors

Click elements in Rose are functions:

```
element :: [Conf] -> [Stream] -> [Stream]
```



A simple router:

```
simpR = seqOfe [fromDevice [Eth 0], counter [], discard []]
```

Using the connector:

Higher order

```
seqOfe :: [ [Stream] -> [Stream] ] -> [Stream] -> [Stream]
```

DSEL drawbacks

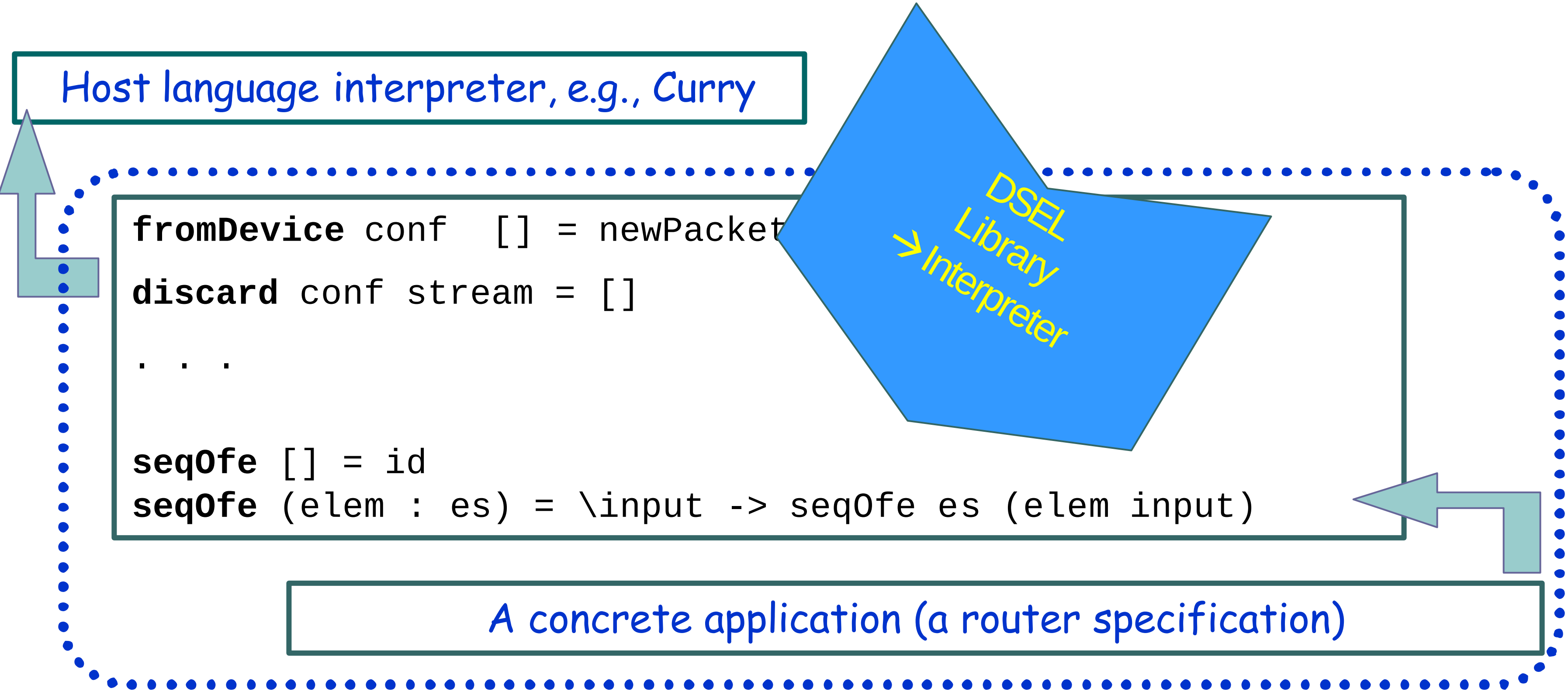
However, DSELs have the following problems:

- Host languages can not analyze DSEL data structures, e.g.,
  - They can not perform type checking
  - Error messages are related to host languages, not to DSELs
- The generated code is slow
  - Many interpretation layers

We are focused on the reduction of interpretation layers

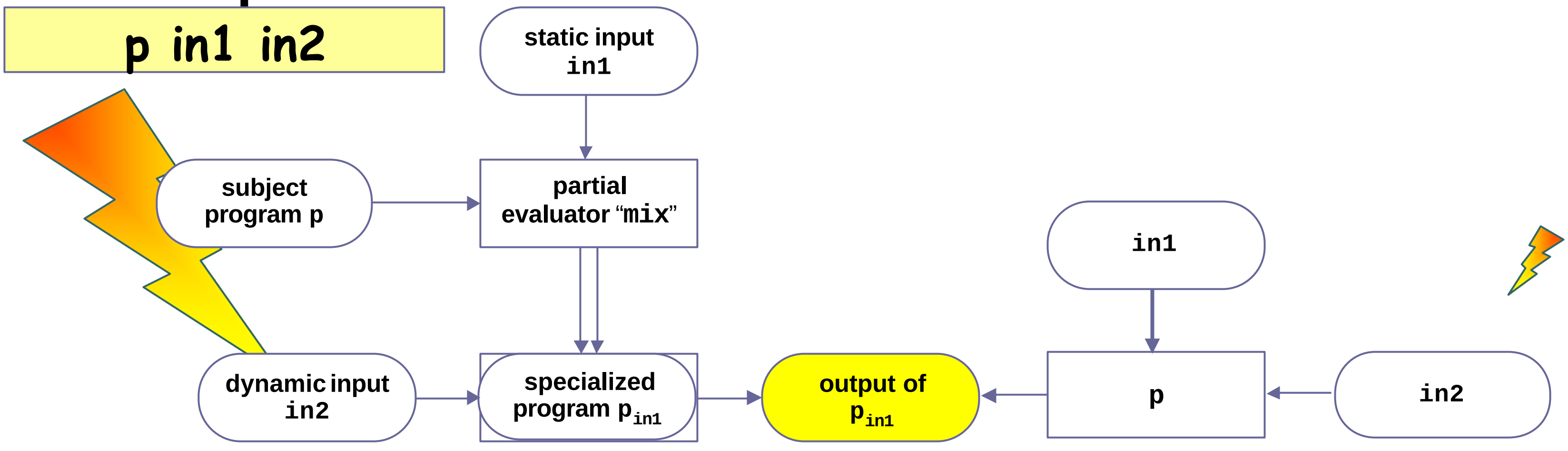
DSEL drawbacks

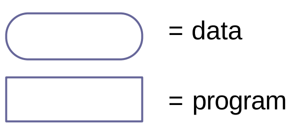
Interpretation layers



Solution: Partial evaluation of interpreters

Partial evaluation





The specialized program with the remaining data produces de same result as the original one with all data

For instance x^n:

```
power x n = if n == 0
               then 1
               else (x * (power x (n - 1)))
```

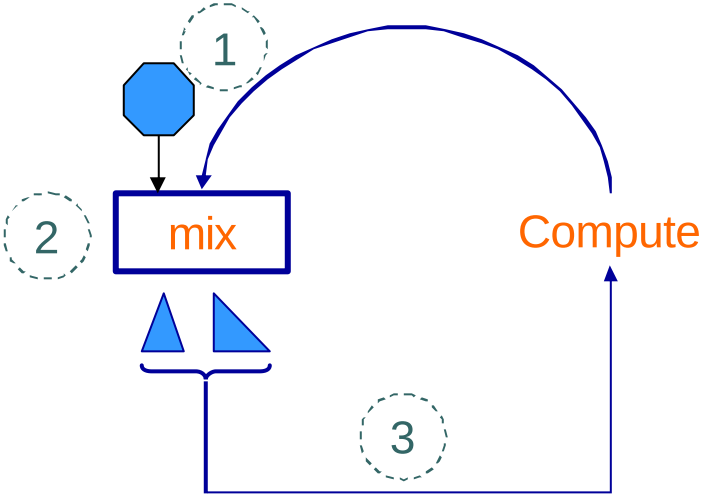
```
power x 3
```

```
power₃ x = x * x * x
```

Partial evaluation

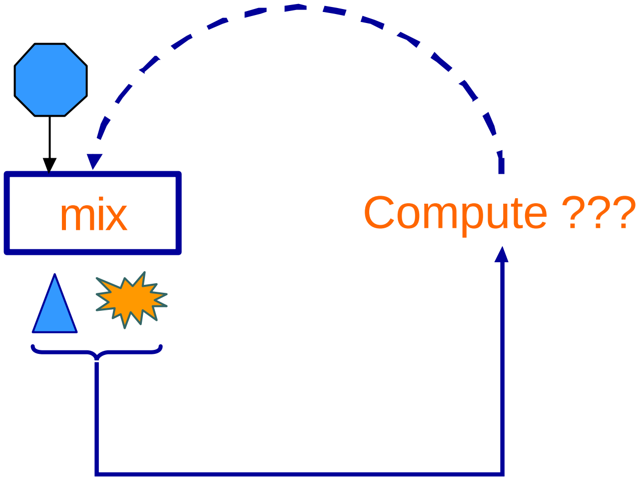
Partial evaluation is **a process that iteratively**

1. takes a function call,
2. performs some symbolic evaluations (e.g., **power x 3**), and
3. extracts from the partially evaluated expression the set of pending function calls to be computed in the next iteration of the process



Termination of partial evaluation

- It is not easy to identify which terms (function calls) should be processed.
- Some terms can produce infinite computations
- Usually, some form of generalization is applied to terms in order to stop infinite computations (reducing precision)
- When should dangerous terms be generalized?



Partial evaluators

The decision on which terms should be generalized can be taken online or offline

Online partial evaluators are

- more precise since they have more information available at partial evaluation time
- usually more expensive

Offline partial evaluators proceed in two stages

- The first stage returns an annotated program to guide the partial computations
- The partial evaluation stage only obeys the annotations
- Offline partial evaluators are faster but less precise than online partial evaluators

Narrowing driven partial evaluation

- In order to perform symbolic computations in a functional context, an extension of the standard semantics is required: **narrowing** (**basis of** the functional logic languages, as Curry)

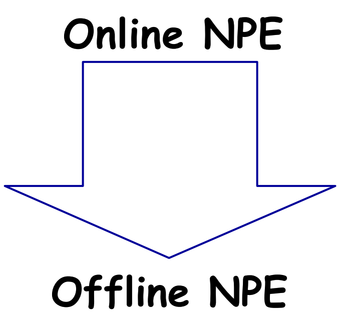
- **NPE (narrowing-driven partial evaluation)** is a powerful specializing scheme for first-order functional (logic) programs.

An online NPE tool is already integrated into the PAKCS environment for the declarative multi-paradigm language Curry

Narrowing driven partial evaluation

- In NPE, if a term embeds some previous one in the same computation (w.r.t. homeomorphic embedding), a form of generalization is applied and partial evaluation continues with the generalized terms
- Homeomorphic embedding tests together with the associated generalizations make NPE very expensive

- Although online NPE gives good results on small programs, it does not scale up well to realistic problems



An offline approach to NPE

- Is well known that, if the partial computations are **quasi-terminating**, i.e., they contain only a finite number of different function calls (modulo variable renaming)
- then, the partial evaluation process terminates (using a sort of **memoization**)

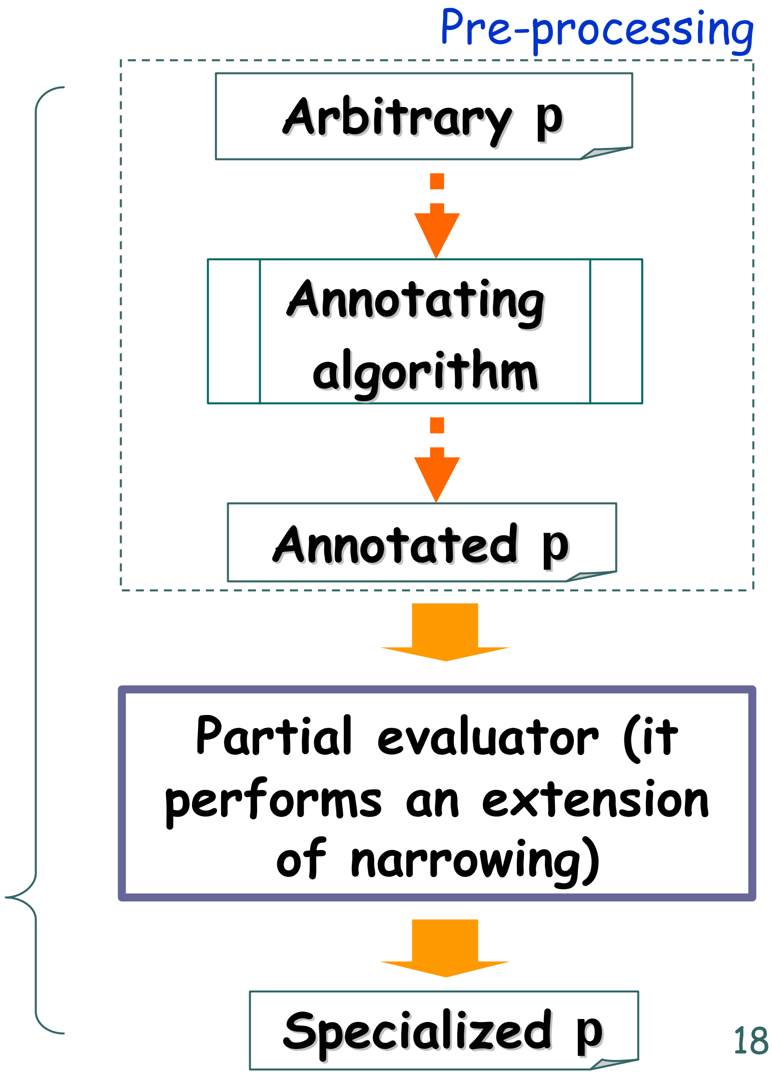
- Recently, at the International Conference on Functional Programming '05, we have introduced a syntactic characterization for programs (**nonincreasing programs)** that guarantees the quasi-termination of computations

very restrictive

An offline approach to NPE

- In order to accept more programs, we defined an **algorithm that annotates** those terms that cause non quasi-termination
- We presented an **extension of narrowing** which performs computations **generalizing** annotated terms

Our offline approach



A simple interpreter of arithmetic expressions

```
data Nat   = Z | S Nat | E
data Token = Cst Nat | Var Nat | Plus Token Token | Minus Token Token | Mult Token Token

int :: Token -> [Nat] -> [Nat] -> Nat
int (Cst   x  ) _    _     =  x
int (Var   x  ) vars vals  =  lookup  x  vars  vals
int (Plus  x y) vars vals  =  add     (int x vars vals) (int y vars vals)
int (Minus x y) vars vals  =  minus   (int x vars vals) (int y vars vals)
int (Mult  x y) vars vals  =  mult    (int x vars vals) (int y vars vals)

--- auxiliar functions
. . .
--- arithmetic engine
. . .

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

minus    x    Z  =  x
minus (S x) (S y)= minus x y

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

DSEL for arithmetic expressions

Programs are written indicating operations as Plus, Multiplication, Minus of constants (Cst) or variables (Var) of natural numbers

An application program to be specialized

```
main y = int (Plus (Cst (S (S Z))) (Cst y))   [] []
```

emacs-x@cmm2.dsic.upv.es

File Edit Options Buffers Tools In/Out Signals Help

In the first stage we apply the annotating algorithm

```
OffPeval> annotate "ictccd/simpleInt2"
Offline Narrowing-Driven Partial Evaluator
(Version 0.1 of July 2005)
(Technical University of Valencia)

(Pre-processing stage ... )

 Writing annotated program in <<ictccd/simpleInt2_ann.fcy>>

OffPeval> :l ictccd/simpleInt2_ann
Compiling 'ictccd/simpleInt2_ann.fcy' into Prolog program '/tmp/pakcsprog3851.pl'...

ictccd/simpleInt2_ann(module: simpleInt2)> :show
No source program file available, generating source from FlatCurry...

-- Program file: ictccd/simpleInt2_ann

data Nat = Z | S Nat | E
data Token = Cst Nat| Var Nat| Plus Token Token| Minus Token Token| Mult Token Token

main :: Nat -> Nat
main v0 = int (Plus (Cst (S (S (S Z)))) (Cst v0)) [] []

int :: Token -> [Nat] -> [Nat] -> Nat
int eval flex
int (Cst   v3    ) v1 v2 = v3
int (Var   v4    ) v1 v2 = lookup v4 v1 v2
int (Plus  v5 v6 ) v1 v2 = add  (GEN (int v5 v1 v2)) (GEN (int v6  v1 v2))
int (Minus v7 v8 ) v1 v2 = minus(GEN (int v7 v1 v2)) (GEN (int v8  v1 v2))
int (Mult  v9 v10) v1 v2 = mult (GEN (int v9 v1 v2)) (GEN (int v10 v1 v2))
. . .

add :: Nat -> Nat -> Nat
-u:**  *shell*        (Shell:run)--L1650--98%--------------------------------------
```

annotations for generalization

Aplicaciones Lugares Escritorio guadalu... emacs-x... emacs-x... guadalu... mié 26 de oct, 18:29

emacs-x@cmm2.dsic.upv.es

File Edit Options Buffers Tools In/Out Signals Help

The partial evaluation stage

Here we specialize the annotated program

```
OffPeval> mix "ictccd/simpleInt2"
Offline Narrowing-Driven Partial Evaluator
(Version 0.1 of July 2005)
(Technical University of Valencia)
(Partial evaluation stage ...)

Writing original program into "ictccd/simpleInt2_pe.fcy"...

OffPeval> :l ictccd/simpleInt2_pe
Compiling 'ictccd/simpleInt2_pe.fcy' into Prolog program '/tmp/pakcsprog3221.pl'...

ictccd/simpleInt2_pe(module: simpleInt2)> :show

-- Program file: ictccd/simpleInt2_pe
data Nat = Z | S Nat | E
data Token = Cst Nat | Var Nat | Plus Token Token | Minus Token Token| Mult Token Token

main :: b -> a
main v0 = add_pe1 int_pe2 (int_pe3 v0)

add_pe1 :: c -> b -> a
add_pe1 eval flex
add_pe1 Z v5 = v5
add_pe1 (S v304) v5 = S (add_pe1 v304 v5)

int_pe2 :: a
int_pe2 = S (S (S Z))

int_pe3 :: b -> a
int_pe3 v0 = v0
-- end of module ictccd/simpleInt2_pe

ictccd/simpleInt2_pe(module: simpleInt2)> main (S (S Z))
Result: (S (S (S (S (S Z))))) ?
-u:**  *shell*        (Shell:run)--L843--Bot----------------------------
```

The specialized program is shorter than the original interpreter and application

testing

Aplicaciones Lugares Escritorio guadalupe@c... emacs-x@cm... dom 30 de oct, 17:35

Benchmarks

benchmark	codesize (bytes)	onlineNPE (ms.)	speedup1 (online)	offlineNPE		speedup2 (offline)
				ann (ms.)	mix (ms.)	
ackermann	1496	20290	1.006	100	590	4.750
allones	1191	180	1.065	50	200	1.050
fliptree	1861	1940	0.985	100	240	0.977
foldr.allones	2910	3633	1.024	120	430	2.034
foldr.sum	3734	6797	1.311	170	3340	1.293
fun_inter	4266	28955	—	160	5190	—
gauss	1241	11090	1.040	100	757	1.013
kmp_matcher	3222	11670	5.346	157	9410	1.219
power	1693	160	3.087	110	280	1.012
Average	**2402**	**9413**	**1.858**	**119**	**2271**	**1.668**

speedup = orig/spec

Advantages

- The offline partial evaluation time is a 25% of the online partial evaluation time
- The tool is able to process bigger programs than online approach

Disadvantages

- Less precision, runtimes of the offline specialized programs are a 10% slower than online

Conclusion & future work

- DSLs are an appropriate tool for teaching an introducing the non expert programmers in domain specific solutions of software by means of programming languages

- The offline approach to narrowing driven partial evaluation scale up better to realistic programs

- Preliminary experiments (for specialization of DSELs) have been performed with a partial evaluation prototype which follows the offline scheme and the results are promising

Future work

- Include support for a broad set of Curry features
- Introduce a binding-time analysis