Building DSLs 2

Pat Hanrahan

CS343d

Fall 2021

Recap

External (Extrinsic) DSL

- Standalone language
- e.g. `matlab, R`

Embedded (Intrinsic) DSL

- Embedded in a host language
- e.g. `pytorch, tensorflow`

Recap

Shallow Embedding

- Runs directly in the host language

Deep Embedding

- Represents the DSL as an AST and compiles/interprets that AST

PL Features for DSLs

Types

- Algebraic data types for creating ASTs
- Parameterized types and polymorphism
- Metaclasses

Higher order functions and lambdas

Metaprogramming

Flexible and extensible syntax

Today's Topics

Macros

Functors

Dependent types

Partial evaluation

Next week: Notation

Macros

Relevance

Macros are programs evaluated at compile-time, not run-time (staged program)

Resurgence in interest in providing macros for programming languages (Terra, Rust, ...)

Text macros (e.g. cpp) vs Lisp macros.

```
% cpp | gcc -E
#define NULL 0
#define SQUARE(x) x*x

SQUARE(1+2)

// results in
1+2*1+2

// defensive programming
#define SQUARE(x) ((x)*(x))

// cpp is not “aware” of C
```

```
// conditional macros

// expressions?

#ifdef LINUX
…
#endif

// expressions in predicate?
#if defined(LINUX) …
#if VERSION > 1.0 …
```

```
# prep: uses python mako templating engine
<%
n = 10
ns = range(10)
%>\
#include <stdio.h>

void main(void) {
  int a = ${n};
% for i in ns:
%   if i != 5:
  ${i};
%   endif
% endfor
}
```

Lisp Macros

```
# lisp/scheme s-expressions

$ racket
> (+ 1 2)
3
> (* (+ 1 2) 4)
12
> (define x 5)
> (/ 10 x)
2
> (define (square x) (* x x))
> (square 5)
25
```

```
; homoiconic: lists = code|data

> (define l (list 1 2 3))
> l
'(1 2 3)
> (car l)
1
> (cdr l)
'(2 3)
> (cadr l)
2
```

```
; special forms

; normally function arguments
; are evaluated left-to-right
; before the function is called

; sometimes function arguments
; are evaluated differently.
; these functions are special forms
(define x 2)
(if cond true-expr false-expr)
(or expr1 expr2)
(for [(i 10)] (displayln i))
```

Macro Definitions in Lisp

Timothy Harris

Abstract

In LISP 1.5 special forms are used for three logically separate purposes: a) to reach the alist, b) to allow functions to have an indefinite number of arguments, and c) to keep arguments from being evaluated. New LISP interpreters can easily satisfy need (a)

```
; quote

> (+ 1 2)
3
> (quote (+ 1 2))
'(+ 1 2)
> (list '+ 1 2)
'(+ 1 2)
; notation
> '(+ 1 2)
'(+ 1 2)
> (eval '(+ 1 2 3))
6
```

```
# meta-programming
#  implement (when pred expr)

> (define (convert whenlist)
   (list 'if (nth whenlist 1)
             (nth whenlist 2)
             (void)))
> (define s '(when (> 2 1)
              (display "true\n")))
> (convert s)
'(if (> 2 1) (display "true\n") #<void>)
> (eval (convert s))
true
```

```
l quasiquote

> (define x 2)
> (quasiquote (+ 1 x))
'(+ 1 x)
> (quasiquote (+ 1 (unquote x)))
'(+ 1 2)
> (quasiquote (+ 1 (unquote x)
                   (unquote-splicing '(2
2))))
'(+ 1 2 2 2)

; short-hand (note backquote `)
> `(+ 1 ,x ,@(list 2 2))
'(+ 1 2 2 2)
```

```
# meta-programming
#  implement (when pred expr)

> (define (convert when)
   `(if ,(nth when 1)
        ,(nth when 2)
        ,(void)))
> (define s '(when (> 2 1)
                  (display "true\n")))
> (convert s)
'(if (> 2 1) (display "true\n") #<void>)
> (eval (convert s))
true
```

```
# macros

> (define-macro (when test expr)
    `(if ,test ,expr ,(void)))

> (when (> 2 1) 1)
1

; 1. the arguments to the macro
;    are NOT evaluated (they are quoted)
; 2. The returned list is evaluated


; it's that simple!
```

Extensions

Hygienic macros

- Variable capture
- ...

Syntax macros

- ...

Terra

Zach DeVito

Terra is meta-programmed from Lua

Evaluation Semantics

```
function gen_square(x)
  return `x * x
end
```

In Lua, a quotation creates a Terra expression.

```
terra mse(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end
```

In Terra, an escape splices the value of a Lua expression into Terra code.

Evaluation Semantics

```
print("lua execution")
> lua execution
function gen_square(x)
  return `x * x
end
```

1. Lua code evaluates normally until it reaches a Terra function or quote expression

2. The Terra expression is specialized, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end


print(mse(3,2))
```

Evaluation Semantics

Evaluation Semantics

1. Lua code evaluates normally until it reaches a Terra function or quote expression

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

2. The Terra expression is specialized, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float): float
  return [      `a * a] - [gen_square(b)]
end


print(mse(3,2))
```

Evaluation Semantics

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end


terra sqd(a: float, b: float)
  return [       `a * a ] - [ `b * b        ]
end


print(mse(3,2))
```

1. Lua code evaluates normally until it reaches a Terra function or quote expression

2. The Terra expression is specialized, by evaluating all *escaped* Lua expressions.

Evaluation Semantics

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

1. Lua code *evaluates* normally until it reaches a Terra function or quote expression

2. The Terra expression is *specialized*, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float)
  return     a * a   -   b * b
end


print(mse(3,2))
```

Evaluation Semantics

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

1. Lua code evaluates normally until it reaches a Terra function or quote expression

```
terra sqd(a: float, b: float)
  return          a * a    -    b * b
end
```

2. The Terra expression is specialized, by evaluating all *escaped* Lua expressions.

```
print(mse(3,2))
> 5
```

3. The Terra function is evaluated as Terra

Lua

- Dynamically-typed, polymorphic
- Garbage collection
- Efficient interpreter (LuaJIT)

Terra

- Statically-typed, monomorphic
- Manual memory management
- Staged (JIT) compilation via LLVM

Terra-Lua system

- Lua meta-programs Terra
- Similar syntax, shared lexical state
- Co-embedded languages (call back/forth)

References

Paul Graham, On Lisp
(http://www.paulgraham.com/onlisp.html)

Doug Hoyte, Let over Lambda
(https://letoverlambda.com/index.cl/toc)

Racket macros
(https://docs.racket-lang.org/guide/macros.html)

Terra
(https://terralang.org/)

Functors

Array Language

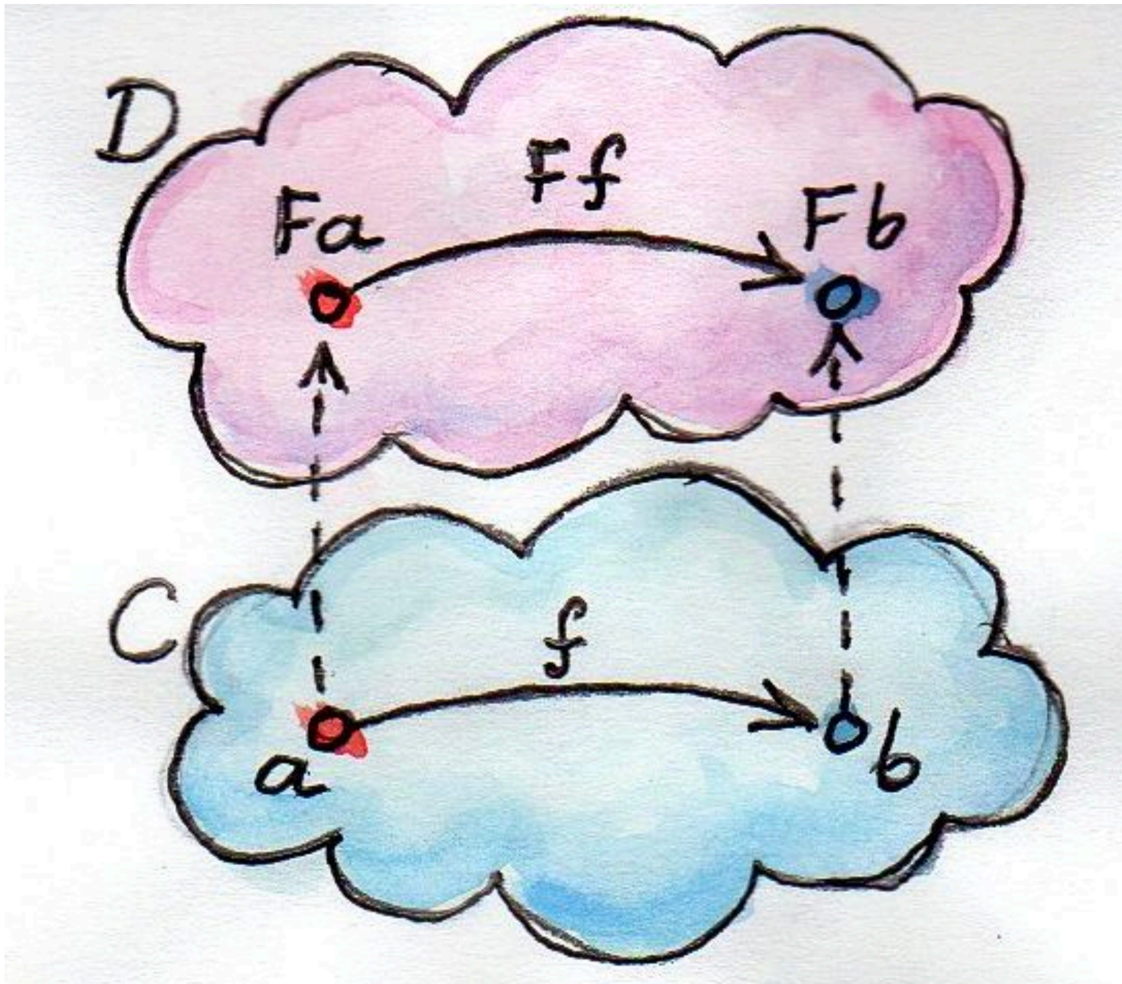
A way to create a DSL ...

Define a type Array[T]

- A vector of elements of type T
- All the operations on T apply to Array[T]
  - e.g. a, b: Array[Float], a+b is allowed

Functors F(a)

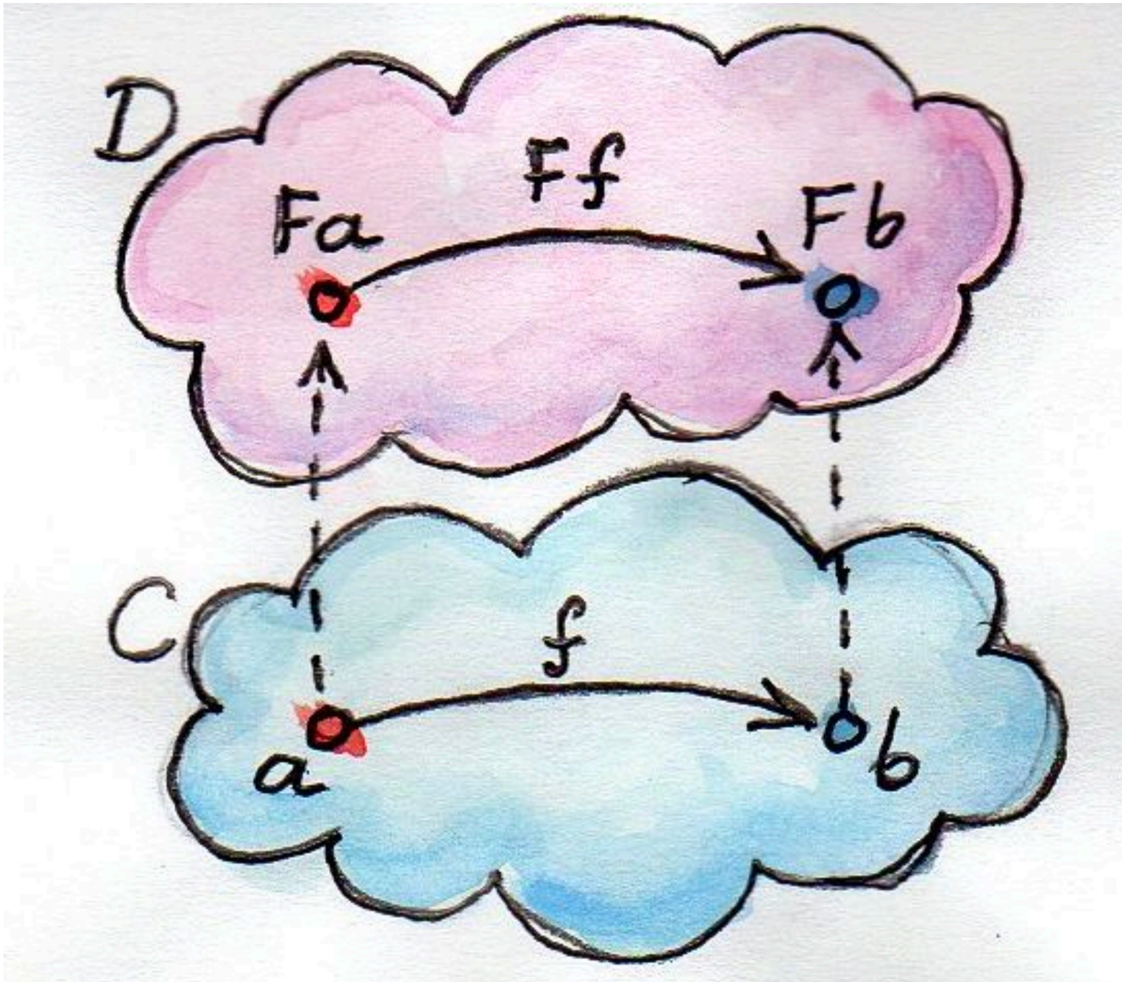
F is a function that maps type a to type b



https://bartoszmilewski.com/2015/01/20/functors/

Functors F(a)

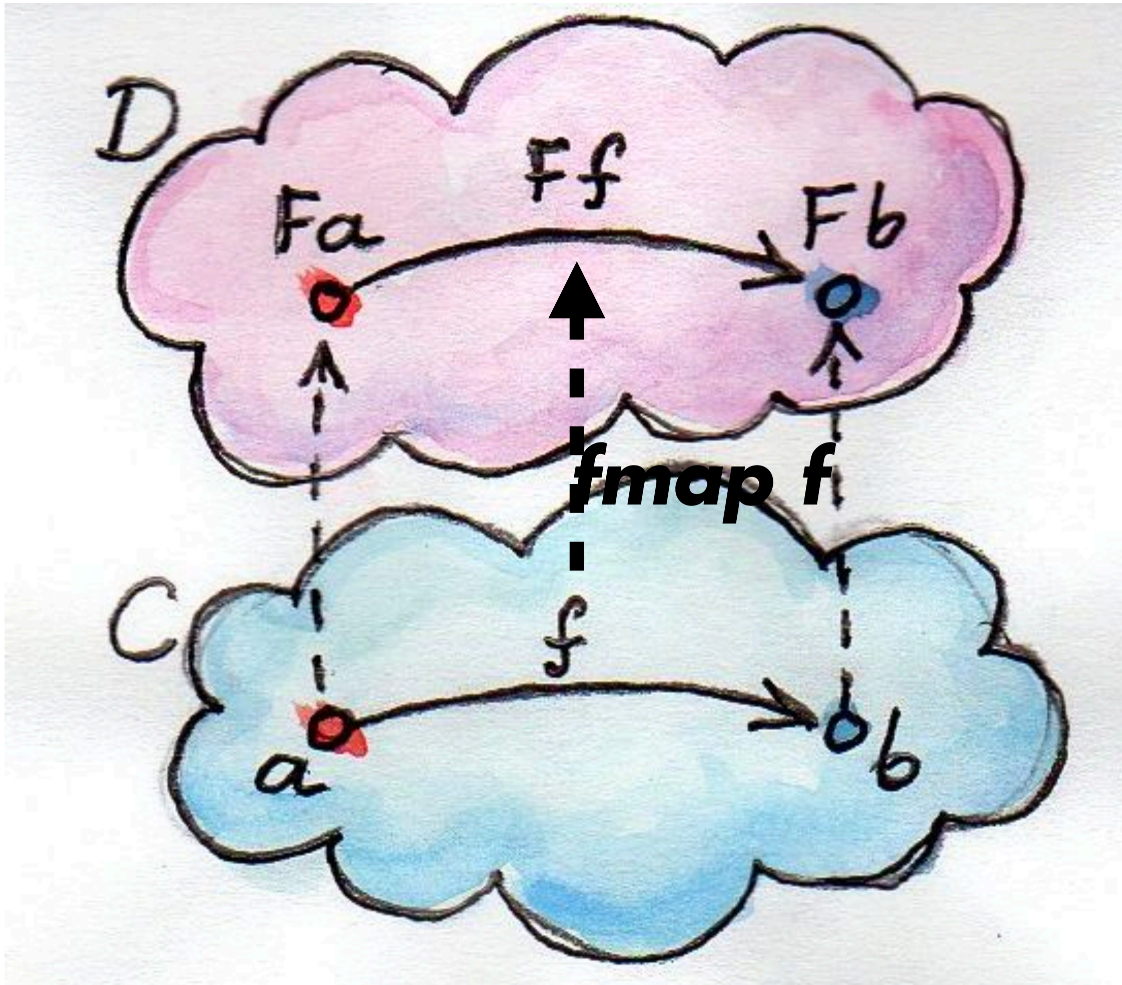
F is a function that maps type a to type b



b = f (a)

f maps values of type a to values of type b

Functors F(a)




```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

```
data List a = Nil | Cons a (List a)

fmap :: (a -> b) -> (List a -> List b)
fmap f Nil = Nil
fmap f (Cons x xs) = Cons (f x) (map f xs)

– Must obey the Functor Laws …
fmap id = id
fmap f ○ g = fmap f ○ fmap g

– example
add = fmap (curry (+))
sub = fmap (curry (-))
```

```
# kore - python implementation of k
import operator
from kore import every # recursive map

neg = every(operator.neg)
add = every(operator.add)
sub = every(operator.sub)
mul = every(operator.mul)
div = every(operator.div)
floordiv = every(operator.floordiv)
mod = every(operator.mod)
min2 = every(min)
max2 = every(max)

sign = every(lambda x: x if x ==0 else -1 if x
< 0 else 1)
…
```

Array[T, n]

What if you want to parameterize an array by its length?

Can't be done using today's type systems!

Need dependent types

```
– A dependent type is a parameterized type that
– depends on a value (not just other types)

data Vec (A : Set) : Nat → Set where
    [] :Vec A 0
    _::_ : {n : Nat} → A → Vec A n → Vec A (n+1)

infixr 5 _::_

– A dependent function type is where the type of
– the output can be different depending on
– the runtime value of the input type parameters.

_++_: Vec A m → Vec A n → Vec A (m + n)
[] ++ Vec ys = ys
(x :: xs) ++ Vec ys = x :: (xs ++ Vec ys)
```

Generators and DSLs

Dependent types allow you to write generators that depend on values, not just types.

```
gen_mux : T->int->((Vec T n)->(Vec int (clogn n))->T)

gen_Mux T n = …
```

A type can define a language via an AST type, and values of the type are programs in that language.

Dependent types allow you to create type-safe interpreters.

References

Dependent Types at Work
Ana Bove and Peter Dybjer

Programming and Proving in Agda
Jesper Cockz

Programming Language Foundations in Agda,
Philip Wadler, Wen Kokke, and Jeremy Siek
(https://plfa.github.io/)

Certified Programming with Dependent Types
Adam Chilipala
(http://adam.chlipala.net/cpdt/)

Partial Evaluation

(Specialization)

Partial Evaluation

Partial evaluation takes a function with some known and some unknown inputs.

```
partial :: (known -> unknown -> output)
        -> known
        -> (unknown -> output)
```

It converts a general function to a specialized function

A two input program

p =

```
a(m,n) = if m = 0 then n+1 else
              if n = 0 then a(m-1,1) else
              a(m-1,a(m,n-1))
```

Program p, specialized to static input m = 2:

p_2 =

```
a2(n) = if n=0 then a1(1) else a1(a2(n-1))
a1(n) = if n=0 then a0(1) else a0(a1(n-1))
a0(n) = n+1
```

See Partial Evaluation and Automatic Program Generation, Neil Jones, Carsten Gomard, Peter Sestoft (https://www.itu.dk/people/sestoft/pebook/pebook.html)

Techniques

1. Constant folding
2. Loop unrolling / unfold functions
3. Remove conditionals
4. Function inlining

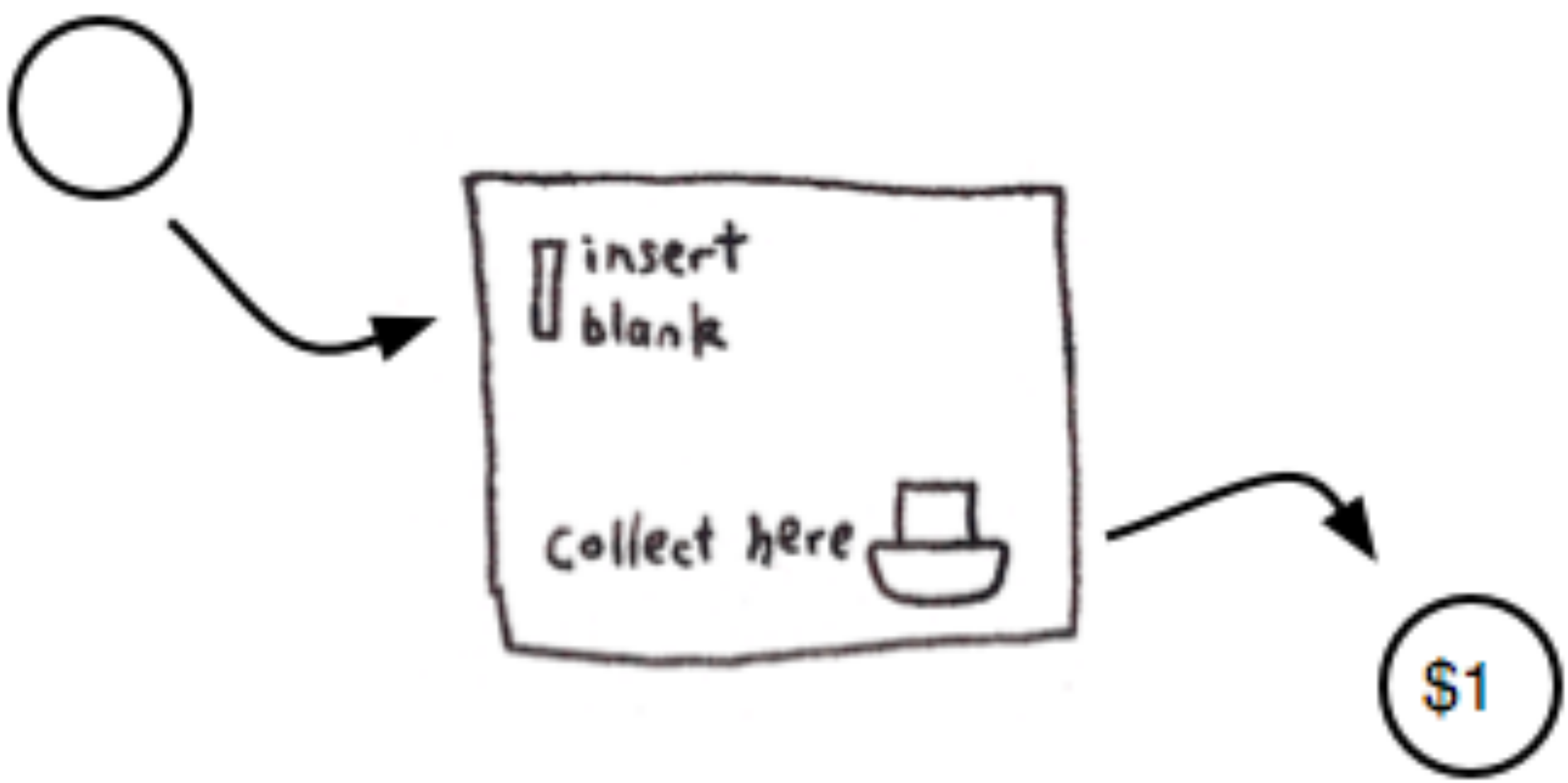
The Three Projections

of

Doctor Futamura

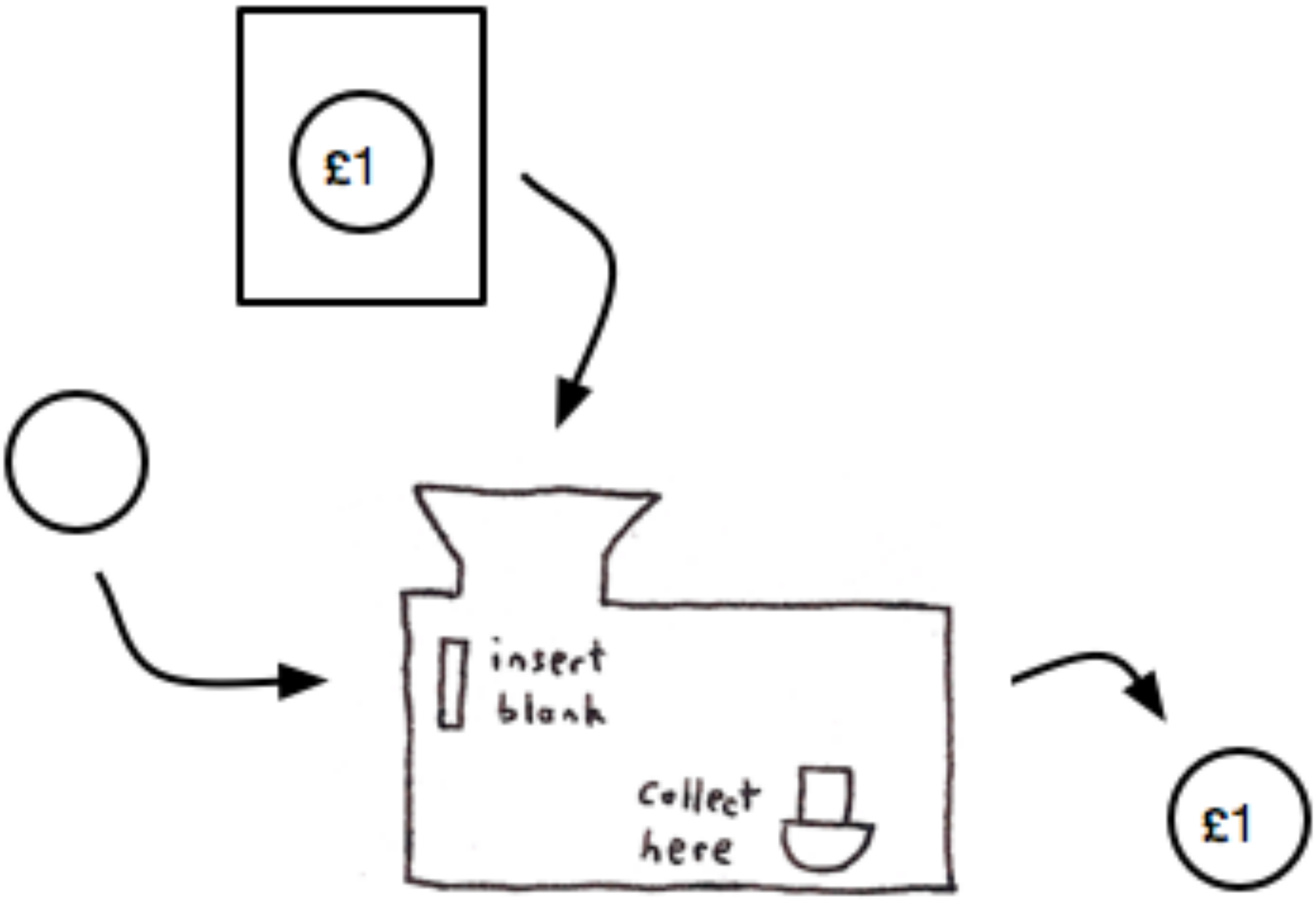
http://blog.sigfpe.com/2009/05/three-projections-of-doctor-futamura.html

Specialized Machine



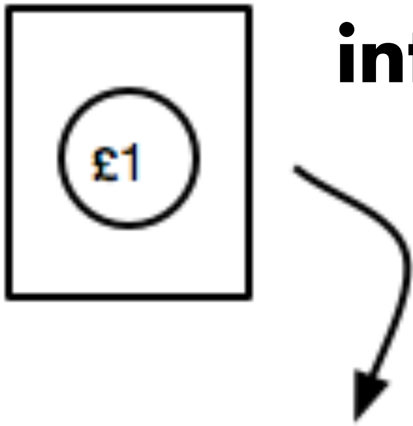

```
machine :: input -> output
```

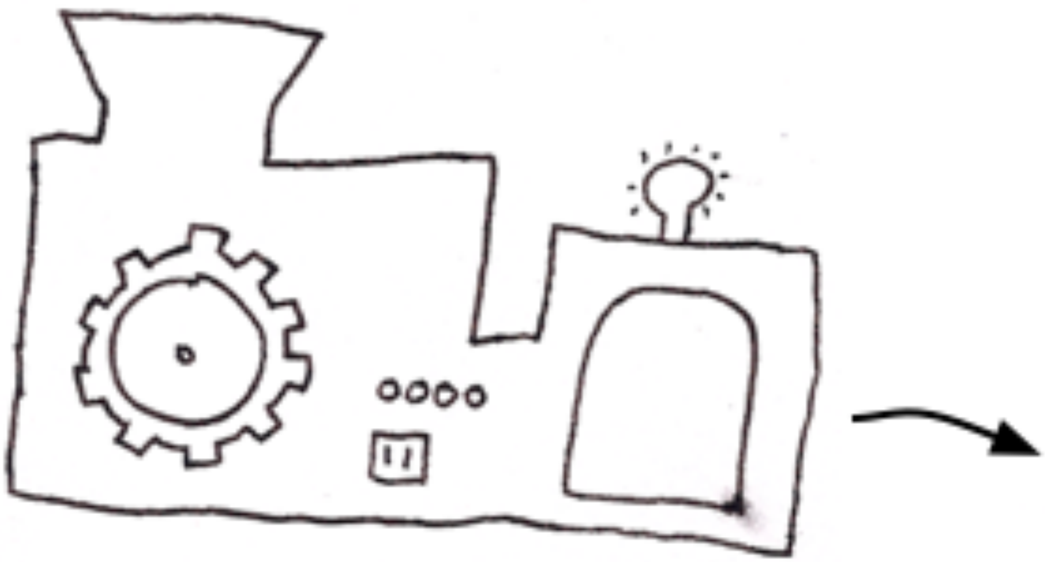
Programmable (CNC) Machine

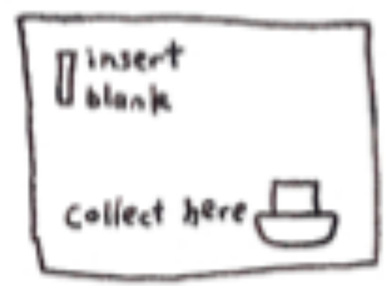



```
interpreter :: program -> input -> output
```

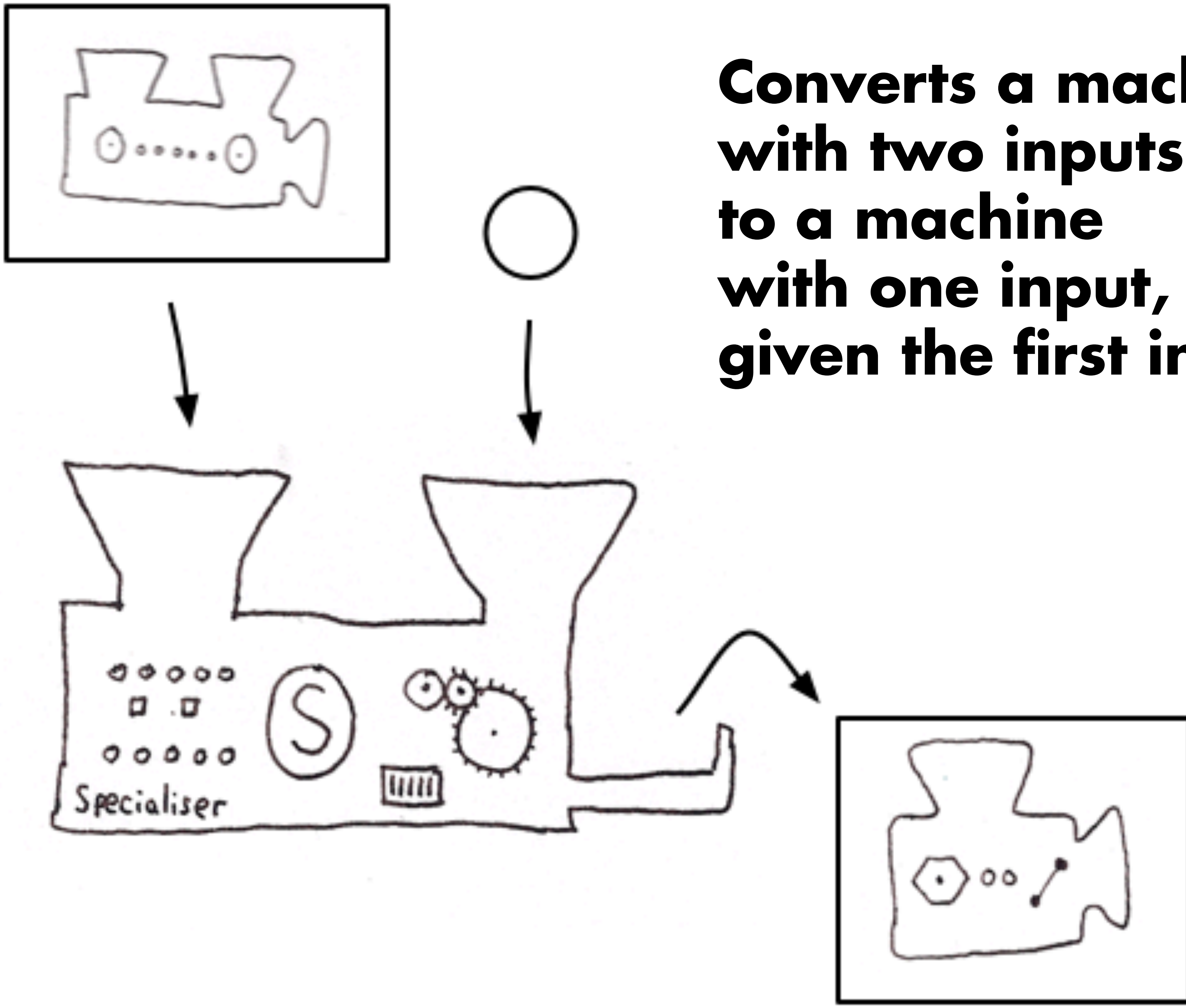
Convert a Programmable Machine into a Specialized Machine







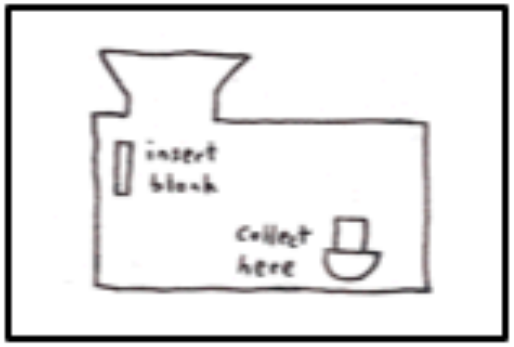
Compiling the program!

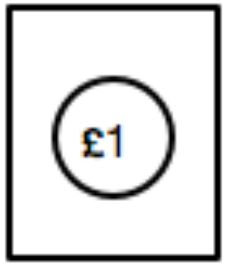


Converts a machine with two inputs to a machine with one input, given the first input

```
specializer : (input1 -> input2 -> output) -> input1 -> (input2 -> output)
```

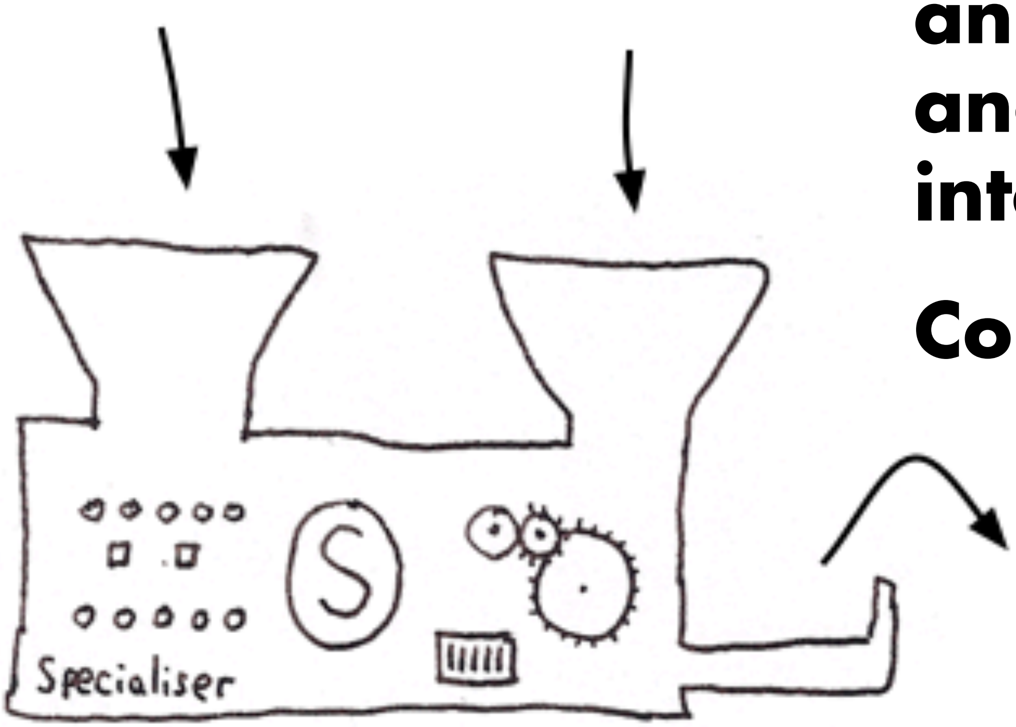
Dr. Futamura's First Projection

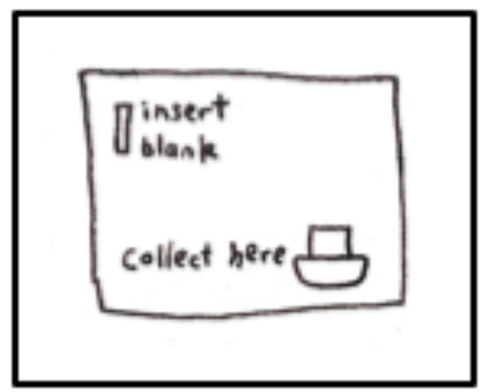




Use a specializer to convert an interpreter and a program into a specialized program

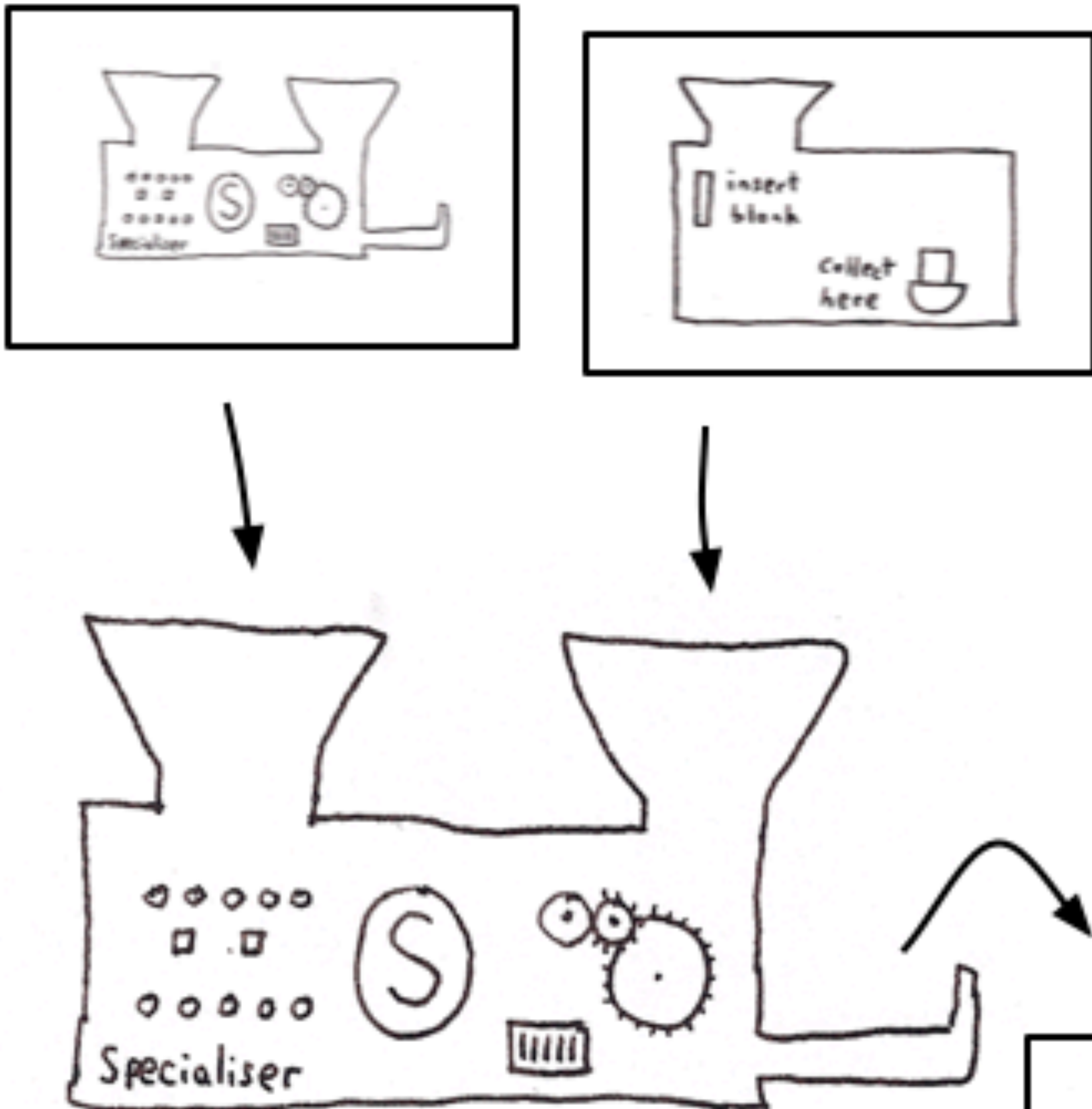
Compiling the program!





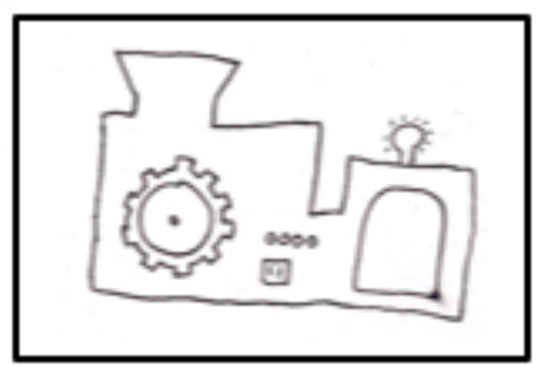

```
compiled = specializer interpreter program
```

Dr. Futamura’s Second Projection

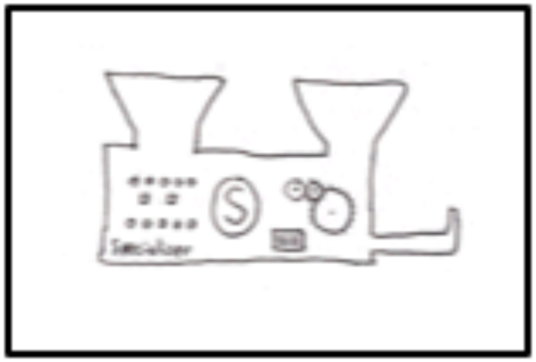


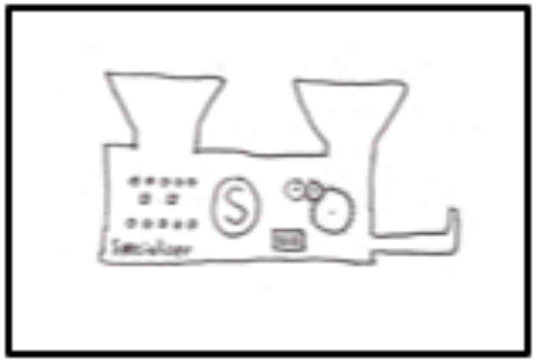
Use a specializer to convert a specializer and interpreter into a specialized specializer for this interpreter

Create a Compiler!



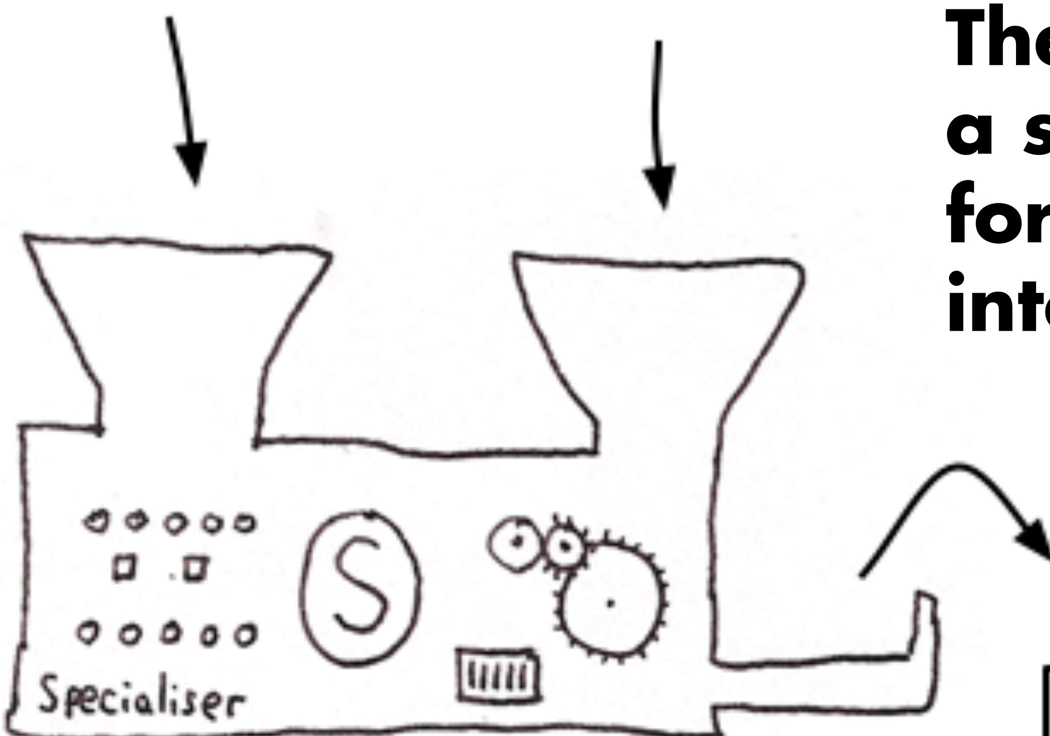
```
compiler = specializer specializer interpreter
```



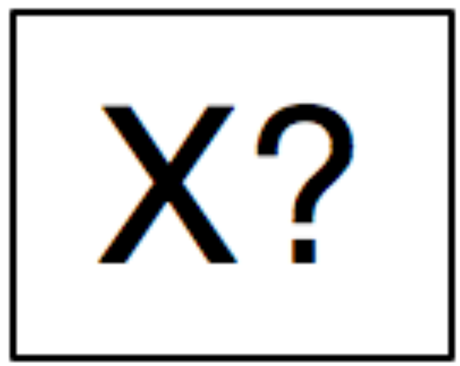


Dr. Futamura's Third Projection

The output X is a specializer optimized for converting interpreters into compilers



A compiler compiler!




```
compiler* = specializer specializer specializer
```

References

Partial Evaluation and Automatic Program Generation, Neil Jones, Carsten Gomard, Peter Sestoft (https://www.itu.dk/people/sestoft/pebook/pebook.html)

Finally Tagless, Partially Evaluated Tagless Staged Interpreters for Simpler Typed Languages Jacques Carette, Oleg Kiselyov and Chung-chieh Shan

AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries
Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt
Proceedings of the ACM on Programming Languages (PACMPL), 2(OOPSLA), 2018. (HiPEAC 2018 Paper Award)

Doctor Futamura’s Three Projections

1. Compiling specific programs to specialized machines.
2. Making a compiler from an interpreter.
3. Making a compiler compiler for converting interpreters into compilers.

Summary

Mechanisms that makes it easier to create DSLs

- **Macros**
- **Functors**
- **Dependent types**
- **Partial evaluation**

The C# Programming Language

Third Edition

Anders Hejlsberg

This book, too, is in its third edition. A complete technical specification of the C# programming language, the third edition differs in several ways from the first two. Most notably, of course, it has been updated to cover all the new features of C# 3.0, including object and collection initializers, anonymous types, lambda expressions, query expressions, and partial methods. Most of these features are motivated by support for a more functional and declarative style of programming and, in particular, for Language Integrated Query (LINQ), which offers a unified approach to data querying across different kinds of data sources. LINQ, in turn, builds heavily on some of the features that were introduced in C# 2.0, including generics, iterators, and partial types.

B

C#'s Functional Journey

Mads Torgersen, Microsoft

Transcript

Torgersen: I'm Mads Torgersen. I am the current lead designer of C#. I've been that for a good half decade now, and worked on the language for about 15 years. It's just a bit older than that, about two decades old. During that, it's gone through a phenomenal journey of transformation. Started out as a very classic, very turn of the century mainstream object-oriented language, and has evolved a lot. Many of the things that happened over time, were inspired/borrowed/stolen from the functional world. There's been a lot of crossover there.

Recor

FEB 2

by

REL

https://www.infoq.com/presentations/c-sharp-functional-features/