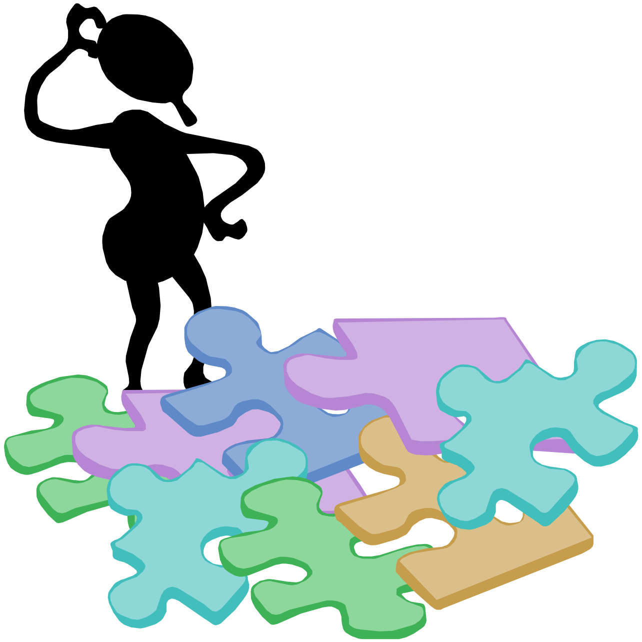
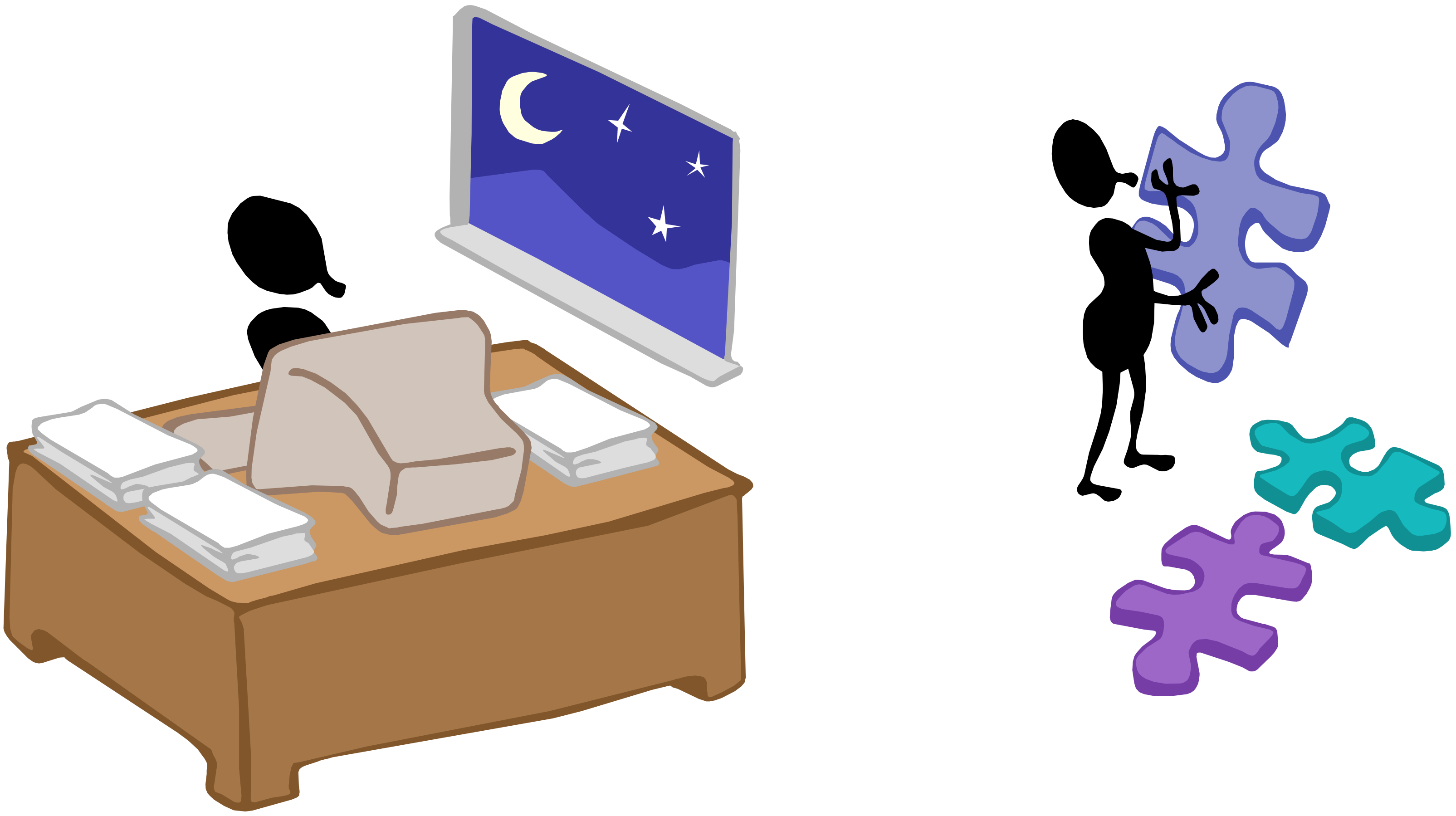
Systematic search for functional programs

Susumu Katayama
University of Miyazaki

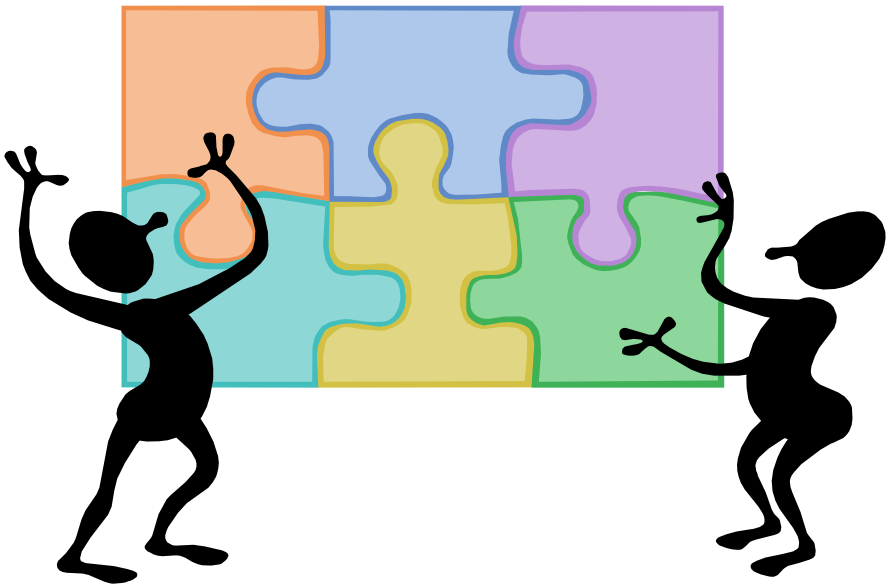
Exploiting strong typing



Exploiting strong typing



Exploiting strong typing



Inductive synthesis of functional programs

Inductive synthesis of functional programs (Inductive functional programming)
automatic generation of **functional programs** from **examples/properties**.

examples (I/O pair)
{("dsl",3), ("ak",2)}

or

properties (predicate)
f s.t. f "dsl" == 3 &&
f "ak" == 2

→

functional program

f "" = 0
f (c:s) = 1 + f s

* Properties can be more flexible than examples.

The implemented system

Please watch the demo….

Inductive synthesis of functional programs

Inductive synthesis of functional programs (Inductive functional programming):
automatic generation of **functional programs** from **examples/properties**.

examples (I/O pair)
{("dsl",3), ("ak",2)}

or

properties (predicate)
f s.t. f "dsl" == 3 &&
f "ak" == 2

functional program

f "" = 0
f (c:s) = 1 + f s

* Properties can be more flexible than examples.

Human programmers usually:

could be automated

Write a function → review it → test it

You should still do that!

Outline

- **Introduction**

  motivation, policy, & conventional approaches

- **The implemented system**

  rough specification & detailed implementation

- **Experiments**

  results on some easy problems

- **Conclusions**

Placement

- “Deductive” synthesis from complete spec.
- “Inductive” synthesis from incomplete spec.
  - by generating & folding computational traces from examples
  - search-based, or by trial and error
    - Genetic Programming (GP)
    - The proposed method

related work(1): Synthesis via computational traces

- Step 1: Generate computational traces
  - by constraints on examples (Summers 1977)
  - by hand (a.k.a. programming by demonstration (PBD))
  - by a universal planner (Schmid 2001)
  - by genetic programming (Schmid 2004)

  by search
- Step 2: Fold the traces into a recursive program
  - by pattern matching

Drawback: **unfolded traces are longer than programs**

search space bloats much earlier

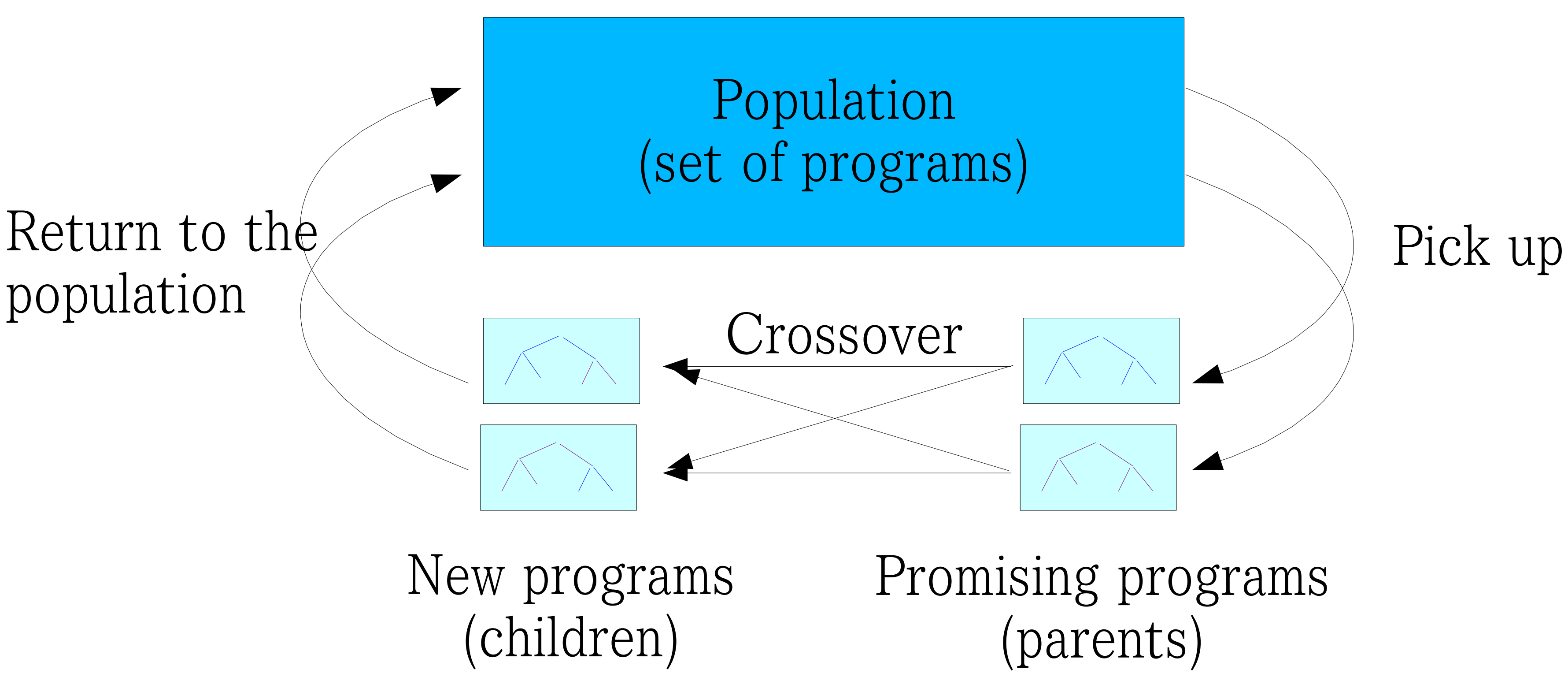
→ (than direct search methods like GP)

related work(2): *Genetic programming* (1/3)

assumption:

"useful programs consist of useful subexpressions"

Search by recombination of subexpressions



related work(2): *Genetic programming* (2/3)

GP applied to inductive algorithm synthesis

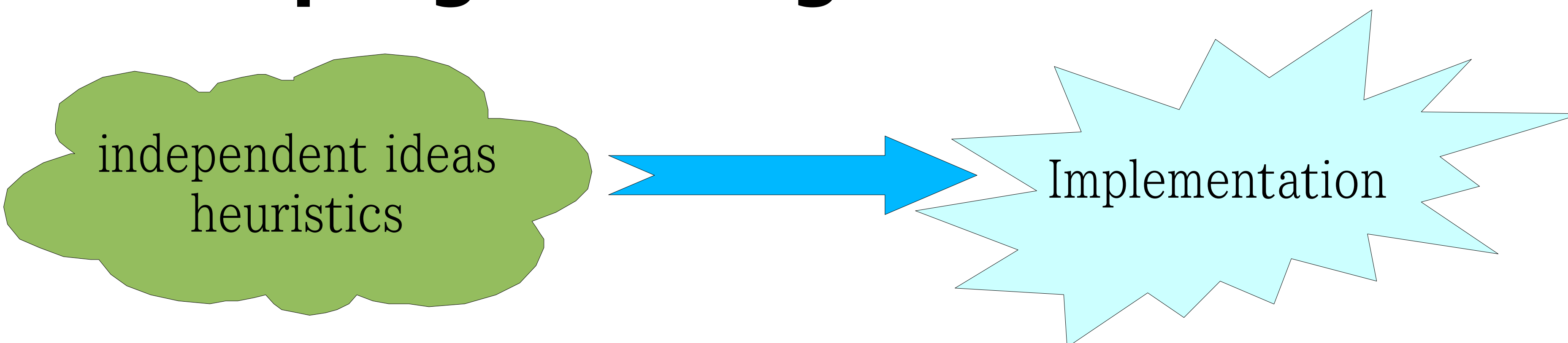
our interest:
synthesis of **reusable** (often **typeful** and **recursive**) algorithms

NB: typical use of GP: typeless CSP and function approx.

- ADATE (Olsson 1995)
  - monomorphic first-order type system
  - requires a file with tens of lines written for each synthesis
  - interesting results reported, but not reproducible
- PolyGP (Yu 1998)
  - polymorphic higher-order type system
  - requires a file with tens of lines written for each synthesis
  - exhaustive search is faster (Katayama 2004)

related work(2):
Genetic programming (3/3)

Genetic programming research



- different scientists use different settings
  “100 researchers, 100 genetic algorithms
- no comparison with non-heuristic approaches

It is often unclear if/which heuristic worked.

Motivation

1) First we should investigate systematic search.

- How can we optimize its implementation?
- How efficient is it? Is it slower/faster than GP?
- etc.

2) Heuristics (incl‘ GP) should be added based on the above knowledge.

- better modularity
- steady improvement

Policy (1)

- exploration of **exhaustive breadth-first search**
  - construct a basis on which to build heuristic approaches like GP,
  - control the exponential bloat in the search space.

to be more concrete,

- Limit to **(grammatically & type)-correct** expressions (of course!)
- Avoid multiple-count of **equivalent expressions**

  using known **reducible patterns** and (maybe) **transformation rules**

  e.g. *do not try both* `foldr foo bah []` *and* `bah`

  do not try both `(¥x -> foo x) bah` *and* `foo bah`
- **Optimize** the implementation --- **memoization**, etc.

Policy (2)

- **ease of use**

  at least it should be easier than writing the programs directly

  - <u>just writing a boolean function as predicate</u> invokes the search
  - use <u>general-purposed component combinator set</u>, not tailor-made function set for each synthesis

The implemented system

- **Introduction**
  motivation, policy, & conventional approaches
- **The implemented system**
  rough specification & detailed implementation
- **Experiments**
  compared with polymorphic GP
- **Conclusions**

Specification

keyboard

- property,
  e.g., \f. f “slkd”==4
- type (optional)
  e.g. [a] ──► Int

* The type is inferred when omitted.

component library file

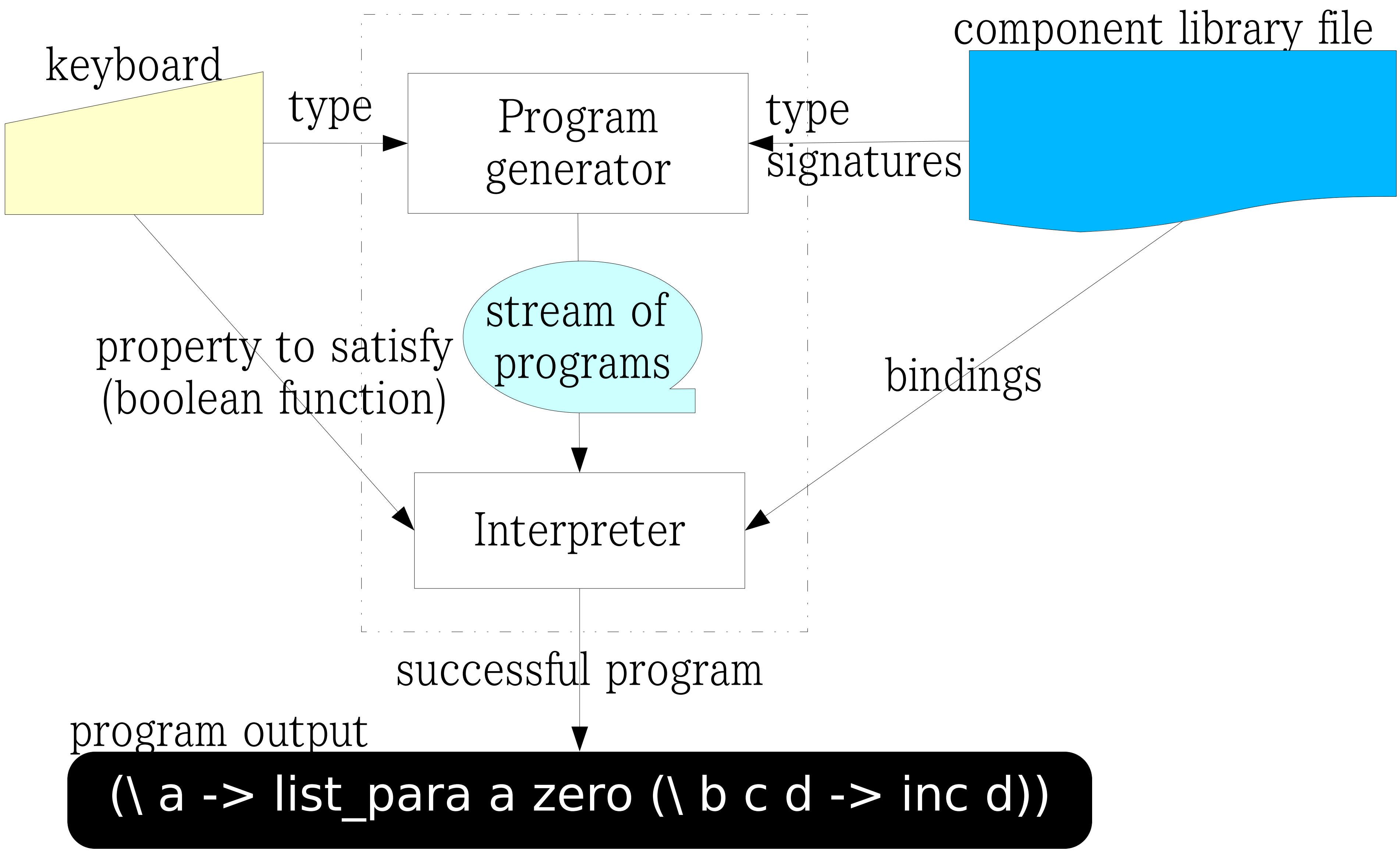
```
zero :: Int
zero = 0
inc   :: Int -> Int
inc   = \ x -> x+1
nat_para :: Int -> a->(Int->a->a)->a
nat_para = \ i x f ->
               if i==0 then x else
               f (i-1) (nat_para (i-1) x f)
     ...
```

synthesizer

program output

```
(\ a -> list_para a zero (\ b c d -> inc d))
```

Internal structure



Language

The language: Haskell subset

- Hindley-Milner type system

  without functions in containers, such as [a->b], (a,b->c), ...(for efficiency reasons)

- Frontend … usual lambda calculus

  Backend … de Bruijn lambda calculus

Katayama (2004) used typed *SIBC* combinators, but

- Such combinatory expressions are redundant;
- Search tree branches a lot in the shallow stages;
- The complexity of involved types bloats.

Implementation

Story:

- Preparation
  - Monad for breadth-first search
  - Monad transformer for type inference
- Arity known cases **where functions never return type vars**
- Arity unknown cases

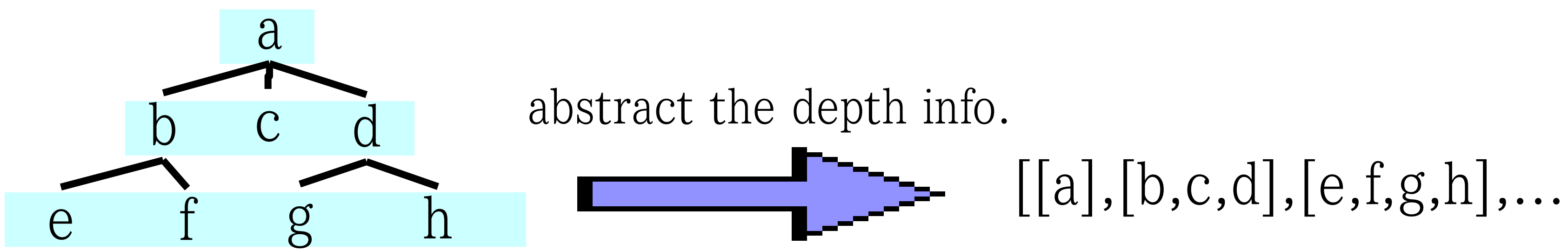
  where there can be a function of the form “forall a. Foo → Bar → … → a”
- Memoization **& devices for making keys hit often**

Monad for breadth-first search

- Spivey (2000)'s monad:

  Matrix a = Stream (Bag a)   (actually defined as [[a]])

  1st bag for depth 0, 2nd for depth 1, … of the search tree



  $\times$ **for combinations and $+$ for alternatives (defined as usual)**

  Eases implementation a lot but gobbles the heap a lot

- Recomputing variant:

  Recomp a = $\underbrace{\text{Int}}_{\text{depth}}$ $\rightarrow$ Bag a suppresses heap consumption

Monad trans. for type inference

- Typical implementation:

  TI a = Subst → Int → Maybe (a, Subst, Int)

  states

  Subst: "current" substitution

  Int: fresh variable ID

- **Monad transformer** considering **alternative states:**

  TI Recomp a

  where TI m a = Subst → Int → m (a, Subst, Int)

Naïve implementation (arity known)

```
data Expression = Lambda Expression          -- lambda abstraction
                | X        Int               -- de Bruijn variable
                | Expression :$ Expression   -- function application

uniExprs :: [(Expression,Type)] -> [Type] -> Type -> TI Recomp Expression
uniExprs prims avails (t0:->t1) = do result <- uniExprs prims (t0:avails) t1
                                     return (Lambda result)
uniExprs prims avails reqret
    = do (expr,typ) <- msum $ map return $ (zip (map X [0..]) avails ++ prims)
         typ'    <- freshVariablesForForalls typ
         setMGU reqret (returnTypeOf typ')    -- setMGU :: Type -> Type -> TI m ()
         spine avails typ' expr
   where spine :: Type -> Expression -> TI Recomp Expression
         spine (t:->ts) fun                  -- (:->) represents function type constructor
           = lift delay (do subst <- getSubst
                            arg <- uniExprs prims (map (apply subst) avails) (apply subst t)
                            spine ts (fun :$ arg) )
         spine _avs   _t      fun = return fun
```

***simplified to improve presentation rather than the efficiency**
e.g. Expressions with the same type are actually put together to [Expression]

Primitives returning type variables (arity unknown)

$\forall a. T_1 \rightarrow T_2 \rightarrow a$ can be specialized

$T_1' \rightarrow T_2' \rightarrow R, T_1'' \rightarrow T_2'' \rightarrow (A_1 \rightarrow R), T_1''' \rightarrow T_2''' \rightarrow (A_1 \rightarrow A_2 \rightarrow R), \ldots$

($T_1', T_2', T_1'', T_2'', T_1''', T_2''', \ldots$ are specialization of T_1 and T_2)

➡ **Arity undecidable**

- solution 1: generate all alternatives

  (a $\mapsto$ R) case `*mplus*` (a $\mapsto$ b -> R) case `*mplus*` (a $\mapsto$ b -> c -> R…

  problem: create lots of equivalent programs

- solution 2(current): introduce new constructor representing direct product of type variables:

  b, (b,c), (b,c,d), …

Memoization

The same simple type is requested many times

Memoize $\quad Type \to [Type] \to TI\ Recomp\ [Expression]$

(or* $\underline{Type} \to \underline{[Type]} \to \underline{Int} \to [([Expression], Subst, Int)]$)

request type

Types of available variables

depth

Queries often hit.

This list becomes long.

* **Exactly,** $\mathrm{Type} \to [\mathrm{Type}] \to \mathrm{Subst} \to \mathrm{Int} \to \mathrm{Int} \to [(\mathrm{Expression}, \mathrm{Subst}, \mathrm{Int})]$

but you could apply the substitution and rename the type variables to normalize its numbering.

Reorganizing (idea)

Point:

Only the *set* of available *types* matters

$\times$ list

e.g. for $x_0, x_1 :: Char$

$$\ldots x_0 \ldots x_0 \ldots \text{ is type correct} \Leftrightarrow \ldots x_0 \ldots x_1 \ldots \text{ is type correct}$$
$$\Leftrightarrow \ldots x_1 \ldots x_0 \ldots \text{ is type correct}$$
$$\Leftrightarrow \ldots x_1 \ldots x_1 \ldots \text{ is type correct}$$

$available = \{x_0, x_1 :: Char\}$ case and
$available = \{x_0 :: Char\}$ case can share the same memo entry*

* *post processing required*

Reorganizing (implementation)

Available variables: $x_0 :: Char, x_1 :: Bool, x_2 :: Char$

preprocess (sort & nub)

Memo entry: $x_0 :: Char, x_1 :: Bool$

Retriever map: $\{0 \mapsto [0,2], 1 \mapsto [1]\}$

Memo function wrapper

1. sort the argument types and assign **one variable name** for **one type**
2. invoke memo function
3. generate all cases by replacing variables (using retriever map).

Avoiding reducible expressions

- Optimization rules suggest redundancy in the program space.
- Because expressions with the same type are put together, rules that cannot detected from type info. might not help a lot.
- Current rule to identify reducible points:
  the strict argument of consumer functions
  (like case, cata, para) must not be constant
  (i.e. must include a free variable)

(Note: quite limited)

Experiments

- **Introduction**

  motivation, policy, & conventional approaches

- **The implemented system**

  rough specification & detailed implementation

- **Experiments**

  results on some easy problems

- **Conclusions**

Problems

Problems from the previous work (Katayama 2004) for comparison

- nth :: Int → [a] → a
  nth 5 “widjfgwi” == 'f' &&
  nth 1 “wddidjfgwi” == 'w'
- map :: (b → a) → [b] → [a]
  map (=='c') “stock” == [False,False,False,True,False] &&
  map (=='e') “peeped” == [False,True,True,False,True,False]
- length :: [a] → Int
  length “hageho” == 6 && length “hoge” == 4

The component combinators

Previous work

- **S**, **K**, **I**, **B**, and **C**
- constructors
- curried paramorphisms
- head, tail and pred

New algorithm

- constructors
- curried paramorphisms
- head, tail and pred

Results

Evaluation of proposed method:

Computation time (sec.)	nth	map	length
Old algorithm (real)	5.3	2.2	0.03
(user)	5.1	2.2	0.02
New algorithm (real)	0.8	1.9	0.03
(user)	0.6	1.2	0.02

- Improved for the problems used in the previous work.
- Greater programs (sized more than 12) still require more than a minute or cannot be synthesized.

Discussion towards a usable system:
# of equivalent programs

The results so far are happened-to-work toy examples!

Synthesis of *take::Int->[a]->[a]* requires tens of seconds.

.... How much it might **potentially** be optimized?

→ Do (very) light-weighted random testing

EXPERIMENT TO GIVE SOME ESTIMATE:

for each *f :: Int -> [a] -> [a]* in the generated expressions,

- compute *(f 2 "12332", f 0 "56789", f 2 "k", f 0 "")* > file
- compare wc file and sort file | uniq | wc

file	... 251940 lines
sort file | uniq	... 514 lines

LOTS OF ROOM FOR IMPROVEMENT

Conclusions

- **Introduction**
  motivation, policy, & conventional approaches
- **The implemented system**
  rough specification & detailed implementation
- **Experiments**
  results on some easy problems
- **Conclusions**

Summary

- investigated breadth-first exhaustive search for de Bruijn expressions
- easy problems sized around ten can be solved within seconds
- Still room for efficiency improvement

Future work

- Controlling exponential bloat

  Solution: **remove more equivalent programs**

  or lower the priorities of seem-to-be-equivalent programs

  - transformation rules
  - use lightweighted random testing for small expressions
- Type classes: **dealing with contexts**
  - contexts "generic in nature" e.g. Eq, Show, Read, etc. … just ignore.
  - other "ad hoc" classes like Num … needs implementation.
- Heuristics:

  e.g. to prioritize subexpressions of expressions
  that returned the correct outputs to some of the inputs