Declarative Programming

Efficient and reliable programming with human intelligence

Michael Hanus

Kiel University
Programming Languages and Compiler Construction

PADL 2026, Rennes

Programming: A trivial task?

Traditional programming has many pitfalls

Compute the factorial of a natural number:

```
function fac(n:nat):nat =
begin
   z:=1; p:=1;
   while z<n+1 do
   begin p:=p*z; z:=z+1 end;
   return(p)
end
```

Potential errors:

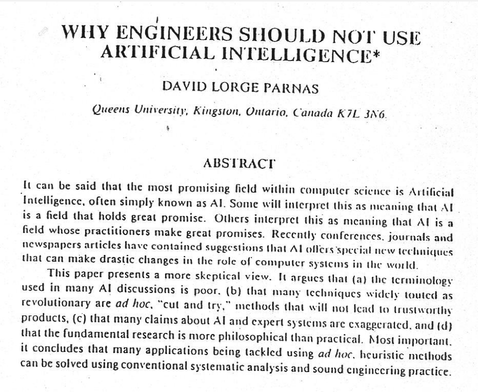
- counter initialization: `z:=1` or `z:=0` ?
- loop condition: `z<n+1` or `z<n` or `z<=n` ?
- statement ordering: `z:=z+1` before or after `p:=p*z` ?

Why programming? Why not ChatGPT, Copilot,. . . ?

Questions from worried parents of prospective students

- Is it necessary to learn programming?
- Or is it necessary to study computer science at all?

David L. Parnas 1988 (INFOR 26(4), 234–246)



WHY ENGINEERS SHOULD NOT USE ARTIFICIAL INTELLIGENCE*

DAVID LORGE PARNAS

Queens University, Kingston, Ontario, Canada K7L 3N6.

ABSTRACT

It can be said that the most promising field within computer science is Artificial Intelligence, often simply known as AI. Some will interpret this as meaning that AI is a field that holds great promise. Others interpret this as meaning that AI is a field whose practitioners make great promises. Recently conferences, journals and newspapers articles have contained suggestions that AI offers special new techniques that can make drastic changes in the role of computer systems in the world.

This paper presents a more skeptical view. It argues that (a) the terminology used in many AI discussions is poor, (b) that many techniques widely touted as revolutionary are *ad hoc*, "cut and try," methods that will not lead to trustworthy products, (c) that many claims about AI and expert systems are exaggerated, and (d) that the fundamental research is more philosophical than practical. Most important, it concludes that many applications being tackled using *ad hoc*, heuristic methods can be solved using conventional systematic analysis and sound engineering practice.

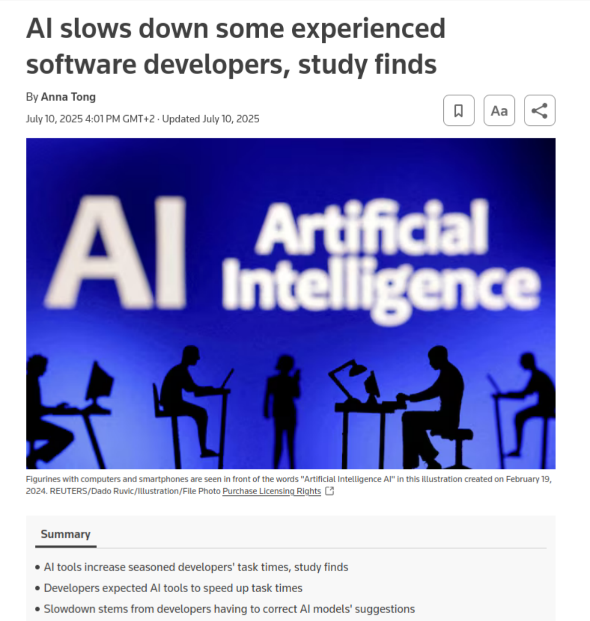
It is indeed a feature of AI programs that their authors do not know what they will do, the field has made a virtue out of a vice.

. . .

The way humans solve problems is not usually the best way for a machine to solve the same problem.

But now we have LLMs!

ACM Technews, Reuters, July 10, 2025



AI slows down some experienced software developers, study finds

By **Anna Tong**

July 10, 2025 4:01 PM GMT+2 · Updated July 10, 2025

Figurines with computers and smartphones are seen in front of the words "Artificial Intelligence AI" in this illustration created on February 19, 2024. REUTERS/Dado Ruvic/Illustration/File Photo Purchase Licensing Rights

Summary

- AI tools increase seasoned developers' task times, study finds
- Developers expected AI tools to speed up task times
- Slowdown stems from developers having to correct AI models' suggestions

Reuters, July 10, 2025

Contrary to popular belief, using cutting-edge artificial intelligence tools slowed down experienced software developers when they were working in codebases familiar to them, rather than supercharging their work, a new study found.

. . .

Before the study, the open-source developers believed using AI would speed them up, estimating it would decrease task completion time by 24%. Even after completing the tasks with AI, the developers believed that they had decreased task times by 20%. But the study found that using AI did the opposite: it increased task completion time by 19%.

. . .

The slowdown stemmed from developers needing to spend time going over and correcting what the AI models suggested.

Why not prove code correctness with ChatGPT?

> Define a function to compute the last element of a list.

Certainly!
… *long text omitted*…

> Can you prove that this implementation is correct?

Certainly!

…

We can verify the correctness of the function by running various test cases to cover different scenarios:

…

Here’s how we can implement these tests:

…

Proving code correctness with ChatGPT, 2nd try

$>$ Can you formally prove that this implementation is correct?

…

Let’s formally prove correctness using **mathematical induction** on the **length of the list**.

Base Case: List of length 1

…

Inductive Hypothesis: Assume that for a list of length *n*…

Inductive Step:

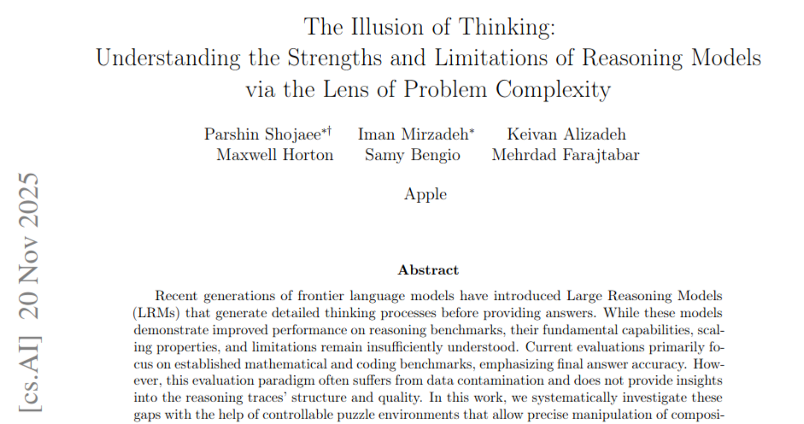
…

informal explanation without using inductive hypothesis!

…

Problem: no formal claim to prove, i.e., what is the last element?

Apple Research, June 2025



The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity

Parshin Shojaee*† Iman Mirzadeh* Keivan Alizadeh
Maxwell Horton Samy Bengio Mehrdad Farajtabar

Apple

Abstract

Recent generations of frontier language models have introduced Large Reasoning Models (LRMs) that generate detailed thinking processes before providing answers. While these models demonstrate improved performance on reasoning benchmarks, their fundamental capabilities, scaling properties, and limitations remain insufficiently understood. Current evaluations primarily focus on established mathematical and coding benchmarks, emphasizing final answer accuracy. However, this evaluation paradigm often suffers from data contamination and does not provide insights into the reasoning traces' structure and quality. In this work, we systematically investigate these gaps with the help of controllable puzzle environments that allow precise manipulation of composi-

- LLMs *seem* to reason on small problem sizes
- LLMs hallucinate on larger problems
- ok for language-oriented tasks, but for reliable programming?

Developing trustworthy code

Reliable code: important but difficult to assure

- Unit testing: find the right test cases
- Property-based testing: formulate *properties*
- Verification: formally prove *properties*

Property “last element of a list”

> *If we add a single element to some list, then this element is the last element of the concatenated list.*

Formally (“++” denotes list concatenation, `[...]` denotes lists):

$$\forall \texttt{xs,x:}\ \ \texttt{last (xs ++ [x]) = x}$$

⇝ use this condition for property-based testing and verification

Declarative programming

Property/definition "last element of a list"

```
∀xs,x:  last (xs ++ [x]) = x
```

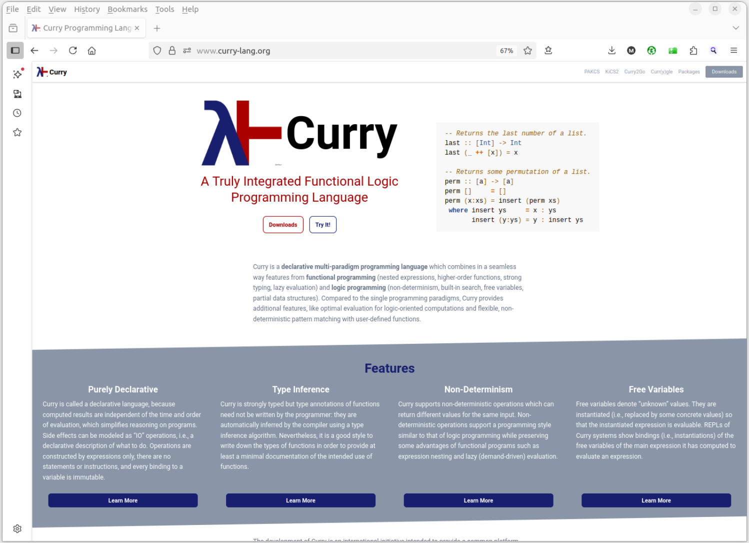
valid definition and, thus, executable in declarative programming!

Declarative programming:

- describe/specify *what* is the problem to solve
- do not write steps/statements *how* to solve the problem
- main difference to imperative programming:
  - *referential transparency*: the value of an expression depends on the values of subexpressions but not on evaluation time
  - substitution principle (replace equals by equals), no side effects
- formalisms/logics to describe problems
  - lambda calculus ⇝ functional programming (e.g., Haskell)
  - predicate logic ⇝ logic programming (e.g., Prolog)

amalgamate both ⇝ Curry

Declarative programming with Curry-lang.org


File Edit View History Bookmarks Tools Help
Curry Programming Lang ×
www.curry-lang.org
67%
Curry
PAKCS KiCS2 Curry2Go Curr(y)gle Packages Downloads

Curry

A Truly Integrated Functional Logic Programming Language

Downloads Try It!

```
-- Returns the last number of a list.
last :: [Int] -> Int
last (_ ++ [x]) = x

-- Returns some permutation of a list.
perm :: [a] -> [a]
perm []     = []
perm (x:xs) = insert (perm xs)
 where insert ys     = x : ys
       insert (y:ys) = y : insert ys
```

Curry is a **declarative multi-paradigm programming language** which combines in a seamless way features from **functional programming** (nested expressions, higher-order functions, strong typing, lazy evaluation) and **logic programming** (non-determinism, built-in search, free variables, partial data structures). Compared to the single programming paradigms, Curry provides additional features, like optimal evaluation for logic-oriented computations and flexible, non-deterministic pattern matching with user-defined functions.

Features

Purely Declarative

Curry is called a declarative language, because computed results are independent of the time and order of evaluation, which simplifies reasoning on programs. Side effects can be modeled as “IO” operations, i.e., a declarative description of what to do. Operations are constructed by expressions only, there are no statements or instructions, and every binding to a variable is immutable.

Learn More

Type Inference

Curry is strongly typed but type annotations of functions need not be written by the programmer: they are automatically inferred by the compiler using a type inference algorithm. Nevertheless, it is a good style to write down the types of functions in order to provide at least a minimal documentation of the intended use of functions.

Learn More

Non-Determinism

Curry supports non-deterministic operations which can return different values for the same input. Non-deterministic operations support a programming style similar to that of logic programming while preserving some advantages of functional programs such as expression nesting and lazy (demand-driven) evaluation.

Learn More

Free Variables

Free variables denote “unknown” values. They are instantiated (i.e., replaced by some concrete values) so that the instantiated expression is evaluable. REPLs of Curry systems show bindings (i.e., instantiations) of the free variables of the main expression it has computed to evaluate an expression.

Learn More

The development of Curry is an international initiative intended to provide a common platform

Declarative programming with Curry-lang.org

Curry $=$ HaskellHaskell $+$ non-determinism $+$ free variables

Functional programming: factorial function
Mathematical definition:

$$\begin{aligned} \mathit{fac}(n) &= 1 * 2 * \cdots * (n-1) * n \\ &= \mathit{fac}(n-1) \qquad * n \end{aligned}$$

Recursive/constructive definition:

$$\mathit{fac}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \mathit{fac}(n-1) * n & \text{if } n > 0 \end{cases}$$

Implementation in Haskell/Curry:

```
fac n | n == 0 = 1
      | n > 0  = fac (n-1) * n
```

Declarative programming with Curry-lang.org

Curry = Haskell + non-determinism + free variables

Typical scheme: define operations by case distinctions on data constructors

```
data Bool = False | True  -- Boolean values

not :: Bool  → Bool
not False = True
not True  = False
```

Type of polymorphic lists: [τ] ≈ list with elements of type τ

```
data [a] = [] | a : [a]
```

List concatenation:

```
(++) :: [a]  →  [a]  →  [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Declarative programming with Curry-lang.org

Curry = Haskell + non-determinism + free variables

Curry applies all (applicable) rules defining an operation

Non-deterministic list insertion:

```
insert :: a → [a] → [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

```
> insert 0 [1,2]  ⇝ [0,1,2] ? [1,0,2] ? [1,2,0]
```

Some permutation of a list:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

```
> perm [1,2,3]  ⇝ [3,2,1] ? [3,1,2] ? [2,3,1] ?
                  [2,1,3] ? [1,3,2] ? [1,2,3]
```

Non-deterministic operation!

Declarative programming with Curry-lang.org

Curry = Haskell + non-determinism + free variables

Free variable: unknown value

Unification (=:=): equality constraint with guessing/binding free variables

Last element of a given list:

```
last :: [Int]  → Int
last ys | ys =:= xs ++ [x]
        = x                 where xs,x free
```

Useful abbreviation: *functional patterns*

```
last :: [Int]  → Int
last (xs ++ [x]) = x
```

Checking palindromes:

```
palindrome (xs ++ reverse xs)         = "even"
palindrome (xs ++ [_] ++ reverse xs) = "odd"
```

Declarative programming with Curry-lang.org

Curry = Haskell + non-determinism + free variables

+ encapsulated search
+ set functions
+ constraints
+ default rules
+ standard class `Data` (typing free variables)
+ generic (fair) search
+ determinism types (PPDP'25)
+ . . .

Semantics of declarative programs

High-level (declarative) semantics

- specifies *intended* result values (of expressions)
- unspecified: operational/implementation aspects (e.g., strategies)

Operational semantics

- specifies computation strategy and computed values
- reason about time (computation steps) and space

Both views should coincide!

. . . with well-defined restrictions (LP: compute general representatives of values)

Advantage:

- programmer considers declarative view, abstract from operational aspects
- analysis or verification tools use an appropriate view

High-level semantics for Curry: CRWL

Constructor-based ReWriting Logic [González-Moreno et al. JLP'99]

- execution-independent specification of possible *values* of expressions
- *values*: variables ($\mathcal{V}$) and data constructor ($\mathcal{C}$) applications

To cover non-strict, non-deterministic operations:

- *partial values*: values extended by $\bot$ (*undefined value*)
  (`1:2:⊥` approximates `1:2:[]`, `1:2:3:[]`,...)
- CRWL specifies approximation statements $e \twoheadrightarrow t$

CRWL rules

$$\frac{}{x \twoheadrightarrow x} \quad x \in \mathcal{V} \qquad\qquad \frac{e_1 \twoheadrightarrow t_1 \quad \cdots \quad e_n \twoheadrightarrow t_n}{C\ e_1 \ldots e_n \twoheadrightarrow C\ t_1 \ldots t_n} \quad C \in \mathcal{C}$$

$$\frac{}{e \twoheadrightarrow \bot} \qquad\qquad \frac{e_1 \twoheadrightarrow \theta(t_1) \quad \cdots \quad e_n \twoheadrightarrow \theta(t_n) \quad \theta(r) \twoheadrightarrow t}{f\ e_1 \ldots e_n \twoheadrightarrow t} \quad \begin{array}{l} f\ t_1 \ldots t_n = r \in \mathcal{P} \\ \theta \in \mathit{CSubst}_\bot \end{array}$$

$\mathit{CSubst}_\bot$: partial constructor substitutions

Semantics of non-deterministic operations

Choice operation

```
x ? _  =  x                    coin = 0 ? 1
_ ? y  =  y                    dup x = (x,x)
```

Values of `(dup coin)`

- **Call-time choice**: values of arguments fixed before function call
  $\leadsto$ `(0,0)  (1,1)`
- **Run-time choice**: values of arguments fixed when they are used
  $\leadsto$ `(0,0)  (0,1)  (1,0)  (1,1)`

Decision by language design:

- run-time choice: computed value might depend on strategy
- call-time choice: semantics with “least astonishment”,
  can be implemented with call-by-value or call-by-need (sharing!)

CRWL specifies call-time choice

$$\frac{}{x \twoheadrightarrow x}\; x \in \mathcal{V} \qquad \frac{e_1 \twoheadrightarrow t_1 \;\cdots\; e_n \twoheadrightarrow t_n}{C\; e_1 \ldots e_n \twoheadrightarrow C\; t_1 \ldots t_n}\; C \in \mathcal{C}$$

$$\frac{}{e \twoheadrightarrow \bot} \qquad \frac{e_1 \twoheadrightarrow \theta(t_1) \;\cdots\; e_n \twoheadrightarrow \theta(t_n) \quad \theta(r) \twoheadrightarrow t}{f\; e_1 \ldots e_n \twoheadrightarrow t}\; \begin{array}{l} f\; t_1 \ldots t_n = r \in \mathcal{P} \\ \theta \in CSubst_\bot \end{array}$$

```
x ? _  =  x                   coin = 0 ? 1
_ ? y  =  y                   dup x = (x,x)
```

To reduce `(dup coin)` by CRWL:

reduce argument `coin` to either `0` or `1` before reducing right-hand side of `dup`

CRWL rules contain a lot of guesses:

- Apply $\bot$-rule or another?
- Which program rule and which θ to reduce a function?

⇝ use *narrowing* to avoid or delay these guesses

Operational semantics: narrowing

Narrowing step: $e \rightsquigarrow_{p,\, l=r,\, \sigma} \sigma(e[r]_p)$

p : non-variable position in e

$l = r$: program rule (variant)

σ : most general unifier for $e|_p$ and l

$\rightsquigarrow$ no $\perp$-guesses, no θ-guesses

$\rightsquigarrow$ guess rule and position p

Use a *strategy* to select narrowing position

Needed narrowing [JACM 2000]

- constructive method to compute positions and specific unifiers
- demand-driven (needed) strategy
- originally defined on *inductively sequential* rewrite systems where all rules of an operation can be organized in a (definitional) tree

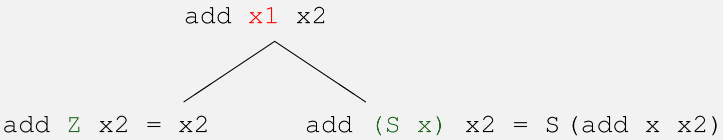
Definitional tree [Antoy’92]

- nodes marked with patterns
- consists of branch nodes (case distinction), rule nodes
- contains all rules of a function, root with most general pattern

Addition on Peano numbers

```
data Nat = Z | S Nat

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

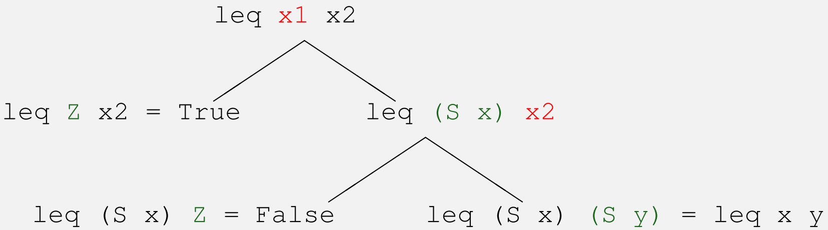


```
add x1 x2
add Z x2 = x2
add (S x) x2 = S (add x x2)
```

Definitional tree [Antoy'92]

Less-or-equal on Peano numbers

```
leq Z       _     = True
leq (S _) Z       = False
leq (S x) (S y) = leq x y
```



- can be computed at compile time
- guide strategy of needed narrowing

Needed narrowing [JACM 2000]

Evaluate outermost function call $(f\ e_1 \ldots e_n)$ (informal)

Find an argument i *needed* by all rules: if e_i

- function call: evaluate e_i
- constructor-rooted: select corresponding rules (and proceed)
- variable: instantiate it to constructors needed by rules (and proceed)

Properties of needed narrowing

- Sound: computed values are derivable by CRWL
- Complete: any CRWL-value is an instance of some computed value
- Optimal strategy:
  1. No unnecessary steps: each step is needed, i.e., unavoidable for some value
  2. Shortest derivations: derivations have minimal length (with sharing)
  3. Minimal value set: any two distinct derivations compute independent values
  4. Determinism: no guessing during evaluation of variable-free expressions

$\leadsto$ Consideration of evaluation details not necessary for reliable programming

Declarative programming: intermediate summary

Trustworthy software is important

- AI/LLMs can be a helpful component
- reliable programs require precise descriptions/specifications
- need to be provided by humans (“precise prompt engineering”)
- declarative programming supports high-level executable specifications
- can be used to test or verify more efficient implementations

Further advantage of declarative programming

case distinctions, no side effects $\rightsquigarrow$ automatic analysis and verification

In the following:

- contract verification
- inference and verification of non-fail conditions

Contracts for Curry [PADL 2012]

Given: f `::` $\tau_1 \to \cdots \to \tau_n \to \tau$

Contract for *f*: pre- and postcondition

Precondition:

f`'pre ::` $\tau_1 \to \cdots \to \tau_n \to$ `Bool`

Postcondition:

f`'post ::` $\tau_1 \to \cdots \to \tau_n \to \tau \to$ `Bool`

Dynamic contract checking

Curry preprocessor transforms contracts into dynamic checks:

- precondition ⇝ check arguments before each call
- postcondition ⇝ check arguments/result after evaluation

Contract verification [LOPSTR 2017, FI 2020]

Verify contracts at compile time ⇝ omit run-time checking, improve trust

Factorial operation with contract:

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

fac'pre  n   = n >= 0
fac'post n f = f > 0
```

Verify precondition of recursive `fac` call:

`n>=0` (by precondition)

$\neg\texttt{(n==0)}$ (since `else` branch is chosen)

$\texttt{n>=0} \wedge \neg\texttt{(n==0)} \Longrightarrow \texttt{(n-1)>=0}$ (by SMT solver)

⇝ precondition of recursive call always satisfied, omit run-time check

Contract verification

Verifying postcondition

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

fac'post n f = f > 0
```

Consider value of right-hand side:

1. `then` branch: $1 > 0$ ⇝ postcondition satisfied
2. `else` branch:
   `n>=0` (by precondition)
   `¬(n==0)` (since `else` branch is chosen)
   `fac(n-1)>0` (by postcondition)
   `n>=0 ∧ ¬(n==0) ∧ fac(n-1)>0 ⟹ n*fac(n-1)>0` (by SMT)
   ⇝ postcondition satisfied

Altogether: postcondition always satisfied, omit run-time checks

Contract verification: number summation

Number summation

```
sum :: Int  → Int
sum n = if n==0 then 0
                else n + sum (n-1)

sum'pre n     = n >= 0
sum'post n f = f == n * (n+1) `div` 2
```

Contract verifier:

- precondition of recursive call satisfied
- postcondition satisfied

⇝ fully automatic verification of postcondition (proof of Gauss' formula)

Contract verification: implementation

Assertion-collecting semantics [LOPSTR 2017, FI 2020]

1. compute with symbolic values instead of concrete ones
2. collect properties that are known to be valid
3. do not evaluate functions but collect their pre- and postconditions

Use SMT solver (Z3) to verify collected assertions:
if verifiable, omit dynamic contract check

. . . to make it feasible on Curry programs:

- compile Curry programs into simpler intermediate language: FlatCurry
- remove local declarations by lambda lifting
- translate complex patterns into case/or expressions
- standard Curry front end produces FlatCurry programs (used by interpreters, compilers, and other tools)

FlatCurry

Abstract syntax of FlatCurry

$$
\begin{array}{lcll}
D & ::= & f(x_1,\ldots,x_n) = e & \text{(function definition)} \\[1ex]
e & ::= & x & \text{(variable)} \\
  & \mid & c(e_1,\ldots,e_n) & \text{(constructor call)} \\
  & \mid & f(e_1,\ldots,e_n) & \text{(function call)} \\
  & \mid & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1;\ldots;p_n \to e_n\} & \text{(case expression)} \\
  & \mid & e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
  & \mid & \mathit{let}\ \{x_1 = e_1;\ldots;x_n = e_n\}\ \mathit{in}\ e & \text{(let binding)} \\
  & \mid & \mathit{let}\ x_1,\ldots,x_n\ \mathit{free}\ \mathit{in}\ e & \text{(free variables)} \\[1ex]
p & ::= & c(x_1,\ldots,x_n) & \text{(pattern)}
\end{array}
$$

- only top-level functions
- each function defined by one rule
- corresponds to textual representation of definitional tree

From Curry to FlatCurry

Curry: source program

```
leq Z     _     = True
leq (S _) Z     = False
leq (S x) (S y) = leq x y
```

FlatCurry (pretty printed)

```
leq x1 x2 = case x1 of
              Z     →  True
              S x3  →  case x2 of
                         Z     →  False
                         S x4  →  leq x3 x4
```

Assertion-collecting semantics:

- top-down pass through right-hand side expression
- collect properties and generate proof obligations

Verify programs as non-failing

Motivation

- ensure that a program has no (internal) run-time failure
- allow partially defined operations but control them

Partially defined operations

```
head :: [a]  → a                 tail :: [a]  → [a]
head (x:xs) = x                  tail (x:xs) = xs
```

Controlling inputs to partially defined operations

```
readCommand = do
  putStr "Input a command:"
  ws ← fmap words getLine
  case null ws of True  → readCommand
                  False → processCommand (head ws) (tail ws)
```

Fail-free program verification

Objective

- allow programming with partially defined operations and failures
- encapsulate (logic) subcomputations containing failures
- prove non-failure of (top-level) functional computation

$\rightsquigarrow$ fix Tony Hoare's "billion dollar mistake" for declarative programs

Fail-free programs [PPDP 2018]

- add *non-fail conditions* to operations
- if satisfied at call site $\rightsquigarrow$ computation does not fail

Fully automatic method [FLOPS 2024, SciCo 2026]

- infer *abstract call types* for each operation
- approximation of non-fail conditions
- approximate input/output behavior by *in/out types*
- for each call in a function definition: check call type requirement

Call types and abstract types

Call type of an operation

- set of argument values so that operation does not fail
- precise call types complex or intractable

Abstract types $\mathcal{A}$

- approximate sets of values (regular types, depth-bounded terms $\mathcal{A}_k, \ldots$)
- example abstract domain: *top-level constructors*

$$\mathcal{A}_1 = \{D \subseteq \mathcal{C} \mid \text{all constructors of } D \text{ belong to same type}\} \cup \{\top\}$$

Abstract call type examples

```
head (x:xs) = x                 tail (x:xs) = xs
```

Abstract call type of `head` and `tail`: $\{$`:`$\}$

(infer by considering patterns in left-hand sides)

In/out types

Verifying calls depend on context:

```
... case null ws of True  → readCommand
                    False → processCommand (head ws) (tail ws)
```

In/out type of an operation

- approximate input/output behavior
- set of abstract argument/result types:

$IO(f_n) \subseteq \{a_1 \ldots a_n \hookrightarrow a \mid a_1, \ldots, a_n, a \in \mathcal{A}\}$

Example

```
null []     = True
null (x:xs) = False
```

$IO(\texttt{null}) = \{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\}$

Inference by analyzing pattern/case structure of defining rules

Automatic non-failure checking

Overall method w.r.t. abstract type domain $\mathcal{A}$:

1. Infer (abstract) in/out types for all operations
2. Infer initial (abstract) call types for all operations
3. For each defined operation:
   check calls in right-hand side for satisfaction of their call types
4. If not successful for some operation:
   refine its call type with call-type constraints from unsatisfied call types
   and start again with step 3.

Termination of fixpoint method: finite abstract type domain or widening steps

Worst case: inference of empty call types ⇝ encapsulate its use

```
readCommand = do
  ws ← fmap words getLine
  case null ws of True  → readCommand
                  False → processCommand (head ws) (tail ws)
```

$$IO(\texttt{null}) = \{\{\texttt{[]}\} \hookrightarrow \{\texttt{True}\}, \{\texttt{:}\} \hookrightarrow \{\texttt{False}\}\} \qquad \rightsquigarrow \quad \texttt{ws} \mapsto \{\texttt{:}\}$$

Inference and verification of non-fail conditions

Non-fail conditions with abstract types [FLOPS 2024]

Fully automatic with fixpoint iteration as sketched above:

```
last [x]        = x
last (x:y:zs) = last (y:zs)
```

```
⇝ Non-fail condition: {:}
```

Non-fail conditions with arithmetic conditions [APLAS 2024]

Fully automatic by using SMT solver for arithmetic conditions:

```
fac n | n == 0 = 1
      | n > 0  = n * fac (n - 1)
```

```
⇝ fac'nonfail n = n==0 || n>0
```

Conclusions

Declarative programming

- high-level programming style
- reliability by expressing *what* should be implemented
  (without such a specification, AI-generated programs are not trustworthy)
- formal semantics supports analysis and verification of programs

Human intelligence is necessary to specify the “*what*”!

Some further aspects of Curry

- demand-driven evaluation strategy ⇝ optimal evaluation [JACM 2000]
- translate logic programs to Curry ⇝ smaller search spaces [TPLP 2022, LOPSTR 2024]
- various programming tools (Visual Studio Code,...), packages (> 150), applications, e.g., module information system at Kiel University (`https://moduldb.informatik.uni-kiel.de/`)
- implementations (⇝ `www.curry-lang.org`)
  - PAKCS: compiles to Prolog (efficient compiler)
  - KiCS2: compiles to Haskell (efficient executables)
  - Curry2Go: compiles to Go (fair parallel search)
  - KMCC (under development): compiles to Haskell (monadic target code, fair search, efficient executables)
  - ...