

A Language-based Approach to Functionally Correct Imperative Programming

Edwin Westbrook, Aaron Stump, Ian Wehrman

Washington University in Saint Louis

Purpose: Program Verification

- Language-based
 - Specifications and proofs are part of the language
 - Checked statically by compiler
- Functionally correct
 - If data is returned by a function, it is valid
- Imperative
 - Mutable state

The Language: RSP1

- Dependent pattern-matching
 - Functional construct: extends lamda
- Dependent records
 - Not needed, but convenient for passing multiple values
- First-order LF: no LF lambda
 - No bound variables in mutable state
 - No Higher-order abstract syntax
- Dependent attributes (imperative feature)
 - We use ML-style references here for expository purposes
- General recursion

The Language: RSP1

- Type safe
- Decidable type checking
- Technical idea: *Purity*
 - Keeps “unsafe” features out of types
 - Similar to features of Xi’s ATS and Schürmann’s ∇ -calculus
- Reasons about heap indirectly
 - No explicit modeling of the heap
 - Avoids more complicated systems like linear types

Outline

- Some useful syntax
- Motivating example: binary search tree insert
 - Unverified without dependent types
- Safe BSTs with dependent types
- Purity
- Verified BST insert
- Typing dependent pattern matches

Some Syntax

- RSP1 is based on pattern (or ρ) abstractions:

$$\mathbf{x}_1 \backslash \mathbf{P}_1 \backslash \Gamma_1 \rightarrow \mathbf{M}_1 \quad | \quad \dots \quad | \quad \mathbf{x}_n \backslash \mathbf{P}_n \backslash \Gamma_n \rightarrow \mathbf{M}_n$$

- ρ is a “pattern function” (combines λ with **case**):
 - Returns body (\mathbf{M}_i) of first case (blue box) that matches arg
 - Case matches when $\mathbf{P}_i = \text{arg}$ modulo variables in Γ_i
 - \mathbf{x} is bound to whole argument when pattern matches
- Example:

$$(\mathbf{x} \backslash \mathbf{c} \ \mathbf{y} \ \mathbf{z} \backslash \mathbf{y}:\text{nat}, \ \mathbf{z}:\text{nat} \rightarrow \mathbf{d} \ \mathbf{x} \ \mathbf{z} \ \mathbf{y}) \ (\mathbf{c} \ \mathbf{a} \ \mathbf{b})$$
$$\nabla$$
$$\mathbf{d} \ (\mathbf{c} \ \mathbf{a} \ \mathbf{b}) \ \mathbf{b} \ \mathbf{a}$$

Syntactic Sugar

- We elide pattern contexts when they can be inferred
 - Written $\mathbf{x} \setminus \mathbf{P} \setminus \rightarrow \mathbf{N}$
- We elide pattern when it is a single variable
 - $\mathbf{x} \setminus \rightarrow \mathbf{N}$ stands for $\mathbf{x} \setminus \mathbf{y} \setminus \mathbf{y}:\mathbf{T} \rightarrow \mathbf{N}$
 - Matches everything: similar to λ -abstractions
- `match M with N` is shorthand for $\mathbf{N} \ \mathbf{M}$
- Dependent function type: $\mathbf{x}:\mathbf{T} \Rightarrow \mathbf{U}$
 - Used for LF constructors: same as $\Pi x : T.U$
- Dependent pattern function type: $\mathbf{x}:\mathbf{T} =\mathbf{c}> \mathbf{U}$

Motivating Example

```
rec insert = x \ ->
  T \ empty \ -> (* empty case *)
    tree x (ref empty) (ref empty) | (* new data node *)
  T \ tree data left right \ -> (* left recursive case *)
    if x <= data then
      left
        := insert x !left; (* update sub-tree *)
      T (* return T *)
    else
      (* ...symmetrical case *)
```

Motivating Example

```
rec insert = x \ ->
  T \ empty \ -> (* empty case *)
    tree x (ref empty) (ref empty) | (* new data node *)
  T \ tree data left right \ -> (* left recursive case *)
    if x <= data then
      data
        := insert x !left; (* update sub-tree *)
      T (* return T *)
    else
      (* ...symmetrical case *)
```

Motivating Example

```
rec insert = x \ ->
  T \ empty \ -> (* empty case *)
    tree x (ref empty) (ref empty) | (* new data node *)
  T \ tree data left right \ -> (* left recursive case *)
    if x <= data then
      data Non-dependent types catch datatype bugs
      := insert x !left; (* update sub-tree *)
    T (* return T *)
  else
    (* ...symmetrical case *)
```

Motivating Example

```
rec insert = x \ ->
  T \ empty \ -> (* empty case *)
    tree x (ref empty) (ref empty) | (* new data node *)
  T \ tree data left right \ -> (* left recursive case *)
    if x <= data then
      right
      := insert x !left; (* update sub-tree *)
    T (* return T *)
  else
    (* ...symmetrical case *)
```

Motivating Example

```
rec insert = x \ ->
```

```
  T \ empty \ -> (* empty case *)
```

```
    tree x (ref empty) (ref empty) | (* new data node *)
```

```
  T \ tree data left right \ -> (* left recursive case *)
```

```
    if x <= data then
```

```
      right Non-dependent types do not catch BST bugs
```

```
        := insert x !left; (* update sub-tree *)
```

```
      T (* return T *)
```

```
    else
```

```
      (* ...symmetrical case *)
```

- This insert can create invalid BSTs!

Safe BSTs

- To verify BST programs, we need a “safe” BST type
 - Can represent all and only the valid BSTs

- First step: **index** the BST type by its range:

`BST :: l:nat => u:nat => type`

- Example: `BST 1 10` is the type of BSTs with data from 1 to 10

Safe BSTs

- Second step: verified constructors
- Empty BST `l u` node:
 - Needs valid range, i.e. $l \leq u$

`empty ::`

`l:nat => u:nat =>`

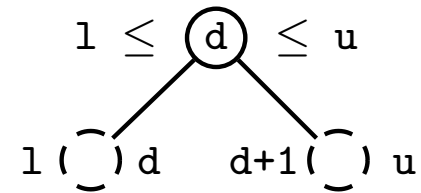
`leq l u => BST l u;;`

- Example: `empty 1 10` is a `BST 1 10`
- Relies on “proof objects”
 - `leq x y` is inhabited iff $x \leq y$

Safe BSTs

- Data node is more complex:

- Data must be in range
- Subtrees partition the range



bst ::

$l:\text{nat} \Rightarrow u:\text{nat} \Rightarrow d:\text{nat} \Rightarrow$

$\text{leq } l \ d \Rightarrow \text{leq } d \ u \Rightarrow$

$\text{Ref BST } l \ d \Rightarrow \text{Ref BST } (\text{succ } d) \ u \Rightarrow$

$\text{BST } l \ u;;$

- Also relies on proof objects

Proof Objects

- Third step: define proof objects
- Number theory: $x \leq y$ iff
 - $x = y$, or
 - $x \leq y - 1$
- Can convert these to constructors for `leq x y`:
 - `leq_eq :: x:nat => leq x x;;`
 - `leq_less :: x:nat => y:nat =>`
`leq x y => leq x (succ y);;`

Mutable State

- But there's one small problem

- Consider:

```
let x = ref 1 in
```

```
let g = (* build leq 1 !x object *) in
```

```
x := 0; g
```

- Now g proves $1 \leq 0$!
- The problem: cannot let dereferences into types

Purity

- Solution: only *pure* terms can index types
- Reads and writes of references are impure
 - Note: paper uses attributes instead of references
- Recursive constructs (like **fix**) are impure
 - Recursion in types makes type checking undecidable
- Pattern abstractions are impure
 - Evaluation with bound variables (in types) is unclear
 - Straightforward approach violates Church-Rosser
 - Can allow α -equivalence of patterns

Purity

- Two application rules:

$$\frac{\begin{array}{l} x \in FV(T_2) \qquad M_2 \text{ pure} \\ \Gamma \vdash M_1 : \Pi x : T_1.T_2 \quad \Gamma \vdash M_2 : T_1 \end{array}}{\Gamma \vdash M_1 M_2 : [M_2/x]T_2} \text{ t-pure-app}$$

$$\frac{\begin{array}{l} x \notin FV(T_2) \\ \Gamma \vdash M_1 : \Pi x : T_1.T_2 \quad \Gamma \vdash M_2 : T_1 \end{array}}{\Gamma \vdash M_1 M_2 : T_2} \text{ t-impure-app}$$

- Premise $x \in FV(T_2)$ of t-pure-app for determinism

insert Revisited

```
rec insert :: l:nat =c> u:nat =c> x:nat =c> L1:leq l x
           =c> L2:leq x u =c> T:bst l u =c> bst l u =
```

insert Revisited

```
rec insert :: l:nat =c> u:nat =c> x:nat =c> L1:leq l x
           =c> L2:leq x u =c> T:bst l u =c> bst l u =
```

(Only show left recursive case *)*

```
T \ bst ... data ... left right (* other vars omitted *)
  \ ... left:Ref (BST l data) ...->
```

insert Revisited

```
rec insert :: l:nat =c> u:nat =c> x:nat =c> L1:leq l x
           =c> L2:leq x u =c> T:bst l u =c> bst l u =
```

(Only show left recursive case *)*

```
T \ bst ... data ... left right (* other vars omitted *)
```

```
\ ... left:Ref (BST l data) ...->
```

```
match (compare x data) with
```

(compare returns leq x data or leq data (succ x) *)*

insert Revisited

```
rec insert :: l:nat =c> u:nat =c> x:nat =c> L1:leq l x
           =c> L2:leq x u =c> T:bst l u =c> bst l u =
```

(Only show left recursive case *)*

```
T \ bst ... data ... left right (* other vars omitted *)
  \ ... left:Ref (BST l data) ...->
```

match (compare x data) with

(compare returns leq x data or leq data (succ x) *)*

```
cmp \ cmp_leq L_x_d \ L_x_d:leq x data ->
```

```
  left := insert l data x L1 L_x_d !left;
```

```
  T (* return T *)
```

Dependent Patterns

- Final piece: typing dependent patterns
- Consider `cmp0 :: x:nat => leq 0 x`:
`rec cmp0 = x \ zero \ -> leq_eq zero |`
`x \ succ y \ -> leq_less zero y (cmp0 y);;`
- Bodies have types `leq 0 0` and `leq 0 (succ y)`
 - Substitution instances of what we want, `leq 0 x`
 - Need the following typing rule

$$\frac{\Gamma \vdash \rho : \Pi^c x : T_1.T_2 \quad \Gamma \vdash P : T_1 \quad \Gamma \vdash [P/x]M : [P/x]T_2}{\Gamma \vdash x \backslash P \backslash \Gamma \rightarrow M \mid \rho : \Pi^c x : T_1.T_2} \text{t-rho}$$

Conclusion and Future Work

- Language-based Approach to Functionally Correct Imperative Programming
- RSP1 is:
 - Dependent pattern matches + Dependent records +
 - First-order LF + Mutable state + General recursion
- Purity removes “bad” terms from types
- Future work:
 - What data structures can be verified with this approach?
 - Can we make more terms pure?