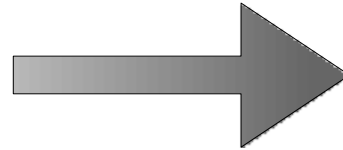


# Extensible Type Systems

# Problem

$$\begin{aligned}\psi(x) &= Axe^{-\frac{x^2}{L^2}} \\ \frac{\delta\psi}{\delta x} &= Ae^{-\frac{x^2}{L^2}} - \frac{2Ax^2}{L^2}e^{-\frac{x^2}{L^2}} \\ &= \left(1 - \frac{2x^2}{L^2}\right)\frac{\psi}{x} \\ \frac{\delta^2\psi}{\delta x^2} &= \left(-\frac{4x}{L^2}\right)\left(\frac{\psi}{x}\right) + \left(1 - \frac{2x^2}{L^2}\right)\left(-\frac{2\psi}{L^2}\right) \\ -\frac{2m}{\hbar^2}(E-U)\psi &= \psi\left(-\frac{4}{L^2} - \frac{2}{L^2} + \frac{4x^2}{L^2}\right) \\ U(x) &= \frac{\hbar^2}{2mL^2}\left(\frac{4}{L^2}x^2 - 6\right) \\ &= \frac{4\hbar^2}{2mL^4}x^2 - \frac{6\hbar^2}{2mL^2}\end{aligned}$$



```
int main(){
    int x = 2;
    char ** y = malloc(x*x);
    printf("%d\n", *y);
    Int z = x*x - y[0] + y[1];
    Int k = 1 - z / pow(x, 9);
    ...
}
```

1. Elegant?
2. Bugs?
3. Maintainable?

# Elegance

*The degree to which a concrete system can be mapped to its model.*

## Java

- Functions
- Classes
- Exceptions

## Haskell

- 1<sup>st</sup> class Functions
- Abstract Data Types
- Monads

Limited semantic abstractions

Limited type abstractions



Syntactic abstraction



# Solutions

## Domain Specific Language

- Tailored for the problem at hand

- Hard to build
- Not reusable

## Third party tool

- Easier to build
- Not tied to a single implementation

- Isolated from existing toolchain
- Hard to compose with other tools

# Proposed solution

## *Extensible type system*

A mechanism for providing allowing the user to create abstractions in the type domain.

## *Key idea*

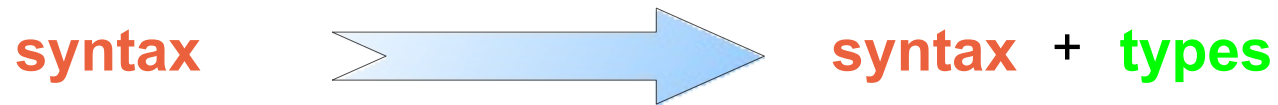
Leverage syntactic abstraction mechanisms to add invariants onto the abstract syntax tree

## *Research goals*

- Show how a type system can be defined in terms of modular pieces
- Understand the limits of type systems

# Solution Makeup

Extend the syntactic abstraction mechanism to support arbitrary invariants



Define an evaluator for those invariants

Compile time analysis

Define a core framework for existing invariants systems

C-style type system  
ML-style type system

# PLT Macros

```
(define (zoo animal)
  (if (animals? my-zoo animal)
      (lookup-animal my-zoo animal)
      (map (lambda (x)
            (get-animal my-zoo x))
           (all-animals my-zoo))))))
```

```
(define (zoo animal)
  (if (animals? my-zoo animal)
      (lookup-animal my-zoo animal)
      (map (lambda (x)
            (animal-type x
             (lambda (z)
              (animal-get my-zoo z)))
            (all-animals my-zoo))))))
```

**get-animal** is a function that accepts syntax and returns new syntax

$$f(\text{AST}) = \text{AST}'$$

```
(define-syntax (get-animal ast)
  ...
  (let ([ast' ...])
    ...
    ast'))
```

# Implementation

Add extra attributes to abstract syntax tree nodes that deal with a set of invariants

- Type constraints
- Type results

## Type constraints

Attached to expressions

**(+ a b)**

a has type integer  
b has type integer

## Type results

Expressions reduce to single types

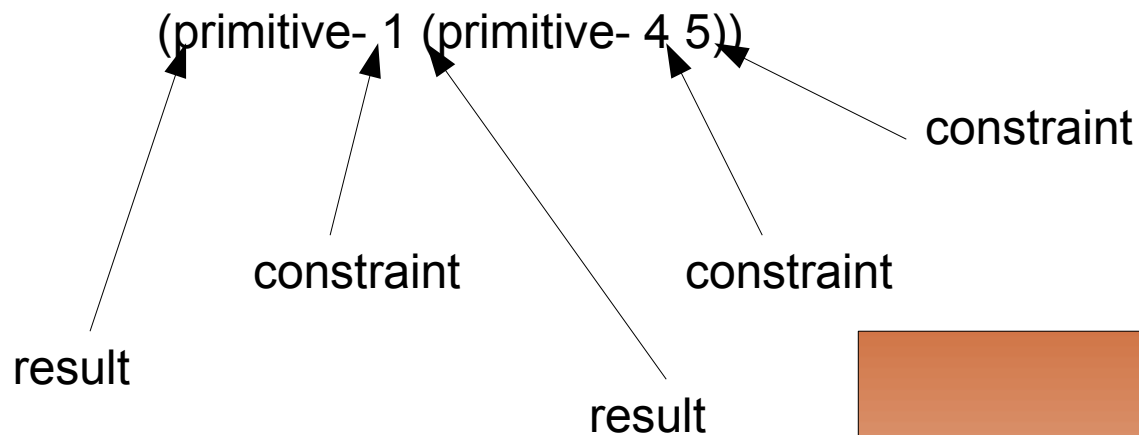
**(+ a b)**

Expression has type integer

# Example

```
(define-syntax (+ a b)
  (constraints ([a (has-type 'integer)]
               [b (has-type 'integer)]))
  #'(primitive-+ a b))
(results (type 'integer))))
```

(+ 1 (+ 4 5))



```
(has-type 4 'integer) (has-type 5 'integer)
-----
(has-type 1 'integer) (primitive-4 5) :- integer
-----
(primitive-1 (primitive-4 5)) :- integer
```

# Plan of work

## *Design the rest of the framework*

- Discover the right set of combinators
- Decide how macros can be overridden

## *Implement existing and theoretical type systems*

- Polymorphism
- Type inference
- Qualifiers (const, synchronized, etc.)

# Conclusion

- ★ The ability to create abstractions greatly increases a language's usefulness
- ★ We are going to extend syntactic abstraction to provide support for type system abstraction

*Future of language design*

