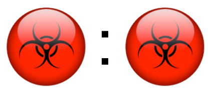
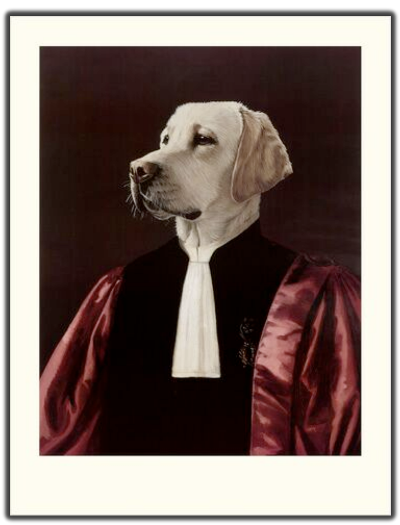
# Types Considered Harmful



Nate Foster, Michael Greenberg,
Benjamin C. Pierce
University of Pennsylvania

MFPS – May 23, 2008

## I have long advocated type systems...



2

# ...but I've changed my mind


3

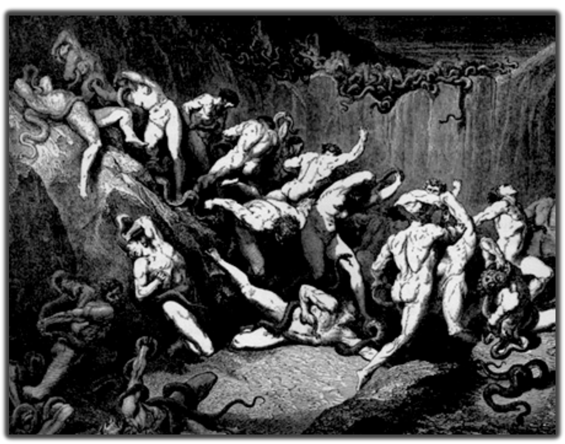
# Types are frustrating...

4

# painful...


5

# hellish...



6

## Down with types!



## Conclusion

**Scheme** is the optimal programming language!

## Thank you

Any questions?



## OK, just kidding

## But seriously...

- The talk:
  - Arguments *for* and *against* static type systems, especially very precise ones
  - The *Boomerang* language as a case study in the pros and cons of precise types
  - *Contracts* as a way of balancing concerns

11

## What's good about types?

12

## Why do types help?

- Complex definitions tend to be wrong when first written down
- In fact, not only wrong but *nonsensical*

Most programming errors are not subtle!



## Corollary

- Attempting to prove ***any* nontrivial theorem** about your program will expose lots of bugs
- The particular choice of theorem makes little difference!
- Typechecking is good because it proves *lots* and *lots* of little theorems about your program

Types good ⇒ More types better?

# What's bad about types?

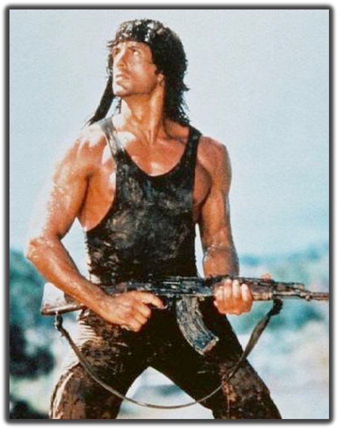
15

## "Strong types are for weak minds"

Does he look like he needs a type system?

16

## “Strong types are for weak minds”



Does he?

17

## “Strong types are for weak minds”

What about him?

18

## Types ⇒ memory safety ⇒ GC ⇒ slow

- The classic retort:

  Computers are fast; programmers are not

- The rational retort:

  Types enable better compiler analyses and make programs run *faster,* not slower

- The new retort:

  Java

> "If you can't make it fast and correct, make it fast."
> -- L. Cardelli
> [paraphrased]

19

## @$#%^&# cryptic compiler error messages

20

# Fancy types make the programmer's head hurt

- Type systems – especially very precise ones – force programmers to think rigorously about what they are doing
- This is good... up to a point!
  - Do we want languages where a PhD* is required to understand the library documentation?

Is it better for Jane Programmer to write ~20 more or less correct lines of code / day or ~0 perfect ones?

* two PhDs for Haskell

21

# Fancy types make *my* head hurt

- Complex type systems can lead to complex language definitions
- Easy to blow the overall complexity budget

22

# I don't know when to stop typing

- Why is Hindley-Milner such a sweet spot?
- One reason: A term's HM principal type is the *most general theorem* that can be expressed in the "program logic" of the type system

The Library Problem

23

# Precise types reveal too much

- Precise types can force details of issues like resource usage into interfaces

The Visible Plumbing Problem

cf. Morrisett's story about region types in Cyclone...

And similar stories with security types...

24

# Fancy typecheckers require a lot of coddling and cajoling

- Type structure is calculated from program structure

→ So program structure must be carefully designed to give rise to the desired type structure!

→ The Intersection Problem

25

# Bottom line

- Types – especially very precise ones – are a mixed blessing in practice

*Precision can be useful or even necessary*

*But we need to stay awake to some serious pragmatic issues*

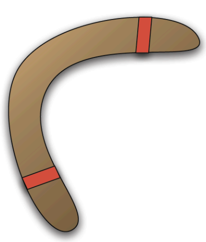
∴ More research is needed!

26

# Rest of talk…

- **Boomerang** language design as an example of
  1. the need for very precise types
  2. some of the technical problems they raise
- **Contracts** as an attractive way of addressing some of these issues

27



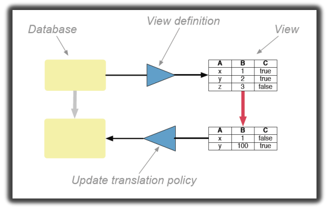
# Boomerang

» life with a very precise type system…

28

## Background: Bidirectional programs

- Computing is full of situations where we want to compute some function, *edit* the output, and *"push the edits back" through the function* to obtain a correspondingly edited input.




29

## Bijective Lenses

Boomerang also handles non-bijective lenses, but that's another story…

A **bijective lens** (or, for this talk, just **lens**) *l* from *S* to *T* is a pair of total functions

$$l.get \in S \rightarrow T$$
$$l.put \in T \rightarrow S$$

such that

$$l.get\ (l.put\ t) = t$$
$$l.put\ (l.get\ s) = s.$$

The set of lenses from *S* to *T* is written $S \Leftrightarrow T$.

30

## Bidirectional *languages*

- How do we write down lenses?
- **Bad answer**: Write down two functions and prove that they are inverses.
- **Better answer**: Build big lenses from smaller ones. (I.e., design a programming language where every expression denotes a lens.)
  - Single description
  - Bijectivity guaranteed by construction

31

## A simple bidirectional language

- Let's design a little language for bijective **string** transformations…

32

# Copying

$$copy\ S \in S \Longleftrightarrow S$$

$$\begin{aligned}(copy\ S).get\ s &= s\\(copy\ S).put\ s &= s\end{aligned}$$

33

# Composition

$$\frac{l \in S \Longleftrightarrow T \qquad k \in T \Longleftrightarrow U}{l\,;k \in S \Longleftrightarrow U}$$

$$\begin{aligned}(l\,;k).get\ s &= k.get\ (l.get\ s)\\(l\,;k).put\ u &= l.put\ (k.put\ u)\end{aligned}$$

So lenses form a category… whew!

34

## Inversion

$$\frac{l \in S \Longleftrightarrow T}{\mathit{invert}\ l \in T \Longleftrightarrow S}$$

$$\begin{aligned}(\mathit{invert}\ l).\mathit{get}\ t &= l.\mathit{put}\ t \\ (\mathit{invert}\ l).\mathit{put}\ s &= l.\mathit{get}\ s\end{aligned}$$



## Rewriting

$$s \Leftrightarrow t \in \{s\} \Longleftrightarrow \{t\}$$

$$\begin{aligned}(s \Leftrightarrow t).\mathit{get}\ s &= t \\ (s \Leftrightarrow t).\mathit{put}\ t &= s\end{aligned}$$

# Concatenation

$$\frac{l_1 \in S_1 \Longleftrightarrow T_1 \qquad l_2 \in S_2 \Longleftrightarrow T_2 \qquad S_1 \cdot^! S_2 \qquad T_1 \cdot^! T_2}{l_1 \cdot l_2 \in S_1 \cdot S_2 \Longleftrightarrow T_1 \cdot T_2}$$

$$(l_1 \cdot l_2).get\ (s_1 \cdot s_2) = (l_1.get\ s_1) \cdot (l_2.get\ s_2)$$
$$(l_1 \cdot l_2).put\ (t_1 \cdot t_2) = (l_1.put\ t_1) \cdot (l_2.put\ t_2)$$

$S_1 \cdot^! S_2$ means "the concatenation of $S_1$ and $S_2$ is uniquely splittable"

37

# Iteration

$$\frac{l \in S \Longleftrightarrow T \qquad S^{!*} \qquad T^{!*}}{l^* \in S^* \Longleftrightarrow T^*}$$

$$(l^*).get\ (s_1 \cdots s_n) = (l.get\ s_1) \cdots (l.get\ s_n)$$
$$(l^*).put\ (t_1 \cdots t_n) = (l.put\ t_1) \cdots (l.put\ t_n)$$

38

# Union

$$\frac{l_1 \in S_1 \Longleftrightarrow T_1 \qquad l_2 \in S_2 \Longleftrightarrow T_2 \\ S_1 \cap S_2 = \emptyset \qquad T_1 \cap T_2 = \emptyset}{l_1 \mid l_2 \in S_1 \cup S_2 \Longleftrightarrow T_1 \cup T_2}$$

$$(l_1 \mid l_2).get\ s = \begin{cases} l_1.get\ s & \text{if } s \in S_1 \\ l_2.get\ s & \text{if } s \in S_2 \end{cases}$$

$$(l_1 \mid l_2).put\ a = \begin{cases} l_1.put\ t & \text{if } t \in T_1 \\ l_2.put\ t & \text{if } t \in T_2 \end{cases}$$

39

# Example: An escaping lens

```
let XML_ESC : regexp = "&lt;" | "&gt;" | "&amp;" | [^<>&]

let escape_xml_char : (lens in ANYCHAR <=> XML_ESC) =
    '<' <=> "&lt;"
  | '>' <=> "&gt;"
  | '&' <=> "&amp;"
  | copy (ANYCHAR - [<>&])


let ANY : regexp = ANYCHAR*
let XML_ESC_STRING : regexp = XML_ESC*

let escape_xml : (lens in ANY <=> XML_ESC_STRING ) =
  escape_xml_char*

test escape_xml.get
  <<
    <hello"world>
  >>
=
  <<
    &lt;hello"world&gt;
  >>
```

char escaping lens

string escaping lens

unit test

40

# Another escaping lens

```
let ESC_SYMBOL : regexp = "\\\"" | "\\\\" | [^\\""]

let escape_quotes_char : (lens in ANYCHAR <=> ESC_SYMBOL) =
    '"' <=> "\\\""
  | '\\' <=> "\\\\"
  | copy (ANYCHAR - [\\""])


let ESC_STRING : regexp = ESC_SYMBOL*
let escape_quotes_string : (lens in ANY <=> ESC_STRING ) =
  escape_quotes_char*

test escape_quotes_string.get
  <<
    <hello"world>
  >>
=
  <<
    <hello\"world>
  >>
```

A similar lens for a different escaping convention (escaping quotes and backslashes)

41

# A composite escaping lens

```
let quotes_to_xml : (lens in ESC_STRING <=> XML_ESC_STRING) =
  (invert escape_quotes_string) ; escape_xml
```

invert quote-escaper

and compose

with XML-escaper

```
test quotes_to_xml.get
  <<
    <hello\"world>
  >>
=
  <<
    &lt;hello"world&gt;
  >>
```

the composite lens maps from quote-escaped strings to XML-escaped strings

42

## Regular expressions as types

- Types of compound expressions are calculated compositionally from types of subexpressions
- Typechecking can be carried out mechanically
  - ... Requires devoting some care to the engineering!
- Type soundness = totality + bijectivity (!)

43

## Code Reuse

44

## Building larger programs

- Programming with these combinators is fun for a while, but it loses its charm as programs become larger
- Need facilities for naming, abstraction, code reuse…
  - i.e., we want a real programming language



## Boomerang

Lenses + Lambdas
=
Boomerang

# An escaping library

A generic function for building character-escaping lenses:

```
let escape_char (raw:char) (esc:string)
    (R:regexp where not ((matches R raw) || (matches R esc)))
    : (lens in (raw | R) <=> (esc | R)
=
    ( raw <=> esc | copy R )
```

The XML-escaping lens again:

```
let escape_xml_char : (lens in ANYCHAR <=> XML_ESC) =
  ( escape_char '&' "&amp;" [^&]
  ; escape_char '<' "&lt;"  ([^&<] | "&amp;")
  ; escape_char '>' "&gt;"  ([^&<>] | "&amp;" | "&lt;") )
```

47

# Or better...

A more uniform version of the XML-escaping lens:

```
let escape_xml : (lens in ANY <=> XML_ESC_STRING ) =
  let l1 = escape_char '&' "&amp;" [^&] in
  let l2 = escape_char '<' "&lt;"  ((codomain_type l1) - "<") in
  let l3 = escape_char '>' "&gt;"  ((codomain_type l2) - ">") in
  (l1;l2;l3)*
```

48

# Or better yet...

A function mapping a *list* of pairs of (character, escape code) to an escaping lens:

```
let escape_chars
      (esc:char)
      (pairs: (char * string) List.t where
            contains_esc_char esc pairs
         && no_repeated_esc_codes pairs)
   : (lens in ANY <-> (escaped esc pairs)* ) =
  let l : lens =
    List.fold_left{char * string}{lens}
      (fun (li:lens) (p:char * string) ->
         let cj,sj = p in
         let lj = escape_char cj (esc . sj) ((codomain_type li) - cj) in
         li;lj)
      (copy ANYCHAR) pairs in
    l*

let escape_xml : lens =
  escape_chars '&' [('&',"amp;");('<',"lt;");('>',"gt;")]
```

49

# Taking stock

- The requirements of lens programming have led us to a type system with:
  - dependent function types
  - regular expressions (for lenses)
  - type refinements

    ```
    (R:regexp where not ((matches R raw)
                        || (matches R esc)))
    ```
  - polymorphism (for lists)

=

- This precision is *necessary* to support code reuse while guaranteeing bijectiveness and totality.
- But I have no idea how to write a **typechecker** for this beast!

50

# Idea

- Split typechecking into multiple phases
  - **Phase I**: Function types and polymorphism
    - Typecheck functional program, treating regular expressions and refinement types as uninterpreted "blobs"
  - **Phase II**: Refinements and regular expressions
    - Execute functional program to produce a lens, checking type refinements and preconditions of lens primitives as they are encountered
  - **Phase III**: Evaluation
    - Apply resulting "straight line lens" to its string argument

51

# Pointing the finger

- Problem: We've taken a static type analysis and turned it into a dynamic check
    - Not so bad in terms of *when* type errors appear (always during Phase I or II)
    - Not so good in terms of *where* they appear
- When precise type checking fails for a lens-assembling primitive (union, concatenation, etc.), all we can do is print a stack trace
  - But this is *anti-modular!* To debug a stack trace, you have to look at all the modules between the one that failed that the one that actually caused the problem.

We need one more idea…

52

# Contracts



## The Big Picture

- Postpone some static checks to runtime as dynamic casts

refinement type

```
Even = { x:Int | x mod 2 = 0 }

(<Even⇐Int> 2)
➥ 2
```

contract

base type

# The Big Picture

- When a contract violation is detected, the program location (blame label) of the contract is "blamed"

violated contract

blame label

```
(<Even⇐Int>^b 3)
➥ b is blamed!
```



# Looks simple!

But actually there are some subtleties…

# Higher-order contracts

- Contracts at functional types

  $<T_1 \rightarrow T_2 \Leftarrow S_1 \rightarrow S_2>\ f$

  cannot be checked directly. Instead, they are compiled into separate checks for the domain and codomain.

- Surprisingly, there are *two* ways to do this!

$<T_1 \rightarrow T_2 \Leftarrow S_1 \rightarrow S_2> \ \sim \ \lambda f.\ \lambda x.\ <T_2 \Leftarrow S_2>\ (f\ (<S_1 \Leftarrow T_1>\ x))$ ("contravariant")

$<T_1 \rightarrow T_2 \Leftarrow S_1 \rightarrow S_2> \ \sim \ \lambda f.\ \lambda x.\ <T_2 \Leftarrow S_2>\ (f\ (<T_1 \Leftarrow S_1>\ x))$ ("covariant")

- More surprisingly, both are reasonable!

57

# What is the type of a contract?

Makes sense in precisely typed languages, where refinements ⊆ types.

$<T \Leftarrow S> \ \in \ S \rightarrow (T \cup \{blame\})$ "**manifest**" contracts (visible in type of result)

$<T \Leftarrow S> \ \in \ S \rightarrow (S \cup \{blame\})$ "**latent**" contracts (hidden in type of result)

Makes sense in untyped or simply typed languages, where types are not expressive enough to talk about refinements.

58

(Part of)
^ The contract landscape

| Latent contracts | | Manifest contracts |
| --- | --- | --- |
| *Untyped* | *Simple static types* | *Precise static types* |
| Scheme contracts<br>Findler, Felleisen '02<br>Blume, McAllester '06<br>Findler, Blume '06 | Quasi-Static Typing<br>Thatte, '89<br><br>Typed Contracts for Haskell<br>Hinze, Jeuring, Löh '06<br><br>Typed Scheme<br>Tobin-Hochstadt, Felleisen '08 | Gradual Typing<br>Siek, Taha '06<br><br>Hybrid Types<br>Flanagan '06<br><br>Sage<br>Knowles, Tomb, Gronski, Freund, Flanagan '06<br><br>"Well typed programs can't be blamed"<br>Wadler, Findler '08<br><br>Boomerang |

60

# Contracts in Boomerang

(Ongoing work!)

- We take the "manifest contracts" approach
  - Rich language of types
  - Three-phase execution model
    1. Check simple types
    2. Execute functional code to produce a lens (checking contracts)
    3. Execute lens
  - Contracts assign blame to a program location when a dynamic check fails in Phase II

61

# Conclusion

- A precise type system with contracts can offer an attractive compromise between expressiveness of types, dynamism of checking, and language complexity
- But many technical challenges remain…
  - How do we state "type soundness"?
  - What is the algebra of blame?
  - How do we make programs run fast enough with all these dynamic checks ?
  - What are the pragmatics of programming in such a language?
    - How to deal with the *Intersection Problem*, the *Library Problem*, the *Visible Plumbing Problem*, etc.?

62

# Larger Challenges

- **Complex programs** have interesting properties, which require **complex contracts** to check
  - Contracts are software!
  - Need suitable language design, software engineering methodologies, etc.
- Interesting connections with testing
  - Every function needs a **unit test**, *and so does its contract!*

63

# Finishing up...



# What have we learned?

## More broadly…

- Mechanical checks of simple properties enormously improve software quality
  - **Types** ~ *General but weak* theorems (usually checked statically)
  - **Contracts** ~ *General and strong* theorems, checked dynamically for particular instances that occur during regular program operation
  - **Unit tests** ~ *Specific and strong* theorems, checked quasi-statically on particular "interesting instances"
- Needed: Better ways of integrating these different sorts of checks

66

## Thank you!

- **Things to play with:** Boomerang sources/demos:

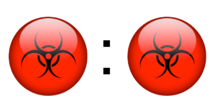
http://www.seas.upenn.edu/~harmony

- **Collaborators:**

  Nate Foster and Michael Greenberg

- **Acknowledgments:**

  James McKinna, Greg Morrisett, Conor McBride, Andrew Myers, Alexandre Pilkiewicz, Stephanie Weirich, Penn PL Club



67