Interlanguage Migration

From Scripts to Programs

Sam Tobin-Hochstadt and Matthias Felleisen

Northeastern University

DLS 2006

A Story

About a programmer

Who needed to manage his budget

And so, he wrote a simple little program

In his favorite (dynamic) language:

PLT Scheme

He did well for himself, and now he needed to manage some investments as well

So he added more pieces to his program

Then he decided he wanted to access the system remotely

So he added a web front-end

He kept it all nicely organized

Since, after all, the program was managing

$5,000

Soon, his friends noticed that he was making lots of money on the stock market

And they wanted to use his system as well

And soon the system was managing

$50,000

Of course, having his friends use his system entailed new responsibilites

Like testing ...

And lots more code

Fortunately, he was very productive in his favorite language

Which was good - after all, the system managed

$500,000

But his friends

(and their friends,

and their grandmothers,

and their grandmothers' friends)

kept wanting more features

To help them manage

$5,000,000

But he was still very productive

So the system handled

$50,000,000

very nicely

Then, one day, the suits gave our hero a call

The suits paid him a lot of money for his application

But then the suits took a look at all the code

They said "Some of this code is very important!"

"We need assurance that the key portions of this code are safe!"

So, they rewrote the whole application in C++

How can we avoid this (all-too-common) result?

How can we avoid this (all-too-common) result?

How can we statically check parts of our programs - without rewriting them?

Overview

Goals

- Migrate a program in a dynamic language by adding some static checking

Goals

- Migrate a program in a dynamic language by adding some static checking
- Don't rewrite the whole thing

Goals

- Migrate a program in a dynamic language by adding some static checking
- Don't rewrite the whole thing
- Use the same language everywhere

Goals

- Migrate a program in a dynamic language by adding some static checking
- Don't rewrite the whole thing
- Use the same language everywhere
- Continue maintaining the code

Goals

- Migrate a program in a dynamic language by adding some static checking
- Don't rewrite the whole thing
- Use the same language everywhere
- Continue maintaining the code
- Be sure of what we get in the end

Assumptions

- All code is in modules

Assumptions

- All code is in modules
- Each module can be typed independently

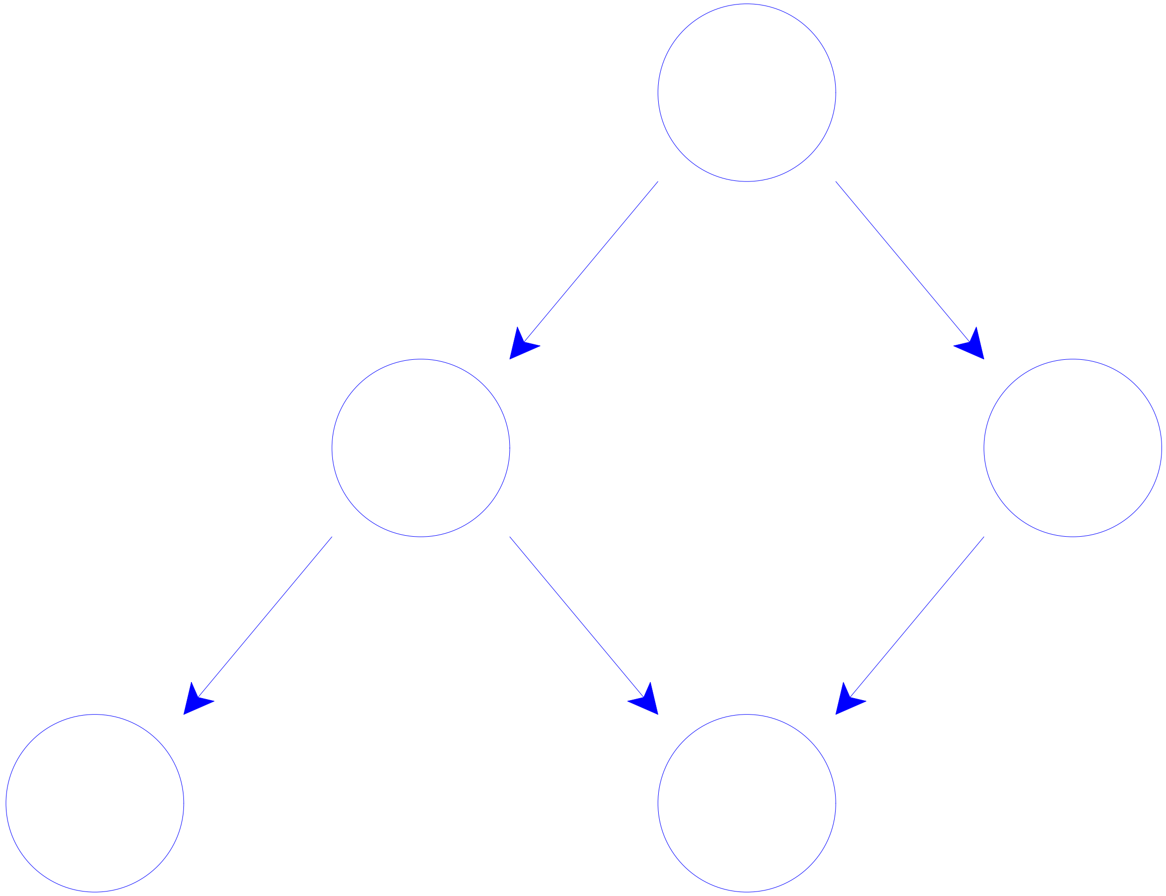
Assumptions

- All code is in modules
- Each module can be typed independently
- We have a type system that can check lots of the code

Assumptions

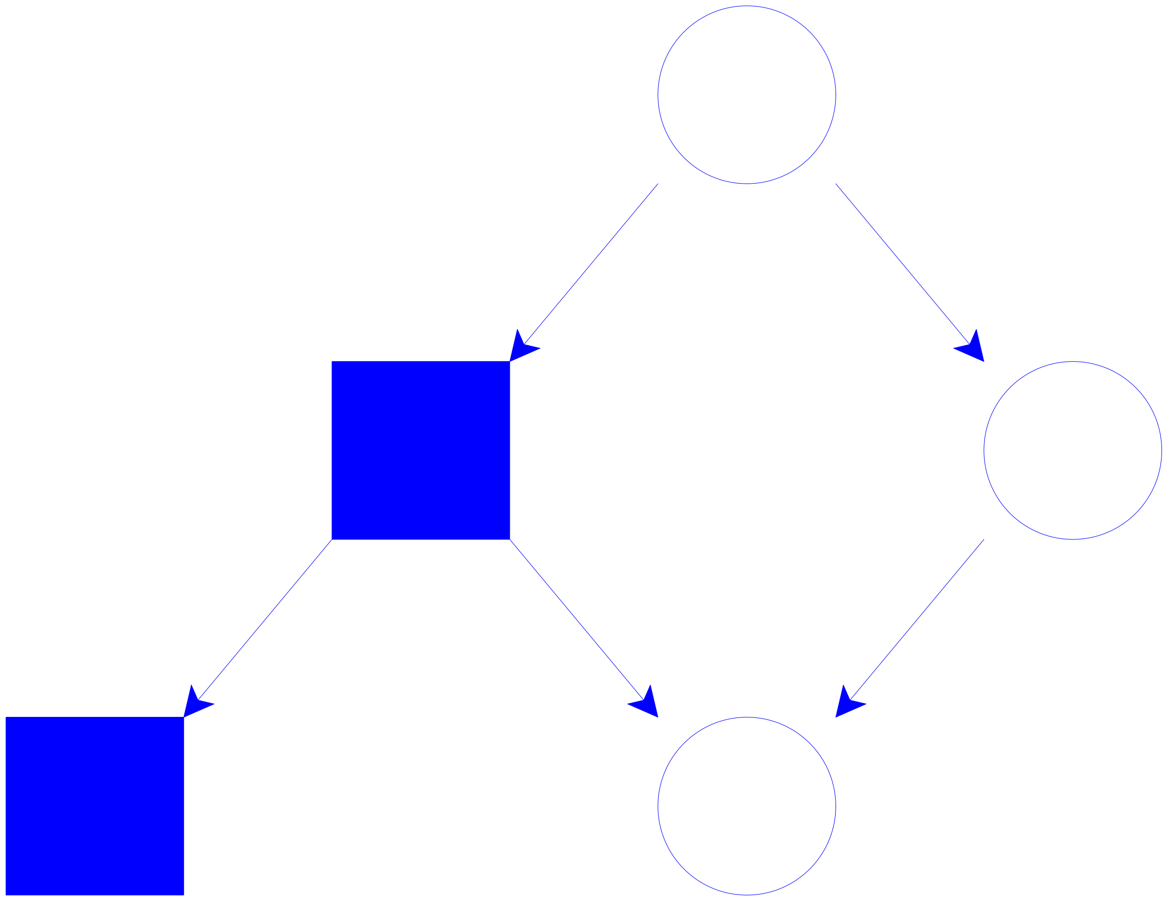
- All code is in modules
- Each module can be typed independently
- We have a type system that can check lots of the code
- We add types a module at a time

Migration



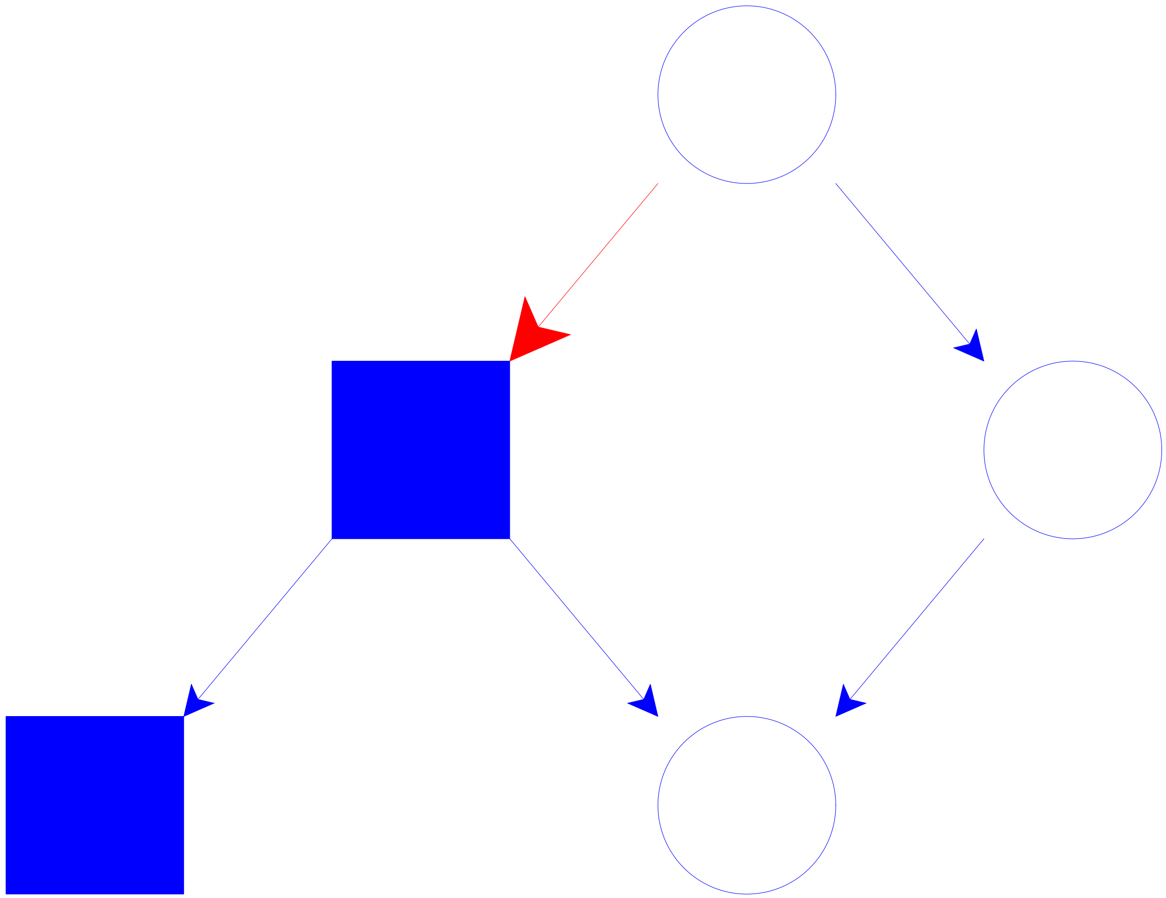
A system built out of untyped modules

Migration



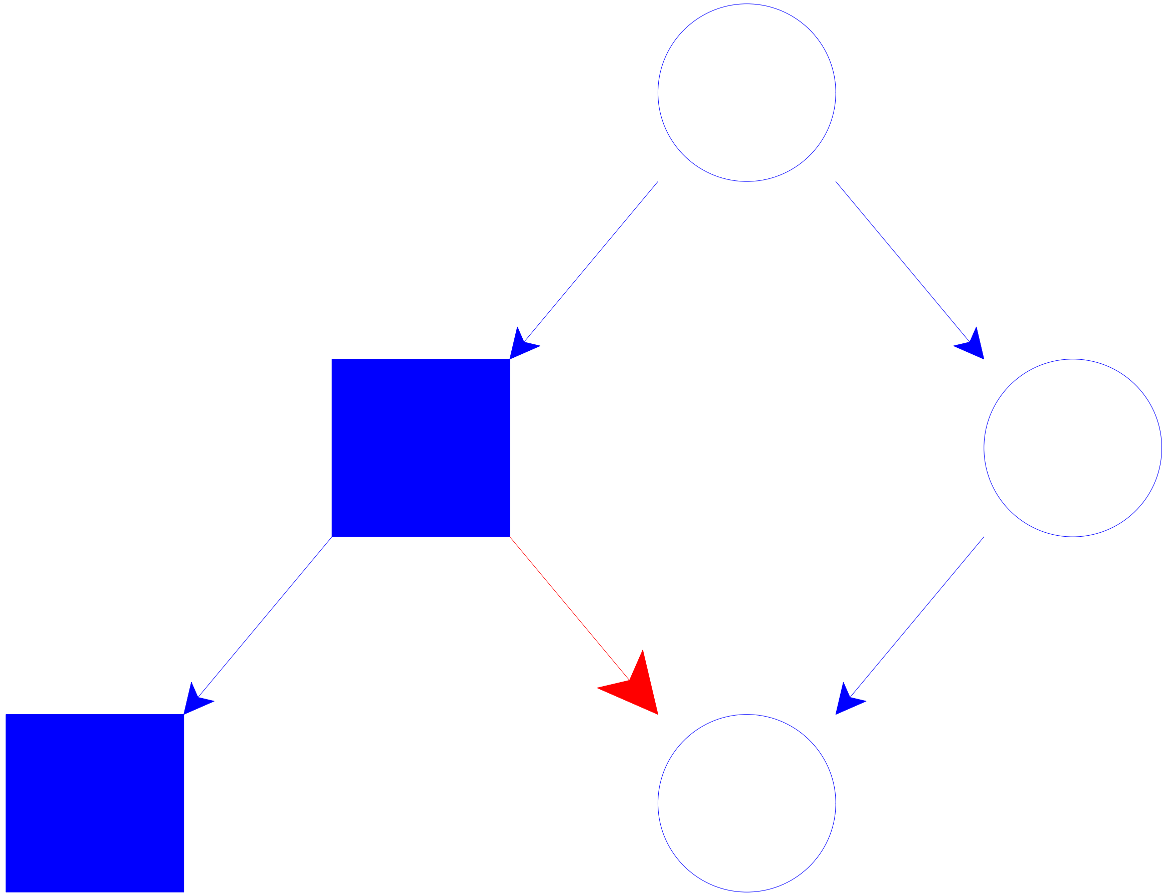
Add types to some of the modules

Migration



Untyped code depending on typed code

Migration



Dependencies go both ways

Questions

- What do we check?

- How much code change is acceptable?

- How do we integrate typed and untyped code?

Questions

- What do we check?
    - Precisely what modern type systems can check:
    - That we don't misapply operations - those we define, or those the language defines
- How much code change is acceptable?
- How do we integrate typed and untyped code?

Questions

- What do we check?
  - Precisely what modern type systems can check:
  - That we don't misapply operations - those we define, or those the language defines
- How much code change is acceptable?
  - As little as possible, as much as neccessary
- How do we integrate typed and untyped code?

Questions

- What do we check?
  - Precisely what modern type systems can check:
  - That we don't misapply operations - those we define, or those the language defines
- How much code change is acceptable?
  - As little as possible, as much as neccessary
- How do we integrate typed and untyped code?
  - Flows in both directions
  - Callbacks

How do we do it?

Specify the language of particular modules

How do we do it?

Specify the language of particular modules

Enforce contracts at module boundaries

How do we do it?

Specify the language of particular modules

Enforce contracts at module boundaries

Infer required contracts

Modules

A group of definitions, with explicit export of some of them

Imports specified explicity

Internal linking

Modules

A group of definitions, with explicit export of some of them

Imports specified explicity

Internal linking

A close resemblance to the {PLT Scheme, Python, Ruby, ...} module systems

Modules

Each module is either typed or untyped

Typed modules specify the types of their exports

Either kind of module can refer to the other kind

Contracts

Dynamic checks on steroids

Allow us to check both data and functions

Higher-order contracts allow callbacks (and objects) to work in both directions

Contracts allow richer specifications

Contracts

Dynamic checks on steroids

Allow us to check both data and functions

Higher-order contracts allow callbacks (and objects) to work in both directions

Contracts allow richer specifications

See [Findler & Felleisen, OOPSLA 2001]

Contracts

When we encounter a boundary-crossing, one of the sides must have a type

Convert that type to a contract

Add the contract to the interface of the exporting module

Examples

Simple Example

```
(module fast-mul mzscheme
  (provide fast-mul)

  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
```

Simple Example

```
(module fast-mul mzscheme
  (provide fast-mul)

  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
(module interest mzscheme
  (define (interest x)
    (+ x (fast-mul x 0.05))))
```

Simple Example

```
(module fast-mul mzscheme
  (provide fast-mul)

  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
```

```
(module interest typed-scheme
  (define: (interest (x : number)) : number
    (+ x (fast-mul x 0.05))))
```

Simple Example

```
(module fast-mul mzscheme
  (provide/contract fast-mul
                    (number number . -> . number))
  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
```

```
(module interest typed-scheme
  (define: (interest (x : number)) : number
    (+ x (fast-mul x 0.05))))
```

Simple Example

```
(module fast-mul mzscheme
  (provide/contract fast-mul
                    (number number . -> . number))
  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
```

```
(module interest typed-scheme
  (define: (interest (x : number)) : number
    (+ x (fast-mul x 0.05))))
```

But how did we know the type of `fast-mul`?

Simple Example

```
(module fast-mul mzscheme
  (provide/contract fast-mul
                    (number number . -> . number))
  (define (fast-mul a b) (if (zero? a) 0 (* a b))))
```

```
(module interest typed-scheme
  (define: (interest (x : number)) : number
    (+ x (fast-mul x 0.05))))
```

But how did we know the type of **`fast-mul`**?

From how **`fast-mul`** is used in the typed module, we can infer the required type and contract.

Contracts that fail

```
(module add-interest-mod mzscheme
  (require inc-mod interest)
  (define (add-interest balance)
    (increment (interest balance))))

(module inc-mod mzscheme
  (provide increment)
  (define increment 999))

(module main mzscheme
  (require add-interest-mod)
  (add-interest 10000.0))
```

Contracts that fail

```
(module add-interest-mod typed-scheme
  (require inc-mod interest)
  (define: (add-interest (balance : number)) : number
    (increment (interest balance))))
```

```
(module inc-mod mzscheme
  (provide increment)
  (define increment 999))
```

```
(module main mzscheme
  (require add-interest-mod)
  (add-interest 10000.0))
```

Contracts that fail

```
(module add-interest-mod typed-scheme
  (require inc-mod interest)
  (define: (add-interest (balance : number)) : number
    (increment (interest balance))))

(module inc-mod mzscheme
  (provide/contract increment (number . -> . number))
  (define increment 999))

(module main mzscheme
  (require add-interest-mod)
  (add-interest 10000.0))
```

Contracts that fail

```
(module add-interest-mod typed-scheme
  (require inc-mod interest)
  (define: (add-interest (balance : number)) : number
    (increment (interest balance))))
```

```
(module inc-mod mzscheme
  (provide/contract increment (number . -> . number))
  (define increment 999))
```

```
(module main mzscheme
  (require add-interest-mod)
  (add-interest 10000.0))
```

Now `main` will fail when run, because `increment` does not meet its contract.

Handling incompatible uses

```
(module n-mod mzscheme
  (require inverse-mod)
  (define n
    (if (not (inverse true))
        (inverse 5)
        7)))

(module inverse-mod mzscheme
  (provide inverse)
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (inverse true))
        (inverse 5)
        7)))
```

```
(module inverse-mod mzscheme
  (provide inverse)
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (inverse true))
        (inverse 5)
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse            )
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

What contract could we add to **inverse**?

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (inverse true))
        (inverse 5)
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ((or/c boolean number)
                              . -> .
                              (or/c boolean number)))
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (inverse true))
        (inverse 5)
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ((or/c boolean number)
                              . -> .
                              (or/c boolean number)))
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

But that's insufficient for safety

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (inverse true))
        (inverse 5)
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ((or/c boolean number)
                              . -> .
                              (or/c boolean number)))
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

But that's insufficient for safety

```
(define (inverse x)
  (if (boolean? x) 1 true))
```

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (boolean <= (inverse true)))
        (number <= (inverse 5))
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ((or/c boolean number)
                              . -> .
                              (or/c boolean number)))
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Adding casts recovers safety

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse-mod)
  (define: n : number
    (if (not (boolean <= (inverse true)))
        (number <= (inverse 5))
        7)))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ((or/c boolean number)
                             . -> .
                             (or/c boolean number)))
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Adding casts recovers safety

Can we avoid casts?

Handling incompatible uses

```
(module n-mod typed-scheme
  (require inverse1 inverse2)
  (define: n : number
    (if (not (inverse1 true))
        (inverse2 5)
        7)))
```

```
(module inverse1 mzscheme
  (require inverse-mod)
  (provide/contract inverse1 (boolean . -> . boolean))
  (define inverse1 inverse))
(module inverse2 mzscheme
  (require inverse-mod)
  (provide/contract inverse2 (number . -> . number))
  (define inverse2 inverse))
```

```
(module inverse-mod mzscheme
  (provide/contract inverse ---)
  (define (inverse x)
    (if (boolean? x) (not x) (* x -1))))
```

Theoretical Contributions

Modeling our system

Start with the λ-calculus with numbers

Modeling our system

Start with the λ-calculus with numbers

Add modules and contracts

Modeling our system

Start with the λ-calculus with numbers

Add modules and contracts

Add simple types and typed modules

Modeling our system

Start with the λ-calculus with numbers

Add modules and contracts

Add simple types and typed modules

Define a migration process with inference

Theorems

What can we prove about such a system?

Theorems

What can we prove about such a system?

- Programs in the untyped portion can go wrong
- But the typed portions should be safe

Theorems

What can we prove about such a system?

- Programs in the untyped portion can go wrong
- But the typed portions should be safe

Use the blame annotations from contracts to track where errors occur

Prove that all runtime type errors are blamed on untyped code

Contributions

Theoretical Contributions

- A solid foundation for interlanguage migration
- Reformulating type soundness for mixed programs

Contributions

Theoretical Contributions

- A solid foundation for interlanguage migration
- Reformulating type soundness for mixed programs

Practical Contributions

- A framework for designing systems

Contributions

Theoretical Contributions

- A solid foundation for interlanguage migration
- Reformulating type soundness for mixed programs

Practical Contributions

- A framework for designing systems
- An implementation of the system for PLT Scheme

Related Work

Soft Typing

- Fagan, Wright, Henglein, Flanagan, Meunier, Aiken, and many more

Related Work

Soft Typing

- Fagan, Wright, Henglein, Flanagan, Meunier, Aiken, and many more

Type Dynamic

- Abadi et al, Siek & Taha, Baars & Sweirstra, Leroy & Mauny

Related Work

Soft Typing

- Fagan, Wright, Henglein, Flanagan, Meunier, Aiken, and many more

Type Dynamic

- Abadi et al, Siek & Taha, Baars & Sweirstra, Leroy & Mauny

Type systems for dynamic languages

- Strongtalk [Bracha], Erlang [Marlow & Wadler]

Conclusion

We can avoid C++ and keep using our languages

Modular migration of programs allows for flexibility

Need for new type systems to support dynamic languages

Conclusion

We can avoid C++ and keep using our languages

Modular migration of programs allows for flexibility

Need for new type systems to support dynamic languages

- Create one for your favorite language!

Thank You

http://www.ccs.neu.edu/home/samth