
Playing With Fire: Mutation and Quantified Types

CIS670, University of Pennsylvania
2 October 2002

Dan Grossman
Cornell University

Some context...

- **You've** been learning beautiful math about the power of abstraction (e.g., soundness, theorems-for-free)
- **I've** been using quantified types to design Cyclone, a safe C-like language
- We both need to integrate **mutable data** very carefully

Getting burned...

From: **Dan Grossman**
Sent: Thursday, August 02, 2001 8:32 PM
To: **Gregory Morrisett**
Subject: **Unsoundness Discovered!**

In the spirit of recent worms and viruses, please compile the code below and run it. Yet another interesting combination of **polymorphism**, **mutation**, and **aliasing**. The best fix I can think of for now is ...

Getting burned... decent company

From: Xavier Leroy

Sent: Tue, 30 Jul 2002 09:58:33 +0200

To: John Prevost

Cc: Caml-list

Subject: Re: [Caml-list] Serious
typechecking error involving new
polymorphism (crash)

...

Yes, this is a serious bug with
polymorphic methods and fields. Expect a
3.06 release as soon as it is fixed.

...

The plan...

- C meets α
 - It's not about syntax
 - There's much more to Cyclone
- Polymorphic references
 - As seen from Cyclone (unusual view?)
 - Applied to ML (solved since early 90s)
- Mutable existentials
 - The original part
 - April 2002
- Breaking parametricity [Pierce]

Taming C

- Lack of *memory safety* means code cannot enforce modularity/abstractions:

```
void f() { *((int*)0xBAD) = 123; }
```

- What might address `0xBAD` hold?
- Memory safety is crucial for your favorite policy

No desire to compile programs like this

Safety violations rarely local

```
void g(void**x, void*y) ;  
  
int y = 0;  
int *z = &y;  
g(&z, 0xBAD) ;  
*z = 123;
```

- Might be safe, but not if `g` does `*x=y`
- Type of `g` enough for separate code generation
- Type of `g` not enough for separate safety checking

What to do?

- Stop using C
 - YFHLL is *usually* a better choice
- Compile C more like Scheme
 - type fields, size fields, live-pointer table, ...
 - fail-safe for legacy whole programs
- Static analysis
 - very hard, less modular
- Restrict C
 - not much left

A combination of techniques in a new language

Quantified types

- Must compensate for banning `void*`
- But represent data and access memory as in C
“If it looks like C, it acts like C”
- Type variables help a lot, but a bit different than in ML

“Change void* to alpha”

```
struct L {
    void* hd;
    struct L* t1;
};
typedef
struct L* l_t;

l_t
map(void* f(void*),
    l_t);

l_t
append(l_t,
        l_t);
```

```
struct L<`a> {
    `a hd;
    struct L<`a>* t1;
};
typedef
struct L<`a>* l_t<`a>;

l_t<`b>
map<`a, `b>(`b f(`a),
            l_t<`a>);

l_t<`a>
append<`a>(l_t<`a>,
           l_t<`a>);
```

Not much new here

- **struct Lst** is a recursive type constructor:
$$L = \lambda\alpha. \{ \alpha \text{ hd}; (L \alpha)^* \text{ tl}; \}$$
- The functions are polymorphic:
$$\text{map} : \forall\alpha, \beta. (\alpha \rightarrow \beta, L \alpha) \rightarrow (L \beta)$$
- Closer to C than ML
 - less type inference allows first-class polymorphism and polymorphic recursion
 - data representation restricts `a` to pointers, `int` (why not structs? why not `float`? why `int`?)
- Not C++ templates

Existential types

- Programs need a way for “call-back” types:

```
struct T {  
    int (*f) (int, void*);  
    void* env;  
};
```

- We use an existential type (simplified):

```
struct T { <`a>  
    int (*f) (int, `a);  
    `a env;  
};
```

more C-level than baked-in closures/objects

Existential types cont'd

```
struct T { <`a>
  int (*f) (int, `a);
  `a env;
};
```

- ``a` is the *witness type*
- creation requires a “consistent witness”
- type is just `struct T`

- use requires an explicit “unpack” or “open”:

```
int apply(struct T pkg, int arg) {
  let T{<`b> .f=fp, .env=ev} = pkg;
  return fp(arg, ev);
}
```

The plan...

- C meets α
 - It's not about syntax
 - There's much more to Cyclone
- Polymorphic references
 - As seen from Cyclone (unusual view?)
 - Applied to ML (solved since early 90s)
- Mutable existentials
 - The original part
 - April 2002
- Breaking parametricity [Pierce]

Mutation

- $e_1 = e_2$ means:
 - Left-evaluate e_1 to a location
 - Right-evaluate e_2 to a value
 - Change the location to hold the value
- Type-checks if:
 - e_1 is a well-typed left-expression
 - e_2 is a well-typed right-expression
 - They have the same type
- A surprisingly good model...

Formalizing left vs. right

$$\begin{aligned}
 e & ::= i \mid x \mid \&e \mid *e \mid e=e \mid \dots \\
 H & ::= \cdot \mid H, x \mapsto i \mid H, x \mapsto \&e
 \end{aligned}$$

$$\begin{array}{c}
 \overline{(H, i) \Downarrow_R (H, i)} \quad \overline{(H, x) \Downarrow_R (H, H(x))} \quad \frac{(H, e) \Downarrow_L (H', x)}{\overline{(H, \&e) \Downarrow_R (H', \&x)}} \quad \frac{(H, e) \Downarrow_R (H', \&x)}{\overline{(H, *e) \Downarrow_R (H', H'(x))}} \\
 \frac{(H, e_1) \Downarrow_L (H_1, x) \quad (H_1, e_2) \Downarrow_R (H_2, e_3)}{\overline{(H, e_1=e_2) \Downarrow_R (H_2[x \mapsto e_3], e_3)}} \\
 \overline{(H, x) \Downarrow_L (H, x)} \quad \frac{(H, e) \Downarrow_R (H', \&x)}{\overline{(H, *e) \Downarrow_L (H', x)}} \\
 \overline{\vdash_L x} \quad \overline{\vdash_L *e}
 \end{array}$$

For $\&e$ and $e=e'$, the type system requires $\vdash_L e$.

Polymorphic refs a la Cyclone

- Suppose `NULL` has type $\forall\alpha. (\alpha^*)$
- `e<>` means “do not instantiate”

```
void f(int *p) {  
    ( $\forall\alpha. (\alpha^*)$ ) x = NULL<>;  
    x<int> = p;  
    p = *(x<int*>);  
    *p = 0xBAD;  
}
```

- Note: `NULL` is never used

A closer look...

```
void f(int *p) {
  ( $\forall \alpha. (\alpha^*)$ ) x = NULL<>;
  x<int> = p;
  p = *(x<int*>);
  *p = 0xBAD;
}
```

- Locations x and p have contents' type change
- p changes because x does not hold $\forall \alpha. (\alpha^*)$
- x changes because x<int> has type int*
- But whoever said $\vdash_{\perp} e[\tau]$!?!

One more time, slowly

- If $e[\tau]$ is a valid left-expression, then assignment changes the type of a location's contents
 - Heap-Type Preservation is false
- “Homework”: If $e[\tau]$ is not a valid left-expression, the appropriate type system is sound
- Distinguishing left vs. right led us to a *very* simple solution that addresses the problem directly

The plan...

- C meets α
 - It's not about syntax
 - There's much more to Cyclone
- Polymorphic references
 - As seen from Cyclone (unusual view?)
 - Applied to ML (solved since early 90s)
- Mutable existentials
 - The original part
 - April 2002
- Breaking parametricity (Pierce)

But first, Cyclone got “lucky”

- Hindsight is 20/20; here’s what we really did
- Restrict type syntax to “ $\forall \alpha. (\tau \rightarrow \tau)$ ”
- As in C, variables cannot have function types (only pointers to function types)
- So only functions have function types
- Functions are immutable (not left-expressions)
- So $e [\tau]$ can type-check only if e is immutable

Sometimes fact is stranger than fiction

Now for ML

```
let x = ref None in
x := Some 3;
let (Some y):string = !x in
y ^ "crash"
```

- Conventional wisdom blames type inference for giving x the type $\forall \alpha. (\alpha \text{ option ref})$
- I blame the typing of references...

The references “ADT”

```
let x: (∀α...) = ref None in
x[int] := Some 3;
let (Some y):string = !(x[string]) in
y ^ "crash"
```

- The type-checker was told:

```
type α ref;
ref : ∀α. α → (α ref)
:=   : ∀α. (α ref) → α → unit
!    : ∀α. (α ref) → α
```

- Having masked left vs. right (for parsimony?), we cannot restrict where type instantiation is allowed

What if refs were special?

- It does not suffice to ban instantiation for the first argument of `:=`

```
let x: ( $\forall\alpha...$ ) = ref None in
let z = x[int] in
z := Some 3;
```
- Conjecture: It does suffice to allow instantiation of polymorphic refs only under ! (i.e., `!(e[τ])`)
- ML does not have implicit dereference like Cyclone right-expressions

But refs aren't special

- To prevent bad type instantiations, it suffices to ban polymorphic references
- So it suffices to ban all polymorphic expressions that aren't values (`ref` is a function)
- This “value restriction” is easy to implement and is orthogonal to inference

Disclaimer: This justification of the value restriction is revisionism, but I like it.

The plan...

- C meets α
 - It's not about syntax
 - There's much more to Cyclone
- Polymorphic references
 - As seen from Cyclone (unusual view?)
 - Applied to ML (solved since early 90s)
- **Mutable existentials**
 - The original part
 - April 2002
- Breaking parametricity (Pierce)

C Meets \exists

- Existential types in a safe low-level language
 - why (again)
 - features (mutation, aliasing)
- The problem
- The solutions
- Some non-problems
- Related work

Low-level languages want \exists

- Major goal: expose data representation (no hidden fields, tags, environments, ...)
- Languages need data-hiding constructs
- Don't provide closures/objects; give programmers a powerful type system

```
struct T { <`a>.  
    int (*f) (int, `a);  
    `a env;  
};
```

C “call-backs” use void; we use \exists*

Normal \exists feature: Construction

```
struct T { <`a>.  
    int (*f) (int, `a) ;  
    `a env ;  
};
```

```
int add (int a, int b) {return a+b; }  
int addp (int a, char* b) {return a+*b;}  
struct T x1 = T (add, 37) ;  
struct T x2 = T (addp, "a") ;
```

- Compile-time: check for appropriate [witness type](#)
- Type is just `struct T`
- Run-time: create / initialize (no witness type)

Normal \exists feature: Destruction

```
struct T { <`a>.  
    int (*f) (int, `a);  
    `a env;  
};
```

Destruction via *pattern matching*:

```
void apply(struct T x) {  
    let T{<`b> .f=fn, .env=ev} = x;  
    // ev : `b, fn : int(*f) (int, `b)  
    fn(42, ev);  
}
```

Clients use the data without knowing the type

Low-level feature: Mutation

- Mutation, changing witness type

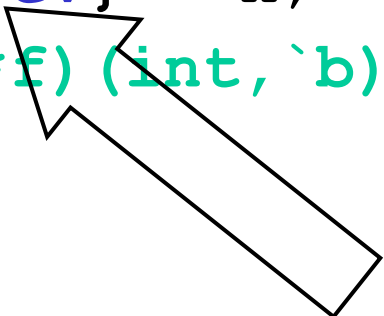
```
struct T fn1 = f();  
struct T fn2 = g();  
fn1 = fn2; // record-copy
```

- Orthogonality encourages this feature
- Useful for registering new call-backs without allocating new memory
- Now memory is not type-invariant!

Low-level feature: Address-of field

- Let client update fields of an existential package
 - access only through pattern-matching
 - variable pattern *copies* fields
- A *reference pattern* binds to the field's address:

```
void apply2(struct T x) {  
    let T{<`b> .f=fn, .env=*ev} = x;  
    // ev : `b*, fn : int(*f) (int, `b)  
    fn(42, *ev);  
}
```



C uses &x.env; we use a reference pattern

More on reference patterns

- Orthogonality: already allowed in Cyclone's other patterns (e.g., tagged-union fields)
- Can be useful for existential types:

```
struct Pr {<`a> `a fst; `a snd; };
```

```
void swap<`a>(`a* x, `a* y);
```

```
void swapPr(struct Pr pr) {  
    let Pr{<`b> .fst=*a, .snd=*b} = pr;  
    swap(a, b);  
}
```

Summary of features

- **struct** definition can bind existential type variables
- construction, destruction traditional
- mutation via **struct** assignment
- reference patterns for aliasing

A nice adaptation to a “safe C” setting?

Explaining the problem

- Violation of type safety
- Two solutions (restrictions)
- Some non-problems

Oops!

```
struct T {<`a> void (*f) (int, `a); `a env;};

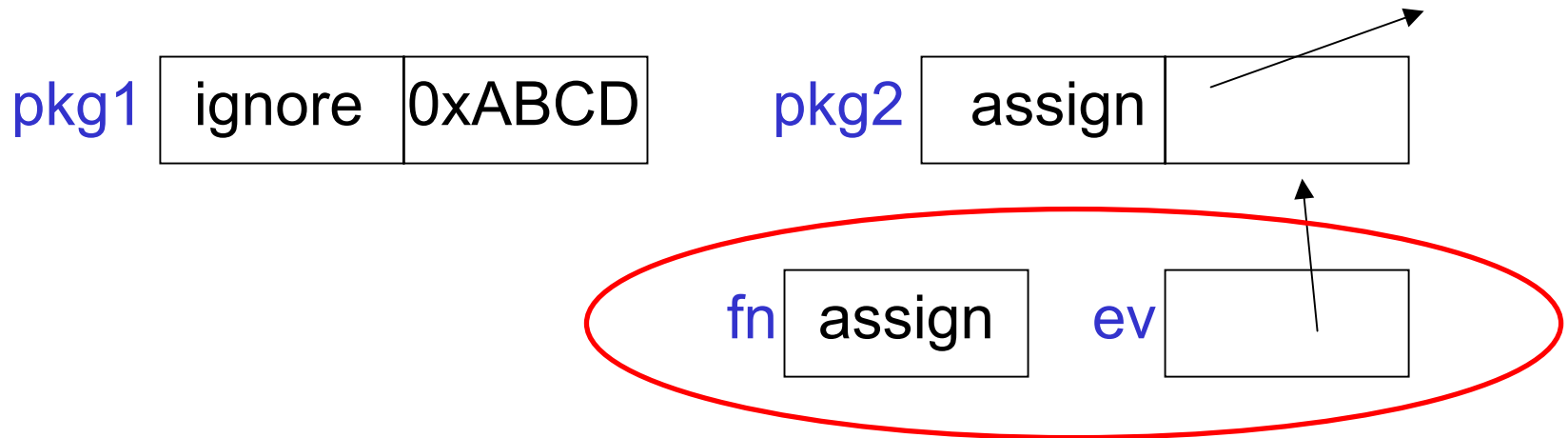
void ignore(int x, int y) {}
void assign(int x, int* p) { *p = x; }

void g(int* ptr) {
    struct T pkg1 = T(ignore, 0xBAD); //α=int
    struct T pkg2 = T(assign, ptr); //α=int*
    let T{<`b> .f=fn, .env=*ev} = pkg2; //alias
    pkg2 = pkg1; //mutation
    fn(37, *ev); //write 37 to 0xBAD
}
```

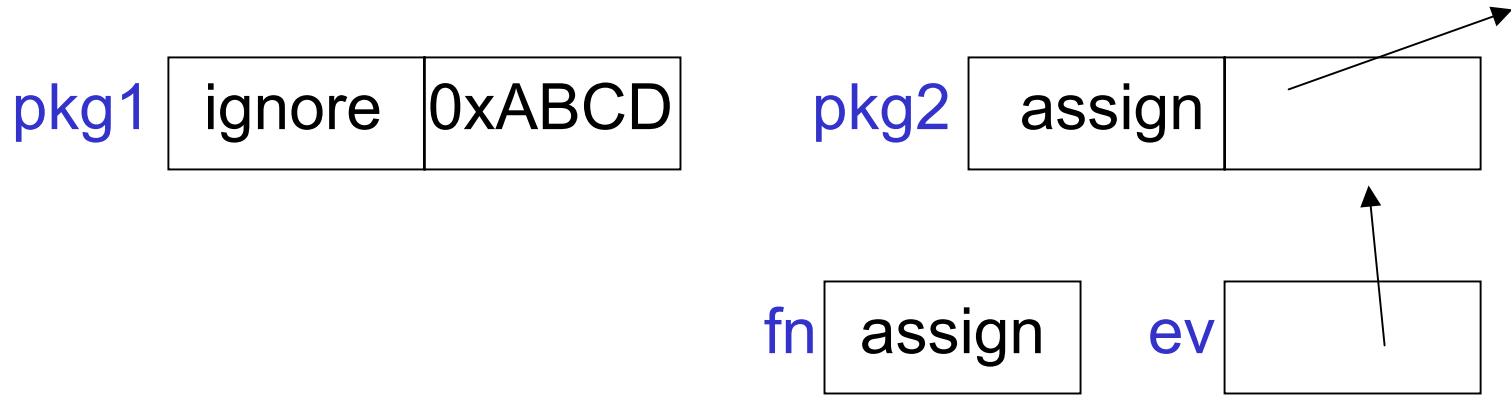
With pictures...



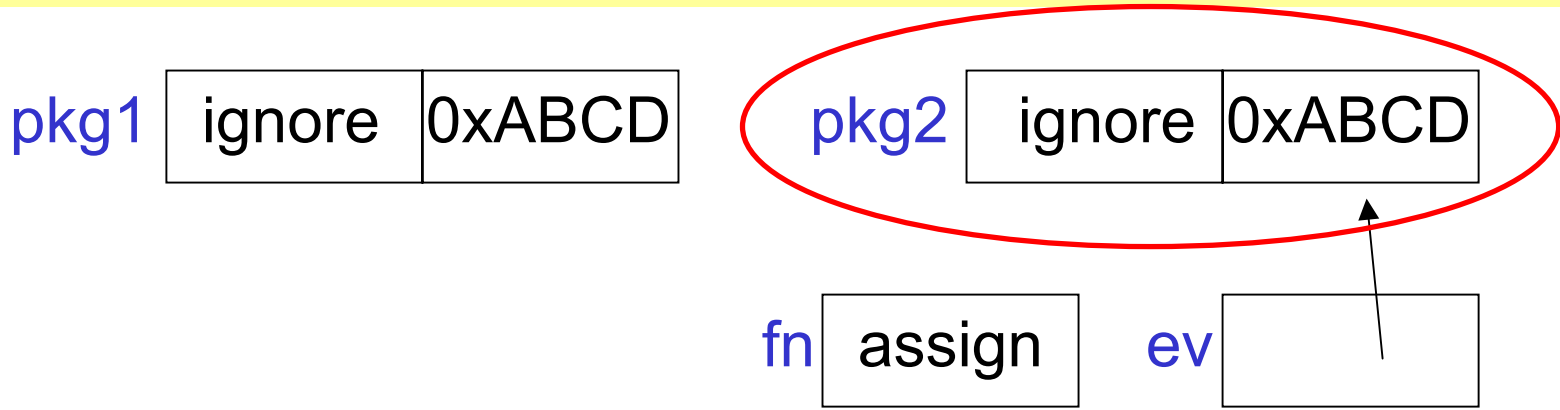
```
let T{<`b> .f=fn, .env=*ev} = pkg2; //alias
```



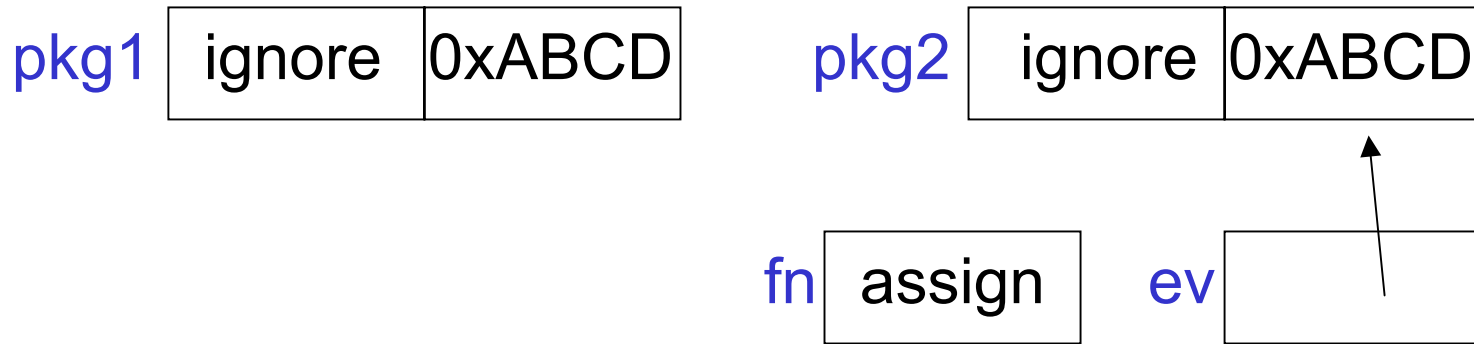
With pictures...



`pkg2 = pkg1; //mutation`



With pictures...



```
fn(37, *ev); //write 37 to 0xABCD
```

call assign with 0xABCD for p:

```
void assign(int x, int* p) {*p = x;}
```

What happened?

```
let T{<`b> .f=fn, .env=*ev} = pkg2; //alias
pkg2 = pkg1; //mutation
fn(37, *ev); //write 37 to 0xABCD
```

1. Type ``b` establishes a compile-time equality relating types of `fn` (`void(*f) (int, `b)`) and `ev` (``b*`)
2. Mutation makes this equality false
3. Safety of call needs the equality

We must rule out this program...

Two solutions

- Solution #1:

Reference patterns do not match against fields of existential packages

Note: Other reference patterns still allowed

⇒ cannot create the type equality

- Solution #2:

Type of assignment cannot be an existential type (or have a field of existential type)

Note: pointers to existentials are no problem

⇒ restores memory type-invariance

Independent and easy

- Either solution is easy to implement
- They are *independent*: A language can have two styles of existential types, one for each restriction
- Cyclone takes solution #1 (no reference patterns for existential fields), making it a safe language without type-invariance of memory!

Are the solutions sufficient (correct)?

- I defined a small formal language and proved type safety
- Highlights:
 - Left vs. right distinction
 - Both solutions
 - C-style memory (flattened pairs)
 - Memory invariant includes novel “if a reference pattern is for a location, then that location never changes type”

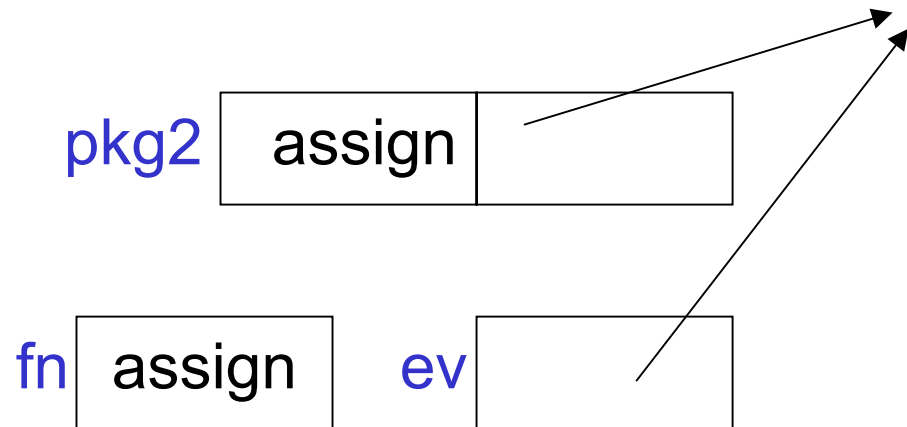
Nonproblem: Pointers to witnesses

```
struct T2 {<`a>  
  void (*f) (int, `a);  
  `a* env;  
};
```

...

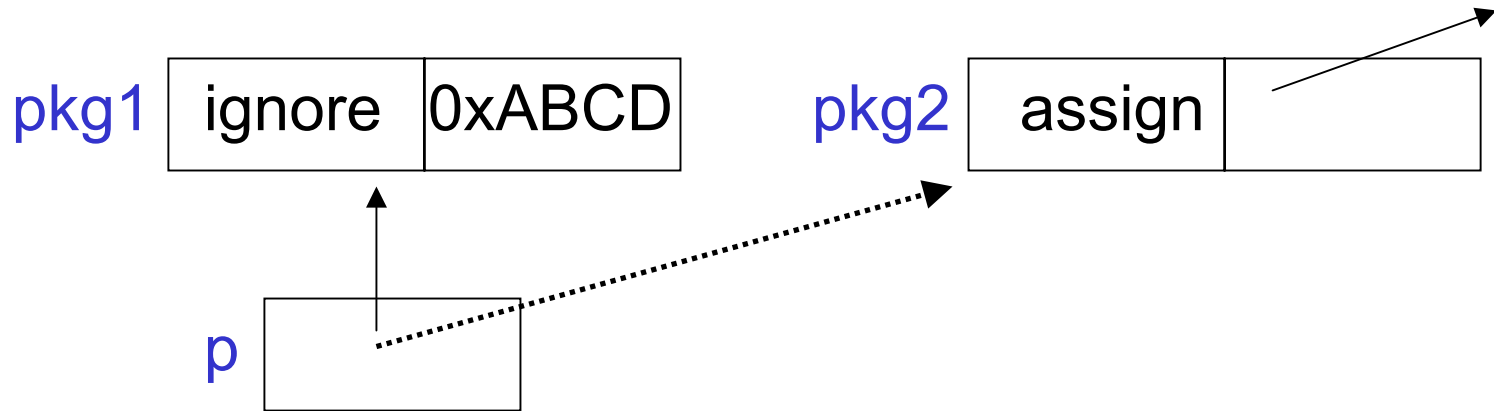
```
let T2{<`b> .f=fn, .env=ev} = pkg2;  
pkg2 = pkg1;
```

...



Nonproblem: Pointers to packages

```
struct T * p = &pkg1;  
p = &pkg2;
```



Aliases are fine.

Aliases of `pkg1` at the “unpacked type” are not.

Problem appears new

- Existential types:
 - seminal use [Mitchell/Plotkin 1988]
 - closure/object encodings [Bruce et al, Minamide et al, ...]
 - first-class types in Haskell [Läufer]

None incorporate mutation
- Safe low-level languages with \exists
 - Typed Assembly Language [Morrisett et al]
 - Xanadu [Xi], uses \exists over ints

None have reference patterns or similar
- Linear types, e.g. Vault [DeLine, Fähndrich]

No aliases, destruction destroys the package

Duals?

- Two problems with α , mutation, and aliasing
- One used \forall , one used \exists
- So are they the same problem?

```
( $\forall \alpha. (\alpha^*)$ ) x = NULL<>;  
x<int> = p;  
p = *(x<int*>);  
*p = 0xBAD;
```

```
struct T pkg1=T(f1,0xBAD);  
struct T pkg2=T(f2,ptr);  
let T{<b>.f=fn,  
      .env=*ev} =pkg2;  
pkg2 = pkg1;  
fn(37, *ev);
```

- Conjecture: Similar, but not true duals
- Fact: Thinking dually hasn't helped me

The plan...

- C meets α
 - It's not about syntax
 - There's much more to Cyclone
- Polymorphic references
 - As seen from Cyclone (unusual view?)
 - Applied to ML (solved since early 90s)
- Mutable existentials
 - The original part
 - April 2002
- Breaking parametricity [Pierce]

Parametricity is cool

- In the polymorphic lambda calculus, we get results so cool they have slogans
 - “related arguments produce related results”
 - “theorems for free”
- Do these results extend to Cyclone or ML?
 - Is ``a f `a` ; the identity function?
 - Is `int f `a` ; a constant function?
 - Given `int g `a, int`, does `g (0, 3) == g (`x, 3)` ?

Some easy counterexamples

- Is `int f(`a);` a constant function?
- No:

```
int f(`a x) {while(true) ; }
int f(`a x) {throw new Failure("!");}
int f(`a x) {return g++;/*global g*/}
int f(`a x) {return getc(stdin);}
```

- ML has divergence, exceptions, free refs, and input.
- Okay, so if `int f(`a);` is a closed, terminating, function that doesn't raise exceptions, is it a constant function? *With enough caveats, yes, the result does not depend on x.*

Another example

- Given closed `int g(`a* x, int* y)`, can the result of `g(e1, e2)` depend on `e1`?
- Hint: `void f(int *p) { g<int>(p, p); }`

Aliases break parametricity

```
int g(`a* x, int* y) {  
    *y = 0;  
    `a z = *x;  
    *y = 1;  
    *x = z;  
    return *y==0;  
}
```

- Returns 1 iff $x==y$, so first argument does matter
- Sufficient to code up ad hoc polymorphism (given the right aliases, g can determine ``a`)
- Does *not* compromise safety
- Works in ML
- Works for any type with two distinguishable values

More observations

```
int g(`a* x, int* y) {  
    *y = 0;  
    `a z = *x;  
    *y = 1;  
    *x = z;  
    return *y==0;  
}
```

- Relies on atomicity and semantics of assignment
- Can prevent by strengthening type system so callers must specify the type at which they pass references to `g`

Conclusions

If you see an α near an assignment statement:

- Do your homework
- Remain vigilant
- Do not expect parametricity
- Do not be afraid of C-level thinking

For related work, see Section 2.7 of my forthcoming dissertation (draft available)

[The presentation ends here. Some auxiliary slides follow.]

Less obvious occurrences

```
struct T { <`i::I>
  tag_t<`i> tag;
  union U {
    `i==1: int* p;
    `i==2: int  x;
  } u;
};
```

- Tagged unions (ML datatypes) *are* existentials
- If they're mutable and you can alias their fields, the problem is identical

Cyclone in brief

***A safe, convenient, and modern language
at the C level of abstraction***

- **Safe:** memory safety, abstract types, no core dumps
- **C-level:** user-controlled data representation and resource management, easy interoperability, “manifest cost”
- **Convenient:** may need more type annotations, but work hard to avoid it
- **Modern:** add features to capture common idioms

“New code for legacy or inherently low-level systems”