Quantified Types in a Safe C-Level Language

CMU POP Seminar

26 January 2005

Dan Grossman

University of Washington

Context: Why Cyclone?

A type-safe language at the C-level of abstraction

- Type-safe: Memory safety, abstract types, …
- C-level: explicit pointers, data representation, memory management. Semi-portable.
- Niche: Robust/extensible systems code
  - Looks like, acts like, and interfaces easily with C
  - Used in several research projects
  - Doesn't "fix" non-safety issues (syntax, switch, …)
- Modern: patterns, tuples, exceptions, …

www.research.att.com/projects/cyclone

Context: Why quantified types?

- The usual reasons:
  - Code reuse, container types, abstraction, …
  - Phantom types, iterators, …
  - ~~Parametricity~~
- *Because* low-level
  - Implement closures with existentials
  - Pass environment fields to functions
- For other kinds of invariants
  - Memory regions, array-lengths, locks
  - Same theory and more important in practice
  - But focus on types today

Context: Why novel?

- Left vs. right expressions and the **&** operator
- Aggregate assignment (record copy)
- First-class existential types in an imperative language
- Types of unknown size

And any new combination of effects, aliasing, and polymorphism invites trouble…

Getting burned… decent company

To: sml-list@cs.cmu.edu
From: Harper and Lillibridge
Sent: 08 Jul 91
Subject: Subject: ML with callcc is unsound

The Standard ML of New Jersey implementation of callcc is not type safe, as the following counterexample illustrates:… Making callcc weakly polymorphic … rules out the counterexample

Getting burned… decent company

From: Alan Jeffrey
Sent: 17 Dec 2001
To: Types List
Subject: Generic Java type inference is unsound

The core of the type checking system was shown to be safe… but the type inference system for generic method calls was not subjected to formal proof. In fact, it is unsound … This problem has been verified by the JSR14 committee, who are working on a revised langauge specification…

Getting burned… decent company

From: Xavier Leroy
Sent: 30 Jul 2002
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.

…

Getting burned…I'm in the club

```
From: Dan Grossman
Sent: Thursday 02 Aug 2001
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
…
```

The plan from here

- Brief tour of Cyclone polymorphism
- C-level polymorphic references
  - Formal model with “left” and “right”
  - Comparison with actual languages
- C-level existential types
  - Description of “new” soundness issue
  - Some non-problems
- C-level type sizes
  - Not a soundness issue

"Change **void*** to alpha"

```
struct L {
  void* hd;
  struct L* tl;
};
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>* tl;
};
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:

  $L = \lambda\alpha.\ \{\ \alpha\ \ hd;\ \ \ (L\ \alpha)\ *\ \ tl;\ \}$

- The functions are polymorphic:

  $map : \forall\alpha, \beta.\ (\alpha\rightarrow\beta, L\ \alpha) \rightarrow (L\ \beta)$

- 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;
};
```

- creation requires a "consistent witness"

- type is just **`struct T`**

- use requires an explicit "unpack" or "open":

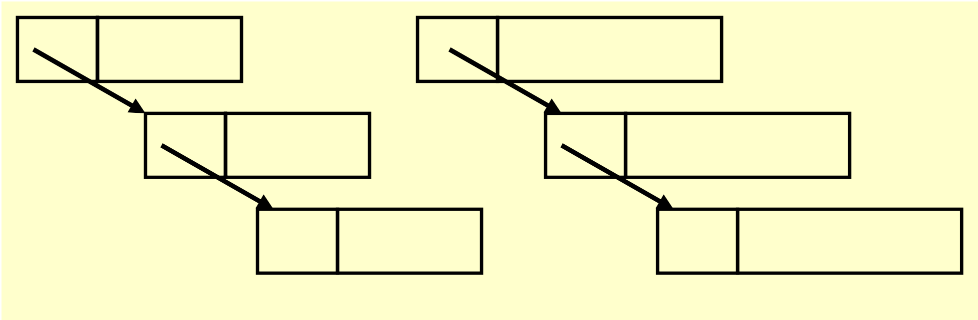
```
int apply(struct T pkg, int arg) {
  let T{<`b> .f=fp, .env=ev} = pkg;
  return fp(arg,ev);
}
```

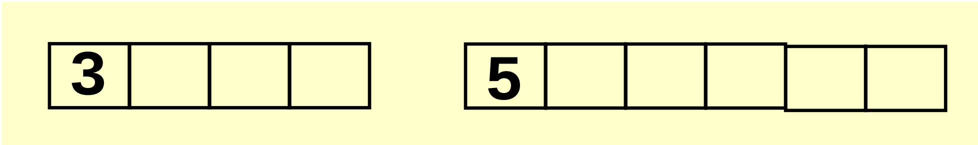
Sizes

Types have known or unknown size (a kind distinction)

- As in C, unknown-size types can't be used for fields, variables, etc.: must use pointers to them
- Unlike C, we allow last-field-unknown-size:

```
struct T1 {
 struct T1* tl;
 char data[1];
};
struct T2 {
 int len;
 int arr[1];
};
```





Sizes

Types have known or unknown size (a kind distinction)

- As in C, unknown-size types can't be used for fields, variables, etc.: must use pointers to them
- Unlike C, we allow last-field-unknown-size:

```
struct T1 {
 struct T1* tl;
 char data[1];
};
struct T2 {
 int len;
 int arr[1];
};
```

```
struct T1<`a::A> {
 struct T1<`a>* tl;
 `a data;
};
struct T2<`i::I> {
 tag_t<`i> len;
 int arr[valueof(`i)];
};
```

The plan from here

- Brief tour of Cyclone polymorphism
- C-level polymorphic references
  - Formal model with “left” and “right”
  - Comparison with actual languages
- C-level existential types
  - Description of “new” soundness issue
  - Some non-problems
- C-level type sizes
  - Not a soundness issue

Mutation

- **`e1=e2`** means:
  - Left-evaluate e1 to a location
  - Right-evaluate e2 to a value
  - Change the location to hold the value
- Locations are "left values": **`x.f1.f2…fn`**
- Values are "right values", include **`&x.f1.f2…fn`**

  (a pointer to a location)
- Having interdependent left/right evaluation is *not a problem*

Left vs. Right Syntax

$$
\begin{array}{rcl}
\tau & ::= & \text{int} \mid \tau \times \tau \mid \tau \to \tau \mid \tau * \\
e & ::= & x \mid i \mid e{=}e \mid \&e \mid *e \mid (e, e) \mid e.i \mid \lambda x : \tau.\ e \mid e(e) \\
v & ::= & i \mid \&\ell \mid (v, v) \mid \lambda x : \tau.\ e \\
\ell & ::= & x \mid \ell.i \\
H & ::= & \cdot \mid H, x \mapsto v \\
P & ::= & H; e
\end{array}
$$

Everything is mutable heap-allocated (ignore memory management)

In C, functions are top-level and closed, but it doesn't matter

Allow aggregate assignment (can assign to $x.i_1 \ldots i_n$ even if $x.i_1 \ldots i_n$ has a pair type)

Small-Step Semantics

- Two forms of evaluation context
- Auxiliary judgment for aggregate assignment

$$R ::= [\cdot]_r \mid L{=}e \mid \ell{=}R \mid \&L \mid *R \mid (R,e) \mid (v,R) \mid R.i \mid R(e) \mid v(R)$$

$$L ::= [\cdot]_l \mid L.i \mid *R$$

$$\frac{H; e \xrightarrow{l} H'; e'}{H; R[e]_l \rightarrow H'; R[e']_l} \qquad \frac{H; e \xrightarrow{r} H'; e'}{H; R[e]_r \rightarrow H'; R[e']_r}$$

$$H; *\&\ell \xrightarrow{l} H; \ell$$

$$\begin{aligned}
H; *\&\ell &\xrightarrow{r} H; \ell \\
H; x &\xrightarrow{r} H; H(x) \\
H; (v_1, v_2).i &\xrightarrow{r} H; v_i \\
H; \ell = v &\xrightarrow{r} \ldots \\
H; (\lambda x : \tau.\ e)(v) &\xrightarrow{r} H, x \mapsto v; e
\end{aligned}$$

Typing

Completely normal ($\Gamma \vdash e : \tau$) except:

Left-expressions (e in $e{=}e'$ and $\&e$) must satisfy syntactic restrictions

$$\vdash_{\mathsf{lval}} x \qquad \vdash_{\mathsf{lval}} *e \qquad \frac{\vdash_{\mathsf{lval}} e}{\vdash_{\mathsf{lval}} e.i}$$

(With an effect system, it's more convenient to have interdependent typing judgments just as we did for evaluation.)

Polymorphism

Adding universal types is completely standard:

$$\tau \; ::= \; \ldots \mid \alpha \mid \forall\alpha.\tau$$
$$e \; ::= \; \ldots \mid \Lambda\alpha.e \mid e[\tau]$$

Polymorphic abstractions are values.

This is what the polymorphic reference problem looks like (with sugar):

```
let id : ∀α.α → α = Λα.λ x:α. x;
let i  : int       = 0;
let p  : int*      = &i;
id [int] = λ x:int. x + 17;
p = (id [int*]) (p)    (* p mutated to "p + 17" *)
```

What went wrong?

The Bottom Line

The key to soundness: $\not\vdash_{\mathsf{lval}} e[\tau]$

- Really, that's it.
- More justification: It is sound for $(\forall\alpha.\tau_1) \leq (\forall\alpha.\tau_1)[\tau_2]$, but not sound to make subsumption a left expression.

Non-problem: Pointers to "top"

If p has type $(\forall\alpha.\alpha)*$, then we can only update $*p$ to (still) hold top.

Semi-problem: Polymorphic pointers

If q has type $\forall\alpha.(\alpha*)$, then $*(q[\tau])$ is allowed.

- But q could never hold a pointer into the heap.
- If q holds a value for which $*(q[\tau])$ is stuck (e.g., `NULL`), then that's life (and we're memory safe).

What we learned

- Left vs. right formalizes just fine
- Instantiation as left-expression is unsound
  - And banning it suffices
- Difference between "$\forall\alpha.\ (\alpha\ *)$" and "$(\forall\alpha.\ \alpha)\ *$"
  - Clear in TAL too

- Now:

  Does this shed any light on Cyclone or ML?

Cyclone got "lucky"

Hindsight is 20/20; here's what really happens:

- Restrict type syntax to "**$\forall\alpha 1\ \forall\alpha 2 \ldots \forall\alpha n(\tau \rightarrow \tau)$**"
- As in C, variables cannot hold functions
  - Function pointers hold *pointers to* functions
- As in C, functions are immutable (not left-expressions)

So: No (mutable) location ever holds a polymorphic value
- Instantiation-as-left-expression a non-issue

Sometimes fact is stranger than fiction

The plan from here

- Brief tour of Cyclone polymorphism
- C-level polymorphic references
  - Formal model with “left” and “right”
  - Comparison with actual languages
- C-level existential types
  - Description of “new” soundness issue
  - Some non-problems
- C-level type sizes
  - Not a soundness issue

C Meets ∃

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

Low-level languages want ∃

- Major goal: expose data representation (no hidden fields, tags, environments, ...)
- Languages need data-hiding constructs
- Don’t provide closures/objects

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

C “call-backs” use **`void*`***; we use ∃*

Normal ∃ feature: Introduction

```
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 ∃ feature: Elimination

```
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 and abstraction encourage this feature
- Useful for registering new call-backs without allocating new memory
- Now memory words are 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!

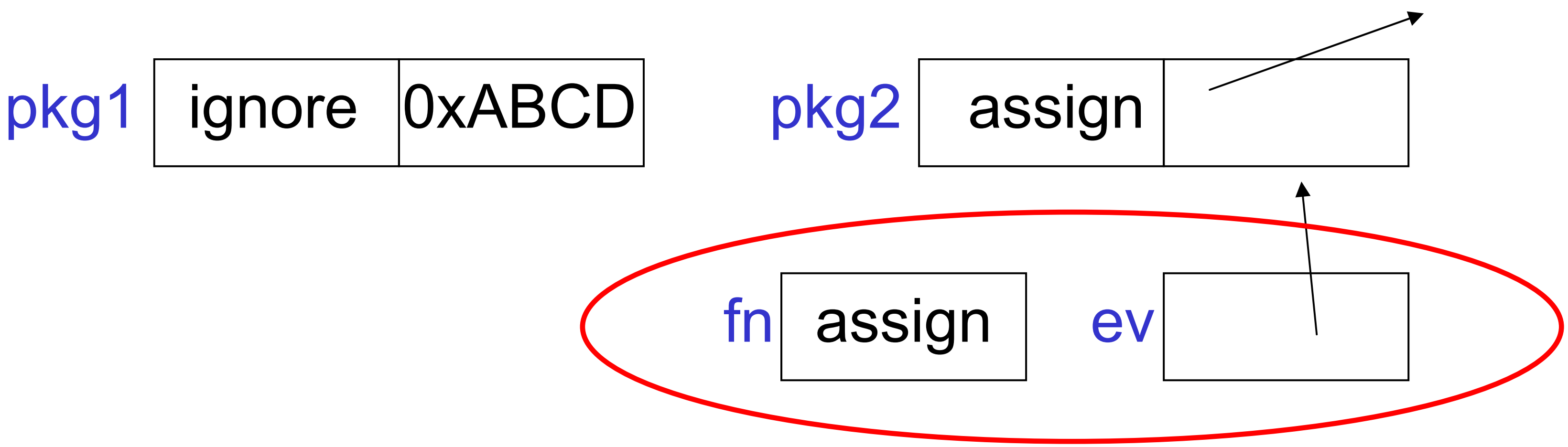
```
struct T {<`a> void (*f)(int,`a); `a env;};

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
}
```

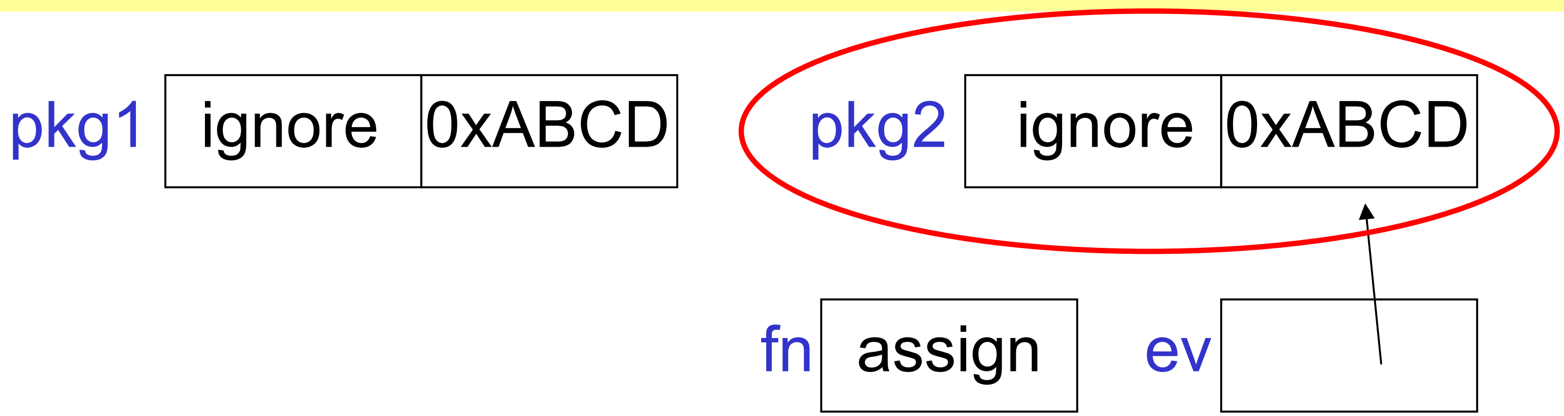
With pictures…


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

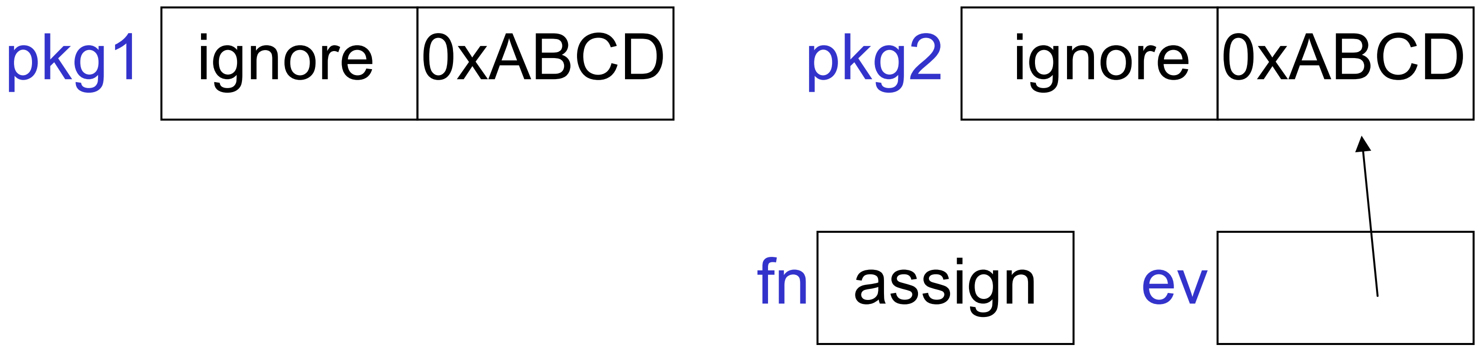


With pictures…


```
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)?

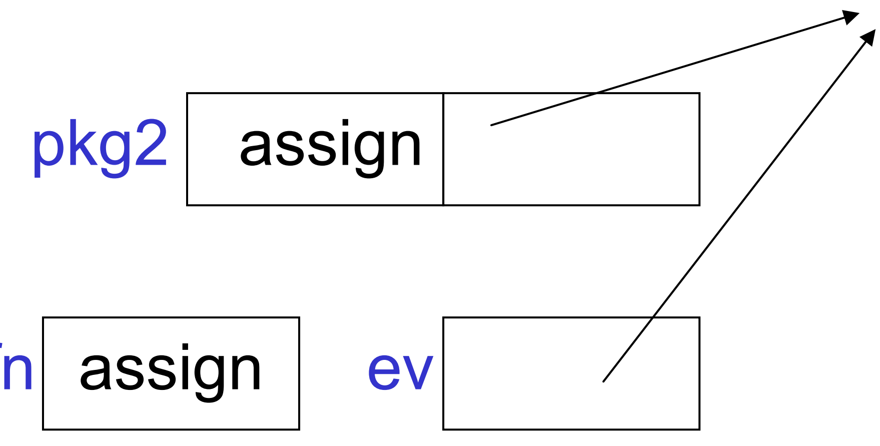
- Small formal language proves type safety

- Highlights:
  - Left vs. right distinction
  - Both solutions
  - Memory invariant (necessarily) includes:

    "if a reference pattern is used for a location, then that location never changes type"

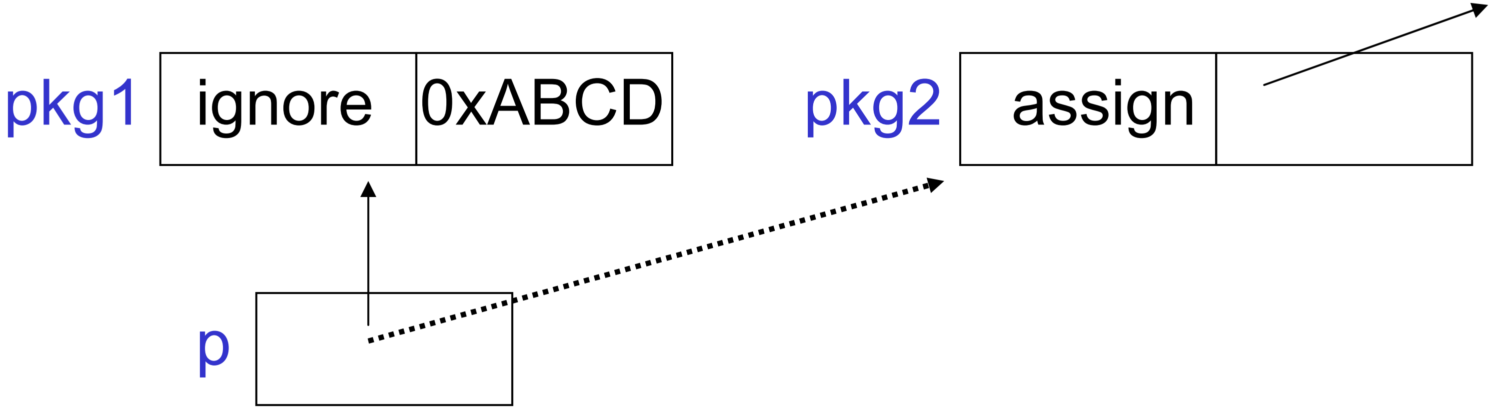
Nonproblem: Pointers to witnesses

```
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 1985]
  - closure/object encodings [Bruce et al, Minimade et al, …]
  - first-class types in Haskell [Läufer]

  None incorporate mutation

- Safe low-level languages with ∃
  - Typed Assembly Language [Morrisett et al]
  - Xanadu [Xi], uses ∃ 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 ∀, one used ∃
  - So are they the same problem?
- Conjecture: Similar, but not true duals
- Fact: Thinking dually hasn't helped me

The plan from here

- Brief tour of Cyclone polymorphism
- C-level polymorphic references
  - Formal model with “left” and “right”
  - Comparison with actual languages
- C-level existential types
  - Description of “new” soundness issue
  - Some non-problems
- C-level type sizes
  - Not a soundness issue

Size in C

C has abstract types (not just **void***):

```
struct T1;
struct T2 {
  int len;
  int arr[*];//C99, much better than [1]
};
```

And rules on their use that make sense at the C-level:*

- E.g., variables, fields, and assignment targets cannot have type **struct T1**.

* Key corollary: C hackers don't mind the restrictions

Size in Cyclone

- Kind distinction among:
  1. *B* "pointer size" <
  2. *M* "known size" <
  3. *A* "unknown size"

(Really not much different than TAL)

- Killer app: Cyclone interface to C functions

```
void mem_copy<`a>(`a*,`a*, sizeof_t<`a>);
```

Should we be worried about soundness?

Why is size an issue in C?

"Only" reason C restricts types of unknown size:

Efficient and transparent implementation:

- No run-time size passing
- Statically known field and stack offsets

This is important for translation, but has nothing to do with soundness

Indeed, our formal model is "too high level" to motivate the kind distinction

Formal (Non)-Example

Illegal-but-useful code:

$\texttt{let memCopy : } \forall\boldsymbol{\alpha}\texttt{:A. } \boldsymbol{\lambda}\texttt{ x:}\boldsymbol{\alpha}\texttt{. x}$

In formalism, works fine:

$\texttt{let y : } \mathbf{int} \texttt{ = memCopy [}\mathbf{int}\texttt{] 10}$

$\texttt{let z : } \mathbf{int} \times \mathbf{int} \texttt{ = memCopy [}\mathbf{int} \times \mathbf{int}\texttt{] (11,81)}$

First call allocates an int, second a pair:

$$H; (\lambda x : \tau.\ e)(v) \quad \xrightarrow{\mathsf{r}} \quad H, x \mapsto v; e$$

Also works fine with "stack allocation" (or de Bruijn indices or substition or ...)

What we hid is that function arguments of unknown size cannot easily be passed.

The plan from here

- Brief tour of Cyclone polymorphism
- C-level polymorphic references
  - Formal model with “left” and “right”
  - Comparison with actual languages
- C-level existential types
  - Description of “new” soundness issue
  - Some non-problems
- C-level type sizes
  - Not a soundness issue
- Conclusions

Polymorphism everywhere!

- Cyclone uses type variables for "everything" (regions, locks, array-lengths, union-tags, …)
- So type variables are very common
  - Any function taking a pointer
  - Bounds-checked arrays
  - …
- With an effects system, left vs. right extends nicely
  - &x does not "access" x
- "Dan's unsoundness" has come up > *n* times
  - Have (and use) datatypes with the "other" solution

Conclusions

If you see an α near an assignment statement:

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

- Surprisingly:
  - This work has really guided the design and implementation of Cyclone
  - The design space of imperative, polymorphic languages is not fully explored

[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

ML?

```
val x :(∀α...) = ref NONE
val _ = x[int]  := SOME 3
val (SOME y):string = !(x[string])
val _ = y ^ "crash"
```

- Conventional wisdom blames type inference for giving x the type "∀**α.(α option ref)**"
- It *is* a bad idea for a type (cf. ∀α. (α *****))
- And "**(∀α. α option)ref**" is not an ML type

Revisionist history?

- The type-checker is told ref has an ML signature

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

- Value restriction makes ref “not special” by banning generalization on *all* function applications

- A simpler type system, but exposing mutability to the type/signature system is certainly practical

What now?

- Cyclone
  - A real module language (CLAMP)
  - Availability
- Compiler construction
  - Error messages via search
- Concurrency: **`atomic { s }`**
  - Slick Caml uniprocessor implementation
  - OO implementations
  - Simpler multiprocessor implementations

Atomic (coming ICFP submission)

Atomic: (Behave as if) no interleaved execution

An easier concurrency primitive:
- Compositional (nests trivially)
- Post-hoc synchronization
- Deadlock-free
- Common intent (see Qadeer, et al)

Clever implementation (own scheduler, code-gen):
- Non-atomic code runs no slower
- Logging and rollback for atomic-writes
- Fair scheduling