Existential Types for Imperative Languages

Dan Grossman
Cornell University

Eleventh European Symposium on Programming
April 2002

Designing safe languages

To design a strong-typed language:

1. Draw on acquired knowledge of well-behaved features

2. Model the parts you’re uncomfortable with (in practice, a simplification)

3. Hope/argue that the model captured everything interesting, so the language is type-safe

But…

- Sometimes you are wrong due to a new combination of features
- You fix it
- You worry enough to model the fix
- You add to acquired knowledge
- Today’s combination: existential types, aliasing, and mutation

How the story goes…

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

Existential types

- Existential types ($\exists\, \alpha\, .\, \tau$) hide types' identities while establishing equalities, e.g.,

$$\exists\, \alpha.\ \{\ \text{zero}: \alpha \quad \text{succ}: \alpha \to \alpha \quad \text{cmp}: \alpha \to \alpha \to \text{bool}\ \}$$

- That is, they describe abstract data types
- *The standard tool* for modeling data-hiding constructs (closures, objects)

Low-level languages want ∃

- Cyclone (this work's context) is a safe language at the C level of abstraction
- Major goal: expose data representation (no hidden fields, tags, environments, ...)
- Don't provide closures/objects; give programmers a powerful type system

```
struct IntIntFn { ∃ α.
  int (*f)(int, α);
  α env;
};
```

C "call-backs" use **`void*`***; we use ∃*

Normal ∃ feature: Construction

```
struct IntIntFn { ∃ α.
   int (*f)(int, α);
   α env;
};
```

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

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

Normal ∃ feature: Destruction

```
struct IntIntFn { ∃ α.
  int (*f)(int, α);
  α env;
};
```

Destruction via *pattern matching*:

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

Clients use the data without knowing the type

Low-level feature: Mutation

- Mutation, changing witness type

```
struct IntIntFn fn1 = f();
struct IntIntFn 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 IntIntFn x) {
   let IntIntFn{<β> .f=fn, .env=*ev} = x;
   // ev : β*,  fn : int(*f)(int,β)
   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 {∃ α. α fst; α snd; };

∀α. void swap(α* x, α* y);

void swapPr(struct Pr pr) {
  let Pr{<β> .fst=*a, .env=*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 of advanced type-systems to a "safe C" setting?

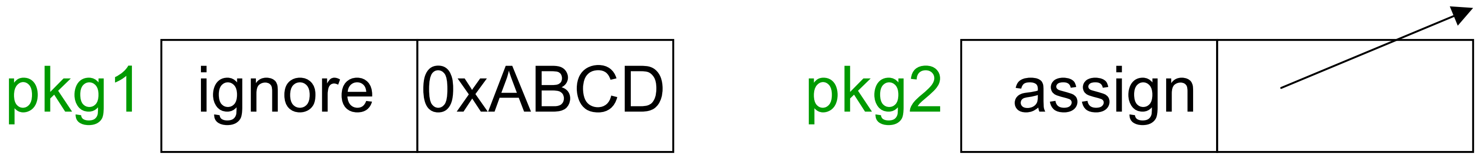
Explaining the problem

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

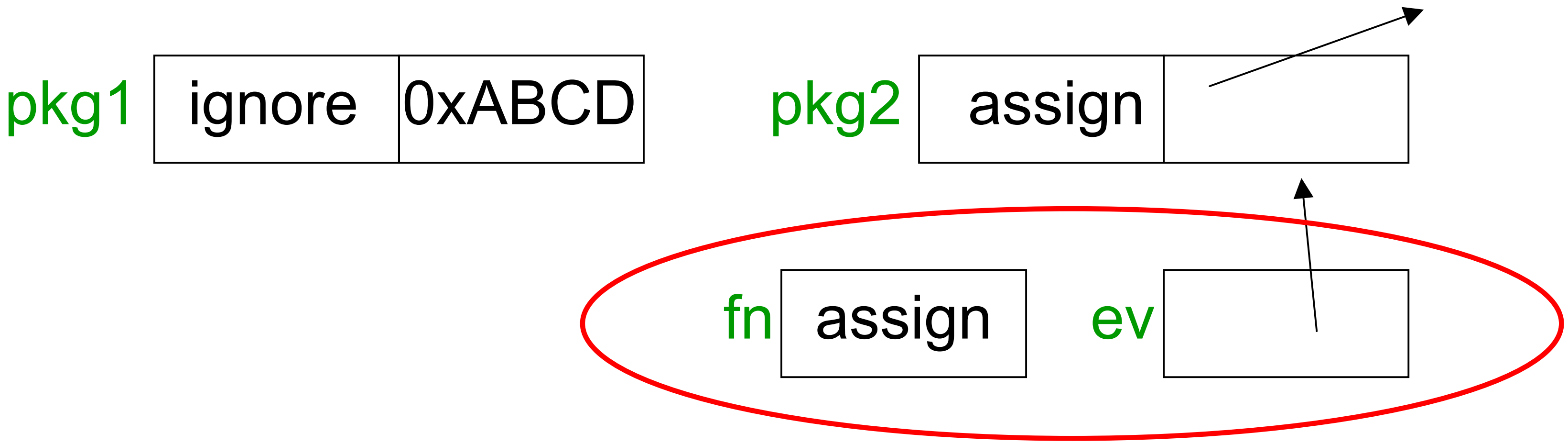
Oops!

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

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

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

With pictures…

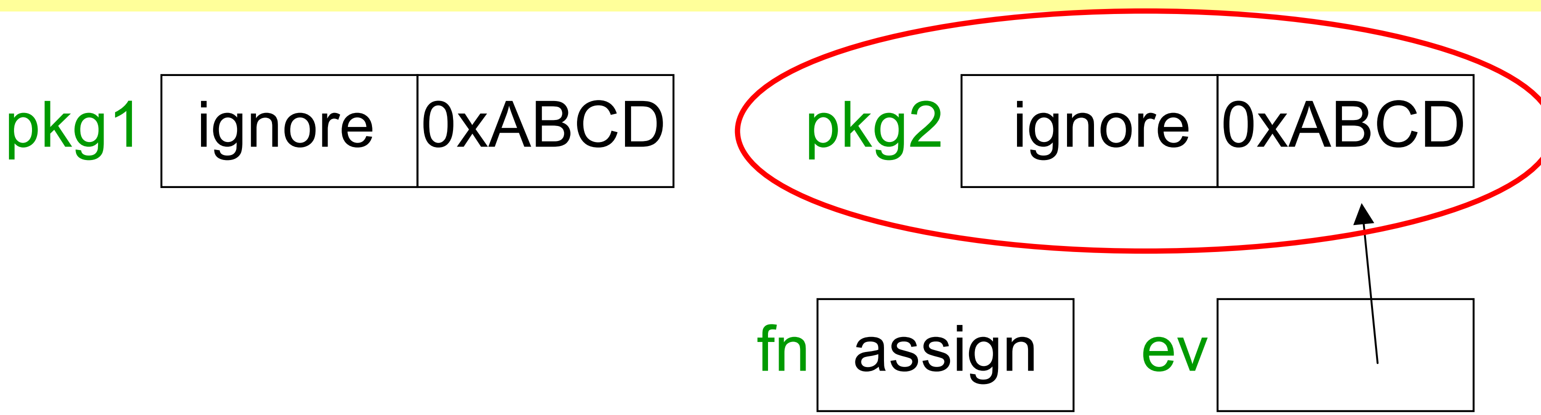



```
let T{<β> .f=fn, .env=*ev} = pkg2; //alias
```

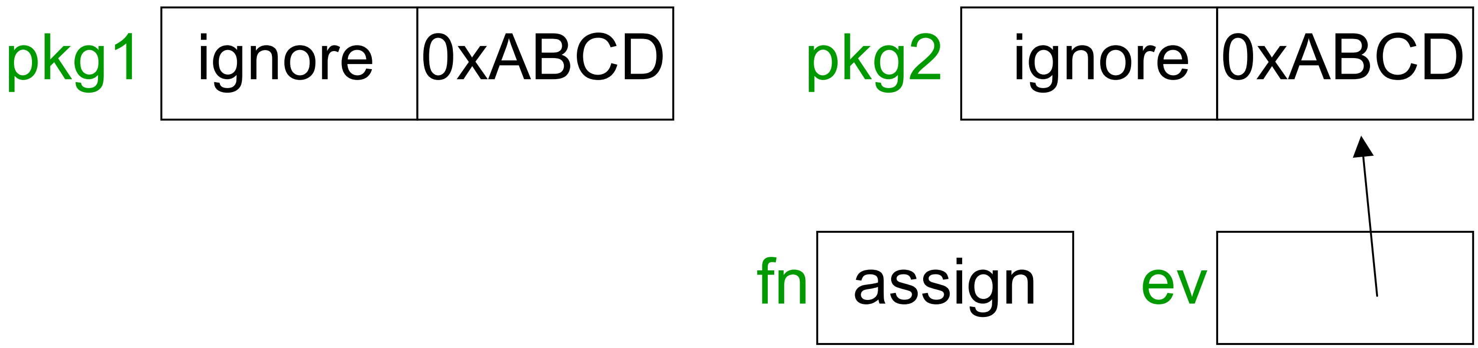


With pictures…


```
pkg2 = pkg1; //mutation
```



With pictures…




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

call **`assign`** *with* **`0xABCD`** *for* **`p`***, the pointer:*

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

What happened?

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

1. β establishes a compile-time equality relating types of `fn` (`void(*f)(int,β)`) and `ev` (`β*`)
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

  $\Rightarrow$ 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

  $\Rightarrow$ restores memory type-invariance

Independent and easy

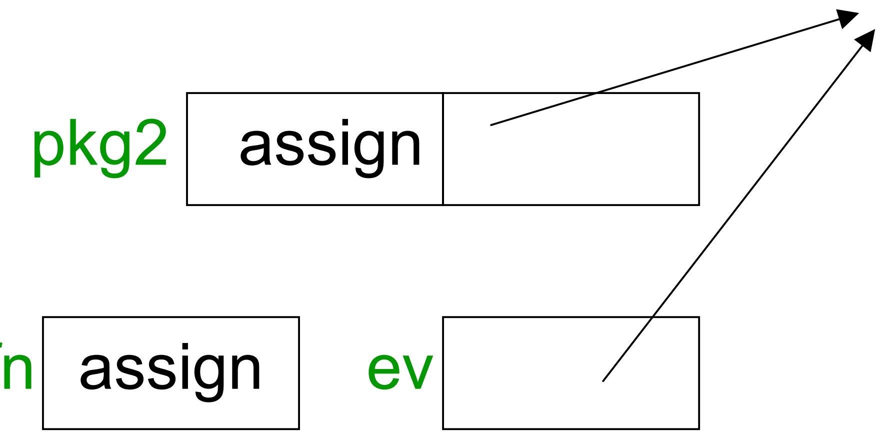
- Either solution is easy to implement
- They are *independent*: A language can have two styles of existential types, one for each restriction
- Cyclone takes solution #1 (no reference patterns for existential fields), making it a safe language without type-invariance of memory!

Are the solutions sufficient (correct)?

- The paper develops a small formal language and proves type safety
- Highlights:
  - Both solutions
  - C-style memory (flattened record values)
  - C-style lvalue/rvalue distinction
  - Memory invariant includes novel "if a reference pattern is for a field, then that field never changes type"

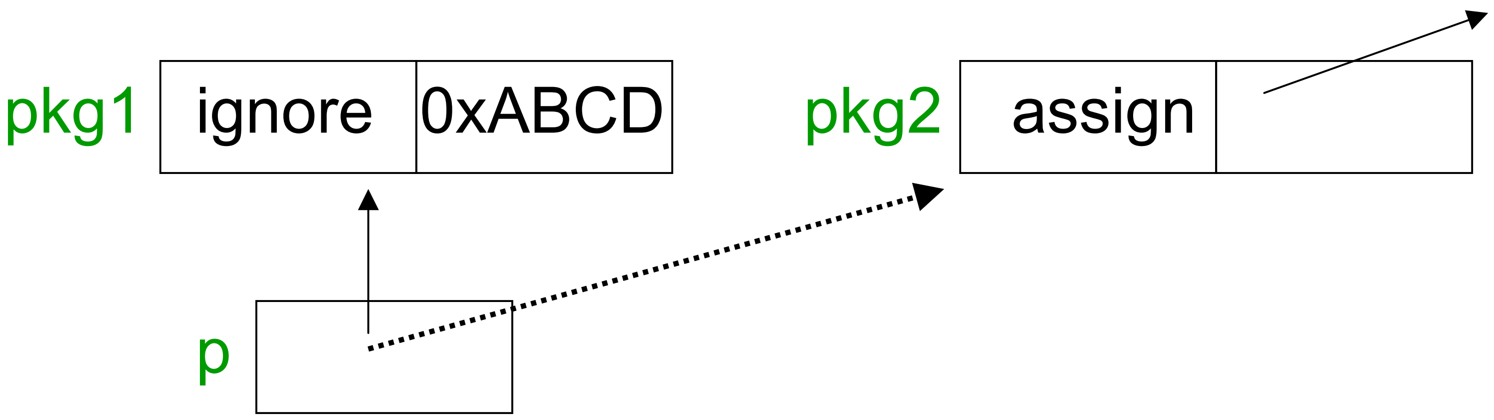
Non-problem: Pointers to witnesses

```
struct T2 { ∃ α.
  void (*f)(int, α);
  α* env;
};
…
let T2{<β> .f=fn, .env=ev} = pkg2;
pkg2 = pkg1;
…
```



Non-problem: Pointers to packages

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



Aliases are fine.
Aliases at the "unpacked type" are not.

Related work

- Existential types:
  - seminal use [Mitchell/Plotkin 1988]
  - 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 (so does Cyclone)

  None have reference patterns or similar

- Linear types, e.g. Vault [DeLine, Fähndrich]

  No aliases, destruction destroys the package

Polymorphic references — related?

- Well-known in ML that you must not give
  `ref []` the type $\forall\alpha.\ \alpha$ **`list ref`**

- Unsoundness involves mutation and aliasing

- Suggests the problem is *dual*, and there are similarities, but it's unclear

- ML has memory type-invariance, unlike Cyclone

Summary

- Existential types are the way to have data-hiding in a safe low-level language
- But type variables, mutation, and aliasing signal danger
- Developed two independent, simple restrictions that suffice for type safety
- Rigorous proof to help us think we've really fixed the problem

New acquired knowledge to avoid future mistakes

[End of Presentation --

Some “backup slides” follow]

Future work — Threads

- For very similar reasons, threads require:
  - atomic assignment (witness-change) of existential packages
  - atomic pattern-matching (destruction) of existential packages
- Else pattern-match could get fields with different witness types, violating type equality
- Future: Type system will enforce a programmer-controlled locking system

What is a good witness?

Without (hidden) run-time types,

we must know the size of (values of) abstract types

```
struct IntIntFn { ∃ α.
  int (*f)(int, α);
  α env;
};
```

α must be int or pointer

```
struct IntIntFn { ∃ α.
  int (*f)(int, α*);
  α* env;
};
```

α can be any type

Interesting & orthogonal issue — come back tomorrow