Summer School on
Language-Based Techniques for Integrating with the External World

Types for Safe C-Level Programming
Part 2: Quantified-Types in C

Dan Grossman
University of Washington
25 July 2007

C-level

- Most PL theory is done for *safe*, *high-level* languages
- A lot of software is written in C
- Me: Adapt and extend our *theory* to make a safe C
  - Last week: review theory for high-level languages
  - Today (+?): Theory of type variables for a safe C
  - Tomorrow: Safe region-based memory management
    - *Uses type variables (and more)!*
  - Off-line: Engineering a safe systems language

How is C different?

- C has "left expressions" and "address-of" operator

```
{ int* y[7]; int x = 17; y[0] = &x; }
```

- C has explicit pointers, "unboxed" structures

`struct T` vs. `struct T *`

- C function pointers are not objects or closures

```
void apply_to_list(void (*f)(void*,int),
                   void*, IntList);
```

- C has manual memory management

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, …

http://cyclone.thelanguage.org/

Context: Why quantified types?

- The usual reasons:
  - Code reuse, container types
  - Abstraction
  - Fancy stuff: phantom types, iterators, …
- *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

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 = λα. { α hd; (L α) * tl; }
- The functions are polymorphic:
  map : ∀α, β. (α→β, L α) → (L β)
- 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":

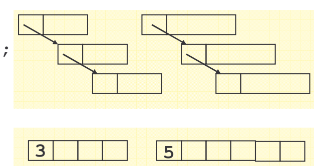
```
int apply(struct T pkg, int arg) {
  let T{<`b> .f=fp, .env=ev} = pkg;
  return fp(arg,ev);
}
```


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 *no problem*

Left vs. Right Syntax

Expressions:

```
e ::= x | λx:τ. e | e(e) | c
    | e=e | &e | *e | (e,e) | e.1 | e.2
```

```
Right-Values: v ::= c | λx:τ. e | &l | (v,v)
Left-Values:  l ::= x | l.1 | l.2
Heaps:        H ::= . | H,x→v
Types         τ ::= int | τ→ τ | (τ, τ) | τ*
```


Of note

Everything is mutable, so no harm in combining variables and locations

- Heap-allocate everything (so fun-call makes a "ref")

Pairs are "flat"; all pointers are explicit

A right value can point to a left value

A left value is (part of) a location

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

Small-step semantics – the set-up

- Two mutually recursive forms of evaluation context

```
R ::= []r | L=e | l=R | &L | *R
    | (R,e) | (v,R) | R.1 | R.2 | R(e) | v(R)
L ::= []l | L.1 | L.2 | *R
```

$$\frac{H,e \to_r H',e'}{H, R[e]_r \to H', R[e']_r} \qquad \frac{H,e \to_l H',e'}{H, R[e]_l \to H', R[e']_l}$$

- Rest-of-program is a right-expression
- Next "thing to do" is either a left-primitive-step or a right-primitive-step

Small-step primitive reductions

$H,\ *(\&l) \to_r H,\ l$ *not a right-value*

$H,\ x \to_r H,\ H(x)$

$H,\ (v1,v2).1 \to_r H,\ v1$

$H,\ (v1,v2).2 \to_r H,\ v2$

$H,\ l=v \to_r$ *need helper since l may be some x.i.j.k (replace flat subtree)*

$H,\ (\lambda x:\tau.e)(v) \to_r H,\ x\to v,\ e$

$H,\ *(\&l) \to_l H,\ l$ *a left-value*

Typing (Left- on next slide)

Type-check left- and right-expressions differently with two mutually recursive judgments

$\Gamma \vdash_r e1:\tau \qquad \Gamma \vdash_l e1:\tau$

- Today, not tomorrow: left-rules are just a subset

$$\frac{}{\Gamma \vdash_r c:int} \qquad \frac{}{\Gamma \vdash_r x:\Gamma(x)} \qquad \frac{\Gamma,x:\tau1 \vdash_r e:\tau2}{\Gamma \vdash_r \lambda x:\tau1.e:\ \tau1\to\tau2} \qquad \frac{\Gamma \vdash_r e1:\tau1\to\tau2 \quad \Gamma \vdash_r e2:\tau1}{\Gamma \vdash_r e1(e2):\ \tau2}$$

$$\frac{\Gamma \vdash_r e1:\tau1 \quad \Gamma \vdash_r e2:\tau2}{\Gamma \vdash_r (e1,e2):(\tau1,\tau2)} \qquad \frac{\Gamma \vdash_r e:(\tau1,\tau2)}{\Gamma \vdash_r e.1:\tau1} \qquad \frac{\Gamma \vdash_r e:(\tau1,\tau2)}{\Gamma \vdash_r e.2:\tau2}$$

$$\frac{\Gamma \vdash_r e:\tau*}{\Gamma \vdash_r *e:\tau} \qquad \frac{\Gamma \vdash_l e:\tau}{\Gamma \vdash_r \&e:\tau*} \qquad \frac{\Gamma \vdash_l e1:\tau \quad \Gamma \vdash_r e2:\tau}{\Gamma \vdash_r e1=e2:\tau}$$

Typing Left-Expressions

Just like in C, most expressions are not left-expressions

- But dereference of a pointer is

$$\frac{}{\Gamma \vdash_l \mathtt{x} : \Gamma(\mathtt{x})} \qquad \frac{\Gamma \vdash_l \mathtt{e} : (\tau 1, \tau 2)}{\Gamma \vdash_l \mathtt{e.1} : \tau 1} \qquad \frac{\Gamma \vdash_l \mathtt{e} : (\tau 1, \tau 2)}{\Gamma \vdash_l \mathtt{e.2} : \tau 2} \qquad \frac{\Gamma \vdash_r \mathtt{e} : \tau *}{\Gamma \vdash_l \mathtt{*e} : \tau}$$

Now we can prove Preservation and Progress

- After extending type-checking to program states
- By mutual induction on left and right expressions
- No surprises
  - Left-expressions evaluate to locations
  - Right-expressions evaluate to values

Universal quantification

Adding universal types is completely standard:

```
e ::= … | Λα. e | e [τ]
v ::= … | Λα. e
τ ::= … | α | ∀α. τ
Γ ::= … | Γ, α
```

L *unchanged*

```
R ::= … | R [τ]
(Λα. e) [τ] →r e{τ/α}
```

$$\frac{\Gamma, \alpha \vdash_r \mathtt{e} : \tau}{\Gamma \vdash_r (\Lambda\alpha.\ \mathtt{e}) : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash_r \mathtt{e} : \forall\alpha.\tau 1 \quad \Gamma \vdash \tau 2}{\Gamma \vdash_r \mathtt{e}\ [\tau 2] : \tau 1\{\tau 2/\alpha\}}$$

Polymorphic-references?

In C-like pseudocode, core of the poly-ref problem:

```
(∀α. α →α) id = Λα. λx:α. x;
int         i  = 0;
int*        p  = &i;
id [int] = λx:int. x+17;
p = (id [int*]) (p); /* set p to (&i)+17 ?!?!*/
```

Fortunately, this won't type-check

- And in fact Preservation and Progress still hold
- So we never try to evaluate something like `(&i) + 17`

The punch-line

Type applications are not left-expressions

- There is no derivation of $\Gamma \vdash_l \mathtt{e}[\tau 1] : \tau 2$
- Really! That's all we need to do.
- Related idea: subsumption not allowed on left-expressions (cf. Java)

Non-problems:

- Types like `(∀α. α list)*`
  - Can only mutate to "other" `(∀α. α list)` values
- Types like `(∀α. ((α list)*))`
  - No values have this type

What we learned

- Left vs. right formalizes fine
- `e[τ]` is not a left-expression
  - Necessary and sufficient for soundness
- In practice, Cyclone (and other languages) even more restrictive:
  - If only (immutable) functions can be polymorphic, then there's no way to create a location with a polymorphic type
  - A function pointer is `(∀α. …)*`, not `(∀α.(… *))`

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

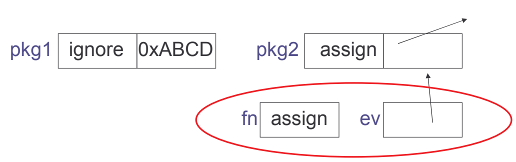
- Violation of type safety
- Two solutions (restrictions)
- Some non-problems

Oops!

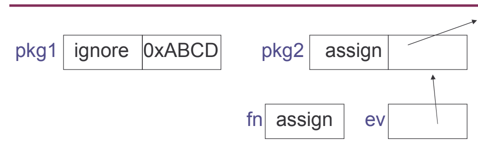
```
struct T {<`a> void (*f)(int,`a); `a env;};

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…

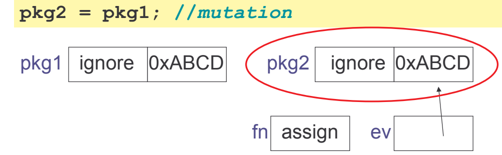

```
let T{<`b> .f=fn, .env=*ev} = pkg2; //alias
```



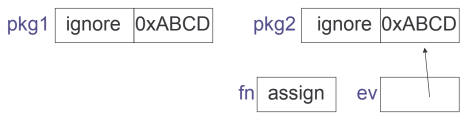
With pictures…




```
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

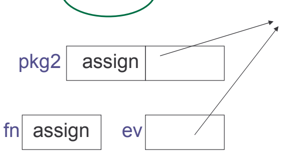
- Either solution is easy to implement
- They are *independent*: A language can have two styles of existential types, one for each restriction
- Cyclone takes solution #1 (no reference patterns for existential fields), making it a safe language without type-invariance of memory!

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"

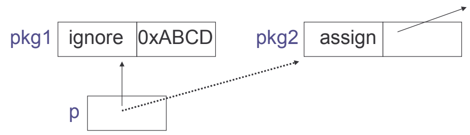
Nonproblem: Pointers to witnesses

```
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 here

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"
- Killer app: Cyclone interface to C functions

```
void memcopy<`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

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

Conclusions

If you see an α near an assignment statement:

- Remain vigilant
- 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
  - "Dan's unsoundness" has come up > *n* times
    - Have (and use) datatypes with the "other" solution