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

Dan Grossman
University of Washington
25 July 2007

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 (*)(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

• 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: smllist@cs.cmu.edu
From: Harper and Lillibrige
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 language 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 List** is a recursive type constructor:
  \[
  L = \text{List}\{ \text{a hd; (\text{L } \alpha \times \text{tl};) } \}
  \]
- 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
  - data representation restricts \text{a} to pointers, \text{int}
  - (why not structs? why not \text{float}? why \text{int}?)
- Not C++ templates

Existential types

- Programs need a way for “call-back” types:
  \[
  \begin{align*}
  \text{struct T} & \{
  \text{int } (*f)(\text{int}, \text{a});
  \text{ a env;}
  \};
  \\
  \end{align*}
  \]
- We use an existential type (simplified):
  \[
  \begin{align*}
  \text{struct T} & \{ \langle \text{a}\rangle
  \text{int } (*f)(\text{int, a});
  \text{ a env;}
  \};
  \\
  \end{align*}
  \]
- **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”
- `struct T` type is just `struct T`
- use requires an explicit “unpack” or “open”:
  \[
  \begin{align*}
  \text{int apply}(\text{struct T pkg, int arg}) & \{
  \text{let T} \langle b \rangle . f = f.p, . env = ev = pkg;
  \text{return} f.p(\text{arg, ev});
  \}
  \\
  \end{align*}
  \]

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:
  \[
  \begin{align*}
  \text{struct T1} & \{ \text{struct T1* tl;}
  \text{ char data[1];}
  \};
  \\
  \text{struct T2} & \{ \text{int len;}
  \text{ int arr[1];}
  \};
  \\
  \end{align*}
  \]

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

• \( e_1 = e_2 \) means:
  – Left-evaluate \( e_1 \) to a location
  – Right-evaluate \( e_2 \) to a value
  – Change the location to hold the value
• Locations are "left values" \( x.f_1.f_2.f_n \)
• Values are "right values", include \( \alpha.x.f_1.f_2.f_n \)
  (a pointer to a location)
• Having interdependent left/right evaluation is no problem

Left vs. Right Syntax

Expressions:
\[
\begin{align*}
e & ::= x | \lambda x.t . e | e (e) | c \\
\alpha & ::= \alpha e | \lambda e | \alpha (e, e) | e_1 | e_2
\end{align*}
\]
Right-Values: \( v ::= c | \lambda x.t . e | \alpha l | (v, v) \)
Left-Values: \( l ::= x | l.1 | l.2 \)

Types \( \tau ::= \text{int} | \tau \rightarrow \tau | (\tau, \tau) | \tau^* \)

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
\[
\begin{align*}
R & ::= [] | L.e | l = R | \alpha L | R \}
\end{align*}
\]
Left-Values: \( l ::= x | l.1 | l.2 \)

Types \( \tau ::= \text{int} | \tau \rightarrow \tau | (\tau, \tau) | \tau^* \)

Small-step primitive reductions

\[
\begin{align*}
H, * (e_1) & \rightarrow H, l \quad \text{not a right-value} \\
H, x & \rightarrow H, H(x) \\
H, \{v_1, v_2\}.1 & \rightarrow H, v_1 \\
H, \{v_1, v_2\}.2 & \rightarrow H, v_2 \\
H, l = v & \rightarrow \text{need helper since \( l \) may be some } x, i, j, k \text{ (replace flat subtree)} \\
H, (\lambda x.t . e)(v) & \rightarrow H, x \rightarrow v \cdot e \\
H, * (e_1) & \rightarrow H, l \quad \text{a left-value}
\end{align*}
\]

Typing (Left- on next slide)

Type-check left- and right-expressions differently with two mutually recursive judgments
\[
\begin{align*}
\Gamma & \vdash e_1 : t \\
\Gamma & \vdash e_2 : t
\end{align*}
\]
• Today, not tomorrow: left-rules are just a subset
\[
\begin{align*}
\Gamma & \vdash c : \text{int} \\
\Gamma & \vdash x : t (x) \\
\Gamma & \vdash \lambda x : t . e : t \rightarrow t \\
\Gamma & \vdash (e_1, e_2) : (t_1, t_2) \\
\Gamma & \vdash e_1 : t_1 \\
\Gamma & \vdash e_2 : t_2 \\
\Gamma & \vdash * e : e \\
\Gamma & \vdash * t : t
\end{align*}
\]

Typing Left-Expressions

Just like in C, most expressions are not left-expressions
• But dereference of a pointer is
  \[ \Gamma \vdash *e : \mathcal{T} \]
  \[ \Gamma \vdash \text{ref}(\alpha) \]

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

Polymorphic-references?

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

\[(\forall \alpha. \alpha \to \alpha) \text{id} = \lambda \alpha. \lambda \alpha. \alpha;\]
\[
\text{int} \quad i = 0;\]
\[
\text{id [int]} = \lambda \alpha. \lambda \alpha. \alpha \times 17;\]
\[
p = (\text{id [int*]}) (p); /* set p to \{i|17 \} ?!?!*/\]

Fortunately, this won’t type-check
• And in fact Preservation and Progress still hold
• So we never try to evaluate something like \{i|17 \} + 17

What we learned

• Left vs. right formalizes fine
• \(e \ [\tau]\) 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 \(\forall \alpha. \text{...}\text{*}, \text{not} \ (\forall \alpha. \text{...*})\)

Universal quantification

Adding universal types is completely standard:

\[ e ::= \ldots | \lambda \alpha. e \ [e \ [\tau]] \]
\[ \tau ::= \ldots | \alpha \ [\forall \alpha. \tau] \]
\[ \Gamma ::= \ldots | \Gamma, \alpha \]
\[ \text{L unchanged} \]
\[ R ::= \ldots | R \ [\tau] \]

\[ (\lambda \alpha. e) \ [\tau] \rightarrow \text{e}[\text{\tau}/\alpha] \]

\[ \Gamma, \alpha, e : \tau \rightarrow \Gamma, e : \forall \alpha. \tau \rightarrow \tau \]

\[ \Gamma, \alpha, e : \forall \alpha. \tau \rightarrow \Gamma, e : [\tau] : \tau \rightarrow \tau \alpha \]

The punch-line

Type applications are not left-expressions
• There is no derivation of \(\Gamma \vdash e \ [\tau]: \tau \)
• Really! That’s all we need to do.
• Related idea: subsumption not allowed on left-expressions (cf. Java)

Non-problems:
• Types like \(\forall \alpha. \alpha \text{ list}\)•
  – Can only mutate to “other” \(\forall \alpha. \alpha \text{ list}\) values
• Types like \(\forall \alpha. ((\alpha \text{ list})\text{*})\)
  – No values have this type

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 Ω

• 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 Ω

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

```c
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");
```

C “call-backs” use void*; we use Ω

Normal Ω feature: Introduction

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

Normal Ω feature: Elimination

Destruction via pattern matching:
```c
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

```c
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:

```c
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 6x.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:

```c
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!

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

With pictures...

```
<table>
<thead>
<tr>
<th>pkg1</th>
<th>ignore 0xABCD</th>
<th>pkg2</th>
<th>assign</th>
</tr>
</thead>
</table>

let T(<`b> .f=xn, .env=*ev) = pkg2; //alias
```

```
<table>
<thead>
<tr>
<th>pkg1</th>
<th>ignore 0xABCD</th>
<th>pkg2</th>
<th>assign</th>
</tr>
</thead>
</table>
```

fn | assign | ev |

```
| pkg2 = pkg1; //mutation |
| pkg1 | ignore 0xABCD |
| pkg2 | ignore 0xABCD |

fn | assign | ev |
With pictures...

```
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(*) (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; 
... 
```

pkg2 assign 
```
Nonproblem: Pointers to packages

```c
#include <stdlib.h>

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/Piotkin 1985]
  - closure/object encodings [Bruce et al, Minimade et al, …]
  - first-class types in Haskell [Lüer]
  None incorporate mutation
- Safe low-level languages with $\exists$
  - Typed Assembly Language [Morisset et al]
  - Xanadu [Xj], 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 $\alpha$, mutation, and aliasing
  - One used $\forall$, one used $\exists$
  - 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 memcpy<\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 a 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