Non-Standard Types

- Using types to collect properties of program constructs
  - Points-to sets for pointers
  - Analysis of pointer safety
  - References:
    - G. Necula, S. McPeak, W. Weimer, "Ccured: Type-safe Retrofitting of Legacy Code", POPL'02
    - J. Condit, M. Harren, S. McPeak, G. Necula, W. Weimer, "Ccured in the Real World", PLDI'03

Non-standard Types

- Non-standard types
  - Use machinery of type systems and type reconstruction to solve for other properties of variables in programming languages
  - E.g., can be used to prove properties of pointer-valued variables
    - CCured - A dialect of C that guarantees pointer usage safety properties
    - Calculating points-to sets using non-standard types

Pointer Analysis w Constraints

- **We had equality constraints in our type reconstruction work.**
  - **How about extension to PLs that allow subtypes?**
    - **Then inequality constraints meaning subtyping, e.g., int<=float.**
- **With systems of inequality and equality constraints, more interesting properties can be described**
  - **E.g., 'points-to sets' for pointer-valued variables, where the inequality refers to set inclusion,**

    int *p,*q; … p = q // means that anything q points to now p can point to, or PtsTo(q) <= PtsTo(p) (i.e., PtsTo(q) ⊆ PtsTo(p))
  - **A satisfying assignment of sets is a solution to these relations and gives a safe estimate of what a pointer can point to during execution.**

CCured

- **A way of ensuring type safety of legacy C code through type inferencing and runtime checks**
  - **Type pointers by their usage**
  - **Insert runtime checks where necessary to check safety (e.g., in presence of pointer arithmetic)**
  - **Use to find bugs involving pointers in real C programs**

Assumptions

- Even in C, a large portion of the program can be shown type safe statically; rest of program will need runtime checks
- Loss of performance due to runtime checks is bearable to ensure type safety
- Pointer types can be designed that match ordinary C usage
  - Physical types (that is, OO coding style using structs)
  - Array traversal through pointer arithmetic

CCured - Intuition

POPL'02, p129-130

- IDEA: think of C as a PL that is a union of 2 PLs, one strongly typed and one an untyped PL requiring run-time type checking
  - SAFE pointers (no pointer arithmetic, no casts; only needs null ptr checks)
  - SEQ pointers (pointer arithmetic, but no casts; needs bounds and null ptr checks)
  - WILD pointers (need type tags, can be cast to other pointer types, needs runtime type checks before any operation)

Example - Ugly C code

POPL'02, p 129

```
int **a, **p; int k, acc; int *e; //declarations
acc = 0;
for (k=0; k<100; k++){
    p = a  + k; //ptr arith
    e = *p; //read array element
    while ((int) e % 2 == 0) { //check tag
        e = * (int **) e; //unbox integer value
    }
    acc += ((int) e >> 1); //strip tag off e
}
```


Example in CCured

POPL'02, p 131

```
dynamic ref SEQ  a;//array
int ref SAFE  p_k; // index
int ref SAFE p_acc; //accumulator
dynamic ref SAFE ref SAFE p_p; //element ptr
dynamic ref SAFE p_e; //unboxer --- all these are implicit in Ccured prgm
p_acc = 0;
for (p_k := 0; !p_k<100; !p_k+1) {
    p_p := (dynamic ref SAFE) (a  ⊕ !p_k); //ptr arith
    p_e :=  !! p_p;//read array element
    while ((int) !p_e % 2 == 0) { //check tag
        p_e := !! p_ e; //unbox integer value
    }
    p_acc := !p_acc + ((int) !p_e >>  1);
    //strip tag off e before adding unboxed value
}
```

! means explicit derereference,
⊕ stands for pointer addition

Replaced all vars by pointers;
Use explicit memory derferencing for all value accesses

Some CCured Inference Rules

Expressions:

POPL'02, p 132

$\frac{\vdash e1: int, \vdash e2: int}{\vdash e1\ op\ e2: int\ (arith)}$ $\quad$ $\frac{\vdash e:t', t' <= t}{\vdash (t)e: t\ (upcast)}$

$\frac{\vdash e1: t\ ref\ SEQ, \vdash e2:int}{\vdash e1 \oplus e2: t\ ref\ SEQ}$ $\quad$ $\frac{\vdash e: t\ ref\ SAFE}{\vdash\ !e : t\ (ptr\ deref)}$

(ptr arith;needs runtime check on bounds)

$\frac{\vdash e1 : WILD, e2: int}{\vdash e1 \oplus e2: WILD}$ $\quad$ $\frac{\vdash e: WILD}{\vdash\ !e: WILD}$

t<= int, int <=t ref SEQ, int <=WILD,

t ref SEQ <= t ref SAFE (with array bounds check)

Soundness of Type System

- **Whole-program type inference**
  - **Pointer arithmetic implies SEQ or WILD**
  - **Bad casting implies WILD**
  - **Try to find as many SAFE and SEQ ptrs as possible**
- **An untyped and typed pointer can never point to same memory location (as aliases)**
  - **Or there would be a way for an untyped pointer to corrupt the memory pointed to by the typed pointer**
- **Cannot have untyped pointer point to a typed pointer**

PLDI'03 Extensions

- **Added mechanism to categorize casts as *upcasts* or *downcasts* to support *physical subtyping* (RTTI -- runtime type info pointer type added)**
  - **Added information to check downcasts**
  - **Allows handling of OO mechanisms such as dynamic dispatch, subtyping polymorphism, checked downcasts**
- **Programmer-specified checking at library boundaries**
- **New separate pointer representation (metadata not interleaved with program data)**

Type checks

- **Fat pointer representation**
  - **rep (t * SEQ) = struct (Rep(t) * p, * b, * e) where b is base and e is end of area that pointer ranges over**
  - **Runtime range check becomes**

    x.b <= x.p <= x.e-sizeof(t)
  - **WILD pointers have bounds within memory area itself;**
    - **Runtime type tag checking as in Lisp**
    - **Writes need to update WILD pointer type tags**
  - **Writes need to verify that a stack pointer is not being written into the heap to prevent dangling pointers**

Casts

PLDI'03

```
struct Figure {
  double (*area)(struct Figure * obj); };
struct Circle {
  double (*area)(struct Figure * obj);
  int radius; } *c;
double Circle_area(Figure *obj) {
  Circle *cir = (Circle*)obj;     // downcast
  return PI * cir->radius * cir->radius;
}
c->area((struct Figure *)c);      // upcast
```

- **Circle is physical subtype of Figure**
- **Blue(upcast) and yellow(downcast) casts are both 'bad' (POPL'02)**
- **Empirical data reports 63% casts were between identical types, leaving 37% of which 93% safe upcasts and 6% downcasts; less than 1% were neither of these.**

Casts and Physical Subtyping

- **Physical subtyping**
  - **If an aggregate t' is laid out in memory exactly as a prefix of the layout of the aggregate t, then t is a *physical subtype* of t'**
- **SAFE, SEQ pointers can be *upcast* in physical subtypes, with some qualifications**
- **Downcasts between physical subtypes handled through new pointer type RTTI that is run-time checked**
  - **Need to save physical type info in new data structure**
  - **Need to encode current run-time type as part of pointer representation**

C Libraries - Pragmas

- **Programmer created wrapper specification for external functions with arguments containing pointers**
  - **About 100 standard library function wrappers included**

```
#pragma ccuredWrapperOf("strchr_wrapper", "strchr")
char* strchr_wrapper(char* str, int chr) {
__verify_nul(str); // check for NUL termination
// call underlying function, stripping metadata
char *result = strchr(__ptrof(str), chr);
// build a wide CCured ptr for the return value
return __mkptr(result, str);
}
```


C Libraries - Metadata

- **Split metadata from actual data about pointers using separate parallel structures**
  - **Need user specification and aid user in finding all places annotation is necessary**
  - **SPLIT pointers cannot point to NOSPLIT types for library compatibility; NOSPLIT pointers can point to SPLIT types**
- **All data operations are split into data and metadata operations**
- **Limitations: library can change data structure that require changes to the metadata**
  - **Requires validation by Ccured on return**

PLDI'03 Data

Module Name	Lines of code	% sf/sq/w/rt	CCured Ratio
asis	149	72/28/0/0	**0.96**
expires	525	77/23/0/0	**1.00**
gzip	11648	85/15/0/0	**0.94**
headers	281	90/10/0/0	**1.00**
info	786	86/14/0/0	**1.00**
layout	309	82/18/0/0	**1.01**
random	131	85/15/0/0	**0.94**
urlcount	702	87/13/0/0	**1.02**
usertrack	409	81/19/0/0	**1.00**
WebStone	n/a	n/a	**1.04**

More PLDI'03 Data

Name	Lines of code	% sf/sq/w/rt	CCured Ratio	Valgrind Ratio
pcnet32	1661	92/8/0/0	**0.99**	
ping			**1.00**	
sbull	1013	85/15/0/0	**1.00**	
seeks			**1.03**	
ftpd	6553	79/12/9/0	**1.01**	9.42
OpenSSL	177426	67/27/0/6	**1.40**	42.9
cast			**1.87**	48.7
bn			**1.01**	72.0
OpenSSH	65250	70/28/0/3		
client			**1.22**	22.1
server			**1.15**	
sendmail	105432	65/34/0/1	**1.46**	122
bind	336660	79/21/0/0	**1.81**	129
tasks			**1.11**	81.4
sockaddr			**1.50**	110

Summary

- **CCured is a viable approach to avoiding errors in C systems code using type inferencing**
- **Works semi-automatically with user annotation of external fcns and some casts necessary for efficiency**
- **Split metadata representation seems useful**