

Union and Cast in Deductive Verification

Yannick Moy

ProVal INRIA Project
France Télécom R&D
INRIA Futurs, LRI, Univ Paris-Sud (Orsay)

July 2007, 2nd

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping
- 3 Cast as Physical Subtyping
- 4 Beyond Hierarchical Unions and Casts
- 5 Conclusion

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping
- 3 Cast as Physical Subtyping
- 4 Beyond Hierarchical Unions and Casts
- 5 Conclusion

Our goal is the **modular** and **automatic** verification of **annotated real C** programs.

Problems with Existing Approaches

	Caduceus/WHY	Tuch-Klein-Norrish
memory model	typed	byte-level
alternate view	no access to bytes	lifted typed heaps
malloc and free	✓	✓
pointer arithmetic	✓	✓
address-of operator	✓	✓
union and cast	synchronization	✓
modularity	✓	✓
automation	possible	difficult
Burstall-Bornat	✓	on typed heaps
separation	aliasing zones	separation logic

Deductive verification of C on a typed memory model is difficult:

- `malloc` and `free`
 - ↳ explicit allocation table + restriction on `malloc` argument
- pointer arithmetic
 - ↳ *logic shift* : $pointer, int \rightarrow pointer$
- address-of operator
 - ↳ introduce indirection level

Deductive verification of C on a typed memory model is difficult:

- malloc and free
 - ↳ explicit allocation table + restriction on malloc argument
- pointer arithmetic
 - ↳ *logic shift* : $pointer, int \rightarrow pointer$
- address-of operator
 - ↳ introduce indirection level
- plus... union and cast

Deductive verification of C on a typed memory model is difficult:

- malloc and free
 - ↳ explicit allocation table + restriction on malloc argument
 - ↳ **makes sense only in a low-level model**
- pointer arithmetic
 - ↳ *logic shift* : $pointer, int \rightarrow pointer$
 - ↳ **breaks type safety**
- address-of operator
 - ↳ introduce indirection level
 - ↳ **not always possible**
- plus. . . union and cast

99% of Unions and Casts Are Hierarchical

Importance of hierarchical unions and casts already noted:

- *Coping with Type Casts in C*, Siff et al., Bell Labs 1999
- *CCured in the Real World*, Necula et al., PLDI'03

What is a hierarchical union or cast ?

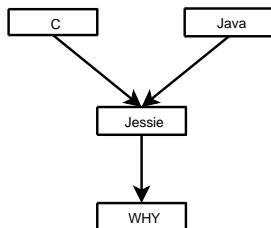
- hierarchical union: Ada variant record or ML datatype
- hierarchical cast: C++ or Java class

Two issues:

- ➔ recognize them: physical subtyping
- ➔ exploit them:
 - ➔ RTTI pointers in CCured
 - ➔ inheritance-aware deductive verification in Jessie

Jessie is:

- an intermediate language suitable for deductive verification
- a target of translation for C and Java
- a tool that translates Jessie programs to WHY programs



Jessie features:

- ✗ union and cast
- ✗ address-of operator
- ✗ untyped malloc
- ✓ explicit structural inheritance
- ✓ typed new and free
- ✓ pointer arithmetic
- ✓ exceptions

Memory Model For Separation and Inheritance

Burstall-Bornat memory model:

- $p \rightarrow f$ becomes $select(f, p)$
- $p \rightarrow f = v$; becomes $update(f, p, v)$
- classical axioms of the theory of arrays:
 - $select(update(f, p, v), p) = v$
 - $select(update(f, p, v), q) = select(f, q)$ if $p \neq q$

Polymorphic Burstall-Bornat memory model:

- $select : (\tau, \rho, \sigma)$ memory, (ρ, σ) pointer $\rightarrow \tau$
- $update :$
 (τ, ρ, σ) memory, (ρ, σ) pointer, $\tau \rightarrow (\tau, \rho, \sigma)$ memory
- τ : translation of a C type in Jessie
- ρ : identifies an aliasing zone
- σ : identifies a type hierarchy

Inheritance Related Predicates

```
type Point = {...}
```

```
type ColorPoint = Point with {...}
```

- σ is Point, the root of the type hierarchy
- each type in the hierarchy is identified by a tag
 - ↳ tag for Point : *Point_tag*
 - ↳ tag for ColorPoint : *ColorPoint_tag*
- dynamic type of a pointer is stored in a functional array
 - ↳ $p <: \text{ColorPoint}$ becomes *instanceof(a, p, ColorPoint_tag)*
 - ↳ $p = \text{new Point}[1]$; becomes *update(a, p, Point_tag)*
- inheritance relations are axiomatized
 - ↳ *instanceof(a, p, ColorPoint_tag) \Rightarrow instanceof(a, p, Point_tag)*
 - ↳ remark that *instanceof(a, p, ColorPoint_tag)* is not defined as *select(a, p) = ColorPoint_tag*

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping**
- 3 Cast as Physical Subtyping
- 4 Beyond Hierarchical Unions and Casts
- 5 Conclusion

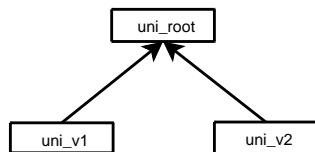
Motivating Example

```
struct packet {
    int d;
    union {
        int v1;
        float v2;
    } uni;
};

float get(struct packet *p) {
    switch (p->d) {
        case 0: return p->uni.v1;
        case 1: return p->uni.v2;
    }
}
```

Motivating Example

```
struct packet {
    int d;
    union {
        int v1;
        float v2;
    } uni;
    //@ invariant d0(p) = p.d == 0 => variantof(p.uni,v1)
    //@ invariant d1(p) = p.d == 1 => variantof(p.uni,v2)
};
float get(struct packet *p) {
    switch (p->d) {
        case 0: return p->uni.v1;
        case 1: return p->uni.v2;
    }
}
```



```
type uni_root = { }  
type uni_v1 = uni_root with {  
    integer v1;  
}  
type uni_v2 = uni_root with {  
    real v2;  
}
```

```
type packet = {  
    integer d;  
    uni_root[0] uni;  
    invariant d0(p) = p.d == 0 ⇒ p.uni <: uni_v1;  
    invariant d1(p) = p.d == 1 ⇒ p.uni <: uni_v2;  
}
```

Coercion and Strong Coercion

Union read as coercion:

- $p \rightarrow f$ becomes in Jessie $(p :> T).f$
- $p :> T$ becomes in WHY `assert (p <: T); p`

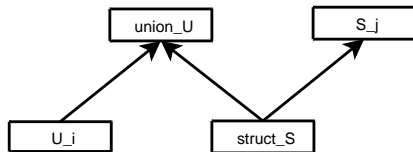
Union write as strong coercion:

- $p \rightarrow f = v$; becomes in Jessie $(p :>= T).f = v$;
- $p :>= T$ becomes in WHY `update(tag_table, p, T_tag); p`

➡ 11 verification conditions generated and proved automatically

Union, Address-of Operator and Pointer Arithmetic

```
union U {  
    int i;  
    struct S {  
        float f;  
        int j;  
    } s;  
} *u;
```



- ➔ `&u->i` becomes `u`
- ➔ `&u->s.j` becomes `u` and forces multiple inheritance
- ➔ `u + k` becomes `u + k`
- ➔ `&u->i + k` is not allowed
- ➔ `&u->s.j + k` is not allowed

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping
- 3 Cast as Physical Subtyping**
- 4 Beyond Hierarchical Unions and Casts
- 5 Conclusion

Motivating Example

```
struct header {
    int d;

};

struct packet1 { int d; int v1; };
struct packet2 {
    struct header h;
    float v2;
};

float get(struct header *p) {
    switch (p->d) {
        case 0: return ((struct packet1*) p)->v1;
        case 1: return ((struct packet2*) p)->v2;
    }
}
```

Motivating Example

```
struct header {
    int d;
    //@ invariant d0(t) =
    //@      t→d == 0 ⇒ typeof(t,struct packet1)
    //@ invariant d1(t) =
    //@      t→d == 1 ⇒ typeof(t,struct packet2)
};
struct packet1 { int d; int v1; };
struct packet2 {
    struct header h;
    float v2;
};
float get(struct header *p) {
    switch (p→d) {
        case 0: return ((struct packet1*) p)→v1;
        case 1: return ((struct packet2*) p)→v2;
    }
}
```

A platform-dependent notion ...

$t \triangleleft t'$ iff for every $((t*)p) \rightarrow f$ at offset o , there is a $((t'*)p) \rightarrow f$ at offset o .

Originates in *Coping with Type Casts in C*, Siff et al., 1999.
Developed in *Physical Type Checking for C*, Chandra & Reps, PASTE'99.

made platform-independent.

$t \triangleleft t'$ iff every $((t*)p) \rightarrow f$ is also a $((t'*)p) \rightarrow g$.

Used in *CCured in the Real World*, Necula et al., PLDI'03

Formal Layout and Layout Algorithm

- *sub-object*: a pair (o, t) of an offset and a type.
- *layout*: set of base type sub-objects of a type.
- *physical subtyping*: subset relation on layouts.
- *layout algorithm*: given a list of types, it builds a layout.

NOT a standard notion BUT...

compilers all have a layout algorithm that takes the definition of a type and returns its layout.

Most Precise Physical Subtyping Relation

Platform-independent layout algorithm returns symbolic offsets:

$$((3 * \text{sizeof}(\text{int}) + \text{alignof}(\text{float}) - 1) \% \text{alignof}(\text{float})) * \text{alignof}(\text{float})$$

Reducible to nonlinear arithmetic over the integers.

Simple algorithm flattens a type to a list of:

- ↳ basic types: `int`, `float`, `int**`, ...
- ↳ pseudo-type t_s for beginning/end of structure

Deciding physical subtyping is just list prefixing.

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping
- 3 Cast as Physical Subtyping
- 4 Beyond Hierarchical Unions and Casts**
- 5 Conclusion

Intersection layout: retain all boundaries from union members.

Translation to Jessie:

- best case: intersection is a union member
- worst case: intersection is an array of characters
- reads translated as accessors on intersection layout
- writes translated directly on intersection layout

Simpler than synchronization:

- synchronization: n^2 interpretation functions
- intersection layout: n interpretation functions

Zero-Initialization and Copying

- common C idiom: `memset` and `memcpy`
- cast to `char*` is particular:
 - ↳ it has a local scope \Rightarrow changes do not propagate
 - ↳ it is opaque \Rightarrow no need for an interpretation function
- ↳ solution: use a local and opaque low-level memory model
- type of consecutive bytes: *type range*
- functions, predicates, axioms from the typed memory model:
select_bytes : ρ memory, ρ pointer, *int*, *int* \rightarrow *range*
- concatenation axioms:
$$j = n + i \Rightarrow \text{concat_bytes}(\text{select_bytes}(m, q, n, i), \text{select_bytes}(m, q, i, j)) = \text{select_bytes}(m, q, n, j)$$
- lifting axioms:
$$\text{sizeof}(\tau) \leq n \wedge v = \text{to_bytes}(w, n, 0) \Rightarrow \text{select}(\text{update_bytes}(m, p, n, 0, v), p) = w$$

- 1 Deductive Verification of C Programs
- 2 Union as Subtyping
- 3 Cast as Physical Subtyping
- 4 Beyond Hierarchical Unions and Casts
- 5 Conclusion**

Some Unions and Casts in Deductive Verification

- 99% of unions and casts are hierarchical:
 - discriminated union
 - cast to physical super-type/subtype
- memory model with inheritance (and separation)
- platform independent version of physical subtyping
- other unions and casts:
 - undiscriminated union
 - ➔ intersection layout
 - cast to `char*` for zero-initialization and copying
 - ➔ local and opaque low-level memory model