class A {
public:
    A();
    virtual void func();
};

class B : public A {
public:
    B() : A() {}
    virtual void func();
};

A::A() {
    func();
}

B *d = new B();

// A::func or B::func?
class A {
public:
    A();
    virtual void func();
};

class B : public A {
public:
    B() : A() {}  // A::func or B::func?
    virtual void func();
};

Motivation: C++ "strange" behaviour

Coding rule HICPP 3.3.13
“Do not invoke virtual methods of the declared class in a constructor or destructor.”
Definition (Coding rules)

Coding Rules constrain admissible constructs of a language to help produce better code.

- Aim at improving
  - Reliability
  - Security
  - Maintainability
  - Portability
  - ...

- Coding standards
  - MISRA-C, MISRA-C++
  - Parasoft’s High-Integrity C++ (HICPP)
  - CERT’s (CMU) Secure Coding Standards for C, C++, Java

- Need of automation and extensibility

- Problem: big gap between semi-formal definition of rules and actual implementation
Diversity of Coding Rules

- "Do not use the ‘inline’ keyword for member functions."
- "Do not call the malloc() function."
- "Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !)."
- "If a virtual function in a base class is not overridden in any derived class, then make it non virtual."
- "All automatic variables shall have been assigned a value before being used."
- "Behaviour should be implemented by only one member function in a class."
Previous prototype based on GCC

- Coding rules (in English)
- Coding rules formalized in CRISP_{C++}
- Coding rules compiled into Prolog
- C++ project source files
- g++' (project build)
- Project facts in Prolog
- Ciao Prolog engine
- Rule violations report

- Coding rule(s) written once in a logic-based formalism called CRISP
- Extract program information using GCC, and store
- Search (using a Prolog engine) for a counterexample
Previous prototype based on GCC

1. Coding rule(s) written once in a logic-based formalism called CRISP
2. Extract program information using GCC, and store it
3. Search (using a Prolog engine) for a counterexample
Previous prototype based on GCC

1. Coding rule(s) written **once** in a logic-based formalism called CRISP
2. Extract program information using GCC, and store it
3. Search (using a Prolog engine) for a counterexample
Previous prototype based on GCC

1. Coding rule(s) written once in a logic-based formalism called CRISP.
2. Extract program information using GCC, and store it.
3. Search (using a Prolog engine) for a counterexample.
Rule HICPP 3.3.13
“Do not invoke virtual methods of the declared class in a constructor or destructor.”
Example of rule formalisation in CRISP

Rule HICPP 3.3.13
“Do not invoke virtual methods of the declared class in a constructor or destructor.”

rule HICPP 3.3.13
violated by Caller : MemberFunction; Callee : VirtualFunction
when exists R : Record such that ( 
  R hasMember Caller 
  and R hasMember Callee 
  and ( 
    Caller is Constructor 
    or Caller is Destructor 
  ) 
  and Caller calls+ Callee 
) .
CRISP$_{C++}$ building blocks

- **Sorts**
  
  *Variable, DataMember, LocalVariable*
  
  *Function, MemberFunction, Constructor*
  
  *Type, PointerType, Record*
  
  *Scope, Namespace, Record, CompoundStatement*
  
  *RecordMember*

- **Predicates**

  *Function calls Function*

  *Record hasImmediateBase Record*

  *Variable hasType NonFunctionType*

  *Function hasType FunctionType*

  *Function.isDefinedIn Scope*

  *Record hasMember RecordMember*

  *PointerType hasPointedType Type*

  *FunctionType hasReturnType Type*
Rules that need some kind of static analysis

Rule HICPP 3.4.2
“Do not return non-const handles to class data from const member functions.”

class A {
public:
    int* foo() const {
        return m_pa;    // permits subsequent mod. of private data
    }

private:
    int* m_pa;
};

void bar() {
    const A a;
    int* pa = a.foo();
    *pa = 10;  // modifies private data in a!
};
Layered rule definition
Integration of information from external analysers

Coding rules (in English)

Coding rules formalized in CRISP$_{C++}$

Coding rule compiler

C++ project source files

Project build

Coding rules compiled into Prolog

Project facts in Prolog

Prolog engine

Rule violations report
Layered rule definition
Integration of information from external analysers

Coding rules (in English) → Coding rules formalized in CRISP_{C++} → Coding rule compiler

C++ project source files → project build

External Analyser → Translation

Knowledge Base about the compiled program

Prolog engine

Rule violations report
Layered rule definition
Integration of information from external analysers

- Coding rules (in English)
- Coding rules formalized in CRISP\textsubscript{C++}
- Coding rule compiler
- C++ project source files
- Project build (LLVM plugin)
- External Analyser
- Translation
- Knowledge Base about the compiled program
- Prolog engine
- Rule violations report
LLVM: Low-Level Virtual Machine

- "Modular and reusable compiler and toolchain technologies" (http://llvm.org)
- Useful components and features
  - clang C/C++ compiler front-end
  - Well specified code representation (LLVM IR)
  - Target-independent optimiser/analyser (-analyze)
  - Alias analysis
  - Link-time optimisation (and analysis)
- LLVM IR is front-end independent
  - Advantage: analysis info can be reused across languages
  - Disadvantage: need to keep track of the correspondence between source-level constructions and IR elements
Alien analysis in LLVM

- **API**
  ```cpp
  enum AliasResult { NoAlias = 0, MayAlias = 1,
                   PartialAlias = 2, MustAlias = 3 }
  
  virtual AliasResult
  alias (const Location &LocA, const Location &LocB);
  
  A **Location** represents a memory location in the LLVM IR
  (assembler) code: Pointer + Size

  - **Diversity of implementations**
    - -basicaa
    - -steens-aa: Steensgaard’s algorithm
    - -ds-aa: Data Structure Analysis algorithm
    - -scev-aa: Scalar Evolution queries
    - -tbaa: Type-based
A domain for alias analysis results

MayAlias (≡ T)

NoAlias  MustAlias  PartialAlias

ConstMem

⊥

• Complementary LLVM API

  virtual bool pointsToConstantMemory (const Location &Loc);

• CRISP expressions for alias queries, examples:

  L1 alias L2 ⊑ NoAlias

  L1 alias L2 = PartialAlias

• Idea: use the above expressions as constraints in CLP
class A {
public:
    int* foo(bool b) const {
        int* local = 0;
        if (b) local = m_pa;
        return local;
    }
private:
    int* m_pa;
};

define linkonce_odr i32* @_ZNK1A3fooEb(%class.A* nocapture %this, i1 zeroext %b)
    nounwind readonly align 2 {
entry:
    tail call void @llvm.dbg.value(metadata !{%class.A* %this}, i64 0, metadata !28)
    tail call void @llvm.dbg.value(metadata !42, i64 0, metadata !30)
    br i1 %b, label %if.then, label %if.end

if.then: ; preds = %entry
    %m_pa = getelementptr inbounds %class.A* %this, i64 0, i32 0
    %tmp2 = load i32** %m_pa, align 8
    tail call void @llvm.dbg.value(metadata !{i32* %tmp2}, i64 0, metadata !30)
    br label %if.end

if.end: ; preds = %if.then, %entry
    %local.0 = phi i32* [ %tmp2, %if.then ], [ null, %entry ]
    ret i32* %local.0
}
class A {
    public:
        int* foo(bool b) const {
            int* local = 0;
            if (b) local = m_pa;
            return local;
        }
    private:
        int* m_pa;
};

• New CRISP predicates
    MemberFunction hasThisLocation Location
    Function returnsLocation Location
    Location isGottenAsOffsetFrom Location

define linkonce_odr i32* @ZNK1A3fooEb(%class.A* nocapture %this, i1 zeroext %b)
nounwind readonly align 2 {
    entry:
        tail call void @llvm.dbg.value(metadata !{%class.A* %this}, i64 0, metadata !28)
        tail call void @llvm.dbg.value(metadata !42, i64 0, metadata !30)
        br i1 %b, label %if.then, label %if.end

    if.then: ; preds = %entry
        %m_pa = getelementptr inbounds %class.A* %this, i64 0, i32 0
        %tmp2 = load i32** %m_pa, align 8
        tail call void @llvm.dbg.value(metadata ![i32* %tmp2], i64 0, metadata !30)
        br label %if.end

    if.end: ; preds = %if.then, %entry
        %local.0 = phi i32* [ %tmp2, %if.then ], [ null, %entry ]
        ret i32* %local.0
}
Rule HICPP 3.4.2
“Do not return non-const handles to class data from const member functions.”
Rule HICPP 3.4.2
“Do not return non-const handles to class data from const member functions.”

rule HICPP 3.4.2
violated by M : ConstMemberFunction
when exists Data, This, Ret : Location;
    FuncT, RetT, PointT : Type
such that (M hasType T
    and T hasReturnType RetT
    and RetT is PointerType
    and RetT hasPointedType PointT
    and not PointT is ConstType
    and M hasThisLocation This
    and M returnsLocation Ret
    and Data isGottenAsOffsetFrom This
    and Ret alias Data ⊆ NoAlias
    )
• Define more rules that need Alias Analysis and find the right abstractions of low-level concepts.

• Find other analyses useful for formalising coding rules and define suitable domains for them.

• Devise a method for solving analysis domain constraints.

• Study rules in which results from different analyses are necessary, and how to combine them.