The pivot – a brief overview

Bjarne Stroustrup
Gabriel Dos Reis
Texas A&M University
http://www.research.att.com/~bs
Overview

• The Pivot
  – Context
  – Aims
  – Organization
  – Basic representations

• High-level program representation for HPC
  – Concept-based checking and transformation
Bell Labs proverbs

• Library design is language design
• Language design is library design

But the devil is in the details
Context for the Pivot

• Semantically Enhanced Library (Language)
  – Enhanced notation through libraries
  – Restrict semantics through tools
    • And take advantage of that semantics

• Provide the advantages of specialized languages
  – Without introducing new “special purpose” languages
  – Without supporting special-purpose language tool chains
  – Avoiding the 99.9% language death rate

• Provide general support for the SELL idea
  – Not just a specialized tool per application/library
  – The Pivot fits here
Example SELL: Safe C++

- **Add**
  - Range-checked std::vector
    - iterators
  - Resource handles
  - Any (if needed) (a typesafe union type)

- **Subtract**
  - Arrays
  - Pointers
  - New/delete
  - Unions
  - Excessively complex/obscure code
    - Uses of undefined construct not caught by compilers (e.g. a[++i] = i)

- **Transforms**
  - Pointers into iterators and resource handles (if porting)
  - New/delete into resource handle uses
Aims

• To allow fully general analysis of C++ source code
  – “What a human can do”
  – Foci
    • Templates (e.g. specialization)
    • C++0x features (e.g. concepts, generalized initializers)
    • Distributed programming
    • Embedded systems
  – Limitation: we work after macro expansion
• To allow transformation of C++ code
  – i.e. production of new code from old source
• Non-aim: handling other languages
  – e.g. Fortran, Java
  – but C and C++ dialects are relatively easy
Related work

• Lots
  – 20+ tools for analyzing C++

• But
  – Most are specialized
    • E.g. alias analysis, flow analysis, numeric optimizations
  – Most are attached to a single compiler/parser
  – None handles all of C++
    • E.g. C++ classes, C++ but not standard libraries
    • Hardly two tools handle the same subset
  – Some are proprietary
  – No serious interoperability
The Pivot

C++ source

Compiler

Object code

Tool 1

IDL

Tool 2

C++ source

Tool 3

Specialized representation (e.g. flow graph)

Tool 4

“information”
The Original Project

• Communication with remote mobile device
  – Calling interface
    • CORBA, DCOM, Java RMI, …, homebrew interface
  – Transport
    • TCP/IP, XML, …, homebrew protocol

• Big, Ugly, Slow, Proprietary, …
  – Why can’t I just write ISO Standard C++?
The original Project
Distributed programs in ISO C++

// use local object:
X x; // remote at “my host”
A a;
std::string s("abc");
// …
x.f(a, s); // a function call

// use remote object :
proxy<X> x;
x.connect("my_host");
A a;
std::string s("abc");
// …
x.f(a, s); // a message send

• “as similar as possible to non-distributed programming, but no more similar”
IRO high-level principles

• Complete: Direct representation of C++
  – Built-in types, classes, templates, expressions, statements, translation units …
  – Can represent erroneous and incomplete C++ programs

• Regular
  – The structure contains all of C++ but doesn’t mimic irregularities

• Programming effort proportional to complexity of task
  – IPR is not just a data structure

• Extensible
  – Node types
  – Information associated with a node
  – Operations

• No integration with compilers
IPR design choices

• Type safe
• IPR (not its users) handles memory management
• Minimal (run-time and space)
  – Minimal number of nodes (unification)
  – Minimal number of checked indirections (usually, virtual function calls)
• Expression-based regular superset of C++
  – E.g. statements, declarations are expressions too
  – C++0x features (most important: concepts – types have types)
• Interfaces:
  – Purely functional., abstract classes, for most users
    • No mutation operation on abstract classes
    • Users don't get pointers directly
  – Mutating (operates on concrete. classes)
    • Users get to use pointers for in-place transformation
  – Traversals (and queries)
    • Several, most not in “the Pivot core”
IPR is minimal

• Necessary for dealing with real-world code
  – Multi-million line programs are not uncommon

• Given the constraint of completeness
  – C++ is complex
    • especially when we use the advanced template features essential for
      high-performance work

• Unified representation
  – E.g., there is only one `int` and only one `1`
  – Type comparison becomes pointer comparison

• Indirections are minimized
  – An indirection (only) when there is a choice of different
    types of information
Original idea (XTI)

- Too large, too slow
Current hierarchy (IPR)

- Compact
- minimal call overhead
XPR (eXternal Program Representation)

- Can be thought of as a specialized portable object database
  - Easy/fast to parse
  - Easy/fast to write
- Compact
  - About as compact as C++ source code
- Robust
  - Read/write without using a symbol table
- LR(1), strictly prefix declaration syntax
- Human readable
- Human writeable
- Can represent almost all of C++ directly
  - No preprocessor directives
  - No multiple declarators in a declaration
  - No <, >, >>, or << in template arguments, except in parentheses
XPR

i : int  // int i;
C : class {  // class C {
    m : const int  //    const int m;
    mm : *const int  //    const int* mm;
    f : (:int,:*char) double  //    double f(int,char*);
    f : (z:complex) C  //    C f(complex z);
}  //
vector : <T> class {  // template<class T> class vector {
    p : *T  //    T* p;
    sz : int  //    int sz;
}  //
Current and future work

• **Complete infrastructure**
  – Complete EDG and GCC interfaces
  – Represent headers (modularity) directly
  – Complete type representation in XPR

• **Initial applications**
  – Style analysis
    • including type safety and security
  – Analysis and transformation of STAPL programs

• **Build alliances**