Libraries as Language Extensions

Sibylle Schupp[1]

[1] Department of Computing Science
Chalmers University of Technology

Why software libraries?

- Domain-specific language (DSL) within a general-purpose language
  - Tools available along the entire tool chain
- High-quality code
  - By definition of a library (ideally)
  - High degree of reuse
- Wide coverage of domains
  - Standard Template Library (STL), MTL, BTL, GTL, VTL, . . .
  - Standardizations (Boost, GLAS, . . .)

Problems in practice

```
vector<Employee> employees;
// initialize employees
Employee& bjarne = *find_if(employees.begin(),
                            employees.end(),
                            HasFirstName("Bjarne"));
```

- Usability
  - Error messages
  - Steep learning curve
  - Particular semantics
- Efficiency
  - “Abstraction penalty”
- Genericity
  - Sufficiently generic?

Libraries as languages

Required:

- Support at library level
    - In the “language of the library”, not just at the level of C++
- Full support for user programs
    - Error messages
    - Semantic checks
    - Optimizations

```
vector<Employee> employees;
// initialize employees
Employee& bjarne = *find_if(employees.begin(),
                            employees.end(),
                            HasFirstName("Bjarne"));
```

Example (bug at library level):

- No mistake in C++ : syntax and semantics correct
- Bug! Program can: abort; read user data; overwrite data
- Bug can be detected *only* at library level!

- STLlint
  - (with Doug Gregor, now Indiana University)
  - Symbolic analysis for the proper use of STL and related generic libraries
- Simplicissimus
  - (with David Musser, R.P.I.; Silicon Graphics and others)
  - Program transformation (“simplification”) of abstract data types and according to abstract algebraic laws
- Conceptual change-impact analysis (CCIA)
  - (with Marcin Zalewski, Chalmers)
  - Reachability analysis for changes in the specification of generic libraries

/ This is the loop optimization pass of the compiler... It identifies basic and general induction variables. .. any quantity that is a linear function of a basic induction variable, i.e. giv = biv * mult_val + add_val. */*

```
static int
general_induction_var (const struct loop *loop, rtx x,
                       rtx *src_reg, rtx *add_val,
                       rtx *mult_val, rtx *ext_val,
                       int is_addr, int *pbenefit, enum ma
  switch (GET_CODE (x)) {
     case PLUS:
      /* Either (plus (biv) (invar)) or
         (plus (mult (biv) (invar_1)) (invar_2)).  */
      if (GET_CODE (XEXP (x, 0)) == MULT)     {
          *src_reg = XEXP (XEXP (x, 0), 0);
          *mult_val = XEXP (XEXP (x, 0), 1);
      }   else  { ....
```

Induction variables in (generic) libraries

. . . *much* more complex!

```
while (iter != students.end()) {
        if (fgrade(*iter)) {
            fail.push_back(*iter);
            iter = students.erase(iter);
        } else
            ++iter; }
```

Show: Iterator “iter” not invalidated (in C++)

i: int	iter: Iterator
+: int $\times$ int $\rightarrow$ int	Iterator operator++(Iter&), Iterator erase(iter)
N	end() *// not loop-invariant!*
$i < N$	position $<$ size *// depends on 'iter'!*

Program analysis at language level:

- Interproc. analysis + pointer analysis + temporaries . . .
- Too imprecise, thus infeasible.

Loop analysis at library level

Abstraction enables symbolic loop analysis

STLlint: *derives* induction expressions (no mere syntactic pattern match as in GCC)

The steps:

1. Assign symbolic values to all variables.
2. Symbolically execute the loop and derive induction expressions by finite differencing and symbolic interpolation.
3. Perform symbolic trip count determination.
4. Solve loop exit conditions.

The power lies in step 1: all variables are treated the same, regardless of the way they are accessed (e.g., pointer chains)

Symbolic analysis is (also) used in parallelizing compilers.

Simplicissmus: Extensible simplification

How to optimize ADTs that are not known at design time of the optimizer?

Problem:

- Optimizer: knows *how* to optimize, but not *what*
- ADT-designer: knows *what* to optimize, but not *how*

Required:

- User-provided semantic information
  - but stable under extensions (of the optimizer)
- Level of indirection
  - but without (run-time) overhead

Simplicissimus:

- For user-defined ADTs: expression-based optimizations
- Experiments (MTL<LiDIA>): saved temporaries, some speedups (1%, 4%, 6%)

Algebraic simplification:

$$x + 0 \longrightarrow 0 \text{ iff } x \in \text{Rightmonoid } M{=}(M{,}0)$$

- Specializes to 16 (!) concrete rules in the SGI compiler
- Extensible: users “register” their types as rightmonoids
- Localized extension: related to ADT, not to simplifier

Implementation:

```
template<class BinaryOpClass, class LeftOperand, class RightOperand>
struct Simplify<Expr<BinaryExpr<BinaryOpClass,LeftOperand,RightOperand>>,
                     RightIdentitySimp>
{
    typedef typename Simplify<LeftOperand>::result result;
};
```

- Template meta-program (“expression templates”)
- Not restricted to math. domains (e.g., array bounds check)

Simplicissimus: “Open compilation”
User-extensible simplification

// User program

g++

Front end

Back end

AST

AST

IR

IR

// Semantic
// information

Simpl gen

Simpl

Simplifier

Conceptual Change-Impact Analysis (CCIA)

Safe refactoring of generic libraries (Marcin Zalewski)

```
// specification
template<typename InputIterator, typename Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

// invocation
... find_if(employee.begin(), employee.end(), HasFirstName(``Bjarne''));
```

Generic libraries

- Specifications and interfaces are “minimal”
- Changes hard to detect
  - “Fat” instances of generic parameters
  - High degree of non-locality

CCIA

- Graph-based dependence analysis
  - Forward & backward reachability; stages
- Applied to “concepts” (upcoming C++ extension)
  - Applicable also to interfaces/implementation pairs etc.

Summary

Library-level abstractions

- add the next layer of abstraction
  - from assembler to higher programming languages, from higher programming languages to libraries
- require proper support
  - static checks, optimizations, refactoring, . . .
  - STLlint, Simplicissimus, CCIA
- enable new kinds of support
  - loop analysis over iterators, ADT-optimizations, specification-tracing
- raise interesting computational questions
  - “X” in the presence of unknown types
  - abstractions as optimization opportunities