Improving the Type System and Variable Access in the Dyninst API

Jeff Hollingsworth

John Davis

© Copyright 1999, Jeffrey K. Hollingsworth, All Rights Reserved.
Need for Type Support in Dyninst

- Access to local (stack variables)
- Complex types
  - non-integer scalars
  - structures
  - arrays
- Correctness debugging
Type Related Classes

**BPatch_type**
- **getName** - returns the symbolic name
- **getSize** - returns the size of the type
- **getComponents** - returns the fields of struct/union
- **type** - returns data class (structure, union, array, …)
- **getType** - return the type of the pointer, array element
- **getLow, getHigh** - returns bounds for arrays
- **isCompatible(Bpatch_type *t2)** - test compatibility of two types

**BPatch_field**
- **getName** - returns the field’s name
- **getType** - returns the Bpatch_type of the field
- **getOffset** - returns the first byte of the field
Interaction with other Classes

- **BPatch_variableExpr**
  - `getType` - returns the type of the variable

- **BPatch_image and BPatch_module**
  - `findType` - looks up a type

- **BPatch_function and BPatch_point**
  - `findVariable` - looks up a variable in a local scope
Implementation

- Use Compiler debugger info (stab records)
  - access to user defined types
  - information about local variables
  - type information for all variables
  - line number to text segment address mapping

- Incremental parsing
  - parse stabs for a module on first use

- dyninst User can define types
  - allows the creation of new types for patched code
Stab Records May Not be Available

- Reasons for lack of Stabs
  - Programs are “stripped”
  - Individual modules may not be compiled for debugging

- User type construction reduces problem
  - Users can create “required” types
  - Can define types for
    - Global variables: often know address
    - Parameters: named by position
    - Define structs and array types
      - setType method of variableExpr

- Local variable access
  - Not possible without stabs
Type Checking

- Ensures that snippets are type compatible
  - can disable type checking at any time
- Based on structural equivalence
  - rules:
    - scalars: same type
    - structures: each field must be compatible
    - unions: each field must be compatible
    - pointers: each points to a compatible type
      - allows more flexibility for missing types
- Error Reporting
  - snippets lack line numbers
Example of Structural Equivalence

- Patched code using a parameter struct
  - If debug info is guaranteed to be available:
    • code can access type, and refer to field
    • full type checking is possible
  - If debug info might be available:
    • can’t depend on program’s definition of struct
    • patch code create structure that is identical to program’s version
    • permits type checking if debug info available
  - If debug info is not available:
    • patch code creates structure
    • no parameter type checking possible
// find all variables defined in an image
BPatch_Vector<BPatch_variableExpr *> vars =
applImage->getGlobalVariables()

for (i=0; i < vars->size(); i++) {
    BPatch_variableExpr *v = (*vars)[i];
    switch (v->getType()->type()) {
        case BPatch_scalar:
            printf("%s is a scalar of type %s\n", v->getName(),
                    v->getType()->getName());
            break;
        case BPatch_structure:
            FieldVector *fields = v->getType()->getComponents();
            for (j=0; j < fields->size(); j++) {
                Bpatch_field *f = (*fields)[j];
                printf("field %s is of type %s\n", f->getName(),
                        f->getType()->getName());
            }
            break;
    }
}
Non-integer Scalars

• **Key types**
  - floats - requires generating floating point expressions
  - different sized integers - 16, 32, and 64 bits are needed

• **Code Generation Issues**
  - register management
    • floats require different registers
    • 64 bit integers often need 32 bit register pairs
  - expression generation
    • many instruction types needed
    • platform specific code for all supported platforms
Re-working Dyninst Code Generation

- **Goals**
  - support floats and ints other than 32 bit
  - enable a peephole optimizer
  - allow better register allocation

- **New register abstraction**
  - aware of types: int, floats, paired registers
  - allow “virtual registers” for register optimization

- **Table driven instruction selection**
  - eases support of multiple types
  - allows description of complex instruction
    - example: increment memory
New Dyninst Utility

- **TCL-based command line tool**
  - provides access to most dyninst features
  - easier to program for simple applications
  - can be used as a simple command-line debugger
    - fast conditional breakpoints
    - dynamic addition of printfs

- **Command Summary**
  - declare: create a new variable in the application
  - cbreak: insert conditional breakpoint
  - print: show contents of application data structures
  - at: insert a code snippet into the application
  - load, run, exit: process creation and manipulation
TCL Command Example

% load application
% declare int counter
% at main entry { counter = 0; }
% at importantFunc entry { counter++; }
% at main exit {
   printf("function called %d times\n", counter);
}
% run
Status

- **Stab Parsing Working**
  - currently only GNU compilers

- **Array and Structure access**
  - completed

- **TCL Command Tool**
  - mostly done - demo today
  - more features needed

- **In Progress**
  - local variable access
  - non-integer scalars