Introduction to libFirm

Sebastian Hack
Daniel Grund
Christoph Mallon

(hack|grund|mallon)@cs.uni-sb.de

Saarland University

Wintersemester 2011/2012
State of the Art Intermediate Representations

- LLVM
- Java HotSpot
- libFirm
Firm Concepts

- Explicit dependency graph
  → All nodes point at their operands
- Every operation is represented by a node
- Every dependency is modelled as edge
  → If there is no (transitive) edge between two nodes, there is no dependency
- Always in SSA form
- Functional program representation
  → Memory is handled as SSA value, too (memory monad)
- One kind of IR in all phases of optimisation and code generation
Modes

- Every node has a mode
- Use Proj nodes to get an element of a tuple
- Header libfirm/irmode.h
mode_b

- Mode representing boolean values
- Produced by `cmp`, consumed by `cond`
- Cannot be directly mapped onto a machine
- Use `mode_Is` for MiniJava boolean parameters, fields, variables
  - All fields having the same size simplifies offset calculation
- Convert integer-boolean to `mode_b`:
  - `cmp ir_relation_less_greater` with 0
  - Opposite direction left as exercise
Entities and Types

- Several kinds of types: primitive (e.g. integers), pointers, classes, methods
- An instance of a type is an entity
  - Entities represent objects, fields, methods, functions
- Every entity is contained in a type
  - Global variables and functions are entities of the global type
  - `get_glob_type()` returns the global type
Type Construction

- **Header** `libfirm/typerep.h`
- `new_type_primitive(mode)`
- `new_type_pointer(type)`
- `new_type_method(n_parameters, n_results)`
  - `set_method_param_type(method_type, pos, type)`
  - `set_method_res_type(method_type, pos, type)`
- **Make an identifier**: `new_id_from_str(char const*)`
- `new_type_class(id)`
- **Add field/method**: `new_entity(owner_type, id, type)`
  - Stepwise construction breaks cycles, e.g. for
    ```c
    struct X { X* next; }
    ```
  - Layout: `set_entity_offset(entity, offset /*in bytes*/)`
Type Construction

Inheritance

- `add_class_supertype(type, supertype)`
- Overwrite method:
  `add_entity_overwrites(entity, overwritten)`
- Dump graph of all types with their relations:
  `dump_typegraph(FILE*)`
System.out.println()

- Model `System.out.println()` as external global function
- Construct function type, which takes one `int` as parameter
- Construct a function entity `print()` in the global type
- Do not build a graph for it
  → Will be implemented in a library
- Call it:

```c
union symconst_symbol sym;
sym.entity_p = print_entity;
ir_node* p = new_SymConst(mode_P, sym,
                          symconst_addr_ent);
ir_node* in[] = { int_value /*value to print*/ }; 
new_Call(mem, p, 1 /*argument*/, in, print_type);
/* Project memory result, set_store(), ... */
```
Graph Construction

- Headers libfirm/irgraph.h and libfirm/ircons.h
- `new_ir_graph(entity, n_local_vars)`
- Block where new nodes get inserted: `set_cur_block(ir_node*)` → Initially there is an empty block ready to use
- Create node in current block: `new_${KIND}(arguments...)`, e.g. `new_Add(left, right, mode)`
- Create node in arbitrary block (later, after construction): `new_r_${KIND}(block, arguments...)`
- Support for on-the-fly SSA construction while building CFG
- Every local variables is assigned a unique number → Identifies current definition in each block
**Graph Construction**

**intra-block**

```cpp
int x, y;
/* ... */
x += y;
```

Let `x` and `y` have id 0 and 1, respectively:

```cpp
set_cur_block(block);
ir_node* const x = get_value(0, mode_Is);
ir_node* const y = get_value(1, mode_Is);
ir_node* const add = new_Add(x, y, mode_Is);
set_value(0, add);
```
Graph Construction

inter-block

- Blocks know their predecessors, not their successors (dependence graph!)
- Create new immature block: `new_immBlock()`
- Immature: The block does not know all its predecessors or they are not completed, yet
- Maturing a block completes the SSA construction for this block: `mature_immBlock(block)`
  → Calculates data dependencies across blocks
- ** Necessary preconditions** for maturing a block:
  - All predecessors are added
  - The predecessors contain all their nodes, in particular no more `set_value()` anymore
- Add a predecessor: `add_immBlock_pred(block, pred)`
- Predecessors are control flow instructions, not blocks
  - `jmp = new_Jmp()`
  - `cond = new_Cond(cmp); t = new_Proj(cond, pn_Cond_true);`
- Return nodes are predecessors of the end block
  `(get_irg_end_block(irg))`
Kind of a Node

- `is_${KIND}(irn)` determines whether a node is of a certain kind, e.g. `is_Add(irn)`
  → Useful for if: `if (is_Add(irn)) { ... }`

- `get_irn_opcode(irn)` returns the kind of a node as enum `iro_${KIND}`, e.g. `iro_Add`
  → Useful for switch:
  ```
  switch (get_irn_opcode(irn)) { case iro_Add: ... }
  ```
get_irn_n()

- get_irn_n(irn, n) gives the $n$th operand of node irn
- Generic function, better use specific functions!
  → Improves readability
  → Dynamic checks are done
- E.g. get_Load_ptr(irn) instead of get_irn_n(irn, 1) to get the pointer where the Load loads from
- get_${OPERATION}_${OPERAND}(irn) are in libfirm/nodeops.h
- ir_node* get_nodes_block(ir_node*) returns the block of a node (i.e. all nodes except blocks)
Projection Nodes

- Created by `new_Proj(pred, mode, proj)`
  - `pred`: Tuple to project an element from
  - `mode`: Mode of the value of the proj
- `proj` is the index of the element to project from the tuple
- Use the symbolic names from `libfirm/nodopes.h`
  - Names are of the form `pn_${OPERATION}_${ELEMENT}`
  - E.g. `pn_Load_res` for the loaded value of a Load
The Bad Node

- If any operand of a node is Bad, the node evaluates to Bad
  → Conceptually a Bad node equals bottom of a lattice
- Exceptions: Basic Blocks and Phis
  → They are lazy, i.e. only the bad input has no value
- Typical use: Some control flow path is determined to be not taken, so is replaced by Bad
The NoMem Node

- Represents the empty subset of the memory monad
- Used e.g. if analysis determines a function does not modify the memory, so the memory input for the call may be NoMem
void exchange(ir_node* old, ir_node* nw);

- Replace `old` by `nw`
- Semantics: All edges pointing at `old` now point at `nw`
- Replace the Add node of `5 + 3` by a Const node `8`
- Replace not-taken conditional jump (ProjX) by Bad
- Works for blocks, too: All nodes in block `old` now are in block `nw` (nodes point at their block)
  → Useful for control flow simplification
- If you suddenly see an `Id` node, you look at a replaced node
  → You should not have remembered this node!
```c
#include "tv.h"

ir_tarval* new_tarval_from_str(str, len, mode);

/* Make Const for the integer literal 123 */
char const* val = "123";
ir_tarval* tv = new_tarval_from_str(
val, strlen(val), mode_Is);
ir_node* cnst = new_Const(tv);

ir_tarval* tarval_add(ir_tarval* a, ir_tarval* b);
ir_tarval* tarval_sub(ir_tarval* a, ir_tarval* b,
ir_mode* dst_mode);

/* ... */

ir_node* new_Const(tarval*);
```

- **target values**
- Model constants of the target machine
- For calculations on constants only use tarvals
More on tarvals

/* Return relation of two tarval: */
* ir_relation_{less,equal,greater,unordered} */
ir_relation tarval_cmp(ir_tarval*, ir_tarval*);

/* For common values (see the header for more) */
ir_tarval* get_tarval_null(ir_mode*);
ir_tarval* get_tarval_one(ir_mode*);
ir_tarval* get_tarval_b_false(void); // Boolean false
ir_tarval* get_tarval_b_true(void); // Take a guess (:
int tarval_is_null(ir_tarval*);
int tarval_is_one(ir_tarval*);

/* Useful for analyses */
ir_tarval* get_tarval_bad(void); // Top
ir_tarval* get_tarval_undefined(void); // Bottom
Example

```c
void fold_Mul(ir_node* mul)
{
    assert(is_Mul(mul));
    ir_node* const l = get_Mul_left(mul);
    if (!is_Const(l))
        return;
    ir_node* const r = get_Mul_right(mul);
    tarval* const tv_l = get_Const_tarval(l);
    if (tarval_is_one(tv_l))
    {
        exchange(mul, r);
    }
    else if (is_Const(r))
    {
        tarval* const tv_r = get_Const_tarval(r);
        tarval* const tv_m = tarval_mul(tv_l, tv_r);
        ir_node* const cnst = new_Const(tv_m);
        exchange(mul, cnst);
    }
}
```
Walker

```
#include "irgwalk.h"

typedef void irg_walk_func(ir_node*, void* env);

void irg_walk_graph(ir_graph*,
    irg_walk_func* pre, irg_walk_func* post, void* env);
void irg_block_walk_graph(ir_graph*,
    irg_walk_func* pre, irg_walk_func* post, void* env);

▶ Performs depth-first search starting at End node/block
▶ pre: Called before any operand is visited
▶ post: Called after all operands are visited
▶ pre/post may be null pointers
▶ env passed to walker in turn passes it to each invocation of
  the callbacks (pointer to arbitrary context)
▶ Operand visited from other node before will not be visited
  again
```
Out Edges

```c
#include "iredges.h"
#include "irprintf.h"

void print_users(ir_node* const irn) {
  ir_edge_t const* edge;
  foreach_out_edge(irn, edge) {
    ir_node* const src_irn = get_edge_src_irn(edge);
    int const src_pos = get_edge_src_pos(edge);
    ir_printf("%+F␣--%d-->␣%+F
", src_irn, src_pos, irn);
  }
}

dges_activate(irg);  // Calculate out edges
dges_assure(irg);    // ... if not done already
dges_deactivate(irg); // Switch them off
```

- For worklist algorithm: Analysis information changes, so add all users back into work list
- Problem: You might see dead code!
The Link Field

- Every `ir_node` has a `void*` pointer
- Access: `get_irn_link(irn)` and `set_irn_link(irn, link)`
- `libFirm` does not use this pointer internally
- You can put any information there
  → Useful for analysis information, e.g. . . .
  → Tarvals for constant folding
  → Linked list of phi nodes for control flow simplification
- Field may contain garbage
  → First initialise to known good state with `irg_walk_graph()`
  → E.g. null pointer, bottom, . . .
Debug Support

$ FIRMDBG=".create1234" gdb --args ./mjavac test.java

▶ Stops in debugger when node with number 1234 is created.

(gdb) print gdb_node_helper(irn)

▶ Prints information about the node
▶ More events and several useful macros for gdb: www.libfirm.org/Debug_Extension
www.libfirm.org

- Graph viewer: yComp
- Several papers
- (Some) documentation