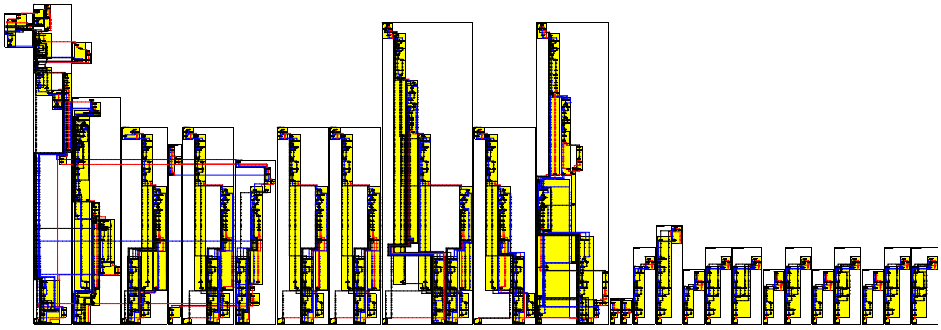
libFIRM
A Library for Compiler Optimization Research Implementing Firm.

Götz Lindenmaier
goetz@ipd.info.uni-karlsruhe.de



Technical Report Nr. 2002-5
ISSN 1432-7864

Institut für Programmstrukturen und Datenorganisation
Prof. Dr. G. Goos
Fakultät für Informatik
Universität Karlsruhe

September 18, 2002

Contents

List of Figures

Chapter 1

Introduction

This tutorial describes the Firm library and how to use it. The Firm library implements the Firm intermediate representation (ir) as described in [TLB99]. In addition it supplies data structures to represent the type structure of the source program, a constant table and other modules necessary to represent a complete source program. Further it contains interfaces for construction of intermediate code from the front end and to access the ir. Some basic optimizations are included.

This tutorial explains how the library supports constructing intermediate code from a front end and describes the constructs expected to represent object oriented and imperative constructs and languages. Two chapters deal with existing standard optimizations and existing analyses. Another chapter explains how to transform the ir to implement additional optimizations. Finally the tutorial explains how to handle debug information for the compiled program.

Basically the Firm library supplies data structures to represent most of the information available in a program. It contains modules to represent type information, all entities specified by a program and, of course, program code. To represent this program code the library implements a representation according to the intermediate language Firm. Firm and the employed data structure allow to efficiently perform optimizations that rely on SSA form.

In addition the library supplies an interface for the attachment of a front end that automatically constructs the SSA representation and some standard optimizations.

This tutorial assumes that the reader is familiar with the operations, modes and basic structure of Firm [TLB99].

Chapter 2

Using libFIRM

libFIRM is written in C and all its interfaces are C function calls. It is not necessary to access any data structures directly. Nevertheless this is possible by including additional headers (those ending in `_t`). These headers do not belong to the distribution and are subject of change.

To program with libFIRM only one header needs to be included: `firm.h`. This makes all the functionality to construct intermediate representation available. For debugging and testing purposes `irdump.h` and `irvrfy.h` are useful. These headers needs to be included additionally.

To compile with the library you must pass your compiler the name of the library, the path to the library and the path to the header files. In addition you need the Gnu multi precision library gmp and gmp.h.

The comments in the library are in `autodoc` format. You can generate documentation in HTML by calling configure with `--enable-autodoc` and then `make autodoc`. You need to have `robodoc`.

Chapter 3

Representation of a Program with the Firm Library.

libFIRM supports representing most of the information available in a program as well as structures defined by the translated programming language. It contains modules to represent type information, all entities specified by a program and, of course, program code.

Entities represent all program known objects that occupy some memory as, e.g., procedures, global variables, constants, dynamically allocated data structures or stack frames. A type is a description of the size and content of an entity.

3.1 Representation of Types

The type module supplies functionality to represent source language types. These include language defined, programmer defined and implicitly defined types, as in C the basic type `int`, a type defined with a `typedef` and the type of a method. These types serve two purposes within Firm:

They are used to hide access functionality (field access, polymorphy, inheritance). To generate this functionality the type information is necessary. Further full type information allows certain analysis techniques.

Different concepts in programming languages, the memory layout of a type and the associated functionality allow to distinguish different kinds (or classes) of types. The representation of a type in libFIRM depends on the kind of type. libFIRM distinguishes the following kinds of type: class types, structure types, method types, union types, array types, enumeration types, pointer types and primitive types. These kinds of types are explained

in more detail below.

libFIRM associates with the type kinds a basic notion of a memory model, e.g., when a certain set of memory locations are to be treated together as a larger structure. It does not specify a full memory model or calling conventions. These must be modeled explicitly, e.g., by creating a data type and entity representing a dispatch table.

Functionality for the representation of types is supplied in the header `type.h`. Functionality to distinguish the kinds of types is to be found in header `tpop.h`.

3.1.1 Atomic and Compound Types

Often it is necessary to know whether an entity represents a value that can be handled directly by the target code – in this case Firm nodes – or whether first parts of the entity must be selected. Atomic entities map directly to a Firm mode. Compound entities can not be manipulated as such, only parts of compound entities can be handled by the processor. This property of entities basically depends on the type of the entity. Therefore we also distinguish atomic and compound types.

Atomic types are primitive, pointer and enumeration types. Compound types are class, union, structure and array types. Method types are none of both, as they can not be accessed at all. See also 3.2.5.

3.1.2 The Basic Representation of a Type

A libFIRM `type` is a data structure used to describe the types in the program. It contains some attributes common to all types, a flag indicating the kind of type and a set of attributes specific for a certain kind of type.

The common attributes are:

kind A type tag always set to `k_type`.

tpop An opcode representing the kind of type. This opcode is readable, but not writable. libFIRM types can only be allocated with a certain type opcode.

name The name used for this type in the source program. For types that are not explicitly named (e.g., method types) the front end must come up with some name. This name should differ from the name of entities of this type. The name is useful for debugging the compiler, for debug support and to implement functionality as reflection.

state The state of the type. The state represents whether the layout of the type is undefined or fixed (values: `layout_undefined` or `layout_fixed`). Compound types can have an undefined layout. The layout of the basic types primitive and pointer is always `layout_fixed`. If the layout of compound types is fixed all entities of the type must have an offset and the size of the type must be set. A fixed layout for enumeration types means that each enumeration is associated with an implementation value. See also Section 3.1.6.

mode The mode to be used to represent this type. For an explanation of modes see Section 3.3.5. Only atomic types have this attribute. For these it also describes the size of the type. The mode must be set in atomic types if their layout is fixed.

size The size of an entity of this type in memory. This number is given in bytes. The size is determined when fixing the layout of compound types or can be computed depending on the mode. This attribute is only set if the type is in the state for fixed memory layout.

visited This attribute can be used for traversing the type information. For a description how to traverse the type information see Chapter 8.

link A field to refer to additional information about the type, e.g., to be used by analyses to store temporary information.

The following sub chapters describe the semantics of the different kinds of types and the attributes private to these types.

3.1.3 Primitive Types

Basic values as integer values, floating point values or characters are values of a primitive type. The primitive type represents a program language type for such values. Each primitive type must be associated with a mode that specifies how the type will be represented on the target machine. The mode automatically specifies the parameters of the representation as size or alignment. The size attribute of primitive types is set when setting the mode of the primitive type and can not be changed individually. Primitive types are always in state `layout_fixed`. Only integer, floating point and character modes are allowed for primitive types.

Primitive types have no private attributes.

3.1.4 Pointer Types

Pointer types allow to carry information about pointers from the source program to the ir. This information is carried by the attribute `points_to` that

refers to the type of entities the pointer can point to. (For an introduction to libFIRM entities see Section 3.2.) Values of a pointer type are always represented with `mode_p` on the target machine. Therefore the mode and size of a pointer type are fixed. Pointer types are always in state `layout_fixed`.

Attributes of pointer types:

points_to Refers to the type of the entity this pointer points to.

Figure 3.1 gives an example of the use of pointer types.

```
class A { ... }
class B {
    a A;
}
```

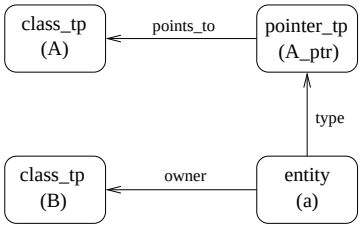


Figure 3.1: Use of pointer type.

3.1.5 Method Types

Method types represent the types of methods, functions and procedures. They contain a list of the parameter and result types, as these are part of the type description. Programming languages sometimes define implicit parameters and results of methods. These must be represented explicitly in the libFIRM method types.

Method types do not require a layout. The layout is by default `layout_fixed`. Further methods do not require a mode or a size as their memory is neither readable nor writable by Firm operations. But as entities of type method represent a pointer to the method the mode of method types is set to mode_p, and the size is set corresponding to the size of the mode. Setting the layout or size of a method completes without error but has no effect.

Attributes of method types:

n_params Number of parameters to the procedure.

param_type A list with the types of parameters. This list is ordered. The n^{th} type in this list corresponds to the n^{th} element in the parameter tuple that is a result of the start node. (See `ircons.h` for more information.)

n_ress The number of results of the method. In general, procedures have zero results, functions one.

res_type A list with the types of results. This list is ordered. The n^{th} type in this list corresponds to the n^{th} element in the result tuple returned by a *Call* node. (See `ircons.h` for more information.)

3.1.6 Type Kinds Requiring a Layout

Compound data types as arrays, structures, unions and class types require a layout. In programming languages as Java the layout of such data structures is not specified by the source program. Instead it can be determined by the compiler and be subject of optimization. Further enumeration types need not be completely specified by the source language. The mode to implement the enumeration and the mapping of the enumerators to the values of this mode can be specified by the compiler.

To represent this, two states exist for types. In the first state the layout is unspecified, in the second it is fixed. If the layout of a type is unspecified this requires access by *Sel* and *SymConst* nodes. Optimizations of the type as removing unnecessary entities and determining the layout are possible. The size of entities must be represented by symbolic constants.

If the type has a fixed layout the offset attribute in the entities of the type must be set. The entity access by *Sel* nodes can be replaced by explicit pointer arithmetic. The size attribute of the type must be set. Symbolic constants representing the size of the type can be replaced by explicit constants. Specifying the final layout for all data types is a part of lowering.

3.1.7 Enumeration Types

Enumeration types allow to delay mapping the verbose names of enumerators to target values. If the source language does not specify this mapping, then unnecessary enumerators can be removed by an analysis to achieve a more compact representation.

Undefined layout of an enumerator means that the mapping of the enumerators to target values is not yet established. If the layout is fixed the

mode of the type must be set and all enumerators must be mapped to a target value of this mode. The mode of an enumerator must be an integer mode.

Attributes of enumeration types:

n_enums The number of enumerators in this enumeration.

nameid A list of source program names for the enumerators. An enumerator can be referenced by a *SymConst* using this name.

enum A list of target values representing an enumerator. This list must be filled if the type is in state `layout_fixed`. A target value at position n represents the enumerator referenced with the name at position n in the `nameid` list. *SymConst* nodes can be replaced by *Const* nodes with the corresponding target value.

3.1.8 Array Types

An array type describes a set of entities of equal type. A single Firm entity data structure that represents all these entities is associated with the array type. We call the entities of arrays 'elements of the array' instead of 'members of the array'. Elements can be of all types except method types.

Elements of an array are accessed hierarchically. We call the steps in the hierarchy dimensions. Each dimension defines a set of virtual addresses of either sub-arrays or elements. These sub-arrays have exactly one dimension less. These sets of virtual addresses are indexed by a contiguous set of integers. This set is either an interval or increasing or decreasing from a starting point.

Sel nodes used for selecting elements from an array must specify the indexes of the element to select. The order of the bounds in the array type as well as the order of the index operands of the *Sel* node are fixed. They correspond by their position.

The indexes are resolved by a fixed order. This order is specified explicitly in the array type. To compute the address of an element the address operand of the *Sel* node and the first index (first in the order specified) is taken to compute the address of the first sub array. This address with the second index delivers the address of the third sub array and so forth.

Figure 3.2 shows an example of a *Sel* node that accesses a three dimensional array. Index value `2` corresponds to dimension `0..5`, index value `3` to dimension `3..10` and so forth. The order attribute in the type specifies that the *Sel* node dereferences the pointer passed using the second index to get an array of size `[0..5, 0..7]`. Within this array it uses the first index

```
int intfield [0..5, 3..10, 0..7]

... = intfield[2,3,6];
```

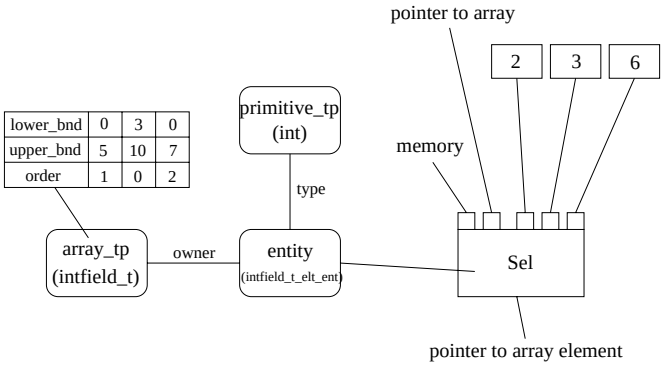


Figure 3.2: A three dimensional array.

to get an array of size `[0..7]`. Now it accesses the element specified by the third index to get the array element.

This allows to represent a rich variety of different array semantics. Increasing and decreasing index sets allow to represent arrays with a dynamic size. (The size of the array is determined during run-time, but then constant. The array type can not represent growing arrays.) If both bounds of an array are unknown at compile time, one of the bounds must be used as an offset to normalize the index to a static starting point of the array. This representation of arrays also abstracts from the implementation of the array. It does not determine whether multidimensional arrays must be represented by one dimensional arrays where each dimension points to an array of the lower dimension, or whether the dimensions can be resolved by pure address computation.

The virtual address spaces are represented by intervals `[lower_bound, upper_bound[`. If the array has a dynamic size one of the bound is unspecified. One of the bounds must always be specified. The bounds are represented by Firm nodes allocated as constant code, see Section 3.2.6.

If the layout of an array is undefined the dimensions of the array can be changed. Here all possible changes are allowed, as reordering the dimensions, linearizing the array or changing the entity type. If the layout is fixed, C integers ordering the dimensions must be assigned to the order fields and no further change of the type is allowed.

Attributes of array types:

n_dimensions The number of array dimensions.

lower_bound A list of lower bounds of each array dimension.

upper_bound A list of upper bounds of each array dimension.

order The order of the array dimensions. This must be integers starting at 0. The index of dimension 0 is resolved first, the index with the highest order number accesses the array elements.

element_type The type of the array elements.

element_ent The entity representing the array elements. This entity is automatically allocated with the array type.

3.1.9 Structure Types

A structure type is a collection of named components which are entities themselves. These components are called members. An entity of a structure type occupies a piece of memory which contains disjunct memory regions for the members. A member can be of a type of any kind except method types (`type_method`). Nevertheless a member can be a pointer to a method. A member is represented by an entity whose owner is the structure type.

The representation for a structure has two states. The first does not specify the layout of the members within the memory region of the structure nor the size of the structure. It is not specified how members can be accessed within the structure. This is only possible by utilizing *Sel* nodes that return a pointer to a member entity given a pointer to a structure entity. This state allows optimizations as removing members, reordering members or even splitting the structure.

The second state fixes the memory layout. All member entities are annotated with an offset. The members can still be accessed with *Sel* nodes, but also by explicit pointer arithmetics adding the offset to a pointer to the structure entity. Further the total size in Bytes of the structure is set in the `size` attribute. Optimizations of the layout are forbidden.

If the layout is fixed the entities of the structure type may not be of dynamic or static allocation. They are allocated automatically.

Attributes of structure types:

n_members The number of members of the structure.

member A list of the member entities. The owner of these entities must be the structure type.

3.1.10 Union Types

A union type describes a memory region that is big enough to hold any of its members. It contains only one of its members at a time – depending on the interpretation of its content.

If the layout of a union type is fixed the size is set.

Attributes of union types:

n_members The number of members of the union.

member A list of the member entities. The owner of these entities must be the union type.

3.1.11 Class Types

A class type is a collection of named components which are entities. These components are called members. An entity of a class type occupies a piece of memory which contains disjunct memory regions for the members. In this it resembles structure types. But there are two differences to structure types:

Class types allow to specify inheritance. For this they can refer to a list of super- and subclasses. Further class types can contain method members. These method members must be static entities, see 3.2.1.

A class type does not specify all members it contains. It inherits all members of its super classes that are not overwritten by a member of this class. Entities representing the members of a class can specify the members of super classes they overwrite. In some cases this straight forward representation of inheritance is not appropriate for the compiled source language. Then it is necessary to construct additional entities in the subclasses that specify this inheritance explicitly, see also the example in Figure 3.4.

If the entity specified in a *Sel* node is overwritten by entities in a subclass the *Sel* will select the proper entity in the subclass if it gets a pointer to the subclass as argument. There are no members that are neither inherited nor overwritten. Members must be at least inherited to guarantee proper access and to reserve memory.

The libFIRM representation of inheritance is very general. It is designed to represent the type structure of a variety of programming languages. Lowering the full functionality of the type concept requires to foresee situations that are irrelevant for specific programming languages. Therefore the lowering phase should exploit additional knowledge about the programming language or establish certain properties (as, e.g., single inheritance) by simple analyses.

```
class A {
     int a;
     void f() { }
     void g() { }
}
```

```
class B extends A {
     int b;
     void f() { }
     void h() { }
}
```

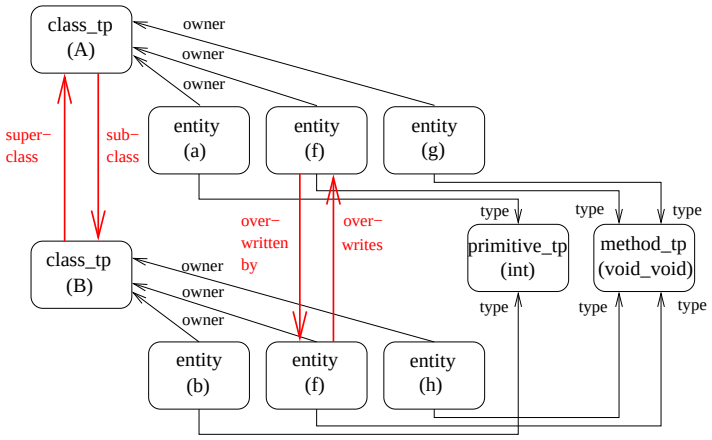


Figure 3.3: Use of sub/superclass and overwrites relation.

The representation of class types has the same two states as structure types, see Section 3.1.9.

Fixing the layout of classes must respect the specified inheritance relations. If a class inherits or overwrites members from a superclass this must be respected by the offsets used for the member entities if the layout is fixed. The size of a class must include the space for members of the super classes. In some cases as multiple inheritance it can be necessary to introduce complex functionality to assure correct access. It can be necessary to change *Sel* nodes to reflect this functionality. The super/subtype and overwrites/overwritten relations are oblivious after fixing the class layout.

Further libFIRM distinguishes two peculiarities of class types. The peculiarity describes the purpose of the type. The standard class type is of peculiarity `existent`. This means the type describes entities that can actually appear at runtime. A class type that is never allocated during the runtime of the program is of peculiarity `description`. It is only used to describe certain properties of class types as, e.g., Java Interfaces do. This information can be used for analyses or directly for optimizations.

Figure 3.3 gives an example of the use of class types.

```
class A {
     int a;
}
```

```
class B1 extends A {}
class B2 extends A {}
class C extends B1, B2 {}
```

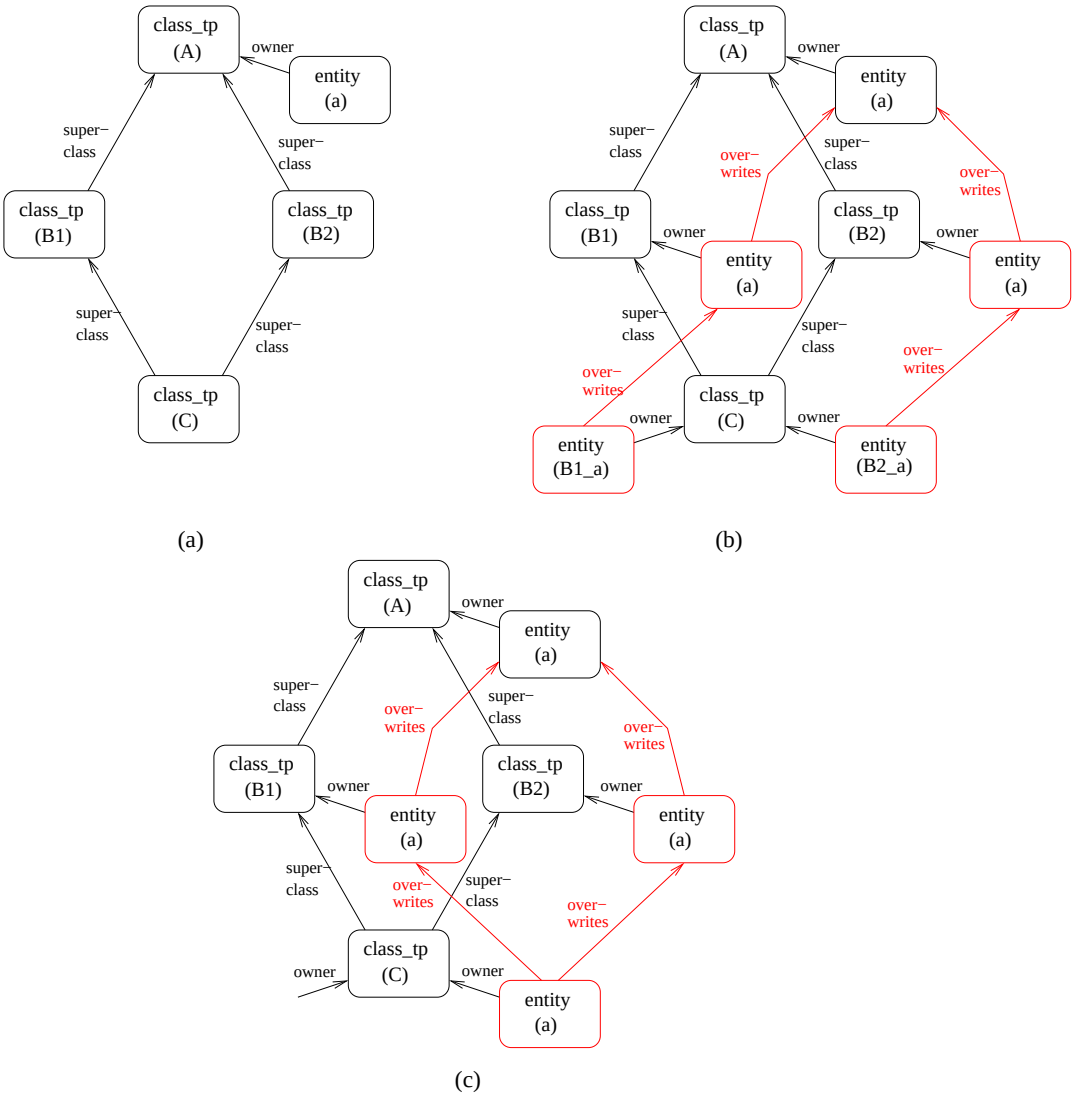


Figure 3.4: Example for multiple inheritance.
Representations (a) and (b) are equivalent in Firm. (b) can be generated from (a) in a later compiler phase. If a source language requires a representation according to (c) for the program example, this representation must be built directly by the front end.

Attributes of class types:

n_members The number of members listed in this class type. Does not include inherited members except if these are represented explicitly.

member A list of the member entities. The owner of these entities must be this class type.

n_supertypes The number of super types of this class type.

supertype A list of the super types of this class type.

n_subtypes The number of subtypes of this class type.

subtype A list of the subtypes of this class type.

peculiarity A flag indicating the peculiarity of the type. Possible values are `description` and `existent` (default value). The peculiarity `inherited` can only be used with entities, see 3.2.3.

3.2 Representation of Entities

The entity module is implemented in the header `entity.h`. An entity represents any program known object that occupies some memory. The size and content of this memory is specified by a type (except method types, here the size depends on the code size of the method.) The entity has the task to represent this memory and the address that identifies this memory. The address of an entity is the lowest address of an addressable unit of the memory occupied by the entity, i.e., parts of the entity can not be accessed by negative offsets to this address.

3.2.1 Allocation of Entities

Further entities have the task to specify how, when and where this memory is allocated.

There are three cases how and when entities are allocated:

- The entity is statically allocated, i.e., during compile time. This is the case for global variables and method entities. To represent a statically allocated entity a special flag in the entity needs to be set (`static_allocated`).
- The entity is dynamically allocated. This is the case for entities that are allocated by explicit allocation with the *Alloc* node. These entities are not explicitly represented as they are not statically known. This is also the case for stack frames of methods which are dynamically allocated with the entry of a method. (`dynamic_allocated`)

- The entity is automatically allocated. This is the case for members of compound objects. They are allocated whenever an entity of the compound object is allocated. (`automatic_allocated`)

Where an entity is allocated is specified by the owner of the entity. If entities are members of compound types (class, structure, union, array) this type is the owner of the entity. Entities representing parameters, results or local variables of methods are owned by a method.

Local and Global Variables as Entities

Local and global variables are a special case as no natural owner exists for them in the representation.

Global variables are statically allocated in a dedicated memory region. libFIRM models this region as a dedicated class type called Global_Type. Per definition exists exactly one entity of this type which roughly corresponds to the data segment of the program. This class type is automatically generated when initializing the Firm library. Entities for global variables and for methods that are not owned by a class of the program have this type as owner.

To get a pointer to a global entity it can be selected from the global pointer that is supplied by the *Start* node.

Local variables of a method are allocated with every entry to the method, typically by using a stack frame. The stack frame is modeled explicitly by a class type in Firm. Methods in this class type are "inner" methods as, e.g., in Pascal[1]. Members of the frame type will be newly allocated with every execution of the method.

To get a pointer to a local entity it can be selected from the frame pointer that is supplied by the *Start* node.

3.2.2 External Visible Entities

We need a special marking in case of partial compilation. In this case entities differ in their visibility to program parts not compiled with the current compilation run. We call the part not included in the current compilation the external part of the program. There exist three cases:

[1]Rational: We must model these local variables as entities. The matter is the owner of this entity. The method type is inappropriate as owner as there can be several method entities for one method type with different local variables. The entity representing the method would be suitable as owner as it would be a one to one relation. Unfortunately only types are allowed as owners. So we need to model a separate type for the stack frame.

local The entity is not visible to the external part.

external visible The entity is visible to the external part. This is only possible for static entities. The memory for the entity is allocated within the currently translated part of the program.

external allocated The current part of the program uses the entity, but it is defined and allocated in the external part. This also is only possible for static entities.

External visible entities may not be optimized away.

3.2.3 Peculiarity of Method Entities

libFIRM distinguishes three peculiarities of method entities. The standard method entity is of peculiarity `existent`. A method entity that is never called during runtime is of peculiarity `description`. Such an entity is only used to describe the method that can be called at a certain call cite. Method entities that are inserted by the front end to describe special inheritance features of the source language have peculiarity `inherited`.

This information can be used for analyses or directly for optimizations.

3.2.4 Volatile Entities

The value of a volatile entity can be accessed and changed from outside the program. A local volatile entity can not be resolved to data flow edges. Further volatility has consequences for the optimization of *Load* and *Store* operations to that entity. Two subsequent *Load*s can produce different results, two subsequent *Store*s must be executed. A *Load* after a *Store* does not necessarily return the stored value.

A volatile entity is marked by a special flag in the entity.

libFIRM does not perform optimizations for volatile entities if they are accessed directly. If the address of a volatile entity is used indirectly, e.g., by first storing it to another variable, the indirect access is optimized.

```
volatile int a;     volatile int a;
int *b = &a;        int *b = &a;

x := *b;            x := *b;
x := *b;
```

Figure 3.5: Indirect reference to volatile entity.

A volatile entity does not restrict code motion, i.e., it can not be used to implement a lock. Transformation in both directions between the two program fragments in Figures 3.5 and 3.6 is legal on Firm.

```
volatile int a;          volatile int a;
B b = new B();           B b = new B();

a := 1;                  a := 1;
while (a) { };           x = b.f;
x = b.f;                 while (a) { };
```

Figure 3.6: Optimization and volatile entities. `b` is a pointer to an object of type `B`, `f` a field of `b`. The optimizer finds out that `b` never points to `a`. Therefore it can move the access to the field of `b` before the loop.

3.2.5 Atomic and Compound Entities

Often it is necessary to know whether an entity represents a value that can be handled directly by the target code or whether first parts of the entity must be selected. Atomic entities map directly to a Firm mode. On first sight these are entities of primitive and pointer types. Here the mode of the value of the entity is given. Further enumeration entities are atomic. These must be mapped to a mode at some point of the compilation. Finally method entities are atomic. They are represented by a pointer to the method.

Compound entities are entities of compound types.

We can define the compound graph. The nodes of this graph are types and entities. The edges are the member relations of compound types (including the array type – element entity relation) and the edges from entities to their types. This graph is acyclic. All its leaves are atomic types. See also 3.1.1.

3.2.6 Constant Entities

An entity primarily contains information about a piece of memory as allocation, visibility or method of access. It also specifies the possible content of this piece of memory: It specifies a type. Further it allows to specify the variability of the content. We can distinguish uninitialized, initialized, partially constant and constant entities. An uninitialized entity has random content after allocation. An initialized entity has the content given in this

entity after allocation. [2] A constant entity has the content given in this entity for its whole lifetime. A compound entity can be partially constant: only some of it's members are constant.

If an entity is constant it means that each run time instance of the entity has the same constant value. E.g., if the entity is a member of a class type every instance of the class has the same content associated with this member. Especially the member is never written. It obviously makes no sense to keep many constant members in an entity of a compound type – a compiler phase should move these fields to a constant part that is reachable from all instances of this type.

If an entity is initialized or constant we must represent the value of the entity. This is simple if the entity is atomic. We associate a Firm node that represents the value with the entity. For methods we do not explicitly represent the constant value: this is a symbolic constant referring to the entity – the final value can only be determined by the linker. Figure 3.7 illustrates this case.

```
const int a = 13;
```

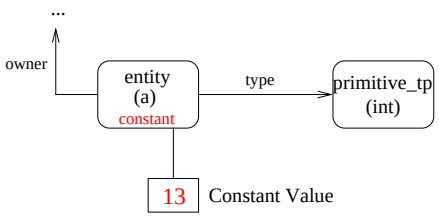


Figure 3.7: Simple constant entities.

For compound types that contain only atomic entities we could represent the value as a list of Firm nodes that correspond to the entities of the type. The correspondence can be established by the order within the list of values and the list of the members. But if the type is optimized the order of entities can change, entities can be removed or new ones added. When optimizing the type we do not want to change the constant value information, too. Anyways this information is not easy reachable. Therefore we represent the correspondence of values to entities explicitly. We keep two lists in

[2] It is planned to change the semantics of the *Alloc* node accordingly: If a graph is in state `phase_high` the *Alloc* initializes the memory according to the information in the type, if it is in state `phase_low` the initialization must be made explicit.

constant compound entities, one containing the values as Firm nodes, the other referencing the corresponding entities.

```
struct A {                    struct B {
   int a;                        int b;
   B s;                       }
}                             const A constA = (1, (7))
```

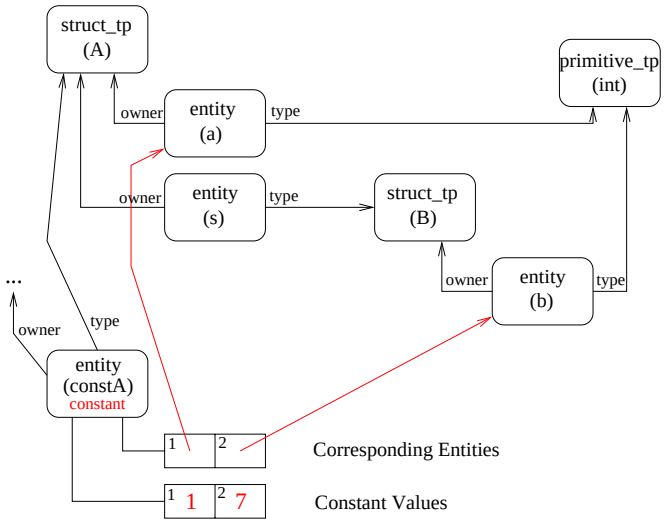


Figure 3.8: Compound constant entities.

It is more complicated if the entity is of a compound type that itself consists of members of compound types. As we can not represent the compound member of the type by a single value we need a nested representation. We need constant values for all leaves of the compound graph reachable from the type of the constant compound entity.

This representation is illustrated for a small example in Figure 3.8.

For constant arrays only one entity is available to associate the constant value with. A constant entity of type array contains the values as Firm nodes in the list of values and the element entity of the array in each corresponding position of the entity list. The number of entries in the list must match the array bounds (the first entry is lower_bound + 0). If the array has a dynamic bound the given bound must be matched, the other can be computed. Entities of type array can not be partially constant.

Figure 3.9 shows a constant array and nested compounds.

```
struct A {
     int a;
     int[0..1] b;
}
const A constA = (1, (7, 5))
```

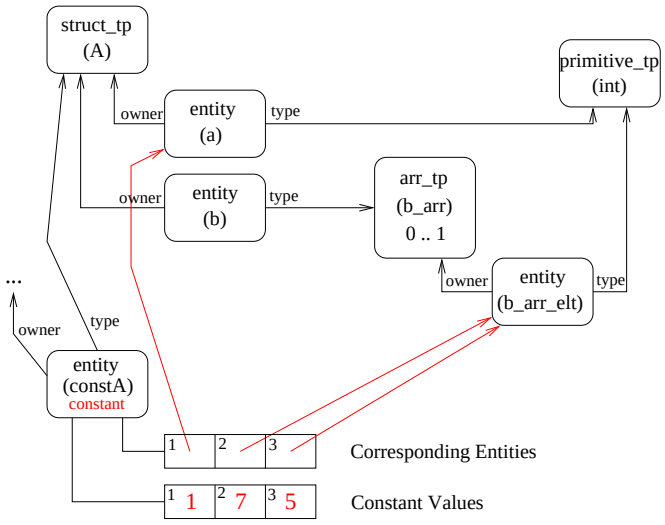


Figure 3.9: Constant array entities.

With this design constant entities refer to other entities to represent their constant content. When these other entities are optimized the content representation of the constant entity gets invalidated. libFIRM allows to update the content representation lazy. In this case entities removed completely must be turned into Id-entities. A later pass over the constant representation can compute the correct representation: It removes value and entity if the entity is an Id-entity or if the entity is no more reachable within the compound graph of the constant entity.

Constant values are always represented by Firm nodes. This can be arbitrary Firm graphs that do not use any control flow operations. These graphs can include *Call* nodes and *Load*s from other constant entities, but they can not have any side effects. These nodes must be allocated with the comfortable construction interface, `current_ir_graph` must be set to `get_const_code_irg()`.

3.2.7 Entities of Classes

Entities with a class type as owner can specify which entities of a superclass they overwrite. Further the inverse of this relation is represented explicitly. The entities referred to must be members of a (direct) superclass of the owner of the entity.

A method entity that is only represented to represent an inheritance that is not directly representable by the Firm semantics must be of peculiarity `inherited`. Such entities are necessary if, e.g., a class type has as super types classes of peculiarity `description` and `existent`. If the `existent` superclass contains an entity that implements one of the entities of the `description` superclass, an entity of peculiarity `inherited` must be inserted that overwrites both of these entities so that proper access is guaranteed. Without explicit representation of the `inherited` entity both of these entities would be inherited to the common subclass. See also the example in Figure 3.4.

3.2.8 Representation of Entities

To implement these features an entity has the following attributes:

type A type that describes the type of this entity.

owner A type that describes the owner of this entity.

name The name used for this entity in the source program. For entities that are not explicitly named the front end must come up with some name. This name should differ from the name of the type for this entity. The name is useful for debugging the compiler, for debug support and to implement functionality as reflection.

ld_name A unique name for the entity. This name can be set by the user of the library. If it is not set when it is first read it is automatically generated by concatenating the name of the owner with the name of the entity. This name is used to generate linking information, e.g., when outputting the value of a *SymConst* node.

allocation This attribute specifies whether the entity is allocated statically or dynamically. Possible values: `automatic_allocated`, `static_allocated` or `dynamic_allocated` (default).

visibility This attribute specifies the visibility of the entity. Possible values: `external_visible`, `external_allocated` or `local` (default).

variability The variability of the entity. Possible values: `uninitialized` (default), `initialized`, `part_constant` and `constant`.

`part_constant` is only possible for entities of class and structure types. Entities of type method are always `constant`.

volatility The volatility of the entity. Possible values: `non_volatile` (default) and `is_volatile`.

offset The offset to be added to a pointer to the owner to compute a pointer to this entity. This attribute is only set if the owner of the entity is in the state for fixed memory layout.

link A `void*` to associate some additional information with the entity.

visited This attribute can be used for traversing the entities. For a description how to traverse the entities see Chapter 8.

Attributes of entities of type method:

irg The `ir_graph` that represents the program code of the entity. Entities where irg is set must have a method type and are of `static_allocation`.

peculiarity The peculiarity of the entity. Is one of `existent`, `inherited` or `description`. If the entity is not of peculiarity `existent` the field `irg` is `NULL`.

Attributes of entities of types primitive, pointer and enumeration:

value A Firm node representing the value for this entity. Only available if the entity is `initialized` or `constant`.

Attributes of entities of types class, struct, union and array:

n_values The number of constant values associated with this entity. Only available if the entity is `initialized`, `part_constant` or `constant`.

value A list of constant values for the leave entities of the compound graph of this entities type. Only available if the entity is `initialized`, `part_constant` or `constant`.

member The leave entities of the compound graph of this entities type corresponding to the value list. Only available if the entity is `initialized`, `part_constant` or `constant`.

Attributes of entities of type class:

n_overwrites The number of entities overwritten by this entity.

overwrites A list of entities in super classes overwritten by this entity. If the class of this entity is treated as the superclass a *Sel* node will return this entity instead of the overwritten one. The entities overwritten by this one must be members of direct super classes of this entities owner.

n_overwrittenbys The number of entities that overwrite this entity.

overwrittenby The inverse of the overwrites relation.

3.3 Representation of Program Code

The representation of program code is organized hierarchically. At the top there is a data structure (`ir_prog`) representing a whole program. It contains references to all procedures and types as well as general information about the program. Below there is a data structure representing single procedures. The code of a procedure is represented as a graph of Firm nodes. The data structure for the procedure holds general information about the procedure and entry points to the graph. The nodes of the graph are the third level of the hierarchy. (There is no distinction in hierarchy between basic blocks and operations due to the structure of Firm.) Finally all nodes are typed with a mode.

3.3.1 The Program

A program is represented by several procedures, types and entities. libFIRM supplies a data structure (`ir_prog`) to merge all this information, and to hold further central information about the program. Functionality for the representation of a program is supplied in the header `irprog.h`. The data structure contains the following attributes:

program entry point The program entry point is the method that needs to be started by the operating system to execute the program. This method may not be of visibility `local`. If this field is `NULL` no explicit program entry point exists. This can be the case if the program is translated only partially.
Besides this method all methods that are `external_visible` can be called from outside the compiled code.

irg A list of all methods (ir graphs) compiled.

global type The global type needed as owner for global entities, see 3.2.

types A list of all types compiled. This list does not contain the `global_type` nor the frame types.

const code irg An ir graph that contains all constant expressions needed to represent array bounds and initializers.

The entities of a program can be found by inspecting the types. All static entities are found as members of the Global type.

The irprog data structure is allocated and initialized by the initialization function of the library.

3.3.2 A Method

The representation for a method (`ir_graph`) mainly serves as entry point to the graph of the method. It is specified in the header `irgraph.h` and has the following attributes:

entity The entity for the method.

frame type A class type containing local variables as members. It can also contain "inner" methods.

start node The start node of the Firm graph.

start block The block that contains the start node.

end node The end node of the Firm graph.

end block The block that contains the end block.

frame pointer A node that represents the stack frame pointer. This pointer is needed to model selection of local entities with the *Sel* node.

global pointer A node that represents the pointer to the global entity, see 3.2. This pointer is needed to model selection of global defined entities with the *Sel* node.

procedure arguments A *Proj* node that returns the tuple of all arguments.

current block This attribute is only relevant for the construction of ir. During construction this is the block to which new nodes are added automatically by the node constructors.

number of local variables This attribute is only relevant for the construction of ir. It contains the number of all local variables and the procedure parameters, see also 4.4.2. It is initialized by the constructor and can not be accessed further.

bad A reference to the unique *Bad* node in the graph. When building code use the constructor `new_Bad()` instead.

unknown A reference to the unique *Unknown* node in the graph. When building code use the constructor `new_Unknown()` instead.

link A `void*` to associate some additional information with the graph.

visited A flag for graph traversal.

block visited A flag for block wise graph traversal.

States of the Representation

The representation of a method can be in different states. The states reflect different phases during the compilation and the validity of additional analysis data. The states allow to control the interactions of different computations on the ir graphs.

State for Compiler Phase Three states represent different compiler phases: `phase_building`, `phase_high`, `phase_low`. During construction a graph is in `phase_building`. After the construction it is in `phase_high`. Here high level abstractions as *Sel* nodes are allowed. If all high level abstractions are removed the graph is in `phase_low`.

Other states are documented with the analysis/transformations establishing the state.

3.3.3 The Code of a Method

The code of a Method is represented by a Firm graph. All nodes of this graph are represented by a single data type specified in the header `irnode.h`. The different Firm operations are distinguished by an opcode of the node and a mode. The data type for a node has the following attributes:

firm kind A tag specifying that this is a Firm node. This is useful for dynamically checking the type of a node.

opcode The opcode of the node. There are routines to get the opcode but also to access the attributes of the opcode directly.

mode The mode of the node. There are routines to get the mode but also to access the attributes of modes directly.

arity The number of predecessors in the Firm graph not counting the block predecessor.

in A list with the predecessors in the Firm graph. There is functionality to access the predecessors individually depending on the opcode of the node and to iterate over all predecessors.

node nr A unique number for the node. Available only if compiled for debugging. (Configure with flag –enable-debug).

visited A flag for node traversal.

link A `void*` to add some information to the node.

3.3.4 An Operation in the Code

An Operation is identified by an opcode. Opcodes are defined in header `irop.h`. The Opcode holds all generic information about an operation. Operations can have specific attributes which are stored in the nodes. An opcode has the following attributes:

name A name for the opcode represented as an `ident`. To be used in development and debugging.

opcode An enum identifying the opcode. To be used for switching.

pinned A flag indicating that the node always must be pinned, i.e., the node may not be move to another block. Has one of the values `pinned` or `floats`.

For the structure of a Firm graph and the semantics of the nodes see [TLB99]. This report also lists the possible predecessors stored in the `in` list specified above and the result of the node.

The following nodes have specific attributes:

The *Block* Node

block visited A flag to traverse the control flow graph.

cg_cfgpred Predecessors in inter procedural graph. This list is automatically accessed by all access functions if `interprocedural_view` is set.

The *Start* Node

To project individual values from the result tuple of the start node the following enumerators exist:

pns_initial_exec The initial control flow.

pns_global_store The initial global memory.

pns_frame_base The pointer to the base of the procedures stack frame.

pns_globals The pointer to the global memory.

pns_args A tuple containing all arguments of the procedure.

The *EndReg* and *EndExcept* Node

irg The ir graph this node belongs to. This is needed to set current_ir_graph correctly when iterating over an interprocedural graph.

The *Cond* Node

We distinguish three kinds of *Cond* nodes. These can be distinguished by the mode of the selector operand and an internal flag **kind** of type `cond_kind`. First we distinguish binary *Cond*s and switch *Cond*s. A binary *Cond* has as selector a boolean value. *Proj*(0) projects the control flow for case "False", *Proj*(1) the control flow for "True". A binary *Cond* is recognized by the boolean selector. The switch *Cond* has as selector an unsigned integer. It produces as result an n+1 Tuple (cf0, ... , cfn) of control flows.

We differ two flavors of this *Cond*. The first, the dense *Cond*, passes control along output i if the selector value is i, $0 \leq i \leq n$. If the selector value is >n it passes control along output n.

The second *Cond* flavor differs in the treatment of cases not specified in the source program. It magically knows about the existence of *Proj* nodes. It only passes control along output i, $0 \leq i \leq n$, if a node *Proj*(*Cond*, i) exists. Else it passes control along output n (even if this *Proj* does not exist.) This *Cond* we call "fragmentary". There is a special constructor `new_defaultProj()` that automatically sets the flavor. Default flavor is "dense". Flavor "fragmentary" is experimental.

Figure 3.10 shows a dense *Cond* node. The front end constructing this node must determine that the switch chooses the default block for value 1 to add the corresponding *Proj* node.

```
switch (x)
  case 0:  ...break;
  case 2, 3:  ...break;
  default:  break;
end switch
```

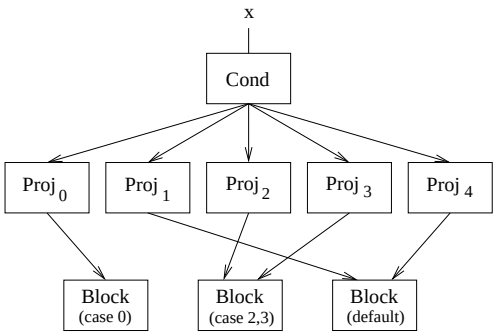


Figure 3.10: A dense *Cond* node.

The *Const* Node

con The target value (see 3.4.1) representing the constant.

The *SymConst* Node

symconst kind A tag specifying the kind of the *SymConst*. Possible values are `type_tag`, `size` or `linkage_ptr_info`.

type This attribute only exists if the *SymConst* is a `type_tag` or a `size`. It contains the type it corresponds to.

ptrinfo This attribute only exists if the *SymConst* is a `linkage_ptr_info`. It contains an `ident` representing a string to be used for linking.

The *Sel* Node

ent Contains the entity to select from the argument. The argument is a pointer to an entity of the owner type of **ent (or of a subtype of this type).**

The *Call* Node

type The type of the called function.

callee A list of all functions that can be called from this node. This list must be computed by an analysis.

The *CallBegin* Node

irg The ir graph this node belongs to. This is needed to set current_ir_graph correctly when iterating over an interprocedural graph.

call The call node visible in the intra procedural view.

The *Filter* Node

proj The projection number for the Filter node in the intra procedural view.

cg_pred Interprocedural predecessors of this node. This list is automatically accessed by all access functions if `interprocedural_view` is set.

The *Alloc* Node

type The type of which to allocate an entity.

where A flag indicating where to allocate an entity: on the stack or in the heap. Default value: `heap_alloc`.

The *Free* Node

type The type of the freed entity.

The *Proj* Node

proj The position of the projected value in the Tuple. (First position is '0'.)

3.3.5 The Mode of an Operation

The modes are defined in `irmode.h`. A mode represents a basic data type that can be represented in the target language, or, that can be handled by the target processor. For further details about modes see [TLB99].

A mode has the following attributes:

modecode An enum identifying the opcode. To be used for switching.

name A name for the mode represented as an `ident`. To be used in development and debugging.

size The size of values of this mode in bytes.

ld_align The alignment of values of this mode in bytes.

min The smallest representable value of this mode represented as a target value.

max The biggest representable value of this mode represented as a target value.

null The zero value of this mode represented as a target value.

signed Indicates whether the mode represents signed values.

float Indicates whether the mode represents floating point values.

Most of these attributes are useful for constant evaluation. The representation of modes supplies a set of functions that test whether a mode is in a certain subset of all modes according to Table 2.2 in [TLB99]. This allows to test legal modes for the operands of nodes. Further a method `smaller_mode()` tests whether a mode can be converted to an other with the *Conv* node, see Section 2.2.2.5 in [TLB99].

The data structures representing the modes defined in Section 2.2.1 in [TLB99] are constructed by the libFIRM initialization. They can be accessed by global variables.

3.4 Other

To represent a program completely a representation for constants (target values) and for identifiers is needed.

3.4.1 Target Values

A target value (`tv.h`) represents values of Firm modes. It does not represent values of libFIRM types. I.e., a target value can represent a 32 bit integer value, but not a string constant or a dispatch table. There are no target values for special modes as memory, execution or auxiliary modes (M, BB, X, T).

Numerical target values are represented by their bit pattern. Pointer target values can be represented in two ways: By a bit pattern of the pointer value or by referencing an entity. A target value referencing an entity represents the address of this entity. The entity must be `static_allocated`.

The target value module supplies global constants containing specific values including a 'bad' value representing an undefined target value. Further it supplies a set of constructors and access routines.

The target module also implements the evaluation of constant expressions. It supplies routines to execute most arithmetic operations of Firm nodes on two target values. The implementation of this evaluation is independent of the platform compiled on and can be initialized for arbitrary target architectures.

3.4.2 Identifiers

An identifier represents a source language name. Identifiers are implemented in `ident.h`. The module supplies the following routines:

`id_from_str()` The constructor.

`id_to_str()` Returns a pointer to a string that is **not** null terminated.

`id_to_strlen()` Returns the length of the represented string.

`id_is_prefix()` Returns true if the first argument is a prefix of the second.

`id_is_suffix()` Returns true if the first argument is a suffix of the second.

`print_id()` Prints an `ident` to `stdout`.

`fprint_id()` Prints an `ident` to the file passed.

Chapter 4

Connecting to a Front End

This chapter explains how to connect the Firm library to a front end. We assume that the front end produced an abstract syntax tree (AST) that represents a correct program. Further we assume the existence of a definition table. First we introduce the general approach and then discuss individual phases of the construction of Firm code in detail. At some points there are references to small test programs included in the Firm library that build an illustrative Firm graph.

4.1 General Approach

This section describes the steps a compiler has to take to construct intermediate representation with the Firm library in execution order.

- Initialize the Firm library (`init_firm()` in `firm.h`)
- Set optimization flags (See functions in `irflag.h`, skip to use defaults).
- Construct type information.
  - Before a type can be used (i.e., for *Sel* nodes), it must exist. Although it may be possible to create types on-demand, it can be more convenient and simpler to ensure that all types exist before starting Firm construction.
  - Associate Firm types with the corresponding type entry in the definition table. In general, whenever a type identifier is used in the source program, the construction needs the corresponding Firm type.
  - Firm types must be constructed for all (used) language defined types that might have no explicit representation in the AST.

  - Firm types must be constructed for all (used) methods.
  - With compound types also the entities for the members of the compound types (methods or fields) can be constructed. Alternatively this can be done on demand.

- Construct entities for global variables.
  - Firm globals should be associated with the corresponding entry in the definition table.
- Construct procedures.
  - Construct the general data structure for a procedure.
    * Count the number of local variables of this procedure. Here procedure parameters are considered to be local variables. (This number needs not to include possibly aliased variables.) Associate each local variable with a unique number ($0 \leq$ number $<$ number of local variables)
    * Generate an entity for the procedure (if not already done). This requires the type of the procedure and the type of the owner. In an object oriented language the owner is the class (or object) the method belongs to. If the programming languages provides no owner of the procedure use the global type (`get_glob_type()`).
    * Call the constructor for the procedure (`new_ir_graph()` in `irgraph.h`).
    * Perform a `set_value()` for all procedure parameters. (See 4.4.5 and 4.4.2.)
  - Decide which local variables can be resolved to data flow edges. These must be alias free variables. Generate entities as members of the stack frame type for all other variables.
  - Construct the program code for the procedure.
  - Call intra procedural optimizations.
  - libFIRM is designed to construct several procedures at a time.
- Perform optimizations
- Code generation.

4.2 Constructing Types

The constructors for types require some general information about the types. A major part of information about compound types are the members that

are specified for them as well as sub and super types for classes. These need not be supplied with the constructors or right after construction of the type, they can be added on demand. These members (including sub and super types) are held in dynamic arrays, so new members need to be added by a special add routine. All these arrays supply four methods to access them with the following functionality:

- Get the length of the array.
- Add the passed element to the array.
- Get an element of a given position. The position must be valid, i.e. 0 $\leq$ position $<$ length.
- Set an element at a given position. The position must be valid, i.e. 0 $\leq$ position $<$ length.

The constructor for an entity automatically adds the entity to the proper member list.

For the information needed to built a specific type representation see the documentation of the type data structures in 3.1 and the documentation in `type.h`.

4.3 Constructing Entities

The constructor for an entity requires the type of the entity, the owner of the entity and a name of the entity.

The name is any string stored as Firm identifier. It is useful to use variable or field names from the source program.

The type of an entity is either the type of a local or global variable or the type of a member of a compound type.

The owner of an entity that is a member of a compound type is the compound type. The owner of entities for global variables is a special type called "GlobalType". This type is automatically generated with the initialization of the library an can be obtained with a call to `get_glob_type()` defined in `irprog.h`. The entity constructor automatically adds the entity to the owners list of members.

4.3.1 Construction of Constant Entities

First construct an entity of the proper type with the global type as owner. Then set the intended variability with `set_entity_variability()`. Set `current_ir_graph` to `get_const_code_irg()` and build the expressions

specifying the constant values with the comfortable construction interface. These expressions may not contain control flow operations, *Call* nodes are possible. The methods called should not have any side effects, though.

4.4 Constructing Firm Graphs

4.4.1 Support by libFIRM

This library supplies several interfaces to construct a Firm graph for a method, where each is built on top of the other.

- A "comfortable" interface generating SSA automatically. Automatically computed predecessors of nodes need not be specified in the constructors. (`new_`<Node>`()` constructors and a set of additional routines.)
- A less comfortable interface where all predecessors except the block an operation belongs to need to be specified. SSA must be constructed by hand. (`new_`<Node>`()` constructors and `switch_block()`). This interface is called "block oriented".
- An even less comfortable interface where the block needs to be specified explicitly. This is called the "raw" interface. (`new_r_`<Node>`()` constructors). It automatically calls the local optimizations for each new node.

For all three interfaces exist two implementations, one considering debug information, the other not.

To use the functionality of the comfortable interface correctly the front end needs to follow certain protocols. This is explained in the following. To build a correct ir with the other interfaces study the semantics of the firm nodes (See tech-report UKA 1999-14). In the following we explain the comfortable interface.

4.4.2 The Comfortable Interface

The comfortable interface contains the following methods:

```
ir_node *new_immBlock(int arity, ir_node **in);
ir_node *new_Jmp    (void);
ir_node *new_Cond   (ir_node *c);
ir_node *new_Return (ir_node *store, int arity, ir_node **in);
```

```
ir_node *new_Raise  (ir_node *store, ir_node *obj);
ir_node *new_Const  (ir_mode *mode, tarval *con);
ir_node *new_SymConst (type_or_id_p value, symconst_kind kind);
ir_node *new_simpleSel(ir_node *store, ir_node *objptr, entity *ent);
ir_node *new_Sel    (ir_node *store, ir_node *objptr, int arity,
                     ir_node **in, entity *ent);
ir_node *new_Call   (ir_node *store, ir_node *callee, int arity,
                     ir_node **in, type *type);
ir_node *new_Add    (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Sub    (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Minus  (ir_node *op,  ir_mode *mode);
ir_node *new_Mul    (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Quot   (ir_node *memop, ir_node *op1, ir_node *op2);
ir_node *new_DivMod (ir_node *memop, ir_node *op1, ir_node *op2);
ir_node *new_Div    (ir_node *memop, ir_node *op1, ir_node *op2);
ir_node *new_Mod    (ir_node *memop, ir_node *op1, ir_node *op2);
ir_node *new_Abs    (ir_node *op,                ir_mode *mode);
ir_node *new_And    (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Or     (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Eor    (ir_node *op1, ir_node *op2, ir_mode *mode);
ir_node *new_Not    (ir_node *op,                ir_mode *mode);
ir_node *new_Shl    (ir_node *op,  ir_node *k,   ir_mode *mode);
ir_node *new_Shr    (ir_node *op,  ir_node *k,   ir_mode *mode);
ir_node *new_Shrs   (ir_node *op,  ir_node *k,   ir_mode *mode);
ir_node *new_Rot    (ir_node *op,  ir_node *k,   ir_mode *mode);
ir_node *new_Cmp    (ir_node *op1, ir_node *op2);
ir_node *new_Conv   (ir_node *op, ir_mode *mode);
ir_node *new_Phi    (int arity, ir_node **in, ir_mode *mode);
ir_node *new_Load   (ir_node *store, ir_node *addr);
ir_node *new_Store  (ir_node *store, ir_node *addr, ir_node *val);
ir_node *new_Alloc  (ir_node *store, ir_node *size,
                     type *alloc_type, where_alloc where);
ir_node *new_Free   (ir_node *store, ir_node *ptr, ir_node *size,
                     type *free_type);
ir_node *new_Sync   (int arity, ir_node **in);
ir_node *new_Proj   (ir_node *arg, ir_mode *mode, long proj);

void add_in_edge    (ir_node *block, ir_node *jmp);
void mature_block   (ir_node *block);
void switch_block   (ir_node *target);
```

```
ir_node *get_value  (int pos, ir_mode *mode);
void set_value      (int pos, ir_node *value);
ir_node *get_store  (void);
void set_store      (ir_node *store);
void keep_alive     (ir_node *ka);
```

Further it includes the global variable `current_ir_graph`.

The methods of the interface can be separated into three categories: constructors for nodes, support for control flow construction and support for SSA construction. All node constructors add the node to the graph in `current_ir_graph`.

Control Flow

Several basic blocks can be constructed in parallel, but the code within each block needs to be constructed (almost) in program order.

The field `current_block` of `current_ir_graph` (See also 3.3.2) holds the current basic block. All non block nodes generated are added to this block. The current block can be set with `switch_block(`<block node>`)`. If several blocks are constructed in parallel, block switches need to be performed constantly. The block constructor automatically sets `current_block` to the new block it creates.

To generate a Block node (with the comfortable interface) it's predecessor control flow nodes need not be known. With `add_in_edge(block, cfnode)` predecessors can be added to the block. If all predecessors are added to the block `mature_block(block)` needs to be called. Calling `mature_block()` early improves the efficiency of the *Phi* node construction algorithm. But if several blocks are constructed at once, `mature_block()` must only be called after performing all `set_value()`s and `get_store()`s in the block. (See also documentation of `new_immBlock()` constructor in `ircons.h`.)

From the *End* node all useful code that is constructed must be reachable. Endless loops are not reachable by standard control flow edges from *End*. Therefore one block in each endless loop must be added to the *End* node as direct predecessor to be kept alive explicitly. Further all *Phi* nodes of mode memory in endless loops must be kept alive.

The Firm construction interface automatically adds the proper nodes as operands of *End* if endless loops result from optimization of *Cond* nodes. It also adds all memory *Phi* operations. If the source language allows loops

by explicit Goto operation the construction must add the concerned blocks itself with `keepalive()`.

Support for SSA Construction

SSA construction is performed automatically by the interface. The construction algorithm must only tell the interface when a variable is defined and when it is used. This is performed by the two methods `get_value()` and `set_value()`. The same holds for the memory value for which the methods `get_store()` and `set_store()` are available.

A call to `get_value(`<number of local value>`)` returns the valid value of a variable, i.e., the node that last defined the variable. The number used as argument is a unique identifier for the variable and needs to be administered by the front end. If an assignment assigns to a local variable the value assigned needs to be passed to the library by `set_value(`<node>`,` <number of local value>`)`. In straight line code these two operations just remember and return the pointer to nodes producing the value. If the value passes block boundaries *Phi* nodes can be inserted. The call to `get_value()` also triggers the generation of *Phi* nodes.

4.4.3 Procedure Initialization

First the front end needs to decide which variables and values used in a procedure can be represented by data flow edges. These are variables that need not be saved to memory as they cause no side effects visible out of the procedure. Often these are all compiler generated variables, the local variables of the procedure and the procedure parameters. The front end has to count and number these variables. It has to decide which of these can not be represented by data flow edges – in general all those that are dereferenced. For all variables it can not represent as data flow edges it must build entities. These entities are members of the stack frame. A call to `get_irg_frame_type()` returns the type modeling the stack frame for a procedure.

The construction algorithm internally keeps an array for each basic block containing references to the valid value of a variable. The size of this field is derived from the number of local variables passed to the constructor `new_ir_graph()`. The numbers associated with the local variables are used to reference the values in this array. Therefore they must range from zero to the total number of local variables in this procedure. In order to optimize

memory consumption it is possible to skip aliased variables in this numbering, but the effect is negligible as the memory is freed after the construction.

Now the basic data structure for the Firm graph can be constructed with a call to `new_ir_graph()`. The constructor gets the number of local (alias free) variables including parameters as well as the entity representing the procedure. The graph is held in the global variable `current_ir_graph`.

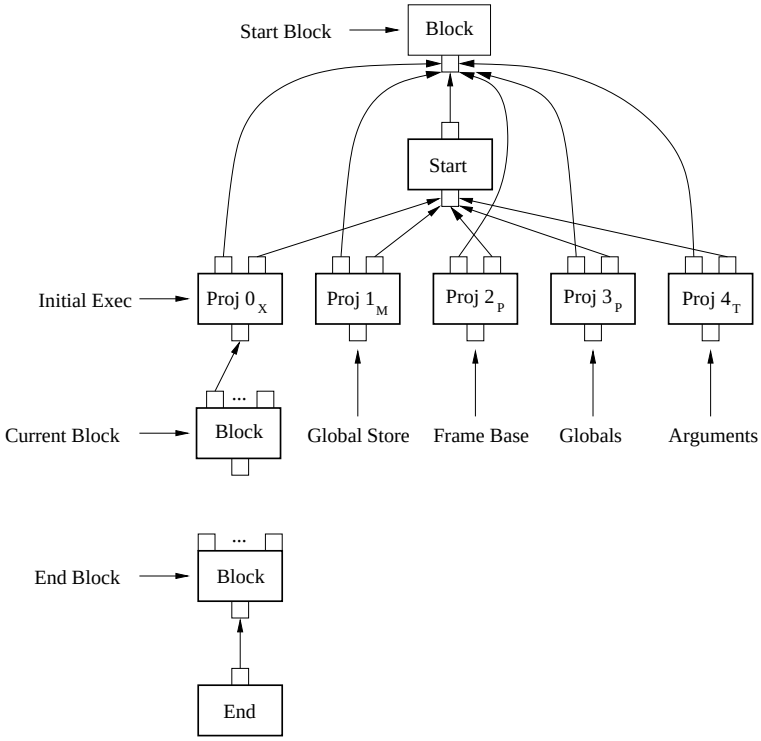


Figure 4.1: Initial graph built by `new_ir_graph()`.

This constructor creates a set of nodes to start with, see figure 4.1. The *Proj* nodes are remembered in the graph data structure, see `irgraph.h`. The start block is mature, the current block and the end block must be matured by the ir construction when all predecessors are known, see 4.4.2.

Here it is convenient to make the procedure parameters visible as known values: Generate *Proj* nodes for these and make them known to the construction algorithm by calling `set_value()` (See 4.4.5 and 4.4.2):

`set_value(new_Proj(get_irg_args(current_ir_graph),` <Proper Mode>`,` <Number of variable>`))`.

4.4.4 Constructing a Simple Node

The algorithm to construct a node must perform three steps: First it must obtain the nodes for the operands in some way. Then it can construct the node for the operation. Finally it must deal with the results of the node. In the following we describe the general approach for expression trees. Statements are considered later.

Operands of a Node

There are three ways to retrieve the nodes needed as operands. If the operands result from constructing Firm for sons of the current AST node the attributed grammar or recursive descend passes these nodes as attributes or results to the node that needs them as operand.

This is not possible for leaves in the AST. In general, leaves of expression subtrees are variables or constants. For constants the algorithm needs to construct a new Firm *Const* node. If the same constant is used in several places the AST traversal can generate a new *Const* every time, the constructor of the node will automatically reuse an already existing equivalent node (as a result of common subexpression elimination). For variables there are two cases depending on whether they are known to be alias free.

Alias Free Variables The initialization of the construction of the current Firm graph identified all alias free variables with an unique number, see 4.4.3. These variables are resolved to data flow edges, i.e., the loading and storing of values in these variables on the stack is not modeled by explicit *Load* or *Store* operations. As operands the AST traversal just uses the node that last defined this variable. This node can be retrieved from the Firm library by a call to `get_value(`<Number of variable>`)`.

Other Values If the variable is a local variable that is not known to be alias free or if it is a global variable the AST traversal must create a *Load* operation. The load operation needs as input a pointer to the loaded variable. This pointer is retrieved with a *Sel* node that operates either on a pointer to the data segment that holds the global variables or on the stack frame pointer. Both these pointers are supplied by the *Start* node. The *Proj* nodes that project them out of the result tuple of the *Start* are stored in the central `ir_graph` data structure and can be obtained by `get_irg_globals()` and `get_irg_frame()`. Further for the *Sel* and *Load* nodes it is necessary to have the entity for the variable available. The AST traversal must supply

a mechanism how to access the entity it created before, e.g., by storing a reference to it in the definition table.

Constructing a Node

To construct a node the AST traversal simply must call the constructor of this node. There are three kinds of parameters of these constructors: Other nodes that are operands of the operation associated with the node, the mode and attributes.

Node Parameters Which operands are needed for a node is defined by the syntax of Firm nodes, see [TLB99, Table 3.1] (Print out this table as a quick guide). Not all of these operands need to be specified as parameters of the constructor.

Some nodes require a list of operands as, e.g., the *Call* node. The AST traversal should assemble this list in a local array and pass it along with the array length in the constructor. The constructor will copy the content of this array.

The Mode Some constructors require that the mode of the operation is mentioned. Whenever the constructor can derive this mode by itself it is omitted. The AST traversal should be able to derive the necessary mode from the type associated with the AST node. This type should allow to navigate to a firm type which then again contains the information to which mode the type is mapped. Modes should be only necessary for AST nodes that are typed with an atomic type that can be mapped to a mode.

Node Attributes Certain nodes need specific attributes. For the purpose of these attributes see the documentation of the nodes ([TLB99]) and 3.3.4.

Results of a Node

For the results of nodes see [TLB99, Table 3.1]. The standard nodes have only one result: the value produced by the node. In an expression tree this value is either passed to the father AST node by an attribute of the attributed grammar or it is returned by the recursive ascend function.

If [TLB99, Table 3.1] lists several operands for a node the actual implementation of the node returns a tuple containing these operands. The construction algorithm must project each of these operands from the node by *Proj* nodes. E.g., to deal with a *Div* node it is necessary to project

the memory result M and call `set_store()` for this *Proj* node. If exception control flow is modeled explicitly the X result must be projected and inserted as predecessor of the exception handler block. Finally the result value is projected and handled as a common value result, e.g., returned by the recursive descend.

The result of the *Call* node contains nested Tuples, i.e., two levels of *Proj* nodes are needed to extract a result.

As a consequence the mode of the *Div* node is tuple. The *Proj* nodes must carry the proper modes for the tuple elements.

Some Special Situations

Memory Operands For the AST traversal there exists only a single value for the Memory. This single value is treated as an alias free local variable: Whenever a Memory operand is needed this can be obtained by a call to `get_store()`. Every Memory result must be announced to the firm library by a call to `set_store(`<node defining Memory>`)`. In general the node defining the memory is a *Proj* node (except for *Free* and *Sync*).

Field Access For field access a *Load* node is necessary. The pointer for this *Load* is the result of a *Sel* node. The *Sel* needs as operands the pointer to the compound containing the field and an entity describing the field. Union members are treated as field accesses. Use the constructor `new_simpleSel()`. See test program `oo_program_example.c`.

Array Access For array access also a *Load* node is necessary. The pointer for this *Load* is the result of a *Sel* node. The *Sel* needs as operands the pointer to the array and an entity describing the arrays elements. Further it needs a list of Firm nodes that compute the array indexes. The computation of the element address is hidden within the *Sel* node. The constructor `new_Sel()` is provided for this purpose. See test program `array_heap_example.c`.

Method Access For method access also a *Sel* node is necessary. This node hides any method access functionality required by the source language, as, e.g., method dispatch. The pointer passed to this method points to the compound whose type specifies the method.

If the method does not belong to a certain type (class) it is a member of the global type and the call is unique. The address of the method can be represented by a *Const* node containing the entity as target value. It is also

possible to select the method address from the the only entity of the global type to which a pointer is available from the start node. The *Proj* node for this pointer can be obtained with `get_irg_globals()`. See test program `oo_program_example.c` for method calls.

4.4.5 Statements

Assignments

Here again we must distinguish the assignment to an alias free local variable and other assignments.

For assignment to an alias free local it is sufficient to call `set_value()` with the node computing the assigned value as parameter. Assignments to all other variables are performed with a *Store* node. The pointer for the *Store* comes from a *Sel* which is built the same way as for loading from variables, see 4.4.4. An assignment to a field or an array is also performed with a *Store* operation. 4.4.4.

Statements Specifying Control Flow

Generating Firm from statements specifying control flow requires a deliberate design. It is not possible to generate and mature the blocks necessary for the statement in the computation for a single AST node. The descend from the AST node for the statement can generate further blocks nested within the blocks necessary to implement this statement.

We propose the following approach to generate control flow for statements. We illustrate the approach with the computations necessary for a while statement with break statements.

Before
`while (` *Condition* `) {`
    Body with break statements
}
After

Figure 4.2: A while statement.

All computations for AST nodes should assure that whenever a recursive descend is performed `current_block` is set to the block that shall contain the code generated by the descend. `Current_block` may not be mature. Further they must assure that they return with `current_block` set to the

immature block that shall take further code. This guarantees that, whenever a descend returns, `current_block` contains a block that can take the place of the block `current_block` was set to when the descend started. Alternatively it is possible to pass the current block as an argument.

In the example a computation above the AST node for the while generates the code for *Before* and *After*. The computation for the while must generate the control flow for the loop. It descends recursive to generate the code for the *Condition* and the loop *Body*. The computation must prepare a data structure to collect the control flow of the break statements.

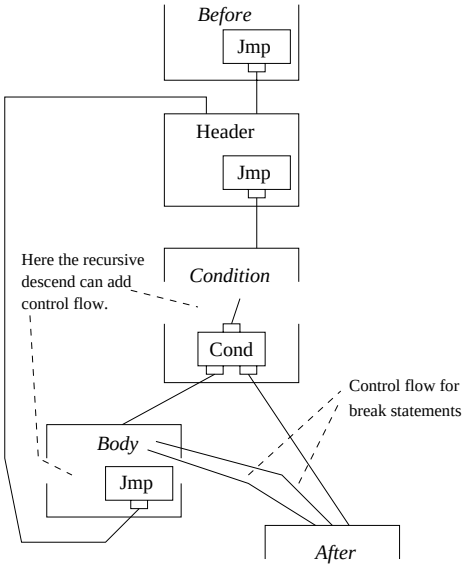


Figure 4.3: Intended control flow for while statement.

First it is necessary to design the control flow required for the statement. It is important that blocks that are current in a descend have all their predecessors before the descend is started. The descend can mature the block making it impossible to add further control flow edges.

Here it is important to separate the *Header* block from the block for the *Condition* code. The recursive descend for the condition can mature the block passed. In this case it is later impossible to add the control flow edge closing the loop. This edge can not be added beforehand as the last block of the *Body* can not be known in advance.

```
ir_node *x;
ir_node *before_bl = get_irg_current_block(current_ir_graph);
x = new_Jmp();
mature_block(before_bl);

ir_node *header_bl = new_immBlock();
add_in_edge(header_bl, x);
x = new_Jmp();

ir_node *condition_bl = new_immBlock();
add_in_edge(condition_bl, x);
ir_node *bool = EXPR_DESCEND(condition);
condition_bl = get_irg_current_block(current_ir_graph);
x = new_Cond(bool);
mature_block(condition_bl);

ir_node *after_bl = new_immBlock();
add_in_edge(after_bl, new_Proj(x, FALSE));
ir_node *rem_break = break_bl;
break_bl = after_bl;

ir_node *body_bl = new_immBlock();
add_in_edge(body_bl, new_Proj(x, TRUE));
STMT_DESCEND(body);
body_bl = get_irg_current_block(current_ir_graph);
x = new_Jmp();
mature_block(body_bl);

add_in_edge(header, x);
mature_block(header_bl);
break_bl = rem_break;
switch_block(after_bl);
```

Figure 4.4: Implementation for while statement. The code assumes a global variable `ir_node *break_bl`.

The implementation for this AST node can allocate all blocks in advance and remember them in local variables. It can add code to these blocks and mature them. At the end it must set `current_block` dedicated for further code.

First the current block (*Before*) is finished by adding a *Jmp* node and maturing the block. The *Header* block is allocated and fixed as target of the *Jmp*. We need to remember the header block to add the loop edge later (and to add edges of eventual continue statements). Now we allocate the *Condition* block and add the single control edge from the *Header* block.

These two blocks will eventually be merged by an optimization. We add the condition code to the condition block by recursive descend. As the condition is an expression the descend should return the node that represents the result of the condition. The descend can add further blocks. Therefore we need to reload `condition_bl` from the current block. We add the conditional jump *Cond* and generate the *After* and *Body* blocks. The *After* block is remembered in a global variable that is used by the break statement to insert the control flow edges. Actually we need a stack of break targets in case of nested loops but here we can use the call stack by preserving the state of the global variable and later restoring it.

Then the descend for the *Body* block is performed. This descend can add edges to the *After* block. When the descend returns we must reload `body_bl` from the current block. We add the loop control edge and mature the *Body* block. Finally we must restore the old value of the global variable and set *After* block as current so that further code can be added to it.

```
add_in_edge(break_bl, new_Jmp())
mature_block(get_irg_current_block(current_ir_graph));
new_immBlock();
```

Figure 4.5: Implementation for break statement.

The implementation for the break statement closes the current block and adds a *Jmp* node to jump to the break block. Then it allocates a new block. This block is unreachable in the program. Nevertheless it is necessary to allocate another block to fulfill the above agreement to leave every computation with an immature current block.

See test programs `if_example.c`, `if_else_example.c`, `if_while_example.c` and `while_example.c` for further examples.

Chapter 5

The Interprocedural Representation

Firm defines an interprocedural data flow representation. In this representation explicit control flow edges connect all call sites with the entry and exit points of possibly called methods. Further all arguments passed to *Call* nodes are connected to the use of arguments in the called methods. If necessary the values are merged by *Phi* nodes. A comparable representation is built for the results of methods. See also [Sch02].

A call to `cg_construct()` (`ircgcons.h`) builds the interprocedural representation. It expects that the attribute callee of all *Call* nodes contains a list of all method entities that can be called by this node. This information must be collected beforehand. libFIRM supplies a simple analysis to compute the callees, see 7.4.

The method `cg_destruct()` removes the interprocedural view.

When the interprocedural representation is constructed it is possible to use the representation of the program in two views: the interprocedural view or the intra procedural view. The flag `interprocedural_view` (see `irgraph.h`) states which view to use. Iterators and access functions as `irg_walk()` or `get_irn_n()` will act according to the view flag.

Chapter 6

Optimizations and Transformations

All optimizations can be turned off by flag `optimize`.

6.1 Constant Evaluation, Algebraic Simplification and Others

A large set of transformations implementing various kinds of optimizations as constant evaluation or algebraic simplification are performed by `local_optimize_graph()`. These optimizations are controlled by flag `opt_constant_folding` .

The terms following below summarize the performed transformations. Firm nodes are represented as a function named with the opcode operating on its operands. Nesting these functions and using variables for nodes describes patterns of in the Firm graph. Attributes to the Firm nodes are represented as subscripts. If necessary, nodes are distinguished by superscripts.

The variables X, Y, V and M represent nodes that are valid, but are not further checked for opcode or attributes; nodes denoted by M can be understood to represent a memory value, while V can be understood to represent a real value (not a synthetic value like memory or execution).

The variables tv_1, tv_2 etc. denote target values; operations specified on such values must be executed with the semantics of the respective operation on the target platform.

Further conditions are expressed by pseudo-logical clauses. The left pattern is replaced by the right one. Constant evaluations are only performed if

the target value module could compute the result (i.e., no exception would occur).

$$
\begin{aligned}
SymConst_{type}() &\Rightarrow Const_{tv(get\_type\_size(type))}() \\
\wedge\ layout\_fixed(type) & \\
Add(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1+tv2}() \\
\wedge\ mode(tv1) = mode(tv2) \neq mode\_p & \\
Sub(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1-tv2}() \\
\wedge\ mode(tv1) = mode(tv2) \neq mode\_p & \\
Sub(X, X) &\Rightarrow Const_{tarval\_null(mode(Sub))}() \\
Minus(Const_{tv}()) &\Rightarrow Const_{-tv}() \\
Mul(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1*tv2}() \\
Mul(Const_0(), X) &\Rightarrow Const_0() \\
Mul(X, Const_0()) &\Rightarrow Const_0() \\
Quot(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1quottv2}() \\
\wedge\ tv2 \neq 0 & \\
Div(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1divtv2}() \\
\wedge\ tv2 \neq 0 & \\
Mod(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1modtv2}() \\
\wedge\ tv2 \neq 0 & \\
Abs(Const_{tv1}()) &\Rightarrow Const_{|tv1|}() \\
And(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1\wedge tv2}() \\
And(Const_0(), X) &\Rightarrow Const_0() \\
And(X, Const_0()) &\Rightarrow Const_0() \\
Or(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1\vee tv2}() \\
Or(Const_1(), X) &\Rightarrow Const_1() \\
Or(X, Const_1()) &\Rightarrow Const_1() \\
Eor(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1eortv2}() \\
Not(Const_{tv}()) &\Rightarrow Const_{\neg tv}() \\
Shl(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1shltv2}() \\
Shr(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1shrtv2}() \\
Shrs(Const_{tv1}(), Const_{tv2}()) &\Rightarrow Const_{tv1shrstv2}() \\
Conv(Const_{tv}()) &\Rightarrow Const_{tarval\_convert(tv,mode(Conv))}()
\end{aligned}
$$

$$
\begin{aligned}
Proj_n(Cmp(X,X)) &\Rightarrow Const_{true}() \\
\wedge\ n \in \{Eq, Le, Ge, Leg, Ue, Ule, Uge, True,\} & \\
Proj_n(Cmp(X,X)) &\Rightarrow Const_{false}() \\
\wedge\ n \in \{False, Lt, Gt, Lb, Uo, Ul, Ug, Ne\} & \\
Proj_n(Cmp(Const_{tv1}(), Const_{tv2}())) &\Rightarrow Const_{tarval\_cmp(tv1,tv2,n)}() \\
Proj_n(Cmp(Proj(Alloc(...)), Const_{void})) &\Rightarrow Const_{true}() \\
\wedge\ mode(Proj) = mode(Const) = mode\_p & \\
\wedge\ n \notin \{False, Eq\} & \\
Proj_n(Cmp(Proj(Alloc(...)), Const_{void})) &\Rightarrow Const_{false}() \\
\wedge\ mode(Proj) = mode(Const) = mode\_p & \\
\wedge\ n \in \{False, Eq\} & \\
Proj_n(Cmp(Const_{void}, Proj(Alloc(...)))) &\Rightarrow Const_{true}() \\
\wedge\ mode(Proj) = mode(Const) = mode\_p & \\
\wedge\ n \notin \{False, Eq\} & \\
Proj_n(Cmp(Const_{void}, Proj(Alloc(...)))) &\Rightarrow Const_{false}() \\
\wedge\ mode(Proj) = mode(Const) = mode\_p & \\
\wedge\ n \in \{False, Eq\} & \\
Proj_n(Cmp(Proj_1(Alloc_1(...)), Proj_2(Alloc_2(...)))) &\Rightarrow Const_{true}() \\
\wedge\ mode(Proj_1) = mode(Proj_2) = mode\_p & \\
\wedge Alloc_1 \neq Alloc_2 & \\
\wedge\ n \notin \{False, Eq\} & \\
Proj_n(Cmp(Proj_1(Alloc_1(...)), Proj_2(Alloc_2(...)))) &\Rightarrow Const_{false}() \\
\wedge\ mode(Proj_1) = mode(Proj_2) = mode\_p & \\
\wedge Alloc_1 \neq Alloc_2 & \\
\wedge\ n \in \{False, Eq\} & \\
Proj_0(DivMod(Const_{tv1}(), Const_{tv2}())) &\Rightarrow Const_{tv1divtv2}() \\
\wedge\ tv2 \neq 0 & \\
Proj_1(DivMod(Const_{tv1}(), Const_{tv2}())) &\Rightarrow Const_{tv1modtv2}() \\
\wedge\ tv2 \neq 0 & \\
Or(X,X) &\Rightarrow X
\end{aligned}
$$

$$
\begin{aligned}
Add(X, Const_0) &\Rightarrow X \\
Add(Const_0, X) &\Rightarrow X \\
Eor(X, Const_0) &\Rightarrow X \\
Eor(Const_0, X) &\Rightarrow X \\
Sub(X, Const_0) &\Rightarrow X \\
Shl(X, Const_0) &\Rightarrow X \\
Shr(X, Const_0) &\Rightarrow X \\
Shrs(X, Const_0) &\Rightarrow X \\
Rot(X, Const_0) &\Rightarrow X \\
Minus(Minus(X)) &\Rightarrow X \\
Not(Not(X)) &\Rightarrow X \\
Mul(X, Const_1) &\Rightarrow X \\
Mul(Const_1, X) &\Rightarrow X \\
Div(M, X, Const_1) &\Rightarrow Tuple(M, Bad(), X) \\
And(X, X) &\Rightarrow X \\
And(X, Const_{true}) &\Rightarrow X \\
And(Const_{true}, X) &\Rightarrow X \\
Conv(X) &\Rightarrow X \\
\wedge\ mode(Conv) = mode(X) & \\
Conv_1(Conv_2(X)) &\Rightarrow X \\
\wedge\ mode(Conv_1) = mode(X) = mode\_b & \\
Store(Proj(Store(M, X, V)), X, V) &\Rightarrow Store(M, X, V) \\
Store(Proj(X), P, Proj(Y)) &\Rightarrow Tuple(Proj(X), Bad()) \\
\wedge\ X = Y = Load(M, P) & \\
Store(M, P, Proj(Load(M, P))) &\Rightarrow Tuple(M, Bad()) \\
Proj_0(Cond(Eor(X, Const_1()))) &\Rightarrow Proj_1(Cond(X)) \\
\wedge mode(Eor) = mode\_b & \\
Proj_1(Cond(Eor(X, Const_1()))) &\Rightarrow Proj_0(Cond(X)) \\
\wedge mode(Eor) = mode\_b & \\
Proj_0(Cond(Not(X))) &\Rightarrow Proj_1(Cond(X)) \\
\wedge mode(Not) = mode\_b &
\end{aligned}
$$

$$
\begin{aligned}
Proj_1(Cond(Not(X))) &\Rightarrow Proj_0(Cond(X)) \\
\wedge mode(Not) = mode\_b & \\
Eor(Proj_n(Cmp(...), Const_1())) &\Rightarrow Proj_{negated\_pnc(n)}\, Cmp(...) \\
\wedge\ mode(Eor) = mode\_b & \\
Eor(X, Const_1())) &\Rightarrow Not(X) \\
\wedge\ mode(Eor) = mode\_b & \\
Not(Proj_n(Cmp(...))) &\Rightarrow Proj_{negated\_pnc(n)}\, Cmp(...) \\
\wedge\ mode(Not) = mode\_b &
\end{aligned}
$$

6.2 Unreachable Code and Dead Code Elimination

Unreachable code is code in basic blocks that are never executed. Dead code are expressions that are computed but never used. Such code can emerge in two ways. The front end can construct expressions that are never used or blocks that have no control predecessor. Further Optimizations can leave dead computations when replacing predecessors of nodes by better ones or it can remove predecessors of block nodes by constant evaluation of *Cond* nodes.

Dead code is automatically removed from the program as it is no more reachable in the Firm graph as soon as it gets dead. Nevertheless the dead code still occupies memory in the representation.

Unreachable code is optimized during construction and by a call to `local_optimize_graph()`. When *Cond* nodes are optimized the dead control flow is represented by a *Bad* operation. A Block that has only *Bad* nodes as control flow predecessors is replaced by a *Bad* node itself. All operations (except *Block* and *Phi*) that have a single *Bad* predecessor are replaced by *Bad* nodes. Once the *Bad* nodes are propagated completely only *Block* and *Phi* nodes with some *Bad* predecessors will remain. The unreachable code also still occupies memory after being removed from the representation.

The functions `dead_code_elimination()`, `remove_bad_predecessors()` and `optimize_cf()` remove the *Bad* predecessors of *Block*s and *Phi*s. (See `irgopt.h`.)

The function `dead_code_elimination()` further copies the graph of a procedure to a new memory location. It only copies the reachable nodes and thereby frees the memory of dead and unreachable code. Further the

first call frees memory used for construction of the ir. This method depends on the compiler flag `opt_dead_node_elimination`.

Further libFIRM supplies an optimization that removes methods that are never called and that are not external visible. This reduces memory consumption of the compiler and the size of the generated code. `gc_irgs()` in `ircgopt.h` performs the optimization. It requires that the call graph information (in field `callee` of *Call* nodes) is constructed and accepts a list of methods that may not be removed. This can be established by calling `cgana()`.

6.3 Control Flow Optimizations

A call to `optimize_cf()` (see `irgopt.h`) performs a set of control flow optimizations. These optimizations can not be performed by `local_optimize_graph()` as they require precomputed information. `optimize_cf()` reduces the amount of basic block by removing empty blocks (if simplification, loop simplification), merging single exit / single entry blocks and doing further unreachable code elimination. It depends on flags `opt_unreachable_code` and `opt_control_flow`. Independent of the compiler flags it removes *Bad* predecessor of *Block* and *Phi* nodes as well as *Tuple* nodes in control flow.

Further control flow optimizations are performed by `local_optimize_graph()`. It also merges single exit / single entry blocks and removes *Cond* nodes that branch to the same block on both conditions. It catches less optimization opportunities than `optimize_cf()`, but is more efficient. Further it evaluates constant *Cond* nodes. These optimizations are also controlled by the flags `opt_unreachable_code` and `opt_control_flow`.

6.4 Reassociation

The predecessors of the following nodes are sorted to increase the number of common subexpressions:

Add, *Mul*, *Or*, *And* and *Eor*.

Reassociation is performed by `local_optimize_graph()`. It is controlled by flag `opt_reassociation`.

6.5 Common Subexpression Elimination

A Firm graph has a common subexpression (cs) if it contains two nodes with the same predecessors and the same attributes. Only the essential attributes as defined in [TLB99] are relevant for common subexpressions.

If the common subexpression elimination (cse) finds a pair of such nodes, it replaces one by the other. To perform cse on a Firm graph the optimization flag `opt_cse` must be set during the graph construction or a run of `local_optimize_graph()`.

The Firm library implements cse on basic block level and on procedure level. To perform cse on procedure level flag `opt_global_cse` must be set in addition to `opt_cse`.

The global cse (on procedure level) does not consider block predecessors for the comparison of nodes. If two css differ in their block predecessor the remaining node must be placed in the common dominator of the two original nodes. This means global cse implies code motion. Therefore, if global cse is performed partial redundancy elimination or some other transformation placing nodes must be performed, too. Further it is illegal to move certain nodes. These are *Phi* and control flow nodes and nodes with memory operators. Cse does not move nodes if their opcode is marked as `pinned` (opposite: `floats`, see `irop.h`).

Global cse invalidates the block operands of all floating nodes. This is reflected by a flag `pinned` in the `ir_graph` data structure. If all nodes have a valid block operand this flag is set to `pinned`. If global cse was performed for one or more nodes in the graph the flag is set to `floats`.

`Place_code()` in `irgopt.h` performs code placement and thereby validates the block predecessors. It sets the `pinned` flag in `ir_graph` to `pinned`. Therefore a call to this method is necessary after global cse. See also 6.6.

6.5.1 Common Subexpression Condition

The following condition defines when two nodes a and b are common subexpressions. Two nodes / types ... are equal if they are the same object in the ir.

$$\begin{aligned} opcode(a) &== opcode(b) \quad \wedge \\ mode(a) &== mode(b) \quad \wedge \\ \#predecessors(a) &== \#predecessors(b) \quad \wedge \end{aligned}$$

$$
\begin{aligned}
&\forall i \in 0 \ldots \#predecessors(a) - 1 : \\
&\qquad predecessor(a, i) \;==\; predecessor(b, i) \quad \wedge \\
&\qquad (opcode(a) \neq Const \quad \vee \quad const(a) == const(b)) \quad \wedge \\
&\qquad (opcode(a) \neq Proj \quad \vee \quad proj\_nr(a) == proj\_nr(b)) \quad \wedge \\
&\qquad (opcode(a) \neq Filter \quad \vee \quad proj\_nr(a) == proj\_nr(b)) \quad \wedge \\
&\qquad (opcode(a) \neq Alloc \quad \vee \quad (where(a) == where(b) \\
&\qquad\qquad \wedge\, type(a) == type(b))) \quad \wedge \\
&\qquad (opcode(a) \neq Free \quad \vee \quad type(a) == type(b)) \quad \wedge \\
&\qquad (opcode(a) \neq SymConst \quad \vee \quad (kind(a) == kind(b) \\
&\qquad\qquad \wedge\, tori(a) == tori(b))) \quad \wedge \\
&\qquad (opcode(a) \neq Call \quad \vee \quad type(a) == type(b)) \quad \wedge \\
&\qquad (opcode(a) \neq Sel \quad \vee \quad ent(a) == ent(b)) \quad \wedge
\end{aligned}
$$

For global cse the block node is not included in the list of tested predecessors for floating nodes.

6.5.2 Pinned Nodes

Nodes with the following opcodes are pinned by default in libFIRM:

Block, *Start*, *End*, *EndReg*, *EndExcept*, *Jmp*, *Break*, *Cond*, *Return*, *Raise*, *Call*, *CallBegin*, *Quot*, *DivMod*, *Div*, *Mod*, *Phi*, *Filter*, *Load*, *Store*, *Alloc*, *Free* and *Sync*.

6.6 Code Placement and Partial Redundancy Elimination

`Place_code()` in `irgopt.h` performs Code Placement which subsumes Partial Redundancy Elimination. It requires dominator information that can be computed with `compute_doms()`, see 7.2.

The function `place_code()` places all floating nodes in blocks with the least estimated execution frequency. It first places the nodes in the highest possible dominator of its uses. This results in a legal but inefficient placement. In a second phase the node is moved down in the dominator tree as far as possible. The node is never moved into a loop.

6.7 Inlineing

The library offers two ways to perform inlineing. A call to `inline_method()` inlines a given method at a given call site. `Inline_small_irgs()` inlines at call sites that call a static function, i.e., the address passed to the *Call* node must be a *Const* node containing a method entity as constant value. It iterates in an arbitrary order over all procedures and visits all *Call* nodes. It never inlines a method into itself and never visits inlined *Call* nodes. It only inlines method whose representation occupies less that the given amount of memory. The representation size is used as a measure of procedure size. (See `irgopt.h`.)

Chapter 7

Existing Analyses

libFIRM implements a set of basic analyses. Their implementation and use are explained in this chapter.

7.1 Def-Use Edges

Def-Use edges are edges directed in the sense of the control and data flow. The basic Firm graph represents Use-Def edges as these identify a value. The Def-Use edges are not an essential part of Firm. Nevertheless they are useful for certain optimizations and analyses. The Def-Use edges are called `outs`. Functionality to build and access the outs is implemented in `irouts.h`.

The outs are computed by `compute_outs(irg)` for a graph `irg`. `compute_outs()` allocates some private memory for each graph that can be freed by `free_outs()`. Runtime is O(2*#edges), memory consumption is O(#edges).

The consistency of the out information is controlled by a state flag of the graph. `compute_outs()` sets state `outs_state` to `outs_consistent`. Default state of a graph is `no_outs`. Any transformation of the graph must either set the state to `outs_inconsistent`, call `free_outs()` or repair the out edges. To update the out edges a routine `set_irn_out()` is supplied, but updating is restricted as the amount of out edges can not be adjusted. To access the out edges a walker and access routines are available.

7.2 Dominator Information

libFIRM supplies a module to compute dominator information (`irdom.h`). Whether a graph carries valid dominator information is indicated by a flag `dom_state`. The flag can contain the values `no_dom`, `dom_consistent` or `dom_inconsistent`.

Dominator information is computed by a call to `compute_doms()`. The memory allocated to hold the dominator information can be freed with `free_dom()`. The dominator information supplies the immediate dominator as well as the dominance depth of a basic block.

Control dead basic blocks that are not yet removed contain invalid dominator information. The immediate dominator is `NULL`, the dominance depth -1.

7.3 Back Edges and Strongly Connected Regions

The module `irloop.h` computes back edge and loop information. Loops are represented by a tree structure accessible from the `ir_graph` data structure. The representation of a loop has the following fields:

outer_loop The data structure for the loop this one is contained in.

depth The nesting depth of this loop.

son A list of inner loops.

node The list of nodes in this loop.

The procedure body is considered as the outermost loop.

Further the module defines back edge information for *Block*, *Phi* and *Filter* nodes. Each predecessor of these nodes can be annotated with back edge information. The following access methods are supplied: `is_backedge()`, `set_backedge()`, `set_not_backedge()`, `has_backedges()` and `clear_backedges()`.

The loop and backedge information can be computed for a single procedure by `construct_backedges()`. `construct_ip_backedges()` computes the loop information in interprocedural view. It considers all methods without callers as entry points to the program. It finds all loops in the representation, including recursions and loops arising from several sequential calls to the same method. I.e., the representation contains many not realizable loops.

7.4 Call Graph Analysis

libFIRM supplies a straight forward analysis to construct the call graph: `cgana()` in `cgana.h`. It collects for all *Call* nodes the methods possibly called. It evaluates the address expression passed to the *Call* node. If this expression is a *Sel* node selecting a method entity from a class this method entity and all method entities overwriting this entity can be called. If the expression is a *Const* node referring to a method entity this entity can be called. In all other cases an unknown method can be called.

Further `cgana()` collects and returns all free methods, i.e., methods whose address is stored to memory so that they can be called from *Call* nodes where the address can not be analyzed.

Chapter 8

Manipulating the intermediate representation

8.1 Support for Traversing the intermediate representation

The library provides three groups of flags to mark nodes. Traversals of the ir can use these flags to mark visited nodes. Each group of flags consists of a central master flag to compare against and flags in the nodes traversed. Each algorithm using these flags must assert that the master flag is greater or equal to the flags in all of the corresponding nodes after the traversal.

8.1.1 Flag for Firm Nodes

All Firm nodes have a flag `visited`. The corresponding master flag is the flag `visited` in the `ir_graph` data structure. Further exists a master flag `max_irg_visited` that always contains the maximum off all visited master flags in `ir_graph`. Access methods:

```
get_irn_visited()
set_irn_visited()
mark_irn_visited()
inc_irg_visited()
set_irg_visited()
get_irg_visited()
```

8.1.2 Flag for Firm Block-Nodes

All Firm block nodes have a flag `block_visited`. With this flag simultaneous traversals of the full Firm graph and the control flow subgraph are possible. The corresponding master flag is the flag `block_visited` in the `ir_graph` data structure. Access methods:

```
get_Block_block_visited()
set_Block_block_visited()
mark_Block_block_visited()
inc_irg_block_visited()
set_irg_block_visited()
get_irg_block_visited()
```

8.1.3 Flag for Types and Entities

All Firm types and entities have a flag `visited`. The corresponding master flag is the global variable `type_visited` defined in `type.h`. Access methods:

```
get_entity_visited()
set_entity_visited()
mark_entity_visited()
get_type_visited()
set_type_visited()
mark_type_visited()
```

8.2 Existing Traversal Functions

libFIRM supplies a set of traversal functions implemented with the above flags. These are defined in the headers `irgwalk.h` and `typewalk.h`.

The function `irg_walk()` walks over all Firm nodes reachable from the node passed as argument. It has two functions, pre and post, as arguments. For each node it visits it calls pre, then iterates to the nodes predecessors and finally calls post. It increments the master visited flag for nodes before the traversal and sets the flag of a node to the value of the master flag before calling pre. The functions pre or post must deal carefully with the visited flags of new or changed nodes.

The function `irg_block_walk()` walks over the control flow graph reachable from the node passed. It uses the visited flag for *Block* nodes. It increments the corresponding master visited flag before the traversal. Before visiting a node it sets the visited flag of the node to the value of the master flag. For each block node it executes pre. Then it walks to the predecessor

blocks. It finds these by walking from the blocks predecessors, passing *Id*, *Tuple* and *Proj* nodes to the next control operation and taking its block. After the walk it executes post.

The function `cg_walk()` walks over all nodes in all graphs in interprocedural view. Sets `current_ir_graph` properly.

The function `walk_const_code()` walks over all nodes representing constant expressions in `const_code_irg()`.

Several shortcuts exist to call these walkers. See `irgwalk.h` for further information.

The function `type_walk()` walks over all types and entities. `type_walk_irg()` walks only over type information reachable from a certain graph. The function `type_walk_super2sub()` walks over all class types. It guarantees that all super types have been visited before executing pre on a class. Post will be executed at some point after executing pre. All type walkers use the visited flag for types.

8.3 Support for Transformations

8.3.1 Transformation of Firm Graphs

The header `irgmod.h` defines several methods to transform Firm graphs.

The function `exchange()` replaces one Firm node by another by turning the old node into an *Id* node. This is illustrated in Figure 8.1.

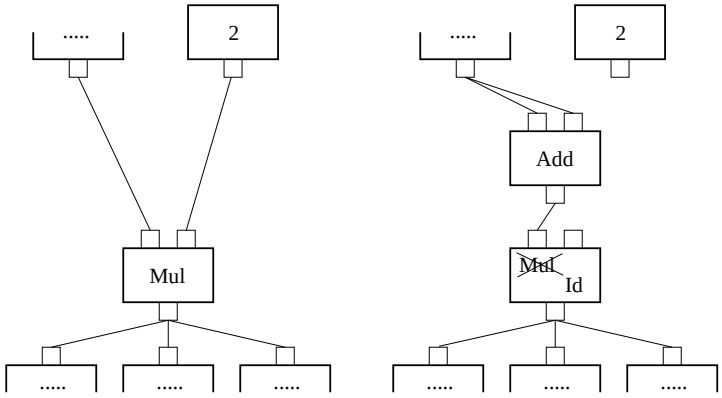


Figure 8.1: Use of *Id* node: replacing a node by turning the old node into an *Id* node. Constant expression evaluation removes the *Id* nodes.

The function `turn_into_tuple(node, arity)` turns `node` into a *Tuple* node. The *Tuple* is in the same block as node. It has `arity` fields for data predecessors that are not initialized. Figure 8.2 illustrates the use of this function. This function can be used to replace nodes of mode tuple. Set the *Tuple*s predecessors with `set_Tuple_pred()`.

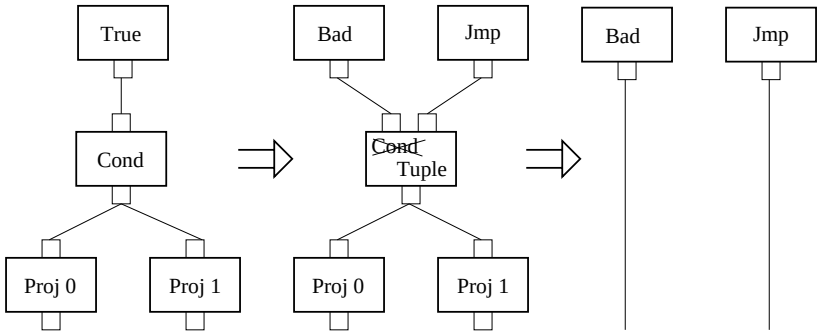


Figure 8.2: Use of *Tuple* node: replacing a node by turning the old node into a *Tuple* node. Constant expression evaluation removes the *Tuple* and the *Proj* nodes.

The function `collect_phiprojs()` collects all *Phi* nodes in a block as a linked list in the link field of the block and all *Proj* nodes as a linked list in the node producing the tuple. This is needed as a precondition for `part_block()`. *Proj*s for nested tuples are collected in the node producing the outermost tuple.

The function `part_block()` parts a block into two. It gets a single node as argument. The block of this node is parted. This is useful to insert other blocks or new control flow within a given block. The function adds a new block in the control flow before the block (old block) of the argument node. It moves node and its predecessors from the old block to the new block. Further it moves all *Proj*s that depend on moved nodes and are in the old block to the new block and it moves all *Phi* nodes from the old block to the new block. To achieve this the routine assumes that all *Phi* nodes are in a list (using the link field) in the link field of the old block. Further it assumes that all *Proj* nodes are accessible by the link field of the nodes producing the tuple. This can be established by `collect_phiprojs()`. `part_block()` conserves this property. The function adds a *Jmp* node to the new block that jumps to the old block. It assumes that node is contained in current_ir_graph. It sets current_block in this ir_graph to the new block. Figure 8.3 illustrates the effect of `part_block()`.

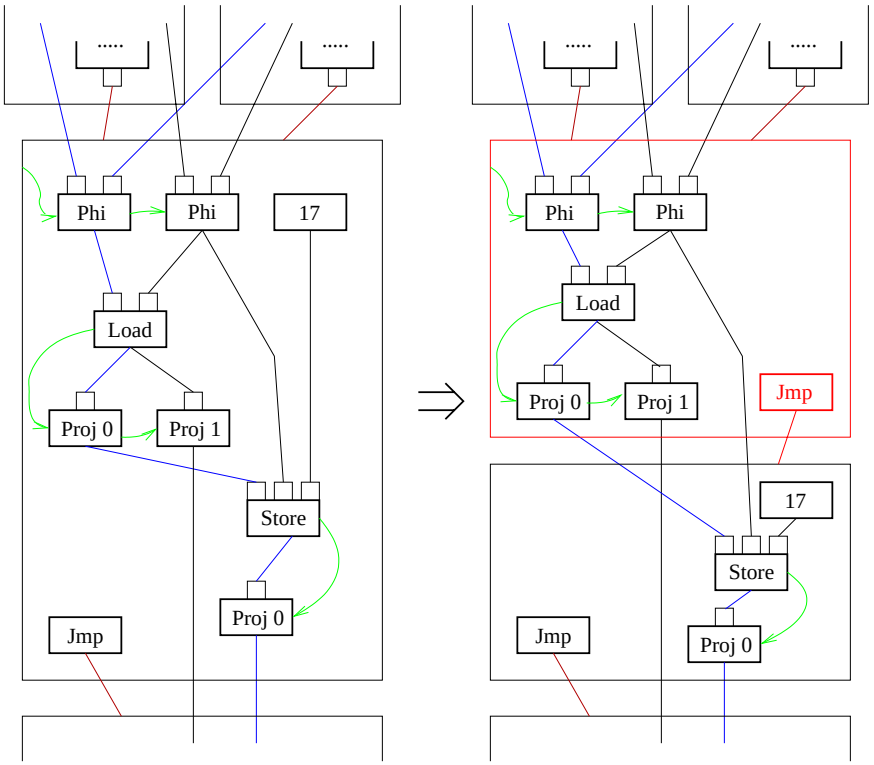


Figure 8.3: The effect of `part_block()` if called for the *Load* node. The green edges indicate the lists of *Phi* and *Proj* nodes. The red nodes are introduced by `part_block()`.

8.3.2 Transformation of Type Information

Functionality supporting transformation of type information is implemented in `typegmod.h`. The method `exchange_types()` replaces a type by a new one turning the old type into an Id type. This transformation is dual to `exchange()` as explained in 8.3.1.

Chapter 9

Firm and Debug Support

9.1 Rational

libFIRM is a library dedicated to develop and test compiler optimizations. Therefore supporting debugging in a compiler using libFIRM is not a primary goal. Nevertheless there is the need to include support to transport information necessary for debugging programs translated with a compiler incorporating libFIRM.

Therefore libFIRM supports an interface to a module implementing debug support. It allows to annotate each Firm node with arbitrary debug information, e.g., the origin file and line and column numbers. Further libFIRM calls routines after performing code transformations to update the debug information. This is necessary if the optimization decides to replace a node annotated with debug information by a new one.

9.2 Interface for Debugging Support

Each Firm node contains a reference to debug information. This reference can be administrated with the two access routines `get_irn_dbg_info()` and `set_irn_dbg_info()`. If the get routine is called for a node without debug information it returns `Null`. Further libFIRM supplies a set of node constructors that allow to pass a reference to the debug information. All optimizations call one of two methods to update this information after a transformation of the Firm graph. The optimizations pass a flag of type `dbg_action` indicating the optimization performed. Everything else is left to the module implementing the debugging support.

libFIRM expects the debug information to be of the type

```
\texttt{typedef struct dbg\_info dbg\_info;}
```

If the optimization can match single nodes it calls a function with the signature

```
void deb\_info\_copy(ir\_node *new, ir\_node *old,
                    dbg\_action info);}
```

where `old` is the node now replaced by `new`. If the optimization replaces larger subgraphs by another subgraph and there is no obvious mapping between single nodes in both subgraphs it calls

```
void deb\_info\_merge(ir\_node **new\_nodes, int n\_new\_nodes,
                     ir\_node **old\_nodes, int n\_old\_nodes,
                     dbg\_action info);
```

I.e., the optimization simply passes two lists to the debug module, one containing the nodes in the old subgraph, the other containing the nodes in the new subgraph.

The initialization routine of the debug module expects a pointer to these two functions. The optimizations call the functions by these pointers.

Index

Bibliography

[Sch02] Hubert Schmid. Explizite Interprozedurale Abhängigkeitsgraphen. Studienarbeit, Dept. of Computer Science, University of Karlsruhe (TH), June 2002.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Dept. of Computer Science, University of Karlsruhe (TH), December 1999.