Interfacing Compilers with PAG

Gergő Bárány

January 17, 2005

1 Introduction

1.1 Overview

PAG is a tool for generating program analyzers that can be used with existing compilers or built into new ones. The analyses themselves are specified in the high-level functional language FULA, which is compiled into a C library performing the analysis.
Since the analyzer needs access to the program's control flow graph (CFG) and the abstract syntax tree (AST) for each statement, some sort of interface between the compiler and the analyzer must be implemented. A tool called GON can automatically generate this interface, but only for compilers that are written from scratch.
The purpose of this document is to describe the glue code that must be written in oder to connect an existing compiler with PAG. It also investigates which parts of the interface can be generated automatically.

1.2 Required files

The interface must provide the following files, each of which is explained in more detail below:

- `edges`: defines the edge types that may occur in the CFG
- `syn`: a tree grammar describing the abstract syntax of the language
- `pagoptions`: describes which access functions for syntactic lists are implemented
- `syntree.h`: defines all types for the abstract syntax tree
- `iface.h`: defines all types that must be provided by the interface
- `syntree.c`: contains the code for access functions (or macros) to the CFG and the AST

The rest of the document gives detailed information on what each of these files should contain. It considers the CFG part first, then the AST.

2 The CFG Interface

This section describes the files, types and functions for CFG access that the interface must provide.
The CFG consists of a set of nodes, each numbered with a unique id starting from 0. Every node represents a basic block, possibly containing several instructions. Interprocedural edges are only allowed from `Call` to `Start` and from `End` to `Return` nodes.

2.1 The `edges` file

The `edges` file lists all edge types occurring in the CFG. Each type name must be listed on a separate line, line comments beginning with `//` are allowed.
The first edge type must be the type for local edges, called for instance `local_edge`; these are the edges from function call nodes to the corresponding return nodes. The second type is the type `bb_intern` of edges connecting statements inside a basic block.

2.2 Required types

The following types must be defined in `iface.h`:

Name	Description	Type Restrictions
`KFG`	The CFG itself	must be a pointer
`KFG_NODE_TYPE`	Type of node classes	enumeration type or `int`
`KFG_NODE_ID`	Type of node identifiers	must be `int`
`KFG_NODE`	Type of CFG nodes	must be a pointer type
`KFG_NODE_LIST`	Type of node lists	must be a pointer type
`KFG_EDGE_TYPE`	Type of edge classes	enumeration type or `int`

The type `KFG_NODE_TYPE` must at least contain the enumeration constants `RETURN`, `CALL`, `START`, `END`, and `INNER` with values 0–4. It may support additional constants with larger values.
Further, `KFG_EDGE_TYPE` must contain constants for local and basic-block-internal edges with values 0 and 1; the rest of the contants should also be analogous to those declared in the `edges` file.

2.3 Required functions

PAG requires the front end to implement a number of functions for accessing, and traversing the CFG in numerous ways.
The basic CFG access functions are:

Prototype	Description
`KFG kfg_create(KFG)`	initialize the CFG
`int kfg_num_nodes(KFG)`	number of nodes in the CFG
`KFG_NODE_TYPE kfg_node_type(KFG, KFG_NODE)`	type of the node
`KFG_NODE_ID kfg_get_id(KFG, KFG_NODE)`	identifier of the node

Prototype	Description
`KFG_NODE kfg_get_node(KFG, KFG_NODE_ID)`	node with the given identitier
`int kfg_get_bbsize(KFG, KFG_NODE)`	number of instructions in the node
t `kfg_get_bbelem(KFG, KFG_NODE, int)`	the n-th instruction of the node, starting with 0; t is the AST type
`void kfg_node_infolabel_print_fp (FILE *, KFG, KFG_NODE, int)`	write a textual description of the n-th instruction of the node to the file (used for visualization)
`KFG_NODE_LIST kfg_predecessors(KFG, KFG_NODE)`	list of predecessors of the node
`KFG_NODE_LIST kfg_successors(KFG, KFG_NODE)`	list of successors of the node
`KFG_NODE kfg_get_call(KFG, KFG_NODE)`	the call node belonging to the given return node
`KFG_NODE kfg_get_return(KFG, KFG_NODE)`	the return node belonging to the given call node
`KFG_NODE kfg_get_start(KFG, KFG_NODE)`	the start node belonging to the given end node
`KFG_NODE kfg_get_end(KFG, KFG_NODE)`	the end node belonging to the given start node
`const int *kfg_get_beginnings(KFG)`	Returns a pointer to an array of procedure numbers, terminated by -1, to start the analysis with. If the function returns an empty list (contains only -1) then the analyzer selects entry point automatically.
`int kfg_replace_beginnings(KFG, const int *)`	replaces the beginnings list of the front end, can be called after initialization of the CFG before the analysis; returns 1 for success, 0 if the feature is not supported, -1 for an error

The node list functions are:

Prototype	Description
`KFG_NODE kfg_node_list_head (KFG_NODE_LIST)`	head of list
`KFG_NODE_LIST kfg_node_list_tail (KFG_NODE_LIST)`	list without the first element
`int kfg_node_list_is_empty (KFG_NODE_LIST)`	1 if the list is empty, 0 otherwise
`int kfg_node_list_length (KFG_NODE_LIST)`	length of node list

The edge type functions are:

Prototype	Description
`unsigned int kfg_edge_type_max(KFG)`	number of different edge types
`KFG_EDGE_TYPE kfg_edge_type (KFG_NODE, KFG_NODE)`	type of the edge from the first node to the second; runtime error if there is no such edge
`int kfg_which_in_edges(KFG_NODE)`	returns a bitmask with a bit corresponding to an edge type set if there is an incoming edge of that type
`int kfg_which_out_edges(KFG_NODE)`	as `kfg_which_in_edges` but for outgoing edges

Procedure access functions are:

Prototype	Description
`int kfg_num_procs(KFG)`	number of procedures in the CFG
`char *kfg_proc_name(KFG, int)`	static pointer to the name of a procedure
`KFG_NODE kfg_numproc(KFG, int)`	entry node of a procedure
`int kfg_procnumnode(KFG, KFG_NODE)`	number of the procedure the node belongs to
`int kfg_procnum(KFG, KFG_NODE_ID)`	number of the procedure the node with the given id belongs to

Functions for accessing collections of nodes are:

Prototype	Description
`KFG_NODE_LIST kfg_all_nodes(KFG)`	list of all nodes
`KFG_NODE_LIST kfg_entrys(KFG)`	list of all entry nodes
`KFG_NODE_LIST kfg_calls(KFG)`	list of all call nodes
`KFG_NODE_LIST kfg_returns(KFG)`	list of all return nodes
`KFG_NODE_LIST kfg_exits(KFG)`	list of all exit nodes

All of these functions may optionally be implemented as C macros. They must be declared in `iface.h`.

2.4 Data structures for the CFG

Given the specifications of the access functions, designing an appropriate data structure is easy: `KFG_NODE` can be implemented as a pointer to a structure containing an id, a type, a list of statements (the basic block represented by this node) and the size of this list, lists of predecessors and successors (possibly with the appropriate edge types), the number of the procedure the node belongs to, and precomputed bitmasks for the `kfg_which_in_edges` and `kfg_which_out_edges` functions.
Depending on the underlying language, it might not be necessary to explicitly store the types of the edges connecting two nodes, since this can often be determined from the types of the nodes alone. For instance, the edge from a 'normal' node will be a normal edge, the edge from a return node will be a return edge; for an if node, the edge will be a true or false

edge depending on whether the successor is stored as the 'true' or 'false' successor of this node. Also, the list of successors will usually have at most two elements (one for the true case, one for false).
`KFG_NODE_LIST` can be defined as `KFG_NODE *`. Lists of nodes are then implemented as null-terminated arrays of `KFG_NODE`, ensuring fast direct access and traversal. Since the CFG is required to be constant once it has been created, one need not worry about the possible costs of modifying such arrays at runtime.
The `KFG` itself can just be a pointer to a structure containing a list of all nodes, lists of special nodes (procedure entry nodes, calls, returns and exits), a list of (procedure number, procedure name) pairs and a list of procedure numbers to start the analysis with. In the list of all nodes, each node can be stored at the index corresponding to its id, ensuring fast lookups and technically eliminating the need for storing the id explicitly.
These types should be made available to PAG through the `iface.h` file.

2.5 Implementation of the CFG functions

For a data structure as described above, implementing the CFG access functions required by PAG is rather straightforward; most functions just return a certain structure field or array element. All operations except `kfg_create` and the procedure number lookups can be implemented to run in constant time.
Automatic conversion of an existing CFG or other intermediate representation from a compiler front end might be possible in principle, especially if the data structures are similar enough to the ones described above. Conversely, it should be possible to generate the required access functions and leave the existing CFG completely unchanged. The problem with these approaches is the difficulty of specifying just what should be converted in which way; the specification for the conversion tool would in general have to be very complicated. Therefore it appears more reasonable to write the necessary code by hand.
The main part of this work consists of collecting all statements to basic blocks, linking these with each other and computing the auxiliary information. Note that the requirement that a node represent a whole basic block is not enforced by PAG, it is merely strongly suggested for efficiency reasons. It is possible to store exactly one statement per node, making creation of the CFG somewhat simpler. The example C compiler front end that is shipped with PAG uses this approach.
The code described in this section should reside in `syntree.c`.

3 The AST Interface

The AST interface consists of a tree grammar describing the structure of the tree, the corresponding C type declarations, and C functions implementing syntax tree access, type tests and type conversions, and syntactic list traversal.

3.1 The `syn` file

The `syn` file describes the abstract syntax of the language under consideration by a tree grammar. Figure 1, taken from [1], shows a brief excerpt of an example `syn` file.
`START` specifies the start symbol of the grammar, every instruction inside a CFG node must be associated with an AST of this type. Alternatives for a production are grouped together.

```
SYNTAX
START: mirStmt

mirStmt: CFGCall(exp: mirExpr)
       | CFGEndCall(exp: mirExpr, sym: mirSymbol*)
         ...
       ;

mirExpr: mirChar(str: CHAR, type: mirType)
         ...
       ;

CHAR == chr;
INT  == snum;
```

Figure 1: An example `syn` file

For instance, a `mirStmt` is either a node of type `CFGCall` with a child node `exp` of type `mirExpr` or a node of type `CFGEndCall` with two child nodes `exp` as above and `sym` of type `mirSymbol*`. The `*` indicates that this type is a syntactic list of `mirSymbol` terms. This abbreviation is provided by PAG since lists are so common.
The production for `mirExpr` contains a reference to the type `CHAR`. There is no grammar rule for this type; rather, it is an alias type defined by the equivalence `CHAR == chr`. This means that it corresponds to the built-in FULA type `chr`. The interface needs to provide conversion functions for each such alias type to enable the analysis to use values of these types.

3.2 Required types

For each type (including alias types) defined in the `syn` file, a C type with the same name must be declared. For each syntactic list over a type T used in the tree grammar, a type named `LIST_`T must be defined.
Further, it is possible to define cursors for syntactic lists. These are abstract data types for traversing syntactic lists, enabling them to be used in ZF expressions in the FULA language. For each list over a type T, the types `_LIST_`T`_cur` and `LIST_`T`_cur` must be defined, where the latter is a pointer to the former.
The types described in this section must be declared in `syntree.h`.

3.3 Required functions

For every type constructor c`(`n_1`:`t_1`, ...,`n_k`:`t_k`)` of type t in the `syn` file, PAG requires functions for element access and for type testing.
The access functions are t_i t`_`c`_get_`n_i`(`t`)` for accessing the child named n_i of type constructor c for type t. The functions take a node of type t as their sole argument and return a node of type t_i.
The test functions are `int is_op_`t`_`c`(`t`)`. These return 1 if the tree node of type t passed in is labeled with the constructor c, 0 otherwise.

Alias types, declared as t `==` p in the `syn` file, need special treatment. They are internally represented as C types but need the ability to be converted to FULA types. This must be realized by functions of the form `char *`t`_get_value(`t`)` returning a string representation of the value of t. PAG can then convert this string to the primitive FULA type p.
For each syntactic list over a type T the following functions must be defined:

Prototype	Description
`int LIST_`T`_empty(LIST_`T`)`	1 if the list is empty, 0 otherwise
T `LIST_`T`_hd(LIST_`T`)`	head of the list
`LIST_`T `LIST_`T`_tl(LIST_`T`)`	tail of the list

As explained above, it is possible to define syntactic list cursors. The cursor functions for lists over a type T are given in the following table:

Prototype	Description
`void LIST_`T`_cur_reset(LIST_`T`_cur, LIST_`T`)`	initialize the cursor
`void LIST_`T`_cur_is_empty (LIST_`T`_cur)`	1 if the list is empty, 0 otherwise
T `LIST_`T`_cur_get(LIST_`T`_cur)`	current element of the list
`void LIST_`T`_cur_next(LIST_`T`_cur)`	advance the cursor by one element
`void LIST_`T`_cur_destroy(LIST_`T`_cur)`	destructor of the cursor (optional)

All of the AST functions must be defined via `syntree.c`.

3.4 The `pagoptions` file

The `pagoptions` file tells PAG which features respecting syntactic lists are supported by the front end. The contents of this file should almost always be the following:

```
LIST_is_empty       : 1
LIST_hd             : 1
LIST_tl             : 1
LIST_cursor         : 1
LIST_cursor_destroy : 1
```

The first three lines indicate that the basic syntactic list features are supported. Since these are always required if syntactic lists are used, there is not much choice in whether to define them.
The last two lines indicate that support for list cursors and for list cursor destructors is present. Implementing cursors efficiently should not be very difficult either once lists are supported at all, so these can be defined as well.
If the implementor should decide not to support a certain feature, the indicator in the corresponding line can be set to `0` to reflect this.

3.5 Additional requirements

The PAG manual lists a few other functions that must be present if a front end is written from scratch. See page 89 of [1] for details.

3.6 Automatic AST interface generation

The `syn` file alone is enough to generate an implementation of almost all of the abstract syntax tree data types and functions automatically.
This is not possible for alias types, however: The converter cannot know which C type should be used as the internal representation for the type. Thus, the user must provide an appropriate type definition and an implementation of the corresponding `get_value` function for each alias type.
The integration of the syntax tree of an existing front end can be accomplished quite simply by generating the required functions for accesses and tests of this tree. For this idea to work, it is possible to write a simple tool that parses the `syn` file and another file provided by the user, describing the functions to be generated with a simple macro language.
Consider a production c_j`(`n_1`:`t_1`, ...,`n_k`:`t_k`)` for a type t in the `syn` file. It is reasonable to assume that most AST nodes share a basic structure, making accesses to their fields all very similar, depending only on some of the following:

- the type name t of the node itself
- an expression denoting the particular node object
- the name c_j of the type constructor
- the index j of the constructor, denoting that this is the j-th alternative for type t in the `syn` file
- the name n_i of the requested field
- the index i of the requested field
- the type name t_i of the requested field (this might be useful for type casts)

If the code is always the same except for these details, it is rather simple to create it as an instance of a macro description with these parameters. These macros would be provided by the user, an interface generation tool would then turn them into code for all AST functions required by PAG. The accesses are expected to be uniform in most, but not in all cases, so special cases (for certain types) must be handled as well.
Here is a fictional example of the possible syntax of such a specification:

```
get("mirStmt", NODE, CONSTR, CONSTR_IDX, FIELD, FIELD_IDX, FIELD_TYPE)
{
    return NODE->children[CONSTR_IDX][FIELD_IDX];
}

get(TYPE, NODE, CONSTR, CONSTR_IDX, FIELD, FIELD_IDX, FIELD_TYPE)
{
    return NODE->children.f_##CONSTR.FIELD;
}
```

The indended meaning of this snippet is the following: Functions for type `mirStmt` are created by the first rule because the head matches just this type. All functions for the other types are matched by the second rule.

Children of a node of type `mirStmt` are accessed by indexing a two-dimensional array of child nodes. Thus, to access the first child (`expr`) of the second production (`CFGEndCall`), the function

```
mirExpr mirStmt_CFGEndCall_get_exp(mirStmt node)
{
    return node->children[1][0];
}
```

would be generated.
For all other types the second rule would apply, producing for instance the code

```
mirType mirExpr_mirChar_get_type(mirExpr node)
{
    return node->children.f_mirChar.type;
}
```

for accessing the second child node in the first production for the `mirExpr` type. Notice the use of the C preprocessor's token pasting operator `##` to construct the field name `f_mirChar` from the constant prefix `f_` and the constructor's name.
Things are a bit more complicated for the test functions: The AST presumably already has some sort of type test using integer constants (or even strings). However, the numbering or naming for these might be different from the one that the conversion tool would create by itself. In this case, the user would have to specify some sort of mapping between the constructor names in the `syn` file and the actual constants used in the AST.
For the alias types defined in the `syn` file, an interface generation tool cannot know which C type was intended to implement this alias type. Thus the tool cannot generate code for alias types, the user must provide this code himself.

3.7 Comparison to the CFG interface

The difference to the CFG interface is the following: While the CFG uses comparatively few different functions, the AST calls for a quite large number of functions, all of which are instances of just one of two patterns (assuming that all AST nodes have the same basic structure).
Thus generating the AST functions from a simple description should be rather easy, while in the CFG case much more detailed specifications would be needed. This would in many cases lead to descriptions that are just as complex as a manual implementation of the function, thus losing the advantages of automatic code generation.

4 Summary

It is possible to write a tool which generates large parts of the interface between an existing compiler front end and PAG automatically. The input to this tool would consist of a `syn` file describing the AST structure in the compiler, and a second file of macros describing the way fields of the AST are accessed.
The tool creates from these the complete AST interface implementation and stubs for the alias type conversion functions as well as the CFG access functions. The programmer must then fill in the function definitions and provide the necessary type definitions.

The automatically generated AST saves the programmer time if the specification is significantly shorter than the resulting program, i.e. if writing the specification is less tedious than writing the access code by hand. This should be the case if access to the AST nodes is similar in most cases, subsuming many functions under one macro specification.

References

[1] AbsInt Angewandte Informatik GmbH: *PAG, The Program Analyzer Generator: User's Manual*, 2002