CompilerForCAP

This package allows to "compile" the code of CAP categories.

2020.07.09

9 July 2020

Fabian Zickgraf

Fabian Zickgraf
Email: fabian.zickgraf@uni-siegen.de
Homepage: https://github.com/zickgraf/
Address: Walter-Flex-Straße 3
57068 Siegen
Germany

Contents

Chapter 1

Using the compiler

WARNING: This package is still in alpha and not tested or validated extensively!

1.1 Terminology

The compiler is a just-in-time compiler, that is, it needs some arguments to infer types of variables. These arguments are refered to as *JIT arguments*. For a given CAP operation, these are usually the arguments of the first call of the operation.

The compiler uses GAP's syntax trees for optimizing functions. The term *tree* always refers to the syntax tree of the function to be compiled. Note that a node of the tree always knows its children, so technically it is also a tree. That is, the terms *tree*, *subtree*, and *node* technically describe the same thing but we use them for different emphases.

We often replace a node in the tree by another tree representing the "value" of the original node. Examples:

- Replace a global variable referencing an integer, a string, or a boolean by EXPR_INT, EXPR_STRING, EXPR_TRUE or EXPR_FALSE.
- Replace a global variable referencing a plain function by the syntax tree of this function.
- Replace a record access of a global function by the value of this record access.
- Replace an operation by a concrete method.

We call this *resolving* the global variable, operation, etc. Note that this does not change the basic "layout" of the tree.

On the contrary, in the following examples we change the "layout" of the tree:

- If we have a function call of a resolved function, we can assign the argument values to local variables at the beginning of the function. This way we can avoid passing arguments completely.
- If a function call of a resolved function occurs in the right hand side of a variable assignment, we can insert the body of the resolved function right before the variable assignment. This way we can avoid the function call.
- We can replace all references to a local variable by the right hand side of the variable assignment and then drop the assignment.

We call this *inlining* the function arguments, functions, or variable assignments.

1.2 Capabilities of the compiler

The compilation process has two phases: the resolving phase and the rule phase.

During the resolving phase, operations and global variables are resolved:

- An operation is resolved by executing the function to be compiled with the JIT arguments to determine the arguments of the operation. These arguments are used to call `ApplicableMethod`, and methods annotated with the pragma `CAP_JIT_RESOLVE_FUNCTION` are resolved.

- CAP operations are handled separately: instead of using `ApplicableMethod`, the functions added to the category via `Add` functions are considered, and those do not have to be annotated with `CAP_JIT_RESOLVE_FUNCTION`. In particular, caching, pre functions, etc. are bypassed.

- References to global functions are resolved if the function is annotated with the pragma `CAP_JIT_RESOLVE_FUNCTION`.

For details see `CapJitResolvedOperations` (2.5.10) and `CapJitResolvedGlobalVariables` (2.5.9). If no operation or global variable can be resolved anymore, we continue with the rule phase.

In the rule phase, the tree is optimized using several rules and techniques. Function arguments, functions, and assignments to local variables are inlined. Unused variables are dropped. Handled edge cases are dropped, that is, if the same edge case is caught multiple times via `if condition then return ...; fi;` statements, only the first such statement is kept. Finally, "logic" is applied to the tree. For example, calls of `CallFuncList` are replaced by calls to the actual function. The logic can be extended by the user, see chapter 2.

For all details, see the list of compilations steps in 2.5.

1.3 Requirements

There are three main requirements for the steps described above to be correct:

- The code must not depend on side effects (otherwise dropping "unused" variables or inlining variables could change results). See `CapJitThrowErrorOnSideEffects` (2.4.8) for details.

- The methods selected for the operations during the resolving phase must be independent of the JIT arguments, that is, they must yield correct results for all allowed arguments of the function to be compiled. Thus, be careful which methods you annotate with `CAP_JIT_RESOLVE_FUNCTION`. In particular, the CAP categories of objects and morphisms appearing during the execution must be independent of the JIT arguments.

- All results of applications of filters in logic templates must be independent of the JIT arguments. Thus, be careful when adding new logic templates.

There is an additional weak requirement: The compiler mainly optimizes the code paths covered when executing the function with the JIT arguments. Thus, the JIT arguments should represent a "generic" call, i.e., they should not run into edge cases which do not happen with a "generic" call. Still, the execution using JIT arguments should be fast to improve compilation times.

Additionally, there is not detection for recursive function calls currently, so resolving such a function call leads to an infinite loop.

1.4 Activating the compiler

You can activate the compiler by passing the option `enable_compilation` to any category constructor. If `enable_compilation` is set to `true`, any basic operation will be compiled when called for the first time. If `enable_compilation` is a list of strings, compilation will only happen if the function name of the basic operation appears in this list.

1.5 Stopping the compiler at a certain level

You can use `StopCompilationAtCategory` to prevent the compiler from inlining and optimizing code of a given category. You can revert this decision using `ContinueCompilationAtCategory`.

1.5.1 StopCompilationAtCategory

▷ `StopCompilationAtCategory(`*`category`*`)` (function)

Stops the compiler from inlining and optimizing code of *`category`*.

1.5.2 ContinueCompilationAtCategory

▷ `ContinueCompilationAtCategory(`*`category`*`)` (function)

Allows the compiler to inline and optimize code of *`category`* (this is the default).

1.6 Getting information about the compilation process

You can increase the info level of `InfoCapJit` to get information about the compilation process.

1.6.1 InfoCapJit

▷ `InfoCapJit` (info class)

Info class used for info messsages of the CAP compiler.

1.7 Compiling a function manually

Use `CapJitCompiledFunction` (1.7.1) to compile a function `func` with JIT arguments `jit_args`. `jit_args` should represent a "generic" call, i.e., they should not run into edge cases which do not happen with a "generic" call. Still, the execution using `jit_args` should be fast to improve compilation times.

1.7.1 CapJitCompiledFunction

▷ `CapJitCompiledFunction(`*`func, jit_args`*`)` (function)

Returns: a function

Returns a compiled version of the function *`func`*. The arguments *`jit_args`* are used to infer the types of variables.

1.8 FAQ

- Q: Why is my function not resolved?

A: Only functions annotated with the pragma `CAP_JIT_RESOLVE_FUNCTION` are resolved. Additionally, a function can only be resolved if it appears as a global variable in the tree during the resolving phase of the compilation. That is, it must be referenced from a global variable from the beginning on, or after global variables are resolved by `CapJitResolvedGlobalVariables` (2.5.9). Possibly you have adapt `CapJitResolvedGlobalVariables` (2.5.9) to your setting.

- Q: Why is my operation not resolved?

A: The compiler must be able to get the arguments of the call of the operation from the JIT arguments. Then the rules in the description of `CapJitResolvedOperations` (2.5.10) apply.

- Q: Why do I get the error "cannot find iteration key"?

A: For each syntax tree node type, the tree iterator has to know which record names it should use for continuing the iteration. Please add the missing keys to `CAP_INTERNAL_JIT_ITERATION_KEYS`.

- Q: Why do I get the error "tree has no kown type" when calling `CapJitPrettyPrintSyntaxTree` (2.4.3)?

A: `CapJitPrettyPrintSyntaxTree` (2.4.3) needs to handle every syntax tree node type separately. Please add the missing type to `CapJitPrettyPrintSyntaxTree` (2.4.3).

- Q: Why is do I get the error "a local variable with name <name> is assigned more than once (not as a part of a rapid reassignment), this is not supported"?

A: For reasons of correctness, variables cannot be inlined if a variable is assigned more than once in the body of a function (this includes function arguments which are assigned at least once, namely when the function is called). An exception is made for "rapid reassignments": if the same variable is assigned and then reassigned immediately in the next statement, this only counts as a single assignment.

- Q: Why do I get one of the following errors: "tree includes statements or expressions which indicate possible side effects", "tree contains an assignment of a higher variable with initial name <name>, this is not supported", or "tree contains for loop over non-local variable, this is not supported" ?

A: We can only drop unused variables, inline variables, etc. if we assume that the code contains no side effects. Statements like STAT_PROCCALL or assignment to higher variables cause (or at least indicate) side effects, so continuing with the compilation would probably not lead to a correct result.

Chapter 2

Improving and extending the compiler

The easiest way to extend the compiler is by adding more logic to it, see 2.1. For writing logic functions you also have to iterate over enhanced syntax trees, see 2.2 and 2.3. You might also want to use available tools, see 2.4. If you want to improve an existing compilation step or add a completely new one, see 2.5.

For debugging you can:

- use `CapJitPrettyPrintSyntaxTree` (2.4.3),
- set `debug` to `true` in `CapJitCompiledFunction` (1.7.1) (Note: this causes informational break loops which are not actual errors),
- use the `debug` and `debug_path` record entries of logic templates (see `CapJitAddLogicTemplate` (2.1.2)).

2.1 Logic

2.1.1 CapJitAddLogicFunction

▷ `CapJitAddLogicFunction(`*`func`*`)` (function)

Adds the function *func* to the list of logic functions. For a list of pre-installed logic functions, which can be used as guiding examples, see `CompilerForCAP/gap/Logic.gi`. Technically, *func* should accept an (enhanced) syntax tree and some JIT arguments and return an (enhanced) syntax tree. Semantically, *func* should use some kind of "logic" to transform the tree. For example, *func* could look for calls of `CallFuncList` and replace them by calls to the actual function. Note: Often it is easier to use a logic template (see `CapJitAddLogicTemplate` (2.1.2)) than a logic function.

2.1.2 CapJitAddLogicTemplate

▷ `CapJitAddLogicTemplate(`*`template`*`)` (function)

Adds the logic template *template* to the list of logic templates. For a list of pre-installed logic templates, which can be used as guiding examples, see `CompilerForCAP/gap/LogicTemplates.gi`. Logic templates are records with the following entries:

- `src_template` and `dst_template` (required): strings containing valid GAP code
- `variable_names` (required): a list of strings
- `variable_filters` (optional): a list of filters with the same length as `variable_names`, defaults to a list of `IsObject`
- `returns_value` (required): a boolean
- `new_funcs` (optional): a list of lists of strings, defaults to the empty list
- `needed_packages` (optional): a list of pairs (i.e. lists with two entries) of strings, defaults to the empty list
- `debug` (optional): a boolean
- `debug_path` (optional): a path

Semantics: `src_template` is a piece of code which should be replaced by `dst_template`:

- The function `CapJitAppliedLogicTemplates` (2.5.8) tries to find occurences of `src_template` in a tree and potentially replaces them by `dst_template`.
- When trying to find an occurence of `src_template` in a tree, all strings occuring in the list `variable_names` are considered as variables, i.e., they match any value in the tree. If a variable occurs multiple times, the corresponding parts of the tree must be equal.
- `CapJitAppliedLogicTemplates` (2.5.8) uses *`jit_args`* to compute a concrete value of each variable. The template is only applied if all values match the corresponding filters in `variable_filters`.
- For each function in `dst_template`, `CapJitAppliedLogicTemplates` (2.5.8) tries to find a corresponding function in `src_template`. The functions are matched by comparing the lists of names of local variables. If for a function in `dst_template` no corresponding function in `src_template` exists, you have to add the list of names of local variables of this function to `new_funcs`.
- `returns_value` must be `true` if `src_template` defines an expression, `false` if it defines a statement.
- `needed_packages` has the same format as `NeededOtherPackages` in `PackageInfo.g`. The template is only evaluated if the packages in `needed_packages` are loaded in the correct versions.
- `debug` can be set to `true` to print more information while `CapJitAppliedLogicTemplates` (2.5.8) tries to apply the template. (Note: this causes informational break loops which are not actual errors).
- `debug_path` can be set to a specific path to get exact information why the subtree at this path does or does not match `src_template`.

Technical note: `src_template` is only replaced by `dst_template` if the result is well-defined, i.e., if all function variables reference only functions in their function stack. This can be used to move "static" variables (i.e. variables not depending on local variables) outside of functions. Example: consider a template with `src_template` given by `Sum( List( L, x -> x^2 * value ) )` and `dst_template` given by `Sum( List( L, x -> x^2 ) ) * value` (assuming distributivity). This replacement is only valid if `value` is independent of `x`. However, we do not need to make this explicit at any point, because if `value` depends on `x`, the result `Sum( List( L, x -> x^2 ) ) * value` is not well-defined, so the template is not applied anyway.

2.2 Enhanced syntax trees

To simplify the handling of syntax trees, the CAP compiler enhances syntax trees in the following ways:

- All node types starting with STAT_SEQ_STAT are replaced by STAT_SEQ_STAT.
- All node types starting with EXPR_FUNCCALL_ are replaced by EXPR_FUNCCALL.
- All node types starting with EXPR_PROCCALL_ are replaced by EXPR_PROCCALL.
- All node types starting with STAT_FOR are replaced by STAT_FOR.
- Nested STAT_SEQ_STATs are flattened.
- A final STAT_RETURN_VOID in functions is dropped.
- Branches of STAT_IF etc. are given the type BRANCH_IF.
- If the body of a BRANCH_IF is not a STAT_SEQ_STAT, the body is wrapped in a STAT_SEQ_STAT.
- The key-value pairs of EXPR_RECs are given the type REC_KEY_VALUE_PAIR.
- A globally unique ID is assigned to each function.
- The handling of local variables and higher variables is unified by the concept of function variables: function variables (FVARs) reference local variables in functions via the function id (`func_id`) and the position (`pos`) in the list of arguments/local variables of the function. For easier debugging, the name of the local variable is stored in the entry `initial_name` of the FVAR.

2.2.1 ENHANCED_SYNTAX_TREE

▷ `ENHANCED_SYNTAX_TREE(`*`func[, globalize_hvars]`*`)` (function)

Returns: a record

Returns an enhanced syntax tree of the plain function *`func`* (see above). If the second argument is set to `true`, higher variables pointing to variables in the environment of *`func`* are assigned to global variables and referenced via these global variables in the tree. Otherwise, an error is thrown if such higher variables exist.

2.2.2 ENHANCED_SYNTAX_TREE_CODE

▷ ENHANCED_SYNTAX_TREE_CODE(*tree*) (function)

Returns: a function

Converts the enhanced syntax tree *tree* to a function.

2.3 Iterating over a syntax tree

2.3.1 CapJitIterateOverTree

▷ CapJitIterateOverTree(*tree, pre_func, result_func, additional_arguments_func, additional_arguments*) (function)

Returns: see description

Iterates recursively over a syntax tree and calls *pre_func* and *result_func* for each node. Overview:

- *pre_func* allows to modify a (sub-)tree before the recursion over its children. For example, you could detect occurrences of `if true then <body> fi;` and simply return the body to simplify the tree.
- *result_func* allows to construct the return value of a (sub-)tree from the return values of its children. For example, if you want to check if a node of a certain type occurs in the tree, return `true` if *tree* has the type or any of the children returned `true`, otherwise return `false`.
- *additional_arguments[_func]* allows to create and pass additional data to the children of *tree*, for example the path or the function stack.

Details:

- First, *pre_func* is called with the following arguments: *tree* and *additional_arguments*. If it returns `fail`, the recursion is skipped and *result_func* is called immediately with `result` set to `fail` (see below). Otherwise it must return a syntax tree, which is then used as the value of *tree* for the remaining computation. If you do not need this function, use `ReturnFirst`.
- Secondly, for each child of *tree*, *additional_arguments_func* is called with the following arguments: *tree*, the key of the child, and *additional_arguments*. If *tree* is a list, the key of a child is its list index. If *tree* is a record, the key of a child is the corresponding record name of *tree*. The return value is used in the next step.
- Next, the recursion starts: for each child, `CapJitIterateOverTree` is called again with the following arguments: the child, *pre_func*, *result_func*, *additional_arguments_func*, and the return value of the call of *additional_arguments_func* in the step above.
- The results of the recursive calls are stored in the variable `result`: If *tree* is a list, `result` is also a list and the `i`-th entry of this list is the return value of the `result_func` of the `i`-th child. If *tree* is a record, `result` is also a record and `result.(key)` is the return value of the `result_func` of the child named `key`.
- Next, *result_func* is called with the following arguments: *tree*, `result`, and *additional_arguments*. The return value should be the result of the current tree formed by combining the results of the children. For an example see `CapJitResultFuncCombineChildren` (2.4.5).

- Finally, the return value of *result_func* is returned.

Note: This function on its own does not modify the tree. However, you can make modifications in *pre_func*, *result_func*, and *additional_arguments_func*. If you do not want to make these modifcations in-place, you can replace a (sub-)tree by a modified version in *pre_func* and combine the modified (sub-)trees again using `CapJitResultFuncCombineChildren` (2.4.5) as *result_func*.

2.4 Tools

2.4.1 CapJitGetFunctionCallArgumentsFromJitArgs

▷ `CapJitGetFunctionCallArgumentsFromJitArgs(tree, path, jit_args)` (function)

Returns: a list

Computes the arguments of the function call at the given *path* in *tree* when executing the function defined by *tree* with arguments *jit_args*. If the arguments cannot be determined, a list with first entry `false` is returned. Otherwise, a list with `true` as the first value and the computed arguments as the second value is returned.

2.4.2 CapJitGetExpressionValueFromJitArgs

▷ `CapJitGetExpressionValueFromJitArgs(tree, path, jit_args)` (function)

Returns: a list

Computes the value of the expression at the given *path* in *tree* when executing the function defined by *tree* with arguments *jit_args*. If the value cannot be determined, a list with first entry `false` is returned. Otherwise, a list with `true` as the first value and the computed value as the second value is returned.

2.4.3 CapJitPrettyPrintSyntaxTree

▷ `CapJitPrettyPrintSyntaxTree(tree)` (function)

Displays an enhanced syntax tree in a more useful way. For example, prints the type of a node on top.

2.4.4 CapJitIsCallToGlobalFunction

▷ `CapJitIsCallToGlobalFunction(tree, condition)` (function)

Returns: a boolean

Checks if *tree* is an EXPR_FUNCCALL with `funcref` EXPR_GVAR such that `gvar` fulfills *condition*. If *condition* is a string, `gvar` must be equal to the string. Othwerwise, *condition* must be a function returning a boolean when applied to `gvar`.

2.4.5 CapJitResultFuncCombineChildren

▷ `CapJitResultFuncCombineChildren(tree, result, additional_arguments)` (function)

Returns: a list or a record

Can be used as a `result_func` for `CapJitIterateOverTree` (2.3.1). Replaces *tree*.(key) (resp. *tree*[key]) by *result*.(key) (resp. *result*[key]) for all keys of children of *tree* and returns the result. See `CapJitIterateOverTree` (2.3.1) for more details.

2.4.6 CapJitContainsRefToFVAROutsideOfFuncStack

▷ `CapJitContainsRefToFVAROutsideOfFuncStack(tree)` (function)

Returns: a boolean

Checks if *tree* contains an FVAR which references a function outside of its function stack. Such a tree is not well-defined.

2.4.7 CapJitGetOrCreateGlobalVariable

▷ `CapJitGetOrCreateGlobalVariable(value)` (function)

Returns: a string

Assigns *value* to a global variable and returns the name of the global variable. If *value* has already been assigned to a global variable by this function before, simply returns the name of that global variable.

2.4.8 CapJitThrowErrorOnSideEffects

▷ `CapJitThrowErrorOnSideEffects(tree)` (function)

Checks if *tree* contains statements or expressions indicating side effects. If yes, it throws an error. The following checks are performed:

- *tree* must be an enhanced syntax tree. In particular, it may not contain LVARs or HVARs.
- The following statements and expressions are forbidden: STAT_ASS_GVAR, EXPR_ISB_GVAR, STAT_UNB_GVAR, EXPR_ISB_FVAR, EXPR_UNB_FVAR, STAT_PROCCALL.
- An FVAR must not reference functions outside of its function stack (see also `CapJitContainsRefToFVAROutsideOfFuncStack` (2.4.6)).
- An FVAR must be assigned at most once (this includes function arguments, which are assigned at least once, namely when the function is called). An exception is made for "rapid reassignments": if the same variable is assigned and then reassigned immediately in the next statement, this only counts as a single assignment.

2.4.9 CapJitFindNodeDeep

▷ `CapJitFindNodeDeep(tree, condition_func)` (function)

Returns: a record or fail

Finds a node in *tree* for which *condition_func* returns `true`. For each node, *condition_func* is called with the node and current path as arguments, and must return a boolean. If multiple nodes are found, children are preferred over their parents (i.e. a "deep" node is returned). If no node can be found, `fail` is returned.

2.4.10 CapJitGetNodeByPath

▷ `CapJitGetNodeByPath(tree, path)` (function)

Returns: a record

Gets the node of *tree* with path *path*. Throws an error if no such node exists.

2.5 Compilation steps

2.5.1 CapJitDroppedHandledEdgeCases

▷ `CapJitDroppedHandledEdgeCases(tree)` (function)

Returns: a record

Idea: If the same edge case is handled multiple times in the tree by checking a condition and returning a value, all condition checks except the first can be dropped. Details: Keeps a record of conditions which immediately lead to a return, i.e., statements of the form `if condition1 then return value; fi;`. If another statement of the form `if condition2 then return value; fi;` is found later in the tree and if `condition2 = true` implies `condition1 = true`, the second statement is dropped.

2.5.2 CapJitDroppedUnusedVariables

▷ `CapJitDroppedUnusedVariables(tree[, func_path])` (function)

Returns: a record

Drops assignments to local variables which are never referenced. If a path to a function is given as the second argument, only variables in this function are considered. Otherwise, all functions in *tree* are considered. Marks unused variables with the prefix _UNUSED_. Assumes that arguments of function calls are inlined (i.e., you should use `CapJitInlinedArguments` (2.5.3) first).

2.5.3 CapJitInlinedArguments

▷ `CapJitInlinedArguments(tree)` (function)

Returns: a record

Example: transforms `(function(x) return x; end)(1)` into `(function() local x; x := 1; return x; end)()`. Details: Searches for function calls of resolved functions. Assigns the argument values to local variables at the beginning of the function, and drops the arguments (i.e., makes the function a 0-ary function).

2.5.4 CapJitInlinedFunctionCalls

▷ `CapJitInlinedFunctionCalls(tree)` (function)

Returns: a record

Example: transforms `function() local x; x := (y -> y + 2)(1); return x; end` into `function() local x, y, r; y := 1; r := y + 2; x := r; return x; end`. Details: Searches for function calls of a resolved function in the right hand side of a variable assignment or a return statement. Inserts the body of the function right before the variable assignment / return statement to avoid the function call. Assumes that arguments of function calls are inlined (i.e., you should use `CapJitInlinedArguments` (2.5.3) first). Due to the nature of a `return` statement breaking the execution and having no `goto` keyword in GAP, only functions

- ending with a `return` statement, or
- ending with an if-(elif)-else-statement with `return` statements at the end of all branches

and not containing other `return` statements can be inlined.

2.5.5 CapJitInlinedSimpleFunctionCalls

▷ `CapJitInlinedSimpleFunctionCalls(tree)` (function)

Returns: a record

Replaces function calls of the form `(function() return value; end)()` by `value`. Assumes that arguments of function calls are inlined (i.e., you should use `CapJitInlinedArguments` (2.5.3) first).

2.5.6 CapJitInlinedVariableAssignments

▷ `CapJitInlinedVariableAssignments(tree[, inline_gvars_only])` (function)

Returns: a record

Example: transforms `function() local x; x := 1; return x^2; end` into `function() return 1^2; end()`. Details: Searches for local variable assigments. Replace all references to the local variable in the same STAT_SEQ_STAT as the assignment by the right-hand side of the assignment. If the second argument is set to `true`, this is only done if the right-hand side is a reference to a global variable. Assumes that any local variable is assigned at most once (this includes function arguments, which are assigned at least once, namely when the function is called). An exception is made for "rapid reassignments": if the same variable is assigned and then reassigned immediately in the next statement, the right-hand side of the first assignment is inserted into the right-hand side of the second assignment. Assumes that unused variables are dropped (i.e., you should use `CapJitDroppedUnusedVariables` (2.5.2) first). Drops the variables assignment after inlining if possible.

2.5.7 CapJitAppliedLogic

▷ `CapJitAppliedLogic(tree, jit_args)` (function)

Returns: a record

Applies all logic functions (see `CapJitAddLogicFunction` (2.1.1)) and logic templates (see `CapJitAppliedLogicTemplates` (2.5.8)) to *tree*.

2.5.8 CapJitAppliedLogicTemplates

▷ `CapJitAppliedLogicTemplates(tree, jit_args[, cleanup_only])` (function)

Returns: a record

Applies all logic templates (see `CapJitAddLogicTemplate` (2.1.2)) to *tree*. The arguments *jit_args* are used to infer the types of variables. If the third argument is set to `true`, only templates with empty `dst_template` are applied. This can be used to quickly drop unwanted statements from the tree without applying all (possibly expensive) logic templates.

2.5.9 CapJitResolvedGlobalVariables

▷ `CapJitResolvedGlobalVariables(tree)` (function)

Returns: a record

Resolves global variables:

- Replaces a global variable referencing an integer, a string, or a boolean by EXPR_INT, EXPR_STRING, EXPR_TRUE or EXPR_FALSE.
- Replaces a global variable referencing a plain function by the syntax tree of this function.
- Replaces a record access of a global function by the value of this record access.

2.5.10 CapJitResolvedOperations

▷ `CapJitResolvedOperations(tree, jit_args)` (function)

Returns: a record

Tries to resolve operations in *tree*:

- The attribute `CapCategory` is resolved by computing the category using *jit_args* and storing it in a global variable.
- Operations of CAP categories are resolved by taking one of the functions added to the category via an `Add` function.
- Other operations are resolved by considering applicable methods of the operation with regard to arguments infered from *jit_args*. Only methods annotated with the pragma CAP_JIT_RESOLVE_FUNCTION are resolved.

If the arguments of the operation cannot be infered from *jit_args*, the operation is not resolved.

Chapter 3

Examples and tests

3.1 Examples

Example

```
gap> package_loading_info_level := InfoLevel( InfoPackageLoading );;
gap> debug_info_level := InfoLevel( InfoDebug );;
gap> SetInfoLevel( InfoPackageLoading, PACKAGE_ERROR );;
gap> SetInfoLevel( InfoDebug, 0 );;
gap> LoadPackage( "LinearAlgebraForCAP", false );;
gap> LoadPackage( "RingsForHomalg", false );;
gap> SetInfoLevel( InfoPackageLoading, package_loading_info_level );;
gap> SetInfoLevel( InfoDebug, debug_info_level );;
gap> Q := HomalgFieldOfRationals();;
gap> vec := MatrixCategory( Q :
>     enable_compilation := [ "MorphismBetweenDirectSums" ]
> );;
gap> V := VectorSpaceObject( 2, Q );;
gap> alpha := ZeroMorphism( V, V );;
gap> beta := IdentityMorphism( V );;
gap> W := DirectSum( V, V );;
gap> morphism_matrix := [ [ alpha, beta ], [ beta, alpha ] ];;
gap> # compile the primitive installation of MorphismBetweenDirectSums
> MorphismBetweenDirectSums( morphism_matrix );;
gap> tree1 := SYNTAX_TREE(
>     vec!.compiled_functions.MorphismBetweenDirectSums[3]
> );;
gap> # fixup nams
> tree1.nams := [ "S", "morphism_matrix", "T" ];;
gap> tree1.nloc := 0;;
gap> tree1.stats.statements[1].branches[2].body.statements[1].
>     obj.args[12].args[1].args[2].nams := [ "row" ];;
gap> tree1.stats.statements[1].branches[2].body.statements[1].
>     obj.args[12].args[1].args[2].nloc := 0;;
gap> Display( SYNTAX_TREE_CODE( tree1 ) );
function ( S, morphism_matrix, T )
    if morphism_matrix = [  ] or morphism_matrix[1] = [  ] then
        return ZeroMorphism( S, T );
    else
        return ObjectifyWithAttributes( rec(
```

```
            ), CAP_JIT_INTERNAL_GLOBAL_VARIABLE_3, CapCategory,
          CAP_JIT_INTERNAL_GLOBAL_VARIABLE_1, Source, S, Range, T,
          UnderlyingFieldForHomalg, CAP_JIT_INTERNAL_GLOBAL_VARIABLE_2,
          UnderlyingMatrix,
          UnionOfRows( List( morphism_matrix, function ( row )
                return UnionOfColumns( List( row, UnderlyingMatrix ) );
            end ) ) );
    fi;
    return;
end
gap> # compile the default derivation of MorphismBetweenDirectSums
> tree2 := SYNTAX_TREE( CapJitCompiledFunction(
>     vec!.added_functions.MorphismBetweenDirectSums[1][1],
>     [ W, morphism_matrix, W  ]
> ) );;
gap> # fixup nams
> tree2.nams := [ "S", "morphism_matrix", "T" ];;
gap> tree2.nloc := 0;;
gap> tree2.stats.statements[1].branches[2].body.statements[1].
>     obj.args[12].args[1].args[2].nams := [ "row" ];;
gap> tree2.stats.statements[1].branches[2].body.statements[1].
>     obj.args[12].args[1].args[2].nloc := 0;;
gap> Display( SYNTAX_TREE_CODE( tree2 ) );
function ( S, morphism_matrix, T )
    if morphism_matrix = [  ] or morphism_matrix[1] = [  ] then
        return ZeroMorphism( S, T );
    else
        return ObjectifyWithAttributes( rec(
            ), CAP_JIT_INTERNAL_GLOBAL_VARIABLE_3, CapCategory,
          CAP_JIT_INTERNAL_GLOBAL_VARIABLE_1, Source, S, Range, T,
          UnderlyingFieldForHomalg, CAP_JIT_INTERNAL_GLOBAL_VARIABLE_2,
          UnderlyingMatrix,
          UnionOfRows( List( morphism_matrix, function ( row )
                return UnionOfColumns( List( row, function ( s )
                          return UnderlyingMatrix( s );
                      end ) );
            end ) ) );
    fi;
    return;
end
```

3.2 Tests

Index