The TERMITE library

Adrian Prantl
Institut für Computersprachen
Technische Universität Wien
E-Mail: adrian@complang.tuwien.ac.at

January 11, 2012

Contents

1 Introduction

The TERM Iteration and Transformation Environment (Termite) is a Prolog library that allows easy manipulation and analysis of C++ programs. It is particularly well suited to specify source-to-source program transformations, static program analyses and program visualizations. Termite builds upon the intermediate representation of SATIrE. More information on SATIrE can be found at `http://www.complang.tuwien.ac.at/satire`.

1.1 Using Termite

Depending on the desired architecture there are several ways to integrate Termite into the work flow of a larger tool. For a flexible recombination of several analyses and/or transformations it is best to treat Termite programs as interpreted scripts that read/write AST terms from the standard input an output. If performance and stability are sought for, it is also possible to call Termite programs transparently from a SATIrE analyzer.

1.1.1 Using Termite for a standalone process

The most flexible and convenient way to work with the Termite library is by using it to define filter operations on streams of source code. This way one can follow the UNIX tradition of having a collection of small self-contained programs that can be combined to create larger work flows. Depending on the expected input and generated output, several types of Termite programs can be distinguished. Typical examples are:

A source-to-source transformer is a program that reads in an AST, then performs some transformation and outputs the transformed AST. (Example: loop unrolling)

An analyzer is a program that reads in an AST, performs some analysis and outputs the analysis result as attributes of the AST. (Example: loop bound analysis)

A visualization is a program that reads in an AST and outputs a visualization, e. g., in a GUI window or a PostScript file. (Example: Call-graph → Graphviz (DOT))

A source generator is a program that reads in a specification and outputs an AST in termite format.

A compiler is a program that translates an AST into a different language, e. g., melmac[1] or wcetcc.

In order to generate a Termite term from one or more source files a compiler front end must be invoked. Two possibilities are supported and available in the SATIrE distribution:

EDG C/C++ front end from the ROSE compiler

If SATIrE was configured with the ROSE connection enabled[2], conversion tools are available to translate source code to term files and vice versa. To translate source code into a Termite term the program `c2term` is available:

[1] `http://www.complang.tuwien.ac.at/gergo/melmac/`
[2] ROSE must be installed separately beforehand and is available at `http://www.rosecompiler.org`

> **c2term**

```
Usage: c2term [FRONTEND OPTIONS] [--dot] [--pdf] src1.c src2.cpp ... [-o termfile.pl]
  Parse one or more source files and convert them into a TERMITE file.
  Header files will be included in the term representation.

Options:
  [FRONTENT OPTIONS] will be passed to the C/C++ frontend.

  --rose-help
    Display the help for the C/C++ frontend.

  -o, --output <termfile.pl>
    Write the output to <termifile.pl> instead of stdout.

  --dot
    Create a dotty graph of the syntax tree.

  --pdf
    Create a PDF printout of the syntax tree.

This program was built against SATIrE 0.9.0,
please report bugs to <schordan@technikum-wien.at or adrian@llnl.gov>.
```

The `c2term` program invokes the commercial EDG C++ front end embedded into the ROSE compiler to parse one or more source files. The abstract syntax tree (AST) is then translated into the ROSE immediate representation which in turn is converted into the textual term serialization. The program passes additional options to the EDG front end.

The opposite direction is managed by the `term2c` conversion utility. It works by reading in a term file and then rebuilding the ROSE intermediate representation. Finally, this data structure is passed to the ROSE unparser. The EDG front end is not involved in this step any more.

> **term2c**

```
Usage: term2c [OPTION]... [FILE.term]
Unparse a term file to its original source representation.

Options:
  -o, --output sourcefile.c
    If specified, the contents of all files will be concatenated
    into the sourcefile.

  -s, --suffix '.suffix'  Default: '.unparsed'
    Use the original file names with the additional suffix.

  -d, --dir DIRECTORY
    Create the unparsed files in DIRECTORY.

  --dot
    Create a dotty graph of the syntax tree.

  --pdf
    Create a PDF printout of the syntax tree.

This program was built against SATIrE 0.9.0,
```

```
please report bugs to <schordan@technikum-wien.at or adrian@llnl.gov>.
```

Since both converters use standard input and output per default it is possible to concatenate multiple Termite programs with the help of UNIX pipes. This way it is possible to build new chains of program transformations or analyzers on the fly without having to recompile the whole project.

Example:

```
c2term a.c b.c | ./transform1.pl | term2c -s '.transformed'
```

In this example pipeline, two C source file are joined into one project which is dumped to a stream in the Termite format. The stream is then transformed by a Prolog program. Finally the two source files are unparsed by the `term2c` converter with the new suffix ".transformed" attached to the file names.

Using the Clang C/Objective C front end

While the commercial EDG front end offers a high-quality C++ parser, license restrictions encumber its free distribution together with other tools. Most notably, the ROSE compiler redistributes only a 32-bit precompiled binary version of the EDG front end. It is, however, possible to buy other licenses from the Edison Design Group.

If C++ support is not needed, there is a free alternative available from the LLVM compiler project. Designed especially for use with LLVM a front end for C-like languages called `clang` is published under a BSD-style license. The `clang` front end can be downloaded at `http://clang.llvm.org/`. The front end is written in C++ and creates an intermediate representation very similar to that of ROSE and therefore makes a good candidate to replace the EDG front end in SATIrE. The C99 and Objective C languages are supported very well by `clang`, whereas C++ support is still under development.

In order to connect SATIrE with the `clang` front end, we decided to take the route via the Termite representation. This way, the front end is cleanly decoupled from the rest of the system and uses the Termite terms as a stable interface. The Termite term generator is implemented as a pass over the `clang` intermediate representation and is available via the `-emit-term` command line option. The term generator is not integrated with upstream `clang`, but distributed as a patch against a current SVN version together with SATIrE.

To build the `clang` front end for use with SATIrE a special `make clang` target is available at the toplevel which fetches the needed version of `clang` from the subversion repository, applies the patch, and compiles and installs the patched front end to `$prefix/bin`.

Uparsing Termite terms without SATIrE

Invoking the `term2c` program is sometimes too cumbersome, for example, when only a few expressions should be unparsed for debugging purposes. For these occasions an independent term→C converter is implemented in pure Prolog and available both in the Termite library and as a standalone script. The predicate is called `unparse/1` and expects a Termite term as argument.

1.1.2 As part of a SATIrE analyzer

If execution speed is an issue, the steps of writing the Termite representation to disk (or a pipe) and parsing the terms (which, when output as a text, are significantly larger than the original source files) can be optimized away. If SATIrE was configured with SWI-Prolog support enabled, the term representation will be built in memory using the external interface of an embedded SWI-Prolog interpreter. Using this in-memory term, a Termite program can be executed without leaving the current process. The resulting term can again be translated to the ROSE intermediate representation directly from memory using the SWI-Prolog interface.

Using this work flow, the whole analyzer (or transformer, ...) can be distributed as a single self-contained executable.

The Termite term representation

2

SATIrE can export an external term representation of the abstract syntax tree (AST) of a C++ program. This term representation contains all information that is necessary to correctly unparse the program, including line and column information of every expression. The terms are also annotated with the results of any preceding PAG analysis. The syntax of the term representation was designed to match the syntax of Prolog terms. This allows it to be manipulated by Prolog programs very naturally.

2.1 Grammar of TERMITE terms

The following section gives a formal definition of the grammar of Termite terms as it is used by the program `termite_lint` which can be used to verify the validity of arbitrary terms.

```
termite ::=
    project.

project ::=
    project([source_file], default_annotation, analysis_info, file_info).

source_file ::=
    source_file(global, default_annotation, analysis_info, file_info).

initialized_name ::=
    initialized_name(initializer?, initialized_name_annotation,
                     analysis_info, file_info).
```

2.1.1 statements

```
statement ::=
    break_stmt
  | case_option_stmt
  | continue_stmt
  | declaration_statement
  | default_option_stmt
  | expr_statement
  | for_init_statement
  | goto_statement
  | label_statement
  | null_statement
  | return_stmt
  | scope_statement.

break_stmt ::=
    break_stmt(default_annotation, analysis_info, file_info).
```

```
case_option_stmt ::=
    case_option_stmt(expression, statement, expression? /* key_range_end */,
                     default_annotation, analysis_info, file_info).

continue_stmt ::=
    continue_stmt(default_annotation, analysis_info, file_info).

declaration_statement ::=
    class_declaration
  | enum_declaration
  | function_declaration
  | function_parameter_list
  | pragma_declaration
  | program_header_statement
  | typedef_declaration
  | variable_declaration
  | variable_definition.

class_declaration ::=
    class_declaration(class_definition?, class_declaration_annotation,
                      analysis_info, file_info).

enum_declaration ::=
    enum_declaration([initialized_name], enum_declaration_annotation,
                     analysis_info, file_info).

function_declaration ::=
    function_declaration(function_parameter_list, {null}, function_definition?,
                         function_declaration_annotation,
                         analysis_info, file_info).

function_parameter_list ::=
    function_parameter_list([initialized_name],
                            default_annotation, analysis_info, file_info).

program_header_statement ::=
    program_header_statement(function_parameter_list, {null}, function_definition?,
     function_declaration_annotation,
     analysis_info, file_info).

pragma_declaration ::=
    pragma_declaration(todo).

typedef_declaration ::=
    typedef_declaration(declaration_statement? /* base type definition */,
                        typedef_annotation, analysis_info, file_info).

variable_declaration ::=
    variable_declaration([initialized_name], variable_declaration_specific,
                         analysis_info, file_info).

variable_definition ::=
    variable_definition(todo).

default_option_stmt ::=
    default_option_stmt(statement,
                        default_annotation, analysis_info, file_info).
```

```
expr_statement ::=
    expr_statement(expression, default_annotation, analysis_info, file_info).

for_init_statement ::=
    for_init_statement([statement],
                       default_annotation, analysis_info, file_info).

goto_statement ::=
    goto_statement(label_annotation, analysis_info, file_info).

label_statement ::=
    label_statement(label_annotation, analysis_info, file_info).

null_statement ::=
    null_statement(default_annotation, analysis_info, file_info). /* really? */

return_stmt ::=
    return_stmt(expression, default_annotation, analysis_info, file_info).

scope_statement ::=
    basic_block
  | class_definition
  | do_while_stmt
  | for_statement
  | function_definition
  | global
  | if_stmt
  | switch_statement
  | while_stmt.

basic_block ::=
    basic_block([statement], default_annotation, analysis_info, file_info).

class_definition ::=
    class_definition([variable_declaration], class_definition_annotation,
                     analysis_info, file_info).

do_while_stmt ::=
    do_while_stmt(statement /* body */, statement /* condition */,
                  default_annotation, analysis_info, file_info).

for_statement ::=
    for_statement(for_init_statement, statement /* test */,
                  expression /* increment */, statement /* body */,
                  default_annotation, analysis_info, file_info).

function_definition ::=
    function_definition(basic_block,
                        default_annotation, analysis_info, file_info).

global ::=
    global([declaration_statement],
           default_annotation, analysis_info, file_info).

if_stmt ::=
    if_stmt(statement /* condition */, statement /* true */,
            statement? /* else */,
            default_annotation, analysis_info, file_info).
```

```
switch_statement ::=
    switch_statement(statement /* key */, statement /* body */,
                     default_annotation, analysis_info, file_info).

while_stmt ::=
    while_stmt(statement /* condition */, statement /* body */,
               default_annotation, analysis_info, file_info).
```

2.1.2 expressions

```
expression ::=
    binary_op
  | cast_exp(expression, /*expression? * original expression tree ,*/
             unary_op_annotation, analysis_info, file_info)
  | conditional_exp
  | expr_list_exp
  | function_call_exp
  | function_ref_exp
  | initializer
  | new_exp
  | null_expression
  | size_of_op
  | unary_op
  | var_arg_copy_op
  | var_arg_end_op
  | var_arg_op
  | var_arg_start_one_operand_op
  | var_arg_start_op
  | var_ref_exp
  | functors [long_long_int_val, unsigned_long_long_int_val, long_int_val,
        unsigned_long_val, int_val, unsigned_int_val, short_val,
        unsigned_short_val, char_val, unsigned_char_val, float_val,
        double_val, long_double_val, string_val, enum_val]
    with (/*expression?  original expression tree ,*/
          value_annotation, analysis_info, file_info).

binary_op ::=
    functors [add_op, and_assign_op, and_op, arrow_exp, assign_op,
        bit_and_op, bit_or_op, bit_xor_op, comma_op_exp, div_assign_op,
        divide_op, dot_exp, equality_op, greater_or_equal_op,
        greater_than_op, ior_assign_op, less_or_equal_op, less_than_op,
        lshift_assign_op, lshift_op, minus_assign_op, mod_assign_op, mod_op,
        mult_assign_op, multiply_op, not_equal_op, or_op, plus_assign_op,
        pntr_arr_ref_exp, rshift_assign_op, rshift_op, subtract_op,
        xor_assign_op]
    with (expression /* lhs */, expression /* rhs */,
          binary_op_annotation, analysis_info, file_info).

conditional_exp ::=
    conditional_exp(expression /* condition */,
                    expression /* true */, expression /* false */,
                    conditional_exp_annotation, analysis_info, file_info).

expr_list_exp ::=
    expr_list_exp([expression], default_annotation, analysis_info, file_info).
```

```
function_call_exp ::=
    function_call_exp(expression /* function */, expr_list_exp /* args */,
                      function_call_exp_annotation, analysis_info, file_info).

function_ref_exp ::=
    function_ref_exp(function_ref_exp_annotation, analysis_info, file_info).

new_exp ::=
    new_exp({null}, constructor_initializer?, {null},
    new_exp_annotation, analysis_info, file_info).

new_exp_annotation ::=
    new_exp_annotation(type, preprocessing_info).

constructor_initializer ::=
    constructor_initializer(expr_list_exp,
    constructor_initializer_annotation,
    analysis_info, file_info).

constructor_initializer_annotation ::=
    constructor_initializer_annotation(name, type,
       name, name, name, name,
       preprocessing_info).

initializer ::=
    aggregate_initializer
  | assign_initializer.

aggregate_initializer ::=
    aggregate_initializer(expr_list_exp,
                          default_annotation, analysis_info, file_info).

assign_initializer ::=
    assign_initializer(expression, assign_initializer_annotation,
                       analysis_info, file_info).

null_expression ::=
    null_expression(default_annotation, analysis_info, file_info).

size_of_op ::=
    size_of_op(expression?, size_of_op_annotation, analysis_info, file_info).

unary_op ::=
    functors [address_of_op, bit_complement_op, minus_minus_op,
        minus_op, not_op, plus_plus_op, pointer_deref_exp, unary_add_op]
    with (expression, unary_op_annotation, analysis_info, file_info).

var_arg_copy_op ::=
    var_arg_copy_op(todo).

var_arg_end_op ::=
    var_arg_end_op(todo).

var_arg_op ::=
    var_arg_op(todo).

var_arg_start_one_operand_op ::=
    var_arg_start_one_operand_op(todo).
```

```
var_arg_start_op ::=
    var_arg_start_op(todo).

var_ref_exp ::=
    var_ref_exp(var_ref_exp_annotation, analysis_info, file_info).
```

2.1.3 annotations

```
default_annotation ::=
    default_annotation({null})
  | default_annotation({null}, preprocessing_info).

initialized_name_annotation ::=
    initialized_name_annotation(type, name, todo /* storage modifier */,
                                scope_name?, preprocessing_info).

function_declaration_annotation ::=
    function_declaration_annotation(type, name, declaration_modifier,
                                    preprocessing_info).

class_declaration_annotation ::=
    class_declaration_annotation(name, todo /* class kind */, type,
                                 preprocessing_info).

enum_declaration_annotation ::=
    enum_declaration_annotation(name, todo, todo, preprocessing_info).

class_definition_annotation ::=
    class_definition_annotation(file_info, preprocessing_info).

variable_declaration_specific ::=
    variable_declaration_specific(todo /* declaration modifier */,
                                  declaration_statement? /* base type decl */,
                                  preprocessing_info).

label_annotation ::=
    label_annotation(name, preprocessing_info).

size_of_op_annotation ::=
    size_of_op_annotation(type? /* operand */, type /* sizeof expression */,
                          preprocessing_info).

value_annotation ::=
    value_annotation(number_or_string, name, type, preprocessing_info) /* enum */
    | value_annotation(number_or_string, preprocessing_info).

binary_op_annotation ::=
    binary_op_annotation(type, preprocessing_info).

unary_op_annotation ::=
    unary_op_annotation(fixity, type, todo /* cast type */,
                        todo /* throw kind */, preprocessing_info).

var_ref_exp_annotation ::=
    var_ref_exp_annotation(type, name, todo /* storage modifier */,
                           scope_name?, preprocessing_info).
```

```
typedef_annotation ::=
    typedef_annotation(name, type, preprocessing_info).

function_ref_exp_annotation ::=
    function_ref_exp_annotation(name, type, preprocessing_info).

function_call_exp_annotation ::=
    function_call_exp_annotation(type, preprocessing_info).

assign_initializer_annotation ::=
    assign_initializer_annotation(type, preprocessing_info).

conditional_exp_annotation ::=
    conditional_exp_annotation(type, preprocessing_info).
```

2.1.4 other stuff

```
analysis_info ::=
    analysis_info([_]).

file_info ::=
    file_info({_}, {_}, {_}).

preprocessing_info ::=
    preprocessing_info([_]).

type ::=
    basic_type
  | array_type(type, expression?)
  | function_type(type /* return */, todo /* ellipses */, [type] /* args */)
  | modifier_type(type, type_modifier)
  | named_type
  | type_default
  | pointer_type(type).

basic_type ::=
    atoms [type_bool, type_char, type_double, type_ellipse, type_float,
        type_int, type_long, type_long_double, type_long_long, type_short,
        type_signed_char, type_string, type_unsigned_char,
        type_unsigned_int, type_unsigned_long, type_unsigned_long_long,
        type_unsigned_short, type_void].

type_default ::= atoms [type_default].

named_type ::=
    class_type(name, todo, todo)
  | enum_type(todo)
  | typedef_type(name, type).

type_modifier ::=
    type_modifier([todo], todo, todo, todo).

name ::=
    {Name} where atom(Name).
```

```
scope_name ::=  % name of a scope
  {::}
  | class_scope(name, class_kind, preprocessing_info)
  | name.

class_kind ::=
    {class}
  | {struct}
  | {union}.

number_or_string ::=
    {It} where ( numberatom(It) ; number(It) ; string(It) ; atom(It) ).

fixity ::=  % fixity of unary operators
    {prefix}
  | {postfix}.

declaration_modifier ::=
    declaration_modifier(todo, todo, todo, todo).

todo ::=
    {_}.
```

For an optional argument A? , this predicate is tried first.

```
missing(null).
```

Test whether the atom A can be interpreted as a number.

```
numberatom(A) :-
    atom(A),
    catch(atom_number(A, _N), _, fail).
```

Library Reference

3

3.1 asttransform.pl – Properties of abstract syntax trees

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

copyright Copyright (C) 2007-2010 Adrian Prantl

license See COPYING in the root folder of the SATIrE project

This module defines commonly-used transformation utilities for C/C++/Objective C ASTs given in the TERMITE term representation as exported by SATIrE.

simple_form_of(*?Term, ?SimpleTerm***)**

This function is obsoleted, since SATIrE>0.7 defaults to the compact representation.

`simple_form_of/2` is used to convert the verbose *nary_node() terms to a more compact representation

Example:

```
unary_node(int_val, foo ...) <-> int_val(foo, ...)
```

[(*o***)**

ndet]ast_node6?Node, ?Type, ?Children, ?Annot, ?Ai, ?Fi `ast_node/6` (de)construct an AST node

Since all AST nodes follow the same structure, this predicate can be used to quickly compose or decompse a node.

[(*e***)**

t]is_ast_node6+Node, -Type, -Children, -Annot, -Ai, -Fi Faster, uni-directional version of `ast_node/6`.

transformed_with(*+Node, +Transformation, +Info, -Info1, -NodeT***)**

Backwards compatibile version of `transformed_with/5`:

- if it is used with arity 4, default to preorder

collate_ast(*Map, Reduce, A0, Node, A***)**

Perform a postorder traversal on the AST *Node*. For every *Node* visited call *Map* = f(*Node*, A_Children, *A*). For child nodes in the same hierarchy call foldl1(As, *Reduce*, *A*).

[(*e***)**

t]unparse1+Term This predicate prints the original textual (source code) representation of the program encoded in *Term*. Output is written on stdout.

This predicate is especially useful for debugging purposes.

[(*e***)**

t]needs_semicolon1+Node Succeeds if *Node* needs a semicolon ';' after itself during unparsing.

[(*e*)

t]needs_comma1+Node Succeeds if *Node* needs a comma ',' after itself during unparsing.

[(*e*)

t]replace_types3+InitializedNames, +FuncDecl, -InitializedNames1 Replace the instantiated types with the original types from the function declaration.

Needed during unparsing.

[(*e*)

t]indent1+FileInfo Output the indentation that is encoded in *FileInfo*.

[(*e*)

t]unparse_ppi2+Location, +PPIs Print all preprocessing information(s) *PPIs* at *Location*.

Location must be one of [before, after, inside].

3.2 astproperties.pl – Properties of abstract syntax trees

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

copyright Copyright (C) 2008-2009 Adrian Prantl

license See COPYING in the root folder of the SATIrE project

This module defines commonly-used queries about C/C++/Objective C ASTs given in the TERMITE term representation as exported by SATIrE.

Depenedencies:

The user has to define the type predicates [`type_info/3`, `type_interval/2`]

[(*e***)**

midet]ast_equiv2+Expr1, +Expr2 Compare two expressions disregarding the file information Expects compact form.

Todo: rewrite this!

is_transp(*+Expr, +Var, +Scope***)**

Goal succeeds if *Var* is not written to by *Expr*.

- *Expr* must be an Expression.
- *Scope* specifies whether *Var* is global or local.

is_complex_statement(*+Node***)**

Goal succeeds if *Node* introduces new edges into the control flow graph (CFG).

guarantee(*+Node, +Pred***)**
guarantee(*+List, +Pred***)**

Recursively test a predicate *Pred* on an AST *Node* or *List* of AST Nodes, respectively.

strip_file_info(*_, _, _, +Term1, -Term2***)**

Replace file_info(...) with null in all VarRefExps. This facilitates the comparison of AST nodes.

Use this with `transformed_with/4`

var_stripped(*+VarRefExp, -VarRefExpStripped***)**

Non-traversal version of `strip_file_info/5`.

get_variable_id(*+AnalysisInfo, -Id***)**

Extract the numerical variable *Id* from the Analysis Information

var_interval(*+AnalysisInfo, +VarRefExp, -Interval***)**

Employ the analysis result/type info to yield an interval for the *VarRefExp*.

term_stripped(*+Term, -StrippedTerm***)**

Recursively strip all VarRefExps in *Term*

[(*o***)**

ned]isIntVal2?IntVal, ?Value Convert between int_val nodes and integer values.

new_intval(*+Value, -IntVal***)**

Create a new int_val(*Value*, ...) data structure with default annotations.

[(*o***)**

ndet]isVar2?VarRefExp, ?Name True if *VarRefExp* is a var_ref_exp or a cast_exp. *Name* is the name of the variable.

[(*o*)
ndet]var_type2?VarRefExp, ?Type Allows access to the *Type* of *VarRefExp*.

[(*o*)
ndet]var_typemod2?VarRefExp, ?ConstVolatile Allows access to the *ConstVolatile* modifier of *VarRefExp*. Values for *ConstVolatile* are 'const' and 'volatil' (sic!).

[(*e*)
midet]isBinNode7+Node, -Name, -E1, -E2, -Annot, -Ai, -Fi Decompose a binary node *Node*. % isBinNode(-*Node*, +*Name*, +*E1*, +*E2*, +*Annot*, +*Ai*, +*Fi*) is det. Compose a binary node *Node*.

isBinOpLhs(*?BinOp, ?Lhs***)**
Bind *Lhs* to the left-hand-side (1) operator of *BinOp*.

isBinOpRhs(*?BinOp, ?Rhs***)**
Bind *Rhs* to the right-hand-side (2) operator of *BinOp*.

scope_statement(*+Node***)**
True, if *Node* is a scope statement.

> Scope statements are basic_block, catch_option_stmt, class_definition, do_while_stmt, for_statement,function_definition, global, if_stmt, namespace_definition_statment, switch_statement, while_stmt

analysis_info(*+Term, -Ai***)**
Extract the analysis info *Ai* from *Term*.

file_info(*+Term, -Fi***)**
Extract the file info *Fi* from *Term*.

function_signature(*?FunctionDecl, ?Type, ?Name, ?Modifier***)**
Convert between signatures and terms.

is_function_call(*?Term, ?Name, ?Type***)**
(De-)construct a function call.

is_function_call_exp(*?Term, -Name, -Type***)**
(De-)construct a function call expression.

function_body(*?FuncDecl, ?Body***)**
Get the function body *Body* from a function declaration *FuncDecl*.

pragma_text(*?Pragma, ?Text***)**
`pragma_text/2` defines the relation between a pragma statement and the String *Text* inside the "#pragma *Text*" Statement.

get_annot(*+Stmts, -Annotterm, -Pragma***)**
Find *Pragma* in *Stmts* and treat its contents as a Prolog term. If successuful, unify Annot-Term with the parsed term.

get_annot_term(*?Stmts, ?Annotterm, ?Pragma***)**
Quicker version of `get_annot/3` without term parsing.

get_preprocessing_infos(*+Node, -PPIs***)**
Todo: move to `annot.pl`!

type_interval(*+Type, -Interval***)**
Use the user-defined `type_info/2` to return an *Interval* (Min..Max) denoting the maximum value range of *Type*

3.3 astwalk.pl – Flexible traversals of abstract syntax trees

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

copyright Copyright (C) 2008-2009 Adrian Prantl

license See COPYING in the root folder of the SATIrE project

This module defines commonly-used transformation utilities for the AST exported by SATIrE. It represents an more flexible alternative to the transformation interface provided by module ast_transform.

[(*e***)**

t]zip2+Tree, -Zipper Creates a new *Zipper* from *Tree*.

How to move around a tree and replace subtrees=branches?

At each node, we cut out a branch and replace it with a free variable <Gap>. The original branch is given as a separate argument, allowing us to bind the new branch to <Gap>.

In a way, this works just like Huet's zipper!

[(*e***)**

t]unzip3?Zipper, ?Tree, ?Context Converts between the *Zipper* data structure and its contents.

[(*e***)**

midet]walk_to3+Zipper, +Context, -Zipper1

[(*e***)**

midet]down3+Zipper, +BranchNum, -Zipper1 Navigate downwards in the tree to child *#BranchNum*.

- Works also with lists.

[(*e***)**

midet]up2+Zipper, -Zipper1 Navigate upwards in the tree.

[(*e***)**

midet]right2+Zipper, -Zipper1 Navigate to the next sibling in a tree or a list.

[(*e***)**

midet]top2+Zipper, -Zipper1 Navigate back to the root of our tree.

To be done Could be implemented more efficiently, too

[(*o***)**

ndet]goto_function3+Zipper, ?Template, +Zipper1 find a function like *Template* in a project or file and return its body if there is only a declaration available, the declaration will be returned

[(*e***)**

t]distance_from_root2+Zipper, -Distance Return the current distance from the root node

next_preorder(*+Zipper, -Zipper***)**

To be done change name to next_tdlr Return the "next" node in a left-to-right traversal fashion. Algorithm: (rechtssucher) try down(1) else while not try right() do up()

3.4 callgraph.pl – Create a call graph from an AST

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

license See COPYING in the root folder of the SATIrE project

[(*e***)**

t]callgraph2+P, -Graph Construct a call graph from an AST. *Graph* is in library(ugraphs) form. The nodes in the graph have the form Name/Type

To be done NO function pointers or virtual methods yet!

3.5 utils.pl – A collection of useful general-purpose predicates.

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

copyright Copyright (C) 2007-2009 Adrian Prantl

license See COPYING in the root folder of the SATIrE project

The predicates `drop/3`, `foldl/4`, `foldl1/3`, `last/2`, `replicate/3`, `split_at/4` and `take/3` are inspired by the Haskell Prelude, but are implemented declaratively: They can be used to generate as well as test.

drop(*?N, ?List, ?Tail***)**
Drop *N* elements from *List*, yielding *Tail*.

```
drop(N, List, Tail) :-
  length(Head, N),
  append(Head, Tail, List).
```

last(*?List, ?Elem***)**
Elem is the last element of *List*.

```
last(List, Elem) :-
  reverse(List, [Elem|_]).
```

replicate(*?A, ?Num, ?As***)**
Replicate *A Num* times yielding *As*.

```
replicate(A, Num, As) :-
  length(As, Num),
  maplist(=(A),As).
```

split_at(*?N, ?List, ?Head, ?Tail***)**
Split *List* at element *N* yielding *Head*, *Tail*

```
split_at(N, List, Head, Tail) :-
  length(Head, N),
  append(Head, Tail, List).
```

take(*?N, ?List, ?Head***)**
Head is unified with the first *N* elements of *List*

```
take(N, List, Head) :-
  length(Head, N),
  append(Head, _Tail, List).
```

foldl1(*?List, ?Pred, ?Result***)**
Fold *List* left-to-right using *Pred*, starting with the first element of *List*.

foldl(*?List, ?Pred, ?Start, ?Result***)**
Fold a list left-to-right using *Pred*, just as you would do in Haskell.

```
pred(LHS, RHS, Result)
```

Thanks to Markus Triska for the definition.

[(*e***)**

t]string_to_term2+Text, -Term Convert a String to a *Term*, stripping whitespaces

[(*e***)**

t]atom_to_string2+Atom, -String Convert an *Atom* to a *String*

[(*e***)**

t]term_to_string2+Term, -String

[(*e***)**

t]list_from_to3+Start, +End, -List Create a list of integers [*Start*..*End*]

[(*e***)**

t]repeat_string3+S, +N, -Res

[(*e***)**

t]replace_nth5+Xs, +N, +E, +R, -Ys replace the nth element of a list with *R* and return it in *E*

[(*o***)**

ndet]term_mod3+Term, +M, -ModTerm Try to apply *M* on *Term* recursively

3.6 loops.pl – Properties of loops

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

[(*e***)**

midet]is_const_val2+Term,[(V)al
ndet]is_const_val2-Term, +Val

To be done implement constant analysis result for VarRefExp

[(*e***)**

midet]isStepsize3+Term, -InductionVar,[(V)al
ndet]isStepsize3-Term, +InductionVar, +Val

is_fortran_multicond_for_loop(*+ForStatement, +I, ForInit, ForTest, ForStep, Body***)**
generate multiple *ForTest* on backtracking if multiple conditions are combined with logical and operators

is_fortran_for_loop(*+ForStatement, -I, -ForInit, -ForTest, -ForStep, -Body***)**

[(*e***)**

midet]isSimpleForInit3+InitStatement, -InductionVar, -InitVal Extracts the induction variable and the initial value from *InitStatement*

[(*e***)**

midet]isEmptyForInit1+InitStatement

[(*e***)**

midet]isForTestLE2+TestOp, -LeOp Any < test will be converted into a `=<`

[(*e***)**

midet]isForTestGE2+TestOp, -GeOp Any > test will be converted into a `>=`

[(*o***)**

ndet]isForTestOp2+TestOp, -TestOp removes the sourrounding expression statement

[(*e***)**

midet]isWhileStatement7+WhileStmt, -Condition, -Var, -Body, -Annot, -Ai[(e)Fi
t]isWhileStatement7-WhileStmt, +Condition, +Var, +Body, +Annot, +Ai, +Fi FIXME rename this!

[(*e***)**

midet]isDoWhileStatement7+DoWhileStmt, -Condition, -Var, -Body, -Annot, -Ai[(e)Fi
t]isDoWhileStatement7-DoWhileStmt, +Condition, +Var, +Body, +Annot, +Ai, +Fi

[(*e***)**

midet]isMin2Func3+MinFunc, -Expr1, -E[x(p)2
t]isMin2Func3-MinFunc, +Expr1, +Expr2 FIXME move to `annot.pl`

max_nesting_level(*+Loop, -N***)**
return the maximum number of loops nested inside *Loop*

3.7 markers.pl – Properties of abstract syntax trees

author Adrian Prantl <adrian@complang.tuwien.ac.at>

version 0.9.0

This module defines commonly-used transformation utilities for C/C++/Objective C ASTs given in the TERMITE term representation as exported by SATIrE.

3.8 loopbounds.pl

loop_bounds(*+Info, -InfoInner, -InfoPost, +Fs, -Fs_Annot***)**

[(*e***)**

t]expr_constr3+Expr, +AM, -Expr1 *AM* is (Analysisresult-Map)

loop_constraints(*+Fs, -Fs_Annot, +RootMarker, +Map***)**

3.9 termlint.pl – Term type checker

author Gergo Barany <gergo@complang.tuwien.ac.at>

version 0.9.0

license See COPYING in the root folder of the SATIrE project

This is `term_lint.pl`, a small tool for checking terms against a tree grammar (abstract syntax).

term_match(*+Term, +NonterminalSymbol***)**

OK. Here is the semantics to the structures defined above:

The grammar is a sequence of grammar rules. Each rule is of the form Nonterminal ::= Body. There should be no two rules for the same nonterminal (use `|` instead), but this is not checked.

Body has the following meaning:

atom
: a nonterminal, references another rule to be used

A `|` B
: match A; if that fails, match B

A **where** C
: match A; if that succeeds, call Prolog goal C

`{}`**(***A***)**
: unify with term A; {_} means "any term", {foo} is terminal foo

atoms As
: match one of the atoms in As

functors Fs *with* Fs
: match term F(A1,...,An) where F is a functor in Fs

f(*A1, ..., An***)**
: match a term with functor f, where arguments match A1,...,An

Argument expressions (in argument tuples) can be:

atom
: a nonterminal

A `?`
: term is "missing" (see below) or matches argument expression A

`{}`**(***A***)**
: unify with term A

`.`**(***A, ***)**
: list of terms matching argument expression A

As a special case, `[_]` means "list of any type".

Options (A?) are resolved as follows: If the term under consideration is a solution of predicate `missing/1` (to be defined along with the grammar rules), there is a match; otherwise a term of type A must be matched.

Predicates to be defined along with the grammar rules are:

`missing`*() / 1*
: defines what A? can match except for A

`start_symbol`*() / 1*
: single solution is the start symbol of the grammar

Here are a few example grammars to illustrate the explanations above.

Arithmetic expressions, simple verbose version:

```
var ::= {VarName} where atom(VarName).
num ::= {Number} where number(Number).
expr ::=
  var
| num
| expr + expr
| expr - expr
| expr * expr
| expr / expr.
```

Arithmetic expressions, more condensed version:

```
expr ::=
  {Leaf} where (atom(Leaf) ; number(Leaf))
| functors [+, -, *, /] with (expr, expr).
```

Simple type system:

```
type ::=
  atoms [number, character, string]
| function_type([type] /* argument types */, type? /* return type if any */).

missing(none).    /* what to match if no return type is given */
```

Partial specification (some parts are not constrained):

```
allowed ::=
  lst([_])        /* argument is list of some unknown things */
| opt(allowed?)   /* argument is missing, or of type allowed */
| any({_}).       /* argument is any term (anything unifies with _) */

missing(nothing).
```

In this last example, some allowed terms are:

```
  lst([]), lst([f(f)]), lst([x,y]), ...
  opt(nothing), opt(lst([])), opt(opt(nothing)), ...
  any(foo), any(g(f(x))), ...
```

One more thing, which might not be explicit from the stuff above: The anonymous variable _ (free variables in general) may appear in the grammar, but **only** inside [] or {}:

.**(***Arg1,* **)**
: (list of any type) or

{}**(***Arg1***)**
: (any term) or

`{}`**(***foo(_G1433)***)**
(any term with functor foo)

but:

`foo`*()* **::=** `foo`*(Arg1)*
is not allowed; use `foo ::= foo({_})`

`foo`*()* **::=** X *where* X
is not allowed; use `foo ::= {X} where cond(X)`

The interpreter does not check these things at the moment. Which means that grammars containing variables in weird places will misbehave in weird ways.

Index