Term Representation Generator for C Files + Unparser

User and Extension Manual (with Test Report)

IMPORTANT:
This software is provided as is, with no warranty of any kind.
The entire risk of using the program is with the user.

Table of Contents

Usage

Creating the term representation

To create a first order term representing a C/C++ program call
```
generatePrologTerm input
```
to read the C/C++ file input and write the output to stdout
OR
```
c2term input output
```
to read teh C/C++ file input and write the output to file output. If output exists previously, it is overwritten

Transforming the term representation back to source code

To create source code from the term representation
EITHER pipe a term representation followed by EOF to
```
termparser
```
OR call
```
term2c prologfile newsourcefile
```
to read the term representation from prologfile and write the created source code

Inspecting a term representation

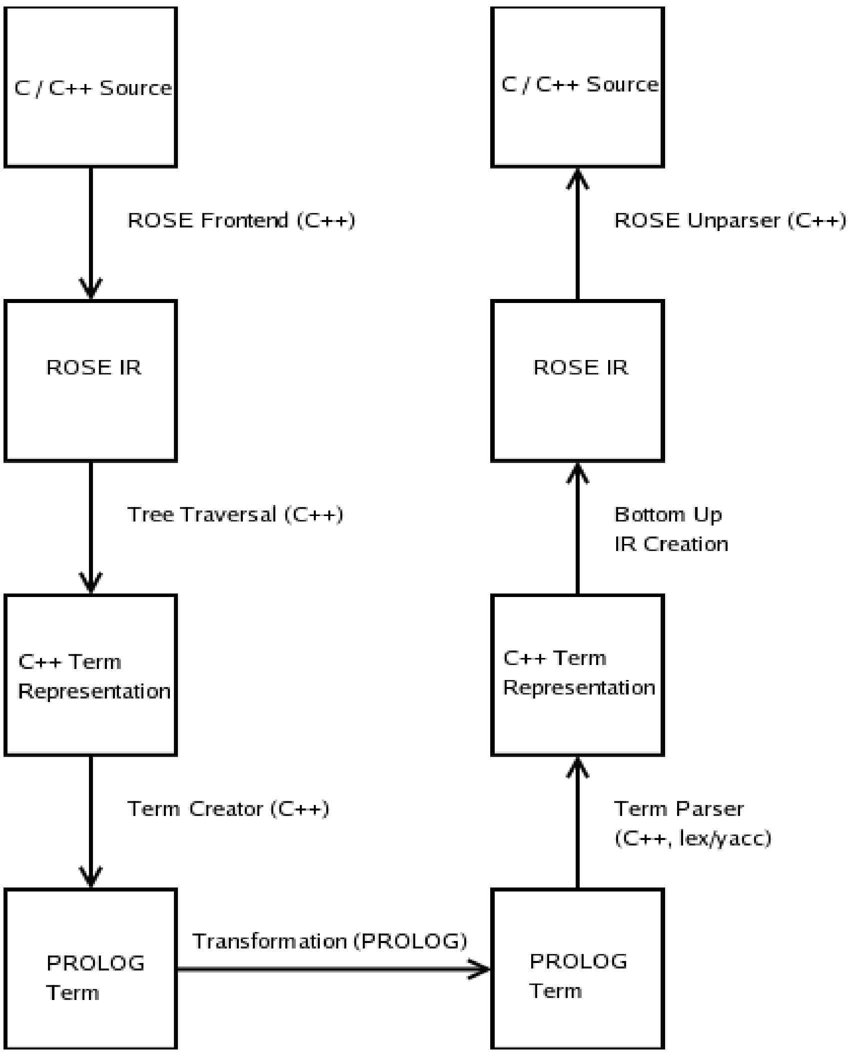
As terms get pretty large (about 40 times the size of the source program), they are useless to the human reader in their pure form. An experimental prettyprinter called termpretty is provided. It does indentaion and adds vertical lines consisting of ':' symbols. It reads from stdin and writes to stdout.

An example transformation

in the folder prolog/ there is a PROLOG file called
transformer.pl
When the program is consulted and the predicate testrun/0 is called, the program reads a term from input.pl and writes the transformed term to output.pl It consists of some rewriting infrastructure and two example transformations. (tested with SWI-PROLOG)

Extension

ntroduction
This document is an overview of the term generation and term parsing as well as unparsing process, which is illustrated in this figure.



All of the classes and member functions are documented (with doxygen) as well as inline documentation about what is going on. The aim of this document, therefore, is only to give an overview about the process and give pointers to where to find, add or some specific behaviour.

Process

The following steps are executed when creating the term representation.

- The C++ source code is parsed by the rose frontend
- The ROSE-IR is traversed by subtyping an AstBottomUpProcessing class, using a class called PrologTraversal.
- The result of the traversal is a C++-object-representation of the PROLOG term that corresponds to the C++ source (using class PrologTerm and subclasses)
- The PrologTerm classes come with a method getRepresentation() that, recursively, creates their textual representation. Note that this step is completely independent of the generation of the term representation.

When creating source code again, this is what happens:

- a lex/yacc parser (written in C++) parses the PROLOG term and creates a PrologTerm object representing it.
- The object is passed to the static member function toRose(PrologTerm*) of class PrologToRose. This function recurses into the term structure and creates a ROSE-IR bottom-up.
- The ROSE-IR is unparsed, resulting in C++ code again.

Term Creation, a little more detail

- The C++ file `toProlog.C` creates the ROSE-IR for some file, creates a `PrologTraversal`-object (see `PrologTraversal.h/PrologTraversal.c`) on them, runs its the `traverseInputFiles` method, gets the `PrologTerm`-class representation by running the traversal's `getTerm` method and then outsputs the result of the PrologTerm's `getRepresentation` method.
- Every node term consists of a generic part, created in the `PrologTraversal` and a node specific part, depending on the node type. The PrologTraversal uses `PrologSupport::addSpecific` to create the latter and add it to the generic term. This method, then, depending of the node types, calls the private static member function corresponding to the type, that returns a node specific `PrologCompTerm`, which is then added to the generic node term.

Term unparsing, a little more detail

- `main.C` calls the `yylex()` function created from `termplerxer.l++/termparser.y++`, which, after parsing the PROLOG term, saves a `PrologTerm*` object in the global varialbe `prote`.
- `main.C` then calls the static member function `SgNode* PrologToRose::toRose(PrologTerm*)` which is a factory method creating the corresponding ROSE-ir
- `toRose`, depending on the node arity, calls `leafToRose, unaryToRose` etc., which then call the class specific factory methods. This is where the actual node creatoin happens.

- When a SgGlobal node is encountered and was created, its `unparseToCompleteString` method is called and the result (C++ source code) written to stdout.

Change Howto

Inspecting the Term Representation

The make-target *termpretty* (automatically called by make all) creates an executable file with the same name, which reads a PROLOG term from stdin and writes an indented version of it to stdout. To improve readability, vertical dotted lines (made of „:"-symbols) are added. Therefore, this representation is just for human reading, it is no longer valit PROLOG.

New IR class

- Create a C/C++ file containing the corresponding Construct.
- Create PROLOG term with current version, use it to determine the arity of the construct (*termpretty* may be useful here.).
- Determine which information, apart from a Sg_File_Info and the children that are automatically added to the term by the PrologTraversal will be necessary to call the class' constructor.
- In class PrologSupport, write a static member function (assuming class name SgExampleClass here).
  `static PrologTerm getExampleClassSpecific(SgExampleClass*);`
- The function should create a PrologCompTerm* of name
  `exampe_class_annotation(...)`
  that contains the necessary information in any desired form.
- Document the Function:
  ```
  class: SgExampleClass
  term: [term structure here]
  arg something: explain what subterm somthing represents
  ```
- in Function `PrologSupport::addSpecific`, there is a long if/then/else sequence that attempts to cast the current SgNode* to more specific types and, if successful, calls the specific functions. Add such a cast and subroutine call there.

This is all that's necessary to create the annotation term. Now the Node has to be recreated in the unparser section:

- define a private static member function `SgExampleClass* PrologToRose::createExampleClass(Sg_File_Info*,[children],PrologCompTerm*)`
- Depending on the node terms arity, add a call to it in PrologToRose::`leafToRose`, `PrologToRose::unaryToRose` etc.
- create the actual node. This very much depends on the class that should be created, the

following things are noteworthy, though

- The children come as SgNode* pointers, therefore they will have to be casted and, if they are necessary, tested for not being NULL.
- The PrologCompTerm* points to the complete node term, to get the annotation and cast it to PrologCompTerm*, just call `PrologToRose::retrieveAnnotation(PrologCompTerm*)`
- If the class you want to implement contains references to previous declarations, the current way of doing things is creating a dummy declaration, not traversing the new ROSE-IR to finde the declaration. Note that such a declaration usually needs to be in some parent scope or unparsing will fail. Therfore, `PrologTORose::fakeParentScope(SgDeclarationStatement*)` was created to fix this for dummy nodes.

Modify the way an existing node is treated

The process described in the previous section was followed for all the nodes already implemented. To change anything about the way a class SgExampleClass is unparsed, one has to change either one or both of

- `PrologSupport::getExampleClassSpecific(SgExampleClass*)`
- `PrologToRose::createExampleClass(...)`

There are three exceptions to this: SgBinaryOp, SgUnaryOp and SgValueExp for which all the term creation / node creation of their subtypes is `getBinaryOpSpecific/createBinaryOp` etc..

Test Report

All the tests can be called via

```
make check
```

in the `src/` directory. It is a phony target which calls the script test.sh. This tranforms all files with the suffix .C in the directory `src/tests/` to their term representation. The term representation for `src/tests/X.C` is named `src/tests/results/X.C.pl`. The term representation is then, by using the term parser, transormed back to source code. `src/tests/X.C` is unparsed to `src/tests/results/X.C.unparsed.C`. Furthermore, the result of ROSE unparsing the original file's ROSE-IR is saved as `src/tests/results/X.C.rose.C`.

X.C = generatePrologTerm => results/X.C.pl =termparser=>results/X.C.unparsed.C

Furthermore, the .pdf and .dot representation as well as the output to stderr of each file is moved to the `src/tests/results/` directory.

A diff-output of the .rose.C and the .unparsed.C-Files is saved as src/test.log

To remove all test output, call

```
make clean_check
```

Tested Language Features

C

- Structures
- Unions
- Enums
- Typedefs
- Function declarations and calls
- unary and binary operators
- control structures (including goto and labels)

C++

- Class Declarations
- Member function delarations and calls
- try/catch
- new
- delete
- namespace declarations

Test files

This section contains the list of C and C++ files tested. „identical“ means that there are no differences except comments (which aren't preserved). Correct means semantically equal (see Peculiarities). Typedef means that the

C

File name	*identical*	*correct*	*typedef*	*differnces*
tests/test_control.	+		-	
tests/test_enum.C	+	+	-	
tests/test_minimal.C	+	+	-	
tests/test_struct.C	+	+	-	
tests/test_transformme.C	-	+	-	parentheses
tests/test_typedef.C	-	-	+	Typedefs not unparsed correctly
tests/test_control.C	-	+	-	parentheses, booleans

File name	***identical***	***correct***	***typedef***	***differnces***
tests/test_enum.C	+	+	-	
tests/test_minimal.C	+	+	-	
tests/test_struct.C	+	+	-	

C++

Filename	***Identical***	***Correct***	***Contains „new“***	***differences***
tests/test[1-9].C	-	-	+	New not unparsed correctly, typedefs
tests/test_class1.C	-	-	+	New not unparsed correctly
tests/test_class1.C	-	-	+	New not unparsed correctly

What works

C

- Control structures (including goto and labels)
- Variables (declarations, calls etc)
- Unary and binary operators
- Types
- Unions
- Enums
- Typedefs with no nested declarations

C++

The above plus

- Variables in global scope
- Classes with global scope, their member functions (including calls) and variables
- try/catch
- delete

Peculiarities

This section documents the language features where unparsing differs between the term representation and the original ROSE-IR but that don't change the semantics. The bullets illustrate what's different in the term representation's unparsed code.

Both C and C++

- Comments are not preserved
- Enums: member names are in parentheses: `enum x {(a),(b)};`
- Enums: foo(x) is replaced by its value
- Booleans: 1/0 instead of true/false
- Assignments: outermost parentheses ommited (`x = 3 + 2` instead of `x = (3 + 2)`).

Problematic Language Features

C

- Typedefs with nested declarations (declarations aren't unparsed)

C++

- New-operator (`new foo(x)` becomes `::new foo foo(x)`)
- Classes: no inheritance
- Member functions: no „throws“ declarations
- No scope information except in class declarations and member function declarations

Test summary

Transformation from and to C works well with the given restriction of typedefs that are nested. Some language features of C++ (mentioned above) work too.