

Manual of TreeGen 2023

Hans de Nivelle

March 19, 2026

Abstract

TreeGen is a program that automatically creates C^{++} -class definitions from definitions of recursive data types. TreeGen is by itself written in C^{++} . The generated class definitions use automatic memory management (RAII), and start sharing subtrees when assigned or copied, while at the same time strictly preserving value semantics, which means that there will be no side effects caused by sharing. Memory management is done by means of reference counting.

It is possible to use tree constructors that have an unbounded number of subtrees. Trees constructed by these constructors have vector-like behavior. Their size can grow and shrink dynamically.

The generated tree classes support in-place rewriting. In-place rewriting is possible when the tree is not shared at all subtrees that do not have an ancestor that is shared with another tree.

TreeGen is released under the Aladdin Free Public License (AFPL).

1 Structure of Generated Tree Types

TreeGen generates C^{++} implementations of tree-like types, which are usually recursive. A tree type can be in different states, dependent on the way it has been constructed. We will call these states *options*. Dependent on its option, a tree variable can have different fields. This implies that fields do not always exist. There are four types of fields:

1. *Prefix fields* are stored locally in the tree, and do not depend on the option. They can be used for storing a type, or storing source information.
2. *Local fields* depend on the option, but are still stored locally.
3. *Scalar fields* are stored on the heap.
4. Repeated fields are stored on the heap, and may be repeated arbitrarily often. Repeated fields behave like vectors. They have to be indexed, and their number can grow and shrink at runtime.

Recursion is possible only through scalar and repeated fields, because recursion must involve a pointer. Earlier defined tree types can be used in local fields, or prefix fields without restriction. The pointers are not visible to the user.

The generated tree types fully support RAII, which means that their memory management is completely automatic, and invisible to the user.

When a tree type is copied or assigned, the prefix fields and local fields are also copied or assigned. Scalar and repeated fields are not copied, which implies that they may become shared. Reference counting is used internally for memory management.

Sharing creates a potential problem with assignment: If one assigns to a shared part of a structure, this may result in unexpected changes in other variables that appear to be unrelated. In order to prevent this, we allow only const access to scalar and repeated fields.

Scalar and repeated fields can be updated, for which we use a form of copy-on-write (COW). When performing an update, the new value is compared with the old value. If the values are the same, nothing is done. If the values differ and the updated field is shared, a unique copy is created. After that the field is assigned to. Since comparing values can be expensive, we use a combination of value and reference equality for equality testing. This combination is called *very equal*, and its opposite is called *distinct*.

The default behavior is that local and prefix fields are compared by operator `!=`. In case the operator `!=` is inadequate, it is possible to specify different ways of deciding distinctness. Scalar and repeated fields are compared by the pointer that points to them, so they are not looked at during comparison. Updating is subtle because in general, one wants to restrict the number of reallocations. This is important for efficient rewriting. In particular, one wants to avoid that replacement in a subtree that is recursively unique (not shared), results in a reallocation. This can be only achieved by using `std::move`. We discuss the interaction between moving, rewriting and updating separately in Section 5.

The current option of a tree type is determined by an `enum` type called `selector`. Values of the selector are mapped to options. It is possible that different selector values use the same option. In order to determine which option is currently used, one can either query the selector using method `selector sel() const`, or call a predicate `bool option_is_X() const`, where `X` is the name of the option. The advantage of the first method, is that one can use `switch`.

If a tree has fields whose existence depends on the current option, (either local, scalar, or repeated fields), then these fields cannot be directly accessed from the tree. One must first create a *view* that corresponds to the current option. After that, the fields can be accessed as fields of the view. The view can be considered as a kind of downcast. Details are discussed in Section 4.2.

2 Running TreeGen

TreeGen must be called with exactly one argument, which is the name of the input file. It will create output files whose names are specified in the input file. It can create any number of output files, as long as they are in the same directory. If the directory does not exist, TreeGen will not create it. There are no requirements on the name of the input file, but I usually use `.rec` as

extension.

3 Input Format

The input file must start with a `%dir` directive, that specifies into which directory the output files will be written. If the given directory is not absolute, the directory will be relative to the directory from which TreeGen was called. The following are examples:

```
%dir .          // Use current directory
%dir logic     // Use ./logic
%dir /home/nivelle/project // absolute directory
```

After that follows a namespace specification of form `namespace N1 :: ... :: Nn`. This is the namespace in which all C^{++} classes will be defined. The specification can be empty, in which case the classes will be defined without namespace. Here are a few examples:

```
%namespace      // Without (or in top level) name space.
%namespace logic // In namespace logic.
```

If one wants to create tree types in different directories, one has to create different files.

3.1 Definitions of Tree Types

After the header, which consists of the output directory and namespace specification, there can be an arbitrary number of type definitions. They will be all written in the same directory. Each type definition consists of a name, a list of prefix fields, followed by a list of options. Every option consists of a list of selector values that have the option, followed by the list of fields of this option.

Each type definition must start with `%define N (E)`, where `N` is the name of the type, and `E` is a selector that specifies the empty (moved out) option. Both must be valid C^{++} identifiers. The defined by `E` cannot have fields on the heap, i.e. no scalar or repeated fields.

We give simply typed first-order logic as example. The definition is as follows:

Definition 3.1 *We recursively define terms:*

- If t_1, \dots, t_n ($n \geq 0$) are terms, f is a function symbol, then $f(t_1, \dots, t_n)$ is a term as well.

We assume a set of sorts \mathcal{S} . Using sorts, simply typed formulas can be recursively defined as follows:

- If p is a predicate symbol, t_1, \dots, t_n ($n \geq 0$) are terms, then $p(t_1, \dots, t_n)$ is a formula.

- If t_1, t_2 are terms, then $t_1 \approx t_2$ is a formula.
- Both \perp and \top are formulas.
- If F is a formula, then $\neg F$ is a formula.
- If F_1 and F_2 are formulas, then $F_1 \rightarrow F_2$ and $F_1 \leftrightarrow F_2$ are also formulas.
- If F_1, \dots, F_n are formulas, then $F_1 \wedge \dots \wedge F_n$, and $F_1 \vee \dots \vee F_n$, are formulas.
- If v_1, \dots, v_n is a sequence of variables, S_1, \dots, S_n a sequence of sorts, and F is a formula, then $\forall v_1:S_1 \dots v_n:S_n F$ and $\exists v_1:S_1 \dots v_n:S_n F$ are formulas.

This is the same definition using TreeGen:

```
%define term ( term_var )
%option var { term_var } => # var : std::string
%option func { term_func } => [ sub : term ], # f : std::string

%define formula ( form_bot )
%option atom { form_atom } => [ arg : term ], # pred : std::string
%option equals { form_equals } => # arg1 : term, arg2 : term
%option nullary { form_bot, form_top } =>
%option unary { form_not } => sub : formula
%option binary { form_implies, form_equiv }
=> sub1: formula, sub2: formula
%option nary { form_and, form_or } => [ sub : formula ]
%option quant { form_forall, form_exists }
=> body : formula, [ var : varwithsort ]
```

Class varwithsort can be defined as follows:

```
struct varwithsort
{
    std::string var;
    sort tp;

    varwithsort( const std::string& var, const sort& tp )
        : var( var ), tp( tp ) { }

    varwithsort( std::string&& var, sort&& tp )
        : var( std::move( var ) ), tp( std::move( tp ) ) { }

    bool operator == ( const varwithsort& other ) const
        { return var == other. var && tp == other. tp; }
};
```

In principle, the `nullary` option is redundant, because $\top = \bigwedge\{\}$ and $\perp = \bigvee\{\}$. We keep it here, because we want an option without arguments in the example, and we can use it for the moved-out state.

3.2 Copied Code

After each tree type, after the options, it is possible to specify code that will be copied into the output files, either the `.h` file or the `.cpp` file. The code to be copied must be between braces `{ }`. There are six possibilities, dependent on where the code must be placed.

- `h_incl` : At the beginning of the header file, where normally the `#includes` are written. Note that there are a few includes that must be always present, see Section 4
- `h_before` : Before the definition of the tree type, inside its namespace. This can be used defining small helper classes. Class `varwithsort` in Section 3.1 should be defined with `h_before`:
- `h_methods` : Inside the definition of the tree type, after the methods created by TreeGen. This is useful if one wants to define additional methods of the tree type.
- `h_after` : After the definition of the tree class, but still within its namespace. Useful if one wants to define functions involving the tree type, (e.g. `operator <<`)
- `cpp_front` : At the front of the `.cpp` file. This can be used for putting things in anonymous namespace. No namespace is added. This means if the copied must be in the same namespace as the tree class, one has to add the namespace prefix by oneself.
- `cpp_back` : At the back of the `.cpp` files. No namespace is added.

Each of the copy commands can be repeated arbitrarily often, and there is no fixed order. Don't include copy commands with an empty code block, because the parser does not like them.

4 Using the Generated Tree Classes

Before the tree classes can be used, they have to be compiled. Make sure that the input file contains the following include

```
#include "tvm/includes.h"
```

for every defined tree type. Make sure that the compiler finds the `tvm` directory for including.

4.1 Constructors

For each tree type, each option defines a constructor. The arguments of the constructor must fit to the option being constructed. The order of the arguments must be the same as listed in the option, with the exception that the selector must always be first. So, the order is:

- The selector.
- Values for the prefix fields, if the tree type has any.
- Values for the scalar fields, if the selected option has any.
- If the option has repeated fields, then either an initializer list, or a pair of iterators (begin/end).
- Values for the local fields, if the option has any.

The total number of constructors can be quite big, and it is sometimes tricky to call the right one.

- The parameters of the constructor must correspond to the option specified by the selector. Unfortunately, the *C++* type system is unable to check this at compile time. If the field `bool check` is set to `true`, the selector value will be checked at runtime. It is set by default, but it can be turned off by editing the file.
- In case there are similar options with different number types, it can be tricky to ensure that the right constructor is called. For example, if one has options:

```
%option int { sel_int } # i : int
%option unsigned { sel_unsigned } # u : unsigned int
%option double { sel_double } # d : double

mytree( sel_int, 4 ) // Ok.
mytree( sel_double, 4 ) // Not ok, will call constructor for int.
mytree( sel_double 4.0 ) // Ok.
mytree( sel_double, (double)4 ) // Ok.
mytree( sel_unsigned, 10 ) // Not ok, calls constructor for int.
mytree( sel_unsigned, 10u ) // Ok.
mytree( sel_unsigned, (unsigned) 10 ) // Ok.
```

- In case there are repeated fields, they can be put in a single initializer list. If there is exactly one repeated field, the values for this field can be listed between `{ }`. For example, if the repeated field is `[s : std::string]`, one can write `{}`, `{ "hallo" }`, or `{ "hello", "world" }` as parameter. If there is more than one repeated field, each repetition must be put in its own pair of `{ }`. For example, if the repeated fields are `[i : int, s : std::string]`,

one can write `{}`, `{ { 1, "hallo" } }`, or `{ { 2, "hello" }, { "world" } }` as parameter.

There is a potential problem when the initializer list is empty, namely that it can bind to an initializer list of any type. If there are two options

```
[ s : std::string ]
[ i : int ]
```

and no further distinguishing fields, then the empty initializer list will bind to both. In that case, one has to write `std::initializer_list< std::string > { }` or `std::initializer_list< int > { }`.

When there are two repeated fields, one has to use `std::pair< >`. When there are more, use `std::tuple`.

```
[ i : int, s : std::string ]
    // use std::initializer_list< std::pair< int, std::string >> { }
[ i : int, s : std::string, d : double ]
    // Use std::initializer_list< std::tuple< int, std::string, double >> { }
```

- Instead of giving a single initializer list for a repeated field, one can also provide a pair of iterators to the constructor. (We know about `std::range` but we don't like it.) The iterators must be equality comparable, have `operator - ()`, have an `operator ++ ()`, and also `operator * ()`, such that `*` yields the type of the repeated part. The rules are the same as for the empty initializer list: If there is one repeated field, `operator *` must yield the type of this field. If there are two repeated fields, `operator *` must yield an `std::pair` of these fields. If there are more repeated fields, `operator *` must yield an `std::tuple` of all repeated fields.

Repeated fields behave like vectors. This means that in general, no separate container is needed because the tree type itself can be used as container.

4.2 Field Access

In order to access the selector, use `sel() const`. The only way to change the selector is by assigning a new tree to the variable.

If the tree has prefix fields, they can be accessed by their name, used as method of the tree. This means that if the original name was `f`, the field has to be accessed as `f()`. This function has a `const` and a non-`const` version. This is safe because prefix fields are always stored locally.

The remaining fields depend on the selector and cannot be accessed directly. Before they can be accessed, one first needs to check that the tree has the right option. This can be done either by calling `sel() const`, after which one knows the option, or by directly calling `option_is_X()`,

Once the option is known, one create a view and access the remaining fields through this view. The view is similar to a down cast (if one would have used

a derived class). In order to create a view, call method `view_X()`, where `X` is the name of the option. It has two versions, a const version and a non-const version.

The view can be stored in a local variable, or repeated every time when a field is accessed. If one wants to store the view, write `auto v = t.view_X()`, where `X` is the name of the option that `t` is supposed to have. If the current option of `t` is not `X`, calling `view_X` will result in an `invalid_argument` exception. In the examples that follow, we assume that `auto v = t.view_X()`; was called before. If one does not want to store the view, one can write `t.view_X()` instead of `v`:

- If the local field has name `f`, one can use `v.f()` to access it. If `t` is not const, it is possible to assign to the field.
- If the scalar field has name `f`, one can use `v.f()` to access it. It is not possible to assign to scalar fields.
- If the repeated field has name `f`, one can use either `v.f(i)` to access it, where `i` must be expression of type `size_t`. In order to determine how many repetitions there are, use `v.size()`. This method exist whenever the option has repeated fields. It is not possible to assign to repeated fields.

Scalar and repeated fields cannot be assigned to, but they can be *updated*. Moreover, the number of repetitions of the repeated fields can be changed by pushing or popping. So we get:

- Local fields can be assigned to by writing `v.f() = ...`.
- Scalar fields can be updated by writing `v.update_f(...)`.
- Repeated fields can be updated by writing `v.update_f(i, ...)`. Their number can be increased by calling `v.push_back(t1, ..., tn)`, where the arguments must provide a value for each repeated field.

The number of repetitions can be decreased by calling `v.pop_back()`. This removes one copy of each repeated field.

The implementation of `push_back()` is such that it has $O(1)$ complexity (amortized).

Update methods first compare the new value to the old value. If they are the same, nothing is done. Otherwise, a unique copy is ensured (possibly needing a reallocation), after which the field is assigned. We explain in the next section how the update methods compare the new to the old value.

Adding or removing repeated fields will always result in reallocation, when representation is shared. A tree can be used as container, so that it is usually not necessary to construct repeated fields in another container: One can first call a constructor with empty initializer list (possibly specifying its type, if needed), then create a view on the constructed tree, and call `push_back()`. There is no way to insert repeated fields in the middle.

5 Update and Very Equal

Scalar and repeated fields cannot be directly assigned to, because they are potentially shared, which would destroy value semantics. Instead of being assigned to, scalar and repeated fields can be updated. The difference between updating and assigning is that updating is a method of the tree class, so that the tree knows when a change takes place. In order to enable assignment, the tree would have to give out a non-const reference, after which it has no control any more about modifications.

Update methods reallocate when the new value is distinct from the old value, and the tree is currently sharing its representation. In that case, a unique representation is created. Since comparing two values can be expensive, one should try to use pointer identity whenever possible. This avoids the need for deep comparisons because all nested structures are built with pointers.

Which types are compared by value, and which types will be compared by pointer, is determined by the `distinct` method. The update method calls `distinct(t1,t_2)`, which is defined as follows, dependent on the type `T` of `t1` and `t2`:

1. If either of `very_equal(t1, t2)` or `t1. very_equal_to(t2) const` is defined, then `distinct(t1, t2)` is the same as `!very_equal(t1, t2)` or `!t1. very_equal_to(t2) const`. It is not allowed that both are defined.
2. Otherwise, `distinct(t1, t2)` defaults into `t1 != t2`.

Tree types generated by TreeGen have a `very_equal_to() const` method, which is defined as follows:

1. If the selectors are different, then the trees are not very equal.
2. At this point, we know both trees have the same option, so that they have the same fields.
If there are prefix fields, and one of the prefix fields in the first tree is `distinct` from the corresponding field in the other tree, then the trees are not very equal.
3. If there are local fields, and one of the local fields in the first tree is `distinct` from the corresponding field in the other tree, then the trees are not very equal.
4. If there are scalar or repeated fields, then these fields are stored on the heap below a pointer. The trees are very equal iff the pointers are equal.

By default, types that are not generated by TreeGen have no implementation of `very_equal`, so that they will be compared by `!=`. If that is too expensive, one can define `very_equal()` for the desired type. The intuitive meaning should always be: Everything local should be equal by value, and everything below a pointer should be identical. This is an example of an implementation of `very_equal` for vectors:

```

template< typename T >
bool very_equal( const std::vector<T> & v1, const std::vector<T> & v2 )
    { return v1. data( ) == v2. data( ); }

```

6 In-Place Rewriting

Updating is sufficient for many applications, but it cannot be used for in-place rewriting. Consider the following tree type for arithmetic on natural numbers:

```

#define nat ( nat_zero )
%option zero { nat_zero } =>
%option succ { nat_succ } => pred : nat
%option unary { nat_neg } => sub : nat
%option binary { nat_add, nat_mul, nat_sub, nat_div }
    => sub1 : nat, sub2 : nat

```

Suppose that we have a `nat` n of form `succi(add(zero, succj(zero)))`, and a rewrite system

$$\begin{aligned}
 \mathbf{add}(X, 0) &\Rightarrow X \\
 \mathbf{add}(X, \mathbf{succ}(Y)) &\Rightarrow \mathbf{succ}(\mathbf{add}(X, Y))
 \end{aligned}$$

Obviously, n will be normalized into `succi+j(zero)`. In order to implement rewriting, one must create a function with signature `nat try_rewrite(const nat& n)`, which returns a `nat` that is either very equal to n , or rewritten. For the subcases, one has to write

```

switch( n.sel( ) ) {
...
case nat_add:
case nat_mul:
    { auto n1 = n; // Cannot update n directly, it's const.
      auto v = n1.view_binary( );
      v. update_sub1( try_rewrite( v.sub1( ) );
      v. update_sub2( try_rewrite( v.sub2( ) );
      return n1; }
}

```

This code, although nice and simple (and hence often acceptable) is incapable of in-place rewriting. Because `n` is `const`, it cannot be updated directly. Hence it has to be copied into `n1`. But now `n1` is shared, and update will always make a unique copy if the subterm is changed.

In order to solve this problem, one must observe that the original `n` is nearly always irrelevant. It is probably a subterm of a surrounding term which we are trying to rewrite. After returning, it will be updated with our result. As a consequence, the sharing 'is not real'. The same applies when we are the top level call. Very likely, the programmer has written `m = try_rewrite(m)` and

the variable that we are sharing with, will be overwritten on return. In order to avoid this irrelevant sharing, one can use `std::move`. Since we want to do this recursively, we need a method that moves a subtree out of a tree if it is not shared, and otherwise creates a copy. This operation is guaranteed to be safe, because once we encounter a shared tree, we will be working on a copy, and all trees below it will also be shared and copied because of that. We call the new operation *extract*. We could also have called it *borrow* because that is the intended use, but since there is no guarantee that the user will give it back, 'extract' seemed more appropriate. Using extraction, the rewrite system can be implemented as follows:

```

nat try_rewrite( nat n ) {
  if( n.sel( ) == nat_add ) {
    auto v = n.view_binary( );

    if( v.sub2( ).sel( ) == nat_zero )
      return v.sub1( );    // X + 0 => X:

    // X + succ(Y) => succ( X + Y ) :

    if( v.sub2( ).sel( ) == nat_succ )
      return nat( nat_succ, nat( nat_add,
        v.sub1( ), v.sub2( ).view_succ( ).pred( ) ));
  }

  switch( n.sel( ) )
  {
    ... ( similar cases omitted )

    case nat_add: case nat_mul: case nat_sub: case nat_div:
      { auto v = n.view_binary( );
        v.update_sub1( try_rewrite( v.extr_sub1( ) ));
        v.update_sub2( try_rewrite( v.extr_sub2( ) ));
        return n; }
  }
}

```

Function `try_rewrite` must be called as `m = try_rewrite(std::move(m))`. If one forgets the move, the term will be still copied on every replacement. Our solution is related to the solution in [?]. Matching can be viewed as extracting all subtrees at once, after which the tree can be disabled (moved-out). Upon return, the allocator tries to reuse a disabled tree if there is one.

7 Possible Problems (with solutions)

We give a list of frequently occurring problems with their solutions: and their solutions:

- `inconsistent local/heap distribution for ({ }, { size_t }, { })`

This error occurs when two options have the same fields in the same order, but with a different distribution between local and heap fields. In that case, the constructor does not know where to place the fields. Since the optimal placement depends on the size of the types, there is no reason to use different placements for the same sequence of types. As a consequence, it is easy to fix this error.

- Empty initializer lists (`{ }`) may create problems at compile time, because the compiler is unable to determine the intended type when seeing `{ }` alone. If this error occurs, write `std::initializer_list<T> ()`. This problem will not occur when the initializer list is non-empty.
- Local fields cannot be recursive. The error will be

`option: field is a recursion in local field.`

- If the compiler complains that it cannot find some constructor of the `symbol` class, the most likely cause is that no symbol was declared with the required attribute type.
- Conversions between integer types may create unexpected problems during compilation. If one has constructors `c(selector, unsigned int)` `c(selector, size_t)`.
- `empty option has scalar fields`

`empty state has repeated fields`

These errors occur when the moved-out option (defined by the `%define` command) has scalar or repeated fields. This is not allowed, because scalar and repeated fields are stored on the heap, and the moved out state should not hold resources. If possible, the fields can be preceded by `#` to make them local.

- The current implementation cannot handle `#if #endif` in copied code. I am not sure if it is important enough to try to fix it.

8 Concluding Remarks

Here are some problems that we may want to fix in the future:

- Add moving constructors. Since most constructors have many arguments, this would potentially result in an exponential explosion of constructors. Maybe it can be solved by introducing helper concepts.
- Add moving update operators. They would move the new value into the old value.

9 Acknowledgements

Thanks to Aleksandra Kireeva, Akhmetzhan Kussainov, Dina Muktubayeva, and Olzhas Zhandeldinov.

We thank Nazarbayev University for supporting this project through the Faculty Development Competitive Research Grant Program (FDCRGP), grant number 021220FD1651.