Trees for Logic and Parsing

Hans de Nivelle

Nazarbayev University, Astana, Kazakhstan

Aspen, 12.05.2023

About Myself

- Have always worked in teaching and research.
- Currently working at Nazarbayev University, Astana, Kazakhstan.

  Before I worked at university of Wrocław, Poland, and Max-Planck Institut für Informatik in Saarbrücken.
- Interested in logic and theorem proving. Have been using C^{++}, Java, and Prolog for implementation.

About Nazarbayev University

- Nazarbayev University was established in 2010. Before that, the best students were sent to universities abroad.
- NU has 4000 undergraduate students, 1500 graduate students, and 700 students in the foundational year.
- Proficiency in English is a requirement for everyone.
- All important decisions are merit only.
- School of Engineering and Digital Sciences has 2000 undergraduate students, 300 graduate students.

Context

- I have been trying to implement logic in C^{++} since 2003.

  Most projects failed.

  At some point (2017), I believed that implementing logic in C^{++} is not possible.

  I looked at other languages, but they all have problems.

- I started a project to develop a programming language, specialized for implementation of logic. For this project, I created Maphoon (See CppNow 2022).

- While working on the compiler, I studied the question of how to efficiently represent logical formulas in C.

  I made progress.

Context (2)

- Last year (summer), I asked the question: Why not use the C-implementation technique in C^{++}.

  It turns out the resulting tree implementations are very nice to use, technically efficient, but very tedious to write.

  I created a generator which does the writing from a recursive definition.

- I consider the problem problem of implementating logic in C^{++} completely solved.

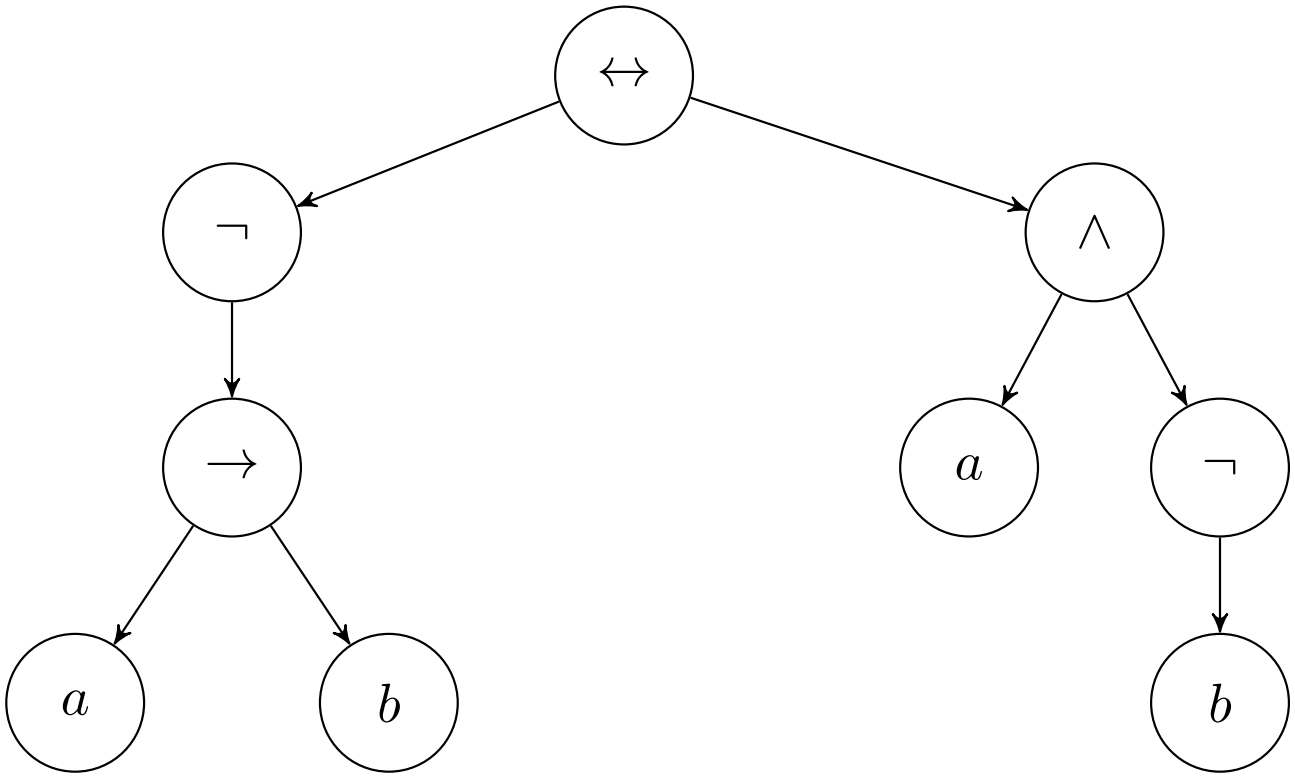
  This is possible because I found new implementation methods, and C^{++} added concepts.

- I must reconsider the programming language. Perhaps it is no longer useful to continue this project.

It's all About Trees

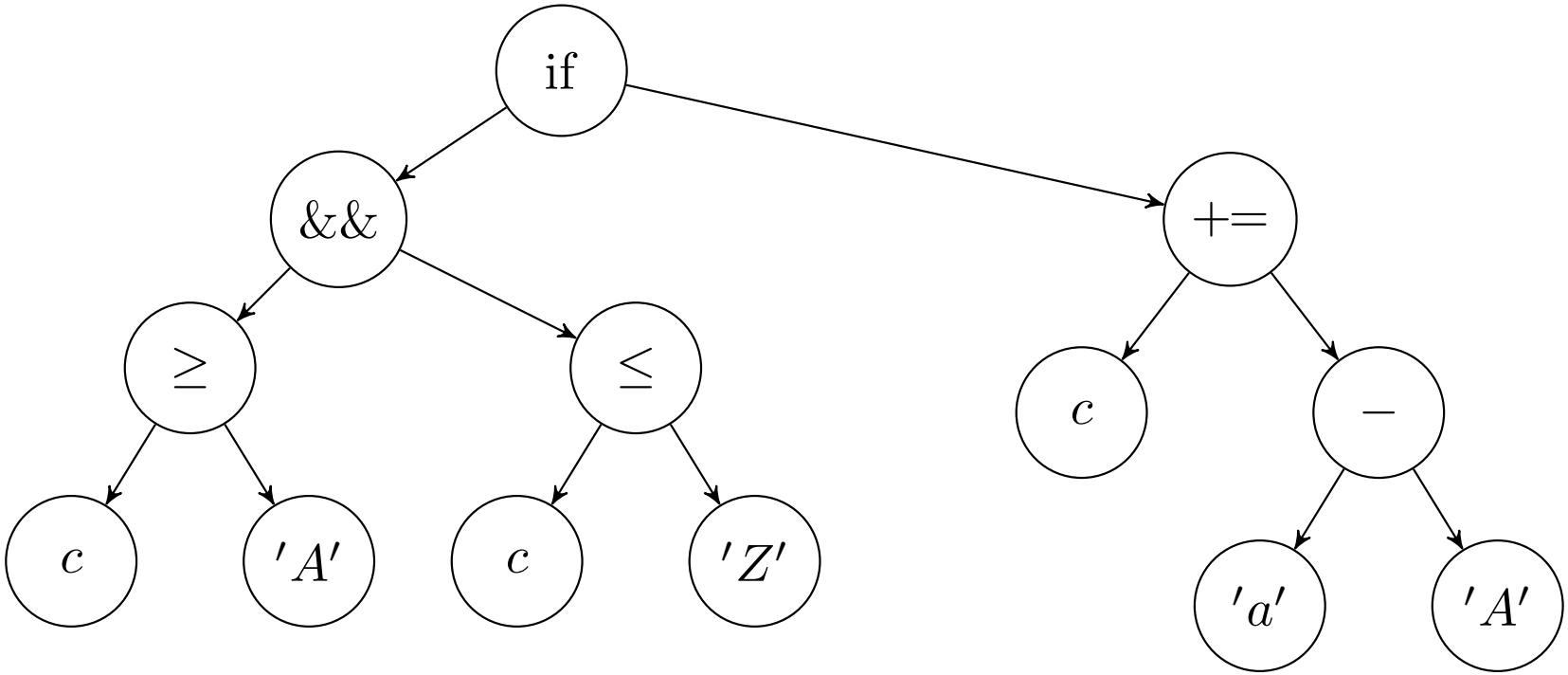
In computer science, nearly everything can be represented as a tree.
Logical formulas are trees:

$$\neg(a \to b) \leftrightarrow (a \wedge \neg b).$$



Example of an AST

```
if( c >= 'A' && c <= 'Z' ) c += 'a' - 'A';
   // If c is upper case, make it lower case.
```



What is Important?

This:

- A nice interface (to the programmer who uses the datatype).
- Implementation difficulties: memory managment, run time polymorphism.
- Time spent implementing/modifying the datatype. (the programmer who creates the datatype).
- Efficiency: Mostly efficiency of rewriting. Avoid reallocations.

On the next slides, I give three examples:

Example 1 : λ-terms

Definition: We recursively define λ-terms as follows:

- A variable v is a λ-term.
- If f and t are λ-terms, then $f \cdot t$ is a λ-term.
- If v is a variable, and t is a λ-term, then $\lambda v\ t$ is a λ-term.

It is possible to add types to λ-terms, but we won't do this here.

I will call the 3 options states of the term.

On the next slides, I show a few operations on λ-terms, so that you see what kind of operations logicians are typically interested in:

λ-terms (Free Variables)

Let w be a variable, let t be a term. We recursively define when w is free in t :

- w is free in v iff $w = v$.
- w is free in $f \cdot t$ iff either w is free in f, or w is free in t.
- w is free in $\lambda v\ t$ iff $v \neq w$ and w is free in t.

λ-terms (Substitution)

We define the result of substituting u for w in t (written as $t[\,w := u\,]$) as follows:

- If v is a variable, then

$$v[\,w := u\,] = \begin{cases} u & \text{if} \quad v = w \\ v & \text{if} \quad v \neq w \end{cases}$$

- $(f \cdot t)[\,w := u\,] = (\ f[\,w := u\,]\ ) \cdot (\ t[\,w := u\,]\ )$
- $(\lambda v\, t)[\,w := u\,] =$

$$\begin{cases} \lambda v\ t[\,w := u\,] & \text{if} \quad v \text{ is not free in } u \\ \lambda v'\, t[\,v := v'\,][\,w := u\,] & \text{if} \quad v' \text{ is not free in } t \text{ or } u \end{cases}$$

λ-terms (β-Reduction)

- $(\lambda v\ t) \cdot u$ reduces into $t[\, v := u \,]$.
- If f_1 reduces into f_2, then $f_1 \cdot t$ reduces into $f_2 \cdot t$.

  If t_1 reduces into t_2, then $f \cdot t_1$ reduces into $f \cdot t_2$.
- If t_1 reduces into t_2, then $\lambda v\ t_1$ reduces into $\lambda v\ t_2$.

Normalization: Apply reductions as long as possible.

λ in Analysis

Laplace Transform:

$$\mathcal{L}\{f(t)\} = F(s) = \int_0^\infty e^{-st} f(t)\, dt.$$

Gaussian Formula:

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)}{2}.$$

Example 2 : Term Rewriting

We recursively define terms as follows:

- 0, **false**, and **true** are terms.
- If t is a term, then **succ**(t), **not**(t), and **fact**(t) are terms.
- If t_1 and t_2 are terms, then **plus**(t_1, t_2), **times**(t_1, t_2), and **equal**(t_1, t_2) are terms.
- If $t_1, \ldots, t_n$ are terms, f is an identifier, then $f(t_1, \ldots, t_n)$.

This time, there are 4 states.

Rewrite Rules

$$
\begin{array}{lcl}
\mathbf{plus}(X, 0) & \Rightarrow & X \\
\mathbf{plus}(X, \mathbf{succ}(Y)) & \Rightarrow & \mathbf{succ}(\mathbf{plus}(X, Y)) \\
 & & \\
\mathbf{times}(X, 0) & \Rightarrow & 0 \\
\mathbf{times}(X, \mathbf{succ}(Y)) & \Rightarrow & \mathbf{plus}(\mathbf{times}(X, Y), X) \\
 & & \\
\mathbf{equal}(0, 0) & \Rightarrow & \mathbf{true} \\
\mathbf{equal}(\mathbf{succ}(X), \mathbf{succ}(Y)) & \Rightarrow & \mathbf{equal}(X, Y) \\
\mathbf{equal}(\mathbf{succ}(X), 0) & \Rightarrow & \mathbf{false} \\
\mathbf{equal}(0, \mathbf{succ}(Y)) & \Rightarrow & \mathbf{false}
\end{array}
$$

Rewrite Rules

For example, $3 \times 2 \times 1 = 1 \times 2 \times 3$, so that

$$\begin{array}{l}\mathbf{equal}(\\ \qquad \mathbf{times}(\,\mathbf{times}(\,\mathbf{succ}^3(0), \mathbf{succ}^2(0)\,), \mathbf{succ}(1)\,), \\ \qquad \mathbf{times}(\,\mathbf{times}(\,\mathbf{succ}(0), \mathbf{succ}^2(0)\,), \mathbf{succ}^3(0)\,) \\ )\end{array}$$

will eventually be rewritten into **true**.

Example 3 : Propositional Logic

- A propositional variable is a formula.
- 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 formulas.
- If $F_1, \ldots, F_n$ $(n \geq 0)$ are formulas, then $F_1 \wedge \cdots \wedge F_n$ and $F_1 \vee \cdots \vee F_n$ are formulas.

There are 4 states, in the last state, the number of subformulas is unbounded.

Interface of the Tree Types

Every type of tree must support the following operations:

1. Constructing a tree.
2. Determining the state of the tree.
3. Accessing subtree (as **const**).
4. Accessing and replacing subtrees.
5. (In case of a variable number of subtrees): Adding or removing a subtree.

Operations (**3**), (**4**), (**5**) depend on the state of the tree.

In principle, one can do without (**4**) if the number of subtrees is fixed, because one can construct a new tree.

Goals

Find a way of implementing these tree-like structures, such that

- They have a nice interface for the programmer who uses them:

  Automatic memory management.

  Value semantics (no side effects).

  Statically type checkable. (Problematic because of states).

- They are efficient:

  Minimize number of reallocations.

  Try to be local as much as possible.

Implementation Choices

- Which part is local, which part on the heap? (Alternatively: Where do we put the pointers?).
- Sharing/Not sharing
- Representing the state polymorphism.

Implementation and interface are connected, but there is still freedom.

Interface: Determining State

For determining state, there are the following options:

1. Use an enumeration type, allowing **switch**.
2. Use a chain of **if**s.
3. Use visitor (also called recursion operator).
4. Use inheritance.

(3) is theoretically nice, also very efficient, but too rigid. You always have to provide exactly one function per state.

Sometimes you want to distinghuish within a single state by operator, sometimes you want to skip states (rewriting), or handle two trees at the same time (when comparing).

Determining State (2)

(4) is also efficient, but forces you to scatter functions through different types.

In case of **isfree**( ) for λ-terms, one would introduce three subtypes **variable**, **application**, and **abstract**, and define an need overload of **isfree** in each of the subtypes.

I don't consider that a nice interface.

Rewriting would be scattered through different functions with aritificial names. Most of these functions would be intuitively meaningless. One could use **dynamic_cast**, but that would be **(2)**.

I choose **(1)** because of efficiency and flexibility.

Option **2** is still available, and you can still implement **3** by yourself if you want.

Selector

We decided to use an enumeration type for determining state on slide 21.

Usually there is more than one tree type in a single project. In my experience, using different enumeration types for the different types is annoying.

Use one enumeration type per project namespace.

It is a simple enum, called selector.

In my experience, **enum class** creates more problems than it solves.

The compiler is not helpful when checking completeness of switches.

Interface: Unbounded number of subtrees (1)

The last state for propositional logic (slide 17) allows an unbounded number of subformulas.

We need constructors for this state. The constructors must be able to construct from **initializer_list** and from other containers.

This leaves an annoying problem: One first has to construct the subtrees somewhere else, and move them into the formula after that:

```
std::vector< form > sub;
... push the subformulas to sub

auto f = form( sel_and,
               make_move_iterator( sub. begin( )),
               make_move_iterator( sub. end( )));
```

Interface: Unbounded number of subtrees (2)

Better: Allow **push_back( )** in states with unbounded number of subtrees:

```
auto f = form( sel_and );
f. push_back( ... );
f. push_back( ... );
f. push_back( ... );

// No additional container needed!
```

For implementation, this means that states with unbounded number of subtrees must be implemented like `std::vector`, i.e. have a **size** and a **capacity**.

Implementation: Where do we put the pointers?

The size of a tree is not known at compile time, hence most of it must be on the heap, so there have to be pointers somewhere.

But not everything needs to be on the heap: In a λ-term v, one can store the variable locally. In term $\lambda v\ t$, one also can store v locally.

For example:

```
struct term {
   selector sel;
   union {
      std::string var;
      struct { term* func; term* appl; } apply;
      struct { std::string var; term* body; } abstr;
   };
};
```

Where do we put the pointers? (2)

Alternatively:

```
struct ap;    struct ab;

struct term
{
   selector sel;
   union {
      std::string var;    ap* apply;   ab* abstr;
  };
};

struct ap { term func;   term arg; };
struct ab { std::string var;   term body; };
```

Implementation/Interface: To Share or Not to Share?

As soon as datastructures have something on the heap, sharing becomes a temptation.

Sharing is problematic in combination with value semantics.

If two values secretly share representation, then modifying one of them will change the other.

C^{++} also supports assignment on heavy objects like maps or vectors.

Hence it may appear at first sight, that not-sharing is the right approach in C^{++}.

To Share or Not to Share? (2)

Consider the following rewrite rule:

$$\mathbf{times}(X, \mathbf{succ}(Y)) \;\Rightarrow\; \mathbf{plus}(\mathbf{times}(X, Y), X)$$

Without sharing, one would need term iterators, and write this rule as follows:

```
if( p -> sel() == sel_times && p[1] -> sel() == sel_succ )
{
   term X = std::move( p[0] );
   term Y = std::move( p[1][0] );
   term X1 = X;                          // Copy constructor.
   *p = term( sel_plus,
          term( sel_times, std::move(X), std::move(Y) ),
          std::move(X1) );
}
```

To Share or Not to Share (3)?

All code will become 'copy aware'. The interface will become unusable.

(Probably assignment should be blocked, and reintroduced as **term clone( ) const** to avoid accidental copies.)

We conclude that we have to share.

We will do it in such a way that it is invisible through the interface.

Interface : Accessing Subtrees

Once we have determined the state of a tree, we know what subtrees it has, and what types they have.

Unfortunately, now something needs to be done that the C^{++} type system cannot check:

We need to cast the tree into a subtype that is guaranteed to have the subtrees.

Since we use a sharing representation, we can allow only offer **const** access to subtrees.

Accessing Subtrees (2)

In my implementation, I use **view**s which view the tree as being in a given state, e.g:

```
bool isfree( const std::string& var, const term& t ) {
  switch( t. sel( )) {
  case sel_var:
    return var == t. view_var( ). var( );
  case sel_apply: {
      auto p = t. view_apply( );
      return isfree( var, p. func( )) ||
             isfree( var, p. arg( ));
    }
  case sel_lambda: {
      auto p = t. view_abstr( );
      return var != p. var( ) && isfree( var, p. body( ));
}}}
```

Accessing Subtrees (3)

Note that matching, or visitor, does not have this problem.

Derived classes would have it.

If you want to use **switch** in the interface, it is unavoidable in current type systems. Possibly, it could be removed with refinement types.

Interface: Replacing Subtrees

If the number of subtrees is fixed, one can simply construct a new tree:

For example, when replacing `"v"` by `"w"` in
`term( sel_lambda, "v", t )`, one can simply call
`term( sel_lambda, "w", t )`.

This is not possible in the case of unbounded number of subtrees, because you have to put the other subtrees somewhere in a container. Making many updates will be quadratic.

Interface: Replacing Subtrees (2)

We want to minimize reallocations, so we need to support subtree replacement. We cannot give out a **non-const** reference because of sharing.

But we can allow Python-style updating:

```
auto p = t. view_binary( );
p. update_sub2( t );
```

For a field F, update$F(u)$ works as follows:

If **(1)** the tree is sharing (can be checked from the reference counter), and **(2)** u is distinct from the current value of field F, then make a unique copy, and assign u to F in the copy.

Interface: Replacing Subtrees (3)

How to check **(2)**? We are dealing with trees, so we don't want a deep equality check.

- Compare values up to the first pointer. When a pointer is encountered, compare the pointers.

I call this very equal. It is almost the same as identity (See Python or Java)

So the correct version is: **(2)** u is not very equal to the current value of field F, then make a local copy, and assign u to F in the copy.

Note that all of this this applies only to fields below a pointer. If field F is local, it can be simply assigned.

Also note that it should be never allowed to assign to the selector.

Interface: Replacing Subtrees (4)

Now we can implement rewriting:

(I show in the code)

Interface: Replacing Subtrees (5)

Do we now have the optimal reallocation strategy?

Interface: Replacing Subtrees (6)

Problem: `topdown` will always reallocate the full path towards a change, even when it is not shared.

We cannot:

- Use `std::move`.
- Make the argument of `topdown` **const**.

Interface: Replacing Subtrees (7)

Solution: Borrow subtrees during rewriting.

For a field F, define extr$F$ as follows:

- If the tree is sharing, then return the field F as copy. The result will be sharing.
- If the tree is not sharing, then return field F as move.

(Again, this applies only to fields below a pointer. For local fields, extr$F$ returns the moved value.

If somewhere on a path in the tree, a field is sharing, extracted subtrees will be sharing all the way to the end of the path.

If no field is sharing, the extract trees will no be sharing.

(I show in the code, and demonstrate that it works).

I think that this is optimal.

Implementation: Use `std::variant`?

`std::variant` is a great thing, and it seems natural to use it here, but it introduces a redundancy:

We decided on slide 23 to use a **selector** tag for determining the state of the tree. This makes it possible to use **switch** and **if**.

If we use `std::variant`, it will have its own tag inside. This will introduce some redundancy. It's small but it is there.

In order to get rid of this redundancy, one would need a different type of **variant**, where the user has access to the state tag, and different values of the tag can represent the same state.

Variant/Derived Classes

Maybe something like

```
variant< option< binary, sel_and, sel_or >,
         option< unary, sel_not >>
```

We could also combine **selector** with derived classes. It would introduce the same redundancy, because C^{++} uses RTTI to determine the concrete type. If we would use derived classes, only the **selector** could be stored locally, so the representations on slides 26 and 27 would be impossible.

Implementation: Merging Vector and Smart Pointer

Since we decided to share parts of trees, we need memory management. Terms are never cirular, so one can safely use reference counting.

Wherever a pointer occurs in the representation, we add a reference counter to the data being pointed to.

This creates some problems in states with unbounded number of subtrees.

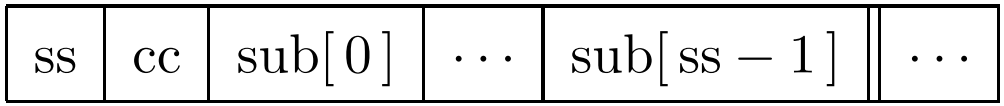
Merging Vector and Smart Pointer (2)

This is propositional logic (See slide 17):

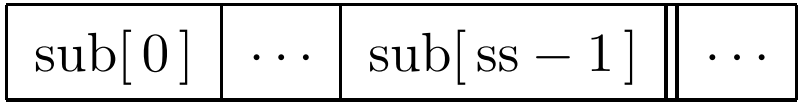
```
struct un;     struct bin;    struct nn;
struct form {
   selector sel;
   union {
      std::string var;
      struct un *unary;
      struct bin *binary;
      struct nn *n_ary;
   };
};
struct un { form sub; };
struct bin { form sub1;    form sub2; };
struct nn { size_t ss;    size_t cc;    form sub[ ]; };
```

Merging Vector and Smart Pointer (3)

`struct nn` is problematic: We need to allocate a fixed part `size_t ss; size_t cc;` followed by a repeated part of unknown size, consisting of `form`s:



I think that C^{++} has no nice way of doing this. `new form[ cc ]` only allocates:



Instead, one could use:

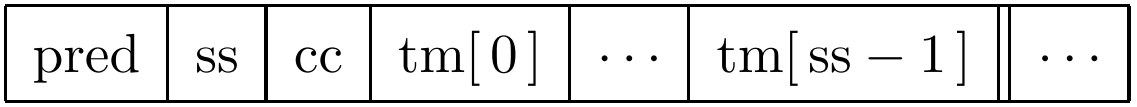
```
struct nn { size_t ss;    size_t cc;      form* sub };
   // sub[ cc ] allocated separately.
```

but it creates another indirection.

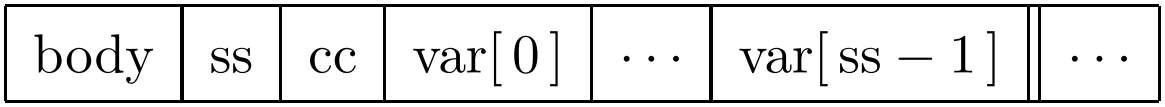
Merging Vector and Smart Pointer (4)

In addition to the previous slide, some states could have both fixed and repeated fields, e.g. if we would define predicate logic, we would have:

- If p is a predicate name, $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula:



- If F is a formula, $v_1, \ldots, v_n$ are variables, then $\forall v_1 \cdots v_n \ F$ and $\exists v_1 \cdots v_n \ F$ is a formula:



Merging Vector and Smart Pointer (5)

I introduce two data types. Their interface is not nice, but they are used only internally in implementation of tree.

- `scalar<S>` is used for states without repeated trees.
- `scalar_repeated<S,R>` is used for states with repeated trees.

For both types, we pretend that pointer (to them) is the main type.

Merging Vector and Smart Pointer (6)

Both `scalar` and `scalar_repeated` have the following operations (we still pretend that the pointer is the main datatype):

```
template< typename S >
scalar<S> * takeshare( scalar<S> * sr )
   // Take a share in sr.
   // (Increase its reference counter)

template< std::destructible S >
void dropshare( scalar<S> * sr )
   // Drop a share in sr. If it was the last share,
   // destroy its fields, and free the memory.
```

Merging Vector and Smart Pointer (7)

```
template< typename S >
bool iswritable( const scalar<S> * sr )
```

True if the reference counter equals 1.

```
template< std::copy_constructible S >
scalar<S> * replacebywritable( scalar<S> * sr )
```

Returns a non-sharing copy, and drop our share in **sr**. Always pretend that the result is new.

Merging Vector and Smart Pointer (8)

In addition, `scalar_repeated` has the following operations:

```
template< typename S, typename R >
scalar_repeated<S,R> *
push_back( scalar_repeated<S,R> * sr, const R& r )

scalar_repeated<S,R> *
pop_back( scalar_repeated<S,R> * sr )

scalar_repeated<S,R> *
clear( scalar_repeated<S,R> * sr )
```

All three methods may decide to make a non-shared copy. Hence the returned pointer need not be equal to **sr**.

Generator

Implementing tree types by hand is very tedious. A simple tree type (like λ calculus) can be done in a day, but bigger ones may take several days. Change management is impossible. ZF set theory has 11 states, 39 selector values. With a generator, the tree type can be generated in a minute.

For λ-terms:

```
%define term ( sel_var )
   // sel_var is the moved-out state.

%option var { sel_var } => # var : std::string
%option apply { sel_apply } => func : term, arg : term
%option abstr { sel_lambda } =>
          body : term, # var : std::string
```

Generator (2)

For comparison, in Haskell, the definition would be :

```
data Lambda =
    Var String |
    Apply Lambda Lambda |
    Abstr String Lambda deriving ( Show, Read )
```

This is the definition of function **isfree**:

```
isfree var1 ( Var var2 ) =
      ( var1 == var2 )
isfree var1 ( Apply t1 t2 ) =
      ( isfree var1 t1 ) || ( isfree var1 t2 )
isfree var1 ( Abstr var t ) =
      ( var1 /= var ) && ( isfree var1 t )
```

Generator (3)

Term rewriting (Example 2 on slide 14):

```
%define term ( sel_zero )
%option nullary { sel_zero, sel_false, sel_true } =>
%option unary { sel_succ, sel_not, sel_factorial } =>
          sub : term
%option binary { sel_plus, sel_times, sel_equals } =>
          sub1 : term, sub2 : term
%option string { sel_string } =>
          [ sub : term ], # str : std::string
```

Refinement Types?

Assumption: Static types > dynamic types > typed instructions.

1. Typed instructions: Types are built-in in the instructions. (Assembly, *C*-style **printf**).
2. Dynamic Typing: Data carries type information. At run time this information is used for selecting the right operations. (C^{++} virtual functions, Python)
3. Static Typing. Compiler checks at compile time. After that, it selects the right typed instructions. At run time, no type checking is needed. (Most of C^{++}.)

Refinement Types (2)

What am I using?

```
bool isfree( const std::string& var, const term& t ) {
switch( t. sel( )) {
   ....
  case sel_apply: {
      auto p = t. view_apply( );
         // Problematic point. It's a typed instruction.
      return isfree( var, p. func( )) ||
             isfree( var, p. arg( ));
    }
```

Static type checking should be able to do without the view.

`t. sel( ) == sel_apply` implies that `t` is in state `apply`, so we know that `t` has fields `func( )` and `arg( )`.

Refinement Types (3)

A similar problem occurs in constructors:

```
term t = ...

t = term( sel_lambda, "var", t );
   // OK.

t = term( sel_appl, "var", t );
   // Not OK.
```

It should be possible to restrict enumeration types to subsets.

Currently not statically checkable. The tree generator inserts optional dynamic checks in the constructors.

Summary, Conclusions

I believe that I solved the problem of how to implement logic in C^{++}, at least on the interface side.

For the implementation, I should probably have a look at efficiency. (Have a look in compiler-explorer, make measurements.)

It would be nice to have an extended **new** that can allocate with a fixed prefix: `new { S; R }(s)[cc]` (`s` is initializer for the prefix `S`, while `cc` is number of repetitions of `R`).

It is nice to dream about refinement types in C^{++}. Only possible with value semantics.

One missing feature: Currently, the tree generator cannot generate template classes.

Thanks!

Thanks to Aleksandra Kireeva, Kaster Kumarbek, Akhmetzhan Kussainov, Dina Muktubayeva, Cláudia Nalon, Olzhas Zhangeldinov, and Olzhas Zhumabek.

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