A Recursive Inclusion Checker for Recursively Defined Subtypes

Hans de Nivelle

Nazarbayev University, Nur-Sultan City, Kazakhstan

Wrocław, 20.01.2022

Long Term Goal of this Project

In this talk, I plan to give much motivation, and little technicality.

The paper contains no proofs, but describes the main notions, and the algorithm.

Long term goal:

Obtain a programming language that is easy to use for logicians. (Think about theorem provers, normal form transformations, verification, lambda calculus)

These things are difficult to implement in existing programming languages.

What is special about Logic

Logic makes extensive use of recusively defined data structures with many constructors, that are very different.

For example:

We assume a countably infinite set of function symbols. The set of terms is recursively defined as follows:

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

A bigger example is on the next slide.

First-Order Logic

The set of first-order formulas is recursively defined as follows:

- If p is a predicate symbol, $t_1, \ldots, t_n$ $(n \geq 0)$ are terms, then $p(t_1, \ldots, t_n)$ is a formula.
- If t_1, t_2 are terms, then $t_1 \approx t_2$ is a formula.
- $\bot$ and $\top$ are formulas.
- If F is a formula, then $\neg F$ is a formula.
- If $F_1, \ldots, F_n$ are formulas, then $F_1 \wedge \cdots \wedge F_n$, and $F_1 \vee \cdots \vee F_n$, are formulas.
- If F_1 and F_2 are formulas, then $F_1 \rightarrow F_2$ and $F_1 \leftrightarrow F_2$ are also formulas.
- If v is a variable, F is a formula, then $\forall v\ F$ and $\exists v\ F$ are formulas.

Problems with C^{++}

- Code is long and repetitive between different inductive types. (Think about printing, equality testing, checking size of a term.) Although such code is similar, C^{++} has no way of sharing it.
- If you want to get the subfields that belong to a given subtype, you have to cast. The type system of C^{++} is of no help.
- You cannot regroup cases in different ways.

Python

1. Lots of problems with reference semantics (unwanted sharing). It causes side effects that are hard to control.
2. Python has automatic memory management. But, because the problem of memory management was solved in C^{++}, I consider this of no advantage.
3. No static types.
4. Disgustingly slow.
5. The edit/reload cycle is slower than the edit/recompile cycle.

Haskell

1. Haskell has inductive types, that is nice.
2. You have to invent names for the constructors. There is conflict when defining similar constructors, to merge or not to merge, that's the question. Two ways to do the same thing is bad.
3. You cannot write concrete formulas in an acceptable way in code, because the type system is too rigid. Constructors are always explicit.
4. Matching is nice, but only if your partition exactly follows the constructors.
5. Functional style is too restrictive. It is difficult to represent state.

Root Cause of the Problems?

Fundamental Type Constructors

product Completely solved since 90-ies. We have `struct/class`, `record`, `tuple, pair`.

union Still a big nightmare in every language that I know: In C^{++}: Pointer casting, use of `nullptr`, `std::variant< >`, `npos`, `std::optional< >`, inheritance, use enumeration type.

In Java: Inheritance, use of `switch` on enumeration type.

In functional languages: Matching, but too rigid, and inefficient.

(repetition) Solved in acceptable fashion. `std::vector`, C-style arrays, functional languages have lists.

I believe we are still struggling with union, both in high level and in low level languages.

Definition of First-Order Logic

```
fol := struct
op : selector
switch op
?atom =>
   pred: identifier,  arg : term[]
?equals =>
   left : term,  right : term
?not =>
   sub : fol
?and | ?or =>
   sub : fol[]
?implies | ?equiv =>
   left : fol,  right : fol
?forall | ?exists =>
   body : fol,  var : identifer[],  type : simptype[]
```

You may wonder, where did $\bot$ and $\top$ go? They are special cases of $\vee$ and $\wedge$.

The definition of `fol` can be viewed as a tree that branches dependent on the value of `op`.

At the leaves of the tree, the last fields can be repeated. This is indicated by writing `[ ]` after the type declaration. Repeated fields implicitly define an array.

The type format combines the three type constructors **product**, **union** and **repetition** into a single format.

It is designed to be convenient in use, while at the same time having efficient low-level representation.

Defining Subtypes

Subtypes are essential to our language design.

I call them adjectives because they are not real subtypes (subtypes are more permanent).

Examples:

Constructed by binary operator:

op(?implies∨?equiv).

Constructed by n-ary operator:

op(?and∨?or).

Constructed by a quantifier:

op(?forall∨?exists).

More Adjectives

Literal:

$$\text{literal:fol} := \bigvee \begin{cases} \text{op}(?\text{atom} \vee ?\text{equals}) \\ \text{op}(?\text{not}) \wedge \text{sub}(\, \text{op}(?\text{atom} \vee ?\text{equals})\,) \end{cases}$$

Negation normal form (NNF):

$$\text{nnf:fol} := \bigvee \begin{cases} \text{literal} \\ \text{op}(?\text{and} \vee ?\text{or}) \wedge \text{sub}(\forall \text{nnf}) \\ \text{op}(\, ?\text{forall} \vee ?\text{exists}\,) \wedge \text{body}(\text{nnf}). \end{cases}$$

Adjectives can be viewed as a non-deterministic, finite tree automata.

Data Format

Data are just trees. They are not aware of their type, and the adjectives that they satisfy.

Primitive types are : **unit**, **bool**, **char**, **integer**, **selector**, **index**, **double**.

selector is a single type that replaces all enumeration types. Its members have form ?id.

Trees are recursively defined as follows:

- Anything that belongs to one of the primitive datatypes is a tree.
- If $t_1, \ldots, t_n$ are trees, then $(t_1, \ldots, t_n)$ is a tree.

The formula

$$\forall xy\ (\ \neg s(x) > y \vee y = s(0)\ )\ )$$

can be represented as

```
( ?forall,
   ( ?or;
     ( ?not,
        ( ?atom, > ; (s; (x; )), (y; ) )
     ),
     ( ?equals, (y; ), ( s; (0; ))
     )
   ); x,y
).
```

I used ; to separate scalar arguments from repeated arguments.
This is not part of the implementation.

Both types and adjectives are tree automata. Hence they can be evaluated in linear time.

Note that linear complexity is too much at run time, but acceptable for static type checking.

Inclusion Checking Between Adjectives (Title of this Talk)

Inclusion checking between adjective is needed for the following:

- Conditional statements:

```
if f.op == ?equals :
   print( f.left, '=', f. right )
```

- Checking preconditions of fields: (same example as above)

- Checking completeness and exclusiveness of switch statements:

```
switch f.op
?atom =>
?equals =>
?not =>
?implies | ?equiv =>
?and | ?or =>
?forall | ?exists =>
```

- Overload resolution for function calls:

```
b : bool = tableaux(f)

function tableaux( f : fol ) => bool
   f = nnf(f); return tableaux(f)

function tableaux( f : fol @ nnf ) => bool
   # tableaux procedure

nnf( f : fol ) => fol @ nnf
   # transformation to negation normal form.
```

Advantages of Adjectives over Matching

Adjectives make the implementation choices invisible to the programmer. This does not only apply to low-level choices, but also to high-level choices. (To merge on not to merge).

We have always been compromising/negotiating with the programming language, but there is no need to do that any more.

Inclusion Checking Between Adjectives

As with most logics, inclusion $A_1 \subseteq A_2$ can be checked by checking satisfiability of $A_1, \neg A_2$.

Completeness of cases in a **switch** can be checked by checking that $A \subseteq A_1 \vee \cdots \vee A_n$.

Disjointness of cases can be checked by checking that A_i, A_j is unsatisfiable when $i \neq j$.

(It still has to be checked at runtime, which case of **if** or **switch** must be chosen. That is another problem.)

I introduce a tableaux style prover that checks satisfiability of sets of adjectives.

It works on a normal form called path normal form.

It uses a blocking rule to cut off infinite cycles.

Path Normal Form

We formula is in path normal form (PNF) if negation is always applied on identifiers.

Bringing an adjective into PNF:

$$
\begin{array}{lcl}
\neg\left(A_1 \vee \cdots \vee A_n\right) & \Rightarrow & \neg A_1 \wedge \cdots \wedge \neg A_n \\
\neg\left(A_1 \wedge \cdots \wedge A_n\right) & \Rightarrow & \neg A_1 \vee \cdots \vee \neg A_n \\
\neg f(A) & \Rightarrow & f(\neg A) \\
\neg \forall A & \Rightarrow & \exists \neg A \\
\neg \exists A & \Rightarrow & \forall \neg A \\
\neg\neg A & \Rightarrow & A.
\end{array}
$$

Tableaux Procedure

Let $\mathcal{A}$ be a set of adjectives. $\mathcal{A}$ is closed if it contains

- $\bot$
- a complementary pair $A, \neg A$
- A pair c_1, c_2, where $c_1 \neq c_2$.

Tableaux (2)

- If $\mathcal{A}$ contains $A_1 \wedge \cdots \wedge A_n$, then replace it by $A_1, \ldots, A_n$.
- If $\mathcal{A}$ contains $A_1 \vee \cdots \vee A_n$, then for i $(1 \leq i \leq n)$, define $\mathcal{A}_i$ as $\mathcal{A}$ with $A_1 \vee \cdots \vee A_n$ removed and $\mathcal{A}_i$ added.

  If all $\mathcal{A}_i$ can be closed, then $\mathcal{A}$ is closed. Otherwise, $\mathcal{A}$ is open.

Tableaux (3)

- If $\mathcal{A}$ contains an identifier v that is defined as adjective A, then replace v by $\mathrm{pnf}(A)$.
- If $\mathcal{A}$ contains $\neg v$, and v is defined as adjective A, then replace $\neg v$ by $\mathrm{pnf}(\neg A)$.

Tableaux (4)

- If $\mathcal{A}$ contains $\forall A$, then replace it by $\mathbf{empty} \vee (\, \mathbf{first}(A) \wedge \mathbf{rest}(\forall A)\,)$.
- If $\mathcal{A}$ contains $\exists A$, then replace it by $\neg\mathbf{empty} \wedge (\, \mathbf{first}(A) \vee \mathbf{rest}(\exists A)\,)$.

Tableaux (5)

If none of the previous rules can be applied, and $\mathcal{A}$ contains no subformulas of form $f(A)$, then $\mathcal{A}$ is open.

Otherwise, create all sets

$$\mathcal{A}_f = \{A \mid f(A) \in \mathcal{A}\} \cup A_{T_f}.$$

Here A_{T_f} is the adjective of the type of f.

For each of these $\mathcal{A}_f$, do the following:

1. If there is an $\mathcal{A}'$ on the branch towards the current $\mathcal{A}$, s.t. $\mathcal{A}' \subseteq \mathcal{A}_f$, then $\mathcal{A}$ is closed.
2. Apply the rules on the previous slides on $\mathcal{A}_f$. If this results in closure, then $\mathcal{A}$ is closed.

If no $\mathcal{A}_f$ was closed, then $\mathcal{A}$ is open.

Concluding Remarks

The paper gives a more algorithmic description of the tableaux procedure.

The procedure terminates, because the set of possible $\mathcal{A}$ is finite, and eventually a blocking will occur.

Blocking is sound, because repetitions can be removed from an open branch.

I implemented the procedure, and its performance seems good enough. If not, I will think about adding learning.

Static type checking is a far-away goal, building an interpreter (using run time type resolution) seems to be within reach with the theoretical tools that we have now.