# An Easy, Almost Functional Language for Logicians

Hans de Nivelle

03.10.2019

## Long Term Goals

I like to implement logical algorithms, for example proof checkers, or theorem provers.

This turns out difficult in existing programming languages, and until now I have not found a language that I am happy with.

On the next slide, I will define first-order logic as an example. It is still simplified. In reality, one probably wants to add simple types to it.

## Terms

We assume a countably infinite set of function symbols $\mathcal{F}$. The set of terms is recursively defined as follows:

- If $t_1, \ldots, t_n$ $(n \geq 0)$ are terms, $f \in \mathcal{F}$, then $f(t_1, \ldots, t_n)$ is a term as well.

## 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 $F_1$ and $F_2$ are formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$ and $F_1 \leftrightarrow F_2$ are 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.
- Enumeration types don't work. You have to define a new one for every inductive type. You have to prefix them (in order to avoid name conflicts). Compiler is too dumb to check for exhaustiveness, so you get annoying warnings all the time.
- 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.

## Enumeration Types

```
enum logop { fol_atom, fol_equals,
             fol_false, fol_true,
             fol_not,
             fol_and, fol_or, fol_implies, fol_equiv,
             fol_forall, fol_exists };
```

1. We need to scope them, because they conflict with reserved words, and with each other between different types.
2. One hopes that compiler will check fall-through in `switch` statements, and unreachability, but it doesn't.

```
X f(  )
{
   switch( op )
   { complete list of cases }
}
```

   Compiler still whines about missing return.

Enumeration types are not useful for us. Why do they even exist?

Of course $C^{++}$ has advantages too:

- When it works, static type checking is useful. It catches a lot of problems at compile time.
- It is reasonably fast.
- It is flexible.

## First Attempt

- Define only one inductive types that is general enough to represent all the other inductive types.
- At this point, every form of type checking is lost.
- Design a type checker (as part of a library) that checks your algorithms in advance.
- If you add a syntax to it, you essentially have a programming language.

## Python

1. Lots of problems with reference semantics. It causes side effects that are hard to control.
2. Python has automatic memory management. But, because the 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. But you cannot write concrete formulas in an acceptable way, because the type system is too rigid. One has to cast everything during construction.
3. Functional style is too restrictive. It is difficult to represent state.
4. Matching is incompatible with information hiding.

## Root Cause of the Problems?

We have a problem with partial functions (fields that do not always exist, and functions that do not always exist.)

Alternatively, one can say: Functions with preconditions.

Question: Why do there exist partial functions?

## Failure of the Type System

I think : Because the type system is too weak to check preconditions.

A partial function is usually total on a (meaningful) subtype.

If the type system allows subtypes, and is able to check them, existence of fields (and functions) can be statically checked.

## A New Programming Language?

See file **easy_fol.txt**.

Main types are called substantive. Refinements of main types are called adjectives.

I come to the technicalities:

Definition: We assume a few primitive types: **bool**, **char**, **integer**, **selector**, **identifier**, **double**.

**selector** is a single type that replaces all enumeration types.

Definition:

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.

(Only one inductive data type needs to be implemented.)

The formula

$$\forall xy\ (\ s(x) > y \vee y > s(x)\ )$$

can be represented as

```
( ?forall,
   (x,y),
   ( ?or,
     ( ?atom, >,
        ( (s, (x, ())), (y, ()) )
     ),
     ( ?atom, >,
        ( (y, ()), ( s, (x, ())) )
     )
   )
).
```

## A simple **struct**

In order to show what the problems are, and how they can be solved, I use a small, artificial example:

```
struct variant :
   double d
   integer{1,2,3,4,5,6} i
   options:                          // Uses i.
   {1,2} :
      integer x
      variant v
   {3,4} :
      bool b
      double x
   {5,6} :
      variant x
```

Note that field `.x` is overloaded:

$$(3.1415, 1, 100, (2.718, 3, \mathbf{false}, 3.16)\ ).\mathbf{x} = 100,$$

$$(0.333333, 3, \mathbf{true}, 0.57721).\mathbf{x} = 0.57721,$$

$$(0.14285, 5, (12.5, 3, \mathbf{true}, 0.9999)\ ).\mathbf{x} = (12.5, 5, \mathbf{true}, 0.9999).$$

It has different types, and different positions (offsets), dependent on field `.b`

## A Possible Adjective

```
adjective restr( variant ) :=
    .i{1}.x>=5.v^restr
    .i{3}.b{true}
```

(Different lines means: Disjunction)

Note there are problems with overloading in the adjectives too. Which `.x` is meant?

## A Program Fragment

```
integer sum( variant^restr v ) :
   integer s := 0
   while v.i != 3 :
      s := s + v.x
      v := v.v
   s := s + double2int( v.x )
   return s
```

Which `.x` is meant? Is it guaranteed that `.v` exists?

What happens when the loop is iterated? Will it work?

## Let's do the Easy Things First:

Translate adjective into FOL:

$$\text{variant}_{\text{P}}(v) \rightarrow (\ \text{restr}(v) \leftrightarrow \bigvee \left\{ \begin{array}{l} v.i \in \{1\} \wedge v.x \geq 5 \wedge \text{restr}(v.v) \\ v.i \in \{3\} \wedge v.b \in \{\textbf{true}\} \end{array} \right. )$$

Translate the Program into a flow diagram, and unravel complicated expressions:

| | | |
|---|---|---|
| $p_0$ | $s := 0$ | $p_1$ |
| $p_1$ | $?\ (v.i = 3)$ | $p_5$ |
| $p_1$ | $?\ (v.i \neq 3)$ | $p_2$ |
| $p_2$ | $s := s + v.x$ | $p_3$ |
| $p_3$ | $v := v.v$ | $p_1$ |
| $p_5$ | $w := \text{double2int}(v.x)$ | $p_6$ |
| $p_6$ | $s := s + w$ | $p_7$ |
| $p_8$ | return $s$ | $p_8$ |

## Type Checking and Overload Resolution

Function names and field names need two sets of identifiers:

- The identifiers that the user uses (when defining adjectives and code) are called unchecked.
- They have to be replaced by checked identifiers, in order to obtain an executable program.

I assume that checked identifiers always have an index, while unchecked identifiers have not.

Table of possible field replacements. In order to replace an unchecked field by a checked field, one has to prove the precondition.

| **unchecked** | **checked** | **precondition** |
|---|---|---|
| $v.d$ | $v.d_0$ | $\text{variant}_{\text{P}}(v)$ |
| $v.i$ | $v.i_0$ | $\text{variant}_{\text{P}}(v)$ |
| $v.x$ | $v.x_1$ | $\text{variant}_{\text{P}}(v),\ v.i_0 \in \{1,2\}$ |
| $v.v$ | $v.v_1$ | $\text{variant}_{\text{P}}(v),\ v.i_0 \in \{1,2\}$ |
| $v.b$ | $v.b_2$ | $\text{variant}_{\text{P}}(v),\ v.i_0 \in \{3,4\}$ |
| $v.x$ | $v.x_2$ | $\text{variant}_{\text{P}}(v),\ v.i_0 \in \{3,4\}$ |
| $v.x$ | $v.x_3$ | $\text{variant}_{\text{P}}(v),\ v.i_0 \in \{5,6\}$ |

## Dealing with the Adjective

$$\text{variant}_{\text{P}}(v) \rightarrow (\text{restr}(v) \leftrightarrow \bigvee \left\{ \begin{array}{l} v.i \in \{1\} \wedge v.x \geq 5 \wedge \text{restr}(v.v) \\ v.i \in \{3\} \wedge v.b \in \{\textbf{true}\} \end{array} \right. )$$

Since we can prove $\text{variant}_{\text{P}}(v)$, we can overload both $v.i$ into $v._{i_0}$. We got

$$\text{variant}_{\text{P}}(v) \rightarrow (\text{restr}(v) \leftrightarrow \bigvee \left\{ \begin{array}{l} v.i_0 \in \{1\} \wedge v.x \geq 5 \wedge \text{restr}(v.v) \\ v.i_0 \in \{3\} \wedge v.b \in \{\textbf{true}\} \end{array} \right. )$$

Now we can resolve both $v.v$ and $v.b$ : Result is:

$$\text{variant}_{\text{P}}(v) \rightarrow (\text{restr}(v) \leftrightarrow \bigvee \left\{ \begin{array}{l} v.i_0 \in \{1\} \wedge v.x_1 \geq 5 \wedge \text{restr}(v.v_1) \\ v.i_0 \in \{3\} \wedge v.b_2 \in \{\textbf{true}\} \end{array} \right. )$$

Finally, we can resolve $\leq$ into $\leq_{\text{int}}$, and $\in$ into $\in_{\text{int}}$ .

## Dealing with the Code

We use abstract interpretation:

Let $p_s$ be the starting state. Start by setting $\Lambda(p_s) = \text{variant}_{\text{P}}(v) \wedge \text{restr}(v)$, and $\lambda(p) = \bot$ for all other states $p \neq p_s$.

- For every statement $s$ between $p$ and $p'$, replace any existing checked identifiers by their corresponding unchecked identifiers.

  After that, use $\Lambda(p)$ to resolve the overloads in $s$ again. If this fails, produce an error message.

- For every program point $p$, let $(p_1, s_1, p), \ldots, (p_n, s_n, p)$ be all the statements that lead to it. Replace $\Lambda(p)$ by

$$\text{simp}(\ \Lambda(p) \vee \text{next}(s_1, \Lambda(p_1)) \vee \cdots \vee \text{next}(s_n, \lambda(p_n))\ ).$$

Keep on doing this, until a fixed point is reached.

## Definition of **Next**

For a statement $s \in S$, and a formula $F$, we define $\text{next}(s, F)$ :

- For an assignment $x := t$ and formula $F$, let $x'$ be a variable that does not occur in $F$ or $t$, and that is distinct from $x$. Then

  $$\text{next}(x := t,\ F) = \exists x'\ (\ x = t[x := x'] \wedge F[x := x']\ ).$$

  There is no need to distinguish between assignment and initialization. If we are dealing with initialization, the substitution $x := x'$ has no effect.

- For a check statement $?\ G$, define $\text{next}(?\ G, F) = F \wedge G$.

- for a never statement, define $\text{next}(\text{never}, F) = \bot$.

- for a return statement return $t$, define $\text{next}(\text{return } t,\ F) = \bot$.

There is no guarantee that this procedure terminates. It all depends on simp().

Main thing is to avoid nested $\exists$, and to restrict statements about **integer**.

(Just throw away everything that seems too hard.)

- In the first step, $\Lambda(p_0) = \text{variant}_{\text{P}}(v) \wedge \text{restr}(v)$.
- $\text{next}(s := 0,\ \text{variant}_{\text{P}}(v) \wedge \text{restr}(v)\ ) =$

$$\exists s'\ (s = 0 \wedge \text{variant}_{\text{P}}(v) \wedge \text{restr}(v)\ ).$$

Since $\exists s'$ is not used, this can be simplified into

$$s = 0 \wedge \text{variant}_{\text{P}}(v) \wedge \text{restr}(v).$$

We can resolve both instances of $v.i$ that start in $p_1$ and replace them by $v.i_0$.

- $\text{next}(\ ?(v.i_0 \neq 3),\ s = 0 \wedge \text{variant}_{\text{P}}(v) \wedge \text{restr}(v)\ ) =$

$$v.i_0 \neq 3 \wedge s = 0 \wedge \text{variant}_{\text{P}}(v) \wedge \text{restr}(v).$$

This makes it possible to resolve $(p_2)\ \ s := s + v.x\ \ (p_3)$ into $(p_2)\ \ s := s +_{\text{int}} +v.x_1\ \ (p_3)$.

## Checking **switch**es

When a fixed point is reached, the **switch** statements need to be checked:

If there exists more than one statement starting in $p$, let $(p, s_1, p_1), \ldots, (p, s_n, p_n)$ be all statements that start in $p$.

- Each statement $s_i$ must have form $s_i =? f_i$, for some formula $f_i$.
- For each pair $i_1 \neq i_2$, it must be possible to prove $\Lambda(p) \wedge f_{i_1} \wedge f_{i_2} \rightarrow \bot$ (No overlap).
- It must be possible to prove $\Lambda(p) \rightarrow (f_1 \vee \cdots \vee f_n)$ (All cases are covered).

## Open Questions

- Is it worth implementing such language? Will it result in papers? Will it attract users? Does it solve my own problems? Is it useful in teaching? What more is needed?