Lecture Notes on
Types as Predicates

15-317: Constructive Logic
Frank Pfenning*

Lecture 16
April 20, 2021

1 Introduction

One of the significant problems in using Prolog is the lack of static typing. Prolog inherited this feature from *predicate calculus*, where it roots lie. In the foundational study of propositions and quantification, types are often omitted because it is said they can already be expressed. For example, instead of saying $\forall x{:}\mathsf{nat}.\, A(x)$ we can say $\forall x.\, \mathsf{nat}(x) \supset A(x)$ if we have a *predicate* nat that expresses the *type* nat. Similarly, we can express $\exists x{:}\mathsf{nat}.\, A(x)$ as $\exists x.\, \mathsf{nat}(x) \wedge A(x)$. Predicates that provide an extensional representation of types are not difficult to come by. For example, we can define (and have defined) the natural numbers with two constructors z and s and the rules

$$\frac{}{\mathsf{nat}(\mathsf{z})\ \mathit{true}}\ \mathsf{nat}_z \qquad \frac{\mathsf{nat}(N)\ \mathit{true}}{\mathsf{nat}(\mathsf{s}(N))\ \mathit{true}}\ \mathsf{nat}_s$$

Foundationally, this approach may have some merit, but it also has some problems. One is that propositions such as $\forall x{:}\mathsf{nat}.\, \mathsf{append}(x, \mathsf{nil}, x)$ which are *meaningless* become either true or false when written in an untyped way: $\forall x.\, \mathsf{nat}(x) \supset \mathsf{append}(x, \mathsf{nil}, x)$. In a language like Prolog this has dire consequences because we compute with intuitively meaningless propositions and bogus proofs, leading to unexpected behavior. A second problem is that the untyped approach does not extend well to higher-order logic, where we want to quantify over propositions and not just data. In

*Edits by André Platzer

fact, several times in history well-regarded researchers such as Frege or Church have attempted to avoid the organizing principles of types, leading to inconsistent logics.

In this lecture we explore the question if we may still be able to use the idea of defining types via (unary) predicates and obtain something we can statically check and that executes efficiently at the same time. The answer is "yes", and the lessons learned from this has also had some impact on functional programming in the guise of *refinement types* [FP91, DP03, Dav97].

There have been multiple approaches to types in the logic programming community (see [Pfe92] for various articles and technical realizations). We will not go into a specific decidable language of types, although much of what we show in this lecture applies to several systems that are different in their respective technical details.

2 Modes and Types

Let's reconsider something simple like addition on unary natural numbers.

$$\frac{}{\mathsf{nat}(\mathsf{z})}\ \mathsf{nat}_z \qquad \frac{\mathsf{nat}(N)}{\mathsf{nat}(\mathsf{s}(N))}\ \mathsf{nat}_s$$

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

Now we want to show the combined *mode and type specification*:[1]

$$\mathsf{plus}(+\mathsf{nat}, +\mathsf{nat}, -\mathsf{nat})$$

which we interpret as follows: if proof search is initiated with a goal $\mathsf{plus}(m, n, P)$ where $\mathsf{nat}(m)$ and $\mathsf{nat}(n)$ and proof search *succeeds*, then $P = p$ with $\mathsf{nat}(p)$.

Rigorously, we would have to prove this by induction over the structure of computation (that is, proof search). In the absence of such an operational semantics, we prove it by induction over the structure of the rules. Assume we are searching for a proof of $\mathsf{plus}(m, n, P)$ for a variable P and terms m and n with $\mathsf{nat}(m)$ and $\mathsf{nat}(n)$.

Case: Rule pz. We know $\mathsf{nat}(\mathsf{z})$ (which adds no new information because nat_z already knew this) and $\mathsf{nat}(n)$. Applying the rule will succeed and instantiate $P = n$ and so $\mathsf{nat}(P)$.

[1] Don't confuse the $+$ in $\mathsf{plus}(+\mathsf{nat}, +\mathsf{nat}, -\mathsf{nat})$ with plus for addition. It refers to the mode where that argument is given as input.

Case: Rule ps. We know $m = \mathsf{s}(m')$ and $\mathsf{nat}(\mathsf{s}(m'))$ and also $\mathsf{nat}(n)$. From the first fact, by inversion (only rule nat_s could be used to prove this) we obtain $\mathsf{nat}(m')$. Now we can appeal to the induction hypothesis: if the subgoal $\mathsf{plus}(m', n, P')$ succeeds, then $P' = p'$ for some term p' and $\mathsf{nat}(p')$. Then $\mathsf{nat}(\mathsf{s}(p'))$ by rule nat_s.

So far, there is not much new or interesting in this when compared to types as we know them from functional languages. But we can define new and interesting types as predicates and reason about them in the same style. For example, we can distinguish the even and odd numbers and reason about the properties of addition.

$$\frac{}{\mathsf{even}(\mathsf{z})}\ \mathsf{ev}_z \qquad \frac{\mathsf{odd}(N)}{\mathsf{even}(\mathsf{s}(N))}\ \mathsf{ev}_s \qquad \frac{\mathsf{even}(N)}{\mathsf{odd}(\mathsf{s}(N))}\ \mathsf{od}_s$$

Let's try to check that adding two even numbers results in an even number.

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

$$\mathsf{plus}(+\mathsf{even}, +\mathsf{even}, -\mathsf{even})$$

Case: Rule pz.

$\mathsf{even}(\mathsf{z})$	mode $+\mathsf{even}$ of first arg.
$\mathsf{even}(N)$	mode $+\mathsf{even}$ of second arg.
$\mathsf{even}(N)$	previous line

Case: Rule ps.

$\mathsf{even}(\mathsf{s}(M))$	mode $+\mathsf{even}$ of first arg.
$\mathsf{odd}(M)$	by inversion from previous line
$\mathsf{even}(N)$	mode $+\mathsf{even}$ of second arg.

At this point we are stuck because we cannot apply the induction hypothesis, only knowing that M is odd.

So we need to generalize our declaration to

$$\begin{array}{ll} (i) & \mathsf{plus}(+\mathsf{even}, +\mathsf{even}, -\mathsf{even}) \\ (ii) & \mathsf{plus}(+\mathsf{odd}, +\mathsf{even}, -\mathsf{odd}) \end{array}$$

and restart our proof whose beginning is unchanged.

Case (i): Rule pz.

$\mathsf{even}(\mathsf{z})$	mode $+\mathsf{even}$ of first arg.
$\mathsf{even}(N)$	mode $+\mathsf{even}$ of second arg.
$\mathsf{even}(N)$	previous line

Case (i): Rule ps.

$\mathsf{even}(\mathsf{s}(M))$	mode $+\mathsf{even}$ of first arg.
$\mathsf{odd}(M)$	by inversion from previous line
$\mathsf{even}(N)$	mode $+\mathsf{even}$ of second arg.
$\mathsf{odd}(P)$	by i.h.(ii)
$\mathsf{even}(\mathsf{s}(P))$	by rule ev_s

Case (ii): Rule pz.

$\mathsf{odd}(\mathsf{z})$	mode $+\mathsf{odd}$ of first arg.
Contradiction	by inversion (no rule concluding $\mathsf{odd}(\mathsf{z})$)

Case (ii): Rule ps.

$\mathsf{odd}(\mathsf{s}(M))$	mode $+\mathsf{odd}$ of first arg.
$\mathsf{even}(M)$	by inversion from previous line
$\mathsf{even}(N)$	mode $+\mathsf{even}$ of second arg.
$\mathsf{even}(P)$	by i.h.(i)
$\mathsf{odd}(\mathsf{s}(P))$	by rule od_s

Note there that the case pz in the proof of (ii) is impossible: the rule pz cannot apply if the first argument of plus is odd. From the contradiction in this case we can infer anything, in particular that the third argument will be odd if the search succeeds, which it never will.

We see two differences here already to a system of types for functional languages such as ML: *types as predicates* have a natural notion of multiple related types (such as the mutually recursive even and odd numbers, as well as arbitrary natural numbers), and a given predicate such as plus may have multiple types, all of them necessary for type-checking purposes.

3 Subtyping

Types defined as predicates come with a natural notion of *subtyping*. For two predicates s and t we write $s \leq t$ if $\forall x.\, s(x) \supset t(x)$, that is, every element satisfying s also satisfies t.

To appreciate the need for subtyping, we consider once again binary numbers starting with least-significant bit and numbers in standard form (no leading zeros). We defined this slightly differently from last lecture by stipulating that in a term $\mathsf{b0}(N)$, N must be positive. This enforces that it cannot be e, which represents zero and is therefore not positive.

$$\frac{}{\mathsf{std}(\mathsf{e})}\ \mathsf{std}_e \qquad \frac{\mathsf{pos}(N)}{\mathsf{std}(\mathsf{b0}(N))}\ \mathsf{std}_0 \qquad \frac{\mathsf{std}(N)}{\mathsf{std}(\mathsf{b1}(N))}\ \mathsf{std}_1$$

$$\text{no rule } \mathsf{pos}_e \qquad \frac{\mathsf{pos}(N)}{\mathsf{pos}(\mathsf{b0}(N))}\ \mathsf{pos}_0 \qquad \frac{\mathsf{std}(N)}{\mathsf{pos}(\mathsf{b1}(N))}\ \mathsf{pos}_1$$

We now recall the increment predicate and try to verify that, if given a standard number it will construct a positive one.

$$\frac{}{\mathsf{inc}(\mathsf{e}, \mathsf{b1}(\mathsf{e}))}\ \mathsf{inc}_e \qquad \frac{}{\mathsf{inc}(\mathsf{b0}(M), \mathsf{b1}(M))}\ \mathsf{inc}_0 \qquad \frac{\mathsf{inc}(M, N)}{\mathsf{inc}(\mathsf{b1}(M), \mathsf{b0}(N))}\ \mathsf{inc}_1$$

$$\mathsf{inc}(+\mathsf{std}, -\mathsf{pos})$$

Case: Rule inc_e.

$\mathsf{pos}(\mathsf{b1}(\mathsf{e}))$	by rules pos_1 and std_e

Case: Rule inc_0.

$\mathsf{std}(\mathsf{b0}(M))$	first arg.
$\mathsf{pos}(M)$	by inversion
$\mathsf{std}(M)$	by $\mathsf{pos} \leq \mathsf{std}$, see below
$\mathsf{pos}(\mathsf{b1}(M))$	by rule pos_1

Case: Rule inc_1.

$\mathsf{std}(\mathsf{b1}(M))$	first arg.
$\mathsf{std}(M)$	by inversion
$\mathsf{pos}(N)$	by i.h.
$\mathsf{pos}(\mathsf{b0}(N))$	by rule pos_0

At this point the proof is complete, if we can show that $\mathsf{pos} \leq \mathsf{std}$. This is now a property that no longer requires appeal to the definition of predicate inc; it is just a property of the two types. We can proceed by induction (actually, just a proof by cases is required) on the definition of pos.

Case: Rule pos_0.

$\mathsf{pos}(N)$	premise
$\mathsf{std}(\mathsf{b0}(N))$	by rule std_0

Case: Rule pos_1.

$\mathsf{std}(N)$	premise
$\mathsf{std}(\mathsf{b0}(N))$	by rule std_1

Next we see how this kind of static type checking (phrased here as theorem proving) can help uncover errors. For example, we may want to check that

$$\mathsf{inc}(-\mathsf{std}, +\mathsf{std})$$

Case: Rule inc_e. Then $\mathsf{std}(\mathsf{e})$.

Case: Rule inc_0.

$\mathsf{std}(\mathsf{b1}(N))$	second arg. of inc
$\mathsf{std}(N)$	by inversion
Need: $\mathsf{pos}(N)$	*not true in general!*
$\mathsf{std}(\mathsf{b0}(N))$	by rule std_0

There is no way to fix the missing step in the second case (we didn't even get around to the third case). $\mathsf{std}(N)$ does not imply $\mathsf{pos}(N)$, with $N = \mathsf{e}$ as a counterexample. Indeed, one solution for

$$\text{?- } \mathsf{inc}(M, \mathsf{b1}(\mathsf{e}))$$

is $M = \mathsf{b0}(\mathsf{e})$ which is *not* in standardard form.

At this point we might consider some other properties. Let's define some new types, such as $\mathsf{zero}(N)$, and $\mathsf{empty}(N)$ which never succeeds:

$$\frac{}{\mathsf{zero}(\mathsf{e})}\ \mathsf{zero}_e \qquad\qquad \text{no rule for } \mathsf{empty}(N)$$

Now we can show, with type checking that a query $\mathsf{inc}(M, \mathsf{e})$ cannot succeed. The type we ascribe is

$$\mathsf{inc}(-\mathsf{empty}, +\mathsf{zero})$$

which expresses that if a query $\mathsf{inc}(M, n)$ with $\mathsf{zero}(n)$ succeeds with $M = m$, then $\mathsf{empty}(m)$. Since there is no such m, this means *if* inc *has the given type* then decrementing zero can not succeed. This means it either doesn't terminate or it fails after a finite number of steps.

Now to the type checking:

Case: Rule inc_e.

$\mathsf{zero}(\mathsf{b1}(\mathsf{e}))$	second argument
Contradiction	by inversion (no rule concludes $\mathsf{zero}(\mathsf{b1}(\mathsf{e}))$)

Case: Rule inc_0.

$\mathsf{zero}(\mathsf{b1}(M))$	second argument
Contradiction	by inversion

Case: Rule inc_1.

$\mathsf{zero}(\mathsf{b0}(M))$	second argument
Contradiction	by inversion

All cases are impossible, so the type $\mathsf{inc}(-\mathsf{empty}, +\mathsf{zero})$ is correct.

As a last example we revisited the even/odd distinction, now on binary numbers. We could just look at the least significant bit, but we arrange it such that $\mathsf{even} \leq \mathsf{std}$ and $\mathsf{odd} \leq \mathsf{pos}$ to ease working with these types.

$$\frac{\mathsf{pos}(N)}{\mathsf{even}(\mathsf{b0}(N))}\ \mathsf{ev}_0 \qquad \frac{\mathsf{std}(N)}{\mathsf{odd}(\mathsf{b1}(N))}\ \mathsf{od}_1$$

We leave it to the reader to now verify that

$$\mathsf{inc}(+\mathsf{even}, -\mathsf{odd})$$
$$\mathsf{inc}(+\mathsf{odd}, -\mathsf{even})$$

4 Refinement types for functional languages

The idea that we have more precise types than just nat (like even and odd) or binary numbers (like std, pos, zero, empty) could be a priori useful for functional languages as well, not just for logic programming.

The main complication is that we also need to include *intersection types* [CDCV81, Rey91] to make this work. We retain the usual data types, but we add *data sort* declarations that declare *refinements* [FP91]. The examples in this section and many more can be found in a conservative extension of Standard ML with datasort refinements called Cidre [Dav97], available on GitHub[2].

For example, we can define `even` and `odd` unary numbers as follows:

```
datatype nat = z | s of nat

datasort even = z | s of odd
     and odd = s of even
```

But now if we have a simple function such as

```
fun succ x = s(x)
```

we find that it has multiple function types at once:

```
succ : (nat -> nat) & (even -> odd) & (odd -> even)
```

So we need to be able to ascribe multiple types to a function or expression. This is what the intersection type operator $A \sqcap B$ achieves, which we write as `A & B` in concrete syntax. Sometimes, several types are needed. For example

```
fun twice f x = f (f x)
```

then we should be able to show (among many other types)

```
twice : ((nat -> nat) -> nat -> nat)
      & (((even -> odd) & (odd -> even)) -> (even -> even))
      & (((even -> odd) & (odd -> even)) -> (odd -> odd))
```

The resulting system has some remarkable properties, such as decidability of type inference, bidirectional type checking, and conservative extension over ML. It is implemented in the Cidre front end, which accepts

[2] https://github.com/rowandavies/sml-cidre

the full syntax of Standard ML and uses stylized comments to assign refinement types that are then checked.

You probably have seen one example where this might have been helpful. For example, for propositions with implication and conjunction, one can define proof terms via the following data type

```
datatype term = Fun of var * term
              | Pair of term * term
              | Var of var
              | App of term * term
              | Fst of term
              | Snd of term
```

In a way, this was a compromise, since we distinguished, in the problem statement and the algorithm, between checkable and synthesizing terms. The corresponding data type declaration would be something like

```
datatype cterm = Fun of var * cterm
               | Pair of cterm * cterm
               | Syn of sterm
     and sterm = Var of var
               | App of sterm * cterm
               | Fst of sterm
               | Snd of sterm
```

but there are two drawbacks: (1) we need to make the transition from synthesizing to checkable terms explicit (see `Syn` constructor), which complicates practical examples, and (2) now *everywhere* that terms are used, even in places where the distinction would be insignificant, we have to be cognizant and specific about whether we are working with checkable or synthesizing terms. With refinement types, we would first declare the type of terms, and then think of checkable and synthesizing terms as refinements.

```
datasort cterm = Fun of var * cterm
               | Pair of cterm * cterm
               | sterm
     and sterm = Var of var
               | App of sterm * cterm
               | Fst of sterm
               | Snd of sterm
```

We can now freely use either `term` (where it doesn't matter) or `cterm` or `sterm` where the distinction is significant.

References

[CDCV81] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

[Dav97] Rowan Davies. A practical refinement-type checker for Standard ML. In Michael Johnson, editor, *Algebraic Methodology and Software Technology Sixth International Conference (AMAST'97)*, pages 565–566, Sydney, Australia, December 1997. Springer-Verlag LNCS 1349.

[DP03] Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In A.D. Gordon, editor, *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, pages 250–266, Warsaw, Poland, April 2003. Springer-Verlag LNCS 2620.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[Pfe92] Frank Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.