Using a dependent type checker as a theorem prover

Randall Holmes

October 24, 2017

We will explain what a dependent type checker is and explain why it can be used as a theorem prover, with examples of interaction with actual software.

Simple type theory

We describe the simple typed theory of functions of Church. There is an underlying sort ι of individuals. For each pair of sorts α, β, there is a sort $\alpha \times \beta$ inhabited by pairs (a, b) where a is of sort α and b is of sort β, and a sort $\alpha \to \beta$ inhabited by functions from type α to type β: if x is of type α and f is of type $\alpha \to \beta$, then $f(x)$ is of type β [the formal similarity to the logical rule of *modus ponens* should not be overlooked]. Other types which might be present are the type 2 of booleans (truth values) and the type ν of natural numbers.

Types should not be thought of as sets, although an interpretation of a type theory in an untyped framework such as the usual set theory ZFC will associate each type with the set of its inhabitants. A better analogy is with grammatical parts of speech. It can be verified reasonably readily that if each variable is assigned a type, then the well-formedness and the type of any composite expression built using function application and the ordered pair can be determined entirely on syntactical grounds.

Another construction should be noticed: if x is a variable of type α and $T[x]$ is a term of type β (the bracket $[x]$ just indicating that this is an expression which might contain the variable x), $(\lambda x : T[x])$ is a term of type $\alpha \to \beta$, representing the function whose definition might be given in the form $f(x) = T[x]$.

Notice that in this theory the notion of function is primitive: we do not consider a function to be a set of ordered pairs (though it might be implemented in this way in a model of the theory in set theory).

Notice also that the notation encourages the traditional view of a function as a rule.

In fact, the natural implementation of a set of objects of type α is as the type of functions $\alpha \to 2$, though we could introduce a primitive construction $\mathcal{P}(\alpha)$ for power types.

The simple type theory here described is a fairly good medium for defining standard kinds of mathematical object. Its syntactical constraints have made it unpopular for doing mathematics, though it is quite popular in theoretical computer science.

Dependent type theory

Dependent type theory admits the same basic constructions as those of the simple type theory just described, namely cartesian products and function spaces. The additional feature is that the types of projections of pairs or of arguments or values of functions may depend on earlier projections (in the case of pairs) or earlier arguments of the same function (in the case of argument of a function) or on arguments of the function (in the case of the value of a function).

A perhaps silly example of a dependently typed function is the function $f(n, a)$ (n being a natural number, a being of type α) which returns the tuple $(a, \ldots, a) \in \alpha^n$ ($(a_1, \ldots, a_n)$ abbreviating $(a_1, (a_2, \ldots, a_n))$ α^n abbreviating $\alpha \times \alpha^{n-1}$, where $n > 2$ and α^1 simply means α.

Formal notation: if $T[x]$ is an expression of sort $F[x]$ possibly depending on x, then $(\lambda x : T)$ is a function of the dependent type $(\Lambda x : F[x])$, again exactly the function f we would define by the rule $f(x) = T[x]$.

In the example above, the function f has type $(\Lambda((n, a) : \alpha^n)$ (if n is understood to be a natural number variable and a to be a variable of type α).

There are similar dependent product constructions, building ordered pairs or longer tuples in which the type of later projections can depend on the type of earlier ones: an example might be a type inhabited by pairs $(n, (a_1, \ldots, a_n))$ whose first component is a natural number and whose second component is an inhabitant of the type α^n. We will not explore notation for these as we will only make use of dependent function types.

There are mathematical constructions (and computer science constructions) for which dependent types afford a natural implementation.

The specific application we will emphasize here is the implementation of mathematical *proofs* rather than of mathematical objects in the usual sense.

We have already hinted at a relationship between function types and proofs of implications which we now make specific. If we have a type $\texttt{proofs}(P)$ of proofs of [or evidence for] a proposition P and a type $\texttt{proofs}(Q)$ of proofs of Q, then any element of the function space $\texttt{proofs}(P) \rightarrow \texttt{proofs}(Q)$ witnesses the fact that $P \rightarrow Q$.

The device of dependent types allows us to represent proofs of universally quantified statements by functions in a similar way.

Suppose that $P(x)$ is of type 2 (a truth value) for each x of type α (the function P represents a predicate of type α objects). Then a function of the type $(\Lambda x : \texttt{proofs}(P(x)))$ is a function which takes each x in type α to a proof of $P(x)$: this function witnesses the truth of $(\forall x : P(x))$ (the quantifier is restricted to type α, but this is indicated by the known type of the variable x).

This correspondence between type constructions and constructions of propositions is called the Curry-Howard isomorphism. Curry and Howard discovered it independently; a third independent discoverer of the same phenomenon was de Bruijn, the developer of the first significant computer assisted reasoning project, known as Automath, a Dutch project of the 1970's.

We give some very simple examples of basic logical proofs which can be constructed in this kind of framework, in terms of our abstract notation (as we will see, there are some differences between the formalism given above and the official formalism of our implementation). Consider the function $(\lambda x : x)$, where x is a variable of type $\texttt{proofs}(P)$. This is a function taking a proof of P to a proof of P, and witnesses the truth of $P \rightarrow P$.

We note that an element of the type $\mathtt{proofs}(P) \times \mathtt{proofs}(Q)$ can be taken to witness the truth of $P \wedge Q$, since it contains a proof of P and a proof of Q as components. Now suppose that we have x a variable of type α as usual and a variable u of type

$$(\Lambda x : (\mathtt{proofs}(P(x)) \rightarrow \mathtt{proofs}(Q(x)))) \times (\Lambda x : (\mathtt{proofs}(Q(x)) \rightarrow \mathtt{proofs}(R(x))$$

You can track down the fact that an object of this type witnesses a proof of

$$(\forall x : P(x) \rightarrow Q(x)) \wedge (\forall x : Q(x) \rightarrow R(x)).$$

Further let p be a variable of type $\texttt{proofs}(P)$.

Now consider the function

$$(\lambda u : (\lambda x : (\lambda p : (\pi_2(u)(x)(\pi_1(u)(x)(p)))))).$$

We have to unwrap this to see if it makes sense. $\pi_1(u)(x)(p)$ is a proof of $Q(x)$.

$\pi_2(u)(x)(\pi_1(u)(x)(p))$ is a proof of $R(x)$.

$(\lambda p : (\pi_2(u)(\pi_1(u)(p))))$ is a proof of $P(x) \to R(x)$.

$$(\lambda x : (\lambda p : (\pi_2(u)(x)(\pi_1(u)(x)(p)))))$$

is a proof of $(\forall x : P(x) \to R(x))$

And finally

$$(\lambda u : (\lambda x : (\lambda p : (\pi_2(u)(x)(\pi_1(u)(x)(p))))))$$

witnesses a proof of

$$((\forall x : P(x) \to Q(x)) \wedge (\forall x : Q(x) \to R(x))) \to (\forall x : P(x) \to R(x)).$$

Notice what we just did. We carried out a basically grammatical process of type checking a term of the language of our dependent type theory and discovered a theorem of logic.

Lestrade introduced

We begin to introduce our computer implementation, Lestrade, which may be viewed as a dialect of the ancient system Automath.

Lestrade draws a distinction between objects and functions. It has object sorts and dependent function sorts.

The object sorts can be described briefly.

There is a sort `prop` inhabited by propositions (things we can prove). For each object p of the sort `prop`, there is a sort `that` p inhabited by evidence for p (or proofs that p).

There is a sort `type` inhabited by objects which we will call "type labels". For each object τ of sort `type`, there is a sort `in` τ inhabited by objects of the type labelled by τ. Typical objects of these sorts might be `Nat` (the type of natural numbers) of sort `type` and 2 (for example) of sort `in Nat`.

There is an additional sort `obj` intended to be inhabited by "untyped" mathematical objects (the sets of an implementation of the usual set theory might be taken to be of this sort; one should notice though that a Lestrade implementation of ZFC would not actually be untyped, as many complex types of evidence for propositions would be present).

Note that while we state different intentions for the constructors `prop/that` and `type/in`, they are actually treated in exactly the same way by Lestrade.

The functions of Lestrade can also be described quite briefly, but the ramifications of the brief description are not immediately obvious.

When a sequence of variable arguments $x_1, \ldots, x_n$ (each of which may be an object or a function) of types $\tau_1, \ldots, \tau_n$ have been declared – in the order indicated, where a sort τ_i can depend on an argument x_j only if $j < i$, and each τ_i depends on no other variables – and a further object sort τ is given, which may depend on no variables other than the x_i's – then a function may be postulated or defined which sends arguments x_i of the indicated types τ_i to output of type τ.

To express this exactly requires some subtlety, The type of this function (we give the function the nonce name f) is written

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow \tau).$$

A term $f(t_1, \ldots, t_n)$ is well-sorted only if t_1 is of sort τ_1 (a necessary but not sufficient condition). If $n = 1$ the term $f(t_1)$ has sort $\tau[t_1/x_1]$ (the result of replacing x_1 with t_1 in τ). Otherwise $f(t_1, \ldots, t_n)$ is well sorted iff $f_2(x_2, \ldots, x_n)$ is well-sorted, and if it is well-sorted has the same sort, where f_2 is a function of sort

$$((x_2 : \tau_2[t_1/x_1]), \ldots, (x_n : \tau_n[t_1/x_1) \Rightarrow \tau[t_1/x_1]).$$

For completeness we need to note that substitution into a function sort

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow \tau)$$

will be carried out after replacement of all the x_i's with fresh variables x_i' not found elsewhere in the context.

If $x_1, \ldots, x_n$ are variables declared earlier and in that order, and T is an object term depending on no variables but the x_i's with type τ depending on no variables but the x_i's then a function f can be defined denoted by

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (T : \tau))$$

of type

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow \tau)$$

$f(t_1, \ldots, t_n)$ can be computed as one would expect (mod the complexities of dependent typing), if it is well-typed. If $n = 1$, then $f(t_1) = T[t_1/x_1]$. Otherwise $f(t_1, \ldots, t_n) = f_2(t_2, \ldots, t_n)$, where the definition of f_2 is $((x_2 : \tau_2[t_1/x_1]), \ldots, (x_n : \tau_n[t_1/x_1) \Rightarrow (T[t_1/x_1], \tau[t_1/x_1]))$.

Where $f(t_1, \ldots, t_n)$ is well-typed and represents an object (recall that output sorts are always object sorts) the notation $f(t_1, \ldots, t_i)$ for $i < n$ represents the function $f_{t_1,\ldots,t_i}$ with definition

$$f_{t_1,\ldots,t_i}(x_{i+1}, \ldots, x_n) = f(t_1, \ldots, t_i, x_{i+1}, \ldots, x_n):$$

this is a variation of the technique of "currying". In Lestrade notation, such truncated terms occur only as arguments, and never in applied position.

Declaration management

A Lestrade session involves the creation and management of primitive and defined atomic terms (single identifiers).

Atomic terms are organized into *moves*. At any given time there are moves $0, 1, \ldots, i, i+1$ (moves 0 and 1 are always present). Move i is called the last move. Move $i+1$ is called the next move.

The semantics of this is that primitive terms of move $i+1$ are currently regarded as variables (and defined terms of move $i+1$ will be complex variable terms). Primitive and defined atomic terms of lower indexed moves are currently viewed as constant. Objects and function declared or defined at move 0 are the primitive object and function constants of the theory (unconditionally constant).

The parameter i may be changed: a new move may be opened, in which case i is incremented and a new move $i+1$ is introduced, empty of declarations. The next move may be closed if $i > 1$, in which case i is decremented and the former move $i+1$ is deleted with all its declarations. Either of these moves changes which primitive terms are currently viewed as possible parameters for functions to be postulated or defined.

This is not as exotic an idea as it seems. In familiar expressions such as $f(x, y) = ax + by$, we *say* that a and b are constants... but of course they are variables. They are at an earlier move in the sense of Lestrade's declaration system, and this exact example is something I have implemented somewhere.

Basic actions available are

1. to declare a variable of any object sort at move $i+1$. The sort may depend on variables already declared at move $i+1$. The order in which variables are declared is remembered by Lestrade.

2. to declare a variable of an object sort at move i (which amounts to postulating a constant). This sort may depend on defined notions at move $i+1$, but in the computation of its sort these definitions must be expanded out, and cannot ultimately depend on any variables of move $i+1$.

3. to declare a primitive atomic function at move i (not $i+1$): the function will be

declared as a function of a list of move $i+1$ primitive variables (not defined terms) appearing in the order in which they were declared, with a stated object sort for its output: the sort of the function will be computed and displayed as above.

4. to define an atomic object term at move i: the new atomic term is equated with an object term, which may depend on defined terms declared at move $i+1$ (which will be expanded out) but not on variables at move $i+1$.

5. to define an atomic function term at move i: this is declared in the familiar form

$$f(x_1, \ldots, x_n) = T,$$

and succeeds if this can be typed successfully as indicated above.

Whenever an object is declared at move i, all defined notions from move $i+1$ are expanded out. A defined function in applied position is eliminated by expanding its definition; a defined function appearing as an argument is displayed in sort information as a term of the form given above (basically as a λ-term). There is a reason for this: move $i+1$ might be closed at any time, so none of its declared identifiers should appear in moves of lower index. The *variables* at move $i+1$ become bound variables with no associated declarations in sort information at move i (and are assigned fresh names distinct from those of the original parameters in the declaration).

Lestrade has a capability not formally described above. Not all arguments of a function need to be shown: a additional move $i+1$ variable appearing in the sort of one of the explicit arguments or in the output sort will be supplied as an implicit argument. Under many but not all conditions, the parser can deduce the values of implicit arguments from the sorts of the explicitly given arguments. This is complex, and does not actually affect the type system: it is an optimization of input/output. Examples will be seen.

Note that the user never types a λ-term or function sort: the user notation is strictly applicative (the only operation is application of functions to argument lists; the user never writes anything but applicative terms and object sorts).

The ability to write truncated object terms allows one to in effect write λ-terms depending on variables in some contexts; the implicit argument feature allows one to completely avoid entering function arguments in contexts where they can be deduced from the sorts of other arguments.

Sort information displayed by Lestrade is slightly different than the abstract notation given above. The sort displayed for a defined function is the same as the abstract representation of the function

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (T : \tau));$$

a primitive function of the same type would have sort information

$$((x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (- : \tau)).$$

A defined object would have a sort of the form (T, τ), identical to τ for purposes of type checking.

Note that a function variable of move $i+1$ can exist, but must have been declared when the move which we would currently call $i+2$ was open. Introduction of function variables will be demonstrated in the examples.

When the close command is issued, move $i+1$ just vanishes. It is possible under suitable circumstances to save a version of move $i+1$ with a name before closing it. The declarations in the saved move $i+1$ can be recovered at any time that one returns to the same version of move i (which may contain further declarations made in the interim). In this way the structure of moves may actually be a tree.