0. What is Tense Logic Programming?

Tense Logic Programming is a new form of logic programming in which the underlying logic is a tense logic - a kind of modal or intensional logic in which the values of expressions and formulas depend on an implicit or hidden time parameter.

In this paper we will discuss a particularly simple form of Tense Logic Programming. The underlying tense logic is one in which the collection of 'moments in time' is the natural numbers, and the only temporal operators are first and next (applied to formulas, not terms). The statements in our programs are universally quantified Horn clauses, as in Prolog. In fact our programs look exactly like Prolog programs, except that first and next can be applied to formulas in the Horn clauses.

Here is a simple example of THLP (Tensed Horn Logic Programming):

\[
\begin{align*}
\text{first} & \text{ count}(0, 0) \leftarrow \\
\text{next} & \text{ count}(s(X), Y) \leftarrow \text{count}(X, Y), \text{light(red)} \\
\text{next} & \text{ count}(X, s(Y)) \leftarrow \text{count}(X, Y), \text{light(green)}
\end{align*}
\]

The program will be explained in a later section, although its meaning can probably be guessed. We are using a cprolog-like syntax, in which uppercase letters are (universally quantified) variables.

1. What are the advantages of using Tense Logic?

Tense Logic allows us to use the prolog style of programming to specify iterative algorithms and continuously operating (nonterminating) computations. It allows us to capture the dynamic aspects of certain problems (such as simulation) in a natural and problem-oriented style. It makes possible a dataflow style of programming and a correspondingly more modular approach to certain applications, one with significantly more potential for concurrency. Finally, it does so without the use of nonlogical primitives like cut or assert and retract.
The essential difference between tense logic and ordinary logic is that in tense logic a formula is not simply true or false. In general, a tense logic formula will be true at some moments and false at others. A formula can therefore be thought of as a time-varying truth value, or as a stream of truth values, generated in chronological order. The notions of time and change are therefore incorporated in the logic itself, not brought in the back door by nonlogical primitives.

In ordinary HLP (Horn Logic Programming - Prolog) a unary predicate \( P \) can be thought of as representing a set of ground terms - namely those ground terms for which the goal \( P(t) \) will succeed. If there is only one such ground term, we can think of \( P \) as representing a single individual. (This is in the context of a program which 'axiomatises' \( P \)).

For example, we might have a predicate \texttt{president} for which all the goals \texttt{president(nixon), president(ford), president(carter), ...} all succeed. In this case \texttt{president} represents the set of all presidents. On the other hand, it might be that only the goal \texttt{president(reagan)} succeeds. In that case, the predicate \texttt{president} is representing only the current president.

In THLP, the truth value of a formula may vary from one 'moment' to the next. Thus the goal \texttt{president(carter)} may succeed for certain times and fail for others. In THLP, we can therefore accurately model the fact that presidency is a time-dependent property.

2. How does a THLP programmer specify dynamic predicates?

He or she simply gives a Horn clause axiomatisation of the way predicates vary in time.

Let us begin with an extremely simple example: a counter. We want the unary predicate \texttt{i} to represent the stream 0, \texttt{s(0)}, \texttt{s(s(0))}, ... of natural numbers in successor notation. The first element in the stream is 0, and thereafter the next element of the stream is the result of applying \texttt{s} to the current element. This description can be formalized by the following program:

\[
\begin{align*}
\text{first } & \texttt{i}(0) \leftarrow \\
\text{next } & \texttt{i}(\texttt{s}(X)) \leftarrow \texttt{i}(X)
\end{align*}
\]

The key to understanding this (and all other) THLP programs is to realize that the statements (Horn clauses) are true at all points in time. The second clause can therefore be paraphrased as

\[
\text{if at any time } t \text{ \( i \) is true of something of the form } X, \text{ then at time } t +1 \text{ \( i \) is true of } s(X).
\]

If these are the only axioms about \( i \), it is easy to see that at time 0 only \texttt{i}(0) will succeed, at time 1 only \texttt{i}(\texttt{s}(0)) will succeed, at time 2 only \texttt{i}(\texttt{s}(\texttt{s}(0))) will succeed, and so on.
Along the same lines, we can axiomatise the "stream" of fibonacci numbers with

\[
\begin{align*}
\text{first } & \text{fib}(0). \\
\text{first } & \text{next } \text{fib}(1). \\
\text{next next } & \text{fib}(K) \leftarrow \text{next fib}(M), \text{fib}(N), K = M+N.
\end{align*}
\]

If these are the only axioms for \text{fib}, it is easy to see that at time \( t \) \text{fib} succeeds only for the ground term which denotes the \( t \)th fibonacci number.

In the same way, it should be clear that the program given in section 0 counts the number of times the 'light' has been red or green. Suppose, for example, that the five goals

\[
\text{light(red)}, \text{light(green)}, \text{light(red)}, \text{light(red)}, \text{light(green)}
\]

succeeded at the first five respective time points. Then at time 5 the goal \( \text{count}(s(s(s(0))), s(s(0))) \) will succeed.

Finally, our axiomatisation of the time varying \text{president} predicate discussed earlier might include clauses like

\[
\begin{align*}
\text{first } & \text{president(washington)} \leftarrow \\
\text{first } & \text{year}(1783) \leftarrow \\
& \ldots \\
\text{next } & \text{president}(p) \leftarrow \text{president}(p), \\
& \text{electionheld}(\text{no}), \\
& \text{resignation}(\text{no}), \\
& \text{impeachment}(\text{no}) \\
\ldots \\
\text{next } & \text{president}(q) \leftarrow \text{electionheld}(\text{yes}), \text{electionwinner}(q) \\
\ldots \\
\text{election}(\text{yes}) & \leftarrow \text{year}(1984) \\
\ldots
\end{align*}
\]

Notice that boolean streams are represented by unary predicates which always succeed, but with \text{yes} or \text{no} as appropriate. THLP, like HLP, is based on a monotonic logic without negation.

3. Can't this all be done in ordinary Prolog?

Not easily. We would have to add an extra explicit time parameter to every predicate, then simulate the effect of \text{first} and \text{next} by explicitly manipulating the these time parameters. For example, the count program would become

\[
\begin{align*}
\text{count}(\text{start}, 0, 0) \leftarrow \\
\text{count}(\text{next}(T), s(X), Y) & \leftarrow \text{count}(T, X, Y), \text{light}(T, \text{red}) \\
\text{count}(\text{next}(T), X, s(Y)) & \leftarrow \text{count}(T, X, Y), \text{light}(T, \text{green})
\end{align*}
\]
This technique for reducing THLP to HLP is formally correct (and could form the basis of a quick and dirty implementation) but it is not really practical. There are two fundamental problems:

Firstly, the added explicit time parameters make the programs harder to read and write, and introduce extra opportunities for errors. People use temporal logic to reason about programs, even though temporal logic can be reduced to ordinary predicate logic with a similar technique. Why not use temporal logic itself to write the programs in the first place?

Secondly, the reduced program would be run on a general purpose HLP interpreter, which would not take into account the special nature of the time parameters. In particular, it would make no attempt to record time-tagged successes and failures - something apparently required by an efficient implementation of THLP (see below).

4. How are the semantics of THLP specified?

Exactly the same way the semantics of ordinary HLP is specified. Or rather, exactly the same ways, because there are at least three very different (though equivalent) ways of giving a meaning to ordinary HLP.

Let us avoid input/output for the moment and formulate the semantics problem as follows: given a program \( P \) and a term \( T \), which ground instances of \( T \) should an implementation produce when given \( T \) as a ‘query’? Here are three ways of answering the question for HLP:

(a) those instances which can be proved from the statements of the program using the Horn logic rules of inference (substitution and modus ponens);

(b) those instances which are logical consequences of the program, that is, those which are true in any structure in which all the statements of the program are true;

(c) those instances which are true in the minimal Herbrand (term) model of the program.

Each of these methods can be adapted to THLP by making appropriate alterations to the notions of “rule of inference”, “structure”, and “model” (details will be given in the full paper). The adapted methods are still all equivalent.

Take, for example, the simple counting program. We stated that at time 2 the goal \( 1(s(s(0))) \) succeeded. We can formalize this as follows: given the count program and the query term \( \text{first} \ \text{next} \ \text{next} \ 1(y) \), the implementation responds with the ground term \( \text{first} \ \text{next} \ \text{next} \ 1(s(s(0))) \). It does so because the latter instance is the only one which can be derived from the program, is true in all models of the program, or is true in the minimal Herbrand model.
5. Are Tense Logic and Horn Clauses enough?
For many applications, yes; but in general, no.
There are two serious limitations of THLP that soon become apparent. There
seems to be no way of getting around these limitations and remaining in
first-order tense logic using only Horn clauses. Fortunately, these limitations can be
overcome by using a slightly more powerful logic - but without using nonlogi-
cal concepts.
The first problem is that pure THLP does not allow the programmer to make
arbitrary but definite choices. The problem first presented itself in connection
with input/output.
One of the advantages of THLP is that it provides a simple solution to the
problem of stream oriented I/O. We designate two unary predicates, input
and output, as playing a special role in the language. The predicate input is
defined by the implementation, and should not appear on the left-hand side of
a program statement. At any time \( t \) there is only one ground instance of
input(\( \bar{x} \)) which succeeds - namely that in which \( \bar{x} \) is instantiated to the \( t^{th} \)
input item - the input to the program being a stream of ground terms.
The output predicate, on the other hand, is axiomatized by the programmer.
In the simplest case, the program should ensure that at any time \( t \) there is
exactly one ground instance of output(\( \bar{x} \)) which succeeds. The unique ground
term which \( \bar{x} \) instantiates to is produced, by the implementation, as the \( t^{th} \)
element of the output stream.
The problems arise when the output predicate does not specify a unique
ground term. And why should it? - perhaps the problem specification itself
does not constrain the programmer to producing one particular output
stream. In that case it would be completely against the spirit of logic pro-
gramming to force the programmer to complicate the program by making it
unnecessarily specific. Clearly, the implementation should choose one of the
possible instantiations as the output.
This is fine - except that the programmer may need to know what was actu-
ally output. For example, suppose that we are required to produce an increas-
ing stream of natural numbers starting at 0. The program

\[
\text{first output}(0) \leftarrow \\
\text{next output}(y) \leftarrow \text{output}(x), \text{less than}(x, y)
\]

appears to work but doesn't. What it really says is that if \( x \) might be the
(say) 10th output, and \( x \) is less than \( y \), then \( y \) might be the 11th output.
What the program really specifies is any arbitrary stream whose time \( t \) com-
ponent is not less than \( t \).
What we need to say is that the time 11 possible outputs are those numbers
which are greater than the time 10 actual output. This is not possible in pure
THLP.
The solution is to provide an extra predicate `!output` which succeeds for only
the ground term that was actually produced. The program should be

```
first output(0) <-
next output(Y) <- !output(X), lessthan(X, Y)
```

Of course the problem is more than just one of I/O. In a dataflow-structured
program some parts are producers and others are consumers; thus every predi-
cate is potentially an 'output stream'. This means that we must supply choice
variables for every predicate symbol - though of course the programmer is
under no obligation to use all or any of them.

The relation between output and !output can be axiomatised, but not in
Horn Logic - we require predicate calculus with equality. The choice variables
therefore represent a very controlled extension of our language beyond Horn
Logic. This is reflected in the changes to the semantic definition. Answers to
queries are no longer logical consequences of the program (because they
depend on arbitrary choices). Essentially (we omit the details) the implementa-
tion is computing a minimal model of the program plus the implicit non-
Horn axioms for the choice variables.

The second problem which arises concerns the interpretation of predicates as
streams. In first order Horn Logic we cannot define 'filters' whose output rate
is different from the input rate. To define such filters we must add higher-
order relations. The higher order logic is fortunately relatively simple, but we
must omit the details.

6. How is THLP implemented?

A very good question. At present there is no really serious implementation at
all.

We saw earlier that we could use explicit time parameters to reduce THLP to
ordinary HLP, and then use an ordinary HLP interpreter. This could provide
a prototype implementation but for reasons mentioned earlier it can never be
practical.

The simplest course is to adapt an existing HLP interpreter. The time param-
eters could be represented as 'tags' on formulas, not as extra arguments to
predicates. Then the matching algorithms could be adapted to take the tags
into account. For example, the goal `foobart(g(X, 3), s(p(m,N))) .10` (10 is the
time tag) matches the rule

```
next foobar(X, s(Y)) <- baz(Y)
```

and generates the subgoal `baz(p(m,N)) .9`.

An efficient THLP interpreter almost certainly requires a "warehouse" or
"variable-value" store to record successes or failures (with associated time
tags). Otherwise it would waste enormous resources recomputing results over
and over again. A THLP implementation must therefore combine features of
an HLP implementation (unification, environments) with features of a
dataflow implementation (tagging, associative memory).
The ‘choice’ variables should not complicate the implementation too much.
We need only keep a record of choices made, as they are made.

7. What about Concurrent Prolog, Parlog, and the like?
Some of the programming methods discussed in this paper can be realized in
Concurrent Prolog and in other extensions of basic Prolog. These programs
would not have explicit time parameters and they might offer a certain degree
of dataflow modularity and concurrency.

Unfortunately these ‘super-Prologs’ all have a fatal flaw: they are not logic.
Not only do they retain cut and other nasty features of Prolog; they add even
more bizarre new ones, usually in the form of annotations for predicate argu-
ments. Furthermore, even the simplest programs require nonlogical primitives,
and can only be understood in terms of a complex nondeterministic and
asynchronous operational model.

These super-prologs are all based on an idea which, we feel, is inherently
flawed. They all extend HLP by allowing infinite terms.

At first sight this seems only natural: after all, the new generation of func-
tional languages (such as KRC) successfully extended LISP-like languages by
allowing infinite lists. These infinite lists can in turn be used to represent data
streams - all without violating basic principles such as referential trans-
parency.

Unfortunately this technique does not work on a nondeterministic language
like HLP. One cannot construct infinite objects in the presence of backtracking:
even if the computation proceeds lazily (incrementally), one can never be
sure of any of the partial results. The programmer must force a process to
commit itself to releasing output which in principle is unreliable. Infinitary
versions of Prolog are therefore inherently illogical - cut (in some form) is
needed in even the most simple examples. And the programmer therefore
loses any a priori confidence that the statements of the program are true of its
output.

At a recent international conference a prominent proponent of one brand of
infinitary prolog was asked about the heavy use of non-logical primitives. “So
what?” was the reply! We cannot understand the willingness (in fact, eager-
ness) to abandon the basic principles of logic.

In THLP there are no infinite data objects and no need for cut. A THLP
implementation performs a (potentially) endless series of finite unifications,
instead of one single endless unification. THLP uses stream of predicates,
rather than predicates on streams. THLP offers a respectable alternative.