

The Intensional Logic Language InTense

W.H.Mitchell and A.A.Faustini

Declarative Languages and Advanced Architecture Group

Department of Computer Science

Arizona State University

Tempe, Arizona 85287

Abstract

A new programming language based on intensional Horn Clause logic is introduced. The language **InTense** allows formulas in clauses to vary with time, in common with Wadge's language Chronolog, but in addition, also allows formulas to vary in space. Actually, an Intense program can, under the current implementation, make use of up to four time and four space dimensions. The resulting language is the spiritual heir to the functional language Field Lucid, yet is suited to problem solving in slightly different domains due to its nondeterministic control strategy and powerful pattern matching.

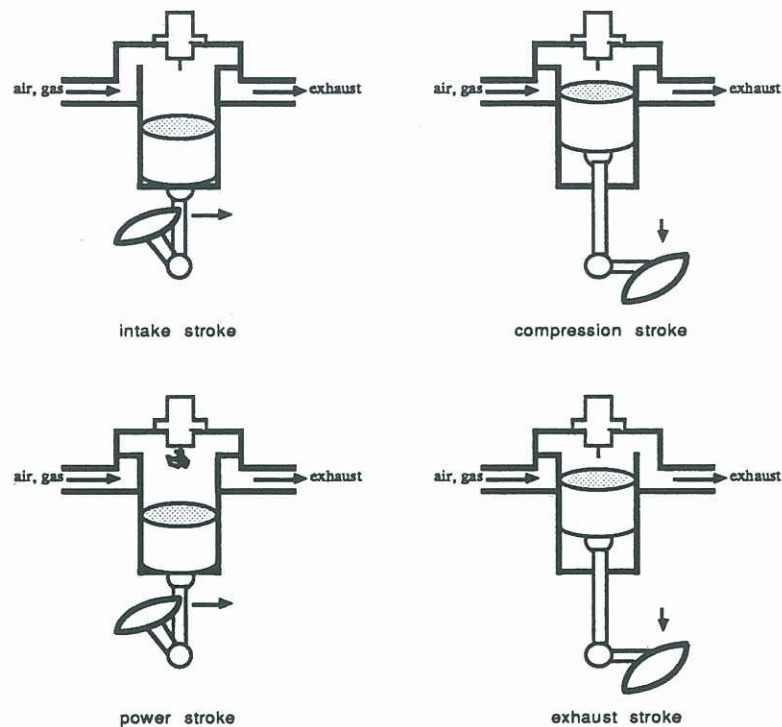
1 Introduction

Horn Clause logic programming in the form of Prolog has proven its worth in applications spanning a wide variety of problem domains [see CoCo80], from AI applications like natural language processing and expert systems to systems programming tasks such as compilation. Prolog's success as a programming language seems to stem from many factors, not the least of which is the fact that since logic was initially developed as a vehicle for formalizing human reasoning, Prolog programs tend to be natural, concise, and easily understandable.

Useful and popular though it may be, Prolog is far from perfect, as evidenced by the hoard of Prolog mutants (ie Parlog [Gr87] and Concurrent Prolog [Sh83]) presently vying for popularity. Although its detractors have mostly picked on other aspects of the language, our major criticism of Prolog (and logic programming in general) is that it is not adept at modeling and manipulating anything but **static** objects. While many of the

entities and relationships that we might like to represent *are* indeed static (for instance, the set of integers), the majority of our world, the objects, relations and events that we would like to be able to model via a logic program, are **dynamic**.

Consider the case in which we would like to model in a somewhat simplified manner, the operation of a four-stroke piston engine. As illustrated below, initially the intake valve is open, allowing the air and fuel mixture to enter the combustion chamber, and the piston is in the 'down' position. Next, the intake valve is closed, and the piston, having reached the bottom of its downward travel, starts its way back up, compressing the fuel and air mixture around the spark plug. In the next stroke, the spark plug generates a spark that ignites the fuel and air mixture, which in turn forces the piston downwards with force proportional to the amount of fuel and air in the combustion chamber. In the final stroke, the exhaust valve opens to expel the exhaust and the cylinder travels upwards once again.



The four strokes of a simple piston engine

A logic program consisting of a static collection of unit clauses is clearly insufficient to model the situation. A piston's position, for instance, can have many different values, but it can only have one value at any particular instant in time. Thus, we may simulate the continuous, time-varying nature of the piston's position by sampling its movement at sufficiently frequent intervals to capture its motion in sufficient detail. In our model, we will only be interested in two general classes of piston motion: piston movement upward and piston movement downward. Accordingly, we may now use two basic unit clause forms to represent piston position:

```
piston_position(TA, piston_upwards).
piston_position(TB, piston_downwards).
```

Given an infinite number of these `piston_position` clauses with time contexts `TA` and `TB` ranging over all of the moments in time, we could simulate the continuous action of the piston. Fortunately, however, in many situations including this one, a periodic relation exists between elements being modelled, allowing us to use Prolog rules to represent concisely the value of the object at multiple positions in time. A complete Prolog program that simulates certain aspects of the piston engine using this approach is given below.

```
piston_movement(0, piston_down).
intake_valve(0, intake_open).
sparkplug(0, no_spark).
exhaust_valve(0, exhaust_closed).

piston_movement(T1, piston_up) :-
    T0 is T1 - 1,
    piston_movement(T0, piston_down).

piston_movement(T1, piston_down) :-
    T0 is T1 - 1,
    piston_movement(T0, piston_up).

intake_valve(T1, intake_closed) :-
    T0 is T1 - 1,
    intake_valve(T0, intake_open).

intake_valve(T1, intake_open) :-
    T0 is T1 - 1,
    exhaust_valve(T0, exhaust_open).

sparkplug(T2, spark) :-
    T0 is T2 - 2,
    intake_valve(T0, intake_open).
```



```

sparkplug(T1, no_spark) :-
    T0 is T1 - 1,
    sparkplug(spark).

exhaust_valve(T1, exhaust_open) :-
    T0 is T1 - 1,
    sparkplug(T0, spark).

exhaust_valve(T1, exhaust_closed) :-
    T0 is T1 - 1,
    exhaust_valve(T0, exhaust_open).

simulation(T) :-
    sparkplug(T, Spark_Status),
    intake_valve(T, Intake_Status),
    exhaust_valve(T, Exhaust_Status),
    piston_movement(T, Piston_Status),
    nl,
    write('ENGINE: '),
    write(Spark_Status), write(' '),
    write(Intake_Status), write(' '),
    write(Exhaust_Status), write(' '),
    write(Piston_Status),
    Next_T is T + 1,
    simulation(Next_T).

```

We can start the simulation by issuing the query

```

?- simulation(0).

ENGINE: no_spark intake_open exhaust_closed piston_down
ENGINE: no_spark intake_closed exhaust_closed piston_up
ENGINE: spark intake_closed exhaust_closed piston_down
ENGINE: no_spark intake_closed exhaust_open piston_up
ENGINE: no_spark intake_open exhaust_closed piston_down
ENGINE: no_spark intake_closed exhaust_closed piston_up
ENGINE: spark intake_closed exhaust_closed piston_down
ENGINE: no_spark intake_closed exhaust_open piston_up
:
:

```

Although this is probably one of the better ways to specify the simulation in Prolog, there are several problems associated with it. We defer the discussion of this program's relative merit, however, until the next section, when we will have something to compare it with.

2 Chronolog and Tense Logic

Because of its demonstrated importance in many of our computations, the notion of time has been built in to several tense logic languages [Wa88, AbMa87]. Particularly simple and elegant is Wadge's language Chronolog. Clauses in Chronolog are merely Horn clauses in which logical formulas may have 'tense prefixes'. The clause

```
next fire :- smoke.
```

for instance, has the loose declarative reading, "if we have smoke currently, then at the next instant in time, we will have fire." The prefixes `first` and `next` may be used to specify tense relationships between the head and body literals of a clause, and between the head of a clause and the current goal.

Returning to our engine simulation, we could translate the Prolog-based time tag manipulating program rather easily into the Chronolog program given below.

```
first piston_movement(piston_down).
first intake_valve(intake_open).
first sparkplug(no_spark).
first exhaust_valve(exhaust_closed).

next piston_movement(piston_up) :- piston_movement(piston_down).
next piston_movement(piston_down) :- piston_movement(piston_up).

next intake_valve(intake_closed) :- intake_valve(intake_open).
next intake_valve(intake_open) :- exhaust_valve(exhaust_open).

sparkplug(spark) :- prev prev intake_valve(intake_open).
next sparkplug(no_spark) :- sparkplug(spark).

next exhaust_valve(exhaust_open) :- sparkplug(spark).
next exhaust_valve(exhaust_closed) :- exhaust_valve(exhaust_open).

simulation :-
    sparkplug(Spark_Status),
    intake_valve(Intake_Status),
    exhaust_valve(Exhaust_Status),
    piston_movement(Piston_Status),
    nl,
    write('ENGINE: '),
    write(Spark_Status), write(' '),
    write(Intake_Status), write(' '),
    write(Exhaust_Status), write(' '),
    write(Piston_Status),
    next simulation.
```

There are several reasons why the Chronolog program above is better than the Prolog program given previously. Most obviously, the Chronolog version is more concise and easier to understand. Not as apparent is the fact that the Chronolog program will run much more efficiently because all time tag manipulation is carried out internally by the interpreter. The Prolog version, on the other hand, must expend vast quantities of logical inferences just manipulating contexts.

3 Intense

Now that we have presented some justifications for building time contexts into logic languages, let us examine related extensions. Following the lead of Lucid [WaAs85], the innovative functional language which uses time dependencies over objects in a given algebra to introduce streams, we might consider incorporating multiple time and space dimensions into a logic language. In fact, a language which contains these features has been implemented and is currently under study at Arizona State University. The intensional Horn-clause logic language **InTense** allows an intensional style of programming similar to that of Lucid, while retaining the powerful pattern matching and inferencing features of Prolog.

We have already presented an example of a logic program that makes use of time, so now let us see to what use the added space dimensions of InTense may be put. The InTense program below is the InTense analog of the Lucid program to generate prime numbers using the "Sieve of Eratosthenes" method.

```

initial ints(2) :- !.
rest ints(X) :-
    ints(Y),
    X is Y + 1.

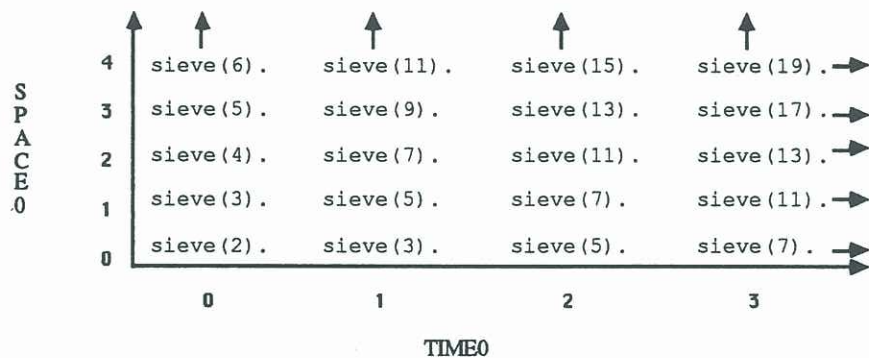
first sieve(X) :- ints(X), !.
next sieve(X) :-
    initial sieve(Y),
    newsmallest(X),
    not(0 is X mod Y).

newsmallest(X) :- sieve(X).
newsmallest(X) :- rest newsmallest(X).

prime(X) :- initial sieve(X).

```

The illustration below shows what the sieve relations look like in time and space dimensions zero during an execution of `prime`.



The sieve relations in time and space

Although Lucid like examples such as this might be construed to justify the use of space as well as time intensionality in logic programs it is still too early to really evaluate the benefits of the InTense "multiple time and space dimensions" approach (however, see [Mi88] for more examples involving space).

4 InTense Syntax

Now that we have seen roughly what an InTense program looks like, let us formalize the syntax. InTense programs are syntactically identical to Prolog programs, except that every formula in a clause may be modified by an intensional keyword. Three symmetric keywords are available for every dimension. For spatial dimensions these are:

```
prior
initial
rest
```

Temporal dimensions use the keywords:

```
prev
first
next
```

In the zeroth time and space dimensions, the above keywords may be used as-is. To specify relations in higher dimensions, an integer corresponding to the number of the dimension must be appended to the end of a keyword. Thus, the formula

```
first next next sunny_day.
```

specifies that the relation `sunny_day` holds in dimension $\text{TIME}(0) = 2$ and the formula

```
initial2 rest2 rest2 rest2 velocity(ball, 10)
```

indicates that the speed of `ball` in dimension $\text{SPACE}(2) = 3$ is 10.

Intensional keywords may be mixed across dimensions and no special ordering is required. As syntactic sugar, InTense provides an "alias" builtin which allows the user to change the names of the intensional keywords to fit the application. For example, a common set of aliases is:

```
alias(initial, 'x_origin').
alias(prior, '-x').
alias(rest, '+x').
alias(initial1, 'y_origin').
alias(prior1, '-y').
alias(rest1, '+y').
```


5 InTense Semantics

InTense programs denote different things, depending on whether they are viewed **intensionally** or **extensionally**. When thought of extensionally, any InTense program is capable of generating a set of ground terms, each of which has time and/or space contexts associated with it, which restricts it to one or more points in the time-space hyperfield. Each point in this hyperfield can be thought of as a *possible world* (after Kripke [Kr63].) Many different ground terms may inhabit a world. Thus, extensionally-speaking, the universe for an InTense program is the set of all possible worlds.

The **intensional** point of view dictates that an InTense program specifies a set of streams of relations that may vary through multiple dimensions of time and space. To illustrate the difference between these two perspectives, consider the InTense program to generate a time stream of Fibonacci numbers, given below.

```
first fib(0).
fib(X) :- prev prev fib(Y), prev fib(Z), X is Y + Z.
```

If we are thinking about the program intensionally, it can be viewed as creating a single, time-varying 'fib' predicate, which is initially

```
fib(0).
```

then, at the following moment in time, is

```
fib(1)
```

and still another moment later is

```
fib(2)
```

Contrast this with the extensional interpretation, in which the program is merely generating a collection of ground terms like:

Because the context manipulation which needs to be done in an InTense program can be done internal to the interpreter, as part of the matching process, the interworld inferencing illustrated above can actually proceed quite efficiently.

6 Implementation

InTense was first implemented in Prolog as a Prolog to InTense clause compiler. This compiler works by adding two context arguments to the front of every InTense formula, one to hold time-tags, and one for space tags. Each of these tag fields is a partially specified list (a list with a variable as it's tail) and thus the implementation supports an infinite number (limited by the allowed list length of the underlying Prolog) of time and space dimensions.

As we have pointed out (and Wadge notes in [Wa88]), this type of implementation is not very efficient. The performance edge that an intensional prematch could give an InTense program is entirely lost in the Prolog-based implementation. More importantly, a reasonable implementation of InTense surely requires some kind of term caching since many typical programs depend on the use of recursively defined time and space varying streams, as we saw, for instance, in the prime number program given earlier. Unfortunately, Prolog does not provide low-level enough access to it's clause base and control mechanism to implement effective caching. Consequently, a second InTense interpreter we have implemented at Arizona State University is essentially a full-blown Prolog interpreter as well as an interpreter for InTense and consists of approximately 5000 lines of "C" code. The interpreter operates at speeds comparable to that of other advanced Prolog interpreters such as C-Prolog when running straight Prolog programs.

7 Future Work

There are many complex interactions that can take place in InTense between it's control strategy and it's multiple dimensionality. For instance, the simple traffic light simulation

```

first light(red).
next light(green) :- light(red).
next light(yellow) :- light(green).
next light(red) :- light(yellow).

```

Will not answer the query

```

?- first next light(red).

```

correctly ('no.' is the correct answer) under the current implementation. In fact, the interpreter will keep drawing recursive inferences, using the `next light(X) :- light(Y)` clauses, throughout negative time until the stack overflows. Chronolog does not have this problem, as it only allows positive time. More detailed studies of the interactions between backtracking and intensional streams and the relation of InTense to Field Lucid also need to be done.

Another area for further work is the parallelization of InTense. As Ashcroft and Faustini [AsFa89] have mentioned, InTense programs can potentially benefit from traditional AND and OR parallelisms, as well as an additional mode of parallelism based on its time and space varying streams of formulas. For this reason, InTense programs have the potential for running even faster than Prolog programs.

References

- [AbMa87] Abadi, M. and Manna, Z. "Temporal logic programming". In *Proceedings of the 1987 Symposium on Logic Programming* (San Francisco, CA, Aug.31-Sept.4, 1987) pp. 4-16.
- [AsFa89] Ashcroft, E.A, and Faustini, A.A. "Parallelism Through Intensionality". Tech. Report # ???, Arizona State University, Dept. of Computer Science, 1989.
- [CoCo80] Coelho, H., Cotta, J. and Pereira, L. *How to Solve it With Prolog*. Laboratorio Nacional de Engenharia Civil, Lisbon, 1980.