logic programming in C# using TED

# Overview

TED is an embedded logic programming language intended to be used for simulation games, particular large-scale character simulations such as Dwarf Fortress or Talk of the Town. Its main features are:

* It’s embedded in C#, meaning that TED code is technically also C# code. This gives you compile-time type checking, makes it easier to mix-and-match with other C# code and lets you lean on the C# IDE (VS or Rider) for things like autocompletion, displaying documentation strings, and so.
* It’s relatively easy to wrap C# methods as predicates and functions within TED.
* Evaluation is mostly in the style of Datalog: bottom-up, tabled evaluation with support for indexing (TED originally stood for Typed Embedded DataLog, although it has since diverged significantly from Datalog). This means execution involves very little pointer chasing and has good cache performance. It also allows some support for parallel evaluation, meaning you can leverage multiple cores if you like.
* It enforces left-to-right execution of subgoals in rules, as in Prolog. This has the same advantages and disadvantages as in Prolog. The disadvantage is that as a programmer, you need to be aware of the left-to-right execution and the order in which variables get bound (left to right). Different orders of subgoals can have different performance properties. Some orders may be disallowed, although the system will at least tell you what’s wrong in those cases. The advantage of left-to-right execution is that it allows you to overlay an imperative semantics on the declarative semantics: you rules can call functions with side-effects, or which are probabilistic. So the language is more expressive.

One thing that TED does not support, which Datalog and Prolog both do, is recursion. Supporting recursion in TED would greatly complicate the evaluation model,[[1]](#footnote-1) and the obvious use cases for recursion are handled as well or better by callouts to C# code.

# Basic concepts

TED programs primarily consist of **predicates** and **functions**. Functions are like functions in C#, and are in fact, just wrappers for functions in C#.

Predicates can be thought of either in the standard math/programming language sense of being functions that return Booleans, or in the logic/database sense of being relations. Predicates are typed, meaning you specify the types of their arguments when you create them, and those types are enforced by the C# compiler; TED code doesn’t generally require run-time type checking.

Predicates are either **primitive predicates**, which again are generally wrappers for Boolean functions in C#, or **table predicates**, which are ultimately tables like in a relational database. Table predicates are essentially big, typed arrays of typed tuples. Table predicates are defined by explicitly providing the set of tuples (from a file or from C# code), or by using **rules** to specify them in terms of other predicates. All the real computation in TED is done using rules.

Rules are essentially database queries: they say find all the values of some set of **variables** that are consistent with some other predicates, and place them as a tuple in the table predicate defined by the rule. A table predicate can have multiple rules, in which case the contents of the table is the union of all the tuples derived by the predicate’s rules.

## Example: Making a table predicate from a CSV file

Suppose we have the following data in a CSV (spreadsheet) file:

|  |  |
| --- | --- |
| Name | Age |
| Fred | 10 |
| Jenny | 12 |
| Tamika | 11 |
| Elroy | 9 |

We can load this data into a table predicate by saying:

var Person = TablePredicate<string,int>.FromCsv("Person", "TestTable.csv");

This will create the table predicate and fill it with the data from the file. The second argument to FromCsv is the path to the CSV file. The first argument is a name for the predicate. The name has no functional significance. It’s just there so that if you see a TablePredicate object in the debugger you’ll at least be able to tell which one it is. You can also just say:

var Person = TablePredicate<string,int>.FromCsv("TestTable.csv");

and it will default the name to “TestTable” rather than “Person”.

The table predicate is functionally an array of (string,int) tuples. It knows from the file that the first column of the table/first element of the tuples is called “Name” and the second is called “Age”. It knows from you type declaration that Name is a string and Age is an int.

There is no direct access to the underlying array, but you can iterate over the data in a TablePredicate using the Rows property:

foreach (var (name, age) in Person.Rows)

Debug.Log($"Name={name}, Age={age}");

The Rows property is a typed enumerator for the rows/tuples of the table. So in this case, its type is IEnumerable<(string,int)>.

## Example: Making a table predicate from a rule

Now let’s define a new predicate, Child, in terms of Person. We do this by giving it the rule “a person *p* whose age *a* is less than 18 is a child”. We can do that by saying:

var p = (Var<string>)"p";  
var a = (Var<int>)"a";  
var Child = Predicate("Child", p).If(Person[p,a], a < 18);

This requires some explanation. First, since TED is an embedded language inside C#, and C# doesn’t let you overload function calls (the “()” operation), but does let you overload indexing (the “[]” operation), you write “calls” to predicate using square brackets rather than parentheses.

Second, for the same reason, rules specified in TED are going to be C# data structures that get mapped at run-time into the internal structures used by TED’s interpreter. So predicates, rules, and variables in TED are C# objects. To make a variable for TED rule, we need to create a C# object. That’s what the first two lines do. They say p and a are TED variables whose types are string, and int, respectively. The string at the end is, again, just a name that’s there so that if you encounter it inside the debugger, you know which Var<string> it’s talking about.

The real work here is the last line, which creates the predicate Child, and says that Child[p] is true any time Person[p,a] is true and a < 18 is also true. So ultimately, this is saying “loop through all the (p,a) in Person, and for each one where a<18, add it to child.” However, it’s able to do this fairly efficiently.

This is a good time to mention that it won’t work if we reverse the order of the conditions in the If:

var Child = Predicate("Child", p).If(a < 18, Person[p,a]);

The reason this won’t work is that it runs the If rules from left to right. In the original version, it starts by finding a tuple in Person. That gives it concrete values for p and a, and the particular a value then gets tested by < to see if it’s viable.

But in the second version, it starts with a < 18. It effectively says “go through all the integers less than 18 and for each one, go through the people to see what people are that age.” The < predicate isn’t a table predicate; it’s a primitive predicate and it only works when you already have concrete values for its arguments that you want to ask about. There isn’t a literal table in the system of all pairs of numbers where the first number is less than the second. And even if there was, there are too many 32 bit integers less than 18 to make it practical to start by guessing a number less than 18 and then looking to see what entries in Person are that age.

Instead of defining this in one line, we could write it this way:

var Child = Predicate("Child", p);  
Child[p].If(a < 18, Person[p,a]);

This is longer, but arguably more readable.[[2]](#footnote-2) We can also use this form to define predicate that use multiple rules, or rules where the conclusion includes things besides variables. For example, we could write something like:

var p = (Var<string>)"p";  
var a = (Var<int>)"a";  
var c = (Var<string>)"c";  
  
var AgeClass = Predicate("AgeClass", p, c);  
AgeClass[p, "child"].If(Person[p,a], a < 18);  
AgeClass[p, "adult"].If(Person[p,a], a >= 18, a <= 65);  
AgeClass[p, "senior"].If(Person[p,a], a > 65);

This makes a table of all the people, labeling them as child, adult, or senior, based on their ages.

## Example: making a Table Predicate from data generated from C# code

Now suppose we didn’t have the data in a file, but we instead had an array of Person objects, each of which had Name and Age fields. Then we could generate the same table by saying:

var n = (Var<string>)"name";  
var a = (Var<int>)"age";  
var Person = Predicate("Person",  
 people.Select(p => (p.Name, p.Age)),  
 n, a);

Here the variables are there just to let the compiler infer the type of the predicate and to provide names for the arguments. There’s a way to do it without passing in variables, but sooner or later, you’ll need to make variables to access the thing, so you might as well use them here.

1. Supporting recursion in a bottom-up language means that evaluation is a fixed-point algorithm that repeatedly runs rules until convergence. Moreover, it needs to run multiple variants of any rule that contains a recursive predicate. It also makes negation more complicated. At the same time, the main use cases for recursion involve transitive closure, which can be supported more efficiently by just giving an optimized implementation in C#. [↑](#footnote-ref-1)
2. You might wonder why we need to pass p as an argument in the first line. You don’t; you can instead say new TablePredicate<string>("Child", “person”) . But given that you’re going to end up making a variable to quantify over people anyway, it’s easier just to pass that to the factory method and let the compiler get both the type and the argument name from the variable. [↑](#footnote-ref-2)