Anonymous and First Class Operators for Tutorial D

Dave Voorhis <dave@armchair.mb.ca>
Version 1.01 August 2012

Introduction

This is an informal account of work-in-progress extensions to Date and Darwen's **Tutorial D** database language to support anonymous functions/procedures. Much of this work was inspired by discussions on *The Third Manifesto* e-mail forum[1], and – except where noted – has been implemented and tested in *Rel*.[2]

The remainder of this introduction provides a brief, pragmatic, incomplete[3], but hopefully relevant general overview of anonymous and first class functions. Readers familiar with anonymous and first class functions may wish to skip to the *Operators in **Tutorial D*** section. This document assumes familiarity with **Tutorial D** syntax.

Anonymous Functions

In languages that do not support anonymous functions – e.g., C, Pascal, SQL and Java – every function/procedure definition must be given a name.

For example, a hypothetical procedural language of conventional design will almost undoubtedly allow a definition of a function named “plus” like this...

```
function plus(x integer, y integer) returns integer {
   return x + y
}
```

...and an invocation of that definition referencing it by its name, “plus”, like this:

```
plus(2, 3)
```

An anonymous function – also known as a lambda expression – is a function (or procedure) definition that does not have a name. A hypothetical procedural language that allows anonymous functions might permit a definition like this:

```
function (x integer, y integer) returns integer {
   return x + y
}
```

Note the absence of any function name. In languages that support anonymous function definitions, they may appear wherever a function invocation would typically reference a function name.

---

1 The TTM mailing list is a discussion forum for parties interested in the relational model and database languages in general, and particularly in publications – predominately those by Hugh Darwen and C J Date – related to these topics. See http://thethirdmanifesto.com

2 *Rel* is an implementation of **Tutorial D**. See http://dbappbuilder.sourceforge.net/Rel.html As of this writing, the version of *Rel* that implements anonymous operators has not yet been publicly released.

3 In particular, absent are any discussions of underlying theory or practical issues like allowable scope of identifier references.

For example, the following defines and then invokes an anonymous function:

```
(function (x integer, y integer) returns integer {
   return x + y
}) (2, 3)
```

The definition of the anonymous function has taken the place of the function name in the invocation.

First Class Functions

Typically, languages that support anonymous functions/procedures also allow functions/procedures to – in addition to being invoked or executed as usual – be assigned to variables or constants, passed to functions/procedures as arguments, returned from functions, and stored in data structures. A language which allows functions/procedures to be treated in this manner, i.e., as function values, is said to support “first class” functions.

For example, we might be able to assign a function to a variable like this...

```
var plus = function (x integer, y integer) returns integer {
        return x + y
}
```

...and perhaps invoke it like this:

```
(plus)(2, 3)
```

Parentheses are used around the variable name to clearly indicate that this is not an invocation of a function named “plus”, but a dereference of a variable named “plus” to obtain its value, which is a function. The function value is then passed arguments (2 and 3) and invoked.

A language that supports anonymous functions need not support first class functions; a language that supports first-class functions need not support anonymous functions. However, these language features almost always appear together. Anonymous functions and first class functions are found in various programming languages including Haskell, assorted LISP dialects, C#, Python, Javascript and PHP. The practical applications for anonymous and first class functions are myriad, but beyond the scope of this document.

*Operators in **Tutorial D***

As defined in published specifications by Date and Darwen, **Tutorial D** supports conventional named procedure and function definitions, with polymorphic invocations based on parameter type inheritance and multiple dispatch[4]. Date and Darwen call functions and procedures “operators”. This document upholds their practice by now dispensing with the terms “procedure” and “function” in favour of “operator”.

4 The details of which are not described here, nor are they relevant to this discourse.

Defining a named operator is straightforward[5], for example:

```
OPERATOR plus(x INTEGER, y INTEGER) RETURNS INTEGER;
     RETURN x + y;
END OPERATOR;
```

The operator can be invoked as follows:

```
plus(2, 3)
```

The published **Tutorial D** specifications do not define anonymous or first class operators.

Anonymous Operators

We propose to extend **Tutorial D** by allowing definition of anonymous operators using familiar **Tutorial D** syntax. The name is simply left out. For example:

```
OPERATOR (x INTEGER, y INTEGER) RETURNS INTEGER;
     RETURN x + y;
END OPERATOR
```

This extension intends to make **Tutorial D** support first class operators, so the above represents a selector for an operator value.[6]

In the *Rel* implementation of **Tutorial D**, the above is an expression that can be evaluated. Evaluating it from a *Rel* command line returns the text of the operator definition enclosed in double quotes[7]. This provides a human-readable representation of a machine-executable operator.

For example, in an interactive session in *Rel*'s DBrowser command-line tool, entering the following...

```
OPERATOR (x INTEGER) RETURNS INTEGER;
     RETURN x + 1;
END OPERATOR
```

...returns the following:

```
"OPERATOR ( x INTEGER ) RETURNS INTEGER ; RETURN x + 1 ; END
OPERATOR"
```

5 The use of upper case keywords is a convention to distinguish keywords from user-defined identifiers. It is not required by the language.

6 Whilst verbose, it is intentionally in keeping with existing syntax. As **Tutorial D** is intended primarily for pedagogical purposes, it is appropriate to favour consistency over brevity. However, the author grew weary of endlessly typing OPERATOR during this phase of *Rel* development and so created a shorthand form of anonymous operator definition, using a pair of digraphs in place of “OPERATOR” and “END OPERATOR” which the author has deemed too aesthetically heinous to mention here.

7 The double quotes surrounding the definition allows client-side parsers to accept operator values without having to parse their contents. They can simply be read as arbitrary character strings.

An anonymous operator definition may be used wherever an operator name can appear, as long as it is inside parentheses. For example, the above can be defined and immediately invoked as:

```
(OPERATOR (x INTEGER, y INTEGER) RETURNS INTEGER;
     RETURN x + y;
END OPERATOR) (2, 3)
```

First Class Operators

Since operators are values, they can be assigned to variables. For example, the following is allowable:

```
VAR myvar INIT(
     OPERATOR (x INTEGER) RETURNS INTEGER;
          RETURN x+1;
     END OPERATOR);
```

The contents of variable “myvar” – which is an operator – can be invoked as follows:

```
(myvar)(2)
```

Note the parentheses around “myvar”. The operator in “myvar” may <u>not</u> be invoked as follows:

```
myvar(2)
```

The above is strictly considered an attempt to invoke a named operator called “myvar”.

Operator Types

Operator types may be explicitly specified. The variable above could have been defined as:

```
VAR myvar OPERATOR (INTEGER) RETURNS INTEGER;
```

An operator type is specified as the keyword OPERATOR, followed by a comma-separated list of parameter types enclosed by parentheses, optionally followed by the RETURNS keyword followed by the return type.

An operator type may appear wherever a type may appear. Thus, parameters, tuple and relation attributes, variables, and return types may all be of type OPERATOR.

Higher-order Operators

Operators may return operators. Thus, higher order operators may be defined.

For example, the following is allowed:

```
VAR myvar INIT (OPERATOR (x INTEGER, y INTEGER) RETURNS
OPERATOR (INTEGER) RETURNS INTEGER;
     RETURN OPERATOR (p INTEGER) RETURNS INTEGER;
          RETURN x + y + p;
     END OPERATOR;
END OPERATOR);
```

The above may be invoked as follows:

```
(myvar)(2, 3)
```

It will return an operator of type `OPERATOR (INTEGER) RETURNS INTEGER`, which can be invoked as follows:

```
((myvar)(2, 3))(5)
```

The return value will be an INTEGER; 10 in this case.

Anonymous Operators in Relations

As noted above, operator types may appear wherever types may appear. Therefore, tuple and relation attributes may be of type OPERATOR. The following is allowed:

```
VAR myvar REAL RELATION {
  x INTEGER,
  y OPERATOR (INTEGER, INTEGER) RETURNS INTEGER
} KEY {x};
```

Attribute 'y' is of type OPERATOR, which accepts two integer arguments and returns an integer. The relation-valued variable (aka relvar) “myvar” can be assigned a relation:

```
myvar := RELATION {
  TUPLE {x 1, y OPERATOR (a INTEGER, b INTEGER) RETURNS INTEGER;
     RETURN a + b; END OPERATOR},
  TUPLE {x 2, y OPERATOR (a INTEGER, b INTEGER) RETURNS INTEGER;
     RETURN a - b; END OPERATOR},
  TUPLE {x 3, y OPERATOR (a INTEGER, b INTEGER) RETURNS INTEGER;
     RETURN a * b; END OPERATOR},
  TUPLE {x 4, y OPERATOR (a INTEGER, b INTEGER) RETURNS INTEGER;
     RETURN a / b; END OPERATOR}
};
```

The variable has been assigned a relation of tuples, each consisting of an integer 'x' and an operator value 'y' conforming to the operator type in the relation heading.

The attribute values of 'y' can be invoked as (for example) follows:

```
EXTEND myvar: {r := (y)(x, 2)}
```

This expression extends myvar with the result 'r' of evaluating the operator in y with arguments x and 2. Evaluating it in *Rel's* DBrowser returns the following:

x *INTEGER*	y OPERATOR (*INTEGER, INTEGER*) RETURNS *INTEGER*	r *INTEGER*
1	OPERATOR (a INTEGER , b INTEGER) RETURNS INTEGER ; RETURN a + b ; END OPERATOR	3
2	OPERATOR (a INTEGER , b INTEGER) RETURNS INTEGER ; RETURN a - b ; END OPERATOR	0
3	OPERATOR (a INTEGER , b INTEGER) RETURNS INTEGER ; RETURN a * b ; END OPERATOR	6
4	OPERATOR (a INTEGER , b INTEGER) RETURNS INTEGER ; RETURN a / b ; END OPERATOR	2

Further Work

Scope and Allowable References

The astute reader will note that none of the above examples make reference to local or global variables or other operators. This is deliberate, as the restrictions – if any – that should be placed on such references are still to be considered.

When storing operators in a relvar, identifier references that are in scope at the point of insertion will not necessarily be present at the point of retrieval. For example, the following is currently allowed:

```
VAR myvar REAL RELATION {
     x INTEGER,
     y OPERATOR (INTEGER) RETURNS INTEGER
} KEY {x};
VAR k INTEGER;
DO k := 1 TO 10;
     INSERT myvar RELATION {
          TUPLE {x k, y OPERATOR (q INTEGER) RETURNS INTEGER;
                    RETURN q + k;
               END OPERATOR}
     };
END DO;
```

Note the reference to transient variable 'k' in the anonymous operator definitions being inserted into myvar.

In *Rel*, such references become unbound at the point of insertion. When the operators are later invoked, *Rel* will attempt to re-bind them in their own isolated run-time scope. If no appropriate binding can be made, an error will be thrown. For example...

```
EXTEND myvar: {r := (y)(2)}
```

...returns:

```
ERROR: 'k' has not been defined.
Line 1, column 53 near 'k'
```

It might be reasonable to attempt to re-bind references to local scope upon invocation, but this raises the potential for unpleasant surprises: unbound references will attempt to bind to whatever matching names they find, whether intended or not.

Alternatively, the values of variables like 'k' could be captured in a closure at the point of inserting the tuple containing the operator. However, that closure would either have to be “hidden” somewhere in the tuple – a violation of Codd's Information Principle – or exposed in some fashion, perhaps in an additional attribute or by automated rewriting of the operator definition. Clearly, these are unsatisfactory solutions.

Therefore, for the sake of predictability, consistency and simplicity, it may be preferable to require that anonymous operators be strictly limited to referencing persistent relvars, and that all other dynamic values must be supplied via parameters. If even that is considered too permissive, references to persistent relvars can be disallowed as well, and thus all non-literal values must be parametrised.

This is a matter for further consideration.

Use of Named Operators as Values

It should be possible to obtain the value of named operators. For example, given the following...

```
OPERATOR op1(x INTEGER) RETURNS INTEGER;
     RETURN x * 2;
END OPERATOR;

OPERATOR op2(p INTEGER, q OPERATOR(INTEGER) RETURNS INTEGER)
RETURNS INTEGER;
     RETURN (q)(p + 2);
END OPERATOR;
```

...it should be possible to evaluate the following:

```
op2(5, op1)
```

Note that op1 is passed as an argument to op2. As of this writing, whilst the two operator definitions may be created in *Rel*, op1 may not be passed as an argument to op2. This is a work in progress, noting that such operator invocations may be polymorphic, dependent on the most specific type of the parameters at the point of invocation.

User-defined Aggregate Operators for SUMMARIZE

SUMMARIZE currently only supports built-in aggregate operators, like AVG, SUM, MIN, EXACTLYD, etc. An obvious application for first class operators is to allow user-defined aggregate operators to be passed as arguments to SUMMARIZE.

Consideration for how best to implement this is a work in progress.

END