File: User-defined relational operators in TD.doc
Printed at: 17:03 on Monday, 18 November, 2013

Extending Tutorial D to Support User-Defined Generic Relation and Tuple Operators

Hugh Darwen

The Third Manifesto's Very Strong Suggestion 6 states

> **D** should provide some means for users to define their own generic **operators**, including in particular generic **relational** operators.

I propose some extensions to **Tutorial D** to this end. I assume that static type checking remains in effect. My proposals concern only relation and tuple operators and they restrict the feature to operators whose declared types, on invocation, can be inferred from the declared types of the expressions substituted for their parameters.

The proposals that follow refer to BNF terms used in the **Tutorial D** grammar given at www.thethirdmanifesto.com.

1. Wild cards in invocations of type generators

The wild card * can be used in place of

- (a) an <*attribute commalist*> contained in a <*heading*>,
- (b) the <*type spec*> contained in an <*attribute commalist*>, or
- (c) possibly, as part of such a <*type spec*> (e.g., INTERVAL_*).

Note that an <*attribute commalist*> having more than one <*attribute*> contains shorter <*attribute commalist*>s. Examples:

- RELATION { * } specifies any relation type.
- RELATION { A INTEGER, * } specifies any relation type whose heading contains an attribute named A of type INTEGER.
- RELATION { A *, * } specifies any relation type whose heading contains an attribute named A of any type.

This feature alone affords limited support for user-defined generic operators. Examples:

- ```
  OPERATOR MATCHING_S (R RELATION { * })
           RETURNS SAME_TYPE_AS ( R );
    RETURN R MATCHING S ;
  END OPERATOR ;
  ```

- ```
  OPERATOR REMOVE_A (R RELATION { A *, * })
           RETURNS SAME_TYPE_AS ( R { ALL BUT A });
    RETURN R { ALL BUT A } ;
  END OPERATOR ;
  ```

- ```
  OPERATOR SUM_A (R RELATION { A INTEGER, * })
           RETURNS INTEGER;
    RETURN SUM( R, A ) ;
  END OPERATOR ;
  ```

But this barely gets us off the ground. For example, consider an operator that operates on two or more relations to yield a relation. The heading of its type might be the union of the headings of its relation operands, as in the system-defined JOIN operators (dyadic and *n*-adic). To support definitions of that kind I introduce a new relational operator, HEADING(<*relation exp*>), where HEADING(*r*) denotes a relation representing the heading of the declared type of *r*. I further propose alternative syntax for invoking the RELATION and TUPLE type generators, in which the operand is a <*relation exp*>, possibly using invocations of HEADING, denoting the required heading.

2. HEADING (*r*)

HEADING (*r*), where *r* is a <*relation exp*>, denotes a relation *s* of heading { NAME CHAR, TYPE CHAR } representing the predicate *TYPE is the declared type of attribute NAME of the heading of r*. Thus a set of tuples giving the names and declared types of all the attributes of *r* constitutes the body of *s*.

Implementation-defined canonical forms for type names are expected to be used in TYPE values denoting relation types and tuple types.

Note: HEADING (*r*) can be computed at compile-time, which is necessary if it is to be used in type specifications for operators and their parameters.

For convenience, HEADING is overloaded for tuples. Thus, HEADING (*t*), where *t* is a <*tuple exp*>, is shorthand for HEADING (RELATION { *t* }).

3. Use of <*relation exp*>s in type specifications

Here is the current BNF for <*relation type spec*>

```
<relation type spec>
    ::= <relation type name>
      | SAME_TYPE_AS ( <relation exp> )
      | RELATION SAME_HEADING_AS ( <nonscalar exp> )

<relation type name>
    ::= RELATION <heading>

<heading>
    ::= { <attribute commalist> }

<attribute>
    ::= <attribute name> <type spec>
```

I propose the following redefinition for <*relation type name*> to admit <*relation exp*> as an alternative to <*heading*>:

```
<relation type name>
    ::= RELATION <heading or relation-or-tuple exp>
```
```

```
<heading or relation-or-tuple exp>
    ::= <heading>
      | ( <relation-or-tuple exp> )

<relation-or-tuple exp>
    ::= <relation exp>
      | <tuple exp>
```

—where <*relation-or-tuple exp*> shall denote a relation or tuple of heading { NAME CHAR, TYPE CHAR } that satisfies the constraint KEY { NAME }. NAME values shall conform to the implementation-defined grammar for attribute names. TYPE values shall conform to the grammar for type names. *Note:* Invocations of HEADING are guaranteed by definition to satisfy these constraints.

If <*relation-or-tuple exp*> contains a <*var ref*> *v*, then *v* shall appear in an invocation of HEADING.[1]

<*tuple type name*> is then redefined as follows:

```
<tuple type name>
    ::= TUPLE <heading or relation-or-tuple exp>
```

Suppose we wish to define a shorthand for the projection of *r1* JOIN *r2* on the common attributes of r1 and *r2*. Well, we can now define the following signature, for example—

```
OPERATOR PROJECT_COMMON ( R1 RELATION { * } ,
                          R2 RELATION { * } )
   RETURNS RELATION ( HEADING ( R1 ) JOIN HEADING ( R2 ) ) ;
```

—but we cannot complete the operator definition, for we have no way of specifying that the projection is to be on the common attributes. I propose, therefore, to allow a <*relation exp*> as an alternative to <*attribute ref commalist*> for specifying a set of attribute names.[2]

3. Use of <*relation exp*>s in place of <*attribute ref commalist*>s

Wherever <*attribute ref commalist*> appears on the right-hand side of a BNF production I propose to replace it by <*attribute ref set spec*>, defined as

```
<attribute ref set spec exp>
    ::= <attribute ref commalist>
      | LIST <relation exp>
```

where <*relation exp*> shall denote a unary relation of heading { NAME CHAR }. NAME values shall conform to the implementation-defined grammar for attribute names.

If <*relation exp*> contains a <*var ref*> *v*, then *v* shall appear in an invocation of HEADING.

---

[1] This restriction ensures that the required heading can be computed at compile time.

[2] I use syntax that was provided in *Business System 12*, whose COLUMNS operator also inspired my HEADING proposal. A presentation and accompanying notes on this early relational DBMS are available at http://www.northumbria.ac.uk/sd/academic/ee/work/research/computerscience/computational_intelligence/third_manifesto/?view=Standard

Now we can complete the definition of PROJECT_COMMON:

```
OPERATOR PROJECT_COMMON ( R1 REL { * } ,
                          R2 REL { * } )
         RETURNS REL ( HEADING ( R1 ) JOIN HEADING ( R2 ) ) ;
  RETURN ( R1 JOIN R2 ) { LIST ( HEADING ( R1 ) JOIN HEADING ( R2 ) )
                               { NAME } } ;
END OPERATOR ;
```

Conclusion

In conclusion I conjecture that support for the extensions proposed in this paper would be sufficient for the definition of any[3] user-defined generic relation or tuple operator whose type, on a particular invocation *inv*, can be determined statically from the types of the expressions substituted for its parameters in *inv*.

End of paper

[3] That might be a bit too bold. For example, if an operator is required that operates on any relation of degree 5, that would require some extra syntax, or a wild card for use in place of an entire <*attribute*>.