Relations Template B/relt

Purpose

To manage run-time storage for relations between objects, and to find routes through relations and the map.

§**1. Relation Records.** See "RelationKind.i6t" for further explanation.

```
Constant RR_NAME      4;
Constant RR_PERMISSIONS 5;
Constant RR_STORAGE 6;
Constant RR_KIND 7;
Constant RR_HANDLER 8;
Constant RR_DESCRIPTION 9;
```

§**2. Valency Adjectives.** These are defined in the Standard Rules; the following routines must either test the state (if `set` is negative), or change the state to `set`.

```
Constant VALENCY_MASK = RELS_EQUIVALENCE+RELS_SYMMETRIC+RELS_X_UNIQUE+RELS_Y_UNIQUE;
[ RELATION_TY_EquivalenceAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & RELS_EQUIVALENCE) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) {
        perms = perms + RELS_EQUIVALENCE;
        if (perms & RELS_SYMMETRIC == 0) perms = perms + RELS_SYMMETRIC;
    }
    if ((set == false) && (state)) {
        perms = perms - RELS_EQUIVALENCE;
        if (perms & RELS_SYMMETRIC) perms = perms - RELS_SYMMETRIC;
    }
    rel-->RR_PERMISSIONS = perms;
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to an equivalence relation ***";
];
[ RELATION_TY_SymmetricAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & RELS_SYMMETRIC) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) perms = perms + RELS_SYMMETRIC;
    if ((set == false) && (state)) perms = perms - RELS_SYMMETRIC;
    rel-->RR_PERMISSIONS = perms;
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to a symmetric relation ***";
```

```
];
[ RELATION_TY_OToOAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & (RELS_X_UNIQUE+RELS_Y_UNIQUE) == RELS_X_UNIQUE+RELS_Y_UNIQUE) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) {
        if (perms & RELS_X_UNIQUE == 0) perms = perms + RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE == 0) perms = perms + RELS_Y_UNIQUE;
        if (perms & RELS_EQUIVALENCE) perms = perms - RELS_EQUIVALENCE;
    }
    if ((set == false) && (state)) {
        if (perms & RELS_X_UNIQUE) perms = perms - RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE) perms = perms - RELS_Y_UNIQUE;
    }
    rel-->RR_PERMISSIONS = perms;
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to a one-to-one relation ***";
];
[ RELATION_TY_OToVAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & (RELS_X_UNIQUE+RELS_Y_UNIQUE) == RELS_X_UNIQUE) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) {
        if (perms & RELS_X_UNIQUE == 0) perms = perms + RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE) perms = perms - RELS_Y_UNIQUE;
        if (perms & RELS_SYMMETRIC) perms = perms - RELS_SYMMETRIC;
        if (perms & RELS_EQUIVALENCE) perms = perms - RELS_EQUIVALENCE;
    }
    if ((set == false) && (state)) {
        if (perms & RELS_X_UNIQUE) perms = perms - RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE) perms = perms - RELS_Y_UNIQUE;
    }
    rel-->RR_PERMISSIONS = perms;
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to a one-to-various relation ***";
];
[ RELATION_TY_VToOAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & (RELS_X_UNIQUE+RELS_Y_UNIQUE) == RELS_Y_UNIQUE) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) {
        if (perms & RELS_X_UNIQUE) perms = perms - RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE == 0) perms = perms + RELS_Y_UNIQUE;
        if (perms & RELS_SYMMETRIC) perms = perms - RELS_SYMMETRIC;
        if (perms & RELS_EQUIVALENCE) perms = perms - RELS_EQUIVALENCE;
    }
    if ((set == false) && (state)) {
        if (perms & RELS_X_UNIQUE) perms = perms - RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE) perms = perms - RELS_Y_UNIQUE;
    }
    rel-->RR_PERMISSIONS = perms;
```

```
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to a various-to-one relation ***";
];
[ RELATION_TY_VToVAdjective rel set  perms state handler;
    perms = rel-->RR_PERMISSIONS;
    if (perms & (RELS_X_UNIQUE+RELS_Y_UNIQUE) == 0) state = true;
    if (set < 0) return state;
    if ((set) && (state == false)) {
        if (perms & RELS_X_UNIQUE) perms = perms - RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE) perms = perms - RELS_Y_UNIQUE;
    }
    if ((set == false) && (state)) {
        if (perms & RELS_X_UNIQUE == 0) perms = perms + RELS_X_UNIQUE;
        if (perms & RELS_Y_UNIQUE == 0) perms = perms + RELS_Y_UNIQUE;
    }
    rel-->RR_PERMISSIONS = perms;
    handler = rel-->RR_HANDLER;
    if (handler(rel, RELS_SET_VALENCY, perms & VALENCY_MASK) == 0)
        "*** Can't change this to a various-to-various relation ***";
];
```

§**3. One To One Relations.** We provide routines to assert a 1-to-1 relation true, or to assert it false. The relation `rel` is represented by a property number, and the property in question is used to store the fact of a relationship: $O_1 \sim O_2$ if and only if `O1.rel == O2`.

There is no routine to test a 1-to-1 relation, since the predicate calculus code in NI simplifies propositions which test these into direct looking up of the property relation.

```
[ Relation_Now1to1 obj1 relation_property obj2 ol; ! Assert 1-1 true
    if (obj2) objectloop (ol provides relation_property)
        if (ol.relation_property == obj2) ol.relation_property = nothing;
    if (obj1) obj1.relation_property = obj2;
];
[ Relation_NowN1toV obj1 relation_property obj2; ! Assert 1-1 false
    if ((obj1) && (obj1.relation_property == obj2)) obj1.relation_property = nothing;
];
[ Relation_Now1to1V obj1 obj2 KOV relation_property ol N; ! Assert 1-1 true
    if (obj2) {
        N = KOVDomainSize(KOV);
        for (ol=1: ol<=N: ol++)
            if (GProperty(KOV, ol, relation_property) == obj2)
                WriteGProperty(KOV, ol, relation_property, 0);
    }
    if (obj1) WriteGProperty(KOV, obj1, relation_property, obj2);
];
[ Relation_NowN1toVV obj1 obj2 KOV relation_property; ! Assert 1-1 false
    if ((obj1) && (GProperty(KOV, obj1, relation_property) == obj2))
        WriteGProperty(KOV, obj1, relation_property, 0);
];
```

§**4. Symmetric One To One Relations.** Here the relation is used for both objects: $O_1 \sim O_2$ if and only if both `O1.relation_property == O2` and `O2.relation_property == O1`.

```
[ Relation_NowS1to1 obj1 relation_property obj2; ! Assert symmetric 1-1 true
    if ((obj1 ofclass Object) && (obj1 provides relation_property) &&
        (obj2 ofclass Object) && (obj2 provides relation_property)) {
        if (obj1.relation_property) { (obj1.relation_property).relation_property = 0; }
        if (obj2.relation_property) { (obj2.relation_property).relation_property = 0; }
        obj1.relation_property = obj2; obj2.relation_property = obj1;
    }
];
[ Relation_NowSN1to1 obj1 relation_property obj2; ! Assert symmetric 1-1 false
    if ((obj1 ofclass Object) && (obj1 provides relation_property) &&
        (obj2 ofclass Object) && (obj2 provides relation_property) &&
        (obj1.relation_property == obj2)) {
        obj1.relation_property = 0; obj2.relation_property = 0;
    }
];
[ Relation_NowS1to1V obj1 obj2 KOV relation_property; ! Assert symmetric 1-1 true
    if (GProperty(KOV, obj1, relation_property))
        WriteGProperty(KOV, GProperty(KOV, obj1, relation_property), relation_property, 0);
    if (GProperty(KOV, obj2, relation_property))
        WriteGProperty(KOV, GProperty(KOV, obj2, relation_property), relation_property, 0);
    WriteGProperty(KOV, obj1, relation_property, obj2);
    WriteGProperty(KOV, obj2, relation_property, obj1);
];
[ Relation_NowSN1to1V obj1 obj2 KOV relation_property; ! Assert symmetric 1-1 false
    if (GProperty(KOV, obj1, relation_property) == obj2) {
        WriteGProperty(KOV, obj1, relation_property, 0);
        WriteGProperty(KOV, obj2, relation_property, 0);
    }
];
```

§**5. Various To Various Relations.** Here the relation is represented by an array holding its metadata. Each object in the domain of the relation provides two properties, holding its left index and its right index. The index is its position in the left or right domain. For instance, suppose we relate things to doors, and there are five things in the world, two of which are doors; then the left indexes will range from 0 to 4, while the right indexes will range from 0 to 1. It's very likely that the doors will have different left and right indexes. (If the relation relates a given kind to itself, say doors to doors, then left and right indexes will always be equal.)

It is possible for either the left or right domain set to be an enumerated kind of value, where the I6 representation of values is 1, 2, 3, ..., N, where there are N possibilities. In that case we obtain the index simply by subtracting 1 in order to begin from 0. We mark the domain set as being a KOV rather than a kind of object by storing 0 instead of a property in the relevant part of the relation metadata: note that 0 is not a valid property number.

The structure for a relation consists of eight `-->` words, followed by a bitmap in which we store 16 bits in each `-->` word. (Yes, this is wasteful in Glulx, where `-->` words store 32 bits, but memory is not in short supply in Glulx and the total cost of relations is in practice small; we prefer to keep all the code involved simple.) The structure is precompiled by the Inform compiler: we do not create new ones on the fly.

In the case of a symmetric various to various relation, we could in theory save memory once again by storing only the lower triangle of the bitmap, but the time and complexity overhead are not worth it. When asserting

that $O_1 \sim O_2$ for a symmetric V-to-V, we also automatically assert that $O_2 \sim O_1$, thus maintaining the bitmap as a symmetric matrix; but in reading the bitmap, we look only at the lower triangle. This costs a little time, but has the advantage of allowing the route-finding routine for V-to-V to use the same code for symmetric and asymmetric relations.

If this all seems rather suboptimally programmed in order to reduce code complexity, I can only say that careless drafts here were the source of some extremely difficult bugs to find.

```
Constant VTOVS_LEFT_INDEX_PROP = 0;
Constant VTOVS_RIGHT_INDEX_PROP = 1;
Constant VTOVS_LEFT_DOMAIN_SIZE = 2;
Constant VTOVS_RIGHT_DOMAIN_SIZE = 3;
Constant VTOVS_LEFT_PRINTING_ROUTINE = 4;
Constant VTOVS_RIGHT_PRINTING_ROUTINE = 5;
Constant VTOVS_CACHE_BROKEN = 6;
Constant VTOVS_CACHE = 7;

[ Relation_NowVtoV obj1 relation obj2 sym pr pr2 i1 i2 vtov_structure;
    if (sym && (obj2 ~= obj1)) { Relation_NowVtoV(obj2, relation, obj1, false); }
    vtov_structure = relation-->RR_STORAGE;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    vtov_structure-->VTOVS_CACHE_BROKEN = true; ! Mark any cache as broken
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else return RunTimeProblem(RTP_IMPREL, obj1, relation);
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else return RunTimeProblem(RTP_IMPREL, obj2, relation);
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    vtov_structure-->pr = (vtov_structure-->pr) | i1;
];
[ Relation_NowNVtoV obj1 relation obj2 sym pr pr2 i1 i2 vtov_structure;
    if (sym && (obj2 ~= obj1)) { Relation_NowNVtoV(obj2, relation, obj1, false); }
    vtov_structure = relation-->RR_STORAGE;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    vtov_structure-->VTOVS_CACHE_BROKEN = true; ! Mark any cache as broken
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else return RunTimeProblem(RTP_IMPREL, obj1, relation);
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else return RunTimeProblem(RTP_IMPREL, obj2, relation);
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    if ((vtov_structure-->pr) & i1) vtov_structure-->pr = vtov_structure-->pr - i1;
];
```

```
[ Relation_TestVtoV obj1 relation obj2 sym pr pr2 i1 i2 vtov_structure;
    vtov_structure = relation-->RR_STORAGE;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    if (sym && (obj2 > obj1)) { sym = obj1; obj1 = obj2; obj2 = sym; }
    if (pr) {
        if ((obj1 ofclass Object) && (obj1 provides pr)) i1 = obj1.pr;
        else { RunTimeProblem(RTP_IMPREL, obj1, relation); rfalse; }
    } else i1 = obj1-1;
    if (pr2) {
        if ((obj2 ofclass Object) && (obj2 provides pr2)) i2 = obj2.pr2;
        else { RunTimeProblem(RTP_IMPREL, obj2, relation); rfalse; }
    } else i2 = obj2-1;
    pr = i1*(vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE) + i2;
    i1 = IncreasingPowersOfTwo_TB-->(pr%16);
    pr = pr/16 + 8;
    if ((vtov_structure-->pr) & i1) rtrue; rfalse;
];
```

§**6. Equivalence Relations.** For every equivalence relation there is a corresponding function f such that $x \sim y$ if and only if $f(x) = f(y)$, where $f(x)$ is a number identifying the equivalence class of x. Rather than inefficiently storing a large relation bitmap (and then having a very complicated time updating it to keep the relation transitive), we store f: that is, for every object in the domain set, there is a property `prop` such that `O.prop` is the value $f(O)$.

```
[ Relation_NowEquiv obj1 relation_property obj2 big little;
    big = obj1.relation_property; little = obj2.relation_property;
    if (big == little) return;
    if (big < little) { little = obj1.relation_property; big = obj2.relation_property; }
    objectloop (obj1 provides relation_property)
        if (obj1.relation_property == big) obj1.relation_property = little;
];
[ Relation_NowNEquiv obj1 relation_property obj2 old new;
    old = obj1.relation_property; new = obj2.relation_property;
    if (old ~= new) return;
    new = 0;
    objectloop (obj2 provides relation_property)
        if (obj2.relation_property > new) new = obj2.relation_property;
    new++;
    obj1.relation_property = new;
];
[ Relation_NowEquivV obj1 obj2 KOV relation_property n big little i;
    big = GProperty(KOV, obj1, relation_property);
    little = GProperty(KOV, obj2, relation_property);
    if (big == little) return;
    if (big < little) {
        little = GProperty(KOV, obj1, relation_property);
        big = GProperty(KOV, obj2, relation_property);
    }
    n = KOVDomainSize(KOV);
    for (i=1: i<=n: i++)
        if (GProperty(KOV, i, relation_property) == big)
```

```
            WriteGProperty(KOV, i, relation_property, little);
];
[ Relation_NowNEquivV obj1 obj2 KOV relation_property n old new i;
    old = GProperty(KOV, obj1, relation_property);
    new = GProperty(KOV, obj2, relation_property);
    if (old ~= new) return;
    new = 0;
    n = KOVDomainSize(KOV);
    for (i=1: i<=n: i++)
        if (GProperty(KOV, i, relation_property) > new)
            new = GProperty(KOV, i, relation_property);
    new++;
    WriteGProperty(KOV, obj1, relation_property, new);
];
```

§**7. Show Various to Various.** The rest of the code for relations has no use except for debugging: it implements the RELATIONS testing command. Speed is unimportant here.

```
[ Relation_ShowVtoV relation sym x obj1 obj2 pr pr2 proutine1 proutine2 vtov_structure;
    vtov_structure = relation-->RR_STORAGE;
    pr = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    proutine1 = vtov_structure-->VTOVS_LEFT_PRINTING_ROUTINE;
    proutine2 = vtov_structure-->VTOVS_RIGHT_PRINTING_ROUTINE;
    if (pr && pr2) {
        objectloop (obj1 provides pr)
        objectloop (obj2 provides pr2) {
                if (sym && obj2 > obj1) continue;
                if (Relation_TestVtoV(obj1, relation, obj2)) {
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ", (The) obj1;
                    if (sym) print "  <=>  "; else print "  >=>  ";
                    print (the) obj2, "^";
                }
        }
        return;
    }
    if (pr && (pr2==0)) {
        objectloop (obj1 provides pr)
        for (obj2=1:obj2<=vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE:obj2++) {
                if (Relation_TestVtoV(obj1, relation, obj2)) {
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ", (The) obj1, "  >=>  ";
                    (proutine2).call(obj2);
                    print "^";
                }
        }
        return;
    }
    if ((pr==0) && (pr2)) {
        for (obj1=1:obj1<=vtov_structure-->2:obj1++)
        objectloop (obj2 provides pr2) {
```

```
                if (Relation_TestVtoV(obj1, relation, obj2)) {
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ";
                    (proutine1).call(obj1);
                    print "  >=>  ", (the) obj2, "^";
                }
            }
            return;
        }
        for (obj1=1:obj1<=vtov_structure-->2:obj1++)
            for (obj2=1:obj2<=vtov_structure-->VTOVS_RIGHT_DOMAIN_SIZE:obj2++)
                if (Relation_TestVtoV(obj1, relation, obj2)) {
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ";
                    (proutine1).call(obj1);
                    print "  >=>  ";
                    (proutine2).call(obj2);
                    print "^";
            }
];
```

§8. Show One to One.

```
[ Relation_ShowOtoO relation sym x relation_property t N obj1 obj2;
    relation_property = relation-->RR_STORAGE;
    t = KindBaseTerm(relation-->RR_KIND, 0); ! Kind of left term
    N = KOVDomainSize(t);
    if (t == OBJECT_TY) {
        objectloop (obj1 provides relation_property) {
            obj2 = obj1.relation_property;
            if (sym && obj2 < obj1) continue;
            if (obj2 == 0) continue;
            if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
            print "  ", (The) obj1;
            if (sym) print "  ==  "; else print "  >=>  ";
            print (the) obj2, "^";
        }
    } else {
        for (obj1=1: obj1<=N: obj1++) {
            obj2 = GProperty(t, obj1, relation_property);
            if (sym && obj2 < obj1) continue;
            if (obj2 == 0) continue;
            if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
            print "  ";
            PrintKindValuePair(t, obj1);
            if (sym) print "  ==  "; else print "  >=>  ";
            PrintKindValuePair(t, obj2);
            print "^";
        }
    }
];
```

§**9. Show Reversed One to One.** There's no such kind of relation as this: but the same code used to show 1-to-1 relations is also used to show various-to-1 relations, since the storage is the same. To show 1-to-various relations, we need a transposed form of the same code in which left and right are exchanged: this is it.

```
[ Relation_RShowOtoO relation sym x relation_property obj1 obj2 t1 t2 N1 N2;
    relation_property = relation-->RR_STORAGE;
    t1 = KindBaseTerm(relation-->RR_KIND, 0); ! Kind of left term
    t2 = KindBaseTerm(relation-->RR_KIND, 1); ! Kind of right term
    if (t2 == OBJECT_TY) {
        if (t1 == OBJECT_TY) {
            objectloop (obj1) {
                objectloop (obj2 provides relation_property) {
                    if (obj2.relation_property ~= obj1) continue;
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ", (The) obj1;
                    print "  >=>  ";
                    print (the) obj2, "^";
                }
            }
        } else {
            N1 = KOVDomainSize(t1);
            for (obj1=1: obj1<=N1: obj1++) {
                objectloop (obj2 provides relation_property) {
                    if (obj2.relation_property ~= obj1) continue;
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  "; PrintKindValuePair(t1, obj1);
                    print "  >=>  ";
                    print (the) obj2, "^";
                }
            }
        }
    } else {
        N2 = KOVDomainSize(t2);
        if (t1 == OBJECT_TY) {
            objectloop (obj1) {
                for (obj2=1: obj2<=N2: obj2++) {
                    if (GProperty(t2, obj2, relation_property) ~= obj1) continue;
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ", (The) obj1;
                    print "  >=>  ";
                    PrintKindValuePair(t2, obj2);
                    print "^";
                }
            }
        } else {
            N1 = KOVDomainSize(t1);
            for (obj1=1: obj1<=N1: obj1++) {
                for (obj2=1: obj2<=N2: obj2++) {
                    if (GProperty(t2, obj2, relation_property) ~= obj1) continue;
                    if (x == 0) { print (string) relation-->RR_DESCRIPTION, ":^"; x=1; }
                    print "  ";
                    PrintKindValuePair(t1, obj1);
```

```
                    print "  >=>  ";
                    PrintKindValuePair(t2, obj2);
                    print "^";
                }
            }
        }
    }
];
```

§10. Show Equivalence.

```
[ RSE_Flip KOV v relation_property x;
    x = GProperty(KOV, v, relation_property); x = -x;
    WriteGProperty(KOV, v, relation_property, x);
];
[ RSE_Set KOV v relation_property;
    if (GProperty(KOV, v, relation_property) < 0) rtrue; rfalse;
];
[ Relation_ShowEquiv relation relation_property obj1 obj2 v c d somegroups t N x;
    print (string) relation-->RR_DESCRIPTION, ":^";
    relation_property = relation-->RR_STORAGE;
    t = KindBaseTerm(relation-->RR_KIND, 0); ! Kind of left term
    N = KOVDomainSize(t);
    if (t == OBJECT_TY) {
        objectloop (obj1 provides relation_property)
            obj1.relation_property = -(obj1.relation_property);
        objectloop (obj1 provides relation_property) {
            if (obj1.relation_property < 0) {
                v = obj1.relation_property; c = 0;
                objectloop (obj2 has workflag2) give obj2 ~workflag2;
                objectloop (obj2 provides relation_property) {
                    if (obj2.relation_property == v) {
                        give obj2 workflag2;
                        obj2.relation_property = -v;
                        c++;
                    }
                }
                if (c>1) {
                    somegroups = true;
                    print "  { ";
                    WriteListOfMarkedObjects(ENGLISH_BIT);
                    print " }^";
                } else obj1.relation_property = v;
            }
        }
        objectloop (obj2 has workflag2) give obj2 ~workflag2;
        c = 0; objectloop (obj1 provides relation_property)
            if (obj1.relation_property < 0) { c++; give obj1 workflag2; }
        if (c == 0) return;
        if (somegroups) print "  and "; else print "  ";
        if (c < 4) { WriteListOfMarkedObjects(ENGLISH_BIT); print " in"; }
        else print c;
        if (c == 1) print " a";
```

```
        print " single-member group";
        if (c > 1) print "s";
        print "^";
        objectloop (obj1 provides relation_property)
            if (obj1.relation_property < 0)
                obj1.relation_property = -(obj1.relation_property);
    } else {
        ! A slower method, since we have less efficient storage:
        for (obj1 = 1: obj1 <= N: obj1++)
            RSE_Flip(t, obj1, relation_property);
        for (obj1 = 1: obj1 <= N: obj1++) {
            if (RSE_Set(t, obj1, relation_property)) {
                v = GProperty(t, obj1, relation_property);
                c = 0;
                for (obj2 = 1: obj2 <= N: obj2++)
                    if (GProperty(t, obj2, relation_property) == v)
                        c++;
                if (c>1) {
                    somegroups = true;
                    print "  {";
                    d = 0;
                    for (obj2 = 1: obj2 <= N: obj2++) {
                        if (GProperty(t, obj2, relation_property) == v) {
                            print " "; PrintKindValuePair(t, obj2);
                            if (d < c-1) print ","; print " ";
                            RSE_Flip(t, obj2, relation_property);
                            d++;
                        }
                    }
                    print "}^";
                } else WriteGProperty(t, obj1, relation_property, v);
            }
        }
        objectloop (obj2 has workflag2) give obj2 ~workflag2;
        c = 0;
        for (obj1 = 1: obj1 <= N: obj1++)
            if (RSE_Set(t, obj1, relation_property)) c++;
        if (c == 0) return;
        if (somegroups) print "  and "; else print "  ";
        if (c == 1) print "a"; else print c;
        print " single-member group";
        if (c > 1) print "s";
        print "^";
        for (obj1 = 1: obj1 <= N: obj1++)
            if (RSE_Set(t, obj1, relation_property))
                RSE_Flip(t, obj1, relation_property);
    }
];
```

§11. Map Route-Finding. The general problem we have to solve here is: given $x, y \in R$, where R is the set of rooms and we write $x \sim y$ if there is a map connection from x to y,

(i) find the smallest m such that there exist $x = r_1 \sim r_2 \sim ... \sim r_m = y \in R$, or determine that no such m exists, and

(ii) find d, the first direction to take from x to lead to r_2, or set $d = 0$ if no such path exists or if $m = 1$ so that $x = y$.

Thus a typical outcome might be either "a shortest path from the Town Square to the Hilltop takes 11 moves, starting by going northeast from the Town Square", or alternatively "there's no path from the Town Square to the Hilltop at all". Note that the length of the shortest path is unambiguous, but that there might be many alternative paths of this minimum length: we deliberately do not specify which path is chosen if so, and the two algorithms used below do not necessarily choose the same one.

Route-finding is not an easy operation in computation terms: the various algorithms available have theoretical running times which are easy (if sobering) to compute, but which are not in practice typical of what will happen, because they are quite sensitive to the map in question. Are all the rooms laid out in a long line? Are there clusters of connected rooms like islands? Are there dense clumps of interconnecting rooms? Are there huge but possibly time-saving loops? And so on. Overhead is also important. We present a choice of two algorithms: the "fast" one has a theoretical running time of $O(n^3)$, where n is the number of rooms, whereas the "slow" one runs in $O(n^2)$, yet in practice the fast one easily outperforms the slow on typical heavy-use cases with large maps.

The other issue is memory usage: we essentially have to strike a bargain between speed and memory overhead. Our "slow" algorithm needs only $O(n)$ storage, whereas our "fast" algorithm needs $O(n^2)$, and this is very significant in the Z-machine where array space is in desperately short supply and where, if $n > 50$ or so, the user is already likely to be fighting for the last few bytes in readable memory.

The user is therefore offered the choice, by selecting the use options "Use fast route-finding" and "Use slow route-finding": and the defaults, if neither option is explicitly set, are fast on Glulx and slow on the Z-machine. If both use options are explicitly set – which might happen due to a disagreement between extensions – "fast" wins.

```
#ifndef FAST_ROUTE_FINDING;
#ifndef SLOW_ROUTE_FINDING;
#ifdef TARGET_GLULX;
Constant FAST_ROUTE_FINDING;
#ifnot;
Constant SLOW_ROUTE_FINDING;
#endif;
#endif;
#endif;
```

§12. Cache Control. We provide code to enable our route-finding algorithms to cache their partial results from one usage to the next (though at present only the "fast" algorithm does this). The difficulty here is that the result of a route search depends on three things, any of which may change:

(a) which subset of rooms we are route-finding through;

(b) which subset of doors we are allowing ourselves to use; and

(c) the current map connections between rooms.

We keep track of (c) by watching for calls to `SignalMapChange()` from the routines in "WorldModel.i6t" which alter the map. (a) and (b), however, require tracking from call to call what the current subset of rooms and doors is. (It is not sufficient to remember the criteria used last time and this time, because circumstances could have changed such that the criteria produce a different outcome. For instance, searching through lighted rooms and using unlocked doors will produce a different result if a door has been locked or unlocked since last time, or if a room has become lighted or not.) We store the set of applicable rooms and doors by enumerating them in the property `room_index` and by the flags in the `DoorRoutingViable` array respectively.

```
Constant NUM_DOORS = {-value:Data::Instances::count(K_door)};
Constant NUM_ROOMS = {-value:Data::Instances::count(K_room)};

Array DoorRoutingViable -> NUM_DOORS+1;

Global map_has_changed = true;
Global last_filter; Global last_use_doors;

[ SignalMapChange; map_has_changed = true; ];

[ MapRouteTo from to filter use_doors count  oy oyi ds;
    if (from == nothing) return nothing;
    if (to == nothing) return nothing;
    if (from == to) return nothing;
    if ((filter) && (filter(from) == 0)) return nothing;
    if ((filter) && (filter(to) == 0)) return nothing;
    if ((last_filter ~= filter) || (last_use_doors ~= use_doors)) map_has_changed = true;
    oyi = 0;
    objectloop (oy has mark_as_room) {
        if ((filter == 0) || (filter(oy))) {
            if (oy.room_index == -1) map_has_changed = true;
            oy.room_index = oyi++;
        } else {
            if (oy.room_index >= 0) map_has_changed = true;
            oy.room_index = -1;
        }
    }
    oyi = 0;
    objectloop (oy ofclass K4_door) {
        ds = false;
        if ((use_doors & 2) ||
            (oy has open) || ((oy has openable) && (oy hasnt locked))) ds = true;
        if (DoorRoutingViable->oyi ~= ds) map_has_changed = true;
        DoorRoutingViable->oyi = ds;
        oyi++;
    }
    if (map_has_changed) {
        #ifdef FAST_ROUTE_FINDING; ComputeFWMatrix(filter, use_doors); #endif;
        map_has_changed = false; last_filter = filter; last_use_doors = use_doors;
    }
    #ifdef FAST_ROUTE_FINDING;
    if (count) return FastCountRouteTo(from, to, filter, use_doors);
    return FastRouteTo(from, to, filter, use_doors);
    #ifnot;
    if (count) return SlowCountRouteTo(from, to, filter, use_doors);
    return SlowRouteTo(from, to, filter, use_doors);
    #endif;
];
```

§**13. Fast Route-Finding.** The following is a form of Floyd's adaptation of Warshall's algorithm for finding the transitive closure of a directed graph.

We need to store a matrix which for each pair of rooms R_i and R_j records a_{ij}, the shortest path length from R_i to R_j or 0 if no path exists, and also d_{ij}, the first direction to take on leaving R_i along a shortest path to R_j, or 0 if no path exists. For the sake of economy we represent the directions as their instance counts (numbered from 0 in order of creation), not as their direction object values, and then store a single word for each pair (i, j): we store $d_{ij} + Da_{ij}$. This restricts us on a signed 16-bit virtual machine, and with the conventional set of $D = 12$ directions, to the range $0 \le a_{ij} \le 5461$, that is, to path lengths of 5461 steps or fewer. A work of IF with 5461 rooms will not fit in the Z-machine anyway: such a work would be on Glulx, which is 32-bit, and where $0 \le a_{ij} \le 357,913,941$.

We begin with $a_{ij} = 0$ for all pairs except where there is a viable map connection between R_i and R_j: for those we set $a_{ij} = 1$ and d_{ij} equal to the direction of that map connection.

Following Floyd and Warshall we test if each known shortest path R_x to R_y can be used to shorten the best known path from R_x to anywhere else: that is, we look for cases where $a_{xy} + a_{yj} < a_{xj}$, since those show that going from R_x to R_j via R_y takes fewer steps than going directly. See for instance Robert Sedgewick, *Algorithms* (1988), chapter 32.

The trouble with the Floyd-Warshall algorithm is not so much that it takes in principle $O(n^3)$ time to construct the matrix: it does, but the coefficient is low, and in the early stages of the outer loop the fact that the vertex degree is at most D and usually much lower helps to reduce the work further. The trouble is that there is no way to compute only the part of the matrix we want: we have to have the entire thing, and that means storing n^2 words of data, by which point we have computed not only the fastest route from R_x to R_y but also the fastest route from anywhere to anywhere else. Even when the original map is sparse, the Floyd-Warshall matrix is not, and it is difficult to store in any very compressed way without greatly increasing the complexity of the code. This is why we cache the results: we might as well, since we had to build the entire memory structure anyway, and it means the time expense is only paid once (or once for every time the state of doors and map connections changes), and the cache is useful for all future routes whatever their endpoints.

```
#ifdef FAST_ROUTE_FINDING;
Array FWMatrix --> NUM_ROOMS*NUM_ROOMS;

[ FastRouteTo from to filter use_doors diri i dir oy;
    if (from == to) return nothing;
    i = (FWMatrix-->(from.room_index*NUM_ROOMS + to.room_index))/No_Directions;
    if (i == 0) return nothing;
    diri = (FWMatrix-->(from.room_index*NUM_ROOMS + to.room_index))%No_Directions;
    i=0; objectloop (dir ofclass K3_direction) {
        if (i == diri) return dir;
        i++;
    }
    return nothing;
];

[ FastCountRouteTo from to filter use_doors  k;
    if (from == to) return 0;
    k = (FWMatrix-->(from.room_index*NUM_ROOMS + to.room_index))/No_Directions;
    if (k == 0) return -1;
    return k;
];

[ ComputeFWMatrix filter use_doors  oy ox oj axy ayj axj dir diri nd row;
    objectloop (oy has mark_as_room) if (oy.room_index >= 0)
        objectloop (ox has mark_as_room) if (ox.room_index >= 0)
            FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = 0;
```

```
    objectloop (oy has mark_as_room) if (oy.room_index >= 0) {
        row = (oy.IK1_Count)*No_Directions;
        for (diri=0: diri<No_Directions: diri++) {
            ox = Map_Storage-->(row+diri);
            if ((ox) && (ox has mark_as_room) && (ox.room_index >= 0)) {
                FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = No_Directions + diri;
                continue;
            }
            if (use_doors && (ox ofclass K4_door) &&
                ((use_doors & 2) || (DoorRoutingViable->(ox.IK4_Count)))) {
                @push location; location = oy;
                ox = ox.door_to();
                @pull location;
                if ((ox) && (ox has mark_as_room) && (ox.room_index >= 0)) {
                    FWMatrix-->(oy.room_index*NUM_ROOMS + ox.room_index) = No_Directions + diri;
                    continue;
                }
            }
        }
    }
    objectloop (oy has mark_as_room) if (oy.room_index >= 0)
        objectloop (ox has mark_as_room) if (ox.room_index >= 0) {
            axy = (FWMatrix-->(ox.room_index*NUM_ROOMS + oy.room_index))/No_Directions;
            if (axy > 0)
                objectloop (oj has mark_as_room) if (oj.room_index >= 0) {
                    ayj = (FWMatrix-->(oy.room_index*NUM_ROOMS + oj.room_index))/No_Directions;
                    if (ayj > 0) {
                        !print "Is it faster to go from ", (name) ox, " to ",
                        !   (name) oj, " via ", (name) oy, "?^";
                        axj = (FWMatrix-->(ox.room_index*NUM_ROOMS + oj.room_index))/
                            No_Directions;
                        if ((axj == 0) || (axy + ayj < axj)) {
                            !print "Yes^";
                            FWMatrix-->(ox.room_index*NUM_ROOMS + oj.room_index) =
                                (axy + ayj)*No_Directions +
                                (FWMatrix-->(ox.room_index*NUM_ROOMS + oy.room_index))%
                                    No_Directions;
                        }
                    }
                }
        }
];
#ENDIF;
```

§**14. Slow Route-Finding.** The alternative algorithm, used when only $O(n)$ memory is available, computes only some of the shortest paths leading to R_y, and is not cached – both because the storage is likely to be reused often by other searches and because there is little gain from doing so, given that a subsequent search with different endpoints will not benefit from the results of this one. On the other hand, to call it "slow" is a little unfair. It is somewhat like Prim's algorithm for finding a minimum spanning tree, rooted at R_y, and grows the tree outward from R_y until either R_x is reached – in which case we stop immediately – or the (directed) component containing R_y has been exhausted – in which case R_x, which must lie outside this, can have no path to R_y. In principle, the running time is $O(dn^2)$, where $d \leq D$ is the maximum vertex degree and n is the number of rooms in the component containing R_y: in practice the degree is often much less than 12, while the algorithm finishes quickly in cases where R_y is relatively isolated and inaccessible or where a shortish route does exist, and those are very common cases in typical usage. There will be circumstances where, because few routes need to be found and because of the shape of the map, the "slow" algorithm will outperform the "fast" one: this is why the user is allowed to control which algorithm is used.

For each room R_z, the property `vector` stores the direction object of the way to go to its parent room in the tree rooted at R_y. Thus if the algorithm succeeds in finding a route from R_x to R_y then we generate the route by starting at R_x and repeatedly going in the `vector` direction from where we currently stand until we reach R_y. Since every room needs a `vector` value, this requires n words of storage. (The `vector` values store only enough of the minimal spanning tree to go upwards through the tree, but that's the only way we need to traverse it.)

The method can be summed up thus:

(a) Begin with every vector blank except that of R_y, the destination.
(b) Repeatedly: For every room in the domain set, try each direction: if this leads to a room whose vector was determined on the last round (*not* on this one, as that may be a suboptimal route), set the vector to point to that room.
(c) Stop as soon as the vector from the origin is set, or when a round happens in which no further vectors are found: in which case, we have completely explored the component of the map from which the destination can be reached, and the origin isn't in it, so we can return "no".

To prove the correctness of this, we show inductively that after round n we have set the `vector` for every room having a shortest path to R_y of length n, and that every `vector` points to a room having a `vector` in the direction of the shortest path from there to R_y.

```
#ifndef FAST_ROUTE_FINDING;
[ SlowRouteTo from to filter use_doors  obj dir in_direction progressed sl through_door;
    if (from == nothing) return nothing;
    if (to == nothing) return nothing;
    if (from == to) return nothing;
    objectloop (obj has mark_as_room) obj.vector = 0;
    to.vector = 1;
    !print "Routing from ", (the) from, " to ", (the) to, "^";
    while (true) {
        progressed = false;
        !print "Pass begins^";
        objectloop (obj has mark_as_room)
            if ((filter == 0) || (filter(obj)))
                if (obj.vector == 0)
                    objectloop (dir ofclass K3_direction) {
                        in_direction = Map_Storage-->((obj.IK1_Count)*No_Directions + dir.IK3_Count);
                        if (in_direction == nothing) continue;
                        !print (the) obj, " > ", (the) dir, " > ", (the) in_direction, "^";
                        if ((in_direction)
                            && (in_direction has mark_as_room)
                            && (in_direction.vector > 0)
```

```
                            && ((filter == 0) || (filter(in_direction)))) {
                            obj.vector = dir | WORD_HIGHBIT;
                            !print "* ", (the) obj, " vector is ", (the) dir, "^";
                            progressed = true;
                            continue;
                        }
                        if (use_doors && (in_direction ofclass K4_door) &&
                            ((use_doors & 2) ||
                            (in_direction has open) ||
                            ((in_direction has openable) && (in_direction hasnt locked)))) {
                            sl = location; location = obj;
                            through_door = in_direction.door_to();
                            location = sl;
                            !print "Through door is ", (the) through_door, "^";
                            if ((through_door)
                                && (through_door has mark_as_room)
                                && (through_door.vector > 0)
                                && ((filter == 0) || (filter(through_door)))) {
                                obj.vector = dir | WORD_HIGHBIT;
                                !print "* ", (the) obj, " vector is ", (the) dir, "^";
                                progressed = true;
                                continue;
                            }
                        }
                    }
        objectloop (obj has mark_as_room) obj.vector = obj.vector &~ WORD_HIGHBIT;
        if (from.vector) return from.vector;
        if (progressed == false) return from.vector;
    }
];
[ SlowCountRouteTo from to filter use_doors obj i;
    if (from == nothing) return -1;
    if (to == nothing) return -1;
    if (from == to) return 0;
    if (from has mark_as_room && to has mark_as_room) {
        obj = MapRouteTo(from,to,filter,use_doors);
        if (obj == nothing) return -1;
        i = 0; obj = from;
        while ((obj ~= to) && (i<NUM_ROOMS)) { i++; obj = MapConnection(obj,obj.vector); }
        return i;
    }
    return -1;
];
#ENDIF;
```

§15. Relation Route-Finding. The general problem we have to solve here is: given $x, y \in D$, where $\sim$ is a relation on a domain set D of objects,

(i) find the smallest n such that there exist $x = r_1 \sim r_2 \sim ... \sim r_n = y \in D$ such that $r_i \sim r_{i+1}$, or determine that no such n exists, and if so

(ii) find a value of r_2 in such a "route" between x and y, or set $r_2 = 0$ if $x = y$ so that $n = 1$.

While in general a relation can have different left and right domains (a relation between doors and rooms, say), route-finding on those relations is unlikely to be very useful, so is discouraged. (In the case of doors and rooms, a route could never be longer than 1 step, since no object is both a door and a room, for instance.) The "fast" V-to-V algorithm requires D to have the same left and right domains; NI compiles the memory caches for V-to-V relations to force any cases with different domains into using the "slow" algorithm.

`MAX_ROUTE_LENGTH` is used simply as a sanity check to prevent hangs if something should go wrong, for instance if the property of a 1-to-V relation has been modified by some third-party code in such a way that it loses its defining invariant.

```
Constant MAX_ROUTE_LENGTH = {-value:Data::Instances::count(K_object)} + 32;

[ RelationRouteTo relation from to count  handler;
    if (count) {
        if (from == nothing) return -1;
        if (to == nothing) return -1;
        if (relation == 0) return -1;
    } else {
        if (from == nothing) return nothing;
        if (to == nothing) return nothing;
        if (relation == 0) return nothing;
    }
    if (from == to) return nothing;
    if (((relation-->RR_PERMISSIONS) & RELS_ROUTE_FIND) == 0) {
        RunTimeProblem(RTP_ROUTELESS);
        return nothing;
    }
    if (relation-->RR_STORAGE == 0) return nothing;
    handler = relation-->RR_HANDLER;
    if (count) return handler(relation, RELS_ROUTE_FIND_COUNT, from, to);
    return handler(relation, RELS_ROUTE_FIND, from, to);
];

[ RelFollowVector rv from to  obj i;
    if (rv == nothing) return -1;
    i = 0; obj = from;
    while ((obj ~= to) && (i<=MAX_ROUTE_LENGTH)) { i++; obj = obj.vector; }
    return i;
];
```

§**16. One To Various Route-Finding.** Here we can immediately determine, given y, the unique y' such that $y' \sim y$, so finding a path from x to y is a matter of following the only path leading to y and seeing if it ever passed through x; thus the running time is $O(n)$, where n is the size of the domain. It would be pointless to cache this.

Note that we can assume here that $x \neq y$, or rather, that `from ~= to`, because that case has already been taken care of.

```
[ OtoVRelRouteTo relation_property from to previous;
    while ((to) && (to provides relation_property) && (to.relation_property)) {
        previous = to.relation_property;
        previous.vector = to;
        if (previous == from) return to;
        to = previous;
    }
    return nothing;
];
```

§**17. Various To One Route-Finding.** This time the simplifying assumption is that, given x, we can immediately determine the unique x' such that $x \sim x'$, so it suffices to follow the only path forwards from x and see if it ever reaches y. The routine is not quite a mirror image of the one above, because both have the same return requirements: we have to ensure that the `vector` properties lay out the path, and also return the next step after x.

```
[ VtoORelRouteTo relation_property from to next  start;
    start = from;
    while ((from) && (from provides relation_property) && (from.relation_property)) {
        next = from.relation_property;
        from.vector = next;
        if (from == to) return start.vector;
        from = next;
    }
    return nothing;
];
```

§**18. Slow Various To Various Route-Finding.** Now there are no simplifying assumptions and the problem is essentially the same as the one solved for route-finding in the map, above. Once again we present two different algorithms: first, a form of Prim's algorithm for minimal spanning trees. Note that, whereas this algorithm was not always so "slow" for the map – because of the fairly low vertex degrees involved, i.e., because most rooms had few connections to other rooms – here the relation might well be almost complete, with almost all the objects related to each other, and then the algorithm will indeed be "slow". So it is likely that the "fast" algorithm will always be better, if the memory can be spared for it.

We use the fast algorithm for a given relation if and only if the NI compiler has allocated the necessary cache memory; the two use options above, for map route-finding, don't control this.

```
[ VtoVRelRouteTo relation from to count obj obj2 related progressed left_ix pr2 i vtov_structure;
    vtov_structure = relation-->RR_STORAGE;
    if (vtov_structure-->VTOVS_CACHE)
        return FastVtoVRelRouteTo(relation, from, to, count);
    left_ix = vtov_structure-->VTOVS_LEFT_INDEX_PROP;
    pr2 = vtov_structure-->VTOVS_RIGHT_INDEX_PROP;
    objectloop (obj ofclass Object && obj provides vector) obj.vector = 0;
    to.vector = 1;
```

```
    while (true) {
        progressed = false;
        objectloop (obj ofclass Object && obj provides left_ix)
            if (obj.vector == 0) {
                objectloop (obj2 ofclass Object && obj2 provides pr2 && obj2.vector > 0) {
                    if (Relation_TestVtoV(obj, relation, obj2)) {
                        obj.vector = obj2 | WORD_HIGHBIT;
                        progressed = true;
                        continue;
                    }
                }
            }
        objectloop (obj ofclass Object && obj provides left_ix)
            obj.vector = obj.vector &~ WORD_HIGHBIT;
        if (from.vector) break;
        if (progressed == false) break;
    }
    if (count) {
        if (from.vector == nothing) return -1;
        i = 0; obj = from;
        while ((obj ~= to) && (i<=MAX_ROUTE_LENGTH)) { i++; obj = obj.vector; }
        return i;
    }
    return from.vector;
];
```

§**19. Fast Various To Various Route-Finding.** Now, as above, a form of the Floyd-Warshall algorithm. The matrix is here stored in the cache of memory pointed to in the V-to-V relation structure. We are unable to combine a_{ij} and d_{ij} into a single cell of memory, so in fact we store two separate matrices: one for a_{ij} (this is `cache` below), the other for n_{ij}, where n_{ij} is the next object in the shortest path from O_i to O_j (this is `cache2` below).

Where $n < 256$ a shortest path must be such that $a_{ij} \leq 255$, so can be stored in a single byte, and we similarly store n_{ij} as the index of the object rather than the object value itself: the index ranges from 0 to $n-1$, so that $0 \leq n_{ij} < 255$ and we can use $n_{ij} = 255$ as a sentinel value meaning "no path". Although the reconversion of n_{ij} back into a valid object value takes a little time, it is only $O(n)$, and of course we know n is relatively small; and in this way we reduce the storage overhead to only n^2 bytes.

Where $n \geq 256$, we resign ourselves to storing two words for each pair (i, j), making $2n^2$ bytes of storage on the Z-machine and $4n^2$ bytes of storage on Glulx, but lookup of a cached result is slightly faster.

```
[ FastVtoVRelRouteTo relation from to count
    domainsize cache cache2 left_ix ox oy oj offset axy axj ayj;
    domainsize = relation-->RR_STORAGE-->2; ! Number of left instances
    left_ix = relation-->RR_STORAGE-->VTOVS_LEFT_INDEX_PROP;
    if ((from provides left_ix) && (to provides left_ix)) {
        if (domainsize < 256) {
            cache = relation-->RR_STORAGE-->VTOVS_CACHE;
            cache2 = cache + domainsize*domainsize;
            if (relation-->RR_STORAGE-->VTOVS_CACHE_BROKEN == true) {
                relation-->RR_STORAGE-->VTOVS_CACHE_BROKEN = false;
                objectloop (oy provides left_ix)
                    objectloop (ox provides left_ix)
                        if (Relation_TestVtoV(oy, relation, ox)) {
```

```
                    offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                    cache->offset = 1;
                    cache2->offset = ox.left_ix;
                } else {
                    offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                    cache->offset = 0;
                    cache2->offset = 255;
                }
            for (oy=0: oy<domainsize: oy++)
                for (ox=0: ox<domainsize: ox++) {
                    axy = cache->(ox*domainsize + oy);
                    if (axy > 0)
                        for (oj=0: oj<domainsize: oj++) {
                            ayj = cache->(oy*domainsize + oj);
                            if (ayj > 0) {
                                offset = ox*domainsize + oj;
                                axj = cache->offset;
                                if ((axj == 0) || (axy + ayj < axj)) {
                                    cache->offset = (axy + ayj);
                                    cache2->offset = cache2->(ox*domainsize + oy);
                                }
                            }
                        }
                }
        }
        if (count) {
            count = cache->((from.left_ix)*domainsize + (to.left_ix));
            if (count == 0) return -1;
            return count;
        }
        oy = cache2->((from.left_ix)*domainsize + (to.left_ix));
        if (oy < 255)
            objectloop (ox provides left_ix)
                if (ox.left_ix == oy) return oy;
        return nothing;
    } else {
        cache = relation-->RR_STORAGE-->VTOVS_CACHE;
        cache2 = cache + WORDSIZE*domainsize*domainsize;
        if (relation-->RR_STORAGE-->VTOVS_CACHE_BROKEN == true) {
            relation-->RR_STORAGE-->VTOVS_CACHE_BROKEN = false;
            objectloop (oy provides left_ix)
                objectloop (ox provides left_ix)
                    if (Relation_TestVtoV(oy, relation, ox)) {
                        offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                        cache-->offset = 1;
                        cache2-->offset = ox;
                    } else {
                        offset = ((oy.left_ix)*domainsize + (ox.left_ix));
                        cache-->offset = 0;
                        cache2-->offset = nothing;
                    }
            for (oy=0: oy<domainsize: oy++)
                for (ox=0: ox<domainsize: ox++) {
```

```
                        axy = cache-->(ox*domainsize + oy);
                        if (axy > 0)
                            for (oj=0: oj<domainsize: oj++) {
                                ayj = cache-->(oy*domainsize + oj);
                                if (ayj > 0) {
                                    offset = ox*domainsize + oj;
                                    axj = cache-->offset;
                                    if ((axj == 0) || (axy + ayj < axj)) {
                                        cache-->offset = (axy + ayj);
                                        cache2-->offset = cache2-->(ox*domainsize + oy);
                                    }
                                }
                            }
                    }
            }
            if (count) {
                count = cache-->((from.left_ix)*domainsize + (to.left_ix));
                if (count == 0) return -1;
                return count;
            }
            return cache2-->((from.left_ix)*domainsize + (to.left_ix));
        }
    }
    if (count) return -1;
    return nothing;
];
```

§**20. Iterating Relations.** The following is provided to make it possible to run an I6 routine on each relation in turn. (Each right-way-round relation, at any rate.)

```
[ IterateRelations callback;
    {-call:Semantics::Relations::relations_command}
];
```