Relation Kind Template B/relkt

Purpose

Code to support the relation kind.

---

§**1. Run-Time Storage.** Inform uses a rich variety of relations, with many different data representations, but we aim to hide that complexity from the user. At run-time, a relation is represented by a block value – that is, by a pointer to a block of data. For a relation conjured up like so:

```
let R be a relation of numbers to texts;
```

this will be a block on the heap, much like a list or an indexed text. For a built-in relation like "containment" or one constructed by a line like

```
Proximity relates various things to various rooms.
```

the block of data will not actually live on the heap but will be elsewhere in memory; it will have the `BLK_FLAG_RESIDENT` bit set, to mark that the heap code isn't allowed to destroy it, but in other respects will look identical.

The data structure contains six words of metadata; for relations on the heap, actual data then sometimes follows on. The low-level routines in "Relations.i6t" access these six words by direct use of `-->`, for speed, and they use the offset constants `RR_NAME` and so on; but we will use the `BlkValueRead` and `BlkValueWrite` routines in this section, which need offsets in the form `RRV_NAME`. (The discrepancy of 4 is to allow for the four-word block header.)

The six words in the data structure hold:

(1) The name of the relation, as a packed string.
(2) A bitmap of its permissions (see below).
(3) A storage word whose meaning depends on its private choice of data representation.
(4) Its strong kind ID, which will always be a pointer to a block reading `RELATION_TY 2 X Y` where `X` and `Y` are in turn strong kinds IDs for the left and right domains of the relation. For instance, the strong kind ID for a relation of numbers to indexed texts is a pointer to the sequence of words `RELATION_TY 2 NUMBER_TY INDEXED_TY`.
(5) A handler routine which tests or asserts the relation (see below).
(6) Descriptive text about the relation, used in SHOWME or run-time problems.

```
Constant RRV_NAME      RR_NAME-4;
Constant RRV_PERMISSIONS RR_PERMISSIONS-4;
Constant RRV_STORAGE RR_STORAGE-4;
Constant RRV_KIND RR_KIND-4;
Constant RRV_HANDLER RR_HANDLER-4;
Constant RRV_DESCRIPTION RR_DESCRIPTION-4;
Constant RRV_USED 6;
Constant RRV_FILLED 7;
Constant RRV_DATA_BASE 8;

! valencies
Constant RRVAL_V_TO_V 0;
Constant RRVAL_V_TO_O RELS_Y_UNIQUE;
```

```
Constant RRVAL_O_TO_V RELS_X_UNIQUE;
Constant RRVAL_O_TO_O RELS_X_UNIQUE+RELS_Y_UNIQUE;
Constant RRVAL_EQUIV RELS_EQUIVALENCE+RELS_SYMMETRIC;
Constant RRVAL_SYM_V_TO_V RELS_SYMMETRIC;
Constant RRVAL_SYM_O_TO_O RELS_SYMMETRIC+RELS_X_UNIQUE+RELS_Y_UNIQUE;

! dictionary entry flags
Constant RRF_USED $0001; ! entry contains a value
Constant RRF_DELETED $0002; ! entry used to contain a value
Constant RRF_SINGLE $0004; ! entry's Y is a value, not a list
Constant RRF_HASX $0010; ! 2-in-1 entry contains a corresponding key
Constant RRF_HASY $0020; ! 2-in-1 entry contains a corresponding value
Constant RRF_ENTKEYX $0040; ! 2-in-1 entry key is left side KOV
Constant RRF_ENTKEYY $0080; ! 2-in-1 entry key is right side KOV

! permission/task constants (those commented out here are generated by I7)
!Constant RELS_SYMMETRIC $8000;
!Constant RELS_EQUIVALENCE $4000;
!Constant RELS_X_UNIQUE $2000;
!Constant RELS_Y_UNIQUE $1000;
!Constant RELS_TEST $0800;
!Constant RELS_ASSERT_TRUE $0400;
!Constant RELS_ASSERT_FALSE $0200;
!Constant RELS_SHOW $0100;
!Constant RELS_ROUTE_FIND $0080;
!Constant RELS_ROUTE_FIND_COUNT $0040;
Constant RELS_COPY $0020;
Constant RELS_DESTROY $0010;
!Constant RELS_LOOKUP_ANY $0008;
!Constant RELS_LOOKUP_ALL_X $0004;
!Constant RELS_LOOKUP_ALL_Y $0002;
!Constant RELS_LIST $0001;

Constant RELS_EMPTY $0003;
Constant RELS_SET_VALENCY $0005;

! RELS_LOOKUP_ANY mode selection constants
Constant RLANY_GET_X 1;
Constant RLANY_GET_Y 2;
Constant RLANY_CAN_GET_X 3;
Constant RLANY_CAN_GET_Y 4;

! RELS_LIST mode selection constant
Constant RLIST_ALL_X 1;
Constant RLIST_ALL_Y 2;
Constant RLIST_ALL_PAIRS 3;
```

§**2. Tunable Parameters.** These constants affect the performance characteristics of the dictionary structures used for relations on the heap. Changing their values may alter the balance between memory consumption and running time.

`RRP_MIN_SIZE`, `RRP_RESIZE_SMALL`, and `RRP_RESIZE_LARGE` must all be powers of two.

```
Constant RRP_MIN_SIZE      8;   ! minimum number of entries (DO NOT CHANGE)
Constant RRP_PERTURB_SHIFT 5;   ! affects the probe sequence
Constant RRP_RESIZE_SMALL  4;   ! resize factor for small tables
Constant RRP_RESIZE_LARGE  2;   ! resize factor for large tables
Constant RRP_LARGE_IS      256; ! how many entries make a table "large"?
Constant RRP_CROWDED_IS    2;   ! when filled entries outnumber unfilled by _ to 1
```

§**3. Abstract Relations.** As the following shows, we can abstractly use a relation – that is, we can use a relation whose identity we know little about – by calling its handler routine `R` in the form `R(rel, task, X, Y)`.

The task should be one of: `RELS_TEST`, `RELS_ASSERT_TRUE`, `RELS_ASSERT_FALSE`, `RELS_SHOW`, `RELS_ROUTE_FIND`, `RELS_ROUTE_FIND_COUNT`, `RELS_COPY`, `RELS_DESTROY`, `RELS_LOOKUP_ANY`, `RELS_LOOKUP_ALL_X`, `RELS_LOOKUP_ALL_Y`, `RELS_LIST`, or `RELS_EMPTY`.

`RELS_SHOW` produces output for the SHOWME testing command. `RELS_ROUTE_FIND` finds the next step in a route from `X` to `Y`, and `RELS_ROUTE_FIND_COUNT` counts the shortest number of steps or returns -1 if no route exists. `RELS_COPY` makes a deep copy of the relation by replacing all block values with duplicates, and `RELS_DESTROY` frees all block values. `RELS_LOOKUP_ANY` finds any one of the `X` values related to a given `Y`, or vice versa, or checks whether such an `X` or `Y` value exists. `RELS_LOOKUP_ALL_X` and `RELS_LOOKUP_ALL_Y` produce a list of all the `X` values related to a given `Y`, or vice versa. `RELS_LIST` produces a list of all `X` values for which a corresponding `Y` exists, or vice versa, or a list of all `(X,Y)` pairs for which `X` is related to `Y`. `RELS_EMPTY` either makes the relation empty (if `X` is 1) or non-empty (if `X` is 0) or makes no change (if `X` is negative), and in any case returns true or false indicating whether the relation is now empty.

Because not every relation supports all of these operations, the "permissions" word in the block is always a bitmap which is a sum of those operations it does offer.

At present, these permissions are not checked as rigorously as they should be (they're correctly set, but not much monitored).

```
[ RelationTest relation task X Y  handler;
    handler = relation-->RR_HANDLER;
    return handler(relation, task, X, Y);
];
```

§**4. Empty Relations.** The absolute minimum relation is one which can only be tested, and which is always empty, that is, where no two values are ever related to each other. The necessary handler routine is `EmptyRelationHandler`.

```
[ EmptyRelationHandler relation task X Y;
    if (task == RELS_EMPTY) rtrue;
    rfalse;
];
```

§**5. Head.** As ever: if there is no heap, there are no relations stored as values.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of heap
```

§**6. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ RELATION_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     arg3 = RELATION_TY_Create(arg2, arg1);
                         return arg3;
        CAST_KOVS:       rfalse;
        DESTROY_KOVS:    return RELATION_TY_Destroy(arg1);
        PRECOPY_KOVS:    rfalse;
        COPY_KOVS:       return RELATION_TY_Copy(arg1, arg2);
        COMPARE_KOVS:    return RELATION_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  rfalse;
        WRITE_FILE_KOVS: rfalse;
        HASH_KOVS:       return arg1;
    }
];
```

§**7. Creation.** Something we have to be careful about is what we mean by copying, or indeed creating, a relation. For example, if we write

```
let Q be a relation of objects to objects;
let Q be the containment relation;
```

...we aren't literally asking for Q to be a duplicate copy of containment, which can then independently evolve – we mean in some sense that Q is a pointer to the one and only containment relation. On the other hand, if we write

```
let Q be a relation of numbers to numbers;
make Q relate 3 to 7;
```

then the second line clearly expects Q to be its own relation, newly created.

We cope with this at creation time. If we're invited to create a copy of an existing relation, we look to see if it is empty – which we detect by its use of the `EmptyRelationHandler` handler. The empty relations are exactly those used as default values for the relation kinds; thus that's what will happen when Q is created. If we find this handler, we intercept and replace it with one of the heap relation handlers, which thus makes the relation a newly constructed data structure which can grow freely from here.

```
[ RELATION_TY_Create kov from rel i skov handler;
    rel = BlkAllocate((RRV_DATA_BASE + 3*RRP_MIN_SIZE)*WORDSIZE,
        RELATION_TY, BLK_FLAG_WORD+BLK_FLAG_MULTIPLE);
    if ((from == 0) && (kov ~= 0)) from = DefaultValueFinder(kov);
    if (from) {
        for (i=0: i<RRV_DATA_BASE: i++) BlkValueWrite(rel, i, BlkValueRead(from, i));
        if (BlkValueRead(from, RRV_HANDLER) == EmptyRelationHandler) {
            handler = ChooseRelationHandler(BlkValueRead(rel, RRV_KIND));
            BlkValueWrite(rel, RRV_NAME, "anonymous relation");
            BlkValueWrite(rel, RRV_PERMISSIONS,
                RELS_TEST+RELS_ASSERT_TRUE+RELS_ASSERT_FALSE+RELS_SHOW);
            BlkValueWrite(rel, RRV_HANDLER, handler);
            BlkValueWrite(rel, RRV_STORAGE, RRP_MIN_SIZE-1);
            BlkValueWrite(rel, RRV_DESCRIPTION, "an anonymous relation");
            BlkValueWrite(rel, RRV_USED, 0);
            BlkValueWrite(rel, RRV_FILLED, 0);
        }
```

```
    } else {
        handler = ChooseRelationHandler(kov);
        BlkValueWrite(rel, RRV_NAME, "anonymous relation");
        BlkValueWrite(rel, RRV_PERMISSIONS,
            RELS_TEST+RELS_ASSERT_TRUE+RELS_ASSERT_FALSE+RELS_SHOW);
        BlkValueWrite(rel, RRV_STORAGE, RRP_MIN_SIZE-1);
        BlkValueWrite(rel, RRV_KIND, kov);
        BlkValueWrite(rel, RRV_HANDLER, handler);
        BlkValueWrite(rel, RRV_DESCRIPTION, "an anonymous relation");
        BlkValueWrite(rel, RRV_USED, 0);
        BlkValueWrite(rel, RRV_FILLED, 0);
    }
    return rel;
];
```

§**8. Destruction.** If the relation stores block values on either side, invoke the handler using a special task value to free the memory associated with them.

```
[ RELATION_TY_Destroy rel  handler;
    handler = BlkValueRead(rel, RRV_HANDLER);
    handler(rel, RELS_DESTROY);
    return rel;
];
```

§**9. Copying.** Same as destruction: invoke the handler using a special value to tell it to perform deep copying.

```
[ RELATION_TY_Copy lto lfrom  handler;
    handler = BlkValueRead(lto, RRV_HANDLER);
    handler(lto, RELS_COPY);
];
```

§**10. Comparison.** It really isn't clear how to define equality for relations, but we follow the doctrine above. What we don't do is to test its actual state – that would be very slow and might be impossible.

```
[ RELATION_TY_Compare rleft rright ind1 ind2;
    ind1 = BlkValueRead(rleft, RRV_HANDLER);
    ind2 = BlkValueRead(rright, RRV_HANDLER);
    if (ind1 ~= ind2)
        return ind1 - ind2;
    return rleft - rright;
];
[ RELATION_TY_Distinguish rleft rright;
    if (RELATION_TY_Compare(rleft, rright) == 0) rfalse;
    rtrue;
];
```

§11. Printing.

```
[ RELATION_TY_Say rel;
    if (rel == 0) print "(null relation)"; ! shouldn't happen
    else print (string) rel-->RR_NAME;
];
```

§12. Naming.

```
[ RELATION_TY_Name rel txt;
    if (rel) {
        BlkValueWrite(rel, RRV_NAME, txt);
        BlkValueWrite(rel, RRV_DESCRIPTION, txt);
    }
];
```

§13. Choose Relation Handler. We implement two different various-to-various handler routines for the sake of efficiency. The choice of handler routines is made based on the kinds of value being related. Each handler also has a corresponding wrapper for symmetric relations.

```
[ ChooseRelationHandler kov sym;
    if (KOVIsBlockValue(KindBaseTerm(kov, 0))) {
        if (sym) return SymHashListRelationHandler;
        return HashListRelationHandler;
    }
    if (sym) return SymDoubleHashSetRelationHandler;
    return DoubleHashSetRelationHandler;
];
```

§14. Valency. "Valency" refers to the number of participants allowed on either side of the relation: various-to-various, one-to-various, various-to-one, or one-to-one. A newly created relation is always various-to-various. We allow the author to change the valency, but only if no entries have been added yet.

```
[ RELATION_TY_SetValency rel val  kov filled cur handler ext;
    filled = BlkValueRead(rel, RRV_FILLED);
    if (filled) { print "*** Illegal valency change ***^"; rfalse; }
    kov = BlkValueRead(rel, RRV_KIND);
    if (val == RRVAL_EQUIV or RRVAL_SYM_V_TO_V or RRVAL_SYM_O_TO_O) {
        if (KindBaseTerm(kov, 0) ~= KindBaseTerm(kov, 1)) {
            print "*** Relation cannot be made symmetric ***^";
            rfalse;
        }
    }
    cur = BlkValueRead(rel, RRV_HANDLER);
    switch (val) {
        RRVAL_V_TO_V: handler = ChooseRelationHandler(kov, false);
        RRVAL_V_TO_O: handler = HashTableRelationHandler;
        RRVAL_O_TO_V: handler = ReversedHashTableRelationHandler;
        RRVAL_O_TO_O: handler = TwoInOneHashTableRelationHandler;
        RRVAL_EQUIV: handler = EquivHashTableRelationHandler;
        RRVAL_SYM_V_TO_V: handler = ChooseRelationHandler(kov, true);
        RRVAL_SYM_O_TO_O: handler = Sym2in1HashTableRelationHandler;
```

```
        default: print "*** Illegal valency value ***^"; rfalse;
    }
    if (cur == handler) rtrue;
    ! adjust size when going to or from 2-in-1
    if (cur == TwoInOneHashTableRelationHandler) {
        ext = BlkValueRead(rel, RRV_STORAGE) + 1;
        BlkValueSetExtent(rel, RRV_DATA_BASE + 3*ext);
    } else if (handler == TwoInOneHashTableRelationHandler) {
        ext = BlkValueRead(rel, RRV_STORAGE) + 1;
        BlkValueSetExtent(rel, RRV_DATA_BASE + 4*ext);
    }
    BlkValueWrite(rel, RRV_HANDLER, handler);
];
[ RELATION_TY_GetValency rel  handler;
    return BlkValueRead(rel, RRV_PERMISSIONS) & VALENCY_MASK;
];
```

§**15. Double Hash Set Relation Handler.** This implements relations which are stored as a double-hashed set. The storage comprises a list of three-word entries (F, X, Y), where F is a flags word. The ordering of the list is determined by a probe sequence which depends on the combined hash values of X and Y.

The "storage" word in the header stores one less than the number of entries in the list; the number of entries in the list is always a power of two, so this will always be a bit mask. The "used" and "filled" words store the number of entries which currently hold a value, and the number of entries which have ever held a value (even if it was since deleted), respectively.

The utility routine `DoubleHashSetLookUp` locates the hash entry for a key/value pair. It returns either the (non-negative) number of the entry where the pair was found, or the (negative) bitwise NOT of the number of the first unused entry where the pair could be inserted. It uses the utility routine `DoubleHashSetEntryMatches` to compare entries to the sought pair.

The utility routine `DoubleHashSetCheckResize` checks whether the dictionary has become too full after inserting a pair, and expands it if so.

```
[ DoubleHashSetRelationHandler rel task X Y sym  kov kx ky at tmp v;
    kov = BlkValueRead(rel, RRV_KIND);
    kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
    if (task == RELS_SET_VALENCY) {
        return RELATION_TY_SetValency(rel, X);
    } else if (task == RELS_DESTROY) {
        ! clear
        kx = KOVIsBlockValue(kx); ky = KOVIsBlockValue(ky);
        if (~~(kx || ky)) return;
        for (at = BlkValueRead(rel, RRV_STORAGE); at >= 0; at--) {
            tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
            if (tmp & RRF_USED) {
                if (kx) BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
                if (ky) BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2));
            }
            at--;
        }
        return;
    } else if (task == RELS_COPY) {
        X = KOVIsBlockValue(kx); Y = KOVIsBlockValue(ky);
```

```
    if (~~(X || Y)) return;
    at = BlkValueRead(rel, RRV_STORAGE);
    while (at >= 0) {
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        if (tmp & RRF_USED) {
            if (X) {
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
                tmp = BlkValueCopy(BlkValueCreate(kx), tmp);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, tmp);
            }
            if (Y) {
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
                tmp = BlkValueCopy(BlkValueCreate(ky), tmp);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, tmp);
            }
        }
        at--;
    }
    return;
} else if (task == RELS_SHOW) {
    print (string) BlkValueRead(rel, RRV_DESCRIPTION), ":^";
    if (sym) {
        kov = KOVComparisonFunction(kx);
        if (~~kov) kov = UnsignedCompare;
    }
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        if (tmp & RRF_USED) {
            X = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
            Y = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
            if (sym && (kov(X, Y) > 0)) continue;
            print "  ";
            PrintKindValuePair(kx, X);
            if (sym) print " <=> "; else print " >=> ";
            PrintKindValuePair(ky, Y);
            print "^";
        }
    }
    return;
} else if (task == RELS_EMPTY) {
    if (BlkValueRead(rel, RRV_USED) == 0) rtrue;
    if (X == 1) {
        DoubleHashSetRelationHandler(rel, RELS_DESTROY);
        for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
            tmp = RRV_DATA_BASE + 3*at;
            BlkValueWrite(rel, tmp, 0);
            BlkValueWrite(rel, tmp + 1, 0);
            BlkValueWrite(rel, tmp + 2, 0);
        }
        BlkValueWrite(rel, RRV_USED, 0);
        BlkValueWrite(rel, RRV_FILLED, 0);
        rtrue;
    }
```

```
    rfalse;
} else if (task == RELS_LOOKUP_ANY) {
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        tmp = RRV_DATA_BASE + 3*at;
        if (BlkValueRead(rel, tmp) & RRF_USED) {
            if (Y == RLANY_GET_X or RLANY_CAN_GET_X) {
                v = BlkValueRead(rel, tmp + 2);
                if (KOVIsBlockValue(ky)) {
                    if (BlkValueCompare(v, X) ~= 0) continue;
                } else {
                    if (v ~= X) continue;
                }
                if (Y == RLANY_CAN_GET_X) rtrue;
                return BlkValueRead(rel, tmp + 1);
            } else {
                v = BlkValueRead(rel, tmp + 1);
                if (KOVIsBlockValue(kx)) {
                    if (BlkValueCompare(v, X) ~= 0) continue;
                } else {
                    if (v ~= X) continue;
                }
                if (Y == RLANY_CAN_GET_Y) rtrue;
                return BlkValueRead(rel, tmp + 2);
            }
        }
    }
    if (Y == RLANY_GET_X or RLANY_GET_Y)
        print "*** Lookup failed: value not found ***^";
    rfalse;
} else if (task == RELS_LOOKUP_ALL_X) {
    if (BlkType(Y) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(Y, 0);
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        tmp = RRV_DATA_BASE + 3*at;
        if (BlkValueRead(rel, tmp) & RRF_USED) {
            v = BlkValueRead(rel, tmp + 2);
            if (KOVIsBlockValue(ky)) {
                if (BlkValueCompare(v, X) ~= 0) continue;
            } else {
                if (v ~= X) continue;
            }
            LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 1));
        }
    }
    return Y;
} else if (task == RELS_LOOKUP_ALL_Y) {
    if (BlkType(Y) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(Y, 0);
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        tmp = RRV_DATA_BASE + 3*at;
        if (BlkValueRead(rel, tmp) & RRF_USED) {
            v = BlkValueRead(rel, tmp + 1);
            if (KOVIsBlockValue(kx)) {
```

```
                if (BlkValueCompare(v, X) ~= 0) continue;
            } else {
                if (v ~= X) continue;
            }
            LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 2));
        }
    }
    return Y;
} else if (task == RELS_LIST) {
    if (X == 0 || BlkType(X) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(X, 0);
    switch (Y) {
        RLIST_ALL_X, RLIST_ALL_Y:
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 3*at;
                if (BlkValueRead(rel, tmp) & RRF_USED) {
                    tmp++;
                    if (Y == RLIST_ALL_Y) tmp++;
                    v = BlkValueRead(rel, tmp);
                    LIST_OF_TY_InsertItem(X, v, false, 0, true);
                }
            }
            return X;
        RLIST_ALL_PAIRS:
            ! LIST_OF_TY_InsertItem will make a deep copy of the item,
            ! so we can reuse a single combination value here
            Y = BlkValueCreate(COMBINATION_TY, 0, kov);
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 3*at;
                if (BlkValueRead(rel, tmp) & RRF_USED) {
                    v = BlkValueRead(rel, tmp + 1);
                    BlkValueWrite(Y, COMBINATION_ITEM_BASE, v);
                    v = BlkValueRead(rel, tmp + 2);
                    BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1, v);
                    LIST_OF_TY_InsertItem(X, Y);
                }
            }
            BlkValueWrite(Y, COMBINATION_ITEM_BASE, 0);
            BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1, 0);
            BlkFree(Y);
            return X;
    }
    rfalse;
}
at = DoubleHashSetLookUp(rel, kx, ky, X, Y);
switch(task) {
    RELS_TEST:
        if (at >= 0) rtrue;
        rfalse;
    RELS_ASSERT_TRUE:
        if (at >= 0) rtrue;
        at = ~at;
        BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
```

```
            if (BlkValueRead(rel, RRV_DATA_BASE + 3*at) == 0)
                BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_USED+RRF_SINGLE);
            if (KOVIsBlockValue(kx)) { X = BlkValueCopy(BlkValueCreate(kx), X); }
            if (KOVIsBlockValue(ky)) { Y = BlkValueCopy(BlkValueCreate(ky), Y); }
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, Y);
            DoubleHashSetCheckResize(rel);
            rtrue;
        RELS_ASSERT_FALSE:
            if (at < 0) rtrue;
            BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) - 1);
            if (KOVIsBlockValue(kx))
                BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
            if (KOVIsBlockValue(ky))
                BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2));
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_DELETED);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, 0);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, 0);
            rtrue;
    }
];
[ DoubleHashSetLookUp rel kx ky X Y  hashv i free mask perturb flags;
    ! calculate a hash value for the pair
    hashv = KOVHashValue(kx, x) + KOVHashValue(ky, y);
    ! look in the first expected slot
    mask = BlkValueRead(rel, RRV_STORAGE);
    i = hashv & mask;
    flags = BlkValueRead(rel, RRV_DATA_BASE + 3*i);
    if (flags == 0) return ~i;
    if (DoubleHashSetEntryMatches(rel, i, kx, ky, X, Y)) return i;
    ! not here, keep looking in sequence
    free = -1;
    if (flags & RRF_DELETED) free = i;
    perturb = hashv;
    hashv = i;
    for (::) {
        hashv = hashv*5 + perturb + 1;
        i = hashv & mask;
        flags = BlkValueRead(rel, RRV_DATA_BASE + 3*i);
        if (flags == 0) {
            if (free >= 0) return ~free;
            return ~i;
        }
        if (DoubleHashSetEntryMatches(rel, i, kx, ky, X, Y))
            return i;
        if ((free < 0) && (flags & RRF_DELETED)) free = i;
        #ifdef TARGET_ZCODE;
        @log_shift perturb (-RRP_PERTURB_SHIFT) -> perturb;
        #ifnot;
        @ushiftr perturb RRP_PERTURB_SHIFT perturb;
        #endif;
    }
```

```
];
[ DoubleHashSetCheckResize rel  filled ext newext temp i at kov kx ky F X Y;
    filled = BlkValueRead(rel, RRV_FILLED);
    ext = BlkValueRead(rel, RRV_STORAGE) + 1;
    if (filled >= (ext - filled) * RRP_CROWDED_IS) {
        ! copy entries to temporary space
        temp = BlkAllocate(ext * (3*WORDSIZE), INDEXED_TEXT_TY, BLK_FLAG_WORD+BLK_FLAG_MULTIPLE);
        for (i=0: i<ext*3: i++)
            BlkValueWrite(temp, i, BlkValueRead(rel, RRV_DATA_BASE+i));
        ! resize and clear our data
        if (ext >= RRP_LARGE_IS) newext = ext * RRP_RESIZE_LARGE;
        else newext = ext * RRP_RESIZE_SMALL;
        BlkValueSetExtent(rel, RRV_DATA_BASE + newext*3);
        BlkValueWrite(rel, RRV_STORAGE, newext - 1);
        BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_USED));
        for (i=0: i<newext*3: i++)
            BlkValueWrite(rel, RRV_DATA_BASE+i, 0);
        ! copy entries back from temporary space
        kov = BlkValueRead(rel, RRV_KIND);
        kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
        for (i=0: i<ext: i++) {
            F = BlkValueRead(temp, 3*i);
            if (F == 0 || (F & RRF_DELETED)) continue;
            X = BlkValueRead(temp, 3*i + 1);
            Y = BlkValueRead(temp, 3*i + 2);
            at = DoubleHashSetLookUp(rel, kx, ky, X, Y);
            if (at >= 0) { print "*** Duplicate entry while resizing ***^"; rfalse; }
            at = ~at;
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at, F);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, Y);
        }
        ! done with temporary space
        BlkFree(temp);
    }
];
[ DoubleHashSetEntryMatches rel at kx ky X Y  cx cy;
    cx = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
    if (KOVIsBlockValue(kx)) {
        if (BlkValueCompare(cx, X) ~= 0) rfalse;
    } else {
        if (cx ~= X) rfalse;
    }
    cy = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
    if (KOVIsBlockValue(ky)) {
        if (BlkValueCompare(cy, Y) ~= 0) rfalse;
    } else {
        if (cy ~= Y) rfalse;
    }
    rtrue;
];
```

§**16. Hash List Relation Handler.** This implements relations which are stored as a hash table mapping keys to either single values or lists of values. The storage comprises a list of three-word entries, either (F, X, Y) or (F, X, L), where F is a flags word distinguishing between the two cases (among other things). In the latter case, L is a pointer to a list (`LIST_OF_TY`) containing the values.

The "storage", "used", and "filled" words have the same meanings as above.

`HashListRelationHandler` is a thin wrapper around `HashCoreRelationHandler`, which is shared with two other handlers below.

```
[ HashListRelationHandler rel task X Y  sym kov kx ky;
    kov = BlkValueRead(rel, RRV_KIND);
    kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
    return HashCoreRelationHandler(rel, task, kx, ky, X, Y, 1);
];
```

§**17. Hash Table Relation Handler.** This is the same as the Hash List Relation Handler above, except that only one value may be stored for each key. This implements various-to-one relations.

```
[ HashTableRelationHandler rel task X Y  kov kx ky;
    kov = BlkValueRead(rel, RRV_KIND);
    kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
    return HashCoreRelationHandler(rel, task, kx, ky, X, Y, 0);
];
```

§**18. Reversed Hash Table Relation Handler.** This is the same as the Hash Table Relation Handler except that the sides are reversed. This implements one-to-various relations.

```
[ ReversedHashTableRelationHandler rel task X Y  kov kx ky;
    kov = BlkValueRead(rel, RRV_KIND);
    kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
    switch (task) {
        RELS_SET_VALENCY:
            return RELATION_TY_SetValency(rel, X);
        RELS_TEST, RELS_ASSERT_TRUE, RELS_ASSERT_FALSE:
            return HashCoreRelationHandler(rel, task, ky, kx, Y, X, 0);
        RELS_LOOKUP_ANY:
            switch (Y) {
                RLANY_GET_X: Y = RLANY_GET_Y;
                RLANY_GET_Y: Y = RLANY_GET_X;
                RLANY_CAN_GET_X: Y = RLANY_CAN_GET_Y;
                RLANY_CAN_GET_Y: Y = RLANY_CAN_GET_X;
            }
        RELS_LOOKUP_ALL_X:
            task = RELS_LOOKUP_ALL_Y;
        RELS_LOOKUP_ALL_Y:
            task = RELS_LOOKUP_ALL_X;
        RELS_SHOW:
        RELS_LIST:
            switch (Y) {
                RLIST_ALL_X: Y = RLIST_ALL_Y;
                RLIST_ALL_Y: Y = RLIST_ALL_X;
            }
    }
```

```
    return HashCoreRelationHandler(rel, task, kx, ky, X, Y, 0);
];
```

§**19. Symmetric Relation Handlers.** These are simple wrappers around the asymmetric handlers defined above. When a pair is inserted or removed, the wrappers insert or remove the reversed pair as well.

`SymDoubleHashSetRelationHandler` and `SymHashListRelationHandler` implement symmetric V-to-V relations. `Sym2in1HashTableRelationHandler` implements symmetric 1-to-1. ("`SymTwoInOneHashTableRelationHandler`" would have exceeded Inform 6's 32-character name limit.)

```
[ SymDoubleHashSetRelationHandler rel task X Y;
    if (task == RELS_ASSERT_TRUE or RELS_ASSERT_FALSE)
        DoubleHashSetRelationHandler(rel, task, Y, X);
    return DoubleHashSetRelationHandler(rel, task, X, Y, 1);
];
[ SymHashListRelationHandler rel task X Y;
    if (task == RELS_ASSERT_TRUE or RELS_ASSERT_FALSE)
        HashListRelationHandler(rel, task, Y, X);
    return HashListRelationHandler(rel, task, X, Y);
];
[ Sym2in1HashTableRelationHandler rel task X Y;
    if (task == RELS_ASSERT_TRUE or RELS_ASSERT_FALSE)
        TwoInOneHashTableRelationHandler(rel, task, Y, X);
    return TwoInOneHashTableRelationHandler(rel, task, X, Y, 1);
];
```

§**20. Hash Core Relation Handler.** This implements the core functionality that is shared between `HashListRelationHandler`, `HashTableRelationHandler`, and `ReversedHashTableRelationHandler`. All three handlers are the same except for whether the left or right side is the "key" and whether or not multiple values may be stored for a single key.

As noted above, the table contains three-word entries, (F, X, Y), where F is a flags word. Only the hash code of X is used. If F includes `RRF_SINGLE`, Y is a single value; otherwise, Y is a list (`LIST_OF_TY`) of values. If `mult` is zero, `RRF_SINGLE` must always be set, allowing only one value per key: a new pair (X, Y') will replace the existing pair (X, Y).

```
[ HashCoreRelationHandler rel task kx ky X Y mult  sym rev at tmp fl;
    if (task == RELS_SET_VALENCY) {
        return RELATION_TY_SetValency(rel, X);
    } else if (task == RELS_DESTROY) {
        ! clear
        kx = KOVIsBlockValue(kx); ky = KOVIsBlockValue(ky);
        if (~~(kx || ky)) return;
        at = BlkValueRead(rel, RRV_STORAGE);
        while (at >= 0) {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
            if (fl & RRF_USED) {
                if (kx) BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
                if (ky || ~~(fl & RRF_SINGLE))
                    BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2));
            }
            at--;
        }
```

```
    return;
} else if (task == RELS_COPY) {
    X = KOVIsBlockValue(kx); Y = KOVIsBlockValue(ky);
    if (~~(X || Y)) return;
    at = BlkValueRead(rel, RRV_STORAGE);
    while (at >= 0) {
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        if (fl & RRF_USED) {
            if (X) {
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
                tmp = BlkValueCopy(BlkValueCreate(kx), tmp);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, tmp);
            }
            if (Y || ~~(fl & RRF_SINGLE)) {
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
                tmp = BlkValueCopy(BlkValueCreate(BlkType(tmp)), tmp);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, tmp);
            }
        }
        at--;
    }
    return;
} else if (task == RELS_SHOW) {
    print (string) BlkValueRead(rel, RRV_DESCRIPTION), ":^";
    ! Z-machine doesn't have the room to let us pass sym/rev as parameters
    switch (RELATION_TY_GetValency(rel)) {
        RRVAL_SYM_V_TO_V:
            sym = 1;
            tmp = KOVComparisonFunction(kx);
            if (~~tmp) tmp = UnsignedCompare;
        RRVAL_O_TO_V:
            rev = 1;
    }
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        if (fl & RRF_USED) {
            X = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
            Y = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
            if (fl & RRF_SINGLE) {
                if (sym && tmp(X, Y) > 0) continue;
                print "  ";
                if (rev) PrintKindValuePair(ky, Y);
                else PrintKindValuePair(kx, X);
                if (sym) print " <=> "; else print " >=> ";
                if (rev) PrintKindValuePair(kx, X);
                else PrintKindValuePair(ky, Y);
                print "^";
            } else {
                for (mult=1: mult<=LIST_OF_TY_GetLength(Y): mult++) {
                    fl = LIST_OF_TY_GetItem(Y, mult);
                    if (sym && tmp(X, fl) > 0) continue;
                    print "  ";
                    if (rev) PrintKindValuePair(ky, fl);
```

```
                    else PrintKindValuePair(kx, X);
                    if (sym) print " <=> "; else print " >=> ";
                    if (rev) PrintKindValuePair(kx, X);
                    else PrintKindValuePair(ky, fl);
                    print "^";
                }
            }
        }
    }
    return;
} else if (task == RELS_EMPTY) {
    if (BlkValueRead(rel, RRV_USED) == 0) rtrue;
    if (X == 1) {
        HashCoreRelationHandler(rel, RELS_DESTROY);
        for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
            tmp = RRV_DATA_BASE + 3*at;
            BlkValueWrite(rel, tmp, 0);
            BlkValueWrite(rel, tmp + 1, 0);
            BlkValueWrite(rel, tmp + 2, 0);
        }
        BlkValueWrite(rel, RRV_USED, 0);
        BlkValueWrite(rel, RRV_FILLED, 0);
        rtrue;
    }
    rfalse;
} else if (task == RELS_LOOKUP_ANY) {
    if (Y == RLANY_GET_Y or RLANY_CAN_GET_Y) {
        at = HashCoreLookUp(rel, kx, X);
        if (at >= 0) {
            if (Y == RLANY_CAN_GET_Y) rtrue;
            tmp = RRV_DATA_BASE + 3*at;
            fl = BlkValueRead(rel, tmp);
            tmp = BlkValueRead(rel, tmp + 2);
            if (fl & RRF_SINGLE) return tmp;
            return LIST_OF_TY_GetItem(tmp, 1);
        }
    } else {
        for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
            tmp = RRV_DATA_BASE + 3*at;
            fl = BlkValueRead(rel, tmp);
            if (fl & RRF_USED) {
                sym = BlkValueRead(rel, tmp + 2);
                if (fl & RRF_SINGLE) {
                    if (KOVIsBlockValue(ky)) {
                        if (BlkValueCompare(X, sym) ~= 0) continue;
                    } else {
                        if (X ~= sym) continue;
                    }
                } else {
                    if (LIST_OF_TY_FindItem(sym, X) == 0) continue;
                }
                if (Y == RLANY_CAN_GET_X) rtrue;
                return BlkValueRead(rel, tmp + 1);
```

```
            }
        }
    }
    if (Y == RLANY_GET_X or RLANY_GET_Y)
        print "*** Lookup failed: value not found ***^";
    rfalse;
} else if (task == RELS_LOOKUP_ALL_X) {
    if (BlkType(Y) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(Y, 0);
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        tmp = RRV_DATA_BASE + 3*at;
        fl = BlkValueRead(rel, tmp);
        if (fl & RRF_USED) {
            sym = BlkValueRead(rel, tmp + 2);
            if (fl & RRF_SINGLE) {
                if (KOVIsBlockValue(ky)) {
                    if (BlkValueCompare(X, sym) ~= 0) continue;
                } else {
                    if (X ~= sym) continue;
                }
            } else {
                if (LIST_OF_TY_FindItem(sym, X) == 0) continue;
            }
            LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 1));
        }
    }
    return Y;
} else if (task == RELS_LOOKUP_ALL_Y) {
    if (BlkType(Y) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(Y, 0);
    at = HashCoreLookUp(rel, kx, X);
    if (at >= 0) {
        tmp = RRV_DATA_BASE + 3*at;
        fl = BlkValueRead(rel, tmp);
        tmp = BlkValueRead(rel, tmp + 2);
        if (fl & RRF_SINGLE)
            LIST_OF_TY_InsertItem(Y, tmp);
        else
            LIST_OF_TY_AppendList(Y, tmp);
    }
    return Y;
} else if (task == RELS_LIST) {
    if (BlkType(X) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(X, 0);
    switch (Y) {
        RLIST_ALL_X:
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 3*at;
                fl = BlkValueRead(rel, tmp);
                if (fl & RRF_USED)
                    LIST_OF_TY_InsertItem(X, BlkValueRead(rel, tmp + 1));
            }
            return X;
```

```
        RLIST_ALL_Y:
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 3*at;
                fl = BlkValueRead(rel, tmp);
                if (fl & RRF_USED) {
                    tmp = BlkValueRead(rel, tmp + 2);
                    if (fl & RRF_SINGLE)
                        LIST_OF_TY_InsertItem(X, tmp, false, 0, true);
                    else
                        LIST_OF_TY_AppendList(X, tmp, false, 0, true);
                }
            }
            return X;
        RLIST_ALL_PAIRS:
            if (RELATION_TY_GetValency(rel) == RRVAL_O_TO_V) rev = 1;
            ! LIST_OF_TY_InsertItem will make a deep copy of the item,
            ! so we can reuse a single combination value here
            Y = BlkValueCreate(COMBINATION_TY, 0, tmp);
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 3*at;
                fl = BlkValueRead(rel, tmp);
                if (fl & RRF_USED) {
                    BlkValueWrite(Y, COMBINATION_ITEM_BASE + rev, BlkValueRead(rel, tmp + 1));
                    tmp = BlkValueRead(rel, tmp + 2);
                    if (fl & RRF_SINGLE) {
                        BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1 - rev, tmp);
                        LIST_OF_TY_InsertItem(X, Y);
                    } else {
                        for (mult = LIST_OF_TY_GetLength(tmp): mult > 0: mult--) {
                            BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1 - rev,
                                LIST_OF_TY_GetItem(tmp, mult));
                            LIST_OF_TY_InsertItem(X, Y);
                        }
                    }
                }
            }
            BlkValueWrite(Y, COMBINATION_ITEM_BASE, 0);
            BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1, 0);
            BlkFree(Y);
            return X;
    }
    rfalse;
}
at = HashCoreLookUp(rel, kx, X);
switch(task) {
    RELS_TEST:
        if (at < 0) rfalse;
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
        if (fl & RRF_SINGLE) {
            if (KOVIsBlockValue(ky)) {
                if (BlkValueCompare(tmp, Y) == 0) rtrue;
            } else {
```

```
            if (tmp == Y) rtrue;
        }
        rfalse;
    } else {
        return LIST_OF_TY_FindItem(tmp, Y);
    }
RELS_ASSERT_TRUE:
    if (at < 0) {
        ! no entry exists for this key, just add one
        at = ~at;
        BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
        if (BlkValueRead(rel, RRV_DATA_BASE + 3*at) == 0)
            BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_USED+RRF_SINGLE);
        if (KOVIsBlockValue(kx)) { X = BlkValueCopy(BlkValueCreate(kx), X); }
        if (KOVIsBlockValue(ky)) { Y = BlkValueCopy(BlkValueCreate(ky), Y); }
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, Y);
        HashCoreCheckResize(rel);
        break;
    }
    ! an entry exists: could be a list or a single value
    fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at); ! flags
    tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2); ! value or list
    if (fl & RRF_SINGLE) {
        ! if Y is the same as the stored key, we have nothing to do
        if (KOVIsBlockValue(ky)) {
            if (BlkValueCompare(tmp, Y) == 0) rtrue;
        } else {
            if (tmp == Y) rtrue;
        }
        ! it's different: either replace it or expand into a list,
        ! depending on the value of mult
        if (mult) {
            fl = LIST_OF_TY_Create(UNKNOWN_TY); ! new list
            BlkValueWrite(fl, LIST_ITEM_KOV_F, ky);
            LIST_OF_TY_SetLength(fl, 2);
            BlkValueWrite(fl, LIST_ITEM_BASE, tmp); ! do not copy
            LIST_OF_TY_PutItem(fl, 2, Y); ! copy if needed
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, fl);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_USED);
        } else {
            if (KOVIsBlockValue(ky)) {
                BlkFree(tmp);
                Y = BlkValueCopy(BlkValueCreate(ky), Y);
            }
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, Y);
        }
    } else {
        ! if Y is present already, do nothing. otherwise add it.
        LIST_OF_TY_InsertItem(tmp, Y, 0, 0, 1);
    }
    rtrue;
```

```
        RELS_ASSERT_FALSE:
            if (at < 0) rtrue;
            ! an entry exists: could be a list or a single value
            fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at); ! flags
            tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2); ! value or list
            if (fl & RRF_SINGLE) {
                ! if the stored key isn't Y, we have nothing to do
                if (KOVIsBlockValue(ky)) {
                    if (BlkValueCompare(tmp, Y) ~= 0) rtrue;
                } else {
                    if (tmp ~= Y) rtrue;
                }
                ! delete the entry
                if (KOVIsBlockValue(ky))
                    BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2));
                .DeleteEntryIgnoringY;
                BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) - 1);
                if (KOVIsBlockValue(kx))
                    BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_DELETED);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, 0);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, 0);
            } else {
                ! remove Y from the list if present
                LIST_OF_TY_RemoveValue(tmp, Y, 1);
                ! if the list is now empty, delete the whole entry
                if (LIST_OF_TY_GetLength(tmp) == 0) {
                    BlkFree(tmp);
                    jump DeleteEntryIgnoringY;
                }
            }
            rtrue;
    }
    rtrue;
];
[ HashCoreLookUp rel kx X  hashv i free mask perturb flags;
!print "[HCLU rel=", rel, " kx=", kx, " X=", X, ": ";
    ! calculate a hash value for the key
    hashv = KOVHashValue(kx, x);
    ! look in the first expected slot
    mask = BlkValueRead(rel, RRV_STORAGE);
    i = hashv & mask;
!print "hv=", hashv, ", trying ", i;
    if (HashCoreEntryMatches(rel, i, kx, X)) {
!print " - found]^";
        return i;
    }
    flags = BlkValueRead(rel, RRV_DATA_BASE + 3*i);
    if (flags == 0) {
!print " - not found]^";
        return ~i;
    }
    ! not here, keep looking in sequence
```

```
    free = -1;
    if (flags & RRF_DELETED) free = i;
    perturb = hashv;
    hashv = i;
    for (::) {
        hashv = hashv*5 + perturb + 1;
        i = hashv & mask;
!print ", ", i;
        flags = BlkValueRead(rel, RRV_DATA_BASE + 3*i);
        if (flags == 0) {
!print " - not found]^";
            if (free >= 0) return ~free;
            return ~i;
        }
        if (HashCoreEntryMatches(rel, i, kx, X)) {
!print " - found]^";
            return i;
        }
        if ((free < 0) && (flags & RRF_DELETED)) free = i;
        #ifdef TARGET_ZCODE;
        @log_shift perturb (-RRP_PERTURB_SHIFT) -> perturb;
        #ifnot;
        @ushiftr perturb RRP_PERTURB_SHIFT perturb;
        #endif;
    }
];
[ HashCoreCheckResize rel  filled ext newext temp i at kov kx F X Y;
    filled = BlkValueRead(rel, RRV_FILLED);
    ext = BlkValueRead(rel, RRV_STORAGE) + 1;
    if (filled >= (ext - filled) * RRP_CROWDED_IS) {
        ! copy entries to temporary space
        temp = BlkAllocate(ext * (3*WORDSIZE), INDEXED_TEXT_TY, BLK_FLAG_WORD+BLK_FLAG_MULTIPLE);
        for (i=0: i<ext*3: i++)
            BlkValueWrite(temp, i, BlkValueRead(rel, RRV_DATA_BASE+i));
        ! resize and clear our data
        if (ext >= RRP_LARGE_IS) newext = ext * RRP_RESIZE_LARGE;
        else newext = ext * RRP_RESIZE_SMALL;
        BlkValueSetExtent(rel, RRV_DATA_BASE + newext*3);
        BlkValueWrite(rel, RRV_STORAGE, newext - 1);
        BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_USED));
        for (i=0: i<newext*3: i++)
            BlkValueWrite(rel, RRV_DATA_BASE+i, 0);
        ! copy entries back from temporary space
        kov = BlkValueRead(rel, RRV_KIND);
        kx = KindBaseTerm(kov, 0);
        for (i=0: i<ext: i++) {
            F = BlkValueRead(temp, 3*i);
            if (F == 0 || (F & RRF_DELETED)) continue;
            X = BlkValueRead(temp, 3*i + 1);
            Y = BlkValueRead(temp, 3*i + 2);
            at = HashCoreLookUp(rel, kx, X);
            if (at >= 0) { print "*** Duplicate entry while resizing ***^"; rfalse; }
            at = ~at;
```

```
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at, F);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, Y);
        }
        ! done with temporary space
        BlkFree(temp);
    }
];
[ HashCoreEntryMatches rel at kx X  cx cy;
    cx = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
    if (KOVIsBlockValue(kx)) {
        if (BlkValueCompare(cx, X) ~= 0) rfalse;
    } else {
        if (cx ~= X) rfalse;
    }
    rtrue;
];
```

§**21. Equivalence Hash Table Relation Handler.** This implements group relations. The table format is identical to that used by `HashCoreRelationHandler`, but we use it differently. Although the relation appears to relate Xs to Xs as far as the game is concerned, the table actually relates Xs to numbers, where each number identifies a group of related items. Any X not listed in the table is implicitly in a single-member group.

When a pair (X, Y) is inserted, one of four cases occurs:

1. Neither X nor Y has a table entry. We search the table to find the next unused group number, then add both X and Y to that group.

2. Both X and Y have existing table entries. If the group numbers differ, we walk through the table and change all occurrences of the higher number to the lower one.

3. X has an existing table entry but Y does not. We add a Y entry using the group number of X.

4. Y has an existing table entry but X does not. We add an X entry using the group number of Y.

When a pair (X, Y) is removed, we first verify that X and Y are in the same group, then delete the table entry for X. This may leave Y in a single-member group, which could be deleted, but detecting that situation would be inefficient, so we keep the Y entry regardless.

This code uses the Hash Core utility functions defined above.

```
[ EquivHashTableRelationHandler rel task X Y  kx at at2 tmp fl i ext;
    kx = KindBaseTerm(BlkValueRead(rel, RRV_KIND), 0);
    if (task == RELS_SET_VALENCY) {
        return RELATION_TY_SetValency(rel, X);
    } else if (task == RELS_DESTROY) {
        ! clear
        if (KOVIsBlockValue(kx)) {
            at = BlkValueRead(rel, RRV_STORAGE);
            while (at >= 0) {
                fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
                if (fl & RRF_USED) {
                    BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
                }
                at--;
            }
        }
```

```
    return;
} else if (task == RELS_COPY) {
    if (KOVIsBlockValue(kx)) {
        at = BlkValueRead(rel, RRV_STORAGE);
        while (at >= 0) {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
            if (fl & RRF_USED) {
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1);
                tmp = BlkValueCopy(BlkValueCreate(kx), tmp);
                BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1);
            }
            at--;
        }
    }
    return;
} else if (task == RELS_SHOW) {
    print (string) BlkValueRead(rel, RRV_DESCRIPTION), ":^";
    ext = BlkValueRead(rel, RRV_STORAGE);
    ! flag all items by negating their group numbers
    for (at=0, X=RRV_DATA_BASE: at<=ext: at++, X=X+3)
        if (BlkValueRead(rel, X) & RRF_USED)
            BlkValueWrite(rel, X + 2, -(BlkValueRead(rel, X + 2)));
    ! display groups, unflagging them as we go
    for (at=0, X=RRV_DATA_BASE, fl=0: at<=ext: at++, X=X+3, fl=0) {
        if (BlkValueRead(rel, X) & RRF_USED) {
            fl = BlkValueRead(rel, X + 2);
            if (fl > 0) continue; ! already visited
            BlkValueWrite(rel, X + 2, -fl); ! unflag it
            ! display the group starting with this member, but only
            ! if there are more members in the group
            tmp = BlkValueRead(rel, X + 1);
            i = 0;
            for (at2=at+1, Y=RRV_DATA_BASE+3*at2: at2<=ext: at2++, Y=Y+3) {
                if (BlkValueRead(rel, Y) & RRF_USED) {
                    if (BlkValueRead(rel, Y + 2) ~= fl) continue;
                    BlkValueWrite(rel, Y + 2, -fl);
                    if (~~i) {
                        ! print the saved first member
                        print "  { ";
                        PrintKindValuePair(kx, tmp);
                        i = 1;
                    }
                    print ", ";
                    PrintKindValuePair(kx, BlkValueRead(rel, Y + 1));
                }
            }
            if (i) print " }^";
        }
    }
    return;
} else if (task == RELS_EMPTY) {
    ! never empty since R(x,x) is always true
    rfalse;
```

```
} else if (task == RELS_LOOKUP_ANY) {
    ! kind of a cheat, but it's faster than searching for a better value to return
    if (Y == RLANY_CAN_GET_X or RLANY_CAN_GET_Y) rtrue;
    return X;
} else if (task == RELS_LOOKUP_ALL_X or RELS_LOOKUP_ALL_Y) {
    if (BlkType(Y) ~= LIST_OF_TY) rfalse;
    LIST_OF_TY_SetLength(Y, 0);
    BlkValueWrite(Y, LIST_ITEM_KOV_F, kx);
    at = HashCoreLookUp(rel, kx, X);
    if (at < 0) {
        LIST_OF_TY_InsertItem(Y, X);
    } else {
        X = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
        for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
            tmp = RRV_DATA_BASE + 3*at;
            fl = BlkValueRead(rel, tmp);
            if (fl & RRF_USED) {
                if (BlkValueRead(rel, tmp + 2) ~= X) continue;
                LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 1));
            }
        }
    }
    return Y;
} else if (task == RELS_LIST) {
    print "*** Domains of equivalence relations cannot be listed ***^";
    return X;
}
at = HashCoreLookUp(rel, kx, X);
at2 = HashCoreLookUp(rel, kx, Y);
switch(task) {
    RELS_TEST:
        if (at < 0) {
            ! X is a loner, but could still be true if X == Y
            if (KOVIsBlockValue(kx)) {
                if (BlkValueCompare(X, Y) == 0) rtrue;
            } else {
                if (X == Y) rtrue;
            }
            rfalse;
        }
        if (at2 < 0) rfalse;
        if (at == at2) rtrue;
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
        if (BlkValueRead(rel, RRV_DATA_BASE + 3*at2 + 2) == tmp) rtrue;
        rfalse;
    RELS_ASSERT_TRUE:
        ! if X and Y are the same, we have nothing to do
        if (KOVIsBlockValue(kx)) {
            if (BlkValueCompare(X, Y) == 0) rtrue;
        } else {
            if (X == Y) rtrue;
        }
        if (at < 0) {
```

```
    if (at2 < 0) {
        ! X and Y both missing: find a new group number and add both entries
        tmp = 0; ! candidate group number
        ext = BlkValueRead(rel, RRV_STORAGE);
        for (i=0: i<=ext: i++) {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 3*i);
            if (fl & RRF_USED) {
                fl = BlkValueRead(rel, RRV_DATA_BASE + 3*i + 2);
                if (fl > tmp) tmp = fl;
            }
        }
        tmp++; ! new group number
        BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 2);
        ! add X entry
        at = ~at;
        if (KOVIsBlockValue(kx)) { X = BlkValueCopy(BlkValueCreate(kx), X); }
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
        if (fl == 0)
            BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_USED+RRF_SINGLE);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, tmp);
        ! add Y entry. at2 might change if X and Y have the same hash code.
        at2 = ~(HashCoreLookUp(rel, kx, Y));
        if (KOVIsBlockValue(kx)) { Y = BlkValueCopy(BlkValueCreate(kx), Y); }
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at2);
        if (fl == 0)
            BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at2, RRF_USED+RRF_SINGLE);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at2 + 1, Y);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at2 + 2, tmp);
        jump CheckResize;
    }
    ! X missing, Y present: add a new X entry
    at = ~at;
    if (KOVIsBlockValue(kx)) { X = BlkValueCopy(BlkValueCreate(kx), X); }
    BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
    fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at);
    if (fl == 0)
        BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
    BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_USED+RRF_SINGLE);
    BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, X);
    tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at2 + 2);
    BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, tmp);
    jump CheckResize;
}
if (at2 < 0) {
    ! X present, Y missing: add a new Y entry
    at2 = ~at2;
    if (KOVIsBlockValue(kx)) { Y = BlkValueCopy(BlkValueCreate(kx), Y); }
    BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
    fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at2);
    if (fl == 0)
```

```
                BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at2, RRF_USED+RRF_SINGLE);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at2 + 1, Y);
            tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
            BlkValueWrite(rel, RRV_DATA_BASE + 3*at2 + 2, tmp);
            jump CheckResize;
        }
        ! X and Y both present: merge higher group into lower group
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2); ! higher group
        fl = BlkValueRead(rel, RRV_DATA_BASE + 3*at2 + 2); ! lower group
        if (tmp < fl) { i = tmp; tmp = fl; fl = i; }
        ext = BlkValueRead(rel, RRV_STORAGE);
        for (at=0: at<=ext: at++) {
            i = RRV_DATA_BASE + 3*at + 2;
            if (BlkValueRead(rel, i) == tmp)
                BlkValueWrite(rel, i, fl);
        }
        .CheckResize;
        HashCoreCheckResize(rel);
        rtrue;
    RELS_ASSERT_FALSE:
        ! if X and Y are already in different groups, we have nothing to do
        if (at < 0 || at2 < 0) rtrue;
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 3*at + 2);
        if (BlkValueRead(rel, RRV_DATA_BASE + 3*at2 + 2) ~= tmp) rtrue;
        ! delete the entry for X
        BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) - 1);
        if (KOVIsBlockValue(kx))
            BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 3*at + 1));
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at, RRF_DELETED);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 1, 0);
        BlkValueWrite(rel, RRV_DATA_BASE + 3*at + 2, 0);
        rtrue;
    }
];
```

§**22. Two-In-One Hash Table Relation Handler.** This implements one-to-one relations, which are stored as a hash table mapping keys to single values and vice versa. To enforce the one-to-one constraint, we need the ability to quickly check whether a value is present. This could be done with two separate hash tables, one mapping X to Y and one the opposite, but in the interest of conserving memory, we use a single table for both.

Each four-word entry (F, E, K, V) consists of a flags word F, an entry key E (which may be a "key" or "value" in the hash table sense), a corresponding key K (when E is used as a value), and a corresponding value V (when E is used as a key). The pair of related values (X, Y) is represented as two table entries: $(F, X, , Y)$ and $(F, Y, X,)$.

To conserve memory when block values are used, we only create one copy of X and/or Y to share between both entries. When adding a key or value which already exists on either side of the relation, the previous copy will be used. Copies are freed when they are no longer used as entry keys.

Each entry's flags word F indicates, in addition to the standard flags `RRF_USED` and `RRF_DELETED`, also whether the entry contains a corresponding key K and/or value V (`RRF_HASX`, `RRF_HASY`) and whether the entry's key is the same kind of value as X or Y (`RRF_ENTKEYX`, `RRF_ENTKEYY`). If both sides of the relation use the same

kind of value, or if both sides are word values, both `RRF_ENTKEYX` and `RRF_ENTKEYY` will be set on every used entry.

Of particular note for this handler is the utility function `TwoInOneDelete`, which clears one half of an entry (given its entry key), and optionally clears the corresponding other half stored in a different entry. That is, given the entries $(F, X, , Y)$ at index `i` and $(F, Y, X,)$ elsewhere, `TwoInOneDelete(rel, i, kx, ky, RRF_ENTKEYX, 1)` will clear both entries and mark them as deleted. If, however, those entries overlap with other pairs – say they're (F, X, A, Y) and (F, Y, X, B) – then the same call to `TwoInOneDelete` will leave us with $(F, X, A,)$ and $(F, Y, , B)$, having cleared the parts corresponding to the pair (X, Y) but not the parts corresponding to the pairs (A, X) and (Y, B), and will not mark either as deleted. (Such overlap is only possible when the domains of X and Y are the same kind of value.)

```
[ TwoInOneHashTableRelationHandler rel task X Y sym  kov kx ky at at2 tmp fl;
    kov = BlkValueRead(rel, RRV_KIND);
    kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
    if (task == RELS_SET_VALENCY) {
        return RELATION_TY_SetValency(rel, X);
    } else if (task == RELS_DESTROY) {
        ! clear
        kx = KOVIsBlockValue(kx); ky = KOVIsBlockValue(ky);
        if (~~(kx || ky)) return;
        at = BlkValueRead(rel, RRV_STORAGE);
        while (at >= 0) {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
            if (fl & RRF_USED)
                if ((kx && (fl & RRF_ENTKEYX)) || (ky && (fl & RRF_ENTKEYY))) {
                    BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1));
                }
            at--;
        }
        return;
    } else if (task == RELS_COPY) {
        X = KOVIsBlockValue(kx); Y = KOVIsBlockValue(ky);
        if (~~(X || Y)) return;
        at = BlkValueRead(rel, RRV_STORAGE);
        while (at >= 0) {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
            if (fl & RRF_USED) {
                if ((X && (fl & RRF_ENTKEYX)) || (Y && (fl & RRF_ENTKEYY))) {
                    ! copy the entry key
                    tmp = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1);
                    if (fl & RRF_ENTKEYX)
                        tmp = BlkValueCopy(BlkValueCreate(kx), tmp);
                    else
                        tmp = BlkValueCopy(BlkValueCreate(ky), tmp);
                    BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 1, tmp);
                    ! update references in X/Y fields pointing here
                    if (fl & RRF_HASX) {
                        at2 = TwoInOneLookUp(rel, kx,
                            BlkValueRead(rel, RRV_DATA_BASE + 4*at + 2),
                            RRF_ENTKEYX);
                        if (at2 >= 0)
                            BlkValueWrite(rel, RRV_DATA_BASE + 4*at2 + 3, tmp);
                    }
```

```
                if (fl & RRF_HASY) {
                    at2 = TwoInOneLookUp(rel, ky,
                        BlkValueRead(rel, RRV_DATA_BASE + 4*at + 3),
                        RRF_ENTKEYY);
                    if (at2 >= 0)
                        BlkValueWrite(rel, RRV_DATA_BASE + 4*at2 + 2, tmp);
                }
            }
        }
        at--;
    }
    return;
} else if (task == RELS_SHOW) {
    print (string) BlkValueRead(rel, RRV_DESCRIPTION), ":^";
    if (sym) {
        kov = KOVComparisonFunction(kx);
        if (~~kov) kov = UnsignedCompare;
    }
    for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
        fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
        if ((fl & (RRF_USED+RRF_ENTKEYX+RRF_HASY)) ==
            (RRF_USED+RRF_ENTKEYX+RRF_HASY)) {
            X = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1);
            Y = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 3);
            if (sym && kov(X, Y) > 0) continue;
            print "  ";
            PrintKindValuePair(kx, X);
            if (sym) print " <=> "; else print " >=> ";
            PrintKindValuePair(ky, Y);
            print "^";
        }
    }
    return;
} else if (task == RELS_EMPTY) {
    if (BlkValueRead(rel, RRV_USED) == 0) rtrue;
    if (X == 1) {
        TwoInOneHashTableRelationHandler(rel, RELS_DESTROY);
        for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
            tmp = RRV_DATA_BASE + 4*at;
            BlkValueWrite(rel, tmp, 0);
            BlkValueWrite(rel, tmp + 1, 0);
            BlkValueWrite(rel, tmp + 2, 0);
            BlkValueWrite(rel, tmp + 3, 0);
        }
        BlkValueWrite(rel, RRV_USED, 0);
        BlkValueWrite(rel, RRV_FILLED, 0);
        rtrue;
    }
    rfalse;
} else if (task == RELS_LOOKUP_ANY) {
    switch (Y) {
        RLANY_GET_X, RLANY_CAN_GET_X:
            at = TwoInOneLookUp(rel, ky, X, RRF_ENTKEYY);
```

```
            if (at >= 0) {
                tmp = RRV_DATA_BASE + 4*at;
                if (BlkValueRead(rel, tmp) & RRF_HASX) {
                    if (Y == RLANY_CAN_GET_X) rtrue;
                    return BlkValueRead(rel, tmp + 2);
                }
            }
        RLANY_GET_Y, RLANY_CAN_GET_Y:
            at = TwoInOneLookUp(rel, kx, X, RRF_ENTKEYX);
            if (at >= 0) {
                tmp = RRV_DATA_BASE + 4*at;
                if (BlkValueRead(rel, tmp) & RRF_HASY) {
                    if (Y == RLANY_CAN_GET_Y) rtrue;
                    return BlkValueRead(rel, tmp + 3);
                }
            }
    }
    if (Y == RLANY_GET_X or RLANY_GET_Y)
        print "*** Lookup failed: value not found ***^";
    rfalse;
} else if (task == RELS_LOOKUP_ALL_X) {
    at = TwoInOneLookUp(rel, ky, X, RRF_ENTKEYY);
    if (at >= 0) {
        tmp = RRV_DATA_BASE + 4*at;
        if (BlkValueRead(rel, tmp) & RRF_HASX)
            LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 2));
    }
    return Y;
} else if (task == RELS_LOOKUP_ALL_Y) {
    at = TwoInOneLookUp(rel, kx, X, RRF_ENTKEYX);
    if (at >= 0) {
        tmp = RRV_DATA_BASE + 4*at;
        if (BlkValueRead(rel, tmp) & RRF_HASY)
            LIST_OF_TY_InsertItem(Y, BlkValueRead(rel, tmp + 3));
    }
    return Y;
} else if (task == RELS_LIST) {
    switch (Y) {
        RLIST_ALL_X:
            fl = RRF_USED+RRF_ENTKEYX+RRF_HASY;
            jump ListEntryKeys;
        RLIST_ALL_Y:
            fl = RRF_USED+RRF_ENTKEYY+RRF_HASX;
            .ListEntryKeys;
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 4*at;
                if ((BlkValueRead(rel, tmp) & fl) == fl)
                    LIST_OF_TY_InsertItem(X, BlkValueRead(rel, tmp + 1), false, 0, true);
            }
        RLIST_ALL_PAIRS:
            tmp = BlkValueRead(X, LIST_ITEM_KOV_F);
            if (KindAtomic(tmp) ~= COMBINATION_TY) rfalse;
            ! LIST_OF_TY_InsertItem will make a deep copy of the item,
```

```
            ! so we can reuse a single combination value here
            Y = BlkValueCreate(COMBINATION_TY, 0, tmp);
            for (at = BlkValueRead(rel, RRV_STORAGE): at >= 0: at--) {
                tmp = RRV_DATA_BASE + 4*at;
                fl = BlkValueRead(rel, tmp);
                if ((fl & (RRF_USED+RRF_ENTKEYX+RRF_HASY)) ==
                    (RRF_USED+RRF_ENTKEYX+RRF_HASY)) {
                    BlkValueWrite(Y, COMBINATION_ITEM_BASE, BlkValueRead(rel, tmp + 1));
                    BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1, BlkValueRead(rel, tmp + 3));
                    LIST_OF_TY_InsertItem(X, Y);
                }
            }
            BlkValueWrite(Y, COMBINATION_ITEM_BASE, 0);
            BlkValueWrite(Y, COMBINATION_ITEM_BASE + 1, 0);
            BlkFree(Y);
            return X;
    }
    return X;
}
at = TwoInOneLookUp(rel, kx, X, RRF_ENTKEYX);
switch(task) {
    RELS_TEST:
        if (at < 0) rfalse;
        fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
        if (~~(fl & RRF_HASY)) rfalse;
        tmp = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 3);
        if (KOVIsBlockValue(ky)) {
            if (BlkValueCompare(tmp, Y) == 0) rtrue;
        } else {
            if (tmp == Y) rtrue;
        }
        rfalse;
    RELS_ASSERT_TRUE:
        if (at < 0) {
            ! create a new forward entry
            at = ~at;
            BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
            fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
            if (fl == 0)
                BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
            fl = RRF_USED+RRF_HASY+RRF_ENTKEYX;
            if (kx == ky || ~~(KOVIsBlockValue(kx) || KOVIsBlockValue(ky)))
                fl = fl + RRF_ENTKEYY;
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at, fl);
            if (KOVIsBlockValue(kx)) { X = BlkValueCopy(BlkValueCreate(kx), X); }
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 1, X);
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 2, 0);
        } else {
            fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
            if (fl & RRF_HASY) {
                ! if the Y we're inserting is already there, we're done
                tmp = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 3);
                if (KOVIsBlockValue(ky)) {
```

```
                if (BlkValueCompare(tmp, Y) == 0) rtrue;
            } else {
                if (tmp == Y) rtrue;
            }
            ! it's different, so delete the reverse entry
            at2 = TwoInOneLookUp(rel, ky, tmp, RRF_ENTKEYY);
            if (at2 >= 0) TwoInOneDelete(rel, at2, kx, ky, RRF_ENTKEYY);
        } else {
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at, fl + RRF_HASY);
        }
        ! use the existing copy of X
        X = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1);
    }
    ! use the existing copy of Y if there is one
    at2 = TwoInOneLookUp(rel, ky, Y, RRF_ENTKEYY);
    if (KOVIsBlockValue(ky)) {
        if (at2 >= 0)
            Y = BlkValueRead(rel, RRV_DATA_BASE + 4*at2 + 1);
        else
            Y = BlkValueCopy(BlkValueCreate(ky), Y);
    }
    BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 3, Y);
    if (at2 >= 0) {
        ! delete existing reverse entry (and its own forward entry)
        TwoInOneDelete(rel, at2, kx, ky, RRF_ENTKEYY, 1);
    } else {
        at2 = ~at2;
    }
    ! create reverse entry
    BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) + 1);
    fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at2);
    if (fl == 0)
        BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_FILLED) + 1);
    fl = fl | (RRF_USED+RRF_HASX+RRF_ENTKEYY);
    if (kx == ky || ~~(KOVIsBlockValue(kx) || KOVIsBlockValue(ky)))
        fl = fl | RRF_ENTKEYX;
    BlkValueWrite(rel, RRV_DATA_BASE + 4*at2, fl);
    BlkValueWrite(rel, RRV_DATA_BASE + 4*at2 + 1, Y);
    BlkValueWrite(rel, RRV_DATA_BASE + 4*at2 + 2, X);
    TwoInOneCheckResize(rel);
    rtrue;
RELS_ASSERT_FALSE:
    ! we only have work to do if the entry exists and has a Y which
    ! matches the Y we're deleting
    if (at < 0) rtrue;
    fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
    if ((fl & RRF_HASY) == 0) rtrue;
    tmp = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 3);
    if (KOVIsBlockValue(ky)) {
        if (BlkValueCompare(tmp, Y) ~= 0) rtrue;
    } else {
        if (tmp ~= Y) rtrue;
    }
```

```
            TwoInOneDelete(rel, at, kx, ky, RRF_ENTKEYX, 1);
            rtrue;
    }
];
[ TwoInOneDelete rel at kx ky ekflag both  fl at2 E i;
!print "[2in1DEL at=", at, " (E=", BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1), ") ekflag=", ekflag
  ...  , " both=", both, "]^";
    fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
    if (ekflag == RRF_ENTKEYX) {
        if (fl & RRF_HASY) {
            i = RRV_DATA_BASE + 4*at + 3;
            if (both) E = BlkValueRead(rel, i);
            BlkValueWrite(rel, i, 0);
            ! delete matching Y<-X entry if needed
            if (both) {
                at2 = TwoInOneLookUp(rel, ky, E, RRF_ENTKEYY);
                if (at2 >= 0) TwoInOneDelete(rel, at2, kx, ky, RRF_ENTKEYY);
                if (at2 == at) fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
            }
            fl = fl & ~RRF_HASY;
        }
    } else {
        if (fl & RRF_HASX) {
            i = RRV_DATA_BASE + 4*at + 2;
            if (both) E = BlkValueRead(rel, i);
            BlkValueWrite(rel, i, 0);
            ! delete matching X->Y entry if needed
            if (both) {
                at2 = TwoInOneLookUp(rel, kx, E, RRF_ENTKEYX);
                if (at2 >= 0) {
                    TwoInOneDelete(rel, at2, kx, ky, RRF_ENTKEYX);
                    if (at2 == at) fl = BlkValueRead(rel, RRV_DATA_BASE + 4*at);
                }
            }
            fl = fl & ~RRF_HASX;
        }
    }
    if ((fl & (RRF_HASX+RRF_HASY)) == 0) {
        ! entry is now empty, mark it deleted
        if (((fl & RRF_ENTKEYX) && KOVIsBlockValue(kx)) ||
            ((ky ~= kx) && (fl & RRF_ENTKEYY) && KOVIsBlockValue(ky))) {
            BlkFree(BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1));
        }
        BlkValueWrite(rel, RRV_DATA_BASE + 4*at, RRF_DELETED);
        BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 1, 0);
        BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 2, 0);
        BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 3, 0);
        BlkValueWrite(rel, RRV_USED, BlkValueRead(rel, RRV_USED) - 1);
    } else {
        BlkValueWrite(rel, RRV_DATA_BASE + 4*at, fl);
    }
];
[ TwoInOneLookUp rel ke E ekflag  hashv i free mask perturb flags;
```

```
!print "[2in1LU rel=", rel, " ke=", ke, " E=", E, " ekf=", ekflag, ": ";
    ! calculate a hash value for the key
    hashv = KOVHashValue(ke, E);
    ! look in the first expected slot
    mask = BlkValueRead(rel, RRV_STORAGE);
    i = hashv & mask;
!print "hv=", hashv, ", trying ", i;
    flags = BlkValueRead(rel, RRV_DATA_BASE + 4*i);
    if (flags == 0) {
!print " - not found]^";
        return ~i;
    }
    if ((flags & ekflag) && TwoInOneEntryMatches(rel, i, ke, E)) {
!print " - found]^";
        return i;
    }
    ! not here, keep looking in sequence
    free = -1;
    if (flags & RRF_DELETED) free = i;
    perturb = hashv;
    hashv = i;
    for (::) {
        hashv = hashv*5 + perturb + 1;
        i = hashv & mask;
!print ", ", i;
        flags = BlkValueRead(rel, RRV_DATA_BASE + 4*i);
        if (flags == 0) {
!print " - not found]^";
            if (free >= 0) return ~free;
            return ~i;
        }
        if ((flags & ekflag) && TwoInOneEntryMatches(rel, i, ke, E)) {
!print " - found]^";
            return i;
        }
        if ((free < 0) && (flags & RRF_DELETED)) free = i;
        #ifdef TARGET_ZCODE;
        @log_shift perturb (-RRP_PERTURB_SHIFT) -> perturb;
        #ifnot;
        @ushiftr perturb RRP_PERTURB_SHIFT perturb;
        #endif;
    }
];
[ TwoInOneCheckResize rel  filled ext newext temp i at kov kx ky F E X Y;
    filled = BlkValueRead(rel, RRV_FILLED);
    ext = BlkValueRead(rel, RRV_STORAGE) + 1;
    if (filled >= (ext - filled) * RRP_CROWDED_IS) {
        ! copy entries to temporary space
        temp = BlkAllocate(ext * (4*WORDSIZE), INDEXED_TEXT_TY, BLK_FLAG_WORD+BLK_FLAG_MULTIPLE);
        for (i=0: i<ext*4: i++)
            BlkValueWrite(temp, i, BlkValueRead(rel, RRV_DATA_BASE+i));
        ! resize and clear our data
        if (ext >= RRP_LARGE_IS) newext = ext * RRP_RESIZE_LARGE;
```

```
        else newext = ext * RRP_RESIZE_SMALL;
        BlkValueSetExtent(rel, RRV_DATA_BASE + newext*4);
        BlkValueWrite(rel, RRV_STORAGE, newext - 1);
        BlkValueWrite(rel, RRV_FILLED, BlkValueRead(rel, RRV_USED));
        for (i=0: i<newext*4: i++)
            BlkValueWrite(rel, RRV_DATA_BASE+i, 0);
        ! copy entries back from temporary space
        kov = BlkValueRead(rel, RRV_KIND);
        kx = KindBaseTerm(kov, 0); ky = KindBaseTerm(kov, 1);
        for (i=0: i<ext: i++) {
            F = BlkValueRead(temp, 4*i);
            if (F == 0 || (F & RRF_DELETED)) continue;
            E = BlkValueRead(temp, 4*i + 1);
            X = BlkValueRead(temp, 4*i + 2);
            Y = BlkValueRead(temp, 4*i + 3);
            if (F & RRF_ENTKEYX) at = TwoInOneLookUp(rel, kx, E, RRF_ENTKEYX);
            else at = TwoInOneLookUp(rel, ky, E, RRF_ENTKEYY);
            if (at >= 0) { print "*** Duplicate entry while resizing ***^"; rfalse; }
            at = ~at;
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at, F);
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 1, E);
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 2, X);
            BlkValueWrite(rel, RRV_DATA_BASE + 4*at + 3, Y);
        }
        ! done with temporary space
        BlkFree(temp);
    }
];
[ TwoInOneEntryMatches rel at ke E  ce;
    ce = BlkValueRead(rel, RRV_DATA_BASE + 4*at + 1);
    if (KOVIsBlockValue(ke)) {
        if (BlkValueCompare(ce, E) ~= 0) rfalse;
    } else {
        if (ce ~= E) rfalse;
    }
    rtrue;
];
```

§23. Foot.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ RELATION_TY_Support t a b c; rfalse; ];
[ RELATION_TY_Say comb; ];
[ RELATION_TY_Name rel txt; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```

§**24. Empty.** This implements the "empty" adjective. We can always check whether a relation is empty. For most relation types, we can cause the relation to become empty by removing all pairs: but this is impossible for equivalence relations, which are never empty, since any X is equivalent to itself. And we can never force a relation to become non-empty, since that would require making up data.

In any case, the implementation is delegated to the relation handler.

```
[ RELATION_TY_Empty rel set  handler;
    handler = rel-->RR_HANDLER;
    return handler(rel, RELS_EMPTY, set);
];
```