ISR2 User's Guide

B. Draper, J. Ross Beveridge
J. Brolio, A. Hanson
R. Heller, L. Williams

COINS TR 90-52

July 1990

This work is supported in part by the Defense Advance Research Projects Agency under contracts DARPA/RADC F30602-87-C-0140, DARPA/Army ETL DACA76-89-C-0017, and by the National Science Foundation under grant DCR-8500332.

ISR2 User's Guide

Bruce A. Draper J. Ross Beveridge John Brolio

Allen R. Hanson Robert Heller Lance R. Williams

June 22, 1990

Contents

1. Introduction

The ISR (Intermediate Symbolic Representation) database management system is tailored to the needs of intermediate-level computer vision. The guiding criteria have been efficiency, ease of use and generality. Given the volume of data we must accomodate and the computational expense of the procedures that manipulate the data, efficiency has been the prime consideration, since that is our most pressing requirement. Because this DBMS must be used by everyone in the laboratory and a researcher's time is even more precious than machine time, ease of use is also important. We have found that, for a research lab, ease of use means providing a minimal complete set of functions at the lowest level of implementation, so that users can combine these functions into a variety of different systems tailored to their own needs. As such user-constructed systems come into general use, they may be included in the database system as function libraries. The justification for generality in a DBMS is simple: if a database operation is needed, and the DBMS does not support it or permit it, then the DBMS will have to be reimplemented or the user will leave the database environment for the experiment. Striving for generality reduces the probability of having to rewrite the DBMS.

The fundamental data structure in most database managements systems (DBMSs) is the vector or table of records. In VISIONS terminology, where the items or objects in the database are called *tokens*, each record in the vector would represent a token (i.e., the properties or attributes of that token).

Further structure has been imposed in the ISR DBMS, so that the vector of records in the ISR is best thought of as a two-dimensional array. Each column in the array represents a token, each row represents a property or attribute of that token, called a *feature*. This representation makes row access as efficient as column access (or more efficient given the manner of its implementation). Thus it is possible to access a set of tokens by the value of one or more features, and to do it very efficiently. In a standard DBMS, this would be accomplished by indexing, i.e., sorting an array of pointers by some feature value and using that pointer array to access the data. In the ISR DBMS this is accomplished dynamically by treating feature values as a row of a 2D array and providing a fast method of selecting a subrange of values. This "dynamic indexing" is somewhat slower than having a permanent index array available, but it is more efficient and economical for most purposes than actually creating and maintaining the permanent index array.

Since the fundamental data structure is the two-dimensional array, the Visions group has enhanced the basic array by (1) making the array element look and feel like a frame slot, while retaining most of the efficiency of an array element, (2) expanding the array "header" or descriptor into a full-fledged frame object, (3) managing storage space allocation and retrieval so that the huge amounts of data used in image understanding can be handled economically, (4) providing tokensubsequences for "associative" access so that items in the

database can be selected very rapidly by the value of one or more fields, and (5) providing varieties of two-dimensional and three-dimensional spatial access and comparison functions.

2. Overview

2.1 Frames and Tokens

The ISR is an in-core database with two types of first-class objects, *frames* and *tokens*. Frames are named objects that are used to store unique collections of data. Each frame has it own set of *features*, which have data values. Thus a user might make two frames, FRED and MARY, but give them unrelated sets of features. As a result, frames are used to store unique groupings of information. Examples familiar to VISIONS group members might include an MBLINES frame that stored the parameters used in running Michael Boldt's algorithm, on the assumption that only one of these will be needed.

Tokens, on the other hand, are used to store data of which there are many instances. Thus, while the parameters for Boldt's algorithm are stored in a frame, the lines extracted by the algorithm will generally be stored as tokens. A token, like a frame, has a set of features defined for it. They differ from frames in two primary respects: 1) tokens are numbered, not named, and 2) tokens come in a set, and all the tokens in a set share the same feature definitions. Automatically naming tokens is a necessity if there are going to be 6000 instances of a type, as with Boldt's lines. Numbering is the simplest automatic naming scheme. By grouping tokens in a set, it is possible to ensure that all the tokens will have the same features.

A *tokensequence* is a set of tokens that share the same features. Every tokensequence is a part of a frame, and every frame has at most one tokensequence. Thus, to stick to the Boldt's algorithm example, the data could be stored as a frame, named MBLINES, with features that describe the parameters, date of computation, etc. The lines would be stored as a sequence of tokens belonging to the MBLINES frame. Each token would have the same set of defined features, for example CONTRAST and LENGTH, but different feature values. Each token also has a token index, the number that distinguishes it from all the other tokens in the set.

2.2 The ISR Tree Structure

The frames and tokens in the ISR are organized into a tree structure[1]. As we said earlier, every token is part of a tokensequence that is identified with some frame. In addition, every frame has a unique parent which is usually another frame (it could be a

[1] Actually, a network, but we'll get to that. Keep reading.

token). When a new frame named FOO is created, its parent is given a feature FOO whose value is the new frame.

When a user logs in to the system, there exists a unique frame called ROOT. It initially has no features or tokens, and is the only frame allowed to have the parent NIL. The first frame the user creates will therefore necessarily have the parent ROOT[2]. The next frame can either be another child of root, or be a child of the first frame. Thus the user can build up an arbitrary tree of frames.

A common organization for frames and tokens is the three level structure familiar to users of the ISR1. In this, the user begins by creating an image frame, for example AMROAD1. This might have the image size and source files as typical features. At the next level, the user might create several frames that are children of AMROAD1, and that correspond to what the ISR1 called tokensets. Examples of these frames might include MBLINES, NAGIN-KOHLER-REGIONS and LINE-REGION-INTERSECTIONS. Each of these frames would have a large tokensequence containing the various lines, regions and intersections.

An alternative organization that might be desirable when working with multiple images would be to start with an AMROAD frame whose features would be image characteristics shared by all the images, e.g. size. It would then have a sequence of tokens corresponding to the individual images. Features for these tokens might include the image's source files. Frames corresponding to various types of data, e.g. MBLINES, etc., could then be made children of the different image tokens as they were calculated.

Of course, the ISR is flexible enough to accomodate almost any organization desired of image data, so long as each frame has at most one tokensequence and every tokensequence belongs to a frame. The examples above are only suggestions, not constraints.

2.3 Features

It should be obvious by now that features, although not fully first class objects, are crucial to the ISR. Features are where data values are stored. The user will also have noticed that there are two slightly different types of features, *frame features* and *token features*. The difference is that a frame feature has only a single value, whereas a token feature has one value for every token in the tokensequence.

Every feature has a name, a value (in the case of a frame feature) or sequence of values (for a token feature), and five additional facets: *datatype, documentation, if-needed, if-getting*, and *if-setting*. Feature names and documentations are strings supplied by the user. The other facets will be discussed below. It is important for the reader to note that although a token feature has a sequence of values, it has only one of every other facet.

[2]The user could start by creating a token of ROOT, but this is not usually done.

Thus a token feature has only one documentation string, one datatype, etc., that applies to every token in the sequence.

2.3.1 Datatypes

The ISR supports eight feature datatype: *BOOLEAN, INTEGER, REAL, STRING, ARRAY. POINTER. HANDLE* and *GRID.* Of these, the first four should be self-explanatory. The ISR supports arrays to the extent that it will store them and regurgitate them, but it will not perform indexing into them. Thus to access an array element, a user must retrieve the array from the ISR and then access the array through the host language. The POINTER datatype is meant as a catch-all; pointer objects are stored as strings and read by the lisp reader. Pointer objects may be inefficient, and should be used only as a matter of last resort.

A handle in the ISR is an abstract "pointer" to another ISR object. Although there are many types of handles depending on exactly what is being pointed at, for most purposes they are identicle. In particular, when declaring a feature, one only has to specify HANDLE. Any type of handle can then be stored as the value. The two most obvious types of handles are frame handles and token handles. Other types of handles will be discussed later, including frame feature handles and token feature handles later in this section.

The grid is a special datatype in the ISR designed to provide efficient access to tokens through their spatial location. A grid is composed of rectangular "cells" that cover the image (or some portion of it). The idea is that each cell will contain tokens that pass through it. A user puts tokens into a grid through rasteriziation functions that determine the spatial extent of a token. These same functions can then be used at a later time to retrieve the set of tokens that lie in some portion of the image. Section 9.1 will explain the uses of grids in more detail.

All ISR datatypes are construed to be the union of the declared type with the set {:UNCALCULATED, :UNDEFINED}. These are special values used to denote that the feature value hasn't been calculated yet or is undefined for this object. For example, a feature whose value is expensive to compute might leave that value as :UNCALCULATED until it was actually needed (lazy evaluation). In contrast, other features cannot be computed, such as the hue of a perfectly white region. In these cases, the value is set to :UNDEFINED.

2.3.2 Feature Values

Frame features are simple in that they have a single value. Token features, on the other hand, have one value for every token in the tokensequence. One way to view this is as a two dimensional array. Each column is a token in the tokensequence, with all of its different feature values. Each row is the sequence of values for one particular feature

across the tokens. Users who adopt this view will find future topics such as token feature handles and the use of '?' in paths easier to understand.

A frame feature handle points at the value of a feature, and is not terribly useful. It is included primarily for completeness. Token feature handles, on the other hand, can be very useful. A token feature handle points at the sequence of values for a given feature, i.e. it points at a row in the two dimensional array mentioned above. When accessing a single feature for many tokens, the token feature handle can be used to improve performance by looking up the feature name only once, getting the token feature handle, and then using it to access the desired token feature values (see PATHS below).

2.3.3 Demons: If-Needed, If-Getting and If-Setting

The three feature facets still to be discussed are lists of demons to be triggered when accessing feature values. The If-Needed facet contains functions that can be used to calculate the value of the slot. If the user asks for a feature value whose value is :UNCALCULATED then the first function in the If-Needed list is invoked to calculate the value. If it returns any value except :UNCALCULATED then that value is returned to the user. If it returns :UNCALCULATED then the next function in the If-Needed list is invoked, and so on. By default, the last function in each feature's If-Needed list is error.

The If-Getting list is a list of functions to be invoked whenever a feature value is accessed. The default If-Getting list is NIL. When a feature value is accessed, the feature value is passed to the first If-Getting demon. The value returned from this function is passed to the second If-Getting demon and so on. The value returned by the last If-Getting function is returned to the user. When the If-Getting list is empty, the feature value itself is returned to the user.

The If-Setting list is like the If-Getting list, except that it is triggered when the user changes a feature value. Then the new value supplied by the user is given to the first If-Setting function. The output of this function is given to the second If-Setting function, and so on. The value returned by the last If-Setting function is stored as the feature value. When the If-Setting list is NIL, which is the default, the value supplied by the user is stored directly

If-setting functions can also be used to block the storage of a new value. If the value of the special variable *set-value-flag* is NIL (the default is T) when the last if-setting function returns, then the value returned by that function is not stored.

2.4 Paths

All the data in the ISR is organized in a tree. A path is how the user tells the system which piece of data he/she wants or where in the tree to store a new value. Every frame, token and feature in the system has a unique *canonical path* specifying its location in the

tree, relative to ROOT. ROOT is not always the most efficient place to begin searching for a piece of data, however. A *path* can therefore start with any handle, and specifies a data location relative to that handle.

2.4.1 Canonical Paths

- The canonical path of ROOT is "".
- The canonical path of a frame is the canonical path of its parent followed by '$' and the frame name. Thus the canonical path of the MBLINES frame in the ISR1-style organization mentioned above is "ROOT$AMROAD1$MBLINES" or simply "AMROAD1$MBLINES" (leading '$'s can be omitted).
- The canonical path of a token is the canonical path of its frame plus a pair of angle brackets containing the token index. Thus the canonical path of line 14 in the example above is "$AMROAD1$MBLINES<14>".
- The canonical path of a frame feature is the canonical path of the frame plus "$feature-name".
- The canonical path of a token feature is slightly confusing since it does not belong to any particular token (remember the 2D array). The canonical path of a token feature therefore uses the wildcard ?, and is the canonical path of the frame plus "¡?¿feature-name". Thus the canonical path of the CONTRAST feature for mblines would be "AMROAD1$MBLINES<?>CONTRAST".
- The canonical path of a feature facet (e.g. the if-needed demons of a feature) is the canonical path of the feature plus "$F_*facet* – *name*". Thus (to continue the example) the if-needed functions of the mblines contrast token feature could be accessed using "AMROAD1$MBLINES<?>CONTRAST$F_IF-NEEDED". In order to avoid ambiguity, the prefix "F_" is reserved for facet access, and no feature names may begin with this prefix.

2.4.2 Specifying Paths

Paths do not have to be written as strings. Lisp symbols can also be used, so that 'amroad1$mblines<14> is also a legitimate path. Paths can also be a list of components, so that '(amroad1 "MBLINES" 14) is also acceptable, as is '(amroad1$mblines 14). When strings are used, the ISR is not case sensitive.

The most important aspects of paths, however, is that they do not have to start at the root. Any handle can serve as the first component in a path. This saves the system from having to search the entire data tree to find a value. It can start at any handle, which

can be much more efficient. Paths which do not start with a handle automatically start from ROOT[3]. THE USE OF HANDLES TO SHORTEN PATHS IS THE FIRST KEY TO EFFICIENT USE OF THE ISR.

The user will also note that while canonical paths are unique, multiple paths may point to the same object. This will be true when handles other than root are used to shorten paths. It is also true when handles are stored in multiple places. A handle can be stored anywhere in the ISR tree structure and used as part of a path.

2.5 Tokensubsequences and Associative Access

Implicit in the discussion of paths above was the idea that tokens are accessed by their index. In fact, it is often more desirable to access tokens according to their feature values, i.e. access them associatively. This is turn requires the ability to represent sets of tokens that are smaller then the tokensequence.

A *tokensubsequence* (TSS) is a set of tokens, all of which belong to the same tokensequence. A TSS is also a type of handle, and can be stored/retrieved like any other ISR object. In fact, a TSS can be used wherever a handle is used, for example at the start of a path. The primary use for TSSs, however, is to reason over sets of objects.

An associative data query implicitly defines a set of tokens, and therefore a TSS. The set of all lines of contrast between 5 and 10, for example, is a set of line tokens. The ISR therefore supplies a set of functions (union, intersection, set-difference) which operate on either TSSs or associative queries. Associative queries can be given either as feature ranges or as predicates. The result is always a TSS.

A *sort* is an ordered TSS. A sort can be used to order a set of tokens by a feature value, for example the set of all high contrast lines ordered by length.

2.6 Advanced Uses of the ISR

2.6.1 Virtual Features

Sometimes there are features that are so inexpensive to compute, that it is more efficient to compute them whenever they are needed than it is to waste memory storing them. Since the ISR does not allocate memory for a token feature until the first value is stored, virtual features are easily implemented by putting the function that calculates the feature on the if-needed list, and adding an if-setting function that blocks the storage of the feature by setting *set-value-flag* to NIL.

A variation of this is a feature that is a simple transformation of another. Assume, for example, that a set of points were defined in terms of (row, col) coordinates, and the

[3]Which is why "AMROAD1$MBLINES" and "ROOT$AMROAD1$MBLINES" are equivalent

user wanted to be able to use (x, y) coordinates as well. X and Y can then be set up as virtual features, where the if-needed function for x gets the value of row, and the if-needed function for y gets the value of (- image-size col). The if-setting functions of X and Y can then be used to set row and col appropriately, as well as blocking the storage of X and Y. This allows X and Y to be treated just like any other value.

A second variation is the creation of composite features. Let us presume we have line segments that are defined by two endpoints. Point features, called POINT1 and POINT2, can be set up that are virtual features built on top of ROW1, COL1, ROW2, and COL2. The if-needed function for POINT1 builds a lisp defstruct with two slots, row and col, and puts the values of row1 and col1 into them. This defstruct is then given to the user as the value of POINT1. The if-setting demons take this defstruct and set row1 and col1 accordingly. In this case, the datatype of POINT1 would be :POINTER, which is not a problem since this value is never stored.

2.6.2 Multiple Worlds

(What follows will not interest users who do not use concurrent processes.) At times, a process may create a token that is not supposed to be "globally visible". For example, a token may be hypothesized by a process on the basis of slim evidence, and until more evidence is found other processes should not "see" that token, i.e. associative queries should not include it, etc. This is accomplished in the ISR by creating a token inside a tokensubset.

Tokens are usually created using the frame handle or a path that specifies a frame handle. The resulting token is then a visible part of the frame. If a token is created using a tokensubset, however, it is not (apparently) added to the frame; it is added only to the tokensubset given it. The new token is then not visible to associative operations unless the tokensusbet in which it was created is used.

It should be noted that this is not a complete multiple worlds facility. Although tokens can be local, features and frames cannot be. Also, any process specifying the correct canonical pathname can always access a token, and creating a token with the same index as a token which is not visible to the creating process is an error (Users of the multiple world facility should use **create-new-token** rather than **create**). Finally, there is no function for globalizing a local token. Instead the user must make a new, global token and copy the contents of the local token into it. This facility has not been expanded because it has rarely been used in practice.

3. Basic system functions

(**isr2:system-status** *&optional (output-stream *standard-output*)*)

Effects: Prints to the output-stream status information on the current state of the ISR. By default the standard output stream is used.

Returns: `nil` .

Error Conditions:

(**isr2:clear-system**)

Effects: By default deletes everything from the root down and frees up all the memory the ISR has allocated. Does some internal checking which will produce warning messages if the internal state was corrupt.

Returns: `nil` if system not corrupted, otherwise non-nil. Optional information regarding the nature of the corruption may therefore be indicated via the return value.

Error Conditions:

4. Frame, token and feature functions

4.1 Paths and handles

(**isr2:frame** *handle*)

Effects: None.

Returns: The frame of HANDLE. If HANDLE is a frame handle (or tokensequence handle – they are identical), this is an identity function since it returns the same handle. If HANDLE is a feature handle, the frame handle of the frame owning the feature's tokensequence is returned. If HANDLE is a token*sub*sequence handle (TSS), then *isr:frame* returns the frame handle. It is very important to use the frame handle when creating or deleting tokens if you want to insure that the action is globally visible (see *isr:create* and *isr:create-new-token* and the section on TSSs.

Error Conditions: HANDLE is not a handle.

(**isr2:parent** *path*)

Effects: None.

Returns: The parent of *the frame implicitly pointed to by path*. For example, *(isr:parent handle)* is equivalent to *(isr:parent (isr:frame handle))*. Any PATH argument is treated as if *isr:frame* were applied to its handle representation and *isr:parent* applied to the resultant handle.

Error Conditions: PATH is not a legitimate path or its "parent" as defined here does not exist.

(**isr2:handle** *path &key (error-p t)*)

Effects: None.

Returns: Returns the handle to the object specified by PATH. For instance, if PATH denotes a frame, then a frame handle is returned. If PATH specifies a specific token, then a token handle is returned. Etc. Since a handle is a legal path, it returns its argument when given a handle. If ERROR-P is nil, returns nil on error.

Error Conditions: ERROR-P is non-nil and PATH does not specify an ISR object of type handle (something that points as a frame, token, feature, TS, TSS, or sort.

(**isr2:handle-p** *object*)

Effects: None.

Returns: Returns T if object is an ISR handle, NIL otherwise.

Error Conditions: None.

(**isr2:handle-type** *handle*)

Effects: None.

Returns: Returns the type of the handle given, as one of the seven following keywords: :frame, :token, :token-subset, :token-sort, :frame-feature, :token-feature, :grid.

Error Conditions: The argument is not a valid ISR handle.

(**isr2:make-copy-of-handle** *handle*)

Effects: None.

Returns: Returns a deep-copy of its argument, i.e. a new handle that points at the same object(s). In the case of a token-subset or token-sort, the resulting sort/subset can have tokens destructively added or removed without affecting the argument handle. *WARNING: the old ISR2 function* **copy-handle** *did not properly copy TSSs or SORTs. Old code should be updated to use make-copy-of-handle.*

Error Conditions: The argument in not an ISR handle .

(**isr2:handle=** *handle1 handle2*)

Effects: None.

Returns: Returns T if handle1 is "equal" to handle2. Equality of frame, token and feature handles is defined in the obvious way, i.e. two handle are equal if they point at the same ISR object. Equality of TSSs is defined as set equality, so that two TSSs are equal if they contain the same tokens. Neither the internal representation of the TSS nor the pick state is considered relevant for this test. The same is true of SORTS. Note also that two

handles must be of the same type to be equal, so that a sort is not equal to a TSS, even if they have the same tokens.

Error Conditions: One or both of the arguments is not an ISR handle.

(**isr2:path?** *path*)

Effects: None.

Returns: Returns T if it is a valid ISR2 path (i.e. it points to an object in memory), nil otherwise

Error Conditions: None.

(**isr2:handle-canonical-path** *handle*)

Effects: None.

Returns: Returns a list of strings, which are the names encountered traversing the tree from root down to the handle given.

Error Conditions: The argument is not a valid ISR handle.

(**isr2:token-index-of** *handle*)

Effects: None.

Returns: The token index of token pointed at by the handle.

Error Conditions: Argument is not a handle of type :token or :token-feature (the only two handle types that point at a specific token, and thus have an index).

4.2 Defining features and creating objects

In the following functions, PATH is a handle (or tokensubsequence), a path symbol or a path description (list) (in Lisp) as described in the section on paths and handles.

(**isr2:define-feature** *path documentation datatype &key if-needed if-getting if-setting*)

Effects: Adds a new feature to the frame or tokensequence as specified in the path and handle. Features will be initialized to UNCALCULATED. The IF-NEEDED, IF-GETTING, IF-SETTING arguments are collectively known as the feature *demons*. The argument for each one is a list of one or more feature-demons (functions) to be run at the appropriate time. (see 6. below.) Note that the last element of the path will be the feature name. DATATYPE is a keyword, one of :integer, :real, :boolean, :array, :pointer, :string, :handle or :grid.

Returns: The feature handle for the new feature if it is a token-feature, otherwise the path.

Error Conditions: The feature is already defined for this object or the path is undefined.

(**isr2:features** *path*)

Effects: None.

Returns: Returns a list of the features names defined for the frame or token specified by PATH. If path specifies a frame, the name of the FRAME FEATURES are returned. If the path specifies a token|sub|sequence or sort, then the names of the FRAME FEATURES of the affiliated frame are returned. If path specifies a token (either an actual token or one specified using - ? -), then the names of the TOKEN FEATURES are returned. The feature names are returned as a list of strings.

Error Conditions: PATH does not specify a frame, token, TSS or sort.

(**isr2:create** *path &key frame-features token-features*)

Effects: Adds a new frame or token to the database as specified in PATH. The last element of PATH will be the name of the new frame or the index of the new token. All frames designated by PATH, except the one being created,must exist at the time of the call. The &key arguments FRAME-FEATURES and TOKEN-FEATURES each take a list of feature-definitions. A feature-definition is of the form: **(name documentation datatype &key if-needed if-getting if-setting)**. (see `isr:define-feature` for more information.)

Returns: The handle for a new frame or the integer index of a new token.

Error Conditions: The object is already defined or the location described by PATH is unreachable.

Example: The following creates a frame with no defined features.

```
(isr:create 'amroad16)
=> <handle to AMROAD16>
(isr:create 'amroad16$mblines)
=> <handle to AMROAD16$MBLINES>
```

The next example creates a token with a specific index. It would be an error if the token already exists.

```
(isr:create '(some-frame-handle 25))
=> <handle to some-frame-handle 25>
```

(**isr2:create-new-token** *path &key values*)

Effects: Selectes any unused token index and creates a new token with that index in the tokenset described by PATH. PATH may be a frame handle or a TSS. If it is a TSS the resultant token will be visible only to that TSS (See section on TSS). VALUES is an initialization list of feature-name and feature-value pairs, e.g., '(length 5.0 contrast 25.5 ...). When the token is created, any features named in the list will be set to the accompanying

value. Any *:if-setting* demons for those features will also be run.

Returns: The handle of the new token.

Error Conditions: The location described by PATH is unreachable.

(**isr2:destroy** *path*)

Effects: Deletes the specified frame or token releasing the memory used to represent it. If a frame is deleted, everything directly below it on the tree will be deleted.

Note for users of the multiple worlds facilty: if PATH is of the form (TSS index), the token will be deleted only in the local *context* of the TSS. It will still exist in the tokensequence (See TSS section).

Returns: The path-symbol for the deleted object.

Error Conditions: The object is undefined.

(**isr2:copy-definition** *source-path destination-path &optional (clobber-p* **nil**))

Effects: If SOURCE-PATH and DESTINATION-PATH describe frames, then copy-definition creates an empty frame located at DESTINATION-PATH, whose feature descriptions (including tokensequence type descriptions) are identical to those of the SOURCE-PATH frame. The result is a frame with identical structure at a different location in the hierarchy. If *CLOBBER-P* is non-nil, then if a destination tokensequence already exists, it is first deleted.

If SOURCE-PATH describes a tokenset (e.g., AMROAD<>), and DESTINATION-PATH a frame, then the destination frame's tokensequence will acquire feature-slots and features identical to those of the source token.

No other combinations are allowed.

Returns: Frame handle of new frame or of owner of new tokensequence

Error Conditions: The *source* does not specify a valid object. The destination object already exists (frame) or is already defined (tokensequence) and CLOBBER-P is NIL.

(**isr2:move** *source-path destination-path &optional (clobber-p* **nil**))

Effects: If SOURCE-PATH and DESTINATION-PATH describe frame locations, then *move* moves the source frame to the destination location. If CLOBBER-P is non-nil, then if a destination frame already exists, it is first deleted (including tokensequence and subtrees). If SOURCE-PATH describes a tokenset (e.g., AMROAD<>) and DESTINATION-PATH a frame, then the tokensequence represented by the source token will be moved to the destination frame. The tokensequence tokens will be deleted from the source frame, although the tokensequence definitions will not be deleted. If *CLOBBER-P* is non-nil, then if a destination tokensequence already exists, it is first deleted. No other combinations are allowed.

Returns: Frame handle of new frame or of new owner of tokensequence

Error Conditions: The *source-path* does not specify a valid object. The destination object already exists (frame) or is already filled (tokensequence) and clobber-p is nil.

(**isr2:rename** *path newname*)

Effects: Changes the name of the object from oldname to newname. This only changes the last element of the pathname to this object. Note, if renaming a frame this changes both the feature name (in the parent frame) and the frame name itself.

Returns: newname

Error Conditions: An object newname is already defined at this node. Tokens, TSSs and Sorts may not be renamed.

(**isr2:describe-isr-object** *path &key (stream *standard-output*) (verbose t)*)

Effects: Prints a description of an object to the specified stream. If STREAM is supplied and is `NIL` , then no printing is done. If verbose is non-nil, then feature values and feature lists are printed.

Returns: `nil` .

Error Conditions: The object does not exist in the ISR.

(**isr2:datatypep** *path type*)

Effects: None.

Returns: Return t if the object exists and is of type *type*, otherwise nil.

Error Conditions: The object is undefined in the ISR.

(**isr2:datatype-of** *path*)

Effects: None.

Returns: Return the object type if the object exists, otherwise nil. Note that this is equivalent to doing (isr2:value (list path '$F_DATATYPE)).

Error Conditions: The object is undefined in the ISR.

(**isr2:compress** *path*)

Effects: Optimizes the storage space for a tokensubsequence. If there have been a lot of tokens deleted, compress may reduce the amount of wasted space. The effect is similar to what would happen if the sparse tokenset were written to a file and then read in with its handle specified as a pointer. Because compression renumbers the tokens in a tokenset, old token handles and token indices are invalidated by this function.

Returns: nil.

Error Conditions: None.

4.3 Feature access

(**isr2:value** *path &key (if-undefined :error)*)

Effects: If the value is defined. The only side effect is the invocation of any *:if-getting* demons. If the value is **uncalculated** then the *:if-needed* demon is invoked, followed by any *:if-setting* demons (to store the new value) and the *:if-getting* demons. The caller can use the IF-UNDEFINED keyword argument to specify a value to return if the feature is **undefined** . The default effect is a fatal error, but the user may specify any return value by passing it in as the value of :IF-UNDEFINED.

Returns: The value of the location accessed by path (or the value returned from the feature-demons). This will be a handle if the value accessed by PATH is a token, frame or TSS.

Error Conditions: The object or feature is not defined or PATH is invalid. If the feature value is UNDEFINED, then it is an error unless the IF-UNDEFINED keyword argument is specified.

To set a value, CommonLisp's setf macro is used. For example, if foo is a frame, and bar a feature of that frame, then (value '(,foo bar)) will access the feature value, and (setf (value '(,foo bar)) 1) will set its value to 1. Setting a value will cause any :if-setting deamons to be fired. To set the value of a feature to be undefined or uncalculated, set the value to one of the keywords :undefined or :uncalculated.

It is up to the user to insure that handles are valid. They will only be invalidated by deleting tokens or frames and using handles that access such deleted items. Most attempts to access a deleted item will be result in an error. However, it a token is deleted and then a new token with the same index is created, handles to the original token may inadvertently be used to access the new token. Users implementing algorithms that do a lot of replacement of this type must take care not to access deleted items.

5. Examples

Create a frame (at the root of the hierarchy) with some features.

```
(setf *fr*
   (isr:create 'amroad
       '((isr_directory "Amroad tokensets live here" :string
            :if-needed query-user)
         (plane_directory "Planes live here" :string
            :if-needed query-user))))
=> <$ AMROAD handle $>
```

```
(setf (isr:value (list  *fr* 'documentation)) "Amherst road images")
=> "Amherst road images"
```

(Documentation is a feature on all frames.) Now we can add a feature:

```
(isr2:define-feature (list *fr* 'year) "When pictures were taken" :integer)
=>YEAR
```

Notice that no feature demons were declared. The tokens in the AMROAD frame will be Amherst road images. Typical features for this tokensequence:

```
(isr2:define-feature (list *fr* "<?>" 'red) "Red plane." :handle)
=>RED

(isr:define-feature (list *fr* "<?>" 'intensity) "" :handle)
=> INTENSITY
```

These will be "plane" frames – they will give the location of the plane and any other necessary information. Note, we have this far not created individual frames

```
(isr2:define-feature (list *fr* "<?>" 'segm3d) "Regions." :handle)
=> SEGM3D
(isr2:define-feature (list *fr* "<?>" 'mblines) "Michael's lines." :handle)
=> MBLINES

(setf *am1* (isr:create (list *fr* 1)))
=> < handle to AMROAD<1> >

(setf lines
   (isr2:restore "amlines$dir:am1mblines"
                 (list *am1* 'mblines)))

=> < handle to AMROAD<1>MBLINES >

(isr2:value (list lines 18 'length))
=> 5.93768

(isr2:value (list lines 22 'spo_pairs))
=> < handle for tokensubsequence of ...spo_pair >
```

6. Feature Demons

As discussed in Section 2.3.3, demons are lisp functions attached to ISR features that are triggered when the feature is accessed, changed, or in the special case of an uncalculated feature being accessed. Demons are stored as lists of symbols in the if-needed, if-setting and if-getting facets of a feature. As with all other feature facets, demons can be accessed and set using value . In addition, a special function, **add-feature-function** , is provided for adding new demons to a feature.

6.1 If-needed Demons

If-needed demons are invoked whenever the user requests the value of a feature whose current value is :uncalculated. The if-needed demons compute the correct value, store it in the database (unless blocked by an if-setting demon; see below) and return it to the user as if the value had never been :uncalculated. The primary use of if-needed demons is to implement "lazy evaluation", where the value of a feature is not calculated until it is needed. Lazy evaluation thus avoids the expense of calculating features that are never used.

If-needed demons are called with the following arguments: feature-name frame-handle token-handle. When a frame feature is being calculated, the last argument is NIL. The if-needed demon calculates a value for the feature and returns that value. If, for some reason, the demon cannot calculate a value, then it must return :UNCALCULATED. The ISR will then call the next demon in the if-needed sequence. If there are no more demons and the feature value is still :UNCALCULATED, then the ISR will cause an error. If any demon returns any value except :UNCALCULATED, the ISR will setf that value in the slot (activating the if-setting demons) and no more if-needed demons will be called. If the user does not want the value returned by the if-needed demon to be stored, he/she must provide an if-setting demon that blocks it.

An example if-needed demon that queries the user for the value looks like:

```
(defun query-if-needed (feature-name frame-handle token-handle)
   (declare (ignore frame-handle))
   (format *standard-output* "Please enter a value for the~
       feature ~a in token ~a"
       feature-name (token-name token-handle)))
=> QUERY-IF-NEEDED

(add-feature-function (some-frame-handle '<?> 'foo$f_if-needed)
   'query-if-needed :end)
```

6.2 If-setting Demons

The *if-setting* demons are called before the value is written into its slot in the ISR. The demon is called with the following arguments: original-value new-value feature-name frame-handle token-handle. When a frame feature is being computed, the last argument is NIL. The value returned by the if-setting demon should be of the appropriate datatype for the feature being set.

When multiple if-setting demons are present, the first demon if invoked with the original feature value as the first argument and the new value (usually from the user) as the second. The second if-setting demon is then invoked with the original value as the first argument, and the value returned by the previous demon as the new value, and then the third, and so on. The value returned by the last if-setting demon is stored as the new feature value.

It is possible that the user may not want to store the result of a calculation in the slot after it is calculated (for example, the slot could be part of a constraint system, monitoring a relation between other ISR objects). For this purpose, there is a special variable `isr2:*set-value-flag*` which is bound to `t` . If an *if-setting* demon decides that the value should not be stored, the demon should declare `isr2:*set-value-flag*` `special` and change its value to `nil` . If `isr2:*set-value-flag*` is non-nil then the feature value is stored in its proper place in the ISR. If `isr2:*set-value-flag*` is `nil` , then the feature value is not stored in the ISR. If-setting functions can check the current value of `isr2:*set-value-flag*` as they would any special variable.

6.3 If-getting Demons

The *if-getting* function is called after the value is read from its slot in the ISR. The if-getting function is called with the following arguments: value feature-name frame-handle token-handle. The *:if-getting* demon *must* return a value and that is the value returned to the user.

(isr2:add-feature-function *path function-name &optional (location :last)*)

Effects: Adds a feature-demon to the front or end of the sequences of demons for PATH. Note that the last part of path will be `$f_if-needed` or `$f_if-getting` or `$f_if-setting` . LOCATION must be either `:first` or `:last` . If LOCATION is `:first` , then this function will be executed before all other demons in the sequence. Note: the sequence of demons can be gotten, as a list, by asking for the `value` of `$f_if-needed` or `$f_if-getting` or `$f_if-setting` . The user can then modify this list and use `set-value` to install the modified list as the new value for that demon type. This is the only way to remove a feature demon.

Returns: The new list for that demon type.

Error Conditions: The feature does not exist.

6.4 An example of Virtual Features

If-setting demons can be combined with if-needed demons to produce a "virtual feature". A virtual feature is one that is never stored, and therefore does not use any memory. Instead, the value of a virtual feature is calculated every time it is needed.

To give an example of a virtual feature, assume that the user has a tokenset of (2D) point tokens, for which the token features "ROW" and "COL" are defined. ("ROW" and "COL" give the location of the point token in the traditional (row, col) coordinates). Suppose, however, that the user has an application in which they need to reference points in an (x,y) coordinate system in which the center of the image is the origin. Then the user can set up two virtual features, x and y, which can be accessed and set like any other feature. Internally, however, x and y are never stored, and setting the value of x or y results in changing the value of row or col.

We begin by declaring the new features. The reader will note that virtual features are defined like any other feature, and that their unique behavior is the result of demons.

```
(isr2:define-feature 'point<?>x
                     "Virtual feature for computing x from col."
                     :real)
(isr2:define-feature 'point<?>y
                     "Virtual feature for computing y from row."
                     :real)
```

The next step is to write demons that will determine how the feature will behave. There will be three of them for each feature. First we write if-needed demons that will calculate the x,y value from the row, col coordinates:

```
(defun x-from-col (feature-name frame-handle token-handle)
  (declare (ignore feature-name frame-handle))
  (- (isr2:value '(,token-handle col)) (/ *image-size* 2)))

(defun y-from-row (feature-name frame-handle token-handle)
  (declare (ignore feature-name frame-handle))
  (- (/ *image-size* 2) (isr2:value '(,token-handle row))))
```

Second we write an if-setting demon that will block the values from getting stored. (Without this demon the features would work perfectly well, but x and y would be stored.)

```
(defun block-set (orig-value new-value feature-name frame-handle token-handle)
  (declare (ignore orig-value feature-name frame-handle token-handle)
           (special isr2:*set-value-flag*))
  (setf isr2:*set-value-flag* nil)
  new-value)
```

It is important in the above example that the if-setting demon return the new value. If the last line is forgotten and the demon returns NIL, then the virtual feature will not work. Every time the user asks for the X (or Y) value, the if-needed demon will calculate it, pass it to the if-setting demon which unfortunately would return NIL, and NIL would be given back to the user as the value of X.

Third we write the if-setting demon that allows the user to move a point by setting X and Y. Internally, the demon reflects this change by resetting the values of row and col.

```
(defun x-to-col (orig-value new-value feature-name frame-handle token-handle)
  (declare (ignore orig-value feature-name frame-handle))
  (setf (isr2:value `(,token-handle col)) (+ new-value (/ *image-size* 2)))
  new-value)

(defun y-to-row (orig-value new-value feature-name frame-handle token-handle)
  (declare (ignore orig-value feature-name frame-handle))
  (setf (isr2:value `(,token-handle col)) (- (/ *image-size* 2) new-value))
  new-value)
```

Finally we add the demons to the X and Y features:

```
(isr2:add-feature-function 'point<?>x$f_if-needed 'x-from-col :first)
(isr2:add-feature-function 'point<?>y$f_if-needed 'y-from-row :first)
(isr2:add-feature-function 'point<?>x$f_if-setting 'block-set)
(isr2:add-feature-function 'point<?>y$f_if-setting 'block-set)
(isr2:add-feature-function 'point<?>x$f_if-setting 'col-from-x :first)
(isr2:add-feature-function 'point<?>y$f_if-setting 'row-from-y :first)
```

7. Functions for saving and retreiving ISR data in disk files

This section describes the file I/O capabilities provided by the ISR. The aim of the I/O system is 1) store and retrieve large amounts of data on any machine in the VISIONS environment, and 2) to allow different researchers to share data and results through the use of ISR files. It is therefore mandatory that all ISR implementations read and write using the same file format.

File storage in the ISR, like run-time memory storage, is organized around the frame. Frames can be saved to, and loaded from, disk files; tokens are saved only with their tokensequence and its frame. Frames may be saved or loaded incrementally. The user may choose to save (load) any subset (including the empty set) of a frame's tokensequence; additionally, the user may select any subset of a frame's features for loading or saving.

Although in memory the ISR is a network database with a hierarchical superstructure, the file structure need not reflect this. Frames may be saved from one location in the tree (i.e. under one pathname), and read into another. This makes it possible for researchers to share data, since each researcher can use frame nomenclature which is appropriate to her research. (For example, what the Line Grouping System calls LGS_LINES the Schema System prefers to call MBLINES.)

When a file is read into memory, the data can either be merged into an existing frame or put into a new one. In the latter case, frame creation happens before any of the data is actually read. Thus the frame and its (token and frame) features are created before any feature values are filled.

When the data being read contains handles, those handles must point at a valid ISR object. When the handle being read points to an object that is already in memory this is not a problem. Such a handle is "referenceable". More precisely, any time the handle being read points at an ISR object whose frame is in memory, the handle is referenceable and therefore can be read directly into memory (this is why frames are read into memory before their data). This protects the user from reading handles to non-existent frames, tokens or features.

If the handle to be read is not referenceable the user has three choices. First, the reading of the current frame can be suspended while the new, referenced frame is read in. Once the new frame and its data have been installed in memory, reading of the original frame can resume. Second, the handle can be replaced with the value :uncalculated. Third, an error can be signalled.

How the system responds to an unreferenceable handle is controlled by the global variable ***resolve-handle-method***. If this variable is set to :error or :uncalculated, the obvious happens. If ***resolve-handle-method*** is set to :ask, the system asks the user for the filename of the frame being referenced and then reads the frame in that file. If ***resolve-handle-method*** is set to :find, it looks in the last known location for the file

and, if it finds it, loads that file. Otherwise it resorts to asking the user for the file name.

7.1 File I/O Functions

(**isr2:describe-file** *filename &key (stream *standard-output*) (features nil) (verbose nil)*)

Effects: Opens the file and prints information about its contents. The exact information printed is implementation dependent, but must include at least: the name of the frame, its canonical pathname, the number of tokens, and the high and low token indices. A system command will also be available to perform the same function from the operating system. If STREAM is supplied and is NIL, then no printing is done. VERBOSE determines the amount of information printed. FEATURES specializes the type of information printed.

Returns:
Case FEATURES
nil: nil;
:FRAME : the list of features defined for the frame;
:TOKEN : the list of features defined for the tokensequence;
:ALL : a list of both types of features. They can be distinguished by the convention that token featurenames are preceded by <?> in feature lists.

Error Conditions: File does not exist or is not an isr format file.

(**isr2:store** *frame-path filename*)

Effects: This is the simple way to save ISR data to a file. All the data in the frame (including its tokens) is saved in the given file. This file becomes the new source file for the frame.

Returns: filename

Error Conditions: Filename is not a legal filename or causes a file system error. Frame-path does not specify a frame in memory.

(**isr2:write-frame** *frame-path filename features tss &optional frame-source-file-replace*)

Effects: Writes out the frame specified by frame-path out to the file named by filename. If features is :ALL, all token features are written, otherwise only the token features specified in the list features are written. If tss is :ALL, all tokens are written, otherwise only the tokens in tss are written. If frame-source-file-replace is non-NIL, the frame's source file is replaced with the filename specified (illegal if either features or tss is not :ALL), otherwise the filename is *added* to the list of previous source file names. NOTE: the convention for distinguishing frame features from token features must be followed. If the handle is a frame pointer: FOO specifies the frame feature FOO, and <?>$FOO specifies the token feature FOO.

Returns: FILENAME.

Error Conditions: FILENAME is not a legal filename. Attempting to open the file results in a filesystem error. PATH does not specify a frame or tokensubsequence. A feature listed is not present in the specified frame/token.

(**isr2:restore** *path filename*)

Effects: This is the simple way to get isr data from a file. PATH locates a feature of the frame (parent-frame) which will be the parent of the file frame (or tree of frames). The complete path (including the feature name) will become the canonical path of the new frame. All the tokens and features in the files are loaded. Handles in the frame data are resolved by restoring any frames not currently in memory.

Returns: The name of the frame loaded.

Error Conditions: FILENAME does not exist or is not an isr format file. Path is not legal.

(**isr2:read-frame** *frame-path filename features tss &key merge-p (sub-frame-action :ask-user)(merge-overlap-action :error)*)

Effects: Read in a frame from file filename into the frame structure at frame-path. If merge-p is non-NIL, merge the new data with an existing frame, otherwise signal an error if frame-path already exists. Sub-frame-action specifies what to do about sub-frames referenced by the frame being loaded. Values can be :ASK-USER - ask the user what to do for each sub-frame, :LOAD - load the sub-frame if posible, :STUB - make a stub-frame, or :ERROR - raise an error if an unresolved sub-frame is referenced. Merge-overlap-action specifies what to do if merge-p is non-NIL and there is data-overlap. Posible values are :ERROR - raise an error, :ASK-USER - ask the user for each instance, :OLD - use existing data, :NEW - use new data, or :WARN-NEW - use new data and issue a warning message (with WARN). Features and TSS control which features and/or tokens to load. :ALL means all, NIL means none. Features should be :ALL, NIL, or a list a feature names, with a posible <? in front. TSS should be :ALL, NIL, or a TSS.

Returns: FILENAME does not exist or is not the name of an isr format file. Path does not specify a frame or a legal location for a frame in memory. An error can also be raised by the user specifying the :error option to one of the arguments if that situation occurs.

Error Conditions:

(**isr2:read-isr1-feature-data** *filename &optional directory*)

Effects: Data saved onto disk in the old (ISR1) format is read into the ISR2's memory. The data is put into a tree of depth two. The ISR1 image becomes the top-level frame (right below root). The ISR1 tokenset becomes a frame under the image frame; the tokens

belong to this frame. The lexicon is read is and used as the token-features for the tokenset frame: the two new frames initially have no frame features. If the image frame already exists, a new one will not be made. If the tokenset frame already exists, an error results.

Returns: filename

Error Conditions: filename does not specify an ISR1 data file. The image-tokenset frame already exists.

(**isr2:read-label-plane** *frame-path filename*)

Effects: Reads data stored in the old label-plane format. In particular, creates tokens in the frame specified by frame-path whose indices correspond to the labels on the label-plane amd whose pixelmaps correspond to the regions specified.

Returns: a handle to the specified frame

Error Conditions: Frame-path does not specify a frame, or specifies a frame that already contains tokens whose indices conflict with those in the label plane. Filename does not specify a label-plane, as defined by LLVS.

8. Tokensubsequence (TSS) operations – associative retrieval

Tokensubsequence (or Tokensubset) functions are a natural evolution of the Associative Access functions in ISR1. The Associative Access functions emulate physical associative memory in its ability to select a class of tokens by the value of one or more features. This emulation has had a very favorable reception. The only real complaint about it was that there is not more of it. A strong impetus for the design of ISR2 was the generalization and regularization of the benefits of Associative Access. This has led to the concept of a Tokensubsequence or Tokensubset (TSS). In all versions of the ISR, there has been a data structure attached to each tokensequence, which indicates which tokens in the sequence actually exist. Although hidden from the user, this is the same kind of data structure necessary to implement *associative state* in ISR1 or a TSS in ISR2. ISR2 takes the obvious step of making these data structures first class objects. In keeping with our use of TSS for token subset, the complete set of tokens in a frame is a token set or TS.

The TSS is a representation which appears uniform to the user in both the C and LISP implementations of the ISR. As far as the user is concerned, the TSS behaves exactly like a handle to a Tokensequence. It indicates the frame (and tokensequence) and may in some cases indicate the feature to be accessed and the index of the token. All the normal access functions can be performed with a TSS handle just as they can with a TS.[4]

[4]There is one major exception: *create-token* on a TSS will create a *local* token, i.e., one known only to that TSS. That token will not be visible to the TS owning it.

There are other benefits to TSSs in addition to the value-based selection of tokens. With it one can

- Select a set of tokens from a tokensequence. Based on an ordered sequence of constraints, choose the tokens which satisfy those constraints and return a TSS corresponding to those tokens.
- Perform set operations. Given two sets of tokens based on the same tokensequence, e.g., *Set1* and *Set2*, it is possible to take the union, intersection, complement, and difference of them and return the result as a tokensubset. Set1 could be a TS or a TSS. Set2 could be a TS or TSS as well, or it could be a set defined by a restriction on a feature value, by a location, or by any other predicate.
- Add a "local" token to a tokensequence. Creating a token in a TSS will add that token to the tokensequence, but it will not be visible globally. It will only be visible within that TSS.
- Delete a token "locally" from a TSS.
- Select a unique member of a TSS, without consing up a list of members. This is a *choice* or *pick* function, each time it is executed on a given TSS, a different token is returned until there are no more to choose from.
- Do something to all members of a TSS. (Similar to LISP "map" function.)

8.1 Basic TSS operations

For any given TS, there is a fundamental TSS, which indicates which tokens are globally visible in the TS. Whenever a path argument in a TSs function indicates a TS, then the global visibility TSS will be the one used.

(**isr2:make-tss** *path*)

Effects: Makes a copy of the TSS indicated by PATH. PATH may be a TS, a TSS or a path to a feature whose value is a TSS.

Returns: A copy of the TSS described by path. If PATH describes a TS, then a **isr2:make-tss** returns a TSS which defines those tokens which are globally visible in the TS. If PATH is (or leads to) a TSS, a copy of that TSS is returned. This function is handy if you want to get a clean copy of a TSS, with which you will then perform a series of destructive operations.

Error Conditions: A bad PATH.

(**isr2:make-null-tss** *path*)

Effects: Creates a null TSS for PATH. PATH may be a TS, a TSS or a path to a feature whose value is a TSS.

Returns: A null TSS for PATH. This function is handy if you want to get a clean empty TSS, with which you will then perform a series of destructive operations.

Error Conditions: A bad PATH.

(**isr2:make-null-tss!** *tss*)

Effects: Removes all tokens from TSS. The altered TSS. This function is handy if you want to clear a TSS.

Returns: (tss) is not a TSS.

Error Conditions:

(**isr2:token-count** *tss*)

Effects: No side effects

Returns: The number of tokens in the TSS. If tokens have been destroyed since the TSS was created, this may overestimate the number of existing tokens in the TSS (i.e. it may count destroyed tokens). Such tokens will not be found when traversing the TSS via pick or *for-every-token*

Error Conditions: (tss) is not a TSS.

8.2 Associative retrieval / Set operations

The functions in this section play two roles in the ISR2:

- **Extensional.** They provide basic set operations over TSS's.
- **Intensional.** They provide associative access capabilities.

The functions are the ISR2 variations on the basic set operations *union*, *intersection* and *difference*. When they are given two TSS as arguments, we say that they are used extensionally, and they perform the set operations as you would expect. Associative access is performed by using these same operations intensionally. To do this, the user provides just one TSS, and a second argument that implicitly defines a TSS. For example, the second argument might specify the set of lines with contrast between 5 and 10. Or, the second argument might specify the set of all tokens that evaluate to true according to some predicate (not unlike the :test keyword in CommonLisp functions such as **member**). The ISR then computes the set operation between the extensional set (the TSS provided by the user as the first argument) and the intensional set (the definition provided as the second argument).

There are two CommonLisp structures currently defined for intensionally defining sets: `isr:range` , and `isr:predicate` .

- `isr:range`
  - `:feature`
  - `:min`
  - `:max`
  - `:circular-p`
- `isr:predicate`
  - `:function (function-name or closure)`
  - `:features`

The `isr:range` structure most closely matches the associative access mechanisms of the ISRI. It allows the user to specify a feature, minval and maxval. The set defined is then all tokens whose value for the given feature is between minval and maxval. Circular features (such as theta or hue) can be handled by setting :circular-p to be non-nil. Then, if minval is less than maxval the feature is handled as usual, but if minval is greater than maxval, all tokens with feature values above minval or below maxval are included in the set.

The `isr:predicate` is the most general way to define a set constraint. Any token that evaluates to non-nil under the predicate is included in the set. The predicate is called as follows: (*pred* frame-handle token-index feature1 feature2...), where arguments 2 through N are the values of the features given in the predicate structure, in the same order as they appear in that structure.

Where the set operation depends on the value of some feature, we must determine what action to take if the value is **uncalculated** or **undefined** . For this purpose we have included &KEY arguments which carry that information.

In the following functions, the value of the IF-UNCALCULATED argument determines what will be done if the feature value is **uncalculated** for a given token. If IF-UNCALCULATED is NIL then that token will be skipped and not included in the TSS returned. If IF-UNCALCULATED is T, then that token will be included in the TSS without any further action. If IF-UNCALCULATED is :compute (or :calculate), then the *:if-needed* function for the feature will be run.

The value of the IF-UNDEFINED argument determines what is to be done if the value of the feature is **undefined** (meaning that the feature is meaningless for that particular token). If IF-UNDEFINED is NIL, then that token is not included in the TSS; if IF-UNDEFINED

is non-NIL, then the token is included in the TSS. However, if IF-UNDEFINED is :calculate or :compute, an error will be issued since it is likely that the user is confusing **uncalculated** with **undefined** in the function call.

We must also decide if we are to call the if-getting demons on each read of a feature. Generally we will not want to do this, but for some kinds of features the user may wish the intervention of the if-getting demons. The value of the IF-GETTING argument determines the action after reading the feature value. Usually the user will not want the *:if-getting* demon(s) run, so the default for the IF-GETTING argument in the following functions is NIL. If the IF-GETTING argument is non-NIL, then the *:if-getting* demon, if any, is run.

Destructive versions of the set operations are also implemented, whereby the user may supply a TSS as the first argument, and the result of the operation will be stored in that TSS, returning it to the user (in a non-destructive version a copy of the TSS would be created to receive the result). This gives the user better control over storage. The system will insure that the *global visibility TSS* for a tokensequence is never destructively altered in this way. An attempt to do so will always result in an error. Destructive TSS operations end in a "!".

(isr2:tss-intersection *path set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*))*

Effects: Creates a new TSS with the tokens in both the TSS described by PATH and SET-DESCRIPTION. See the text at the start of this set of descriptions regarding the rest of the &key arguments.

Returns: A TSS containing those members of the TSS described by PATH for which the SET-DESCRIPTION is also true on the selected features (non-nil in LISP). Note again, the appropriate actions for **undefined** and **uncalculated** features are controlled by the keyword arguments.

Error Conditions: A bad PATH. FEATURES do not designate features of the tokenset.

(isr2:tss-intersection! *TSS set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*))*

Effects: The destructive version of *tss-intersection*. Modifies TSS in accordance with the second argument.

Returns: The modified TSS.

Error Conditions: A bad PATH.

(isr2:tss-union *path set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*))*

Effects: Creates a new TSS with the tokens in the TSS described by PATH plus additional tokens designated by SET-DESCRIPTION.

Returns: A TSS containing those members originally in the TSS described by PATH in addition to those in the TS for which the SET-DESCRIPTION is true (or for which the relevant feature is **undefined** or **uncalculated** and the corresponding argument to **isr2:tss-union** is T.

Error Conditions: A bad PATH.

(**isr2:tss-union!** *TSS set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*)*)

Effects: The destructive version of *tss-union*. Modifies TSS in accordance with the second argument.

Returns: The modified TSS.

Error Conditions: A bad PATH.

(**isr2:tss-difference** *path set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*)*)

Effects: Creates a new TSS with the tokens in the TSS described by PATH minus the tokens for which the SET-DESCRIPTION is true.

Returns: A TSS containing those members originally in the TSS described by PATH minus those in the TS for which the SET-DESCRIPTION is true (or for which the relevant feature is **undefined** or **uncalculated** and the corresponding argument to **isr2:tss-difference** is T.

Error Conditions: A bad PATH.

(**isr2:tss-difference!** *TSS set-description &key (:if-uncalculated* NIL*) (:if-getting* NIL*) (:if-undefined* NIL*)*)

Effects: The destructive version of *tss-set-difference*. Modifies TSS in accordance with the second argument.

Returns: The modified TSS.

Error Conditions: A bad PATH.

8.3 Sorts

A sort is a special case of a TSS, in which the tokens are ordered according to some key, in either ascending or descending order.

Sorts in the isr are based on a single key. Sort, however, is *stable* (i.e. if X comes before Y in a sort, and you sort the sort on a new key, and if X and Y have the same feature values for the new key, then X will come before Y in the resulting sort). As a result, multiple key sorts can be approximated by sorting a single tss multiple times, in the order of least significant key to most significant key.

(**isr2:tss-sort** *tss-path key-feature &key (order :ascending) (calculate-p nil) (value-p nil)*)

Effects: Creates a sort-tss from tss-path. This sort can be accessed in sort order with **pick** and **for-every-token**. ORDER must be either 1) :ascending (smallest item first) or 2) :descending (largest item first). KEY-FEATURE must have a numeric or string datatype.

If the value of key-feature is undefined for a token, it is not included in the sort. If the value is uncalculated and CALCULATE-P is nil (the default), then the token is not included. If CALCULATE-P is non-nil, then the value is calculated (once) if necessary. If VALUE-P is nil (the default) then no deamons (if-needed or if-getting) are triggered by the sort. If VALUE-P is non-nil, then the sort uses **value** internally to fetch the feature values. As a result, all deamons will be triggered, possibly repeatedly.

Returns: TSS-PATH is not a proper path. KEY-FEATURE is not a feature or is a feature whose datatype is not numeric or string.

Error Conditions:

(**isr2:tss-sort-key** *sort-tss*)

Effects: None.

Returns: Two values. The first is the feature handle of the sort key. The second is the sort order, either :ascending or :descending.

Error Conditions: TSS-SORT is not a valid sort.

(**isr2:sort-update!** *sort-tss &key (calculate-p nil) (value-p nil)*)

Effects: Update the SORT-TSS to account for added or deleted tokens using the same sorting key. This will generally be less expensive than sorting the TSS from scratch. The CALCULATE-P and VALUE-P arguments work the same as for *tss-sort*. NOTE: If a token has been neither deleted nor inserted into the TSS, and yet its feature value (for the sort key) has changed, then resort will not reposition that token in the tss-sort. In this case, **tss-sort** should be used.

Returns: TSS-SORT is not a valid sort

Error Conditions:

8.4 Sequence element functions

The final group of functions allow us to choose the tokens in the tokensubsequence one by one; to do some operation to all of them; to check if a given token is in the TSS.

(**isr2:tss-mem** *token-handle TSS*)

Effects: Equivalent to Lisp `member` or `memq` .

Returns: Returns true (`t` in LISP) if TOKEN is in TSS.

Error Conditions: TOKEN and TSS are not from same tokensequence.

(**isr2:pick** *path*)

Effects: Selects a unique token from the TSS described by path. Ensures that the token selected will not be considered for subsequent calls of **isr2:pick** . The use of **isr2:pick** should not affect other uses of the TSS.

Returns: A token-handle if there are any tokens left to choose from, else **nil** .

Error Conditions: TSS is not valid.

(**isr2:pick-reset!** *path*)

Effects: PATH describes a TS, TSS or sort, so that **pick** is a legal operation on it. **Reset** "undoes" any previous picks, i.e. after a reset, the next pick will start at the beginning of the TSS again, and return the first token.

Returns: The handle described by PATH

Error Conditions: PATH does not denote an ist object that can be "picked", i.e. a TS, TSS or sort.

(**isr2:for-every-token** *(token-symbol path &optional feature-list) &body body*)

Effects: Traverses the TSS described by PATH and executes body with the feature-names in feature-list bound to the appropriate feature-handles indexed to the current token.

NOTE: for-every-token destructively modifies the token handle that **token-symbol** is bound to between each iteration. As a result, if the handle is to be saved for use outside of the loop, it must be copied, otherwise it will not have the expected value when the loop exits.

Returns: The value returned by body.

Error Conditions: PATH does not describe a tokensubsequence or tokensequence. FEATURE-LIST contains feature names not in the tokenset features.

(**isr2:for-every-token!** *(token-symbol tokensubset feature-handle-list &key (error-check nil)) &body body*)

Effects: Body is executed once for each token in the given tokensubset. Each time through the loop, token-symbol is bound to the current token handle. Feature-handle-list is a list of feature handles from the given tokensubset. Each time through the loop all feature handles get destructively changed to point to their respective features for the current token. By default, no error checking takes place, but error checking can be forced by specifying the error-check key as non-nil. Error checking tests to see if each feature-handle in the feature-handle-list points to a feature in the given tokensubset.

Returns: NIL

Error Conditions: PATH does not describe a TS or TSS; feature-handle-list contains something other than feature handles: **Error-check** is non-nil and a feature-handle in feature-handle-list does not point to a feature in the given tokensubset.

(**isr2:for-every-feature!** *(feature-symbol feature-list frame-handle &optional token-handle-list &key (error-check nil)) &body body*)

Effects: Body is executed once for each feature in the feature list. Each time through the loop, feature-symbol is bound to the current feature handle. Frame-handle is a reference frame for the feature-list. Token-handle-list, if given, is a list of token handles from the same tokensubset. Each time through the loop all token handles are destructively changed to point to the current feature for their respective tokens. By default, no error checking takes place, but error checking can be forced by specifying the error-check key as non-nil.

Returns: NIL

Error Conditions: **Error-check** is non-nil and a token-handle in **token-handle-list** has a parent frame other than **frame-handle** or a feature in *feature-list* is not a defined token feature of *frame-handle*. It is always an error if **frame-handle** is not a frame handle.

(**isr2:add-tokens** *token-handle-list TSS*)

Effects: Copy the TSS and add the tokens in TOKEN-HANDLE-LIST to the copy.

Returns: The new TSS with the tokens added.

Error Conditions: The tokens in TOKEN-HANDLE-LIST and the TSS are from different tokensequences.

(**isr2:add-tokens!** *token-handle-list TSS*)

Effects: Destructive version of **tss:add** . Add the tokens in TOKEN-HANDLE-LIST to TSS.

Returns: The modified TSS with the tokens added.

Error Conditions: The tokens in TOKEN-HANDLE-LIST and the TSS are from different tokensequences.

(**isr2:remove-tokens** *token-handle-list TSS*)

Effects: Copy the TSS and remove the tokens in TOKEN-HANDLE-LIST from the copy.

Returns: The new TSS with the tokens removed.

Error Conditions: The tokens in TOKEN-HANDLE-LIST and the TSS are from different tokensequences.

(**isr2:remove-tokens!** *token-handle-list TSS*)

Effects: Destructive version of **tss:remove** . Remove the tokens in TOKEN-HANDLE-LIST from TSS.

Returns: The modified TSS with the tokens removed.

Error Conditions: The tokens in TOKEN-HANDLE-LIST and the TSS are from different tokensequences.

8.5 Other tss functions

Internally, TSS's can be stored in two formats (at the discretion of the system). One format is for long TSS's, the other for TSS's with only a few tokens. The point at which one format becomes more efficient than another depends on the machine involved. For the TI Explorers (with a 32 bit word), the short format is more efficient if less than $\frac{1}{32}$ of the tokens in the tokensequence are in the TSS. When a TSS is created, the ISR will make in the most efficient format. Once made, however, it will not reoptimize it every time its size changes. **Optimize-tss-storage** allows the user to instruct the ISR to test a TSS and reformat it, if the result will use less memory.

(isr2:optimize-tss-storage *tss***)**

Effects: TSS *may* be converted into an internal format that saves memory space. Functionally, the TSS is not changed.

Returns: TSS, possibly reformated

Error Conditions:

9. Grid (GSS) operations – spatial retrieval

9.1 Spatial Access using Grids

Since the ISR is a database designed to satisfy the particular needs of computer vision researchers, it is natural that special purpose mechanisms have been included to support efficient retrieval of tokens based on spatial position. In the ISR this is provided through *grids*. Conceptually, the grid is a 2D spatial array of *cells*, each of which contains a set of tokens. Each grid contains tokens from one and only one frame. The grid is a fundamental ISR datatype, and a grid can be stored in any feature of type *:GRID*. Consequently, grids can also be saved to a file as part of a frame and read back in again. A grid is created with the *isr2:make-grid* command.

The ISR provides a set of functions which map tokens to the grid (i.e. *rasterizing* functions) in a manner suitable for the majority of storage and retrieval tasks. At any given time, some subset of the grid cells are selected. Store and retrieve only affect selected cells. This is all the general user of grids really needs to know, however users interested in novel access methods may save and combine these selection states. A distinct type of ISR object, the *grid subset* or *GSS* may be used to capture these states. There is an implicit GSS associated with every ISR grid, and anywhere a GSS is used a grid may be used in its place.

The purpose of a grid, of course, is to provide fast spatial access to a set of tokens. This requires the ability to store tokens in a grid and to retrieve them again. Retrieval operations can be divided into two types. "Coarse" retrieval fetches all the tokens in the specified area.

In addition, due to quantization effects associated with token rasterization, it may fetch an unspecified number of additional tokens. Tokens returned by a coarse retrieval operation will therefore be referred to as "candidate" tokens. There is an implicit assumption that the user will use the spatial access mechanism to retrieve a first approximation to the set of tokens desired. We assume that the user will select from among the candidates returned those tokens satisfying more precise spatial predicates, (i.e. "Fine" retrieval), when additional precision is required. The "Coarse" level of retrieval will be supported directly using grids. Users are free to develop their own "Fine" retrieval filters.

Grids also keep statistics about their use; specifically, the number of accesses, the average number of cells per access and the average number of tokens returned per access.

What follows is a function by function description of the ISR grid facility. Following the function descriptions are several annotated examples of grid usage.

9.2 Making Grid and GSS Objects

(**isr2:make-grid** *frame-path row-min row-max row-size col-min col-max col-size*)

Effects: Makes a new ISR2 grid object. Frame-path must designate a frame and the grid will accept tokens only from this frame. The grid spans from row-min to row-max with cells of width row-size, and from col-min to col-max with cells of width col-size. Cells are numbered for access such that a point (row, col) is mapped onto cell $\lfloor \frac{row}{size_{row}} \rfloor \cdot \lceil \frac{col_{max}}{col_{size}} \rceil + \lfloor \frac{col}{col_{size}} \rfloor$.

Returns: A handle to the new grid.

Error Conditions: Frame-path does not specify a frame; row-max < row-min; col-max col-min; row-size $\geq$ 0; col-size $\geq$ 0.

(**isr2:make-gss** *grid*)

Effects: None.

Returns: Returns a new ISR2 grid subset (GSS) object from the specified grid, capturing its current state. This object is used for rasterization. A GSS indicates which subset of grid cells is selected. There is an implicit GSS associated with each grid, and insofar as the rasterization functions are concerned, grids and GSS's may be used interchangeably. A grid may alway be used in place of a GSS, in which case it indicates that the implicit subset associated directly with the grid should be used.

Error Conditions: Grid does not specify a grid.

9.3 Access/Storage Functions

(**isr2:grid-store** *gss token*)

Effects: Stores token in every grid cell selected in the GSS.

Returns: Returns its token (2nd) argument.

Error Conditions: gss is not a GSS or a grid. Token is not a token.

(**isr2:grid-retrieve** *gss*)

Effects: If logging is enabled, grid access statistics are updated.

Returns: Returns a TSS representing the union of the contents of the cells selected in the GSS.

Error Conditions: gss is not a GSS or grid.

(**isr2:grid-remove** *gss token*)

Effects: Remove token from the grid cells indicated by the GSS. Note: It is not an error to remove a token from a cell at which it is not present. It is simply a no-op.

Returns: Returns its token (2nd) argument

Error Conditions: gss is not a GSS or grid; token is not a token; token is not of the appropriate tokenset for gss.

(**isr2:grid-clear!** *grid*)

Effects: Removes all tokens from grid.

Returns: Returns a handle to the grid.

Error Conditions: Grid is not a grid. Since this a highly destructive operation effecting the grid itself, the grid itself and not a GSS must be specified.

9.4 Rasterization Functions

Remember that a grid (and its implicit GSS) may always be (and most often will be) used in place of a GSS. The rasterization functions listed below are of two types. The first type are rather generic, and are suitable for rasterizing any token that can be mapped in any manner to points, lines and polygons. For example, although it is often desirable to map a line token to a grid as a line segment, it is also possible to map a line token to the grid as two points, to support endpoint proximity queries. It is worth pointing out that the method of rasterization chosen for storage needn't be the same one used for subsequent retrieval. A line token can be rasterized as a line segment, and retrieved based on its proximity to a point for example. The second rasterization function type (with the "token" suffix) are more specific, and allow certain standard tokens to be rasterized in the obvious ways, (i.e. point-token as point, line-token as line segment, etc.)

(**isr2:rasterize-point** *gss row col &optional (initialize t)*)

Effects: The cell of the gss or grid in which the point falls is selected. If the optional argument initialize is t, then the GSS will first be cleared so that the newly selected cell is the only selected cell. Otherwise, cells not directly associated with the specified point will

retain their previous selection state. If either row or col are :uncalculated or :undefined then no cell is selected.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss does not specify a GSS or grid. Row and/or col are not numbers (or :undefined or :uncalculated) or are outside the limits of the grid.

(**isr2:rasterize-line** *gss row1 col1 row2 col2 &optional (initialize t) (out-of-bounds-error nil)*)

Effects: Selects the grid cells intersected by the specified line. If the optional argument initialize is t, then GSS will first be cleared. Otherwise, cells not directly associated with the specified line will retain their previous selection state. If any row or column coordinates are :uncalculated or :undefined then no cell is selected.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss is not a grid or GSS. Any of row1, col1, row2 or col2 are outside the bounds of gss and out-of-bounds-error is true.

(**isr2:rasterize-polygon** *gss coordinates &optional (initialize t) (out-of-bounds-error nil)*)

Effects: selects the grid cells intersected by the specified polygon. The polygon need not be convex. The coordinates are specified as a list of alternating row and column coordinates: (r1 c1 r2 c2 r3 c3 ...). If the optional argument initialize is t, then gss will first be cleared. Otherwise, cells not directly associated with the specified polygon will retain their previous selection state. If any row or column coordinates are :uncalculated or :undefined then no cell is selected.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss is not a grid or GSS. Coordinates is not an even-length list of numbers and/or :uncalculated / :undefined. One or more of the coordinates is outside the bounds of GSS and out-of-bounds-error is true.

(**isr2:rasterize-pixelmap** *gss pixelmap &optional (initialize t) (out-of-bounds-error nil)*)

Effects: Selects the grid cells intersected by the specified ISR2 pixelmap. If the optional argument initialize is t, then GSS will first be cleared. Otherwise, cells not directly associated with the specified pixelmap will retain their previous selection state.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss is not a grid or GSS; pixelmap is not an ISR2 pixelmap. Pixelmap is outside the bounds of GSS and out-of-bounds-error is true.

9.5 Modify and/or Query Grids

The casual user might never need to use these functions, but they support more direct manipulation of grid and grid subset objects. We have purposely not included GSS complement and GSS difference operations on the grounds that such operations violate the "Coarse" retrieval philosophy underlying spatial querying in the ISR. The user should strive to write code which will operate independently of specific grid dimensions and doesn't depend critically on the precision with which tokens are rasterized. "Coarse" retrieval is intended primarily as a mechanism for increased efficiency, under the premise that such efficiency often makes the difference between what is possible and impossible to compute. GSS complementation and GSS differencing have the undesirable property that after use, one can no longer insure that what is retrieved by a "Coarse" retrieval is a superset of what was requested.

(isr2:grid-cells-selected *gss*)

Effects: None.

Returns: A sorted list of integers indicating those cells which are selected.

Error Conditions: gss is not a GSS or grid.

(isr2:gss-union *gss1 gss2*)

Effects: None.

Returns: Return the handle of a new gss (but not a new grid) containing the union of the selected grid cells from gss1 and gss2.

Error Conditions: gss1 and/or gss2 are not grids or GSS's. Gss1 and gss2 are grids of different tokensets.

(isr2:gss-union! *gss1 gss2*)

Effects: GSS1 is destructively modified to the union (in terms of selected cells) of gss1 and gss2.

Returns: The handle of gss1, which has been modified.

Error Conditions: gss1 and/or gss2 are not grids or GSS's. Gss1 and gss2 are grids of different tokensets.

(isr2:gss-intersection *gss1 gss2*)

Effects: None.

Returns: The handle of a new gss containing the intersection of the selected grid cells from gss1 and gss2

Error Conditions: gss1 and/or gss2 are not grids or GSS's. Gss1 and gss2 are grids of different tokensets.

(**isr2:gss-intersection!** *gss1 gss2*)

Effects: Gss1 is destructively modified to be the intersection of the selected cells of gss1 and gss2.

Returns: The handle of gss1 which is modified to contain the intersection.

Error Conditions: gss1 and/or gss2 are not grids or GSS's. Gss1 and gss2 are grids of different tokensets.

(**isr2:grid-unselect** *gss &OPTIONAL (cell-index)*)

Effects: Forces the specified grid cell to be unselected. If no cell is specified then ALL grid cells are unselected.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss is not a grid or GSS; cell-index is out of range for gss.

(**isr2:grid-select** *gss cell-index &optional (out-of-bounds-error nil)*)

Effects: Force the specified grid cell to be selected.

Returns: A handle to gss (if GSS is specified as a handle, the returned value will be the first argument).

Error Conditions: gss is not a grid or GSS; cell-index is out of range for gss and out-of-bounds-error is true.

(**isr2:grid-cell-tss** *gss cell-index*)

Effects: None.

Returns: Return a TSS handle containing the set of tokens stored in the specified cell.

Error Conditions: gss is not a grid or GSS; cell-index is out of range for gss.

9.6 Querying Grid Attributes

These functions retrieve basic attributes of the grid relating to its dimensions.

(**isr2:cell-index** *gss row col*)

Effects: None.

Returns: $\lfloor \frac{row}{size_{row}} \rfloor \quad \lfloor \frac{col_{max}}{col_{size}} \rfloor + \lfloor \frac{col}{col_{size}} \rfloor$.

Error Conditions: gss is not a grid or GSS; row or col is outside the range of gss.

(**isr2:grid-row-min** *gss*)

Effects: None.

Returns: Starting position of grid in row dimension.

Error Conditions: gss is not a grid or GSS.

(**isr2:grid-row-max** *gss*)

Effects: None.

Returns: Ending position of grid in row dimension.

Error Conditions: gss is not a grid or GSS.

(**isr2:grid-col-min** *gss*)

Effects: None.

Returns: Starting position of grid in column dimension.

Error Conditions: gss is not a grid or GSS.

(**isr2:grid-col-max** *gss*)

Effects: None.

Returns: Ending position of grid in column dimension.

Error Conditions: gss is not a grid or GSS.

(**isr2:grid-row-size** *gss*)

Effects: None.

Returns: Width of grid cell in row dimension.

Error Conditions: gss is not a grid or GSS.

(**isr2:grid-col-size** *gss*)

Effects: None.

Returns: Width of grid cell in column dimension.

Error Conditions: gss is not a grid or GSS.

9.7 Grid Usage Monitoring Functions

These functions retrieve grid usage statistics maintained by the ISR. Since efficiency of retrieval is the primary motivation for using grids, and misuse and/or uninformed use of the grid mechanism can often be less efficient than a simple brute force search, it is hoped that these functions will allow the user to determine the optimal grid cell size for his particular application. After the optimal cell size is determined, the user can disable the grid usage monitor. The statistics are stored directly with the grid, and not with a GSS, therefore it is misleading to permit a GSS argument for these functions.

(**isr2:grid-logging-enabledp** *grid*)

Effects: None.

Returns: T if the grid is configured to accumulate statistics on use, otherwise nil. The default is nil for a newly created grid. This value may be set using *setf*.

Error Conditions: Grid does not specify a grid.

(isr2:grid-store-count *grid*)

Effects: None.

Returns: The total number of times the grid has been used to store

Error Conditions: . Grid does not specify a grid.

(isr2:grid-retrieve-count *grid*)

Effects: None.

Returns: The total number of times the grid has been used to retrieve.

Error Conditions: Grid does not specify a grid.

(isr2:grid-cells-per-rasterization *grid*)

Effects: None.

Returns: The average number of grid cells selected per rasterization.

Error Conditions: Grid does not specify a grid.

(isr2:grid-tokens-per-retrieve *grid*)

Effects: None.

Returns: The average number of tokens returned per retrieval.

Error Conditions: Grid does not specify a grid.

(isr2:grid-stats-clear *grid*)

Effects: Reset the grid's store and retrieve access counts to zero, and tokens-per-access and cells-per-access features to :undefined.

Returns: T

Error Conditions: Grid does not specify a grid.

9.8 Annotated Example

Consider first the code to create a grid for storing line tokens associated with a 512x512 image. Let the frame containing the line segments be IMAGE1$LINES.

First add a grid feature to the lines frame. Then make a grid and store it in the newly created feature slot.

```
(isr2:define-feature (list 'image$lines 'coarse)
                     "Coarse Spatial Grid" :grid)
(setf (isr2:value (list 'image$lines$coarse))
      (isr2:make-grid (list 'image$lines)
                      0 511 16 0 511 16))
```

Next store the line segments into the grid.

```
(let ((grid (isr2:value (list 'image$lines$coarse))))
   (isr2:for-every-token (token 'image$lines)
      (isr2:rasterize-line grid row1 col1 row2 col2)
      (isr2:grid-store grid token)))
```

To retrieve all of the line tokens in cells intersected by a dynamically specified line.

```
(defun coarse-retrieve-lines (grid row1 col1 row2 col2)
   (isr2:rasterize-line grid row1 col1 row2 col2)
   (isr2:grid-retrieve grid))
```

Consider as an alternative, a function to retrieve candidate lines within a polygon surrounding a given line token. This provides the user some flexibility in defining the exact character of the bounding polygon. Assume the user has defined a function bounding-polygon-for-line-token which takes a line token and returns a polygon specification. Then the coarse retrieval would be done as follows.

```
(defun coarse-retrieve-within-bounding-polygon (grid line-token)
   (isr2:rasterize-polygon grid (bounding-polygon-for-line-tok line-token))
   (isr2:grid-retrieve grid))
```

10. Pixelmap operations

10.1 Pixelmaps

Pixelmaps are a unique composite datatype in the ISR, used to associate an ISR token with an arbitrary set of image pixels. They are unique because on of the pixelmap's slots – mask – is not a primitive ISR datatype.

Pixelmaps have 2 slots:

- ISR:PIXELMAP
  - :EXTENTS

- :BITPLANE

The bitplane field is a lisp bit array (an array of type 'art-1b) denoting which pixels are in the region. In order to properly interpret it one also needs the extents structure, which records over what part of the image the bitplane array should be placed. The extents structure has 6 fields:

- ISR:EXTENTS
  - :MINX
  - :MAXX
  - :MINY
  - :MAXY
  - :BYTE-BOUND
  - :BYTE-WIDTH

This slot structure has been adopted from the ISR1 for compatibility (i.e. $Y = row$, $X = col$). The :MINX, :MAXX, :MINY, and :MAXY slots are the minimum/maximum coordinates of pixels in the map. :BYTE-BOUND is the leftmost column covered by the mask, which may be padded to the left of :MINCOL. Similarly, :BYTE-WIDTH is the width of the mask, in bytes, and may be padded to the right. Padding is allowed so that machine dependent graphics routines, which may depend on machine specific word lengths, can be used in the implementation of the routines in this section.

10.2 Functions on Pixelmaps

(isr2:pix-emptyp *pixelmap*)

Effects: None.

Returns: T if the bitplane contains zero bits, NIL otherwise

Error Conditions: Pixelmap is not a pixelmap, or bitplane field does not contain a bit array.

(isr2:pix-count *pixelmap*)

Effects: None.

Returns: The number of bits that are on in the pixelmap.

Error Conditions: Pixelmap is not a pixelmap, or bitplane field does not contain a bit array.

(**isr2:pix-aref** *row col pixelmap*)

Effects: None.

Returns: T if the pixel (row, col) is a in the pixelmap, NIL otherwise. Note: pix-aref can also be used in conjunction with the CommonLisp SETF macro to set the value of a pixel.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(**isr2:print-pixelmap** *pixelmap &optional (stream t)*)

Effects: The pixelmap is printed to the stream (if provided) or to the terminal (the default).

Returns: pixelmap

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(**isr2:pix-intersection** *pixelmap1 pixelmap2*)

Effects: None.

Returns: A pixelmap containing the intersection of pixels in pixelmaps 1 & 2.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(**isr2:pix-union** *pixelmap1 pixelmap2*)

Effects: None.

Returns: A pixelmap containing the union of pixels in pixelmaps 1 & 2.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(**isr2:pix-union!** *pixelmap1 pixelmap2*)

Effects: Pixelmap1 may become the union of the two pixelmaps. To be precise, if the extents box of pixelmap1 includes the extents box of pixelmap2, then pixelmap1 will be altered.

Returns: A pixelmap containing the union of pixels in pixelmaps 1 & 2.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(**isr2:pix-set-difference** *pixelmap1 pixelmap2*)

Effects: None.

Returns: A pixelmap containing the set difference of pixels in pixelmaps 1 & 2.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(isr2:pix-set-difference! *pixelmap1 pixelmap2*)

Effects: Pixelmap1 may be altered

Returns: A pixelmap containing the set difference of pixels in pixelmaps 1 & 2.

Error Conditions: Pixelmap is not a pixelmap, its extents field does not contain an extents structure, or its bitplane field does not contain a bit array.

(isr2:specify-pixelmap *&rest points*)

Effects: None.

Returns: Returns a pixelmap which is the closed polygon specified by POINTS. Points should be in the (row, col) format, and should be in order, i.e. they should form a walk around the polygon. If no points are given, NIL is returned. If one point is given, the result is a point. If two points are given a line results, and above that a solid figure is always returned.

Error Conditions: An argument is something other than a list of two numbers. If the arguments are out of order, no error will occur, but the shape of the resulting bitplane is undefined.

(isr2:for-every-pixel *pixelmap &body body*)

Effects: Applies BODY for each pixel in the bitplane, with the following variables set: 1) USER:*ROW* - the current row. 2) USER:*COL* - the current col.

Returns: NIL

Error Conditions: Pixelmap is not a pixelmap.

For every pixel is used to iterate over the pixels in a pixelmap. This is particularly useful when measuring properties of a region, for instance size or shape. For example, consider the following code which counts the number of pixels in a region:

```
(defun pixel-count (pixelmap)
   (let ((sum 0))
      (isr2:for-every-pixel pixelmap
         (incf sum))
      sum))
```

(isr2:for-every-image-pixel *(pixelmap &rest var-image-pairs) &body body*)

Effects: Applies BODY for each pixel in the bitplane, with the following variables set: 1) USER:*ROW* - the current row. 2) USER:*COL* - the current col. 3) The first element of every var-image-pair is bound to the value of the pixel specified by USER:*ROW* and USER:*COL* in the image array provided as the second element of the pair.

Returns: NIL

Error Conditions: Pixelmap is not a pixelmap. Var-image-pairs is not a list of pairs, the first element of which is a variable to be bound and the second element of which is an appropriately-sized image array.

As an example of how for-every-image-pixel is used, consider the following function for computing the average intensity of a region:

```
(defun average-intensity (pixelmap intensity-image)
   (let ((sum 0) (count 0))
      (isr2:for-every-image-pixel (pixelmap (im intensity-image))
         (incf sum im)
         (incf count))
      (unless (zerop count) (/ sum count))))
```

(**isr2:for-every-image-pixel*** *(pixelmap &rest var-image-pairs) &body body*)

Effects: Applies BODY for each pixel covered by the bitplane, whether or not the pixel is "on" in the bitplane. When body is applied the following variables are set: 1) USER:*PIX* - the current value of the pixelmap (1 or 0). 2) USER:*ROW* - the current row. 3) USER:*COL* - the current col. 4) USER:Y-INDEX - the current row index into the bitplane, to be used for setting pixels in the bitplane. 5) USER:X-INDEX - the current col index into the bitplane, to be used for setting pixels in the bitplane. 6) The first element of every var-image-pair is bound to the value of the pixel specified by USER:*ROW* and USER:*COL* in the image array provided as the second element of the pair.

Returns: NIL

Error Conditions: Pixelmap is not a pixelmap. Var-image-pairs is not a list of pairs, the first element of which is a variable to be bound and the second element of which is an appropriately-sized image array.

Function Index