5

Annotated Values and Active Values

In the previous chapters, IVs, CVs, and their properties have been treated as passive entities without structure. ***Annotated values*** are a way of associating behavior and annotations with variables. In keeping with the object oriented programming style of LOOPS, these annotations are objects. Annotation objects are called ***active values***. When a variable containing an annotated value is accessed, a message is sent to the active value. This mechanism is dual to the notion of messages: messages are a way of telling objects to perform operations, which can change their variables as a side effect; active values are a way of accessing variables, which can send messages as a side effect.

This chapter first describes the structure and implementation of annotated values. Functions for explicitly dealing with annotated values are documented. Then the class `ActiveValue` is introduced and the standard protocol for active values is described. Next, the standard subclasses of `ActiveValue` are explained.

5.1. Annotated Values

LOOPS defines a new INTERLISP data type called `annotatedValue`. Each `annotatedValue` contains a single field. This field contains an object, the annotated value's active value. The standard variable access functions described in previous chapters (`GetValue`, `PutValue`, `GetClassValue`, `PutClassValue`) treat values that are annotated values specially. `GetValue` and `GetClassValue` do not return the annotated value. Instead, they send the contained active value a message, and return the result of that message. Similarly, if the current value of a variable is an annotated value, `PutValue` and `PutClassValue` operate by sending the contained active value a message.

`type? annotatedValue` *value* [*Macro*]

Returns true if *value* is an annotated value, false otherwise. This is the standard way to test to see if a value is an annotated value.

`create annotatedValue annotatedValue` ← *object* [*Macro*]

Creates a new annotated value with active value *object*. No checking of *object* is performed.

`fetch annotatedValue of` *value* [*Macro*]

Returns the active value contained in the annotated value *value*. If *value* is not an annotated value, generates an error.

`replace annotatedValue of` *value* `with` *object* [*Macro*]

Replaces the active value contained in the annotated value *value* with *object*. If *value* is not an annotated value, generates an error. No checking of *object* is performed.

`←AV` *av selector . args* [*Macro*]

`←AV` is a message sending form that can be used with annotated values. `(←AV` *av selector . args*`)` → `(← (fetch annotatedValue of` *av*`)` *selector . args*`)` .

`AnnotatedValue` [*Class*]

Sometimes people forget to extract the active value from an annotated value, and they end up trying to use an annotated value as an object. Using the `LispDataType` feature, LOOPS takes care of this for you. Annotated values are considered to belong to the LOOPS class `AnnotatedValue`. If you send a message to an annotated value, the behavior is found in the class `AnnotatedValue`. There, the method for `MessageNotUnderstood` forwards the message off to the contained active value. Similarly, if you attempt to get an IV from an annotated value, the get ends up happening to the wrapped active value.

5.2. The Abstract Class ActiveValue

Active values follow a standard protocol that allow them to be used inside of annotated values.

In the description of methods for active values, the arguments *containingObj*, *varName*, *propName*, and *type* are used to describe the variable containing the active value. *type* is one of `IV`, `CV`, or `NIL` : a *type* of `IV` or `NIL` indicates that the variable is an instance variable or an instance variable property of *containingObj*; a *type* of `CV` indicates a class variable or class variable property of *containingObj*. If *propName* is `NIL`, the variable is either an IV or a CV, otherwise it is an IV or CV property with name *propName*. *containingObj* is the instance or class that contains the variable.

`ActiveValue` [*Abstract class*]

The class `ActiveValue` captures the protocol followed by all active value objects. `ActiveValue` is an abstract class, so you cannot make instances of `ActiveValue`. Specializations of `ActiveValue` need to specialize the `GetWrappedValueOnly` and `PutWrappedValueOnly` methods. Methods that you want to specialize include `AVPrintSource`, `GetWrappedValue`, `PutWrappedValue`, `WrappingPrecedence`, and `CopyActiveValue`.

5.2.1 Displaying Annotated Values

← *self* `AVPrintSource` [*ActiveValue method*]

An annotated value determines how it will print out by sending the `AVPrintSource` message to the its active value. This message returns a form suitable for use by the INTERLISP function `DEFPRINT`. The result should be a pair of the form `(`*item1* `.` *item2*`)`. *item1* will be printed using `PRIN1`, and then *item2* will be printed by `PRIN2` (see the IRM description of `DEFPRINT` for more details).

The default method in `ActiveValue` returns the list

`("#." $AV` *className avNames* `(`*ivName value propName value ...*`) (`*ivName ...*`)...)`

which will cause the annotated value to print out as

`#.($AV` *className avNames* `(`*ivName value propName value ...*`) (`*ivName ...*`)...).`

className is the name of the class of the active value. *avNames* is a list of names of *self*; the last element of *avNames* is the uid of *self*. The lists `(`*ivName value propName value ...*`)` describe the state of the IVs of the active value. Note that the uid of the active value is included in the printed form, so the identity of the active value object can be recovered. In this way, different annotated values can share the same active value, and have this sharing maintained across a dump/load-up.

`$AV` *className avNames* `.` *ivForms* [*Special Form*]

`$AV` is used to reconstruct a dumped annotated value. It returns a new annotated value whose active value is reconstructed from the *avNames* and *ivForms*.

5.2.2 Fetching and Replacing Wrapped Values

← *self* `GetWrappedValue` *containingObj varName propName type* [*ActiveValue method*]

The `GetWrappedValue` message provides a way to perform arbitrary actions when a variable is read. When `GetValue` (or `GetClassValue`) finds an annotated value in an instance, it does not return the annotated value. Instead, it sends the contained active value the `GetWrappedValue` message and returns the result of this message.

The default method in `ActiveValue` sends the message `GetWrappedValueOnly` to *self*. If this value is an annotated value, it is triggered by sending it the `GetWrappedValue` message, and the result is returned; otherwise the value is returned with no further processing.

← *self* `GetWrappedValueOnly` [*ActiveValue method*]

Returns the immediate "local state" of the variable that is wrapped by the active value *self*. If this local state is a nested active value, it is not triggered. The default implementation causes an error by calling `SubclassResponsibility`.

← *self* `PutWrappedValue` *containingObj varName*
newValue propName type [*ActiveValue method*]

The `PutWrappedValue` message provides a way to perform arbitrary actions when a variable is set. When `PutValue` (or `PutClassValue`) attempts to replace an annotated value, it instead sends the contained active value the `PutWrappedValue` message.

The default method in `ActiveValue` checks to see if the current value is a nested active value by sending the GetWrappedValueOnly message to *self*. If the result is an annotated value, `PutWrappedValue` forwards the message on the the nested active value; otherwise it sends the message `PutWrappedValueOnly` to *self* and returns the result.

← *self* `PutWrappedValueOnly` *newValue* [*ActiveValue method*]

Replaces the immediate "local state" of the variable that is wrapped by the active value *self*. The current local state is replaced. If the current value is a nested active value, it is not triggered. The default implementation causes an error by calling `SubclassResponsibility`.

5.2.3 Inheriting Active Values

Typical implementations of `PutWrappedValue` store the new value in the active value. However, if the active value is shared among different instances all these instances would see this change. In particular, if the active value is inherited from the class of the instance, all other instances of the class would see this change. This behavior is usually not desired. The `GetWrappedValue` method of active values is also free to alter the internal state of the active value, causing the same problem. To get around this problem, the annotated value is first copied, and this copy is stored in the instance. The `CopyActiveValue` method implements this copying. When `GetValue` or `PutValue` finds no local value, it first checks to see if the current value is an annotated value inherited from the class. If it is, it sends `CopyActiveValue` to the active value, and stores the result in the instance. The put or get then proceeds.

← *self* `CopyActiveValue` *annotatedValue* [*ActiveValue method*]

annotatedValue is an annotated value that surrounds *self*. `CopyActiveValue` should return a copy of *annotatedValue*, containing a copy of *self*. It is possible, and in some cases desirable, for an implementation of `CopyActiveValue` to return *annotatedValue*.

The default behavior returns a new annotated value wrapped around a copy of *self*. IV values of self are not copied, the values are shared with the copy, except that IVs of self that contain annotated values are copied using the `CopyActiveValue` message.

5.2.4 Adding and Deleting Annotations

← *self* `AddActiveValue` *containingObj varName*
propName type annotatedValue [*ActiveValue method*]

Adds the annotated value *annotatedValue* to the variable specified by *containingObj*, *varName*, *propName*, and *type*. If *annotatedValue* is not specified or is NIL, *annotatedValue* defaults to a newly created annotated value containing the active value *self*. If the variable is already an annotated value, the `AddActiveValue` method uses the `WrappingPrecedence` message (below) to

determine if *annotatedValue* should be nested in the current annotated value or wrapped around it. The method returns *annotatedValue*.

← *self* `WrappingPrecedence` [*ActiveValue method*]

Specifies where an annotated value containing *self* should be added to an existing annotated value. `T` means that this active value must go on the outside of any other annotated values. `NIL` means it must go on the inside. A number specifies a precedence: active values with larger `WrappingPrecedence` values go outside ones with smaller `WrappingPrecedence` values. If two active values have the same (numeric) `WrappingPrecedence`, the order is not determined. The default implementation of `WrappingPrecedence` returns `100`.

← *self* `DeleteActiveValue` *containingObj varName propName type* [*ActiveValue method*]

Finds the first annotated value on the variable specified by *containingObj*, *varName*, *propName*, and *type* that has *self* as its active value and deletes it from that variable. Returns that annotated value if one was found, `NIL` otherwise.

← *self* `ReplaceActiveValue` *newVal containingObj*
varName propName type [*ActiveValue method*]

It is sometimes desirable to replace an annotated value in a variable with some new value. (← *self* `ReplaceActiveValue` *newVal containingObj varName propName type*) replaces the annotated value containing *self* in the variable described by *containingObj*, *varName*, *propName*, and *type* with the new value *newVal*.

5.2.4 Manipulating Active Values

Some programs need to explicitly test and trigger active values. The following functions can be used to access IVs and CVs without triggering active values.

`GetValueOnly` *object varName propName* [*Function*]

`GetValueOnly` is the same as `GetValue`, except that `GetValueOnly` does not trigger any active values. `GetValueOnly` returns the immediate value of the variable. If this is not an annotated value, `GetValueOnly` returns the same value as `GetValue`. If there is no local value, the inherited value is returned. See also the function `GetIVHere`.

`GetClassValueOnly` *object varName propName* [*Function*]

`GetClassValueOnly` is the same as `GetClassValue`, except that `GetClassValueOnly` does not trigger any active values. `GetClassValueOnly` returns the immediate value of the variable. If this is not an annotated value, `GetClassValueOnly` returns the same value as `GetClassValue`. *object* can be either an instance or a class.

`ObjRealValue` *object varName value propName type* [*Macro*]

If *value* is not an annotated value returns *value*, otherwise returns the value of `(←AV GetWrappedValue` *object varName propName type*`)`. This macro is used by GetValue and

GetClassValue to trigger active values, and can be used by programs that explicitly test for active values.

`PutValueOnly` *object varName newValue propName* [*Function*]

`PutValueOnly` is the same as `PutValue`, except that `PutValueOnly` does not trigger any active values. `PutValueOnly` replaces the immediate value of the variable with *newValue*, even if the old value is an annotated value.

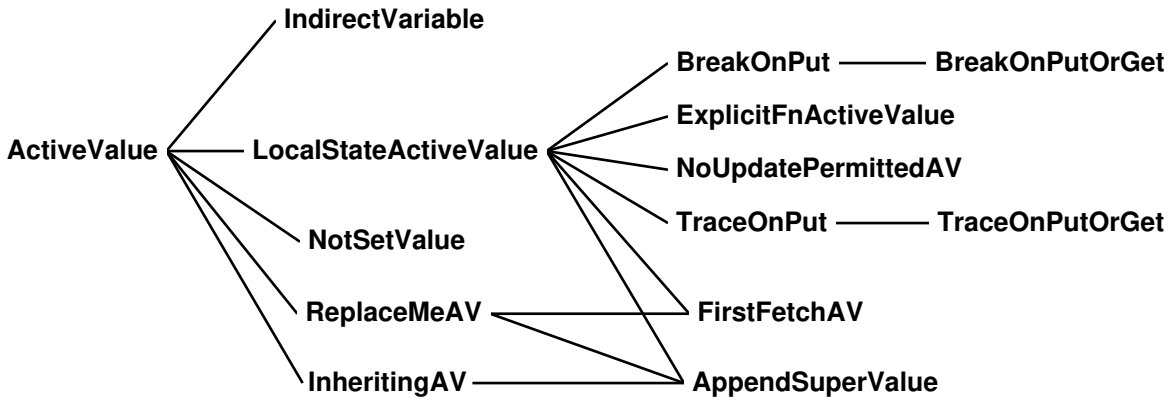
`PutClassValueOnly` *object varName newValue propName* [*Function*]

`PutClassValueOnly` is the same as `PutClassValue`, except that `PutClassValueOnly` does not trigger any active values. `PutClassValueOnly` replaces the immediate value of the variable with *newValue*, even if the old value is an annotated value. *object* can be either an instance or a class.

← *self* `HasAV?` *av* [*ActiveValue method*]

Returns true if the active value (or annotated value) *av* is nested inside in the active value self.

5.3. Specializations of ActiveValue



The ActiveValue Class Hierarchy

5.3.1 NotSetValue and Variable Inheritance

`NotSetValue` [*Variable*]

LOOPS uses annotated values to trigger IV inheritance. When an instance is created, its IVs are initialized to contain (the value of) `NotSetValue`. `NotSetValue` is an annotated value whose active value is the prototype instance of the class `NotSetValue`. The class `NotSetValue` specializes the

default `ActiveValue` protocol to trigger IV inheritance. In this way `GetValue` does not need to do any special check to see if a value needs to be inherited — all it needs to do is see if the value is an annotated value. Note that `GetValueOnly` does need to do a special check for `NotSetValue`, but see the function `GetIVHere`.

`NotSetValue` *form* [*Macro*]

Returns true if *form* evaluates to `NotSetValue`, otherwise false. (`NotSetValue` *form*) → (`EQ` *form* `'NotSetValue`). This is the approved way of testing a value to see if it is `NotSetValue`.

← *self* `AVPrintSource` [*NotSetValue method*]

Returns the pair (`"#." . NotSetValue`). This causes (the value of) `NotSetValue` to print out as `#.NotSetValue`. This will be read in as the value of the variable `NotSetValue`.

← *self* `GetWrappedValue` *containingObj varName propName type* [*NotSetValue method*]

If *type* is `NIL` or `IV`, this evaluates (← *containingObj* `IVValueMissing` *varName propName* `'GetValue`) and returns the result; if *type* is `CV`, evaluates (← *class* `CVValueMissing` *varName propName* `'GetValue`) (where *class* is the class of *containingObj* if *containingObj* is an instance, else *containingObj* if it is a class) and returns the result; otherwise an error is generated. See the methods `IVValueMissing` and `CVValueMissing` on the class `Object`.

← *self* `PutWrappedValue` *containingObj varName*
newValue propName type [*NotSetValue method*]

If *type* is `NIL` or `IV`, this evaluates (← *containingObj* `IVValueMissing` *varName propName* `'PutValue` *newValue*) and returns the result; if *type* is `CV`, evaluates (← *class* `CVValueMissing` *varName propName* `'PutValue` *newValue*) (where *class* is the class of *containingObj* if *containingObj* is an instance, else *containingObj* if it is a class) and returns the result; otherwise an error is generated. See the methods `IVValueMissing` and `CVValueMissing` on the class `Object`.

← *self* `CopyActiveValue` *annotatedValue* [*NotSetValue method*]

Returns `#.NotSetValue`. There is only one `NotSetValue`.

← *self* `WrappingPrecedence` [*NotSetValue method*]

Returns `NIL`. `#.NotSetValue` must always be on the inside of any sequence of nested active values.

`GetIVHere` *object varName propName* [*Function*]

If *propName* is `NIL` and there is a local value for the IV *varName* in the instance *object*, that value is returned. If *propName* is not `NIL` and there is a local value for the IV property *propName* of the IV *varName* in the instance *object*, that value is returned. Otherwise, if *propName* is `NIL` `GetIVHere` returns `#.NotSetValue`, and if *propName* is not `NIL` `GetIVHere` returns (the value of) `NoValueFound`.

`GetCVHere` *object varName propName* [*Function*]

object must be a class. Returns the value of the class variable that is found in the class *object*. If none is found, then returns `#.NotSetValue`.

`GetClassIV` *class varName propName* [*Function*]

Returns the default value or property value of the instance variable *varName* in the class *class*.

`PutClassIV` *class varName newValue propName* [*Function*]

Stores *newValue* as the default value or property value of teh instance variable *varName* in the class *class*. If *varName* is not already local to the class, this will cause an error. Returns *newValue*.

5.3.2. Indirect Variables

In some applications it is important to be able to access values indirectly from other instances. For example, Steele [Steele80] has recommended this as an approach for implementing equality constraints.

`IndirectVariable` [*Class*]

Mumble.

5.3.3. ReplaceMeAV

The active value mixin `ReplaceMeAV` can be used when an active value should be replaced when a variable is first set.

`ReplaceMeAV` [*Abstract class*]

Mumble.

5.3.4. LocalStateActiveValue

Many kinds of active values explicitly store the "real" value of the variable in an IV of the active value.

`LocalStateActiveValue` [*Abstract class*]

Mumble.

5.3.5. InheritingAV

Some kinds of active values want to compute a value based on what would have been inherited if the active value had not been present. For example, it might be desired to append items onto an inherited value (see the class `AppendSuperValue`).

`InheritingAV` [*Abstract class*]

Mumble.

5.3.6. FirstFetchAV

Mumble.

`FirstFetchAV` [*Class*]

Mumble.

5.3.7. Breaking and Tracing Variable Access

Mumble.

`BreakOnPut` [*Class*]

Mumble.

`BreakOnPutOrGet` [*Class*]

Mumble.

`TraceOnPut` [*Class*]

Mumble.

`TraceOnPutOrGet` [*Class*]

Mumble.

`UnBreakIt` *self varName propName type* [*Class*]

Mumble.

5.3.8. NoUpdatePermittedAV

The active value class `NoUpdatePermittedAV` can be used to prevent a value from being updated.

`NoUpdatePermittedAV` [*Class*]

Mumble.

5.3.9. AppendSuperValue

The active value class `AppendSuperValue` can be used to append data to inherited values.

`AppendSuperValue` [*Class*]

Mumble.

5.3.10. ExplicitFnActiveValue

`ExplicitFnActiveValue` [*Class*]

`ExplicitFnActiveValue` explicitly store functions that will be triggered when the variable is fetched or replaced. They have three IVs: `localState`, `getFn`,and `putFn`. The `localState` is the "real" value of the variable (possibly a nested active value), the `getFn` and `putFn` are names of functions that are applied with standard arguments by the GetWrappedValue and PutWrappedValue methods. The `getFn` and `putFn` are called with arguments *containingObj*, *varName*, *oldOrNewValue*, *propName*, *activeValue*, and *type*. `ExplicitFnActiveValue` active values print out as `#.($A` *localState getFn putFn*`)`, where the *localState*, *getFn*, and *putFn* are the values of the corresponding IVs of the active value.

5.4. Compatibility with older versions

The following existed in older versions of LOOPS, which had a different implementation of active values. They are provided for compatibility purposes only. New programs should not use them. They are not fully supported, and will not exist in future releases. The current implementations of these use the new active values. They are fully compatible with the older versions except where noted.

5.4.1. Old Style Active Values

LOOPS used to combine the notion of annotated value and active value. Variable annotations were instances of the INTERLISP datatype `activeValue`.

`activeValue` [*Record*]

In this version of LOOPS, the record `activeValue` is an access record that converts the three fields of the old active values to appropriate functions for accessing annotated values. Forms like `(type?` `activeValue` *form*`)` and `(fetch localState of` *activeValue*`)` will do the right thing. Reading in old style active values automatically converts them to annotated values wrapping an instance of the class `ExplicitFnActiveValue`.

`GetLocalState` *av self varName propName type* [*Function*]

Works just like in the old LOOPS.

`PutLocalState` *av newValue self varName propName type* [*Function*]

Works just like in the old LOOPS.

`GetLocalStateOnly` *av* [*Function*]

Works just like in the old LOOPS.

`PutLocalStateOnly` *av newValue* [*Function*]

Works just like in the old LOOPS.

`ReplaceActiveValue` *av newVal self varName propName type* [*Function*]

Works just like in the old LOOPS.

`MakeActiveValue` *self varOrSelector newGetFn newPutFn newLocalSt propName type* [*Function*]

Works just like in the old LOOPS, except that the interpretation of *newLocalSt* is different. `MakeActiveValue` ignores the value of *newLocalSt*, and always creates a new active value. This is the behavior that the old `MakeActiveValue` produced when *newLocalSt* was `Embed`.

5.4.2. GetFns and PutFns

`DefAVP` *fnName putFlg* [*Function*]

Works just like in the old LOOPS.

`NoUpdatePermitted` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`FirstFetch` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`GetIndirect` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`PutIndirect` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`ReplaceMe` *self varname oldOrNewValue propName activeValue type* [*Function*]

Works just like in the old LOOPS.

`AtCreation` *self varname oldOrNewValue propName activeValue type* [*Function*]

No longer works. Instead, you can use either the `FirstFetch` function, or the `:initForm` property of IVs.

5.5. Summary of Variable Access Functions

The following tables summarizes the available functions for variable access.

	Inherit/Trigger	**Inherit/Don't Trigger**	**Don't Inherit/Don't Trigger**
from instances			
IV	`GetValue` `PutValue`	`GetValueOnly` `PutValueOnly`	`GetIVHere`
CV	`GetClassValue` `PutClassValue`	`GetClassValueOnly` `PutClassValueOnly`	<n.a.> <n.a.>
from classes			
IV	<n.a.> <n.a.>	`GetClassIV` `PutCIVHere`	`GetClassIVHere` `PutClassIV`
CV	`GetClassValue` `PutClassValue`	`GetClassValueOnly` `PutClassValueOnly`	`GetCVHere` `PutCVHere`