Association of properties to definitions

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.9 $

Table of Contents

The input for a text processor usually describes objects that have certain properties, e.g. named entities like variables in a program, or fields of a data base. Their properties are determined, used, or checked according to the context in which an object occurs. An object may occur several times in the input. The occurrences are mapped to a unique identification, a key (see Section “Name Analysis Library” in *Name analysis according to scope rules*). Properties are associated and accessed via those keys. Properties are represented by values of certain types. The Eli tool PDL is used to generate functions that store property values in a data base of the language processor and that retrieve values from it.

The first section describes how the modules are instatiated and used, the others describe the modules contained in this library:

`Usage` Common Aspects of Property Modules

`OccCnt` Count Occurrences of Objects

`SetFirst` Set a Property at the First Object Occurrence

`Unique` Check for Unique Object Occurrences

`FirstOcc` Determine First Object Occurrence

`ObjCnt` Map Objects to Integers

`Kind` Associate Kinds to Objects

`KindSet` Associate Sets of Kinds to Objects

`Reflex` Reflexive Relations Between Objects

`PropLib` Some Useful PDL Specifications

`Defer` Deferred Property Association

The modules of this library can solve a large variety of tasks. E.g. the `OccCnt` module enumerates occurrences of each object. Its result can be used to check whether an identifier is multiply defined. It can also be used to trigger an output operation exactly once for each object. Such an output may say how often the identifier occurs in the text, or it may be a declaration of the identifier in the target code.

1 Common Aspects of Property Modules

The use of any module of this library requires that objects are identified by keys as computed by the consistent renaming modules.

All modules of this library, except `SetFirst`, `Reflex`, and `PropLib`, are instantiated by the same pattern:

```
$/Prop/ModuleName.gnrc+instance=NAME +referto=KEY :inst
```

for example

```
$/Prop/OccCnt.gnrc+instance=Var +referto=CtrlVar :inst
```

The `instance` parameter is used to distinguish several instances of a module that are used in one specification. If only one instance of a module is used the parameter can be omitted. The `referto` parameter is used to specify the name of the `Key` attribute, `CtrlVarKey` in the example above. The value must be the same as that of the `referto` parameter specified for the instance of the consistent renaming module which computed the `Key` attribute. (The `referto` parameter is usually omitted, unless there are symbols that have more than one `Key` attribute.)

The instantiation of the modules `SetFirst`, `Reflex`, and `PropLib` is described in the corresponding section. The modules `PropLib` and `Reflex` provide some useful PDL operations on definition table entries. All other modules of this library provide some computational role to be used in `.lido` specifications. The following applies only to these modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

The computations provided by each module ensure that properties are not accessed before they are set. For this purpose each module provides a computational role named `NAMERangeModName`, where `ModName` is the name of the module and `NAME` is the value of the `instance` parameter. (Exception: in the module `OccCnt` it is named `NAMERangeCnt`.) The root of the grammar automatically inherits this role. Hence, it need not be used in usual cases. The condition that all properties are set is provided by an attribute of the range symbol. It may be used as a precondition for computations which rely on that fact.

In seldom cases it may be necessary that symbols other than the grammar root inherit that range role in order to avoid cyclic dependencies between computations: if the computation of a property value in one range of the program depends on the access of a property in another (e.g. enclosing) range.

Note: The computations of these modules identify program objects by definition table keys. Hence, ranges specified for the computation of the keys by a unique renaming module, e.g. `RangeScope`, are irrelevant for these modules here.

In our running example we use the `OccCnt` module to check for multiply defined identifiers. As it is the first use of this module we can ommit the `instance` parameter. Since we omitted the `referto` parameter in the instance of the consistent renaming module that computes the keys, we omit it here, too:

```
$/Prop/OccCnt.gnrc :inst
```

The central computations of each module are provided by one or several computational roles, e.g. `NAMECount` and `NAMETotalCnt` in case of the `OccCnt` module. These roles are usually associated to grammar symbols representing identifier occurrences. In general they may be associated to any symbol that has a `Key` attribute.

In order to check for multiply defined identifiers in our running example both the `Count` role and the `TotalCnt` role is associated to defining identifier occurrences. As there are several symbols representing defining identifier occurrences which all have to be checked in the same way, we introduce a new role `MultDefChk` that comprises the necessary computations:

```
SYMBOL MultDefChk    INHERITS Count, TotalCnt END;

SYMBOL DefIdent      INHERITS MultDefChk END;
SYMBOL ClassDefIdent INHERITS MultDefChk END;
SYMBOL ModDefIdent   INHERITS MultDefChk END;
```

2 Count Occurrences of Objects

The computations of this module enumerate certain occurrences of objects represented by symbols that have a `Key` attribute. That are usually certain occurrences of identifiers.

The information computed by the module may be used for different purposes, e.g. for statistics about the input text, checks for unique occurrences, computations at the first or the last occurrence, etc.

The module is instantiated by

```
$/Prop/OccCnt.gnrc+instance=NAME +referto=KEY :inst
```

The module provides two computational roles, `NAMECount` and `NAMETotalCnt`. The roles may be associated to one or several grammar symbols.

The role `NAMERangeCnt` is automatically associated to the grammar root. It usually need not be used. It is not intended to provide separate counting in different parts of the tree (see Chapter 1 [Usage], page 3).

Let `k` be a key, then all occurrences of `k` in a `NAMECount` context are enumerated in left-to-right depth-first order. The attribute `NAMECount.NAMECnt` is the number of `k`'s occurrence with respect to this order. The total number of occurrences of `k` is associated as the property `NAMECnt` to `k`. The role `NAMETotalCnt` makes it available as an attribute `NAMETotalCnt.NAMETotalCnt`. If the `NAMECnt` property is accessed directly in user's computations, those have to state `NAMERangeCnt.GotNAMECnt` as precondition.

In (see Chapter 1 [Usage], page 3), we explained how to associate these rule to grammar symbols in order to check for multiply definitions:

```
SYMBOL MultDefChk    INHERITS Count, TotalCnt END;

SYMBOL DefIdent      INHERITS MultDefChk END;
SYMBOL ClassDefIdent INHERITS MultDefChk END;
SYMBOL ModDefIdent   INHERITS MultDefChk END;
```

The check is completed by using the results of this module in computations associated to `MultDefChk`:

```
SYMBOL MultDefChk COMPUTE
   IF (GT (THIS.TotalCnt, 1),
   message (ERROR,
            CatStrInd ("identifier is multiply defined: ",
                       THIS.Sym),
            0, COORDREF));
END;
```

The following example demonstrates a different application of this module. Assume we want to print how often each object in a program is referenced in some context. Hence, any identifier occurrence has to be counted. To avoid that this module application collides with the previous, we have to use a different instance of the `OccCnt` module:

```
$/Prop/OccCnt.gnrc +instance=Prnt:inst
```

That is easily achieved by associating the `Count` and the `TotalCnt` roles to the role `IdentOcc` previously introduced in our running example:

```
SYMBOL IdentOcc  INHERITS PrntCount, PrntTotalCnt COMPUTE
   IF (EQ (THIS.PrntCnt, 1),
   printf ("identifier %s occurs %d times\n",
           StringTable (THIS.Sym), THIS.PrntTotalCnt));
END;
```

3 Set a Property at the First Object Occurrence

This module associates values of type `TYPE` as property `NAME` to objects identified by keys. The property is set at most once at the first occurrence of the object which has the `NAMESetFirst` role. The module may for example be used to associate source coordinates of defining identifier occurrences to objects.

The module instantiation differs from the usual pattern for this library:

```
$/Prop/SetFirst.gnrc+instance=NAME +referto=TYPE :inst
```

Note: This module is not applicable to symbol occurrences that do not have an attribute named `Key`, e.g. due to the use of the `referto` parameter for a consistent renaming module.

Values are associated in the first `NAMESetFirst` context with respect to left-to-right depth-first tree order. The property value to be set has to be provided by a user's computation for the attribute `NAMESetFirst.NAME` (in any, not only the first `NAMESetFirst` context).

The role `NAMERangeSetFirst` is automatically associated to the root of the grammar (see Chapter 1 [Usage], page 3). The attribute `NAMERangeSetFirst.GotNAME` has to be used as a precondition for computations which access the `NAME` property to guarantee that the property is set.

If we want to associate the source coordinates of defining identifier occurrences to object keys in our running example we instantiate this module using the coordinate type `CoordPtr` exported by the error module, and the property name `DefPt`

```
$/Prop/SetFirst.gnrc+instance=DefPt +referto=CoordPtr :inst
```

Then the roles of this module are associated to our grammar symbols:

```
SYMBOL DefPoint      INHERITS DefPtSetFirst COMPUTE
   SYNT.DefPt = COORDREF;
END;

SYMBOL DefIdent      INHERITS DefPoint END;
SYMBOL ClassDefIdent INHERITS DefPoint END;
SYMBOL ModDefIdent   INHERITS DefPoint END;
```

The property computed this way may for example be used to check whether an identifier occurs in an applied context before its definition, as required e.g. in Pascal. For that purpose a function `CoordLess` is used to compare coordinates. (It is provided by the `PropLib` module, which is automatically instantiated when this module is used.)

```
SYMBOL ChkBeforeDef COMPUTE
   IF (CoordLess (COORDREF, GetDefPt (THIS.Key, COORDREF)),
   message (ERROR,
            CatStrInd ("identifier occurs before its definition: ",
                       THIS.Sym),
            0, COORDREF))
   DEPENDS_ON INCLUDING DefPtRangeSetFirst.GotDefPt;
END;
```

The role `ChkBeforeDef` is then associated to all grammar symbols that represent applied identifer occurreces which shall be checked.

4 Check for Unique Object Occurrences

This module associates a boolean property `NAMEUnique` to object keys. It has the value 1 if the object occurs only once in the SYMBOL context `NAMEUnique`. It has the value 0 if it occurs more than once in the `NAMEUnique` context; otherwise the property is not set. The final value of the property is obtained by the attribute `NAMEUnique.NAMEUnique`, e.g. used to issue a message indicating multiple occurrences. (The same task can be solved using the more general module, See Chapter 2 [OccCnt], page 5.)

The module is instantiated by

```
$/Prop/Unique.gnrc+instance=NAME +referto=KEY :inst
```

The role `NAMERangeUnique` is automatically associated to the grammar root (see Chapter 1 [Usage], page 3).

The multiply defined check for our running example, as explained for the `OccCnt` module (see Chapter 2 [OccCnt], page 5), can be also achieved by:

```
SYMBOL MultDefChk    INHERITS Unique COMPUTE
   IF (NOT (THIS.Unique),
   message (ERROR,
            CatStrInd ("identifier is multiply defined: ",
            THIS.Sym),
            0, COORDREF));
END;
```

5 Determine First Object Occurrence

This module determines whether a `NAMEFirstOcc` occurrence of an object is the first one in left-to-right depth-first tree order. The result is obtained by the attribute `NAMEFirstOcc.IsNAMEFirstOcc` that has the value 1 if it is the first occurrence, 0 otherwise. The computations of the module use the property named `NAMEFirstOcc`. (The same task can be solved using the more general module `OccCnt`, See Chapter 2 [OccCnt], page 5.)

The role `NAMERangeFirstOcc` is automatically associated to the grammar root (see Chapter 1 [Usage], page 3).

The module is instantiated by

```
$/Prop/FirstOcc.gnrc+instance=NAME +referto=KEY :inst
```

6 Map Objects to Integers

This module computes a mapping of object keys to non-negative numbers A number is associated as property named `NAMEObjNo` to each object exactly once. In each `NAMERangeObjCnt` subtree the numbers are chosen separately starting from 0 incrementing by 1 (changeable default). `NAMERangeObjCnt` is automatically associated to the grammar root.

The module can be used to just count the objects that occur in a range, to prepare for generating unique identifier names on output, or to map objects of a range to addresses that are incremented by a certain value.

The module is instantiated by

```
$/Prop/ObjCnt.gnrc+instance=NAME +referto=KEY :inst
```

The role `NAMEObjCnt` has to be associated to grammar symbols such that all objects that should be considered have an occurrence in such a context. The attribute `NAMEObjCnt.NAMEObjNo` is the number the object is mapped to.

`NAMERangeObjCnt` is automatically associated to the grammar root (see Chapter 1 [Usage], page 3). The attribute `NAMERangeObjCnt.NAMETotalObjNo` is the total number of objects in that range. The ranges may be nested. The mapping starts anew for each range node. The mappings of inner ranges do not contribute to outer ranges.

The default start value is 0. It can be changed by overriding the computation of `NAMERangeObjCnt.NAMEInitObjCnt`. The computation of `NAMEObjCnt.NAMEIncrementCnt` can be overridden to change the default increment value of 1.

If the `ObjNo` property is accessed by user's computations, they have to state `NAMERangeObjCnt.NAMETotalObjNo` as a precondition.

We demonstrate an application of this module by mapping the objects of out running example to unique numbers, in order to print unique names as they resulted from the consistent renaming task. For that purpose the grammar root can be chosen for the range role. The `ObjCnt` is simply attached to the `IdentOcc` role which represents any identifier occurrence in our example:

```
SYMBOL Program COMPUTE
  printf ("the program references %d different objects\n",
          THIS.TotalObjNo);
END;
SYMBOL IdentOcc INHERITS ObjCnt COMPUTE
  printf ("object %s%d referenced in line %d\n",
          StringTable (THIS.Sym), THIS.ObjNo, LINE);
END;
```

7 Associate Kinds to Objects

Objects in an input text are often classified to belong to one of several kinds, e.g. variables, procedures or labels in programming languages. They may occur in different contexts which determine their kind, require that they belong to a certain kind, or select different computations depending on their kind. Such a classification is often the part of the type analysis task.

This module can be used for any unique classification of objects which is encoded by integral values.

If Objects may belong to more than one kind, or occurrences allow for objects of several kinds the module `KindSet` (see Chapter 8 [KindSet], page 17) should be used instead of this one.

The module is instantiated by

```
$/Prop/Kind.gnrc+instance=NAME +referto=KEY :inst
```

This module associates a property named `NAMEKind` of type `int` to objects. Two computational roles `NAMESetKind` and `NAMEGetKind` are provided.

In a context `NAMESetKind` the `NAMEKind` property of the object is set to the value of the attribute `NAMESetKind.NAMEKind`, which has to be provided by a user's computation. In case that different `Kind` values are stated for one object in some `NAMESetKind` contexts the property value 0 named `IntMultiple` is associated.

In a context `NAMEGetKind` the property is accessed and supplied by the attribute `NAMEGetKind.HasNAMEKind`. It can be used to check if it is the required kind, or if the kind is ambigously set (`IntMultiple`), or if the kind is not set at all (value -1 named (`IntNone`).

The roles `NAMESetKind` and `NAMEGetKind` may be associated to the same grammar symbol. That is necessary if kinds are determined by applications of objects rather than by definitions, or if a language does not distinguish between defining and applied occurrences.

`NAMERangeKind` is automatically associated to the grammar root (see Chapter 1 [Usage], page 3). If the `NAMEKind` property is accessed in other user's computations, those have to state `NAMERangeKind.GotNAMEKind` as precondition.

`IntMultiple` and `IntNone` are defined in a file `KindBad.h`. If their encoding is inconvenient for a particular implementation, that file may be overridden by a user supplied file having the name `KindBad.h`.

8 Associate Sets of Kinds to Objects

Objects in an input text are often classified to belong to one or more of several kinds, e.g. variables, procedures or labels in programming languages. They may occur in different contexts which determine their kind, require that they belong to a certain kind, or select different computations depending on their kind. Such a classification is often the part of the type analysis task.

This module can be used for any classification of objects which is encoded by non negative integral values.

The module is instantiated by

```
$/Prop/KindSet.gnrc+instance=NAME +referto=KEY :inst
```

The module uses sets of kinds implemented by values of type `unsigned int` provided by the `IntSet` module (see Section "Bit Sets of Integer Size" in *Bit Sets of Integer Size*). (That module is instantiated automatically with `referto` parameter `int`, and the `instance` parameter omitted.) Hence the largest code chosen for a kind value must be less than the number of bits of an `unsigned int` (16 or 32 implementation dependent).

This module associates a property named `NAMEKindSet` of type `IntSet` to objects. Three computational roles `NAMEAddKind`, `NAMEAddKindSet`, and `NAMEGetKindSet` are provided.

In a context `NAMEAddKind` the kind value of the attribute `NAMEAddKind.NAMEKind` is added to the set of kinds of the object. The attribute has to be provided by a user's computation.

Similarly in a context `NAMEAddKindSet` the `IntSet` value of the attribute `NAMEAddKindSet.NAMEKindSet` is united to the set of kinds of the object. The attribute has to be provided by a user's computation.

In a context `NAMEGetKindSet` the property is accessed and supplied by the attribute `NAMEGetKindSet.HasNAMEKindSet`. It can be used to compare the set of required kinds to the set of associated kinds using functions of the `IntSet` module.

The roles `AddKind` and `AddKindSet` must not be associated to the same grammar symbol. `GetKindSet` may be combined with one of them. That is necessary if kinds are determined by applications of objects rather than by definitions, or if a language does not distinguish between defining and applied occurrences.

`NAMERangeKindSet` is automatically associated to the grammar root (see Chapter 1 [Usage], page 3). If the `NAMEKindSet` property is accessed in other user's computations, those have to state `NAMERangeKindSet.GotNAMEKind` as precondition.

This module also provides three operations that modify `NAMEKindSet` properties stored in the definition module:

`InsertNAMEKindSet(k,i)` inserts element `i` into the set stored for key `k`, and yields the new set as result.

`UnionNAMEKindSet(k,s)` adds the set `s` to the set stored for key `k`, stores the result and returns it.

`IntersectNAMEKindSet(k,s)` intersects the set `s` with set stored for key `k`, stores the result and returns it.

We demonstrate the use of this module in our running example. It shall be analysed if each variable occurs at least once on the lefthand side of an assignment and on the righthand side. Hence, we introduce the kinds `VarAssigned` and `VarUsed`. A variable can have any set of the values depending on its occurrences. The values are named by a `.head` specification:

```
#define   VarAssigned    1
#define   VarUsed        2
```

In our tree grammar the two occurrences of variables can be distinguished for the symbol `Variable` rather than for the symbol `UseIdent`. Hence we propagate the `Key` attribute of `UseIdent` upto `Variable` and apply the module roles there:

```
SYMBOL Variable: Key: DefTableKey;
RULE: Variable ::= UseIdent COMPUTE
  Variable.Key = UseIdent.Key;
END;

RULE: Statement::= Variable '=' Expression ';' COMPUTE
  Variable.Kind = VarAssigned;
END;

RULE: Expression ::= Variable COMPUTE
  Variable.Kind = VarUsed
END;

SYMBOL Variable INHERITS AddKind END;
```

In the context of a variable declaration the set of kinds is checked using functions of the `IntSet` module:

```
SYMBOL DefIdent INHERITS GetKindSet END;
RULE: ObjDecl ::= TypeDenoter DefIdent COMPUTE
  IF (NOT (InIS (VarAssigned, DefIdent.HasKindSet)),
  printf ("variable %s declared in line %d is never assigned\n",
           StringTable (DefIdent.Sym), LINE));

  IF (NOT (InIS (VarUsed, DefIdent.HasKindSet)),
  printf ("variable %s declared in line %d is never assigned\n",
           StringTable (DefIdent.Sym), LINE));
END;
```

9 Reflexive Relations Between Objects

This module introduces properties that relate object keys pairwise to each other, e.g. a type and its pointer type.

The module is instantiated by

```
$/Prop/Reflex.gnrc+instance=NAME :inst
```

It defines a pair of properties `NAMETo` and `NAMEFrom` that have values of type `DefTableKey` that relate keys pairwise to each other. When the relation is established between two keys `kf` and `kt` `GetNAMETo (kf, NoKey) == kt` and `GetNAMEFrom (kt, NoKey) == kf` hold.

The relation is established by a call `ReflexNAMETo (kf)` that yields a new key `kt`, or by `ReflexNAMEFrom (kt)` that yields a new key `kf`. Any further such call yields the same key as result.

Typical applications of such relations are found in type analysis tasks: Types can be represented by keys. Assume `intKey` represents the type `int`, then a call `ReflexPointerTo (intKey)` yields a key representing a type `pointer to int`. Using the `Reflex` functions guarantee that there is exactly one key representing the type `pointer to int`. Here the module is instantiated with the generic parametr `+instance=Pointer`. The same pattern can be applied for other unary type constructors.

10 Some Useful PDL Specifications

This module specifies a set of useful generic PDL patterns. If such patterns are associated to a property specification PDL generates additional access functions for that property.

For example the PDL property specification

```
Size: int [SetGet, SetDiff];
```

allows to set the `Size` property using the functions `SetGetSize` and `SetDiffSize` besides the basic access functions provided by PDL.

The module does not have generic parameters. It is used by writing

```
$/Prop/PropLib.fw
```

in a `.specs` file.

It provides the following PDL patterns:

`SetGet`: The `SetGet` functions have same effect as the basic `Set` function. But the value which is set is also returned as result of the call.

`SetOnce`: The `SetOnce` functions have one value argument like the `Reset` functions. The given value is only set if that property is not yet set. The current value of the property is returned as result of the call.

`KReset`: The `KReset` functions have same effect as the basic `Reset` functions. But the key is returned as result of the call. By that means one can set several properties for one key using nested calls.

`VReset`: The `VReset` functions have same effect as the basic `Reset` functions. But the value which is set is also returned as result of the call.

`Trans`: The `Trans` functions are applicable for properties of type `DefTableKey`. They have only a key argument. A call `TransProp(k)` for a property `Prop` is recursively applied to the property value until a key is reached for which the property `Prop` is not set. that key is returned. The property chain must not be cyclic. E.g. if `GetTypeOf (a, NoKey) == b` and `GetTypeOf (b, NoKey) == c` and `GetTypeOf (c, NoKey) == NoKey`, then `TransTypeOf (a) == c`.

`SetDiff`: The `SetDiff` functions have two value arguments, like the `Set` functions. The first value argument is set if the property is not yet set. If the property has a value that differs from the first value argument, the property is set to the second value argument.

The module also provides comparison functions `CoordLess` and `CoordLessEqual` for source coordinates.

11 Deferred Property Association

This module implements the technique of deferred property association: Many languages have constructs that define an identifier to denote the same object as another, different identifier does. Properties accessed or set via the one key should yield the same results or effects as if the other key was used. Typical examples for such constructs are type definitions or constant definitions.

The module is instantiated by

```
$/Type/Defer.gnrc +referto=KEY :inst
```

The `referto` parameter modifies the names of `Key` attributes, and hence, has to be the same as the `referto` parameter used for the module instance that supplied those attributes.

The roles of this module relate keys to each other which represent the same object. That relation has to be acyclic. The properties are associated to the keys at the ends of those relation chains. A function is provided that walks down the chain when accessing a property from any of the related keys.

This technique also decouples the computations which establish the equivalence between keys from those which associate properties to keys. It avoids cyclic dependencies between computations in cases where properties of entities may be defined recursively, e.g. recursively defined types.

The property `Defer` implements the relation between keys described here. It should not be set otherwise than by using the `SetDeferId` role of this module.

Setting a property to a key that may be an end of a `Defer` chain should occur in the context of the `SetDeferProp` role.

If properties are accessed for a key `k` that may be on a `Defer` chain, the result of the call `TransDefer (k)` has to be used instead of the the key `k` itself, e.g. `GetKind (TransDefer (k), NoKind)`.

This module uses the `PropLib` module (See Section "Some Useful PDL Specifications" in *Association of properties to definitions*,) to obtain the `TransDefer` function.

The module provides the following computational roles:

`SetDeferId` is a role for a defining occurrence of an identifier. It establishes the `Defer` relation from `SetDeferId.|KEY|Key` to point to `SetDeferId.DeferredKey`. A lower or upper computation for `THIS.DeferredKey` has to be provided. An attempt to complete a `Defer` cycle is not executed.

`ChkSetDeferId` is a role that may be inherited by any identifier occurrence. It checks whether an attempt was made to complete a `Defer` cycle involving this key. The role should be inherited together with `SetDeferId`, if `Defer` cycles are not otherwise excluded.

`SetDeferProp` is a role that characterizes a context where properties may be set to a key at the end of a `Defer` chain. Computations that associate the properties have to establish the postcondition represented by the `VOID` attribute `SYNT.GotDeferProp`. The role provides a default computation for `SYNT.GotDeferProp` that states the empty postcondition.

`RootDefer` is inherited by the grammar root by default.

Index

S

T

U

V