Schöner Coden – PL/SQL analysieren mit PL/Scope

Sabine Heimsath, its-people GmbH

In der Schatzkiste der unbekannten Features der Oracle-Datenbank gibt es seit Version 11.1 ein Tool für Entwickler, das mit dem Release 12.2 noch spannender geworden ist: PL/Scope. Es handelt sich nicht um eine fertige Anwendung, die Code optimiert, aber es ermöglicht detaillierte Analysen, mit denen der Code verbessert werden kann. Außerdem lassen sich die Informationen nutzen, um einen Überblick über die Struktur des Codes zu bekommen. Der Artikel zeigt, welche Neuerungen mit 12.2 hinzugekommen sind und was man damit anfangen kann.

Eine Installation von PL/Scope ist nicht notwendig, wenn die Datenbank mindestens auf Version 11.1.0.7 ist. Für ältere 11er-Versionen sei auf die Doku verwiesen. Das Einzige, was zu tun ist, ist, PL/Scope mit „alter session set plscope_settings = ‚IDENTIFIERS:ALL'" für die aktuelle Session zu aktivieren beziehungsweise in 12.2 mit „alter session set plscope_settings = ‚IDENTIFIERS:ALL,STATEMENTS:ALL'".

Diese Einstellung kann auch im SQL Developer unter „Preferences > Database > PL/SQL-Compiler" vorgenommen werden. Wegen eines Bugs funktioniert dies allerdings nur, wenn die GUI auf Englisch eingestellt ist. Nach der Umstellung werden bei jedem Compile-Vorgang in dieser Session die Metadaten für alle Bezeichner und SQL-Statements der jeweiligen Compilation-Unit generiert. Die Einstellung der aktuellen Session lässt sich abfragen *(siehe Listing 1)*. Möchte man wissen, welche Objekte mit welchen Einstellungen kompiliert wurden, hilft die Abfrage „select owner, name, type, plscope_settings from all_plsql_object_settings".

```
select name, value
   from v$parameter
 where name = 'plscope_settings'
```

Listing 1

Die PL/Scope-Views

Interessanter wird es bei der Frage, wie das Ergebnis in den Views aussieht. Als Beispiel dient die kleine Prozedur „tables_out". PL/Scope nummeriert alle Verwendungen von Bezeichnern in der Reihenfolge ihres Auftretens durch *(siehe Abbildung 1)*. Eine Ausnahme gibt es: SQL-Statements werden in umgekehrter Richtung geparst, daher ist die Nummerierung hier umgedreht. *Abbildung 2* zeigt das Ergebnis der View „USER_IDENTIFIERS".

```
create or replace procedure hr.emps_out
  authid definer
is
  v_string varchar2(2000);
  v_int integer;
begin
  for rec in (select last_name, salary from employees)
  loop
    v_string := rec.last_name;
    v_int := rec.salary;
    sys.dbms_output.put_line(v_string || ' ' || sys.standard.to_char(v_int,99999));
  end loop;
end;
/
```

Abbildung 1: So sieht PL/Scope den Code: Die „USAGE_ID" wird in fortlaufender Reihenfolge vergeben

NAME	SIGNATURE	TYPE	USAGE	USAGE_ID	LINE	COL	USAGE_CONTEXT_ID	OBJECT_NAME	OBJECT_T...
EMPS_OUT	CB5CCED	PROCEDURE	DECLARATION	1	1	14	0	EMPS_OUT	PROCEDURE
EMPS_OUT	CB5CCED	PROCEDURE	DEFINITION	2	1	14	1	EMPS_OUT	PROCEDURE
V_STRING	5C35983	VARIABLE	DECLARATION	3	4	3	2	EMPS_OUT	PROCEDURE
VARCHAR2	FEE1E7E	CHARACTER DATATYPE	REFERENCE	4	4	12	3	EMPS_OUT	PROCEDURE
V_INT	8721790	VARIABLE	DECLARATION	5	5	3	2	EMPS_OUT	PROCEDURE
INTEGER	4EEE7F4	SUBTYPE	REFERENCE	6	5	9	5	EMPS_OUT	PROCEDURE
REC	20A90B4	ITERATOR	DECLARATION	7	7	7	2	EMPS_OUT	PROCEDURE
EMPLOYEES	85DA2A5	TABLE	REFERENCE	9	7	45	8	EMPS_OUT	PROCEDURE
SALARY	55D86C5	COLUMN	REFERENCE	10	7	33	8	EMPS_OUT	PROCEDURE
LAST_NAME	1687ACC	COLUMN	REFERENCE	11	7	22	8	EMPS_OUT	PROCEDURE
V_STRING	5C35983	VARIABLE	ASSIGNMENT	12	9	5	8	EMPS_OUT	PROCEDURE
REC	20A90B4	ITERATOR	REFERENCE	13	9	17	12	EMPS_OUT	PROCEDURE
V_INT	8721790	VARIABLE	ASSIGNMENT	15	10	5	8	EMPS_OUT	PROCEDURE
REC	20A90B4	ITERATOR	REFERENCE	16	10	14	15	EMPS_OUT	PROCEDURE
DBMS_OUTPUT	6CDB513	PACKAGE	REFERENCE	18	11	9	8	EMPS_OUT	PROCEDURE
PUT_LINE	720C63F	PROCEDURE	CALL	19	11	21	18	EMPS_OUT	PROCEDURE
V_STRING	5C35983	VARIABLE	REFERENCE	20	11	30	19	EMPS_OUT	PROCEDURE
STANDARD	26BE90B	PACKAGE	REFERENCE	21	11	53	19	EMPS_OUT	PROCEDURE
V_INT	8721790	VARIABLE	REFERENCE	22	11	70	21	EMPS_OUT	PROCEDURE

Abbildung 2: PL/Scope-Metadaten für die Bezeichner

SIGNATURE	TYPE	USAGE_ID	LINE	COL	USAGE_CONTEXT_ID	SQL_ID	TEXT
3AFC9CE	SELECT	8	7	15	7	0c8jruncvt7tp	SELECT LAST_NAME,

Abbildung 3: PL/Scope-Metadaten für SQL-Statements

Hier findet man zunächst den Namen des Bezeichners, dann eine global eindeutige Signatur (wichtig zum Beispiel zur Unterscheidung überladener Methoden oder bei mehrfacher Verwendung eines Variablennamens in unterschiedlichen Methoden desselben Packages), den Typ des Bezeichners und die – pro Compilation-Unit eindeutige – „USAGE_ID" jedes Bezeichners. „LINE" und „COL" geben die konkrete Fundstelle an und die „USAGE_CONTEXT_ID" die jeweils übergeordnete „USAGE_ID". Diese ist „0" für alle Compilation-Units, also Prozeduren, Funktionen, Packages, Trigger und Synonyme. Warum ist die „USAGE_ID" nicht lückenlos? Bis 12.1 war sie es, was daran lag, dass statische SQL-Statements, die im Code vorkamen, schlichtweg ignoriert wurden – wie auch Kommentare und Leerzeilen. Seit 12.2 werden diese Statements ebenfalls analysiert und können in der neu dazugekommenen View „USER_STATEMENTS" abgefragt werden. Hier finden wir auch unsere „USAGE_ID 8", die zum „SELECT"-Statement gehört *(siehe Abbildung 3)*.

Die Spalten sind größtenteils dieselben wie bei „USER_IDENTIFIERS"; sie umfassen eine globale Signatur, einen Typ (wie „SELECT", „UPDATE", „EXECUTE IMMEDIATE", „COMMIT", „SAVEPOINT", „OPEN" etc.), die „USAGE_ID", „LINE" und „COL" der Fundstelle sowie die „SQL_ID". Zwei gleiche SQL-Statements haben zwar eine unterschiedliche Signatur, lassen sich aber über dieselbe „SQL_ID" finden. Die View enthält noch weitere Spalten, die das jeweilige Statement detaillierter beschreiben. Es gibt zum Beispiel Informationen dazu, ob das Statement einen Hint beinhaltet, ob es „BULK COLLECT INTO" oder „FOR UPDATE" nutzt, ob es eine „RETURNING"-Klausel besitzt oder ob es Bindevariablen enthält.

IDENTIFIERS- und STATEMENTS-View

Über „UNION ALL" lassen sich die beiden Views so verbinden, dass sich über die hierarchische Beziehung zwischen „USAGE_ID" und „USAGE_CONTEXT_ID" ein vollständiger Baum bilden lässt. Dazu eine Anmerkung: Die Implementierung ist möglicherweise noch nicht ganz komplett. In unserem Beispiel fehlt die Funktion „TO_CHAR", was allerdings nicht zu einer Lücke bei „USAGE_ID" führt. Hingegen erzeugen die fehlenden Referenzen der Record-Variablen „rec.last_name" und „rec.salary" in Zeile 9 und 10 offensichtlich eine Lücke *(siehe Abbildung 4)*. Dies wird hoffentlich in kommenden Versionen behoben.

LINE	COL	IDENTIFIER_HIERARCHY	USAGE_ID	USAGE_CONTEXT_ID
1	14	PROCEDURE EMPS_OUT (DECLARATION)	1	0
1	14	PROCEDURE EMPS_OUT (DEFINITION)	2	1
4	3	VARIABLE V_STRING (DECLARATION)	3	2
4	12	CHARACTER DATATYPE VARCHAR2 (REFERENCE)	4	3
5	3	VARIABLE V_INT (DECLARATION)	5	2
5	9	SUBTYPE INTEGER (REFERENCE)	6	5
7	7	ITERATOR REC (DECLARATION)	7	2
7	15	SELECT 0c8jruncvt7tp (STMT)	8	7
7	22	COLUMN LAST_NAME (REFERENCE)	11	8
7	33	COLUMN SALARY (REFERENCE)	10	8
7	45	TABLE EMPLOYEES (REFERENCE)	9	8
9	5	VARIABLE V_STRING (ASSIGNMENT)	12	8
9	17	ITERATOR REC (REFERENCE)	13	12
10	5	VARIABLE V_INT (ASSIGNMENT)	15	8
10	14	ITERATOR REC (REFERENCE)	16	15
11	9	PACKAGE DBMS_OUTPUT (REFERENCE)	18	8
11	21	PROCEDURE PUT_LINE (CALL)	19	18
11	30	VARIABLE V_STRING (REFERENCE)	20	19
11	53	PACKAGE STANDARD (REFERENCE)	21	19
11	70	VARIABLE V_INT (REFERENCE)	22	21

Abbildung 4: Die hierarchische Darstellung der Bezeichner

```
select s.type, s.object_name, s.object_type, s.line
  from user_statements s
 where s.type in ('UPDATE','INSERT','MERGE')
  and exists (select *
         from user_identifiers i
         where s.usage_id = i.usage_context_id
         and s.object_name = i.object_name
         and s.object_type = i.object_type
         and i.name = 'FIRST_NAME'
         and i.type = 'COLUMN')
  and exists (select *
         from user_identifiers j
         where s.usage_id = j.usage_context_id
         and s.object_name = j.object_name
         and s.object_type = j.object_type
         and j.name = 'EMPLOYEES'
         and j.type = 'TABLE')
```

Listing 2: Stellen, an denen die Spalte „EMPLOYEES.FIRST_NAME" geschrieben wird

Beispiel 1: Wo eine bestimmte Tabellenspalte manipuliert wird

Fangen wir gleich mit einem Bespiel an, das die neue Funktionalität der SQL-Analyse nutzt. Die Fragestellung lautet: „Wo wird die Spalte FIRST_NAME der Tabelle EMPLOYEE geändert?" Normalerweise sollte dies nur an wenigen Stellen erfolgen oder noch besser über ein definiertes API, wie einige Evangelisten nicht müde

werden zu predigen, aber in der Praxis werden trotzdem nicht selten SQL-Statements direkt aufgerufen. Dies machen wir uns hier zunutze *(siehe Listing 2)*.

Im Sourcecode stellt man fest, dass tatsächlich an diesen Stellen die gesuchten Statements stehen. Diese können dann als Ausgangspunkt für weitere Analysen dienen *(siehe Abbildung 5)*. Nebenbei angemerkt: Die Zeile 13, die auch eine Referenz auf „FIRST_NAME" enthält, taucht übrigens nicht im Suchergebnis auf. Solche und ähnliche Referenzen wären mit einem klassischen „SELECT * FROM USER_SOURCE" nicht so einfach auszuschließen.

```
12  ,p_PHONE_NUMBER in EMPLOYEES.PHONE_NUMBER%type default null
13  ,p_FIRST_NAME in EMPLOYEES.FIRST_NAME%type default null
14  ,p_COMMISSION_PCT in EMPLOYEES.COMMISSION_PCT%type default null
15  ,p_MANAGER_ID in EMPLOYEES.MANAGER_ID%type default null
16  ) is
17  begin
18    pit.enter('ins');
19  insert into EMPLOYEES(
20  JOB_ID
21  ,EMPLOYEE_ID
22  ,SALARY
23  ,HIRE_DATE
24  ,DEPARTMENT_ID
25  ,LAST_NAME
26  ,EMAIL
27  ,PHONE_NUMBER
28  ,FIRST_NAME
29  ,COMMISSION_PCT
```

Abbildung 5: Der untersuchte Sourcecode mit Vorkommen von FIRST_NAME

Beispiel 2: Naming Conventions checken

Ein kurzes, aber beliebig zu verkomplizierendes Beispiel ist der Check von Namenskonventionen wie in *Listing 3*. Hier hilft die Einschränkung auf die Deklaration, denn dadurch werden nicht nur Doppelungen vermieden, sondern auch alle Bezüge zu Objekten außerhalb der betrachteten Compilation-Unit ausgeblendet, also Tabellen, Spalten, (Standard)-Packages und -Typen.

In *Abbildung 6* sieht man sehr schön, dass die Prozedur „ADD_JOB_HISTORY" und die Konstante „CNT" nicht den Konventionen entsprechen. Über die Verknüpfung von „USAGE_ID" und „USAGE_CONTEXT_ID" lassen sich natürlich wunderbar weitere Sachverhalte abprüfen und der Einsatz von „LIKE"- und „REGEXP"-Funktionen ermöglicht ausgefeiltere Checks.

```
select case i.type when 'VARIABLE'
                   then instr(upper(i.name),'V_')
                   when 'CONSTANT'
                   then instr(upper(i.name),'C_')
                   when 'FUNCTION'
                   then instr(upper(i.name),'FNC_')
                   when 'PROCEDURE'
                   then instr(upper(i.name),'PRC_')
                   when 'FORMAL IN'      -- In-Parameter
                   then instr(upper(i.name),'P_')
            end chk
     , i.name, substr(i.signature,1,7) signature, i.type
  , i.line, u.text user_source_text
 from sys.dba_identifiers i
 join user_source u
     on u.line = i.line and u.name = i.object_name
  and u.type = i.object_type
where i.object_name = 'ADD_JOB_HISTORY' and i.owner = 'HR'
  and i.usage = 'DECLARATION'
order by i.object_name, i.object_type, i.line, i.col
```

Listing 3: Alle Bezeichner auf richtige Benennung prüfen

Unbenutzte Variablen finden

Ein Problem bei der Wartung von Code sind stehengebliebene Reste, die in einer früheren Version einen Sinn hatten, jetzt aber nur noch die Lesbarkeit des Codes verschlechtern. Die Abfrage in *Listing 4* findet solche Variablen (und lässt sich auch auf andere Typen anpassen). Dieses Statement findet einen Treffer in „PRC_UNUSED" *(siehe Abbildung 7)*.

In der Prozedur „PRC_UNUSED" *(siehe Listing 5)* wird die Variable „V_STRING" zwar angelegt und erhält sogar einen Wert; sie wird allerdings nicht weiter benutzt und ist somit überflüssig.

CHK	NAME	SIGNATURE	TYPE	LINE	USER_SOURCE_TEXT
0	ADD_JOB_HISTORY	58A79BD	PROCEDURE	1	procedure add_job_history
1	P_EMP_ID	9B9AD81	FORMAL IN	2	(p_emp_id job_history.employee_id%type
1	P_START_DATE	4F94CA9	FORMAL IN	3	, p_start_date job_history.start_date%type
1	P_END_DATE	D31184C	FORMAL IN	4	, p_end_date job_history.end_date%type
1	P_JOB_ID	7E70403	FORMAL IN	5	, p_job_id job_history.job_id%type
1	P_DEPARTMENT_ID	C2873DF	FORMAL IN	6	, p_department_id job_history.department_id%type
1	C_PROC_NAME	BB863E1	CONSTANT	9	c_proc_name constant varchar2(30) := 'add_job_history';
0	CNT	052F060	VARIABLE	10	cnt pls_integer;

Abbildung 6: Die Guten ins Töpfchen, die Schlechten ...

Beispiel 3: Doppelte Deklarationen von Cursor-Variablen finden

Nehmen wir einen fiktiven Fall an, in dem eine Menge an Code von expliziten auf implizite Cursor umgestellt werden soll, um das lästige (und fehleranfällige) Öffnen und Schließen des Cursors zu vermeiden. Bei der Umstellung wurde allerdings an einigen Stellen vergessen, die Deklaration der Variable für die Ergebniszeilen des Cursors zu löschen. Den Compiler stört das überhaupt nicht, weil für ihn immer klar ist, in welchem Scope welche Deklaration zu verwenden ist. Hier können wir PL/Scope als Hilfsmittel gut gebrauchen.

Listing 6 zeigt, wie unterschiedliche Deklarationen gefunden werden. Die explizite Cursor-Variable erkennt man daran, dass die Variable in ihrem Kontext eine Referenz („REFERENCE") auf den Cursor hat; die implizite Cursor-Variable ist ein Iterator, in dessen Kontext ein Cursor aufgerufen wird („CALL").

Das Ergebnis ist in *Abbildung 8* zu sehen: Wenn die Anzahl unterschiedlicher Deklarationen „größer als 1" ist, weil wir eine Variable als „explizit" und als „implizit" deklariert haben, sollten wir den Code genauer prüfen, so wie in „CURSOR_DEMO_2" und „CURSOR_DEMO_3". In „CURSOR_DEMO_4" und „CURSOR_DEMO_5" gibt es kein Problem, weil hier nur implizite Deklarationen verwendet werden, wie man in der Textspalte rechts sehen kann.

```
select object_name
          , object_type
          , name
          , line
   from user_identifiers u
 where usage = ,DECLARATION'
       and type = ,VARIABLE'
       and not exists (select *
                       from user_identifiers i
                       where i.signature = u.signature
                       and i.usage not in
                           (,DECLARATION','ASSIGNMENT'))
```

Listing 4: Variablen suchen, die deklariert, aber nicht genutzt werden

OBJECT_NAME	OBJECT_TYPE	NAME	LINE
PRC_UNUSED	PROCEDURE	V_STRING	4

Abbildung 7: Die Variable ohne Nutzen und ihr Fundort

```
procedure prc_unused as
   cursor cur is select last_name, salary from employees;
   v_string varchar2(2000) := 'Hurz!';
   v_int integer;
begin
   for r in cur
   loop
      v_int := r.salary;
      sys.dbms_output.put_line(r.last_name || ' ' || v_int);
   end loop;
end prc_unused;
```

Listing 5: Die unbenutzte Variable im Kontext

Beispiel 4: Prüfung von paarweisen Methoden-Aufrufen bei der Verwendung von Logging-Frameworks

Bei vielen Logging-Lösungen gibt es eine Methode, die beim Einstieg in einen Abschnitt aufgerufen wird, und eine, die beim Ausstieg aufgerufen wird, sodass eine geschachtelte Aufrufhierarchie entsteht. Fehlt eine der beiden Methoden, gerät dieser (selbstgebaute) Call-Stack durcheinander. Mit PL/Scope lässt sich leicht feststellen, ob zu jedem „Enter" auf gleicher Ebene ein „Leave" existiert. *Listing 7* zeigt ein kurzes Beispiel mit dem PL/SQL Instrumentation Toolkit (PIT) von Jürgen Sieben. Nach dem Kompilieren dieser Prozedur fragen wir alle Methoden im HR-Schema explizit auf die Enter- und Leave-Prozeduren mit dem Statement aus *Listing 8* ab.

Gleich die erste Prozedur „ADD_JOB_HISTORY" fällt unangenehm auf, denn hier wird nur die Enter-Prozedur aufgerufen *(siehe Abbildung 9)*. Die Prozedur „DEL" im Package „EMPLOYEES_TAPI" ist ein möglicher Kandidat für eine genauere Prüfung,

NAME	SIGNATURE	TYPE	OBJECT_NAME	OBJECT_TYPE	USAGE_ID	USAGE_CONTEXT_ID	LINE	COL	ANZAHL	TEXT
REC	DAF57DB	VARIABLE	CURSOR_DEMO_2	PACKAGE BODY	7	1	4	3	2	rec x%rowtype;
REC	8140490	ITERATOR	CURSOR_DEMO_2	PACKAGE BODY	13	1	9	7	2	for rec in x
REC	5DCC946	VARIABLE	CURSOR_DEMO_3	PACKAGE BODY	7	1	4	3	2	rec x%rowtype;
REC	EAB9C18	ITERATOR	CURSOR_DEMO_3	PACKAGE BODY	13	1	9	7	2	for rec in x
REC	E6F3618	ITERATOR	CURSOR_DEMO_3	PACKAGE BODY	25	1	16	7	2	for rec in x
REC	72BD987	ITERATOR	CURSOR_DEMO_4	PACKAGE BODY	11	1	8	7	1	for rec in x
REC	39BC562	ITERATOR	CURSOR_DEMO_4	PACKAGE BODY	23	1	15	7	1	for rec in x
R	D9266F4	ITERATOR	CURSOR_DEMO_5	PROCEDURE	12	2	8	7	1	for r in cur
R	9123C5A	ITERATOR	CURSOR_DEMO_5	PROCEDURE	24	2	15	7	1	for r in cur
R	CBB0631	ITERATOR	CURSOR_DEMO_5	PROCEDURE	33	2	21	7	1	for r in cur

Abbildung 8: Gleichzeitige Verwendung eines Bezeichners für explizite und implizite Cursor-Variablen (rot)

da hier das Logging entweder vergessen oder absichtlich nicht eingebaut wurde. Die Prozeduren „INS“ und „UPD“ wurden ordnungsgemäß behandelt, die Prozedur „EMPS_OUT“ (siehe oben) besitzt offensichtlich gar kein Logging und die Prozedur „EMPS_OUT_LOG“ ist wie beabsichtigt mit Enter und Leave versorgt. Natürlich ließe sich hier mit noch mehr SQL auch sicherstellen, dass die Aufrufe an der richtigen Stelle stehen – der Ansatz ist beliebig ausbaufähig.

Beispiel 5: Öffentlich deklarierte Variablen finden

Eine Anleihe bei Steven Feuerstein ist das Statement zum Auffinden von Variablen, die in der Package-Spezifikation deklariert sind. Dies sollte aus Sicherheitsgründen nur im Package-Body erfolgen, sodass der Zugriff nur über definierte Methoden möglich ist. Führt man das Statement aus *Listing 9* aus, bekommt man ein Ergebnis wie in *Abbildung 10*.

Im Quellcode in *Listing 10* sieht man nichts Verbotenes, sondern nur eine ganz normale „RECORD“-Deklaration. Offensichtlich werden die Felder des Record-Typs ebenfalls dem Typ „VARIABLE“ zugeordnet. Das heißt, das Statement aus *Listing 9* muss noch ein wenig erweitert werden, um diese Fälle auszufiltern, wie man in *Listing 11* sieht.

Führt man das Statement in *Listing 11* aus, erhält man keine falsch-positiven Ergebnisse mehr. Eine solche Erweiterung ist übrigens auch für das Beispiel „Naming Conventions“ notwendig – es sei denn, es ist wirklich beabsichtigt, dass die Felder des Record mit „v_“ beginnen sollen wie die normalen Variablen.

Die Nachteile von PL/Scope

Natürlich gibt es all diese schönen Möglichkeiten nicht zum Nulltarif, aber die Nachteile beim Einsatz von PL/Scope sind schnell aufgezählt. An erster Stelle steht der Platzverbrauch, denn natürlich verbrauchen die Metadaten etwas Platz in der Datenbank. Dieser lässt sich mit „select space_usage_kbytes from v$sysaux_occupants where occupant_name='PL/SCOPE'“ abfragen. Die Dokumentation sagt nichts Konkretes über den potenziellen Platzverbrauch. Nach Beobachtungen in diversen Umgebungen scheint 1 KB pro Codezeile die obere Grenze zu sein. Es ist allerdings fraglich, ob die Anzahl der Codezeilen als Basis für die Berechnung taugt; es seien nur die Stichwörter „Formatierung“, „Kommentare“ und „Abhängigkeiten“ erwähnt, die diese Ratio beeinflussen.

```
with explizit as (select ui1.*
                    from user_identifiers ui1
                    left outer join user_identifiers ui2
                    on ui1.usage_id = ui2.usage_context_id
                    and ui1.object_name = ui2.object_name
                    and ui1.object_type = ui2.object_type
                    where ui1.type = 'VARIABLE'
                    and ui1.usage = 'DECLARATION'
                    and ui2.type 'CURSOR'
                    and ui2.usage = 'REFERENCE')'
      implizit as (select ui1.*
                    from user_identifiers ui1
                    left outer join user_identifiers ui2
                    on ui1.usage_id = ui2.usage_context_id
                    and ui1.object_name = ui2.object_name
                    and ui1.object_type = ui2.object_type
                    where ui1.type = 'ITERATOR'
                    and ui1.usage = 'DECLARATION
                    and ui2.type = 'CURSOR'
                    and ui2.usage = 'CALL')
select d.name, substr(d.signature,1,7) signature
      , d.type, d.object_name, d.object_type
      , d.usage_id, d.usage_context_id, d.line, d.col
      , count (distinct d.type)
          over (partition by d.object_name
                    , d.object_type
                    , d.usage_context_id) anzahl
      , s.text
from (select * from explizit
      union
      select * from implizit) d
join user_source s
  on s.name = d.object_name
 and s.type = d.object_type
 and s.line = d.line
order by d.object_name, d.object_type, d.usage_id
```

Listing 6: Überflüssige Cursor-Variablen finden

```
procedure hr.emps_out_log
   authid definer
is
   v_string varchar2(2000);
   v_int integer;
begin
   pit.enter('emps_out'); -- neue Ebene im Aufruf-Stack
   for rec in (select last_name, salary from employees)
   loop
      v_string := rec.last_name;
      v_int := rec.salary;
      sys.dbms_output.put_line(v_string || ' '
                  || sys.standard.to_char(v_int,99999));
   end loop;
   pit.leave; -- Prozedur vom Stack entfernen
exception
   when others then
      pit.sql_exception; -- pit.leave + Exception Handling
end;
```

Listing 7: Einfaches Beispiel für den Einsatz eines Logging-Frameworks

```
with methoden as (
   select owner, object_name, object_type, usage_id
       , type, name, usage_context_id
   from dba_identifiers
   where owner = 'HR'
       and type   in ('PROCEDURE','FUNCTION')
       and usage = 'DEFINITION'
)
select m.owner, m.object_name, m.object_type
       , m.name method_name
       , i.name log_package, j.name log_method
       , count (distinct j.name)
       over (partition by m.owner, m.object_name
              , m.object_type, m.name) ok
   from methoden m
   left outer join dba_identifiers i
       on i.object_name = m.object_name
       and i.object_type = m.object_type
       and i.name = 'PIT' and i.type = 'SYNONYM'
       and i.usage_context_id = m.usage_id
       left outer join dba_identifiers j
       on j.object_name = m.object_name
       and m.object_type = j.object_type
       and j.name in ('ENTER','LEAVE')
       and j.usage_context_id = i.usage_id
 order by m.owner, m.object_name, m.object_type, m.name
          , i.usage, i.object_name, j.usage, j.name
```

Listing 8: Aufrufe der Logging-Methoden „PIT.ENTER“ und „PIT.LEAVE“ prüfen

O...	OBJECT_NAME	OBJECT_TYPE	METHOD_NAME	LOG_PACKAGE	LOG_METHOD	OK
HR	ADD_JOB_HISTORY	PROCEDURE	ADD_JOB_HISTORY	PIT	ENTER	1
HR	EMPLOYEES_TAPI	PACKAGE BODY	DEL			0
HR	EMPLOYEES_TAPI	PACKAGE BODY	INS	PIT	LEAVE	2
HR	EMPLOYEES_TAPI	PACKAGE BODY	INS	PIT	ENTER	2
HR	EMPLOYEES_TAPI	PACKAGE BODY	UPD	PIT	LEAVE	2
HR	EMPLOYEES_TAPI	PACKAGE BODY	UPD	PIT	ENTER	2
HR	EMPS_OUT	PROCEDURE	EMPS_OUT			0
HR	EMPS_OUT_LOG	PROCEDURE	EMPS_OUT_LOG	PIT	ENTER	2
HR	EMPS_OUT_LOG	PROCEDURE	EMPS_OUT_LOG	PIT		2
HR	EMPS_OUT_LOG	PROCEDURE	EMPS_OUT_LOG	PIT	LEAVE	2

Abbildung 9: Sind die Logging-Methoden wie beabsichtigt verwendet worden?

```
select object_name, name, line
   from user_identifiers ai
 where ai.type = 'VARIABLE'
       and ai.usage = 'DECLARATION'
       and ai.object_type = 'PACKAGE'
 order by object_name, line
```

Listing 9: Suche nach öffentlich deklarierten Variablen

OBJECT_NAME	NAME	LINE
EMPLOYEES_TAPI	JOB_ID	8
EMPLOYEES_TAPI	EMPLOYEE_ID	9
EMPLOYEES_TAPI	SALARY	10
EMPLOYEES_TAPI	HIRE_DATE	11
EMPLOYEES_TAPI	DEPARTMENT_ID	12
EMPLOYEES_TAPI	LAST_NAME	13

Abbildung 10: Öffentlich deklarierte Variablen in generiertem Code?

Der zweite Faktor, der sich verändert, ist die Compile-Zeit. Im Kurztest ergab sich eine Verlängerung zwischen 10 und 20 Prozent. Ausdrücklich sei aber darauf hingewiesen, dass die Laufzeit gänzlich unbeeinflusst bleibt. Alles in allem handelt es sich also um sehr überschaubare und ohnehin nur temporäre Einschränkungen, die die Produktionsumgebung überhaupt nicht betreffen.

Grenzen von PL/Scope

Es gibt einige Ausnahmen bei der Generierung der Metadaten, an denen sich auf absehbare Zeit wohl auch nichts ändern wird; diese umfassen:

- Wrapped Code
- Anonyme Blöcke
- Dynamisches SQL

Zudem beschränkt sich die Analyse von SQL-Statements auf recht grundlegende Informationen. Will man hier eine wirklich detaillierte Analyse starten und etwa Data-Lineage betreiben, kommt man nicht darum herum, einen weiteren Parser einzusetzen.

Ein Aspekt, den man im Hinterkopf behalten sollte, ist, dass PL/Scope quasi blind für anonyme Blöcke im Code ist. Die Usage-Hierarchie für eine Prozedur mit einem Label vor einem Block sieht also anders aus als für denselben Code ohne Label, da das Label eine eigene Usage-ID erhält und alle Bezeichner in diesem Block eine Hierarchie-Ebene tiefer rutschen.

Selbermachen

Die Hauptquellen zu PL/Scope sind überschaubar und lassen sich sehr schnell googeln: Steven Feuerstein hat einige Artikel geschrieben und auf LiveSQL *(siehe „http://livesql.oracle.com“)* einige „Beispiele to go“ bereitgestellt. Zusätzlich ist ein Blick in die Doku empfohlen („Using PL/Scope“) und ein weiterer Blick zu GitHub, wo die hier benutzten Statements mit Beispielcode abgelegt sind *(siehe „https://github.com/its-people/plscope2)* und wo Philipp Salvisberg einige sehr nützliche Views und Packages für die Arbeit mit PL/Scope bereitstellt *(siehe „https://github.com/PhilippSalvisberg/plscope-utils“)*.

```
package employees_tapi
is
type employees_tapi_rec is record (
      job_id employees.job_id%type
      , employee_id employees.employee_id%type
      , salary employees.salary%type
      , hire_date employees.hire_date%type
      …
```

Listing 10: Deklaration eines Record-Typs

```
select object_name, name, line
   from user_identifiers ai
 where ai.type = 'VARIABLE'
   and ai.usage = 'DECLARATION'
   and ai.object_type = 'PACKAGE'
   and not exists
       (select * from user_identifiers ui
          where ui.object_type = ai.object_type
              and ui.object_name = ai.object_name
              and ui.usage_id = ai.usage_context_id
              and ui.type = ‚RECORD')
 order by object_name, line
```

Listing 11: Im übergeordneten Kontext prüfen, ob es sich um einen „RECORD“ handelt

Fazit

PL/Scope ist in der aktuellen Version 12.2 noch nicht ganz frei von einigen Merkwürdigkeiten, bietet aber umfangreiche Basis-Informationen für viele Anwendungsfälle. Ein paar davon wurden hier angerissen; in der Praxis stößt man mit Sicherheit auf viele weitere Ideen, wie sich PL/Scope in seinen Entwicklungsumgebungen einsetzen lässt. Die Autorin freut sich, Berichte darüber zu hören oder zu lesen.

Sabine Heimsath
sabine.heimsath@its-people.de