Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Master's thesis

Checking the compatibility of data types in Oracle SQL

Bc. Ondřej Pelech

Supervisor: doc. Ing. Jan Janoušek, Ph.D.

13th May 2015

Acknowledgements

I would like to thank Jiří Toušek and Jan Janoušek for their help and guidance.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 13th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Pelech, Ondřej. *Checking the compatibility of data types in Oracle SQL.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Programy v Oracle SQL mohou selhat, když obsahují chyby. V této práci představíme metodu pro odhalování chyb specifického druhu – nekompatibilita datových typů. Analyzujeme datové typy a vestavěné funkce v dialektu Oracle SQL. Navrhneme metodu pro kontrolu kompatibility datových typů, která pracuje nad grafem datových toků. Tuto metodu implementujeme v jazyku Java.

Klíčová slova Oracle, SQL, datový typ, kompatibilita, kontrola

Abstract

Programs in the Oracle SQL language can go wrong if they contain mistakes. In this thesis we present a method for discovering one particular kind of mistakes – incompatibility of datatypes. We analyze the datatypes and built-in functions in the Oracle SQL dialect. Then we design a method for compatibility checking that works on the dataflow graph. We implement the checker in Java.

Keywords Oracle, SQL, datatype, compatibility, checker

Contents

List of Figures

List of Tables

Introduction

SQL (abbreviation of *Structured Query Language*) is a programming language for manipulating data in a *RDBMS* (relational database management system). Theoretically based on relational algebra, it is very widely used today, with many dialects.

One of the most widely used dialects is Oracle SQL[1]. Together with the *Oracle PL/SQL* procedural extension, it offers many additional features over plain *SQL*. Among these features are additional datatypes, (built-in) functions, and procedures.

Oracle SQL is used in data warehouses that can be very big. The SQL programs themselves can be large, too, with many commands, functions and procedures. They are also often written by many people over long periods of time.

This makes introducing errors very easy. There are many kinds of errors possible. Unfortunately, the Oracle compiler will catch only some of them, and the others will manifest themselves only when the program is run. And that can happen only for some particular input, or not even in a deterministic way.

The most used methods for finding errors in programs is a type checker[2]. In this thesis we present a method for static analysis of SQL programs of the Oracle PL/SQL dialect that will check the compatibility of datatypes. This method is similar to the traditional type checker, but is not the same. Traditional type checker are syntax-driven. In contrast, our method works on the dataflow graph.

To build the datatype compatibility checker we will first need to analyze, what datatypes are available in the Oracle PL/SQL language. We will work only with the most widely used ones, and omit the more exotic ones. We will also analyze how, under what circumstances and with what results can a value of one datatype be converted to a value of another datatype. We call this phenomenon a datatype *convertibility*. Of particular interest are: is the conversion even possible or will this always raise a runtime exception? Will

the value fit in? Will the precision of the value be preserved?

Then we take a look at the built-in functions in the Oracle SQL dialect. We analyze what (and how many) datatypes they can take as input and how that influences the output datatype. We also describe a way to symbolically describe the parameters of datatypes. This enables us to capture the relation between datatypes' parameters on the input and on the output.

We design the checker that works on the dataflow graph of the SQL program. The checker works with the simple dataflow between two nodes with specified datatypes, dataflow into and from a built-in function, and even with dataflow into/from a user-defined function/procedure with OUT parameters. Oracle PL/SQL features a support for object-oriented programming, but we omit this feature in this thesis.

The checker is implemented in the Java 6 language. We describe the main classes that we implemented. The dataflow checker is developed as part of the Manta Tools [3].

The thesis is organized as follows. Chapter 1 describes the datatypes in Oracle SQL. Chapter 2 describes the built-in and user-defined functions and procedures. Chapter 3 describes dataflow graph of an SQL program and the checking performed on the graph. Chapter 4 describes the classes implemented. Chapter 5 presents an experimental evaluation. Appendix A lists the formal rules flow between datatypes.

Chapter 1

Datatypes in Oracle SQL

Oracle database offers many datatypes[4] for various uses. Some of which can be parameterized by integers, e.g. character datatypes to specify length or numbers to specify precision or scale. Some are included for compatibility with different databases or the ANSI standard. In this section we give a detailed overview of those datatypes that we work with, those which we translate into their native counterparts and those which we omit entirely.

1.1 Used datatypes

We will be working with the following native Oracle datatypes:

- `CHAR`
- `VARCHAR2`
- `NCHAR`
- `NVARCHAR2`
- `NUMBER`
- `FLOAT`
- `BINARY_FLOAT`
- `BINARY_DOUBLE`
- `LONG`
- `LONG RAW`
- `RAW`
- `DATE`
- `TIMESTAMP`
- `TIMESTAMP WITH TIMEZONE`
- `INTERVAL YEAR TO MONTH`
- `INTERVAL DAY TO SECOND`
- `BLOB`
- `CLOB`
- `NCLOB`
- `BFILE`
- `ROWID`
- `UROWID`
- `BINARY_INTEGER`
- `BOOLEAN`

Here we describe each datatype in detail. For simplicity, we categorize the datatypes into these groups, according to their similarity: *Character*, *Number*, *Long and Raw*, *Datetime*, *Large Object*, *Rowid* and *PL/SQL*.

1.1.1 Character

1.1.1.1 CHAR[(*size*[byte|char])]

The CHAR datatype stores a fixed-length character string. All the strings are the same length. If a shorter string is inserted, it is padded by *space* characters up to the specified length.

The datatype is parameterized by a natural number *size* that specifies its length. The default and minimal *size* is 1 and the maximal 2000.

The *size* parameter can be followed by a *length semantics specifier*.

BYTE means that the resulting datatype will be able to contain *size* bytes.

CHAR means that the resulting datatype will be able to contain *size* characters, even though some characters may take up more than 1 byte. However, the maximal capacity is 2000 *bytes* (not characters).

1.1.1.2 VARCHAR2(*size*[byte|char])

The VARCHAR2 datatype stores a variable-length character string.

The datatype is parameterized by a natural number *size* that specifies its length. *size* can range from 1 to 4000.

The *size* parameter can be followed by a *length semantics specifier*.

BYTE means that the resulting datatype will be able to contain *size* bytes.

CHAR means that the resulting datatype will be able to contain *size* characters, even though some characters may take up more than 1 byte. However, the maximal capacity is 4000 *bytes* (not characters).

1.1.1.3 NCHAR[(*size*)]

The NCHAR datatype stores a fixed-length Unicode character string. All the strings are the same length. If a shorter string is inserted, it is padded by *space* characters up to the specified length.

The datatype is parameterized by a natural number *size* that specifies the number of characters it can take. The default and minimal *size* is 1. However, the maximal capacity is 2000 *bytes* (not characters).

1.1.1.4 NVARCHAR2(*size*)

The NVARCHAR2 datatype stores a variable-length Unicode character string.

The datatype is parameterized by a natural number *size* that specifies its length. The minimal *size* is 1. However, the maximal capacity is 4000 *bytes* (not characters).

1.1.2 Number

The *Number* datatypes store numeric values, positive and negative fixed and floating-point numbers, zero, infinity, and values that are the undefined result of an operation (NaN).

1.1.2.1 `NUMBER[(p [,s])]`

The `NUMBER` datatype stores a rational number in decimal system.

The datatype is parameterized by

- a natural number p (for precision) that specifies the maximum number of significant decimal digits. p can range from 1 to 38.
- an integer s (for scale) that specifies the number of digits from the decimal point to the least significant digit. s can range from -84 to 127, the default is 0.

1.1.2.2 `FLOAT[(p)]`

The `FLOAT` datatype stores a rational number in binary system.

The datatype is parameterized by a natural number p (for precision) that specifies the maximum number of significant binary digits. p can range from 1 to 126.

Scale cannot be specified, but is interpreted from the data.

1.1.2.3 `BINARY_FLOAT`

The `BINARY_FLOAT` datatype stores a 32-bit, single-precision floating-point number.

1.1.2.4 `BINARY_DOUBLE`

The `BINARY_DOUBLE` datatype stores a 64-bit, double-precision floating-point number.

1.1.3 Long and Raw

The *Long and Raw* datatypes store binary data and byte strings.

1.1.3.1 `LONG`

The `LONG` datatype stores character data of variable length up to 2 gigabytes.

1.1.3.2 `LONG RAW`

The `LONG RAW` datatype stores raw binary data of variable length up to 2 gigabytes.

1.1.3.3 RAW(*size*)

The `RAW` datatype stores variable-length raw binary data.

The datatype is parameterized by a natural number *size* that specifies the maximum capacity in bytes. *size* can range from 1 to 2000.

1.1.4 Datetime

The *Datetime* datatypes store date, time and interval related values. The datatypes make use of some of these datetime fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TIMEZONE_ABBR.

1.1.4.1 DATE

The `DATE` datatype stores the datetime fields *YEAR*, *MONTH*, *DAY HOUR*, *MINUTE*, and *SECOND*. It does not store fractional seconds or a time zone.

1.1.4.2 TIMESTAMP[(*fractional_seconds_precision*)]

The `TIMESTAMP` datatype stores the datetime fields *YEAR*, *MONTH*, *DAY HOUR*, *MINUTE*, and *SECOND*. It does not store a time zone.

The datatype is parameterized by a natural number *fractional_seconds_precision* that specifies the number of digits in the fractional part of the *SECOND* datetime field. *fractional_seconds_precision* can range from 0 to 9, default is 6.

1.1.4.3 TIMESTAMP[(*fractional_seconds_precision*)] WITH TIME ZONE

The `TIMESTAMP WITH TIME ZONE` datatype stores the same values as `TIMESTAMP` along with the *TIMEZONE_HOUR* and *TIMEZONE_MINUTE* datetime fields.

The *fractional_seconds_precision* parameter works just as with `TIMESTAMP`.

1.1.4.4 INTERVAL YEAR[(*year_precision*)] TO MONTH

The `INTERVAL YEAR TO MONTH` datatype stores a period of time in years and months.

The datatype is parameterized by a natural number *year_precision* that specifies the number of digits in the *YEAR* datetime field. *year_precision* can range from 0 to 9, default is 2.

1.1.4.5 `INTERVAL DAY[(day_precision)] TO SECOND[(fractional_seconds)]`

The `INTERVAL DAY TO SECOND` datatype stores a period of time in days, hours, minutes, and seconds.

The datatype is parameterized by:

- a natural number *day_precision* that specifies the maximum number of digits in the *DAY* datetime field. *day_precision* can range from 0 to 9, default is 2.
- a natural number *fractional_seconds* that specifies the maximum number of digits in the *DAY* datetime field. *fractional_seconds* can range from 0 to 9, default is 6.

1.1.5 Large Object

The *Large Object* datatypes store large and unstructured data like text, image and video.

1.1.5.1 `BLOB`

The `BLOB` datatype stores unstructured binary large objects up to 4 gigabytes.

1.1.5.2 `CLOB`

The `CLOB` datatype stores single-byte and multibyte character data up to 4 gigabytes.

1.1.5.3 `NCLOB`

The `NCLOB` datatype stores Unicode character data up to 4 gigabytes.

1.1.5.4 `BFILE`

The `BFILE` datatype enables access to binary file that is stored in file systems.

1.1.6 Rowid

The *Rowid* datatypes store addresses of rows in the database.

1.1.6.1 `ROWID`

The `ROWID` datatype stores a string representing the unique address of a row in its table.

1.1.6.2 UROWID[(*size*)]

The UROWID datatype stores string representing the logical address of a row of an index-organized table.

The datatype is parameterized by a natural number *size* that specifies the size of a column of type *UROWID*. The maximum *size* and default is 4000 bytes.

1.1.7 PL/SQL

For this thesis, we take into account only a small subset of the datatypes that are available in the *PL/SQL* language. The *PL/SQL* datatypes can be only used in the PL/SQL *scripts*, but cannot be used in tables.

1.1.7.1 BINARY_INTEGER

The BINARY_INTEGER datatype stores any integer value from -2147483648 to 2147483647.

1.1.7.2 BOOLEAN

The BOOLEAN datatype stores these three values: TRUE, FALSE, NULL.

1.2 Translated datatypes

Oracle database also offers datatypes for compatibility with SQL/DS, DB2 and the ANSI standard. We translate them to their Oracle native counterparts. The table 1.1 describes how we translate them.

- CHARACTER
- CHARACTER VARYING
- CHAR VARYING
- NCHAR VARYING
- VARCHAR
- NATIONAL CHARACTER
- NATIONAL CHAR
- NATIONAL CHARACTER VARYING
- NATIONAL CHAR VARYING
- NCHAR VARYING
- LONG VARCHAR
- NUMERIC
- DECIMAL
- DEC
- INTEGER
- INT
- SMALLINT
- DOUBLE PRECISION
- REAL
- PLS_INTEGER

Table 1.1: Translation of datatypes

Original datatype	Oracle native datatype
CHARACTER(*n*)	CHAR(*n*)
CHARACTER VARYING(*n*) CHAR VARYING(*n*) VARCHAR(*n*)	VARCHAR2(*n*)
NATIONAL CHARACTER(*n*) NATIONAL CHAR(*n*)	NCHAR(*n*)
NATIONAL CHARACTER VARYING(*n*) NATIONAL CHAR VARYING(*n*) NCHAR VARYING(*n*)	NVARCHAR2(*n*)
NUMERIC(*p*,*s*) DECIMAL(*p*,*s*)	NUMBER(*p*,*s*)
INTEGER INT SMALLINT	NUMBER(38,0)
DOUBLE PRECISION	FLOAT(126)
REAL	FLOAT(63)
PLS_INTEGER	BINARY_INTEGER

1.3 Omitted datatypes

Finally, there are some of the native datatypes in Oracle database that we do not take into account. The reason we omit them is because these datatypes are deeply specialized, e.g. for multimedia or geometry, and thus are rarely used.

- ANYTYPE
- ANYDATA
- ANYDATASET
- XMLType
- URIType
- DBURIType
- XDBURIType
- HTTPURIType
- SDO_GEOMETRY
- SDO_TOPO_GEOMETRY
- SDO_GEORASTER
- ORDAudio
- ORDImage
- ORDVideo
- ORDDoc
- ORDDicom
- SI_StillImage
- SI_Color
- SI_AverageColor
- SI_ColorHistogram
- SI_PositionalColor
- SI_Texture
- SI_FeatureList
- ORDImageSignature
- Expression

1.4 Literals

The Oracle SQL language allows expressing values of some datatypes in the literals form. From reading the literal we may learn to which datatype it belongs.

1.4.1 BINARY_INTEGER

Any whole number (a string of digits with no decimal point) with optional sing and in the range from -2147483648 to 2147483647 is of the `BINARY_INTEGER`.

Examples are: `0 -123 098 +45`

However, the `BINARY_INTEGER` is not very useful for the dataflow analysis. So we interpret every `BINARY_INTEGER` literal as `NUMBER` literal. This gives us more information – the *precision* and *scale* of the value.

1.4.2 NUMBER

Any numbers with or without sign or decimal point are `NUMBER` literals. They can also be in the *scientific* format.

Examples are: `2.56 -1.4E3 34e-3 -0`

1.4.3 BINARY_FLOAT

The literals for `BINARY_FLOAT` are almost like those for `NUMBER` except they have `f` at the end.

Examples are: `2.56f -1.4E3f 34e-3f -0f`

1.4.4 BINARY_DOUBLE

The literals for `BINARY_DOUBLE` are almost like those for `NUMBER` except they have `d` at the end.

Examples are: `2.56d -1.4E3d 34e-3d -0d`

1.4.5 CHAR

`CHAR` literals are strings of characters enclosed in one of there delimiters: pair of `'`s, or `q'!` and `!'`, or `q'[` and `]'`, or `q'{` and `}'`, or `q'<` and `>'`, or `q'(` and `)'`.

Examples are: `'hello' q'!brave new!'  q'[world]'`

1.4.6 BOOLEAN

The literals for `BOOLEAN` are these: `NULL FALSE TRUE`.

1.4.7 DATE

A character string descibing a date prepended with `DATE` makes a literal of the `DATE` datatype.

Examples is: `DATE '2015-12-24'`

1.4.8 TIMESTAMP and TIMESTAMP WITH TIMEZONE

A character string descibing a timestamp prepended with `TIMESTAMP` makes a literal of the `TIMESTAMP` or `TIMESTAMP WITH TIMEZONE` datatype, depending of the string.

Example for `TIMESTAMP` is: `TIMESTAMP '2015-12-24 18:01:01'`

Example for `TIMESTAMP WITH TIMEZONE` is: `TIMESTAMP '2015-12-24 18:01:01 +02:00'`

1.4.9 INTERVAL YEAR TO MONTH

Example of `INTERVAL YEAR TO MONTH` literal that describes an interval of 6 years and 1 month is:

```
INTERVAL '6-1' YEAR TO MONTH
```

1.4.10 INTERVAL DAY TO SECOND

Example of `INTERVAL DAY TO SECOND` literal that describes an interval of days, four hours, three minutes, two and 1/100 seconds is:

```
INTERVAL '5 04:03:02.01' DAY TO SECOND
```

1.5 Datatype conversion

Some of Oracle datatypes can be converted into other datatypes. This happens automatically, when a value from a column of one type is inserted to a column of another type.

For most datatypes, this relation between datatypes (value of type one being convertible to another) is symmetric. For example, one can move values between `NVARCHAR2` and `FLOAT` back and forth. On the other hand, there are exceptions. `CHAR` can be converted into `BLOB`, but not the other way around.

But the relation is always reflexive.

When one attempts to perform a conversion between non-convertible datatypes, a database will always raise an exception.

In this section we give an overview of which datatypes are convertible into which. We use a table where the symbol • in the column `A` in the row `B` means that the values of the datatype `A` can be automatically converted to the datatype `B`.

1.5.1 Character

All the *character* datatypes are very flexible and can be converted into almost all other datatypes. The exceptions are `BLOB`, `BFILE`, `ROWID` and `UROWID`.

to \ from	CHAR	VARCHAR2	NCHAR	NVARCHAR2
CHAR	•	•	•	•
VARCHAR2	•	•	•	•
NCHAR	•	•	•	•
NVARCHAR2	•	•	•	•
NUMBER	•	•	•	•
FLOAT	•	•	•	•
BINARY_FLOAT	•	•	•	•
BINARY_DOUBLE	•	•	•	•
LONG	•	•	•	•
LONG RAW	•	•	•	•
RAW	•	•	•	•
DATE	•	•	•	•
TIMESTAMP	•	•	•	•
TIMESTAMP WITH TIMEZONE	•	•	•	•
INTERVAL YEAR TO MONTH	•	•	•	•
INTERVAL DAY TO SECOND	•	•	•	•
BLOB	•			
CLOB	•	•	•	•
NCLOB	•	•	•	•
BFILE				
ROWID		•	•	•
UROWID		•	•	•
BINARY_INTEGER	•	•	•	•
BOOLEAN				

1.5.2 Number

The *number* datatypes are convertible between each other and can be also converted to any *character* datatype. Conversion to other datatypes is impossible. The exception is `NUMBER`, which can be converted to `CLOB` and `NCLOB`.

to \ from	NUMBER	FLOAT	BINARY_FLOAT	BINARY_DOUBLE
CHAR	•	•	•	•
VARCHAR2	•	•	•	•
NCHAR	•	•	•	•
NVARCHAR2	•	•	•	•
NUMBER	•	•	•	•
FLOAT	•	•	•	•
BINARY_FLOAT	•	•	•	•
BINARY_DOUBLE	•	•	•	•
LONG				
LONG RAW				
RAW				
DATE				
TIMESTAMP				
TIMESTAMP WITH TIMEZONE				
INTERVAL YEAR TO MONTH				
INTERVAL DAY TO SECOND				
BLOB				
CLOB	•			
NCLOB	•			
BFILE				
ROWID				
UROWID				
BINARY_INTEGER	•	•	•	•
BOOLEAN				

1.5.3 Long and Raw

The *long and raw* datatypes can be converted between each other, to the *character* types. Some of them can be converted to `BLOB`, `CLOB` or `NCLOB`.

to \ from	LONG	LONG RAW	RAW
CHAR	•	•	•
VARCHAR2	•	•	•
NCHAR	•	•	•
NVARCHAR2	•	•	•
NUMBER			
FLOAT			
BINARY_FLOAT			
BINARY_DOUBLE			
LONG	•	•	•
LONG RAW	•	•	•
RAW	•	•	•
DATE			
TIMESTAMP			
TIMESTAMP WITH TIMEZONE			
INTERVAL YEAR TO MONTH			
INTERVAL DAY TO SECOND			
BLOB		•	•
CLOB	•		
NCLOB	•		
BFILE			
ROWID			
UROWID			
BINARY_INTEGER			
BOOLEAN			

1.5.4 Datetime

The *Datetime* datatypes can all be converted to any *Characeter* datatype. Timestamps and interval can be converted to `LONG`. Also, `DATE` and Timestamps can be converted between themselves and Intervals too.

to \ from	DATE	TIMESTAMP	T.W.TIMEZONE	INTERVAL YEAR	INTERVAL DAY
CHAR	•	•	•	•	•
VARCHAR2	•	•	•	•	•
NCHAR	•	•	•	•	•
NVARCHAR2	•	•	•	•	•
NUMBER					
FLOAT					
BINARY_FLOAT					
BINARY_DOUBLE					
LONG		•	•	•	•
LONG RAW					
RAW					
DATE	•	•	•		
TIMESTAMP	•	•	•		
T.W.TIMEZONE	•	•	•		
INTERVAL YEAR				•	•
INTERVAL DAY				•	•
BLOB					
CLOB					
NCLOB					
BFILE					
ROWID					
UROWID					
BINARY_INTEGER					
BOOLEAN					

1.5.5 Large Object

The *Large Object* datatypes are not very convertible. `CLOB` and `NCLOB` can be converted between each other and to any *Character* datatypes.

to \ from	BLOB	CLOB	NCLOB	BFILE
CHAR		•	•	
VARCHAR2		•	•	
NCHAR		•	•	
NVARCHAR2		•	•	
NUMBER				
FLOAT				
BINARY_FLOAT				
BINARY_DOUBLE				
LONG		•	•	
LONG RAW	•			
RAW	•			
DATE				
TIMESTAMP				
TIMESTAMP WITH TIMEZONE				
INTERVAL YEAR TO MONTH				
INTERVAL DAY TO SECOND				
BLOB	•			
CLOB		•	•	
NCLOB		•	•	
BFILE				•
ROWID				
UROWID				
BINARY_INTEGER				
BOOLEAN				

1.5.6 Rowid

The *Rowid* datatypes are both convertible to all the *Character* datatypes, with the exception of `CHAR`. However, they are not convertible to one another.

to \ from	ROWID	UROWID
CHAR		
VARCHAR2	•	•
NCHAR	•	•
NVARCHAR2	•	•
NUMBER		
FLOAT		
BINARY_FLOAT		
BINARY_DOUBLE		
LONG		
LONG RAW		
RAW		
DATE		
TIMESTAMP		
TIMESTAMP WITH TIMEZONE		
INTERVAL YEAR TO MONTH		
INTERVAL DAY TO SECOND		
BLOB		
CLOB		
NCLOB		
BFILE		
ROWID	•	
UROWID		•
BINARY_INTEGER		
BOOLEAN		

1.5.7 PL/SQL

The `BINARY_INTEGER` datatype is convertible to all the *Character* and *Number* datatypes. `BOOLEAN` datatype is convertible to all the other datatypes (because it has the value `NULL`).

to \ from	BINARY_INTEGER	BOOLEAN
CHAR	•	•
VARCHAR2	•	•
NCHAR	•	•
NVARCHAR2	•	•
NUMBER	•	•
FLOAT	•	•
BINARY_FLOAT	•	•
BINARY_DOUBLE	•	•
LONG		•
LONG RAW		•
RAW		•
DATE		•
TIMESTAMP		•
TIMESTAMP WITH TIMEZONE		•
INTERVAL YEAR TO MONTH		•
INTERVAL DAY TO SECOND		•
BLOB		•
CLOB		•
NCLOB		•
BFILE		•
ROWID		•
UROWID		•
BINARY_INTEGER	•	•
BOOLEAN		•

1.6 Conversion to *character* datatypes

The *character* datatypes hold a special position among others because almost every datatype can be converted into them. For most of datatypes we can derive the maximal length of string result when a value of some datatype is converted into one of the *character* datatypes, sometimes based on the original datatype alone, sometimes we need to know it's parameters as well.

In this section we discuss the maximal length of the result when converter to a *character* datatype.

We use this to determine maximal length as a function *len* in the rules A.

1.6.1 Character

The simplest case if when the original datatype is a *character* datatype itself. The maximal result length is the same as is the parameter of the datatype.

E.g. if we have `NVARCHAR2(`*size*`)`, the value can be up to *size* characters long.

1.6.2 Number

- The `NUMBER(`p`,`s`)` datatype is the most elaborate:
  - if $s > 0$ and $p > s$ then the maximal length is $p + 1$ (+1 because of possible sign)
  - if $s > 0$ and $p <= s$ then the maximal length is $s + 2$ (+2 because of leading zero and decimal point)
  - if $s <= 0$ then the maximal length is $p - s$
- Maximal length of `FLOAT` is 9.
- Maximal length of `BINARY_FLOAT` is 15.
- Maximal length of `BINARY_DOUBLE` is 23.

1.6.3 Raw

- Maximal length of `RAW(`*size*`)` is *size*.
- The length of `LONG` and `LONG RAW` cannot be derived.

1.6.4 Datetime

- Maximal length of `DATE` is 9.
- Maximal length of `TIMESTAMP` is 31.
- Maximal length of `TIMESTAMP WITH TIMEZONE` is 38.
- Maximal length of `INTERVAL YEAR TO MONTH` is 6.
- Maximal length of `INTERVAL DAY TO SECOND` is 19.

1.6.5 Rowid

- Maximal length of `ROWID` is 18.
- Maximal length of `UROWID` is 4000.

1.6.6 PL/SQL

- Maximal length of `BINARY_INTEGER` is 11.
- Maximal length of `BOOLEAN` is 5.

1.7 Flow between datatypes

In the previous section we have explained that some datatypes can be converted to other datatypes. When an a value of a datatype is inserted into a column of an incompatible datatype, the database raises an exception. But that is not the only way an exception can be raised, this can happen even when transferring values between compatible columns.

For example, a value can be to large, like when trying to move the value `'hello'` from `CHAR(5BYTE)` to `VARCHAR2(3BYTE)`. Or only some values can be converted, when trying to move the value `'Lorem Ipsum'` from `NVARCHAR2(30)` to `DATE`.

For this reason, we establish a more refined notion of *flow* between two datatypes that also takes into account the specific parameters of both datatypes. *Flow* can describe i a more nuanced way if and why a conversion will (or can) fail.

1.7.1 Flow labels

Flow between two datatypes can be described by one of the following labels:

Safe is the simplest of all. There is no datatype conversion going on. The precision is kept. The sizes fit. And the operation is guaranteed to success.

ConversionSafe signifies that the datatype will be converted, otherwise the same as *Safe*. The precision is kept. The sizes fit. And the operation is guaranteed to success.

Imprecise signifies that the precision of the value being moved might not be preserved. But there is no datatype conversion going on. The sizes fit. And the operation is guaranteed to success.

ConversionImprecise signifies that the datatype will be converted and that the value being converted might not be preserved. But sizes fit. And the operation is guaranteed to success.

ConversionUnsafe signifies that the specific value might not be convertible, so this operation can fail. The datatype will be converted. But the precision is kept. The sizes fit.

ConversionImpreciseUnsafe signifies that the specific value might not be convertible, so this operation can fail. If it succeeds, some precision might be lost. But the sizes fit.ept. The sizes fit.

WrongSize signifies that the value might not fit into the target datatype, so this operation can fail. But there is no datatype conversion going on. The precision is kept.

ConversionWrongSize signifies that the value might not fit into the target datatype, so this operation can fail. If ti succeeds, the datatype will be converted. But the precision is kept.

Incompatible signifies that the value cannot be under any circumstances converted and this operation is guaranteed to fail.

1.7.2 Flow labels comparison

In the table 1.2 you can see the difference between flow labels.

Table 1.2: Comparison of *flow* labels

	Type	Precision	Size	Run
Safe	same	kept	fit	safe
ConversionSafe	conv	kept	fit	safe
Imprecise	same	lost	fit	safe
ConversionImprecise	conv	lost	fit	safe
ConversionUnsafe	conv	kept	fit	unknown
ConversionImpreciseUnsafe	conv	lost	fit	unknown
WrongSize	same	kept	unfit	unknown
ConversionWrongSize	conv	kept	unfit	unknown
Incompatible				fail

1.7.3 Flow rules

The above mentioned flow labels are used to describe flow between datypes and specified rules decide which label will be used. In this section we will not list all the rules. We will only show how they work on specific examples. The rules are listed in the appendix A.

Suppose we have a flow from x:`NCHAR(10)` to y:`NCHAR(15)`. Here we use the rule

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

where $a = x$, $size_a = 10$, $b = y$, $size_b = 15$ and because $10 \leq 15$. From that we can conclude that the flow from x to y is *Safe*.

Suppose we have a flow from x:`NVARCHAR2(10)` to y:`NCHAR(7)` Here we use the rule

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

where $a = x$, $size_a = 10$, $b = y$, $size_b = 7$ and because $10 > 7$. From that we can conclude that the flow from x to y can fail. If the value is `'hello'` (5 characters long), that is OK, it fits and is successfully converted from `NVARCHAR2` to `NCHAR`. But if it is `'helloworld'` (10 characters long), it raises a runtime exception.

Suppose we have a flow from x:`NUMBER(6, 0)` to y:`NUMBER(7, -3)` Here we use the rule

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{NUMBER}(p_b, s_b) \quad -s_b \leq p_a - s_a \leq p_b - s_b}{a \xrightarrow[Imprecise]{} b}$$

where $a = x$, $p_a = 6$, $s_a = 0$, $b = y$, $p_b = 7$, $s_b = -3$ and because $-(-3) \leq 6 - 0 \leq 7 - (-3)$. From that we can conclude that the flow from x to y can be imprecise. If the value is `64000`, that is OK, that fits and precision is kept. But if it is `65536`, the 3 least significant digits will be discarded.

Suppose we have a flow from x:`NUMBER(7, -3)` to y:`CLOB` Here we use the rule

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{CLOB}}{a \xrightarrow[Incompatible]{} b}$$

where $a = x$, $b = y$. From that we can conclude that the flow from x to y will fail, no value of datatype `NUMBER` can be converted to `CLOB`.

Chapter 2

Functions and Procedures in Oracle SQL

Oracle PL/SQL provides us with many built-in functions and also supports user-defined function and procedures. In this chapter we describe the functions and how they transform their parameters, i.e. how does the result and its size (or precision or scale) depend on the inputs size (or precision or scale).

2.1 Symbolic description of parameters

We use a simple language to describe the parameters of datatypes. The language includes integer, variables and simple arithmetic operators. The grammar of the language is described in the figure 2.1.

The integers are used in situation when we know precisely the parameter, e.g. `NUMBER(7,2)`.

In other cases, we might not know the parameter precisely. It is usually the case when the datatype is a parameter and its size (in case of *character* datatypes) or precision/scale (in case of `NUMBER`) is not specified. In that case the datatype's parameter could be a simple variable, e.g. `NVARCHAR(VAR3)`.

When dealing with functions we can also track the dependency of the result (or *OUT* parameters in case of procedures) in a symbolic manner, without any specific integers. Let's suppose we have a functions to concatenate 3 strings. We don't even need to specify the length of the parameters, that is `a : NVARCHAR2(VAR0)`, `b : NVARCHAR2(VAR1)`, `c : NVARCHAR2(VAR2)`. The result is then the sum of the three variables – `NVARCHAR2((PLUS (PLUS VAR2 VAR3) VAR4))`.

When the function is applied and we know the actual parameters, we can bind them to the formal ones, and thus bind the variables to integers. With this binding of variables to integers (aka environment), we can evaluate expressions.

For example if `VAR0` → 12, `VAR2` → 23 and `VAR2` → 34, we can evaluate the expression `(PLUS (PLUS VAR2 VAR3) VAR4)`. The result is 69.

With this result we can conclude that the function that receives strings that are up to 12, 23 and 34 characters long respectively, it returns `NVARCHAR2(69)`.

Figure 2.1: Grammar for the simple expression language

⟨*expression*⟩ ::= ⟨*int*⟩
| ⟨*var*⟩
| '(' ⟨*intop*⟩ ⟨*expression*⟩ ⟨*expression*⟩ ')'

⟨*int*⟩ ::= 0 | 1 | -1 | 2 | -2 | 3 | -3 | ...

⟨*var*⟩ ::= var1 | var2 | var3 | ...

⟨*intop*⟩ ::= plus | minus | max | min

We also need to be able to compare for inequality not only the `<int>`s, but also any other expressions. This typically happens when we need to decide if the size of a target datatype is sufficient for the flow into it. For example, if we have a flow from `NVARCHAR2(10)` to `NVARCHAR2(`var_1`)`. In this case, $10 \leq var_1$. This means that there can be flow from datatype of a specified size to a datatype of unspecified size. And of course, the opposite is false.

The rules for this weak $\leq$ are as follows:

- $\_ \leq var$ – anything is lesser or equal to any *var*
- $\_ \leq (intop\_\_)$ – anything is lesser or equal to any application of any *intop* to anything
- two *int*s are compared as usual

2.2 Built-in functions

In this section we describe a subset of Oracle SQL built-in functions [5] that we support. Many of the built-in functions, contrary to the user-defined ones, are polymorphic. That means they can accept parameters of various datatypes, without conversion. Only if a value of an unsupported datatype is provided, a conversion is attempted. And only if that fails an exception is thrown. The the returning datatype can also depend on the input datatype(s).

2.2.1 Numeric functions

Here we describe numeric functions, which are functions that take one or more *numeric* datatype and return a *numeric* datatype.

2.2.1.1 `abs`

The `abs(`n`)` function returns and absolute value of n.

n can be any *numeric* datatype and the function returns the same datatype as n.

2.2.1.2 `bitand`

The `bitand(`n_1`, `n_2`)` function returns a bitwise *and* of n_1 and n_2.

n_1 is of datatype `NUMBER(`var_1`, `var_2`)`, n_2 is of datatype `NUMBER(`var_3`, `var_4`)` and the result is of datatype `NUMBER(`var_5`, `var_6`)`

2.2.1.3 `ceil` and `floor`

The `ceil(`n`)` and `floor(`n`)` return the closest larger or smaller-or-equal integer than n.

n can be any *numeric* datatype. If n is `NUMBER(`var_1`, `var_2`)`, the result is NUMBER($max(var_1 - var_2, 1), 0$), otherwise the result is of the same datatype as n.

2.2.1.4 `round` and `trunc`

The function `round(`n_1`<,`n_2`>)` function returns n_1 rounded to n_2 places to the right of the decimal point. The function `trunc(`n_1`<,`n_2`>)` function returns n_1 truncated to n_2 decimal places.

n_2 is an optional argument, default is 0. n_1 and n_2 can be any *numeric* datatypes. If n_2 is provided, the function returns `NUMBER(`var_1`, `var_2`)`, otherwise it returns the same datatype as n_1.

2.2.1.5 `sign`

The `sign(`n`)` function returns -1, 0, or 1 depending on the sign of n.

It returns `NUMBER(1,0)`.

2.2.1.6 Numberic functions with one parameter

These are the functions:

- `acos(`n`)`
- `asin(`n`)`
- `atan(`n`)`
- `cos(`n`)`
- `cosh(`n`)`
- `exp(`n`)`
- `ln(`n`)`
- `sin(`n`)`
- `sinh(`n`)`
- `tan(`n`)`
- `tanh(`n`)`
- `sqrt(`n`)`

The parameter n can be of any *numeric* datatype. The function returns `BINARY_DOUBLE`.

2.2.1.7 Numberic functions with two parameters

These are the functions:

- `atan2`(n_1,n_2)
- `mod`(n_1,n_2)
- `power`(n_1,n_2)
- `log`(n_1,n_2)
- `nanvl`(n_1,n_2)
- `remainder`(n_1,n_2)

Both parameter n_1 and n_2 can be of any *numeric* datatype. The function returns `BINARY_DOUBLE`.

2.2.2 Character functions returning character values

Here we describe character functions that return a *character* datatype.

2.2.2.1 `chr`

The function `chr`(n) returns a character at the n^{th} position of the database character set.

n is to be of type `NUMBER`(var_1`, 0`).

The function return value has datatype `VARCHAR2(1)`.

2.2.2.2 `concat`

The function `concat`($char_1$,$char_2$) returns $char_1$ concatenated with $char_2$.

$char_1$ and $char_2$ can be of any *character* datatype, `CLOB` or `NCLOB`.

The result datatype is chosen so that the concatenation is losses. Therefor if one of the input datatypes is *lob*, the result is also *lob*; if one of the input datatypes has *national charset*, the result also has *national charset*. The size of result is (*plus* len($char_1$) len($char_2$).

2.2.2.3 `initcap`, `lower` and `upper`

The functions `initcap`(*char*), `lower`(*char*) and `upper`(*char*) change the case of *char*.

char can be of any *character* datatype, `CLOB` or `NCLOB`.

The function return value has the same datatype as *char*.

2.2.2.4 `lpad` and `rpad`

The functions `lpad`($char_1$,n`<`,$char_2$`>`) and `rpad`($char_1$,n`<`,$char_2$`>`) return $char_1$ padded with $char_2$ to the length n.

$char_1$ and $char_2$ can be of any *character* datatype, `CLOB` or `NCLOB`. n is of datatype `NUMBER`(var_1`, 0`).

$char_2$ is optional parameter.

If $char_1$ is of `CLOB` or `NCLOB` datatype, the return value is of the same datatype. If $char_1$ is of `CHAR` or `VARCHAR2` datatype, the return value is of

VARCHAR2(*var*) datatype. If *char*$_1$ is of NCHAR or NVARCHAR2 datatype, the return value is of NVARCHAR2(*var*) datatype.

2.2.2.5 ltrim and rtrim

The functions ltrim(*char*$_1$<,*char*$_2$>) and rpad(*char*$_1$<,*char*$_2$>) remove from an end of *char*$_1$ characters contained in *char*$_2$.

char$_1$ and *char*$_2$ can be of any *character* datatype, CLOB or NCLOB.

If *char*$_1$ is of CLOB or NCLOB datatype, the return value is of the same datatype. If *char*$_1$ is of CHAR or VARCHAR2 datatype, the return value is of VARCHAR2(*var*) datatype. If *char*$_1$ is of NCHAR or NVARCHAR2 datatype, the return value is of NVARCHAR2(*var*) datatype.

2.2.2.6 nls_initcap, nls_lower, nls_upper

The functions nls_initcap(*char*$_1$<,*char*$_2$>), nls_lower(*char*$_1$<,*char*$_2$>) and nls_upper(*char*$_1$<,*char*$_2$>) change the case of *char*$_1$ according to the locale defined by *char*$_2$.

char$_1$ and *char*$_2$ can be of any *character* datatype, CLOB or NCLOB.

If *char*$_1$ is of CLOB or NCLOB datatype, the return value is of the same datatype. Otherwise if *char*$_1$ is of CHAR(*var*), VARCHAR2(*var*), NCHAR(*var*) or NVARCHAR2(*var*) datatype, the return value is of VARCHAR2(*var* CHAR) datatype.

2.2.2.7 regex_replace

The function regex_replace(*char*$_1$,*char*$_2$<,*char*$_3$<,*n*$_1$<,*n*$_2$<,*char*$_4$>>>>)) performs replacement using regular expression.

char$_1$, *char*$_2$, *char*$_3$ and *char*$_4$ can be of any *character* datatype, CLOB or NCLOB. *n*$_1$ and *n*$_2$ are of datatype NUMBER(*var*$_1$, 0).

If *char*$_1$ is of CLOB or NCLOB datatype, the return value is of the CLOB datatype. Otherwise if *char*$_1$ is of CHAR(*var*), VARCHAR2(*var*), NCHAR(*var*) or NVARCHAR2(*var*) datatype, the return value is of VARCHAR2(*var* CHAR) datatype.

2.2.2.8 regex_substr

The function regex_substr(*char*$_1$,*char*$_2$<,*n*$_1$<,*n*$_2$<,*char*$_3$<,*char*$_4$>>>>) searches for substring using regular expression.

char$_1$, *char*$_2$, *char*$_3$ and *char*$_4$ can be of any *character* datatype, CLOB or NCLOB. *n*$_1$ and *n*$_2$ are of datatype NUMBER(*var*, 0).

If *char*$_1$ is of CLOB or NCLOB datatype, the return value is of the CLOB datatype. Otherwise if *char*$_1$ is of CHAR(*var*), VARCHAR2(*var*), NCHAR(*var*) or NVARCHAR2(*var*) datatype, the return value is of VARCHAR2(*var* CHAR) datatype.

2.2.2.9 replace

The function replace(*char*$_1$,*char*$_2$<,*char*$_3$>) replaces every occurrence of *char*$_2$ in *char*$_1$ with *char*$_3$.

char$_1$, *char*$_2$ and *char*$_3$ can be of any *character* datatype, CLOB or NCLOB.

If *char*$_1$ is of CLOB or NCLOB datatype, the return value is of the CLOB datatype. Otherwise if *char*$_1$ is of CHAR(*var*), VARCHAR2(*var*), NCHAR(*var*) or NVARCHAR2(*var*) datatype, the return value is of VARCHAR2(*var* CHAR) datatype.

2.2.2.10 substr*

These functions are:

- substr(*char*, n_1<,n_2>)
- substr2(*char*, n_1<,n_2>)
- substrb(*char*, n_1<,n_2>)
- substrc(*char*, n_1<,n_2>)
- substr4(*char*, n_1<,n_2>)

These functions return a portion of *char*$_1$, beginning at character n_1, n_2 characters long.

char can be of any *character* datatype, CLOB or NCLOB. n_1 and n_2 are of datatype NUMBER(*var*, 0).

If *char* is of CLOB or NCLOB datatype, the return value is of the CLOB datatype. Otherwise if *char* is of CHAR(*var*), VARCHAR2(*var*), NCHAR(*var*) or NVARCHAR2(*var*) datatype, the return value is of VARCHAR2(*var* CHAR) datatype.

2.2.2.11 soundex

The function soundex(*char*) returns a character string containing the phonetic representation of *char*.

char can be of any *character* datatype, CLOB or NCLOB.

The function return value is of datatype VARCHAR2(4 BYTE).

2.2.2.12 translate

The function translate(*char*$_1$,*char*$_2$,*char*$_3$) returns *char*$_1$ with all occurrences of each character in *char*$_2$ replaced by its corresponding character in *char*$_3$.

char$_1$, *char*$_2$ and *char*$_3$ can be of any *character* datatype.

The function return value is of the same datatype as *char*$_1$.

2.2.3 Character functions returning number values

Here we describe character functions that return a *numeric* datatype.

2.2.3.1 ascii

The function ascii(*char*) returns the position of *char* in the database character set.

char can be of any *character* datatype.

The function return value is of datatype NUMBER(3,0).

2.2.3.2 instr*

These functions are:

- instr(*char*$_1$, *char*$_2$<, n_1<, n_2>>)
- instrb(*char*$_1$, *char*$_2$<, n_1<, n_2>>)
- instrc(*char*$_1$, *char*$_2$<, n_1<, n_2>>)
- instr2(*char*$_1$, *char*$_2$<, n_1<, n_2>>)
- instr4(*char*$_1$, *char*$_2$<, n_1<, n_2>>)

They search for *char*$_2$ in *char*$_1$.

char$_1$ and *char*$_2$ can be of any *character* datatype, CLOB or NCLOB. n_1 and n_2 are of datatype NUMBER(*var*, 0).

The function return value is of datatype NUMBER(*var*,0).

2.2.3.3 length*

These functions are:

- length(*char*)
- lengthc(*char*)
- length4(*char*)
- lengthb(*char*)
- length2(*char*)

They return the length of *char*.

char$_1$ can be of any *character* datatype, CLOB or NCLOB.

The function return value is of datatype NUMBER(*var*,0).

2.2.3.4 regexp_instr

The function regexp_instr(*char*$_1$,*char*$_2$<,n_1<,n_2<,n_3<,*char*$_3$<,n_4>>>>>) searches in *char*$_1$ using regular expressions.

char$_1$ and *char*$_3$ can be of any *character* datatype, CLOB or NCLOB. *char*$_2$ is of datatype CHAR(512 BYTE). n_1, n_3 and n_4 are of datatype NUMBER(*var*,0). n_3 is of datatype NUMBER(1,0).

The function return value is of datatype NUMBER(*var*,0).

2.2.4 Aggregate functions

These functions are:

- avg(n)
- max(n)
- median(n)
- min(n)
- stats_mode(n)
- stddev(n)
- stddev(n)
- stddev_samp(n)
- sum(n)
- var_pop(n)
- var_samp(n)
- variance(n)

n can be any *numeric* datatype.

If n is of datatype NUMBER, the result is of datatype NUMBER(var_1,var_2). Otherwise the result is of the same datatype as n.

2.2.5 Built-in Operators

Here we describe the built-in operators. These operators are highly polymorphous and work not only on *numeric* datatypes, but also on *datetime* datatypes [6].

The following tables show the result datatypes of each operator. The datatype of the result is always the higher one, according to the following precedence:

- TIMESTAMP WITH TIMEZONE > TIMESTAMP
- INTERVAL DAY TO SECOND > INTERVAL YEAR TO MONTH
- BINARY_DOUBLE > BINARY_FLOAT > FLOAT > NUMBER > BINARY_INTEGER

2.2.5.1 Addition

1^{st} arg. + 2^{nd} arg.	DATE	TIMESTAMP	INTERVAL	*numeric*
DATE			DATE	DATE
TIMESTAMP			TIMESTAMP	DATE
INTERVAL	DATE	TIMESTAMP	INTERVAL	
numeric	DATE	DATE		*numeric*

If 1^{st} arg. is NUMBER(var_1, var_2) and 2^{nd} arg. is NUMBER(var_3, var_4) , then the result is NUMBER(***precision***, ***scale***), where

- ***precision*** $= (max((var_0 - var_1), (var_2 - var_3)) + 1) + max(var_1, var_3)$
- ***scale*** $= max(var_1, var_3)$

2.2.5.2 Subtraction

1^{st} arg. - 2^{nd} arg.	DATE	TIMESTAMP	INTERVAL	*numeric*
DATE	NUMBER	INTERVAL	DATE	DATE
TIMESTAMP	INTERVAL	INTERVAL	TIMESTAMP	DATE
INTERVAL			INTERVAL	
numeric				*numeric*

If 1^{st} arg. is `NUMBER(`var_1`, `var_2`)` and 2^{nd} arg. is `NUMBER(`var_3`, `var_4`)` , then the result is `NUMBER(`*precision*`, `*scale*`)`, where

- *precision* $= (max((var_0 - var_1), (var_2 - var_3)) + 1) + max(var_1, var_3)$
- *scale* $= max(var_1, var_3)$

2.2.5.3 Multiplication

1^{st} arg. `*` 2^{nd} arg.	`DATE`	`TIMESTAMP`	`INTERVAL`	*numeric*
`DATE`				
`TIMESTAMP`				
`INTERVAL`				`INTERVAL`
numeric			`INTERVAL`	*numeric*

If 1^{st} arg. is `NUMBER(`var_1`, `var_2`)` and 2^{nd} arg. is `NUMBER(`var_3`, `var_4`)` , then the result is `NUMBER(`*precision*`, `*scale*`)`, where

- *precision* $= var_0 + var_2$
- *scale* $= var_1 + var_3$

2.2.5.4 Division

1^{st} arg. `/` 2^{nd} arg.	`DATE`	`TIMESTAMP`	`INTERVAL`	*numeric*
`DATE`				
`TIMESTAMP`				
`INTERVAL`				`INTERVAL`
numeric				*numeric*

If 1^{st} arg. is `NUMBER(`var_1`, `var_2`)` and 2^{nd} arg. is `NUMBER(`var_3`, `var_4`)` , then the result is `NUMBER(`var_5`, `var_6`)`.

2.2.5.5 Concatenation

The operator `||` works the same way as the function `concat` 2.2.2.2.

2.3 User-defined functions and procedures

Oracle PL/SQL allows programmer to write custom functions and procedures. In contrast to the built-in functions, the user-defined ones are not polymorphic. That means that the datatypes of input(s) and output of a function/procedure is unambiguous (of course, the usual implicit conversions take place). Procedures (but not functions) can specify, if the parameters are *IN*, *OUT* or *IN/OUT*; the default is simply *IN*.

Typical function declaration can look like 2.1. Typical procedure declaration can look like 2.2.

```
create function myfunction (a VARCHAR2(20), b NUMBER(10,3))
return VARCHAR2(100) is
begin
  ...
  return ...
end;
```

Listing 2.1: Function declaration

```
create procedure myproc (c NUMBER(10,3), d out VARCHAR2(15)) is
begin
  ...
end;
```

Listing 2.2: Procedure declaration

However, the sizes (in *character* datatypes), or precision/scale (in `NUMBER`), do not necessarily have to be specified. Example of this we can see in 2.3 and 2.4.

```
create function myfunction (a VARCHAR2, b NUMBER)
return VARCHAR2 is
begin
  ...
  return ...
end;
```

Listing 2.3: Function declaration without specifying sizes of parameters

```
create procedure myproc (c NUMBER, d out VARCHAR2) is
begin
  ...
end;
```

Listing 2.4: Procedure declaration without specifying sizes of parameters

In that case, we threat treat these sizes/precisions/scales as unknown and use fresh variables to represent them. This gives us these types:

- a : `VARCHAR2(`var_1`)`
- b : `NUMBER(`var_2`,` var_3`)`
- c : `NUMBER(`var_4`,` var_5`)`
- d : `VARCHAR2(`var_6`)`

Chapter 3

Dataflow Checking

In this chapter we describe the dataflow graph of a program we analyze. Then we explain the algorithm for analyzing such a graph. We will deal with various scenarios from the simplest (2 nodes with datatype anotations) to the more complicated where functions or operators are involved.

The basic idea of flow checking is to get appropriate dataflow pairs, apply the flow rules 1.7 to them, and report the flow incidents.

3.1 Dataflow graph

Before we can explain the algorithm for checking dataflow of an SQL program, we first have to describe the dataflow graph of a given program, because the graph is a prerequisite for the algorithm. (The creation of the dataflow graph is not topic of this thesis and we expect that a procedure to generate the graph is available.)

Suppose we have the following program:

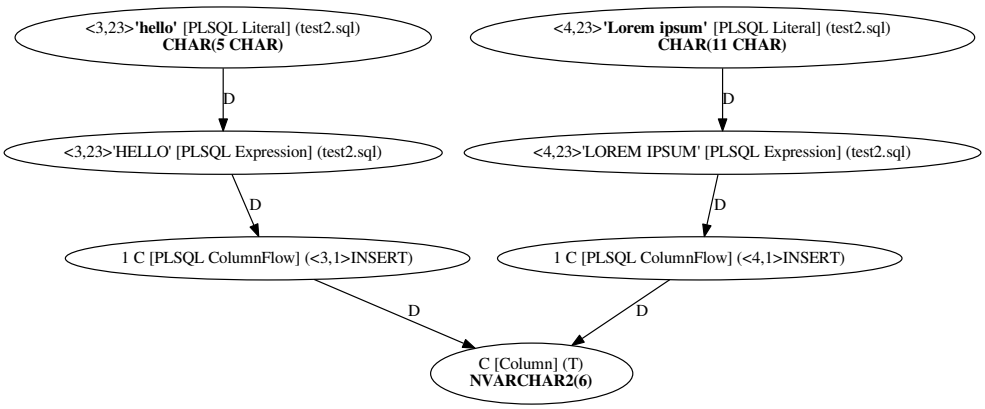
```
CREATE TABLE t (c NVARCHAR(6));
INSERT INTO t VALUES ('hello');
INSERT INTO t VALUES ('Lorem␣ipsum');
```

Listing 3.1: Simple SQL program

We create a table `t` with one column `c` of datatype `NVARCHAR2(6)`. Into the column we insert values `'hello'` and `'Lorem ipsum'`.

This program gives us the dataflow graph in the figure 3.1. There are many nodes, but the important ones are emphasized with bold labels. We see a node that corresponds to the column `c`, with appropriate datatype information. There are also two nodes corresponding to the two literals with the inferred datatypes – we know that `'hello'` is of datatype `CHAR(5 CHAR)` and that `'Lorem ipsum'` is of datatype `CHAR(11 CHAR)`.

Figure 3.1: Dataflow graph of a simple program



The direction of graph edges signifies the flow of data. In this case it means that values of will flow from nodes with datatypes `CHAR(5 CHAR)` and `CHAR(11 CHAR)` respectively to the node with datatype `NVARCHAR2(6)`.

3.2 Checking flow between 2 typed nodes

Checking the flow always begins in the nodes where we know the datatype – those can be nodes that correspond to columns or declared variables in the program – and that are at the same time there is a dataflow into them. (Note that nodes that correspond to literals are always source of dataflow, never target.)

From a target we look against the flow of data and find one or more source nodes. The simplest check is when a source has fixed datatypes. When we have this pair of nodes (source → target), we can use the rules 1.7.3 to describe the flow of data.

For example, if we take the program 3.1 and it's dataflow graph, we have these nodes (let's call them a, b and c):

- $a =$ 'hello' : `CHAR(5 CHAR)`
- $b =$ 'Lorem ipsum' : `CHAR(11 CHAR)`
- $c = C[Column](T)$: `NVARCHAR2(6)`

We also get these dataflow pairs: $a \to c$ and $b \to c$. Using the rules we can annotate the flow in the following way:

- $a \xrightarrow[ConversionSafe]{} c$
- $b \xrightarrow[ConversionWrongSize]{} c$

Finally we can report these two dataflow incidents:

- (a, `CHAR(5 CHAR)`, c, `NVARCHAR2(6)`, $ConversionSafe$)
- (b, `CHAR(11 CHAR)`, c, `NVARCHAR2(6)`, $ConversionWrongSize$)

3.3 Checking built-in function

A more complicated case is where we deal with a built-in function. Oracle SQL provides us with various standard functions, but the way to check the flow of data is the same for all of them:

- check the dataflow to the function, because some functions may expect certain parameters to be of only some specific datatype
- based on the incoming data, compute the resulting datatype (and its parameter, if applicable)
- check the flow from this newly computed datatype to its target

Suppose we have the following program 3.2:

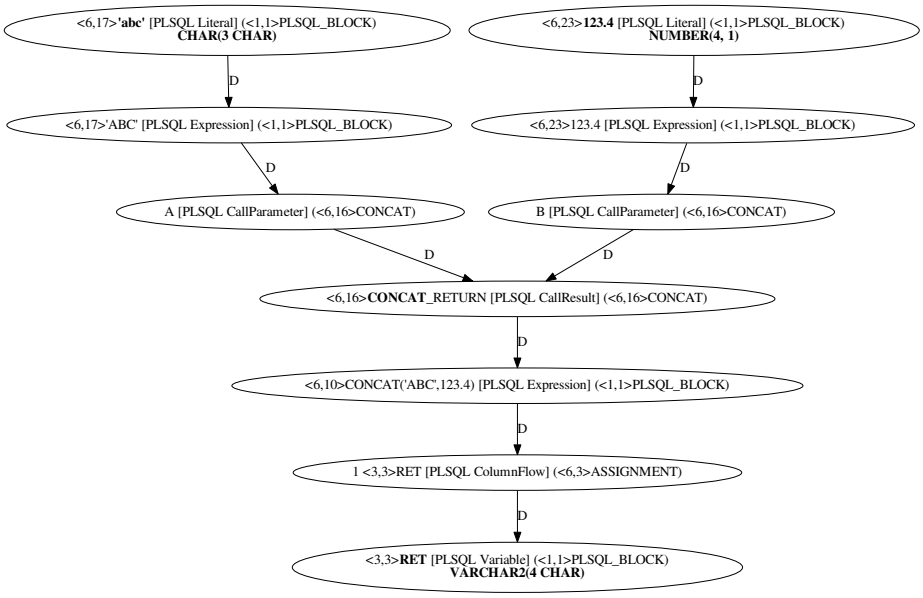
```
DECLARE
  ret VARCHAR2(4);
BEGIN
  ret := CONCAT('abc',123.4);
END;
```

Listing 3.2: SQL program with built-in function

The program gives us the dataflow graph 3.3.
We have these nodes:

- $a =$ 'abc' : `CHAR(3 CHAR)`
- $b = 123.4$: `NUMBER(4, 1)`
- $c =$ A [PLSQL CallParameter] (<6,16>CONCAT)
- $d =$ B [PLSQL CallParameter] (<6,16>CONCAT)
- $e = CONCAT\_RETURN$: *result datatype unknown at this point*
- $f = ret$: `VARCHAR2(4 CHAR)`

Figure 3.2: Dataflow graph of program with built-in function



And we have these dataflow pairs: $a \to c$, $b \to d$, $e \to f$.

From the fact that the built-in function is $CONCAT$, we know that it accepts two *character* datatypes and so the 2nd parameter will be implicitly converted, giving us these flows: $a \xrightarrow[Safe]{} c$ and $b \xrightarrow[ConversionSafe]{} d$.

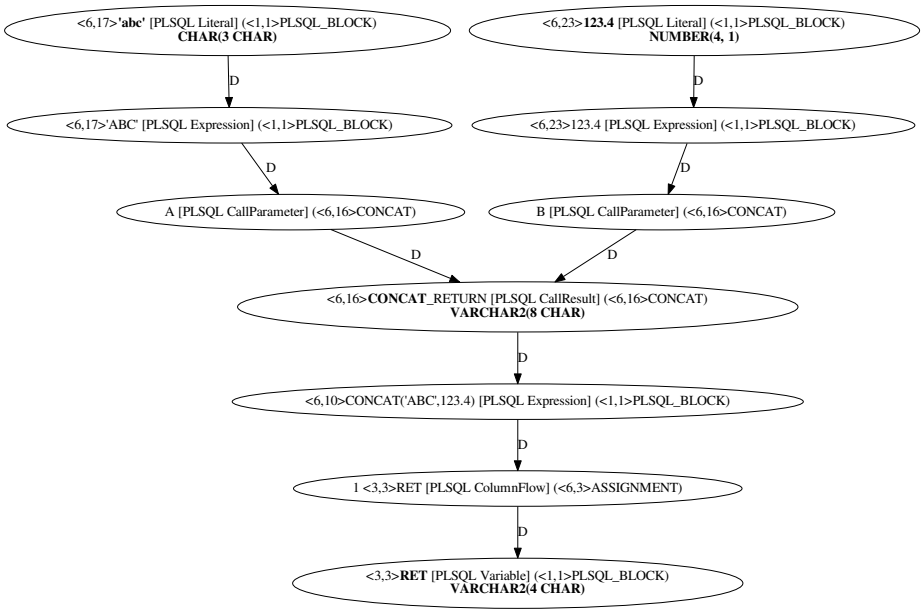
The function $CONCAT$ returns a *character* datatype that is as long as sum of lengths of its parameters. Since $len($`CHAR(3 CHAR)`$) = 3$ and $len($`NUMBER(4, 1)`$) = 5$ we can derive the resulting datatype as `VARCHAR2(8 CHAR)`.

We add this datatype anotation to the node e (as seen on the graph 3.3). That enables us to address the dataflow pair $e \to f$, which resolve based on the given datatypes as $e \xrightarrow[WrongSize]{} f$.

In the end, we report these incidents:

- (a, `CHAR(5 CHAR)`, c, –, $Safe$)
- (b, `NUMBER(4, 1)`, d, –, $ConversionSafe$)
- (e, `VARCHAR2(8 CHAR)`, f, `VARCHAR2(4 CHAR)`, $WrongSize$)

Figure 3.3: Dataflow graph of program with built-in function with computed resulting datatype



3.4 Checking user-defined functions and procedures

Checking user-defined functions differs from checking the built-in ones in the fact that the datatypes of their parameters and result are precisely defined. However, the programmer may choose not to specify the *length* of a parameter (when talking about `NVARCHAR2` and similar) or the precision/scale (when talking about `NUMBER`).

3.4.1 Parameters with specified sizes

When the lengths of parameters are specified, checking of dataflow is easy. We can check the flow to the nodes independently, that is:

- check the flow to the individual parameters of the function
- check the flow in the function body to the function result
- check the flow from the function result further, e.g. to the variable where the result is assigned

For example, if we take the program 3.3, it gives us the dataflow graph with the following nodes:

Those that are part of the *concat3* function:

- d = A [Parameter] (CONCAT3) : `VARCHAR2(10 CHAR)`
- e = B [Parameter] (CONCAT3) : `VARCHAR2(10 CHAR)`
- f = C [Parameter] (CONCAT3) : `VARCHAR2(10 CHAR)`
- $g = A||B$: `VARCHAR2(20 CHAR)`
- $h = A||B||C$: `VARCHAR2(30 CHAR)`
- i = CONCAT3_RETVAL : `VARCHAR2(30 CHAR)`

And those that are not part of the function:

- a = 'a' : `CHAR(1 CHAR)`
- b = 'bb' : `CHAR(2 CHAR)`
- c = 'ccc' : `CHAR(3 CHAR)`
- j = CONCAT3_RETURN : `VARCHAR2(30 CHAR)`
- $k = ret$: `VARCHAR2(10 CHAR)`

From those we can compute the incidents inside the *concat3* function:

- (d, `VARCHAR2(10 CHAR)`, g, –, $Safe$)
- (e, `VARCHAR2(10 CHAR)`, g, –, $Safe$)
- (g, `VARCHAR2(20 CHAR)`, h, –, $Safe$)
- (f, `VARCHAR2(10 CHAR)`, h, –, $Safe$)
- (h, `VARCHAR2(30 CHAR)`, i, `VARCHAR2(30 CHAR)`, $Safe$)

And the incidents in the flow into and out of the function:

- (a, `CHAR(1 CHAR)`, d, –, $ConversionSafe$)
- (b, `CHAR(2 CHAR)`, e, –, $ConversionSafe$)
- (c, `CHAR(3 CHAR)`, f, –, $ConversionSafe$)
- (i, `VARCHAR2(30 CHAR)`, j, `VARCHAR2(30 CHAR)`, $Safe$)
- (j, `VARCHAR2(30 CHAR)`, k, `VARCHAR2(10 CHAR)`, $WrongSize$)

Please note that if all the parameters (and returns) sizes are specified, the order in which we check the flow pairs doesn't matter.

```
create function concat3 ( a VARCHAR2(10)
                        , b VARCHAR2(10)
                        , c VARCHAR2(10)
                        )
return VARCHAR2(30) is
begin
  return a||b||c;
end;

declare
  ret varchar2(10);
begin
  ret := concat3('a','bb', 'ccc');
end;
```

Listing 3.3: SQL program with a user-defined function with fixed lengths

3.4.2 Parameters with arbitrary sizes

When the parameters' sizes are not specified, the situation is a bit more complicated. We have to treat the parameters' sizes as variables that may influence the size of the result. For this reason we proceed in this order:

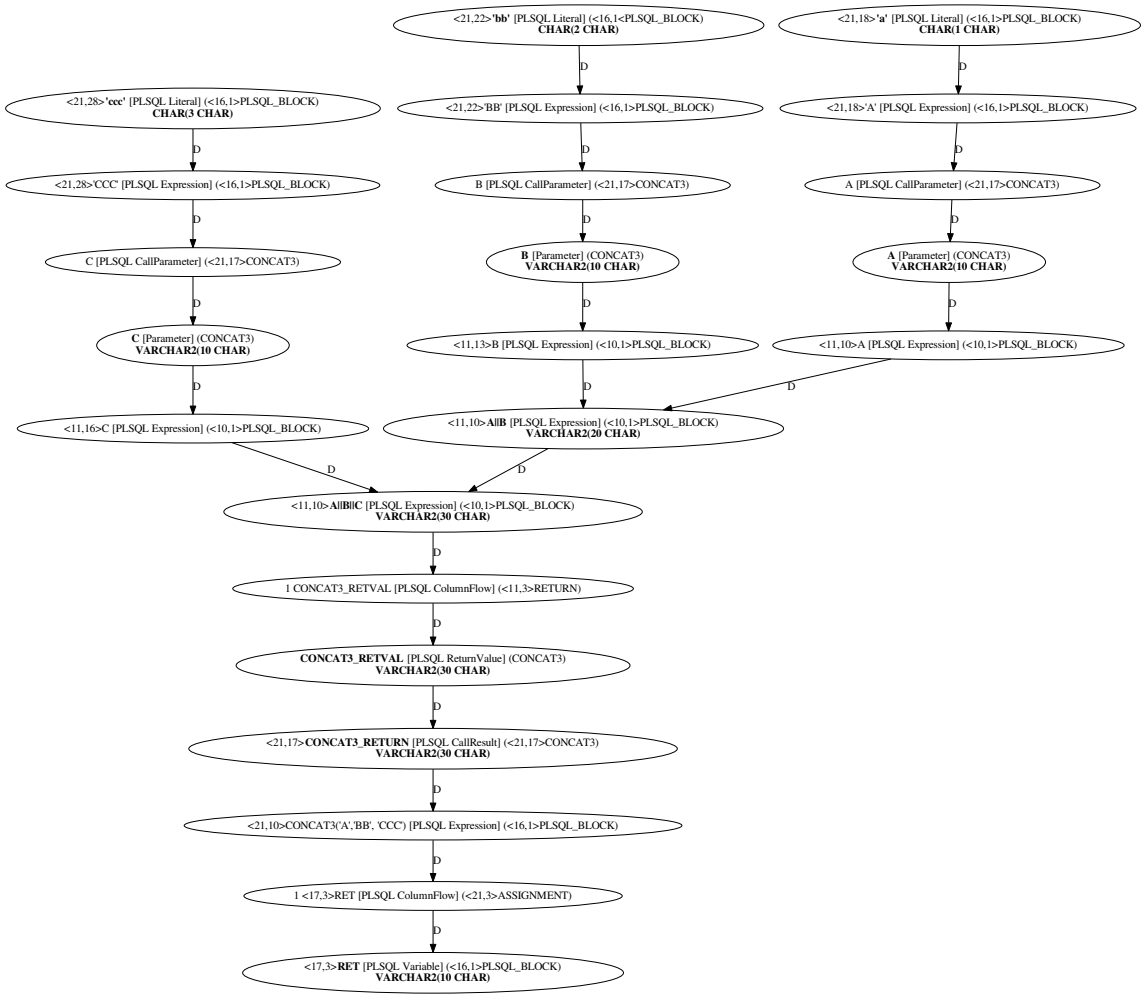
1. check the flow inside the function
2. use that to compute the result of the function symbolically, i.e. how the result depends on the *formal* parameters of the function (can be done only once for a function)
3. for each individual call of the function, compute what are the *actual* parameters
4. for each individual call, with the knowledge of the actual parameters and the symbolic result, compute the actual result
5. then we can check the flow from the result further down

The flow into the formal parameters can be checked independently.

Let's see how this work on a concrete program 3.4. We see that we have a function *concat3* with the following formal parameters:

- A : `VARCHAR2(VAR2 CHAR)`
- B : `VARCHAR2(VAR3 CHAR)`
- C : `VARCHAR2(VAR4 CHAR)`

Figure 3.4: Dataflow graph of program with a user-defined function with fixed lengths



From the body of the function, we compute that the symbolic result of the function is `VARCHAR2((PLUS (PLUS VAR2 VAR3) VAR4) CHAR)`.

There is only one instance of applying the function, and we know that the correspondence of formal parameters to the actual ones is:

- A : `VARCHAR2(VAR2 CHAR)` $\rightarrow$ 'a' : `CHAR(1 CHAR)`
- B : `VARCHAR2(VAR3 CHAR)` $\rightarrow$ 'bb' : `CHAR(2 CHAR)`
- C : `VARCHAR2(VAR4 CHAR)` $\rightarrow$ 'ccc' : `CHAR(3 CHAR)`

From that we conclude that the values of the variable are these:

- $VAR2 \rightarrow len(\texttt{CHAR(1 CHAR)})$
- $VAR3 \rightarrow len(\texttt{CHAR(2 CHAR)})$
- $VAR4 \rightarrow len(\texttt{CHAR(3 CHAR)})$

Which is `VAR2` $\rightarrow 1$, `VAR3` $\rightarrow 2$, `VAR4` $\rightarrow 3$. Then we evaluate the symbolic size of the datatype `VARCHAR2((PLUS (PLUS VAR2 VAR3) VAR4) CHAR)` and we get `VARCHAR2(6 CHAR)`. That is the result of the specific call of the function *concat3*.

```
create function concat3 ( a VARCHAR2
                        , b VARCHAR2
                        , c VARCHAR2
                        )
return VARCHAR2 is
begin
  return a||b||c;
end;

declare
  ret varchar2(10);
begin
  ret := concat3('a','bb', 'ccc');
end;
```

Listing 3.4: SQL program with a user-defined function with variable lengths

3.4.3 Checking user-defined procedures

Checking the procedures is very similar to functions. Instead of only one result (when dealing with a function), there can be multiple *OUT* parameters.

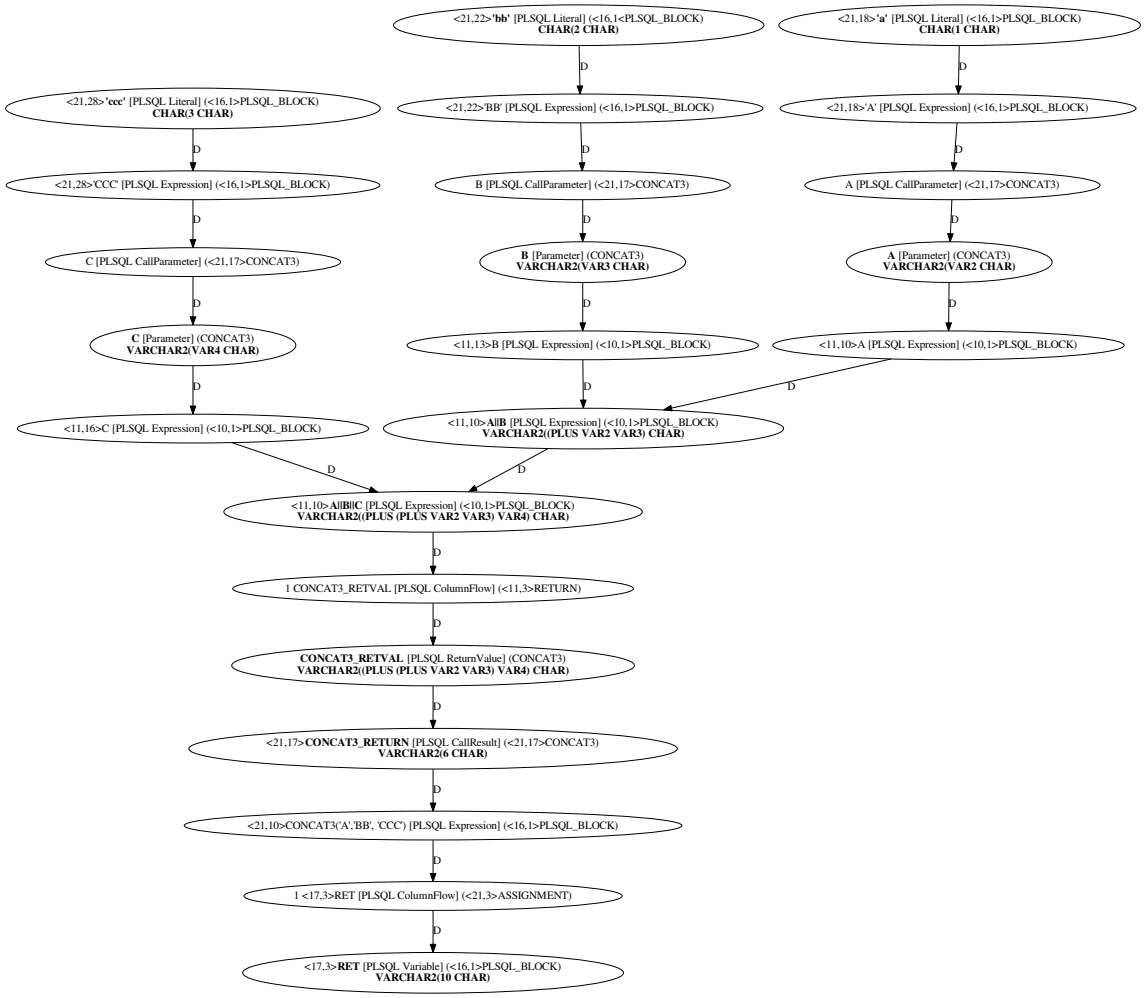
However, we only support the scenario where there is only one flow into the *OUT* parameter and the parameter is only *OUT* and not *IN*. If there are multiple flows, we don't have any procedure to chose one of it to get the "resulting datatype". So in that case, we set the size of the datatype as an unknown variable.

For example, if we define a procedure as `create procedure myproc (a in out VARCHAR2) is ...`, the datatype for parameter *a* will be `VARCHAR2(VAR`x`CHAR)`.

3.5 Graph checking summary

In this section we give a summary of how the checking on a graph works. The input is a dataflow graph, with nodes annotated with corresponding datatypes

Figure 3.5: Dataflow graph of program with a user-defined function with variable lengths



where the datatype is known (columns, variables). The output is a set of found dataflow incidents.

- all nodes with known datatype are gathered
- from each such node, a corresponding dataflow source node is searched
- if the datatype of the source node is already known, flow rules are applied and the resulting dataflow incident is reported

- if the node is an actual return of a built-in function (or an actual *out* parameter, in case of a procedure)
  - if the formal return's (or formal *out* parameter's) datatype is not known, it is inferred from its dataflow source node
  - the formal parameters are paired with the actual parameters corresponding to that particular call
  - the datatype of the actual return (*out* parameter) is computed with the environment (formal parameter $\rightarrow$ actual parameter)
- since the datatype is now known, the dataflow incident is computed and reported as described above

Chapter 4

Implementation

The datatypes, built-in functions and the dataflow checking procedure, as described in previous chapters, are implemented in Java 6. In this chapter we give an overview of the classes that make up the program.

The tools to parse the SQL program and to generate the resulting dataflow graph were readily available and used. Their design and implementation is not part of this thesis.

4.1 Auxiliary classes

Here we describe some of the auxiliary classes that we use in our implementation.

4.1.1 Environment

Class *Environment* consists merely of *public final* fields. These fields describe various properties of the Oracle PL/SQL dialect and other settings, like the default parameters for various parameterizable datatypes or the length of values of given datatypes when converted to a *character* datatype.

4.1.2 Flow

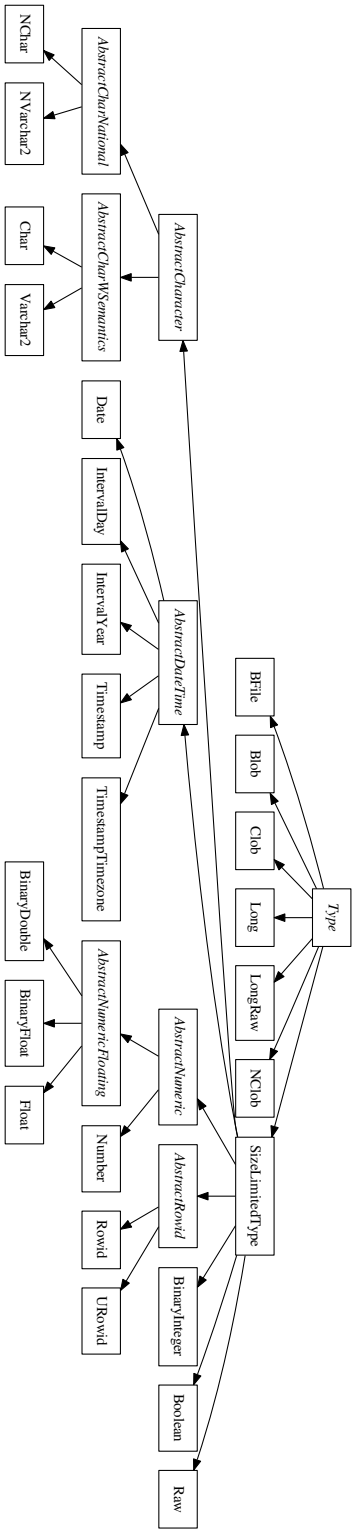
Flow is actually not a class but an *enum*. It describes the possible kinds of flow between datatypes, as described in 1.7.

4.1.3 FlowIncident

Class *FlowIncident* is used a flow between 2 particular nodes in the dataflow graph. It has the following fields:

- *flow : Flow* – what kind of flow is taking place there

Figure 4.1: *Type* class hierarchy



- *source : Node* – the source node in the dataflow graph
- *sourceType : Type* – the datatype corresponding to the source node
- *target : Node* – the target node in the dataflow graph
- *targetType : Type* – the datatype corresponding to the target node

4.1.4 FunctionFlow

Class *FunctionFlow* describes the effect a built-in function has at a specific place with the specific input. It has the following fields:

- *flows : PSequence<Flow>* – describes the kind of flow from the actual into the formal parameters
- *result : Type* – describes the resulting datatype (which may vary depending on the input)

4.2 Datatypes

In this section we describe the classes for representing datatypes of Oracle PL/SQL as we have described them in the chapter 1.

4.2.1 Type

The datatypes are represented by classes in a hierarchy, with the class *Type* as the root superclass. We see the class hierarchy in the figure 4.

Class *Type* is an abstract class, with methods *assignableTo*, *evalParams*, *fromString* and *inferType*.

- *fromString* is a static method that takes an *input : String* and *e : Environment* as parameters and returns a *Type*. *input* is a human readable representation of an instance of *Type*, which this method deserializes and returns.
- *inferType* is a static method that takes an *input : String* and *e : Environment* as parameters and returns a *Type*. *input* is a literal of some datatype. The method infers the datatype of this literal and returns it.
- *assignableTo* is an abstract method that takes a *t : Type* and *e : Environment* as parameters and returns a *Flow*. The method is supposed to return an appropriate *flow label* to the flow from *this* to *t*. Since the flow label is different for for different pairs of datatypes, each subclass of *Type* has to implement the method itself. All the classes implement the method is such a way that if they get a *Type* instance that they don't expect, the superclass implementation is used.

- *evalParams* is an abstract method that takes *env : PMap< Variable, Expression>* and returns *Type*. The method evaluates the datatype's parameters in the environment *env*. E.g. if *this* represents datatype `VARCHAR((PLUS VAR0 VAR1) CHAR)` and *env* is (VAR0 → 1, VAR1 → 2), then *this.evalParams(env)* gives a *Type* instance representing `VARCHAR(3 CHAR)`.

4.2.2 SizeLimitedType

Class *SizeLimitedType* is a subclass of *Type*. It represents all datatypes for which we can deduce their length, if they we converted to a *character* datatype.

It has an abstract method *maxFormatLen* that takes *e : Environment* as a parameter and returns *Expression*.

4.2.3 Character datatypes

The classes that represent *character* datatypes have the class *AbstractCharacter* as their superclass.

4.2.3.1 AbstractCharacter

Abstract class *AbstractCharacter* represents all *character* datatypes and is a subclass of *SizeLimitedType*. It has a field *size : Expression* that describes the upper limit for how long the strings can be. It implements the method *assignableTo* for the target classes *AbstractDateTime*, *AbstractNumeric*, and *BinaryInteger*. It also implements the method *maxFormatLen*, which returns *size*.

4.2.3.2 AbstractCharNational

Abstract class *AbstractCharacter* represents *character* datatypes with national characters (`NCHAR` and `NVARCHAR2`) and is a subclass of *AbstractCharacter*. It implements the method *assignableTo* for the target classes *NChar*, *NVarchar2*, *AbstractCharWSemantics*, *Long*, *Clob*, *NClob*, *Raw*, *Rowid* and *URowid*.

4.2.3.3 AbstractCharWSemantics

Abstract class *AbstractCharWSemantics* represents *character* datatypes with specifiable character semantics (`CHAR` and `VARCHAR2`) and is a subclass of *AbstractCharacter*. It has a field *characterSemantics : boolean* that describes if the datatype uses the *CHAR* semantics or not (*BYTE*). It implements the method *assignableTo* for the target classes *Char*, *Varchar2*, *AbstractCharNational* and *Raw*.

4.2.3.4 NChar

Concrete class *NChar* is a subclass of *AbstractCharNational* and represents the `NCHAR` datatype. It implements the method *evalParams* which returns a new *NChar* instance with the field *size* evaluated in the given environment. It also implements the method *toString* which returns `"NCHAR(" + size + ")"`.

4.2.3.5 NVarchar2

Concrete class *NVarchar2* is a subclass of *AbstractCharNational* and represents the `NVARCHAR2` datatype. It implements the method *evalParams* which returns a new *NVarchar2* instance with the field *size* evaluated in the given environment. It also implements the method *toString* which returns `"NVARCHAR2(" + size + ")"`.

4.2.3.6 Char

Concrete class *Char* is a subclass of *AbstractCharWSemantics* and represents the `CHAR` datatype. It implements the method *assignableTo* for the target classes *Long*, *Clob*, *Blob* and *NClob*. It implements the method *evalParams* which returns a new *Char* instance with the field *size* evaluated in the given environment and the same *characterSemantics*. It also implements the method *toString* which returns `"CHAR(" + size + " " + semantics + ")"`, where *semantics* = "*CHAR*" if *characterSemantics*, else "*BYTE*".

4.2.3.7 Varchar2

Concrete class *Varchar2* is a subclass of *AbstractCharWSemantics* and represents the `VARCHAR2` datatype. It implements the method *assignableTo* for the target classes *Long*, *Clob*, *NClob*, *Rowid* and *URowid*. It implements the method *evalParams* which returns a new *Varchar2* instance with the field *size* evaluated in the given environment and the same *characterSemantics*. It also implements the method *toString* which returns `"VARCHAR2(" + size + " " + semantics + ")"`, where *semantics* = "*CHAR*" if *characterSemantics*, else "*BYTE*".

4.2.4 Datetime datatypes

The classes that represent *datetime* datatypes have the class *AbstractDateTime* as their superclass.

4.2.4.1 AbstractDateTime

Abstract class *AbstractDateTime* represents all *datetime* datatypes and is a subclass of *SizeLimitedType*. It implements the method *assignableTo* for the

target class *AbstractCharacter*.

4.2.4.2 Date

Concrete class *Date* is a subclass of *AbstractDateTime* and represents the `DATE` datatype. It implements the method *assignableTo* for the target classes *Date*, *Timestamp* and *TimestampTimezone*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"DATE"`.

4.2.4.3 IntervalDay

Concrete class *IntervalDay* is a subclass of *AbstractDateTime* and represents the `INTERVAL DAY TO SECOND` datatype.

It has fields *dayPrecision : Expression* and *fracSecPrecision : Expression* that describes the upper limits for precision of days and fractions of second respectively.

It implements the method *assignableTo* for the target classes *IntervalDay*, *IntervalYear* and *Long*.

It implements the method *evalParams* which returns a new *IntervalDay* instance with the fields *dayPrecision* and *fracSecPrecision* evaluated in the given environment.

It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*.

It also implements the method *toString* which returns `"INTERVAL DAY(" + dayPrecision + ") TO SECOND(" + fracSecPrecision + ")"`.

4.2.4.4 IntervalYear

Concrete class *IntervalYear* is a subclass of *AbstractDateTime* and represents the `INTERVAL YEAR TO MONTH` datatype.

It has a field *yearPrecision : Expression* that describes the upper limit for precision of years.

It implements the method *assignableTo* for the target classes *IntervalYear*, *IntervalDay* and *Long*.

It implements the method *evalParams* which returns a new *IntervalYear* instance with the field *yearPrecision* evaluated in the given environment.

It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*.

It also implements the method *toString* which returns `"INTERVAL YEAR(" + yearPrecision + ") TO MONTH"`.

4.2.4.5 Timestamp

Concrete class *Timestamp* is a subclass of *AbstractDateTime* and represents the `TIMESTAMP` datatype.

It has a field *fracSecPrecision : Expression* that describes the upper limit for precision of fractions of second.

It implements the method *assignableTo* for the target classes *Timestamp*, *TimestampTimezone*, *Date* and *Long*.

It implements the method *evalParams* which returns a new *Timestamp* instance with the field *fracSecPrecision* evaluated in the given environment.

It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*.

It also implements the method *toString* which returns `"TIMESTAMP(" + fracSecPrecision + ")"`.

4.2.4.6 TimestampTimezone

Concrete class *TimestampTimezone* is a subclass of *AbstractDateTime* and represents the `TIMESTAMP WITH TIME ZONE` datatype.

It has a field *fracSecPrecision : Expression* that describes the upper limit for precision of fractions of second.

It implements the method *assignableTo* for the target classes *TimestampTimezone*, *Timestamp*, *Date* and *Long*.

It implements the method *evalParams* which returns a new *TimestampTimezone* instance with the field *fracSecPrecision* evaluated in the given environment.

It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*.

It also implements the method *toString* which returns `"TIMESTAMP(" + fracSecPrecision + ") WITH TIME ZONE"`.

4.2.5 Large object datatypes

4.2.5.1 BFile

Concrete class *BFile* is a subclass of *Type* and represents the `BFILE` datatype. It implements the method *assignableTo* for the target class *BFile*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"BFILE"`.

4.2.5.2 Blob

Concrete class *Blob* is a subclass of *Type* and represents the `BLOB` datatype. It implements the method *assignableTo* for the target classes *Blob*, *Raw* and *LongRaw*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"BLOB"`.

4.2.5.3 Clob

Concrete class *Clob* is a subclass of *Type* and represents the `CLOB` datatype. It implements the method *assignableTo* for the target classes *Clob*, *NClob*, *AbstractCharacter* and *Long*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"CLOB"`.

4.2.5.4 NClob

Concrete class *NClob* is a subclass of *Type* and represents the `NCLOB` datatype. It implements the method *assignableTo* for the target classes *NClob*, *Clob*, *AbstractCharacter* and *Long*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"NCLOB"`.

4.2.6 Long and raw datatypes

4.2.6.1 Long

Concrete class *Long* is a subclass of *Type* and represents the `LONG` datatype. It implements the method *assignableTo* for the target classes *Long*, *AbstractCharacter*, *Raw*, *Clob* and *NClob*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"LONG"`.

4.2.6.2 LongRaw

Concrete class *LongRaw* is a subclass of *Type* and represents the `LONG RAW` datatype. It implements the method *assignableTo* for the target classes *LongRaw*, *AbstractCharacter*, *Raw*, *Long* and *Blob*. It implements the method *evalParams* which returns the same instance. It also implements the method *toString* which returns `"LONG RAW"`.

4.2.6.3 Raw

Concrete class *Raw* is a subclass of *SizeLimitedType* and represents the `RAW` datatype. It has a field *size : Expression*. It implements the method *assignableTo* for the target classes *Raw*, *AbstractCharacter*, *LongRaw*, *Long* and *Blob*. It implements the method *evalParams* which returns a new *Raw* instance with the field *size* evaluated in the given environment. It also implements the method *toString* which returns `"RAW(" + size + ")"`.

4.2.7 Numeric datatypes

The classes that represent *numeric* datatypes have the class *AbstractNumeric* as their superclass, which is itself a subclass of *SizeLimitedType*.

4.2.7.1 AbstractNumericFloating

Abstract class *AbstractNumericFloating* represents all floating-point *numeric* datatypes and is a subclass of *AbstractNumeric*. It implements the method *assignableTo* for the target classes *Number*, *BinaryInteger*, and *AbstractCharacter*.

4.2.8 BinaryDouble

Concrete class *BinaryDouble* is a subclass of *AbstractNumericFloating* and represents the `BINARY_DOUBLE` datatype. It implements the method *assignableTo* for the target classes *BinaryDouble*, *BinaryFloat* and *Float*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*. It also implements the method *toString* which returns `"BINARY_DOUBLE"`.

4.2.9 BinaryFloat

Concrete class *BinaryFloat* is a subclass of *AbstractNumericFloating* and represents the `BINARY_FLOAT` datatype. It implements the method *assignableTo* for the target classes *BinaryFloat*, *BinaryDouble* and *Float*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*. It also implements the method *toString* which returns `"BINARY_FLOAT"`.

4.2.10 Float

Concrete class *Float* is a subclass of *AbstractNumericFloating* and represents the `FLOAT` datatype. It has a field *binaryPrecision : Expression* that defines its maximal binary precision. It implements the method *assignableTo* for the target classes *Float*, *BinaryFloat* and *BinaryDouble*. It implements the method *evalParams* which returns a new *Float* instance with the field *binaryPrecision* evaluated in the given environment. It implements the method *maxFormatLen* which returns the length as specified by the given *Environment*. It also implements the method *toString* which returns `"FLOAT(" + binaryPrecision + ")"`.

4.2.10.1 Number

Concrete class *Number* is a subclass of *AbstractNumeric* and represents the `NUMBER` datatype. It has fields *precision : Expression* and *scale : Expression*

that describes the upper limits for precision and scale respectively. It implements the method *assignableTo* for the target classes *Number*, *BinaryInteger*, *BinaryFloat*, *BinaryDouble*, *AbstractCharacter*, *Clob* and *NClob*. It implements the method *evalParams* which returns a new *Number* instance with the fields *precision* and *scale* evaluated in the given environment. It implements the method *maxFormatLen* which returns the length as:

- $precision + 1$ if $scale > 0$ and $precision > scale$
- $scale + 2$ if $scale > 0$ and $precision \leq scale$
- $precision - scale$ if $scale \leq 0$

It also implements the method *toString* which returns `"NUMBER(" + precision + ", " + scale + ")"`.

4.2.11 PL/SQL datatypes

4.2.11.1 BinaryInteger

Concrete class *BinaryInteger* is a subclass of *SizeLimitedType* and represents the `BINARY_INTEGER` datatype. It implements the method *assignableTo* for the target classes *BinaryInteger*, *Number*, *BinaryFloat*, *BinaryDouble* and *AbstractCharacter*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns the maximal number of digits as specified by the given *Environment* + 1 (for sign). It also implements the method *toString* which returns `"BINARY_INTEGER"`.

4.2.11.2 Boolean

Concrete class *Boolean* is a subclass of *SizeLimitedType* and represents the `BOOLEAN` datatype. It implements the method *assignableTo* for the target classes *Boolean*, for all the other classes it is returns *ConversionSafe*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns 5. It also implements the method *toString* which returns `"BOOLEAN"`.

4.2.12 Rowid datatypes

4.2.12.1 AbstractRowid

Abstract class *AbstractRowid* represents all *rowid* datatypes and is a subclass of *SizeLimitedType*. It implements the method *assignableTo* for the target class *AbstractCharacter*.

4.2.12.2 Rowid

Concrete class *Rowid* is a subclass of *AbstractRowid* and represents the `ROWID` datatype. It implements the method *assignableTo* for the target class *Rowid*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns the maximal length as specified by the given *Environment*. It also implements the method *toString* which returns `"ROWID"`.

4.2.12.3 URowid

Concrete class *URowid* is a subclass of *AbstractRowid* and represents the `UROWID` datatype. It has a field *size : int* that defines its size. It implements the method *assignableTo* for the target class *URowid*. It implements the method *evalParams* which returns the same instance. It implements the method *maxFormatLen* which returns the maximal length as specified by the given *Environment*. It also implements the method *toString* which returns `"UROWID(" + size + ")"`.

4.3 Built-in functions

In this section we describe the classes for representing built-in functions of Oracle PL/SQL as we have described them in chapter 2.

4.3.1 BuiltInFunction

The datatypes are represented by classes in a hierarchy, with the class *BuiltInFunction* as the root superclass. The class has these methods:

- *inferFunctionType* is a static method that takes *input : String* and *e : Environment* as parameters, and returns a *BuiltInFunction*. *input* is the name of the built-in function and the method returns its representation.
- *checkFlow* is an abstract method that takes *parameters : PSequence<Type>* and *e : Environment* as parameters, and returns a *FunctionFlow*. *parameters* gives the datatypes of the actual parameters of the built-in function.

4.3.2 Numeric functions

The classes that represent the *numeric* built-in function functions are these:

- *Abs* represents the function `ABS`.
- *Bitand* represents the function `BITAND`.
- *CeilFloor* represents the functions `CEIL` and `FLOOR`.

- *RoundTrunc* represents the functions `ROUND` and `TRUNC`.
- *Sign* represents the function `SIGN`.
- *Numeric2DoubleOneParam* represents the functions `ACOS`, `ASIN`, `ATAN`, `COS`, `COSH`, `EXP`, `LN`, `SIN`, `SINH`, `TAN`, `TANH` and `SQRT`.
- *Numeric2DoubleTwoParams* represents the functions `ATAN2`, `LOG`, `MOD`, `NANVL`, `POWER` and `REMAINDER`.

4.3.3 Character functions

The classes that represent the *character* built-in function are these:

- *Chr* represents the function `CHR`.
- *Concat* represents the function `CONCAT` and the operator `||`.
- *OneConstParam* represents the functions `INITCAP`, `LOWER` and `UPPER`.
- *XPad* represents the functions `LPAD` and `RPAD`.
- *XTrim* represents the functions `LTRIM` and `RTRIM`.
- *NlsMod* represents the functions `NLS_INITCAP`, `NLS_LOWER`, `NLSSORT` and `NLS_UPPER`.
- *RegexpReplace* represents the function `REGEXP_REPLACE`.
- *RegexpSubstr* represents the function `REGEXP_SUBSTR`.
- *Replace* represents the function `REPLACE`.
- *Substr* represents the functions `SUBSTR`, `SUBSTRB`, `SUBSTRC`, `SUBSTR2` and `SUBSTR4`.
- *Soundex* represents the function `SOUNDEX`.
- *Translate* represents the function `TRANSLATE`.

4.3.4 Character to numeric functions

The classes that represent the *character* built-in function functions returning *numeric* values are these:

- *Ascii* represents the function `ASCII`.
- *Instr* represents the functions `INSTR`, `INSTRB`, `INSTRC`, `INSTR2`, and `INSTR4`.
- *Length* represents the functions `LENGTH`, `LENGTHB`, `LENGTHC`, `LENGTH2`, and `LENGTH4`.
- *RegexInstr* represents the function `REGEXP_INSTR`.

4.3.5 Operators

- The class *Plus* represents the operator `+`.
- The class *Minus* represents the operator `-`.
- The class *Multiply* represents the operator `*`.
- The class *Divide* represents the operator `/`.

4.3.6 Aggregate functions

The aggregate built-in functions are represented by the class *AggregateOneParamWithAnalClause*.

4.4 Datatype parameters

In this section we describe the classes that we use for symbolic description of parameters in (some) of the datatypes, as we have described this in the section 2.1. These parameters are represented by a class hierarchy, with the class *Expression* as the root superclass. The class has the following methods:

- *fromString* is a static method. It takes *input : String* and returns a new instance of *Expression*. This method is used for deserializing from human readable form.
- *weakLessOrEq* and *weakGreaterOrEq* are abstract methods that are used for weak comparing two expressions.
- *eval* is an abstract method that takes *env : PMap< Variable, Expression>*, evaluates the current instance in the given environment *env* and returns the resulting *Expression*.

The subclasses of *Expression* are:

- *Int* that represents an integer value
- *Variable* that represents a variable. It has a field *id: private final int* that is used to distinguish between two variables.
- *IntOp* itself is a superclass for all binary operations. The abstract methods `Expression constructor(Expression e1, Expression e2)` and `int op(int x, int y)` are used in the implementation of the method *eval*. It has these subclasses are:
  - *Plus* represents addition.
  - *Minus* represents subtraction.
  - *Max* represents the maximum.
  - *Min* represents the minimum.

4.5 Checking the dataflow

Class *FlowChecker* is used for the actual checking of dataflow in the graph. The constructor takes an instance of the class *Graph*, which is the dataflow where we want to perform the analysis. The method has these methods:

- *checkFlow* is the most important method. It returns a *PSet<FlowIncident>*. It gathers the flow labels for all dataflow pairs according to the rules. For this it uses the following auxiliary methods.
- *getTypedNodes* returns nodes that have a corresponding datatype specified (those could be columns or variables)
- *groupParams* groups together nodes of actual parameters of built-in functions with the actual result; the formal parameters of user-defined functions (and procedures) and the formal result; and the actual parameters of user-defined functions (and procedures) and the actual result
- *getFunname2return* returns a mapping from a user-defined function to its return node
- *getSources* returns all the nodes that flow into the target node. This method also computes the datatype, if the source node corresponds to a result of an operator, function or an *OUT* parameter of a procedure. When it computes the resulting datatype, it also registers the dataflow incidents caused by the flow into the operator/function/procedure parameters.

4.6 Checking programs over multiple files

The SQL programs are often written over many files, so we support this scenario as well. First a symbolic datatype of function's formal return (or procedure's formal *out* parameter) has to be computed. All the nodes with computed datatype annotations are then saved in a repository.

Whenever we came across an actual return (or an *out* parameter) that belongs to a function (procedure) defined in another file, we get the nodes for formal return (or *out* parameter) and formal parameters from the repository (and don't use the ones present in the current dataflow graph). We use these nodes for pairing with actual parameters and for evaluation for the actual result (*out* parameter).

Chapter 5

Experimental Evaluation

In this chapter we present a sample Oracle SQL program 5.1, with mistakes intentionally put in. The program was analyzed by out data compatibility checker. The found dataflow incidents were reported 5.2.

```
create table t1 (c1 number(5,1), c2 varchar2(10));
create table t2 (c3 varchar2(20), c4 varchar2(20));

insert into t1 (c1, c2) values (123.45, 'abcdefghijklmn');
insert into t1 (c1, c2) values (12345.67, 'xyz');

insert into t1 (c1, c2) select c3, c4 from t2;

create or replace function fun1 (a VARCHAR2, b VARCHAR2, c number(3,0))
return VARCHAR2 is
begin
  return a || '␣' || b || '␣qwer␣' ||c;
end;
/

declare
  bytestring varchar2(10 byte);
  charstring varchar2(12 char);
  blob1 blob;
begin
  bytestring := charstring;
  charstring := 1234567891234;
  charstring := fun1('abc', 'def', 987.6);
  select c1 into charstring from t1;
  blob1 := 'abc';
  blob1 := bytestring;
end;
/
```

Listing 5.1: Sample SQL program full of mistakes

```
flow:    ConversionWrongSize
source:  CHAR(14 CHAR)      <4,41>'abcdefghijklmn' [PLSQL
Literal] (test5.sql)
target:  VARCHAR2(10 CHAR)    C2 [Column] (T1)

flow:    ConversionSafe
source:  CHAR(3 CHAR)       <23,22>'abc' [PLSQL Literal]
(<16,1>PLSQL_BLOCK)
target:  VARCHAR2(VAR2 CHAR)     A [Parameter] (FUN1)

flow:    ConversionWrongSize
source:  NUMBER(VAR60, VAR61)     <22,17>1234567891234 [PLSQL
Literal] (<16,1>PLSQL_BLOCK)
target:  VARCHAR2(12 CHAR)    <18,3>CHARSTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)

flow:    ConversionSafe
source:  CHAR(3 CHAR)       <5,43>'xyz' [PLSQL Literal]
(test5.sql)
target:  VARCHAR2(10 CHAR)    C2 [Column] (T1)

flow:    ConversionSafe
source:  NUMBER(VAR49, VAR50)    C [Parameter] (FUN1)
target:  <12,38>C [PLSQL Expression] (<11,1>PLSQL_BLOCK)

flow:    Safe
source:  VARCHAR2((PLUS (PLUS (PLUS VAR2 1) VAR3) 6)
CHAR)    <12,10>A || ' ' ||...| ' QWER ' [PLSQL Expression]
(<11,1>PLSQL_BLOCK)
target:  <12,10>A || ' ' ||...| ' QWER ' [PLSQL Expression]
(<11,1>PLSQL_BLOCK)

flow:    WrongSize
source:  VARCHAR2(12 CHAR)    <18,3>CHARSTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)
target:  VARCHAR2(10 BYTE)    <17,3>BYTESTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)

flow:    Safe
source:  CHAR(1 CHAR)       <12,15>' ' [PLSQL Literal]
(<11,1>PLSQL_BLOCK)
target:  <12,15>' ' [PLSQL Expression] (<11,1>PLSQL_BLOCK)

flow:    WrongSize
source:  VARCHAR2((PLUS 13 VAR51) CHAR)      <23,21>FUN1_RETURN
[PLSQL CallResult] (<23,21>FUN1)
target:  VARCHAR2(12 CHAR)    <18,3>CHARSTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)
```

```
flow:    Safe
source:  VARCHAR2(VAR2 CHAR)     A [Parameter] (FUN1)
target:  <12,10>A [PLSQL Expression] (<11,1>PLSQL_BLOCK)

flow:    Safe
source:  VARCHAR2(VAR3 CHAR)     B [Parameter] (FUN1)
target:  <12,22>B [PLSQL Expression] (<11,1>PLSQL_BLOCK)

flow:    ConversionSafe
source:  CHAR(3 CHAR)      <23,29>'def' [PLSQL Literal]
(<16,1>PLSQL_BLOCK)
target:  VARCHAR2(VAR3 CHAR)     B [Parameter] (FUN1)

flow:    Incompatible
source:  VARCHAR2(10 BYTE)    <17,3>BYTESTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)
target:  BLOB     <19,3>BLOB1 [PLSQL Variable]
(<16,1>PLSQL_BLOCK)

flow:    ConversionUnsafe
source:  VARCHAR2(20 CHAR)    C3 [Column] (T2)
target:  NUMBER(VAR35, VAR36)    C1 [Column] (T1)

flow:    Safe
source:  VARCHAR2((PLUS (PLUS VAR2 1) VAR3) CHAR)     <12,10>A
|| ' ' || B [PLSQL Expression] (<11,1>PLSQL_BLOCK)
target:  <12,10>A || ' ' || B [PLSQL Expression]
(<11,1>PLSQL_BLOCK)

flow:    ConversionSafe
source:  CHAR(3 CHAR)      <25,12>'abc' [PLSQL Literal]
(<16,1>PLSQL_BLOCK)
target:  BLOB     <19,3>BLOB1 [PLSQL Variable]
(<16,1>PLSQL_BLOCK)

flow:    Safe
source:  NUMBER(VAR37, VAR38)    <5,33>12345.67 [PLSQL Literal]
(test5.sql)
target:  NUMBER(VAR39, VAR40)    C1 [Column] (T1)

flow:    WrongSize
source:  VARCHAR2(20 CHAR)    C4 [Column] (T2)
target:  VARCHAR2(10 CHAR)    C2 [Column] (T1)

flow:    Safe
source:  VARCHAR2((PLUS VAR2 1) CHAR)     <12,10>A || ' ' [PLSQL
Expression] (<11,1>PLSQL_BLOCK)
target:  <12,10>A || ' ' [PLSQL Expression]
(<11,1>PLSQL_BLOCK)
```

```
flow:    Safe
source:  CHAR(6 CHAR)      <12,27>' qwer ' [PLSQL Literal]
(<11,1>PLSQL_BLOCK)
target:  <12,27>' QWER ' [PLSQL Expression]
(<11,1>PLSQL_BLOCK)

flow:    Safe
source:  NUMBER(VAR68, VAR69)    <23,36>987.6 [PLSQL Literal]
(<16,1>PLSQL_BLOCK)
target:  NUMBER(VAR70, VAR71)    C [Parameter] (FUN1)

flow:    Safe
source:  NUMBER(VAR41, VAR42)    <4,33>123.45 [PLSQL Literal]
(test5.sql)
target:  NUMBER(VAR43, VAR44)    C1 [Column] (T1)

flow:    ConversionWrongSize
source:  NUMBER(VAR63, VAR64)    C1 [Column] (T1)
target:  VARCHAR2(12 CHAR)    <18,3>CHARSTRING [PLSQL Variable]
(<16,1>PLSQL_BLOCK)
```

Listing 5.2: Output of the datatype compatibility checker

Conclusion and Future Work

We have presented a design and implementation of Oracle SQL datatype compatibility checker.

For this, we have analyzed the available datatypes in Oracle SQL and their literals. We have formalized the rules for flow between the most used ones. Those rules describe the nature of flow between two given datatypes – if it is even possible, loss-less, etc. Special attention has been paid to the conversion to *character* datatypes that gives us information about the length of the result.

We have described a subset of built-in functions from Oracle SQL. User-defined functions and procedures have been also discussed. We have described a simple language for describing parameters of functions/procedures.

We have looked at the dataflow graph corresponding to an SQL program. An algorithm for checking datatype compatibility that performs the checking on this graph and also works with both built-in and user-defined functions and procedures, has been described.

The representation of datatypes, built-in functions and the datatype compatibility checker itself was implemented in the Java 6 language. We have given an overview of the important classes in the implementation. The checker has been incorporated into Manta Tools.

Oracle PL/SQL is a huge language, with many constructs, datatypes, built-in functions and an object system. For the purpose of this thesis, we have implemented only a subset of the available datatypes and built-in functions; and we have omitted the object-system entirely. So an obvious way to extend this work is to implement the rest of the datatypes and built-in functions and consider a way to check objects.

SQL enables programmers to annotate columns with various constraints, such as `NOT NULL`. So another interesting extension of the checker could prevent the `NULL` value from flowing into a `NOT NULL` column.

The current datatype compatibility checker is designed to work only on Oracle PL/SQL. But the basic implementation is extensible to another dialect. New subclasses for *Type* and *BuiltInFunction* would have to be implemented.

Bibliography

[1] Watson, J.; Ramklass, R. *OCA Oracle Database 11g SQL Fundamentals I Exam Guide*. Oracle Press, McGraw-Hill Osborne Media, June 2008.

[2] Aho, A. V.; Lam, M. S.; Sethi, R.; et al. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, September 2006.

[3] Profinit. Manta Tools. Available from: `http://mantatools.com`

[4] Oracle. Database SQL Language Reference – Oracle Data Types. Available from: `https://docs.oracle.com/cd/B28359_01/server.111/b28286/sql_elements001.htm`

[5] Oracle. Database SQL Language Reference – About SQL Functions. Available from: `https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions001.htm`

[6] Oracle. Database SQL Language Reference – Datatypes. Available from: `https://docs.oracle.com/cd/B28359_01/server.111/b28286/sql_elements001.htm`

Appendix A

Flow rules

Here we list the flow rules between datatypes. We list only the rules between convertible datatypes. A flow between datatypes that are not convertible is implicitly labeled as *Incompatible*.

A.1 Character

A.1.1 CHAR

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{CHAR}(size_b \;\; sem_b) \quad sem_a = sem_b \vee sem_b = \texttt{char} \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{CHAR}(size_b \;\; sem_b) \quad size_a > size_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; \texttt{char}) \quad b : \texttt{CHAR}(size_b \;\; \texttt{byte})}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{VARCHAR2}(size_b \;\; sem_b) \quad sem_a = sem_b \vee sem_b = \texttt{char} \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{VARCHAR2}(size_b \;\; sem_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; \texttt{CHAR}) \quad b : \texttt{VARCHAR2}(size_b \;\; \texttt{byte})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{NCHAR|NVARCHAR2}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{NCHAR|NVARCHAR2}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{CHAR} \quad b : \texttt{NUMBER|FLOAT|BINARY\_FLOAT|BINARY\_DOUBLE}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; \texttt{BYTE}) \quad b : \texttt{RAW}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{CHAR}(size_a \;\; sem_a) \quad b : \texttt{RAW}(size_b) \quad size_a > size_b \vee sem_a = \texttt{char}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{CHAR} \quad b : \texttt{DATE|TIMESTAMP|T.W.TIMEZONE|INTERVAL YEAR|INTERVAL DAY}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{CHAR} \quad b : \texttt{LONG|BLOB|CLOB|NCLOB}}{a \xrightarrow[ConversionUnsafe]{} b}$$

A.1.2 VARCHAR2

$$\frac{a : \texttt{VARCHAR2}(size_a \;\; sem_a) \quad b : \texttt{VARCHAR2}(size_b \;\; sem_b) \quad sem_a = sem_b \vee sem_b = \texttt{char} \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a \;\; sem_a) \quad b : \texttt{VARCHAR2}(size_b \;\; sem_b) \quad size_a > size_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ \texttt{char}) \quad b : \texttt{VARCHAR2}(size_b\ \texttt{byte})}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{CHAR}(size_b\ sem_b) \quad sem_a = sem_b \vee sem_b = \texttt{char} \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{CHAR}(size_b\ sem_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ \texttt{char}) \quad b : \texttt{CHAR}(size_b\ \texttt{byte})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{NCHAR|NVARCHAR2}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{NCHAR|NVARCHAR2}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2} \quad b : \texttt{NUMBER|FLOAT|BINARY\_FLOAT|BINARY\_DOUBLE}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ \texttt{byte}) \quad b : \texttt{RAW}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{RAW}(size_b) \quad size_a > size_b \vee sem_a = \texttt{char}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{VARCHAR2} \quad b : \texttt{DATE|TIMESTAMP|T.W.TIMEZONE|INTERVAL YEAR|INTERVAL DAY}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2}(size_a\ sem_a) \quad b : \texttt{LONG|CLOB|NCLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{VARCHAR2} \quad b : \texttt{ROWID|UROWID}}{a \xrightarrow[ConversionUnsafe]{} b}$$

A.1.3 NCHAR

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a > size_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{NVARCHAR2}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{NVARCHAR2}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{CHAR|VARCHAR2}(size_b \ \texttt{char}) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NCHAR}(size_a) \quad b : \texttt{CHAR|VARCHAR2}(size_b \ sem_b) \quad size_a > size_b \vee sem_b = \texttt{byte}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NCHAR} \quad b : \texttt{NUMBER|FLOAT|BINARY\_FLOAT|BINARY\_DOUBLE}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{NCHAR} \quad b : \texttt{RAW}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NCHAR} \quad b : \texttt{DATE|TIMESTAMP|T.W.TIMEZONE|INTERVAL YEAR|INTERVAL DAY}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{NCHAR} \quad b : \texttt{LONG|CLOB|NCLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NCHAR} \quad b : \texttt{ROWID|UROWID}}{a \xrightarrow[ConversionUnsafe]{} b}$$

A.1.4 NVARCHAR2

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{NVARCHAR2}(size_b) \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{NVARCHAR2}(size_b) \quad size_a > size_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{NCHAR}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{CHAR|VARCHAR2}(size_b \ \texttt{char}) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2}(size_a) \quad b : \texttt{CHAR|VARCHAR2}(size_b \ \ sem_b) \quad size_a > size_b \vee sem_b = \texttt{byte}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NVARCHAR2} \quad b : \texttt{NUMBER|FLOAT|BINARY\_FLOAT|BINARY\_DOUBLE}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2} \quad b : \texttt{RAW}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NVARCHAR2} \quad b : \texttt{DATE|TIMESTAMP|T.W.TIMEZONE|INTERVAL YEAR|INTERVAL DAY}}{a \xrightarrow[ConversionUnsafe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2} \quad b : \texttt{LONG|CLOB|NCLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NVARCHAR2} \quad b : \texttt{ROWID|UROWID}}{a \xrightarrow[ConversionUnsafe]{} b}$$

A.2 Number

A.2.1 NUMBER

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{NUMBER}(p_b, s_b) \quad -s_b \leq -s_a \leq p_a - s_a \leq p_b - s_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{NUMBER}(p_b, s_b) \quad -s_b \leq p_a - s_a \leq p_b - s_b}{a \xrightarrow[Imprecise]{} b}$$

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{NUMBER}(p_b, s_b) \quad p_a - s_a < -s_b \ \vee \ p_b - s_b < p_a - s_a}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{NUMBER}(p_a, s_a) \quad b : \texttt{BINARY\_FLOAT|BINARY\_DOUBLE}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{NUMBER}(p, s) \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad len(\texttt{NUMBER}(p, s)) \leq size}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NUMBER}(p, s) \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad len(\texttt{NUMBER}(p, s)) > size}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.2.2 FLOAT

$$\frac{a : \texttt{FLOAT}(p_a) \quad b : \texttt{FLOAT}(p_b) \quad p_a \leq p_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{FLOAT}(p_a) \quad b : \texttt{FLOAT}(p_b) \quad p_a > p_b}{a \xrightarrow[Imprecise]{} b}$$

$$\frac{a : \texttt{FLOAT} \quad b : \texttt{BINARY\_FLOAT}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{FLOAT} \quad b : \texttt{BINARY\_DOUBLE}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{FLOAT} \quad b : \texttt{NUMBER}}{a \xrightarrow[ConversionImpreciseUnsafe]{} b}$$

$$\frac{a : \texttt{FLOAT} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size \geq len(\texttt{FLOAT})}{a \xrightarrow[ConversionImpreciseUnsafe]{} b}$$

$$\frac{a : \texttt{FLOAT} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size < len(\texttt{FLOAT})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.2.3 BINARY_FLOAT

$$\frac{a : \texttt{BINARY\_FLOAT} \quad b : \texttt{BINARY\_FLOAT}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{BINARY\_FLOAT} \quad b : \texttt{BINARY\_DOUBLE}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{BINARY\_FLOAT} \quad b : \texttt{NUMBER}}{a \xrightarrow[ConversionImpreciseUnsafe]{} b}$$

$$\frac{a : \texttt{BINARY\_FLOAT} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size \geq len(\texttt{BINARY\_FLOAT})}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{BINARY\_FLOAT} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size < len(\texttt{BINARY\_FLOAT})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.2.4 BINARY_DOUBLE

$$\frac{a : \texttt{BINARY\_DOUBLE} \quad b : \texttt{BINARY\_DOUBLE}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{BINARY\_DOUBLE} \quad b : \texttt{BINARY\_FLOAT}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{BINARY\_DOUBLE} \quad b : \texttt{NUMBER}}{a \xrightarrow[ConversionImpreciseUnsafe]{} b}$$

$$\frac{a : \texttt{BINARY\_DOUBLE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size \geq len(\texttt{BINARY\_DOUBLE})}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{BINARY\_DOUBLE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size < len(\texttt{BINARY\_DOUBLE})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.3 Long and Raw

A.3.1 LONG

$$\frac{a : \texttt{LONG} \quad b : \texttt{LONG}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{LONG} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{LONG} \quad b : \texttt{RAW|CLOB|NCLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

A.3.2 LONG RAW

$$\frac{a : \texttt{LONG RAW} \quad b : \texttt{LONG RAW}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{LONG RAW} \quad b : \texttt{RAW}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{LONG RAW} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{LONG RAW} \quad b : \texttt{LONG|BLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

A.3.3 RAW

$$\frac{a : \texttt{RAW}(size_a) \quad b : \texttt{RAW}(size_a) \quad size_a \leq size_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{RAW}(size_a) \quad b : \texttt{RAW}(size_a) \quad size_a > size_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{RAW}(size_a) \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_a \leq size_b}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{RAW}(size_a) \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_a > size_b}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{RAW} \quad b : \texttt{LONG RAW|LONG|BLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

A.4 Datetime

A.4.1 DATE

$$\frac{a : \texttt{DATE} \quad b : \texttt{DATE}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{DATE} \quad b : \texttt{TIMESTAMP|TIMESTAMP WITH TIMEZONE}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{DATE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b \geq len(\texttt{DATE})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{DATE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b < len(\texttt{DATE})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.4.2 TIMESTAMP

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{TIMESTAMP}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{TIMESTAMP WITH TIMEZONE}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{DATE}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b \geq len(\texttt{TIMESTAMP})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b < len(\texttt{TIMESTAMP})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{TIMESTAMP} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionSafe]{} b}$$

A.4.3 TIMESTAMP WITH TIMEZONE

$$\frac{a : \texttt{TIMESTAMP WITH TIMEZONE} \quad b : \texttt{TIMESTAMP WITH TIMEZONE}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{TIMESTAMP WITH TIMEZONE} \quad b : \texttt{DATE|TIMESTAMP}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{TIMESTAMP WITH TIMEZONE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b \geq len(\texttt{TIMESTAMP WITH TIMEZONE})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{TIMESTAMP WITH TIMEZONE} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b < len(\texttt{TIMESTAMP WITH TIMEZONE})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{TIMESTAMP WITH TIMEZONE} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionSafe]{} b}$$

A.4.4 INTERVAL YEAR TO MONTH

$$\frac{a : \texttt{INTERVAL YEAR}(y_a)\ \texttt{TO MONTH} \quad b : \texttt{INTERVAL YEAR}(y_a)\ \texttt{TO MONTH} \quad y_a \leq y_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{INTERVAL YEAR}(y_a)\ \texttt{TO MONTH} \quad b : \texttt{INTERVAL YEAR}(y_a)\ \texttt{TO MONTH} \quad y_a > y_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{INTERVAL YEAR TO MONTH} \quad b : \texttt{INTERVAL DAY TO SECOND}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{INTERVAL YEAR TO MONTH} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b \geq len(\texttt{INTERVAL YEAR TO MONTH})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{INTERVAL YEAR TO MONTH} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b < len(\texttt{INTERVAL YEAR TO MONTH})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{INTERVAL YEAR TO MONTH} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionSafe]{} b}$$

A.4.5 INTERVAL DAY TO SECOND

$$\frac{a : \texttt{INTERVAL DAY}(d_a)\ \texttt{TO SECOND}(s_a) \quad b : \texttt{INTERVAL DAY}(d_b)\ \texttt{TO SECOND}(s_b) \quad d_a \leq d_b \quad s_a \leq s_b}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{INTERVAL DAY}(d_a)\ \texttt{TO SECOND}(s_a) \quad b : \texttt{INTERVAL DAY}(d_b)\ \texttt{TO SECOND}(s_b) \quad d_a > d_b \ \vee \ s_a > s_b}{a \xrightarrow[WrongSize]{} b}$$

$$\frac{a : \texttt{INTERVAL DAY TO SECOND} \quad b : \texttt{INTERVAL YEAR TO MONTH}}{a \xrightarrow[ConversionImprecise]{} b}$$

$$\frac{a : \texttt{INTERVAL DAY TO SECOND} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b \geq len(\texttt{INTERVAL DAY TO SECOND})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{INTERVAL DAY TO SECOND} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}(size_b) \quad size_b < len(\texttt{INTERVAL DAY TO SECOND})}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{INTERVAL DAY TO SECOND} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionSafe]{} b}$$

A.5 Large Object

A.5.1 BLOB

$$\frac{a : \texttt{BLOB} \quad b : \texttt{BLOB}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{BLOB} \quad b : \texttt{RAW|LONG RAW}}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.5.2 CLOB

$$\frac{a : \texttt{CLOB} \quad b : \texttt{CLOB}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{CLOB} \quad b : \texttt{NCLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{CLOB} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{CLOB} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.5.3 NCLOB

$$\frac{a : \texttt{NCLOB} \quad b : \texttt{NCLOB}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{NCLOB} \quad b : \texttt{CLOB}}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{NCLOB} \quad b : \texttt{CHAR|VARCHAR2|NCHAR|NVARCHAR2}}{a \xrightarrow[ConversionWrongSize]{} b}$$

$$\frac{a : \texttt{NCLOB} \quad b : \texttt{LONG}}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.5.4 BFILE

$$\frac{a : \texttt{BFILE} \quad b : \texttt{BFILE}}{a \xrightarrow[Safe]{} b}$$

A.6 Rowid

A.6.1 ROWID

$$\frac{a : \texttt{ROWID} \quad b : \texttt{ROWID}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{ROWID} \quad b : \texttt{VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size \geq len(\texttt{ROWID})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{ROWID} \quad b : \texttt{VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size < len(\texttt{ROWID})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.6.2 UROWID

$$\frac{a : \texttt{UROWID} \quad b : \texttt{UROWID}}{a \xrightarrow[Safe]{} b}$$

$$\frac{a : \texttt{UROWID} \quad b : \texttt{VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size \geq len(\texttt{UROWID})}{a \xrightarrow[ConversionSafe]{} b}$$

$$\frac{a : \texttt{UROWID} \quad b : \texttt{VARCHAR2|NCHAR|NVARCHAR2}(size) \quad size < len(\texttt{UROWID})}{a \xrightarrow[ConversionWrongSize]{} b}$$

A.7 PL/SQL

Datatype `BINARY_INTEGER` behaves exactly as `NUMBER(10,0)`.

Datatype `BOOLEAN` flows to all other datatypes as *ConversionSafe*.

Appendix B

Contents of enclosed CD

```
src ..... the directory of source code
└── thesis ..... the directory of LaTeX source codes of the thesis
└── DP_Pelech_Ondrej_2015.pdf ..... the thesis text in PDF format
```