The Implementation of a Deductive Query Language Over an OODB

Andrew Dinn\textsuperscript{1}, Norman W. Paton\textsuperscript{2}, M. Howard Williams\textsuperscript{1}, Alvaro A. A. Fernandes\textsuperscript{1}, Maria L. Barja\textsuperscript{1}

\textsuperscript{1}Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Scotland \{andrew,howard,alvaro,marisa\}@cee.hw.ac.uk
\textsuperscript{2}Department of Computer Science, Manchester University, Manchester, England norm@cs.man.ac.uk

Abstract. The ROCK & ROLL database system cleanly integrates deductive and object-oriented capabilities by defining an imperative programming language, ROCK, and a declarative, deductive language, ROLL, over a common object-oriented (OO) data model. Existing techniques for evaluation and optimization of deductive languages fail to address key requirements imposed by ROLL such as: strict typing; placement of deductive methods (predicates) within classes; encapsulation; overriding and late binding. This paper describes the task of implementing an evaluator and optimizer for ROLL, explaining how existing implementation techniques for deductive languages were adapted to meet these requirements and extended to support novel types of optimization.

1 Introduction

This paper describes the task of implementing of an evaluator and optimizer for a deductive query language (DQL) operating over an object-oriented database. ROCK \& ROLL is a deductive object-oriented database (DOOD) system which cleanly integrates a Horn clause logic language ROLL with an imperative database programming language ROCK, thereby providing complementary programming paradigms for database application development [2]. Both languages operate over a common object-oriented data model OM and hence are cleanly integrated with no impedance mismatch.

Unlike most logic languages proposed as part of DOOD systems ROLL is fully object-oriented catering for encapsulation, multiple inheritance, overriding and late binding. Deductive methods are associated with a defining class, either in its public or private interface, and may be overridden by redefinition in subclasses. Late binding ensures the solution space for a query is partitioned, with solutions referring to subclass or superclass instances derived using the respective subclass or superclass methods. Few previously proposed DOOD systems have combined deductive programming with the benefits of the object-oriented paradigm; even fewer have also been implemented.

The presence of these OO features poses new challenges to existing techniques for deductive language evaluation and optimization. Furthermore, it provides
new scope for optimization, for example using type (and subtype) information to restrict the solution space for queries. Currently favoured implementation methods must be reconsidered, questioning whether they are still feasible and, if so, whether they are the most suitable means of implementing such a language.

ROLL was deliberately specified as a pure logic language with stratified negation, no function symbols, and no extra-logical features such as updates or explicit control mechanisms. This ensures that ROLL has a well-defined and well-known semantics and concedes little in expressive power since the omitted features are compensated for by the availability of an OO data model and the imperative language ROCK. However, another important motivation was that a simple logical language would not rule out the use or adaptation of many existing evaluation and optimization methods. The implementation presented here is based on an existing DQL evaluation and optimization method for relational databases, adapted to cope with the OO nature of ROLL and extended to enable novel types of optimization.

Section 2 provides a brief description of the OO data model, OM, and the DQL, ROLL. Section 3 identifies salient differences between ROLL and a conventional relational DQL. Section 4 justifies the choice of evaluation and optimization strategy and describes the selected techniques. Alternative candidates are assessed in section 5 which presents related work. Section 6 draws some conclusions.

2 OM and ROLL Overview

2.1 The Data Model OM

Type Definitions The OM data model consists of primary and secondary objects, which model atomic values and compound data, respectively. Each object is assigned an object type. Built-in object types, integers, reals and strings, are all primary and have predefined behaviour. User-defined primary types are always aliases for built-in types sharing their behavior. Secondary type definitions define the structure of their instances and declare their imperative and deductive method interfaces.

```plaintext
type city:
    properties
    public:
        name : string,
        population : integer;
    ...
end-type
```

Attributes A type definition specifies the structure of its instances by defining their properties and, optionally, their construction. The definition names a possibly empty set of referenced types which are its attributes, either public or private. Attributes defined in a type are inherited by its subtype. The above definition establishes a type city whose instances have two attributes, name and population, which are aliases for the primary types string and integer (the ROCK: and ROLL: interface specifications will be explained later).