Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 391-403

USING COMPILER TECHNIQUES TO CONVERT AN SQL APPLICATION FOR A NEW DBMS

A.-P. Tuovinen (Helsinki, Finland)

Abstract. We present a conversion system, SQL Converter, that translates VMS/Rdb SQL embedded in the C language to macros and function calls of a portable SQL application programming interface. The system demonstrates straightforward application of compiler construction techniques to high-level language translation. We describe a modular architecture that can be re-used for other similar conversion problems, too.

1. Introduction

In this paper we present a conversion system, *SQL Converter*, that converts SQL embedded in the C language to parametrized macros and function calls of a portable SQL application programming interface. The system demonstrates straightforward application of compiler construction techniques to high-level language translation. The system was produced for KT-DataCentrum Inc. in a research project partly funded by the company. The system will be used to convert a large SQL application to database independent form for porting to a new database system. The system was implemented as a part of the author's M.Sc. thesis [20].

Background. Open, distributed programming environments introduce new requirements for database management systems (DBMS) and database applications. The keywords are interoperability, portability and client-server architecture. Industry standard communication protocols for DBMS clients and servers are being developed by major companies. Also, there are many application programming interfaces (API) for developing portable applications.

The relational data model and the SQL data manipulation language are the *de facto* industry standards. Nowadays there are many CASE tools that provide DBMS transparency and portability by supporting different relational DBMS

and some standard version of SQL [5]. Also, many DBMS vendors provide gateways to systems of other gender [5]. The emergence of new SQL APIs for open environments (e.g. Microsoft's ODBC) will make DBMS interoperability become reality [12]. Developers of new database applications can take full advantage of these tools and create flexible and robust applications. On the other hand, there are great many *legacy systems* which are not interoperable nor portable, but which must somehow be adapted to new open operating environments. However, re-engineering legacy systems is a costly and errorprone business due to the scale of the changes.

Embedded SQL means embedding SQL statements in programs written with general purpose programming languages. Embedded SQL is an important traditional programmatic interface to DBMS; the method is straightforward and the data exchange scheme between SQL statements and the surrounding program is simple. Embedded SQL combines the data manipulation power of SQL and the expressive power of a general purpose programming language and facilities creating complex applications with clear separation of data management functions and application logics. Thus, embedded SQL is widely used. However, because different DBMS vendors provide very different versions of both SQL and embedded SQL, applications written using embedded SQL are not very portable.

Changing from embedded SQL to a new generation SQL API provides a way to increase the interoperability and portability of legacy systems. Manual re-engineering becomes very expensive, because the process requires analyzing and rewriting of every embedded SQL statement. On the other hand, the conversion problem can be viewed as *translation between high-level languages*, and compiler techniques can be used to automate the conversion.

Outline of this paper. In the next section we describe our particular conversion problem in more detail. In Section 3 we present our conversion system. Finally, in the last section we compare our system with other work in the field of translating relational query languages.

2. The conversion problem

KT-DataCentrum Inc. is the second largest software company in Finland. The PRIMA system is a personnel management and payroll application of considerable size and is one of the main software products of the company. The system was originally implemented in VAX/VMS environment with the VMS/Ddb DBMS. Two years ago KT-DataCentrum was faced with the need to port the PRIMA system to the Oracle DBMS. The company launched a joint

investigation with the Dept. of Computer Science at University of Helsinki to determine the requirements of porting the PRIMA system. The author was assigned to the project by the department.

The PRIMA system is a client-server application where workstations send service requests over a network to the database server of the system. The database services are provided by independent programs implemented in C. The underlying DBMS is VMS/Rdb and the programs use the embedded SQL of VMS/Rdb [4] to manipulate the database. The amount of C code containing embedded SQL totals up to a million lines. To port PRIMA to a new DBMS, the server programs and the database would have to be converted. Obviously, converting the programs would be the most costly and risky task. In this paper we concentrate only on the problem of converting the database programs. Converting PRIMA to a new DBMS had to be made an automatic process to be economically feasible. Also, the converted code would have to be maintainable because the converted system would be the new baseline product.

The basic solution to the conversion problem was to translate the embedded SQL to a portable SQL API called Pst/DB (portability software tools/databases) [17]. Pst/DB is a domestic SQL API similar to the SQL Call Level Interface (SQL/CLI) by SQL Access Group [1, 13]. The interface hides the differences in connection management, transaction management and dynamic execution of SQL statements between DBMS of different make.

Pst/DB guarantees DBMS independence of SQL only if standard SQL is used. In practice this means using the SQL dialect conforming to the 1989 ANSI/ISO SQL standard [8]. However, the VMS/Rdb SQL contains many incompatible features annd extensions, that must be dealt with in the conversion process to ensure DBMS independence of SQL.

2.1. Translation requirements

Converting embedded SQL to function calls of an SQL API is clearly a translation problem. However, the principal differences in the two SQL programming interfaces (embedded SQL and Pst/DB) and the differences in SQL dialects (VMS/Rdb-SQL and ANSI SQL) cause many problems for a translator. In the following we briefly introduce the main differences and the requirements they imply.

2.1.1. SQL programming interfaces

Embedded SQL

A programmer inserts an SQL statement as it is into her program when using static embedded SQL. The variables of the surrounding program are used directly in the embedded statement to pass data between the program

and the statement. Figure 1 shows a code fragment from a PRIMA program. The embedded SQL statement (called the singleton select statement) returns the number of rows (COUNT(*)) in the database table given in the **FROM** clause, that satisfy the condition given in the **WHERE** clause. The return value is assigned to the variable **iCount**. Here **iCount** is an *out-parameter* and **szMpvm** used in the boolean expression of the **WHERE** clause is an *in-parameter*.

Queries that may return more than one row of data require using explicit *cursors* for iterating over the result data. We will not consider cursors in this paper due to space limitations.

Programs containing embedded SQL are preprocessed with a *precompiler*, which analyzes the SQL statements, stores them for later execution and replaces them with calls to the executable units containing the stored statements. The precompiler checks the syntax of the statements and ensures that program variables (called host language parameters in VMS/Rdb SQL) are used correctly (e.g. the type of a variable is compatible with the database type of a table column).

```
static void vDbChkMpvmLask(CVP_BUF_HANDLE_TYPE pm_hBuffer, int
*pm_error){  char szTmp[11];
           szMpvm[17];
       int  iCount;
       eCVP_GetString(pm_hBuffer,szTmp,CVP_CLASS_DISPLAY_FIELD,
           PNMAT2_KT_PT_MAKSUPVM);
       lKonvertoiPvm(szMpvm,szTmp,SQLDATE3,DEFAULT_FORM);
           .....
       EXEC SQL
           SELECT COUNT(*)
           INTO: iCount
           FROM: PR_LASKEKAY
           WHERE LN_MAKSUPVM=:szMpvm AND
               LN_LNROTILA='1';

}
```

Fig.1. An embedded SQL statement

Pst/DB

When using the Pst/DB interface the SQL statements are formed incrementally and dynamically at execution time. An SQL statement is passed as a string to the database server that interprets the statement and returns

then the result data, if there are any. Program variables can be used to pass data between the program and the database, but in a rather low-level manner. For instance, the programmer must explicitly declare the host language types of the program variables and declare the length of string variables in bytes. Also, the programmer has to define a number of control objects in the program which are used to manage the connection to the database server and to identify the resources reserved by the server and Pst/DB for constructing and executing SQL statements. Queries are just like other SQL data manipulation statements expect that there is result data to be handled with the iteration services Pst/DB provides.

```
static void vDbChkMpvmLask(CVP_BUF_HANDLE_TYPE pm_hBuffer, int
    *pm_error) {  char szTmp[11];
          szMpvm[17];
      int iCount;
      eCVP_GetString(pm_hBuffer,szTmp,CVP_CLASS_DISPLAY_FIELD,
          PNMAT2_KT_PT_MAKSUPVM);
      lKonvertoiPvm(szMpvm,szTmp,SQLDATE3,DEFAULT_FORM)
        ....
      SELECT_INTO("SELECT \
              COUNT(*) \
          FROM \
              PR_LASKEKAY \
          WHERE \
              LN_MAKSUPVM=:szMpvm AND \
              LN_LNROTILA='1'",
          INTO(INTOINT(0,iCount,NOINDICATOR)),
          PARSYM(DATPAR(szMpvm)));

}
```

Fig.2. The converted SQL statement

Figure 2 shows the example code from Figure 1 converted to a parameterized C macro that is syntactic sugar on top of the low-level Pst/Db. The `SELECT_INTO` macro hides the control objects of Pst/Db which are needed to create and execute an SQL statement in the database server. We see that the embedded statement is converted to a C string which is the first parameter of the macro. The `INTO` clause has been left out from the statement. As the second parameter of the `SELECT_INTO` macro is an `INTO` macro where the out-parameter *iCount* is *bound*, i.e. the variable used to store the return value is

identified and its type, **INT**, is declared. In this case the type is deduced from the C type of **iCount**. As the last parameter of the **SELECT_INTO** macro is **PARSYM** macro that is used to bind the in-parameter **szMpvm**. The value of **szMpvm** is logically a date value, because it is compared to the column **LN_MAKSUPVM** that has the VMS/Ddb type **DATE**. However, because date types require special processing (see Section 2.1.2), the type of **szMpvm** must be declared to be **DAT** instead of a string type, which is the C type of **szMpvm**. Thus a translator must analyze the variable declarations of the host program to produce correct bindings for the in- and out-parameters.

2.1.2. SQL dialects

The 1989 ANSI/ISO SQL standard describes a very limited language compared to the SQL dialects provided by different DBMS vendors. On the other hand all the major SQL dialects contain as a subset the data manipulation language defined in the standard. Also, the standard leaves open many details (e.g. database connection management) which DBMS vendors are free to define as they wish.

For instance VMS/Rdb SQL introduces several new data types and operations on them that are not part of the standard. Also, subqueries may be used more freely as expressions than in standard SQL. VMS/Rdb SQL includes some new predicates, too.

The date-time data types of VMS/Rdb and Oracle proved to be incompatible. Thus the correct handling of date values requires progammatic support that would have to be added on top of Pst/Db. But this means that the translator must notice when date-time data is passed between the program variables and SQL in order to create correct bindings.

A translator cannot transform all the non-standard features of VMS/Ddb SQL to semantically equivalent expressions in standard SQL. For instance in VMS/Ddb SQL it is possible to compare values of different data types (implicit type conversion) in boolean expressions which is not allowed in the standard. Also, the liberal use of subqueries in VMS/Rdb SQL causes difficult if not impossible transformation problems. However, warnings should be issued when non-standard features are encountered during translation.

Many of the shortcomings of the 1989 SQL standard (SQL89) have been corrected in the new 1992 SQL standard, SQL2 [9]. However, SQL89 is still the most portable dialect of the language, because there are only few SQL implementations that claim to be full SQL2 compliant. Thus SQL89 was set as the target for the translations.

3. SQL converter

In this section we describe the SQL converter system. First we explain in general how SQL converter solves the conversion problem defined in Section 2. Then we give a high-level view of the architecture of the system and describe the conversion process.

3.1. Overview

The source languages of the translation process are C and embedded VMS/Rdb SQL. Target languages are C and a subset[1] of VMS/Rdb SQL. To be more precise, embedded SQL statements in the source C modules are translated to parametrized C macros and function calls, where SQL is contained in static C strings. The macro layer on top of Pst/Db is called PPA (PRIMA portability adapter).

Programming interfaces

The translation system maintains a complete symbol table of the type and variable declarations in a source model. Therefore, the C program containing embedded SQL is parsed and the declarations are analyzed.

The system analyzes the static type of expressions in SQL statements to be able to create correct bindings for date-time objects, too. The system uses an in-memory description of the target database to resolve the types of table columns and the expressions where they are used.

There are also a number of other details to take care of. For instance the translator must expand C structures that can be used as parameters in the embedded SQL statements to listings of individual fields in the converted SQL statements.

SQL dialects

We decided not to try transforming non-standard VMS/Rdb SQL features to standard SQL. First, because this is in some cases impossible, and second, because the SQL used in PRIMA conforms well to the standard. Only the problematic date-time data types had to be dealt with, which was done at the level of the PPA macros. However, the system analyzes the SQL statements and issues warnings of non-standard features encountered in the statements.

[1] without the extensions of embedded SQL

Other features

The system produces maintainable code. The original C code surrounding embedded SQL statements is untouched by the conversion process and the generated PPA code is pretty printed. SQL converter leaves optionally the original embedded SQL statements as C comments in the converted programs.

3.2. Architecture

The system was designed and implemented using the well-known techniques of compiler construction. Thus the architecture of the system is modular and the interconnections between the modules are well defined. The separation of concerns makes it easy to modify the system. For instance changing the code generation module has no effect on other parts of the system. The architecture of our system ressembles closely the architecture of a generic query language translator described in [10].

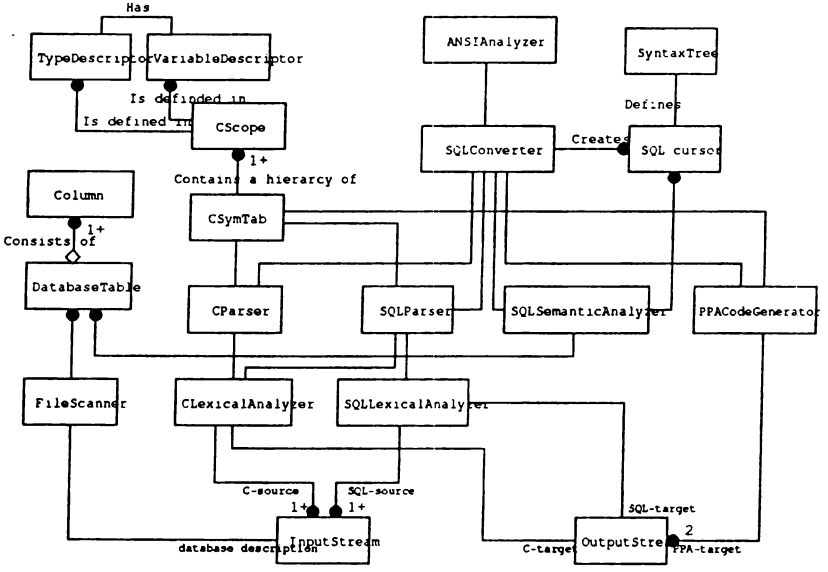

Fig.3. The object model

The system components are modelled by abstract data types or classes, which provide a function interface to other components hiding all the implementation details. A syntax tree structure is used as an intermediate representation of an SQL statement during the different phases of conversion.

3.2.1. Main components

Figure 3 presents the object model of SQL converter. The modelling technique used is OMT [16]. In the following we describe the responsibilities of the main components.

- *CParser* collects symbol table information from type and variable declarations. Inserts type and variable descriptors to the C symbol table. The parser invokes the *SQLConverter* object when the parser encounters an embedded SQL statement.
- *SQLConverter* controls the conversion of an SQL statement by invoking first the *SQLParser* to parse the statement and to produce a corresponding syntax tree. Then the SQLConverter invokes the *SQLSemanticAnalyzer* to annotate the syntax tree with SQL type information concerning table columns, expressions and the corresponding program variables. Next, the *ANSIAnalyzer* checks the ANSI conformance of the SQL statement. Finally, SQLConverter invokes the *PPACodeGenerator* to generate PPA macros from the syntax tree. Other functions of SQLConverter include managing global data concerning cursors.
- *SQLParser* parses the SQL statement and produces syntax tree of the statement. The parser attaches C symbol table descriptors to the in- and out-parameters.
- *SQLSemanticAnalyzer* annotates the nodes that correspond to column names with their database types and deduces the types of expressions based on the types of the columns used in the expressions. The analyzer attaches SQL type information to in- and out-parameters using the database symbol table produced by the *FileScanner*.
- *ANSIAnalyzer* checks the ANSI conformance of the SQL statement represented as a syntax tree. The analyzer prints warnings of non-standard features to the conversion log file.
- *PPACodeGenerator* generates PPA macros from the annotated syntax tree. The generator uses the type information attached to in- and out-parameters and C symbol table information to produce correct bindings for the parameters.
- *FileScanner* reads a database description file that contains the names of the tables in the PRIMA database and names and types of columns of the tables. The scanner creates a symbol table where the information is stored for looking up.

Figure 4 shows the functional view of the conversion process using the functional modelling notation of OMT. The main processes, data repositories

and dataflows between processes are depicted. Thhe dataflows exiting to the right are system output and the flows entering from the left are system output.

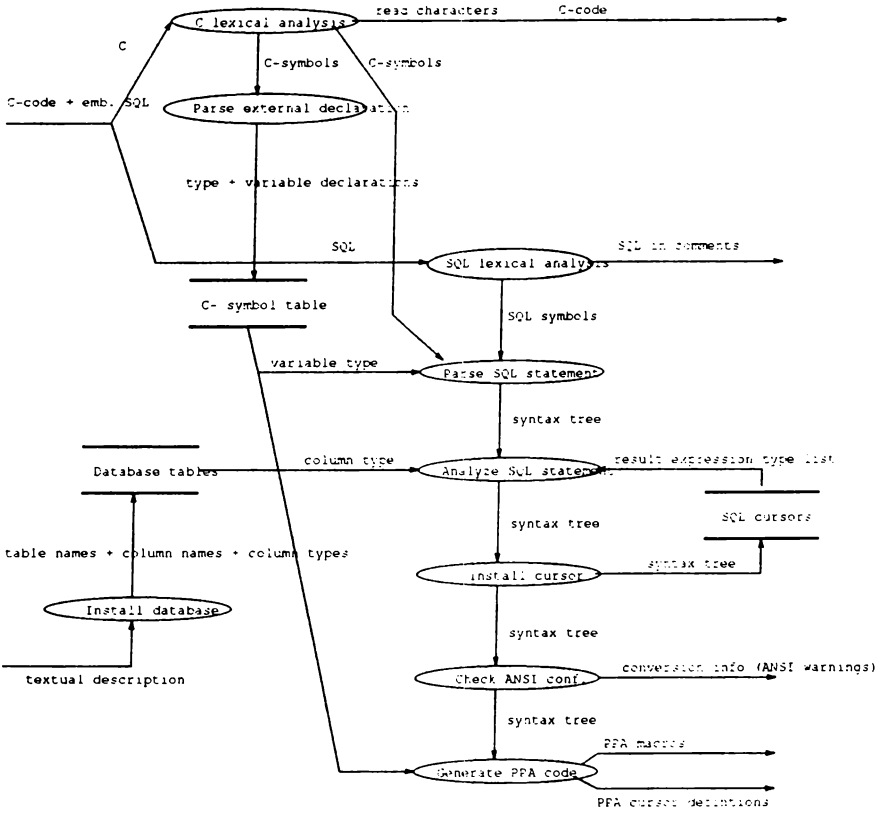

Fig.4. The conversion process

3.3. Implementation

The SQL and C parsers are recursive decent. The C syntax is $LL(2)$ and SQL as much as $LL(11)$. The lexical analyzers provide arbitrary long lookahead in the input symbol stream. Lookahead is used in some cases to resolve parsing conflicts. Despite of the extensive use of lookahead the size of the SQL parser is about 8000 lines compared to the C parser which is only 4000 lines long (including comments). The system is handcoded in ANSI C and totals up to 38000 lines. The design and implementation of SQL converter took about 10 man-months.

4. Comparison with other work

Other approaches

Automating database application conversion received some attention in the late 70's and early 80's [19, 18]. After that there is no much literature on the subject. However, translations between relational query languages have been studied in other contexts.

Translation schemes

Query language translation techniques have been studied in connection with heterogeneous distributed database systems, where queries are passed between different systems. There are at least two different approaches to query language translation.

The *metacompiler approach* uses compiler writing systems, that produce language processors from the specification of language [7, 14, 15]. According to the authors the benefits of this approach are: (1) the language specifications are short and abstract which makes the modifications easy, (2) compared with handcoding a compiler there is much less writing, and (3) the metacompilers are perfect for incremental translator development. However, the translators produced by metacompilers tend to be rather inefficient.

In the *algorithmic approach* a translation algorithm is given for a particular translation problem, e.g. QUEL to SQL [11]. Algorithms are also given for translating SQL to relational algebra or calculus [2, 3, 6]. These algorithms are used, for instance, to give an exact definition for the semantics of SQL.

One common feature in the different translation schemes is that they use some kind of intermediate representation in the translation process. This reduces the number of translators needed in a multilingual environment, e.g. in a distributed heterogeneous DBMS, because only two translators are needed for every language: source language to intermediate language and intermediate language to source language.

Conclusions

Our work demonstrates innovative use of wellknown translation techniques to a concrete re-engineering problem. Tests done by engineers at KT-DataCentrum show that SQL converter performs nearly 100 percent of the work needed to convert the PRIMA database programs.

The use of compiler construction tools would have made the implementation of the system easier. However, KT-DataCentrum did not want this, because they were not familiar with that kind tools. Compared to the

metacompiler approach, our system is less flexible, but modular enough to be re-used for other similar conversion problems. SQL converter is an efficient and reliable tool that has been produced using disciplined engineering practices.

References

[1] *ANSI: SQL Access Group Call Level Interface, Base Document vol. 2.12. ANSI X3H2-92-143*, 1992.

[2] **von Bültzingsloewen G.,** Translating and optimizing SQL queries having aggregates, *Proc.of the 13th Conf. on Very Large Databeses, Brighton, 1987,* eds. P.Stocker, W.Kent and P.Hammersley, Morgan Kaufmann Publishers Inc., 1987, 235-243.

[3] **Ceri S. and Gottlob G.,** Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries, *IEEE Transactions on Software Engineering SE-11,* (4) (1985), 324-345.

[4] *Digital: VMS/Rdb Language Reference Manual for v. 4.1.,* Digital Equipment Corporation, 1993.

[5] **Dowgiallo E.,** Database interoperability and application transportability, *Dr. Dobb's Journal,* **18** (13) (1993), 38-42.

[6] **Gogolla M.,** A note on the translation of SQL to tuple calculus, *ACM SIGMOD RECORD,* **19** (1) (1990), 18-22.

[7] **Howells D., Fiddian N. and Gray W.,** A source to source meta translation system for database query languages - Implementation in Prolog, *Prolog and databases - implementations and new directions,* Ellis Horwood Series in Artificial Intelligence, 1988, 22-38.

[8] *ISO: Information Processing Systems - Database Language SQL,* International Standard ISO: 9075:1987, 1987.

[9] *ISO/IEC: Information Processing Systems - Database Language SQL,* International Standard ISO/IEC 9075:1992, 1992.

[10] **Kimbleton S., Wong P. and Lampson B.,** Chapter 14. Applications and protocols, *Distributed systems - Architecture and implementation,* eds. B.Lampson, M.Paul and H.Siegert, Springer, 1981, 308-370.

[11] **Viet C.,** Translation and compatibility of SQL and QUEL queries, *J.of Information Processing,* **8** (1) (1985), 1-15.

[12] **McCusker T.,** Build links to multiple databases, *DATAMATION,* **21** (1994), 65-67.

[13] **Perkovic P.,** Access and ANSI/ISO SQL and X/Open, *Proc. COMPCON'91 36th IEEE Computer Society Int. Conf., San Francisco, 1991,* eds. L.O'Connor and A.Copeland, IEEE Computer Society Press, 1991, 120-122.

[14] **Piatetsky-Shapiro G. and Jakobson G.,** An intermediate database language and its rule based transformation to different database languages, *Data & Knowledge Engineering,* **2** (1) (1987), 1-29.

[15] **Rusinkiewicz M. and Czejdo B.,** Query transformation in a multi database environment using a universal symbolic manipulation system, *Computing Trends in the 1990's. 17th Annual ACM Comp. Science Conference, Louisville, Kentucky, 1989,* Association of Computing Machinery Inc., 1989, 46-53.

[16] **Rumbaugh J. et al.,** *Object-orientated modelling and design,* Prentice Hall, 1991.

[17] *SEpiiri/ATO: Portable database interface (specification),* specification document, 1993.

[18] **Schneiderman B. and Thomas G.,** An architecture for automatic relational database system conversion, *ACM Transactions on Database Systems,* **7** (2) (1982), 235-257.

[19] **Taylor R. et al.,** Database program conversion: A framework for research, *Proc. 5th Int. Conference on Very Large Databases, Rio de Janeiro, Brazil, 1979,* IEEE Computer Society, 1979, 299-312.

[20] **Tuovinen A.-P.,** *Relaatiotietokantojen kyselykielten väliset muunnokset,* MSc Thesis, Report C-1994-70, Dept. of Computer Science, University of Helsinki, 1994.

A.-P. Tuovinen
Department of Computer Science
University of Helsinki
P.O.B. 26 (Teollisuuskatu 23)
00014 Helsinki, Finland
aptuovin@cs.helsinki.fi