Appeared in: C. Beeri, A.Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, August 1993, Manhatten, New York*, Workshops in Computing. Springer Verlag, Feburary 1994.

Extensible Grammars for Language Specialization

Luca Cardelli   Florian Matthes [*]   Martín Abadi

Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA

Abstract

A frequent dilemma in the design of a database programming language is the choice between a language with a rich set of tailored notations for schema definitions, query expressions, etc., and a small, simple core language. We address this dilemma by proposing extensible grammars, a syntax-definition formalism for incremental language extensions and restrictions based on an initial core language.

The translation of programs written in rich object languages into a small core language is defined via syntax-directed patterns. In contrast to macro-expansion and program-rewriting tools, our extensible grammars respect scoping rules. Therefore, we can introduce binding constructs while avoiding problems with unwanted name clashes.

We develop extensible grammars and illustrate their use by extending the lambda calculus with let-bindings, conditionals, and SQL-style query expressions. We then give a formal description of the underlying parsing, transformation, and substitution rules. Finally, we sketch how these rules are exploited in the implementation of a generic, extensible parser package.

1 Introduction

A frequent dilemma in the design of a database programming language is the choice between a user-friendly language with a rich set of tailored notations for schema definitions, query expressions, etc., and a small, conceptually simple core language. We address this dilemma by proposing extensible grammars, a

[*]The second author was supported by the European Commission, ESPRIT, EC/US-FIDE Collaborative Activity, 006:9829.

syntax-definition formalism for incremental, problem-specific language extensions and restrictions based on an initial core language.

The translation of programs written in rich, user-friendly object languages into a small core language is defined via syntax-directed patterns. In contrast to traditional macro-expansion and program-rewriting tools, our extensible grammars respect scoping rules. Therefore, we can introduce new binding constructs like quantifiers, iterators, and type declarations, while avoiding problems with unwanted name clashes ("variable captures").

Syntax extensions provide syntactic sugar for common problem-specific abstractions. For example, embedded query notations like the relational calculus, the relational algebra, iteration statements, or set comprehensions can be introduced as abstractions defined from more primitive iteration constructs [OBBT89, BTBN91, Tri91, MS91]. Transactions can be introduced as stylized patterns for side-effect control and exception handling. Similarly, structured form definitions in user interface code can be represented as abstractions over low-level routines for data formatting, input, and validation. At the type level, data modeling constructs like classes, objects, and binary relationships can be viewed as syntactic sugar for more complex type expressions involving recursive types, record types, function types, or abstract data types [SSS$^+$92, SSS88, PT93].

Syntax restrictions introduce intentional limitations on the expressiveness or orthogonality of a core language. The rationale behind restrictions is to facilitate meta-level reasoning and optimizations tailored to a particular application domain. While ad-hoc syntax restrictions are generally considered harmful in programming language design (from a pragmatic and a semantic perspective), they are common practice in database models and database languages. For example, many schema definition languages disallow nested declarations (nested sets, nested classes) or limit recursive declarations to top-level class or type definitions. Furthermore, user-defined types frequently do not have first-class status, e.g., they may not appear as arguments to collection-type constructors. Similarly, query languages typically impose restrictions to rule out side-effecting operations or calls to user-defined functions in selection and join predicates [SQL87]. Some query languages require static bindings to function identifiers (disallowing higher-order functions or dynamic method dispatch) [SFL83], and some disallow lambda abstractions within quantified expressions [BTBN91]. Finally, recursive queries or views are often subject to stratification constraints [Naq89].

The form of extensible grammars discussed in this paper was invented during the implementation of a polymorphically typed lambda calculus [Car93]. Here, we develop extensible grammars in a more general context and describe them in more detail. Section 2 gives a conceptual overview of the issues that must be addressed by a syntax-extension formalism. In section 3 we introduce extensible grammars by examples. An initial grammar for the lambda calculus is extended incrementally with new syntactic forms like let-bindings, conditionals, as well as algebraic and calculus-style query notations. In section 5 the static type rules for grammar definitions and the semantics of parsers generated from extensible grammars are defined. We also present a soundness result for the type system with respect to the evaluation semantics. The impact of these foundations on the implementation of an extensible parser module for the Tycoon database environment [Mat93] is highlighted in section 5. Finally, section 6 compares

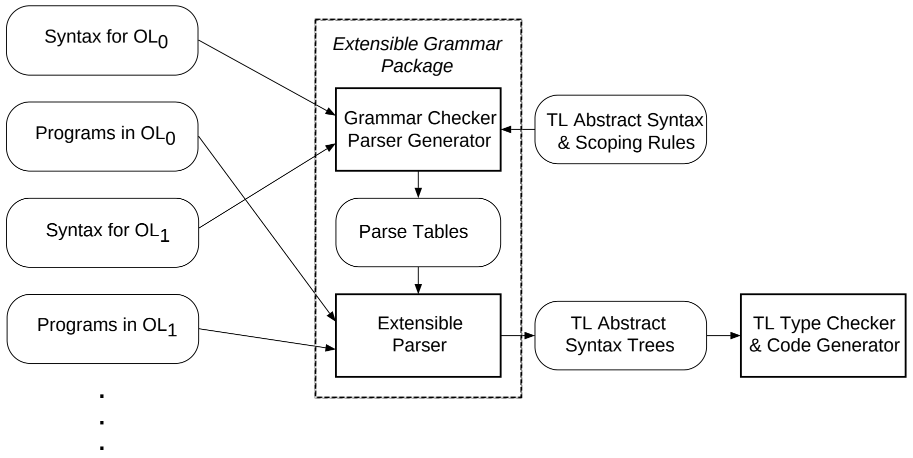
Syntax for OL_0
Programs in OL_0
Syntax for OL_1
Programs in OL_1
Extensible Grammar Package
Grammar Checker Parser Generator
Parse Tables
Extensible Parser
TL Abstract Syntax & Scoping Rules
TL Abstract Syntax Trees
TL Type Checker & Code Generator

Figure 1: The syntax-extension scenario

our concept of extensible grammars with other approaches to syntax extension.

2 Overview

The syntax extension formalism described in this paper assumes the scenario depicted in figure 1. Given the abstract syntax and the scoping structure of a target language TL, a new object language OL_0 can be defined by giving its context-free grammar and the rewrite rules that map OL_0 terms into TL terms. The mapping also defines the scoping structure of OL_0. Our formalism is incremental since it allows also the definition of an object language OL_n by a translation (rewriting) into another object language OL_{n-1}.

For example, assuming TL to be a functional language, the object language OL_0 could have either a Lisp-like list notation or an Algol-like keyword-based notation:

```
(defn succ(x) (plus x 1))
function succ (x); begin return plus(x, 1); end succ;
```

Both syntactic forms translate into the same abstract syntax tree in the target language TL that is passed to the TL type checker and code generator:

```
Abs(x App(App(plus x) 1))
```

Section 3.1 gives a complete example of the target-language and the object-language definition for an untyped lambda calculus.

A simple example of an incremental syntax definition is the definition of a language with infix function application (OL_1) as an extension of a language with only prefix application (OL_0). The notation `A` $\Rightarrow$ `B` is used to indicate that the input `A` in an extended language is equivalent to the input `B` in a non-extended language:

```
function succ (x);                 function succ(x);
begin return x + 1 end succ;   ⇒   begin return plus(x,1) end succ;
```

In a database programming setting, OL_n could be a language with SQL-like query notations that is translated into a lambda calculus, OL_{n-1}, with primitive operations on a collection type (**nil, cons, iter**) [Tri91]:

```
select x.a        iter(X)(nil)(fun(x)fun(z)
from x in X   ⇒     if p(x) then cons(x.a)(z) else z)
where p(x)
```

Incremental grammar definitions are discussed in more detail in section 3.2 and 3.3. The definition of an SQL-like grammar in our formalism is given in section 3.4.

Extensible grammars require extensible parsers. That is, a parser cannot be generated once for a given target language, but has to be extended dynamically to handle programmer-defined object languages. New grammar definitions should be checked to avoid problems typical of macro definitions [KR77], such as grammar ambiguity, non-termination of macro expansion, and generation of illegal syntax trees. Our checking is performed already at grammar-definition time and includes standard grammar analysis [ASU87] to avoid the first two problems. To address the third problem, we develop a sorting discipline on productions (see section 4.1).

A more subtle source of difficulties associated with incremental grammar definition is the binding structure of the target language. The rewriting of object-language expressions into target-language expressions must be sensitive to the scoping rules of the target language and may require renaming operations to avoid name clashes ("variable captures"). A small example using C and the C preprocessor illustrates the issue in a familiar setting:

```
#define swap(x,y)  {int z; z = x; y = x; x = z;}
{int a, b;  swap(a,b);}   /* ok */
{int z, y;  swap(z,y);}   /* name clash */
```

The expansion of `swap(z, y)` leads to the program fragment `{int z; z = z; y = z; z = z}`, where the local declaration of `z` hides the variable `z` that is passed as an argument to the macro. Removing the curly brackets in the macro definition does not solve the problem but yields a name clash between two declarations of the variable `z` in the same scope.

A solution of the scoping issues associated with rewriting inside binding structures requires a formalization of the scoping rules of a specific target language. To adapt our grammar formalism easily to several target languages, we divide the scoping problem into a generic bookkeeping task for the extensible parser and a parameterized language-specific renaming operation. This conceptual division of labor is exploited in the implementation of the extensible grammar package to factor out target-language dependencies. Scoping problems are avoided by distinguishing between binding and applied identifier occurrences, and by renaming when name clashes between identifiers in input programs and in rewrite rules could occur. Note that this solution is not an option for a simple token-based preprocessor. Section 4.2 describes the parsing and renaming rules of our formalism (for initial as well as incremental grammar definitions). We are also able to prove that these dynamic parse rules are consistent with the static type rules given in section 4.1.

3 Grammar Definitions

In this section we introduce our extensible grammar formalism by examples. We start with a small initial grammar for an untyped lambda calculus that is extended incrementally to support database programming language constructs.

```
grammar
  simpleTerm:Term ==
   x=ide                         => mkTermVar(x)
  |"(" a=term ")"                => a
  |"fun" "(" x=ide ")" a=term    => mkTermFun(x a)
  |"{" f=fields "}"              => mkTermRcd(f)
  |a=pIde:Term                   => a

  fields:Fields ==
   x=ide "=" a=term f=fields     => mkFieldCons(x a f)
  |                              => mkFieldNil()
  |f=pIde:Fields                 => f

  term:Term ==
   a=simpleTerm b=termIter(a)    => b

  termIter(a:Term):Term ==
   "(" b=term ")"                => termIter(mkTermApp(a b))
  |"." x=ide                     => termIter(mkTermDot(a x))
  |                              => a
end
```

Figure 2: Definition of a concrete syntax for the lambda calculus

3.1 Initial Grammar Definitions

This section explains how to define the abstract syntax and the scoping rules of a particular target language *TL* as well as the syntax for an initial object language OL_0 (see the oval boxes in figure 1). This information is validated by the grammar checker and then used to generate an initial parser for OL_0 programs.

We use an untyped lambda calculus with records as the target language for our examples. Given a set of identifiers x, the sets of terms (a, b) and fields (f) are recursively defined as follows:

$$\begin{array}{lll} a,\ b & ::= & x \mid \lambda x.a \mid a(b) \mid \{f\} \mid a.x \\ f & ::= & \emptyset \mid x{=}a\ f \end{array}$$

The first step in the definition of an extensible grammar is to define the names of the *sorts* and the signatures of the *constructors* available for the construction of target-language terms. Our example uses the following target-language-specific sorts:

`Term`	terms of the lambda calculus
`Fields`	ordered associations between field names and terms

Since identifiers require particular attention during expression rewriting, there are three predefined sorts to distinguish the binding properties of identifiers:

`Binder`	identifiers appearing in binding positions
`Var`	identifiers appearing as variables inside the scope of a binder
`Label`	identifiers that are not subject to scoping

These sort names appear in the signatures of the term constructors for the lambda calculus:

```
mkTermVar(x:Var):Term
mkTermFun(x:Binder a:Term):Term
mkTermApp(a:Term b:Term):Term
mkTermRcd(f:Fields):Term
mkTermDot(a:Term x:Label)
mkFieldNil():Fields
mkFieldCons(x:Label a:Term f:Fields):Fields
```

Lambda abstractions (`mkTermFun`) introduce identifiers in binding positions, while other identifiers inside terms (`mkTermVar`) appear in non-binding positions. In our example, field labels (`mkTermDot, mkFieldCons`) are not subject to block-structured scoping rules and are therefore defined to be of sort `Label`. For the purpose of grammar definitions it is not necessary to present the binding rules of the target language in more detail.

Given a target-language description in terms of constructors and sorts, a context-free grammar is defined as a collection of productions that translate phrases in an input stream into terms of the target language. A concrete syntax for the lambda calculus with records is defined in figure 2. The notation used is explained in the rest of this section.

This grammar consists of four mutually recursive productions that define precedence of applications over abstractions and left-associativity of applications. Here are examples of input phrases parsed according to the root production `term`:

```
peter         mkTermVar(peter)
peter.age     mkTermDot(mkTermVar(peter) age)
fun(p)p(b)    mkTermFun(p mkTermApp(mkTermVar(p) mkTermVar(b)))
```

The result of parsing is a structured term of the target language. This term can be viewed as a tree in which the inner nodes correspond to term constructor applications and the leaves correspond to identifiers (or literals) extracted from the source text. A token sequence to which no production applies is rejected by the parser with an error message.

A grammar introduces a set of non-terminals (`simpleTerm`, `term`, ...) as identifiers for productions. Productions can be parameterized by terms of the target language (see, e.g., `termIter`). The signature of a non-terminal defines its parameter names and sorts as well as the sort of terms returned by the production.

Each production consists of $n \geq 1$ expression sequences separated from each other by a vertical bar (|). Each expression specifies an input syntax and a result expression (following the `=>` symbol) to construct a term of the target language. Based on the token sequence encountered during parsing, one of the alternative expression sequences is selected and its corresponding result expression is evaluated in an environment that contains the actual parameter bindings and local bindings introduced on the left of the `=>` symbol.

The input syntax accepted by an alternative is defined using the following notation:

`"x"`	accept the keyword **x**
`ide`	accept any non-keyword identifier
`x`	accept the input specified by the production identified by the non-terminal `x`
`x(y)`	accept the input specified by the parameterized production identified by the non-terminal `x` with the argument `y`
`x=y`	bind the term defined by `y` to a local variable `x`
`pIde:S`	accept a pattern variable of sort `S` (see section 3.3)

Each grammar determines a set of keywords reachable from productions of the grammar. The set of identifiers accepted by **ide** in a given grammar **g** excludes the keywords of **g**. Therefore, syntax extensions may introduce new keywords while syntax restrictions may change existing keywords into identifiers.

The binding structure of the concrete syntax is defined implicitly by passing identifier tokens from the input as arguments to term constructors. For example, the variable `x` in the grammar definition

```
    "fun" "(" x=ide ")" a=term   => mkTermFun(x a)
```

appears in a `Binder` position of the term constructor `mkTermFun`. Therefore, it can be deduced that the variable `person` in the source text `fun(person)` ... appears in a binding position.

The recursive production `fields` in figure 2 generates right-associative syntax trees for field lists while the production `termIter` generates left-associative syntax trees for function applications. Because we use an LL(1) parser, left-associative grammars are handled in our grammar formalism by passing the syntax tree for the left context of a phrase as a production argument for the recursive invocation of a production (e.g., `a:Term` in production `termIter` in figure 2).

3.2 Incremental Grammar Definitions

This section explains how to define the syntax of a new object language OL_n as an extension or a restriction of an existing object language OL_{n-1}. Such a syntax redefinition is validated by the grammar checker and used to derive a parser for OL_n from an existing parser for OL_{n-1}.

A grammar defines a mapping from non-terminals (e.g., `simpleTerm`, `term`) to variables that are initialized with productions. Inside a production, each non-terminal denotes the production identified by its variable. Three incremental grammar operations are available: addition, extension, and update. The rationale behind these operations is to allow update and re-use of existing non-terminal definitions, preserving the recursive structure of the grammar.

A grammar addition (==) defines a mapping from a non-terminal to a newly created variable initialized with a production. For example, we could use the standard encoding of let bindings:

```
    let x=a in b                  ⇒ (fun(x) b)(a)
```

to add the new non-terminal `topLevel`:

```
grammar
  topLevel:Term ==
   a=term                       => a
  |"let" x=ide "=" a=term
   "in" b=topLevel              => mkTermApp(mkTermFun(x b) a)
end
```

The non-terminal `topLevel` is mapped to a newly created variable initialized with a production that accepts terms of the base language and (nested) let bindings at the top level, but not inside terms.

A grammar extension (|==) destructively updates the variable identified by a non-terminal with a new production. The new production extends the old production with additional alternatives. For example, to extend `simpleTerm`, we could write:

```
grammar
  simpleTerm:Term |==
   "unit"                       => mkTermRcd(mkFieldNil())
  |"let" x=ide "=" a=term
   "in" b=term                  => mkTermApp(mkTermFun(x b) a)
end
```

This grammar extension affects all productions referring to `term`, allowing **unit** and nested **let** bindings within terms.

A grammar update (:==) destructively updates the contents of a variable identified by a non-terminal with a new production that has the same signature, thereby affecting all productions referring to that non-terminal. For example, the definition of `term` could be updated as follows:

```
grammar
  term:Term :==
   x=ide                        => mkTermVar(x)
  |"(" a=term b=term ")"        => mkTermApp(a b)
  |"{" f=fields "}"             => mkTermRcd(f)
end
```

This redefinition affects all productions referring to `term` (`simpleTerm`, `fields`, `termIter`), thereby restricting the expressiveness of the original language by disallowing abstractions.

3.3 Pattern-based Action Definitions

In the previous section, abstract syntax trees produced by actions are specified with explicit constructor applications. In this section we introduce patterns which allow us to write grammars more conveniently by using the existing target language. For example, the syntax for **let** and **where** bindings could be written more clearly using a pattern:

```
  grammar
    simpleTerm:Term |==
     "let" x=ide "=" a=term
     "in" b=term                   => term<<(fun(x) b)(a)>>
  end
```

Inside the pattern `term<<(fun(x) b)(a)>>`, the variables `x`, `a`, and `b`, introduced on the left-hand side of the production, act as placeholders (pattern variables) of sort `Binder`, `Term`, and `Term`, respectively. A pattern `p<<s>>` in a grammar `g` is translated into constructor applications by parsing the input token stream `s` starting with the production `p`. For example, the pattern `term<<(fun(y) b)(a)>>` yields the nested constructor application `mkTermApp( mkTermFun(y b) a)` when the token stream `(fun(y) b)(a)` is parsed as a `term`.

The keyword **pIde** followed by a sort identifier is used in the initial grammar definition (see section 3.1) to indicates those positions in the input syntax where pattern identifiers may appear. Pattern variables of the sorts `Binder`, `Var`, and `Label` may appear also at those places in the input syntax where the keyword **ide** is used to accept identifier tokens of the appropriate sort.

Many pattern-based syntax extensions require the introduction of fresh identifiers, i.e., identifiers distinct from other identifiers appearing in `Binding` and `Var` positions, to avoid variable captures and name clashes. For example, the syntax for functional composition (`f * g`) could be defined as:

```
  grammar
    termIter(a:Term):Term |==
     "*" b=term x=local      => termIter(term<<fun(x)a(b(x))>>)
  end
```

The notation `x=local` guarantees that a fresh identifier is bound to `x` for every instantiation of this production during parsing. For example, `f*g*h` is expanded to `fun(x2)(f(fun(x1)g(h(x1))) (x2))`, and `x*y` is expanded to `fun(x1)(x(y(x1)))`, avoiding a variable capture of the input variable `x` by a binder introduced in the pattern.

Since grammar definitions can be interspersed with object-language expressions, it is desirable to allow patterns to contain variables that refer to global bindings. For example, the boolean constants true and false are sometimes represented by the following functions which, when applied to two arguments, return one of them:

```
let T = fun(x)fun(y)x
let F = fun(x)fun(y)y
```

In the scope of these definitions, the following grammar could be defined to replace the keywords **true** and **false** by the variables `T` and `F`, respectively.

```
  grammar
    simpleTerm:Term |==
    "true"                          => term<<T>>
   |"false"                         => term<<F>>
   |"if" a=term "then" b=term
    "else" c=term                   => term<<a(b)(c)>>
  end
```

During expansion of a pattern with free variables (T and F in the example above), unwanted variable captures must be avoided. For example, a naive macro expansion of the term **fun**(T) T(**true**) would yield the term **fun**(T) T(T) where the expansion of the keyword **true** is bound incorrectly. Therefore, free variables in extensible grammars are handled as follows: Each occurrence of a free variable **x** in a grammar definition is replaced by a fresh variable x'. During parsing, these modified patterns generate expansions that contain unbound variables (T' and F'). For example, T(**fun**(T) T(**true**)) is expanded to T(**fun**(T) T(T')). After the full input has been parsed, a target-language-specific renaming function is applied to the parsed term. It replaces the binder T and its bound variables by T'' and T' by T. The resulting term T(**fun**(T'') T''(T)) is then submitted to the type checker and code generator.

3.4 Further Examples: Query Notations

In this section we show how some typical database query notations can be viewed as mere "syntactic sugar" for the application of a single higher-order iterator function. The reduction of query notations into a single canonical iteration construct has been exploited in the literature to simplify the type checking of database programming languages [OBBT89], the code generation for query expressions [Tri91], and the verification of functional database programs [SS91, SSS88]. The following examples demonstrate that extensible grammars provide sufficient expressive power to define the syntax of typical database query languages as well as their translation into lambda calculus. This translation preserves the usual scoping rules defined for these query languages.

We assume the grammar extension for booleans defined above and the following global definitions that provide a standard encoding of the list constructors nil and cons and a list iterator iter:

```
let nil = fun(x)fun(n)fun(c) n
let cons = fun(hd)fun(tl)fun(n)fun(c) c(hd)(tl(n)(c))
let iter = fun(l)fun(n)fun(c) l(n)(c)
```

The syntax of a "list algebra" with selection, projection, and binary join can then be defined as follows:

```
grammar
  simpleTerm:Term |==
    "select" x=ide "in" a=term "where" b=term y=local
  => term<<iter(a)(nil)(fun(x)fun(y)if b then cons(x)(y) else y)>>
  | "project" x=ide "in" a=term "onto" f=fieldList(x) y=local
  => term<<iter(a)(nil)(fun(x)fun(y)cons({f})(y))>>
  | "join" x=ide "in" a=term "," y=ide "in" b=term
    "where" c=term x2=local y2=local
  => term<<iter(a)(nil)(fun(x)fun(x2)iter(b)(x2)(fun(y)fun(y2)
          if c then cons({fst=x snd=y})(y2)else y2))>>
  fieldList(x:Var):Fields ==
     y=ide "," f=fieldList(x)     => fields<<y=x.y f>>
  |                               => fields<<>>
end
```

For example, a selection expression with a variable identifier `x`, a range expression `a`, and a selection predicate `b` is translated into an iterative loop. This loop over `a` has `x` as its loop variable and starting with the empty list `nil` it adds those elements that satisfy the selection predicate `b`:

```
iter(a)(nil)(fun(x)fun(y)if b then cons(x)(y) else y)
```

In this expression, `y` is a fresh local variable which is bound during iteration to the result of the previous iteration step. This translation correctly captures the scoping rules for the list algebra, since the variable `x` is visible only in `b` and not in `a`. Furthermore, global identifiers are visible in `a` and `b`.

The parameterized production `fieldList` demonstrates how parameters may be used to distribute terms (in this case a variable identifier `x`) into multiple subterms. Using the extended grammar one can write, for example, the following queries that use global identifiers `Persons, thirty,` and `equal`:

```
select p in Persons where greater(p.age)(thirty)
project p in Persons onto name, age
join p in Persons, s in Students where equal(p.name)(s.name)
```

Furthermore, it is possible to nest queries and to parameterize queries:

```
fun(limit) select p in
    select p in Persons where greater(p.salary)(limit)
where greater(p.age)(thirty)
```

Note that the identifier `p` in the subquery will be correctly bound to the inner `p` in the generated lambda term.

Simulating SQL expressions is slightly more complicated, since SQL allows the repetition of range expressions to express selections, projections, and n-way joins using a uniform notation:

```
select target(x) from x in a where predicate(x)
select target(x)(y) from x in a, y in b where predicate(x)(y)
select target(x)(y)(z) from x in a, y in b, z in c
where predicate(x)(y)(z)
...
```

Therefore, the rewrite rules have to ensure that the target and the selection expressions appear in the scope of n ($n > 1$) **fun** binders in the generated lambda term. The following grammar uses a recursive, parameterized production `rangeIter` to achieve the desired rewriting:

```
grammar
  simpleTerm:Term |==
     "select" a=term "from" x=ide "in" b=term c=rangeIter(a)
  => term<<iter(b)(nil)(fun(x)c)>>
  rangeIter(a:Term):Term ==
     "," x=ide "in" b=term c=rangeIter(a) y=local
  => term<<fun(y)iter(b)(y)(fun(x)c)>>
    |"where" b=term y=local
  => term<<fun(y)if b then cons(a)(y) else y>>
end
```

For example, a two-way join would be expanded as follows:

select {x.a y.b}		iter(X)(nil)(**fun**(x)
from x **in** X, y **in** Y	$\Rightarrow$	**fun**(z1) iter(Y)(z1)(**fun**(y)
where p(x.c)(y.c)		**fun**(z2) **if** p(x.c)(y.c) **then**
		cons({x.a y.b})(z2) **else** z2))

4 Formalizing Grammars and Parsers

In section 4.1 we describe the rules that are used in the grammar checker (see figure 1) to statically decide whether a sequence of grammar definitions and grammar extensions is well-formed. In section 4.2 we formalize the parse rules that define the mapping from an input stream into a constructed term of the target language. We also present a soundness result of the dynamic parse rules with respect to the static type rules of section 4.1 which guarantees that parsers derived from well-typed grammars return well-formed parse trees. This result is generalized in the full paper to parsers derived from incremental pattern-based grammar definitions.

4.1 Static Typing of Grammar Definitions

To describe the type rules for grammar definitions and extensions, we first define the relevant syntactic objects (sorts, signatures, productions, grammars, grammar sequences).

The syntax for term sorts B and signatures S is defined as follows:

B ::=	Unit | Var | Binder | Label	predefined term sorts
|	$\mathrm{B}^1 \mid \ldots \mid \mathrm{B}^n$	target-language-specific sorts ($n \geq 0$)
S ::=	$(B_1, \ldots, B_k)B$	production signatures ($k \geq 0$)

The abstract syntax of productions is slightly more orthogonal than the concrete syntax we have used in the examples. In particular, terminal productions like `ide(B)` or `"x"` may appear nested within constructor and production argument lists. Furthermore, the syntactic separation of productions into a binding sequence and a constructor application (to the right and left of the `=>`, respectively) is no longer enforced. For example, the production `x=ide => mkTermVar(x)` in the concrete syntax is translated into a simple sequential composition $x = \mathit{ide}(\mathrm{Var})$ **`mkTermVar`**(x).

$p ::=$	unit	unit production
$\mid$	$"x"$	keyword token production
$\mid$	$\mathit{ide}(B)$	variable token production (of sort B)
$\mid$	local	fresh object-language variable
$\mid$	$\mathit{global}(x)$	global object-language variable
$\mid$	x	term variable
$\mid$	$p_1\ p_2$	sequential composition
$\mid$	$x = p_1\ p_2$	pattern variable binding
$\mid$	$p_1 \mid p_2$	choice
$\mid$	$x(p_1, \ldots, p_k)$	non-terminal application ($k \geq 0$)
$\mid$	$c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k)$	sorted constructor application ($k \geq 0$)

The set of constructors $c_{(B_1,\ldots,B_k)B}$ with argument sorts B_i and result sort B contains the target-language-specific constructors (e.g., **mkTermVar, mkTermFun**).

A grammar consists of a list of non-terminal definitions that define a signature, a modification operator, and a production.

$g ::=$	$[]$	empty grammar
$\mid$	$g\ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B\ a\ p$	non-terminal definition
$a ::=$	$==$	grammar addition
$\mid$	$:==$	grammar update
$\mid$	$\mid==$	grammar extension

Each grammar is defined in the scope of its preceding grammar definitions:

$\mathit{gseq} ::=$		empty grammar sequence
$\mid$	$\mathit{gseq}\ g$	grammar composition

A global environment E assigns signatures to non-terminals:

$E ::=$	$\oslash$	empty environment
$\mid$	$E, x : S$	non-terminal x has signature S

A local environment L assigns signatures to term variables:

$L ::=$	$\oslash$	empty environment
$\mid$	$L, x : B$	variable x has sort B

Environment concatenation is written as E, E'. The domain of an environment, denoted by $\mathit{Dom}(E)$, is the set of variables x defined in E. A variable name x may occur more than once in an environment. In this case, the type rules for variables retrieve the rightmost sort or signature assigned to x.

The static semantics of grammars involves the following judgements:

$E; L \vdash p : B$	production p has sort B assuming E and L
$E \vdash g :: E'$	grammar g defines signatures E' consistent with E
$E \vdash g\ \mathit{ok}$	grammar g defines productions consistent with E
$\vdash \mathit{gseq} \Rightarrow E$	grammar sequence gseq defines a final environment E

The structure of the sort rules for productions p resembles the structure of typing rules for terms in a simply-typed lambda calculus:

$$E; L \vdash \mathit{unit} : \text{Unit}$$
$$E; L \vdash "x" : \text{Unit}$$
$$E; L \vdash \mathit{ide}(B) : B$$
$$E; L \vdash \mathit{local} : \text{Binder}$$
$$E; L \vdash \mathit{global}(x) : \text{Var}$$
$$\frac{x \notin \mathit{Dom}(L')}{E; L, x : B, L' \vdash x : B}$$
$$\frac{E; L \vdash p_1 : B \quad E; L \vdash p_2 : B'}{E; L \vdash p_1 \ p_2 : B'}$$

$$\frac{E; L \vdash p_1 : B \quad E; L, x : B \vdash p_2 : B'}{E; L \vdash x = p_1 \ p_2 : B'}$$
$$\frac{E; L \vdash p_1 : B \quad E; L \vdash p_2 : B}{E; L \vdash p_1 \mid p_2 : B}$$
$$\frac{E; L \vdash p_i : B_i \quad 1 \le i \le k}{E; L \vdash c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) : B}$$
$$\frac{E; L \vdash p_i : B_i \quad 1 \le i \le k \quad x \notin \mathit{Dom}(E')}{E, x : (B_1, \ldots, B_k)B, E'; L \vdash x(p_1, \ldots, p_k) : B}$$

Since non-terminal definitions can be recursive, the type checking of a grammar g is performed in two passes. A first pass ($E \vdash g :: E'$) collects the signatures E' of all non-terminals in g, verifies that each non-terminal is defined at most once in g, and asserts that all grammar updates ($x : S{:}{==}p$) and grammar extensions ($x : S|{==}p$) refer to non-terminals with matching signatures in the scope E of g:

$$E \vdash [] :: \emptyset$$
$$\frac{E \vdash g :: E' \quad x \notin \mathit{Dom}(E')}{E \vdash g \ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B == p \ :: \ E', x : (B_1, \ldots, B_k)B}$$
$$\frac{\begin{array}{c} E \vdash g :: E' \quad x \notin \mathit{Dom}(E') \quad a \in \{{:}{==}, |{==}\} \\ E \vdash x : (B_1, \ldots, B_k)B \end{array}}{E \vdash g \ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B \ a \ p \ :: \ E', x : (B_1, \ldots, B_k)B}$$

In a second pass ($E \vdash g \ ok$), the bodies p of all non-terminal definitions in g are checked to match their signatures in E. The rules for parameterized non-terminal definitions resemble the type rules for lambda abstractions:

$$E \vdash [] \ ok$$
$$\frac{E \vdash g \ ok \quad E; \emptyset, x_1 : B_1, \ldots, x_k : B_k \vdash p : B \quad a \in \{==, {:}{==}, |{==}\}}{E \vdash g \ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B \ a \ p \ ok}$$

A sequence of grammars is verified by performing the above two passes on each grammar in the sequence using the environment established by its preceding grammars:

$$\vdash \Rightarrow \emptyset \qquad \frac{\vdash \mathit{gseq} \Rightarrow E \quad E \vdash g :: E' \quad E, E' \vdash g \ ok}{\vdash \mathit{gseq} \ g \ \Rightarrow E, E'}$$

It is possible to derive a simple consistency-checking algorithm from these inference rules as follows: Starting with the proof goal $\vdash \mathit{gseq} \Rightarrow E'$, the inference rules have to be applied "backwards" (from the conclusions to the assumptions). Since for each syntactic construct there is exactly one applicable inference rule, the derivation either reaches the axioms (in time proportional to the size of the grammar) or gets stuck in a configuration where no inference rule can be applied. In the latter case the grammar sequence is rejected as ill-typed. In the next section we prove that parsers derived from well-typed grammars never generate ill-formed syntax trees.

4.2 Parsing and Term Construction

Each non-terminal x in a grammar serves a dual purpose. On the one hand, it determines how to parse an input token stream and how to construct a corresponding term of the target language. On the other hand, it defines how to transform a pattern (a token stream inside <<>> brackets) occurring in an incremental grammar definition into an equivalent production. In this section we describe the parsing of input token streams, while pattern parsing is described in the full paper.

For the purpose of parsing it is convenient to rewrite a grammar sequence $gseq$ into a single grammar g of the form $[], x_1 : S_1{==}p_1, \ldots, x_k : S_k{==}p_k$ $(k \geq 0)$ such that $x_i \neq x_j$ for $i \neq j$. We use the notation:

$$gseq \leadsto g \quad \text{grammar sequence } gseq \text{ normalizes to } g$$

In this rewrite process, grammar updates $(x : S{:==}p)$ and grammar extensions $(x : S{|==}p)$ are eliminated by changing their corresponding original definitions $(x : S{==}p')$ into $x : S{==}p$ and $x : S{==}p \mid p'$, respectively. Name conflicts between grammar additions $x : S{==}\ p$ and $x : S'{==}p'$ $(p \neq p')$ in two grammars of $gseq$ are resolved by consistently renaming one of the non-terminals to a fresh non-terminal x' within in its local scope. It is easy to see that normalization preserves typing, that is, if $gseq \leadsto g$ and $\vdash gseq \Rightarrow E$, then $\vdash g \Rightarrow E'$, where E' is equal to E up to duplicate elimination.

We use the following notation to describe how a production of a grammar g applied to an input stream constructs a term t of the target language:

$$g; M \vdash \langle s, i\rangle\, p \Rightarrow \langle s', i'\rangle\, t$$

It states that production p executed in environment $g; M$ starting in the initial configuration $\langle s, i\rangle$ returns a term t and a final configuration $\langle s', i'\rangle$. A dynamic environment M contains local term variable bindings. A configuration $\langle s, i\rangle$ consists of the input stream s and an integer counter i to generate unique fresh identifiers x_B^i distinct from user-defined identifiers of the form x_B.

The parsing rules are given in figure 3. These rules involve syntactic objects of the following categories:

s ::=			**input streams**
		$*$	empty input stream
	$\mid$	$x :: s$	identifier token
b ::=			**terms**
		$unit$	trivial term
	$\mid$	x_{Binder}	binder identifier
	$\mid$	x_{Var}	variable identifier
	$\mid$	x_{Label}	label identifier
	$\mid$	x_B^i	fresh identifier of sort B $(i \geq 0)$
			$B \in \{\text{Binder}, \text{Var}, \text{Label}\}$
	$\mid$	$c_{(B_1,\ldots,B_k)B}(b_1, \ldots, b_k)$	constructed term $(k \geq 0)$
t ::=			**parse results**
		b	term
	$\mid$	$wrong$	type error
M ::=			**dynamic environments**
		$\oslash$	empty environment
	$\mid$	$M, x = b$	term binding

$$g;M \vdash \langle s,i\rangle\, unit \;\Rightarrow \langle s,i\rangle\, unit$$

$$g;M \vdash \langle x :: s,i\rangle\, "x" \;\Rightarrow \langle s,i\rangle\, unit$$

$$g;M \vdash \langle x :: s,i\rangle\, ide(B) \;\Rightarrow \langle s,i\rangle\, x_B \quad x \notin K(g) \quad B \in \{\text{Binder,Var,Label}\}$$

$$g;M \vdash \langle s,i\rangle\, local \;\Rightarrow \langle s,i+1\rangle\, x^i_{Binder}$$

$$g;M \vdash \langle s,i\rangle\, global(x) \;\Rightarrow \langle s,i\rangle\, x_{Var}$$

$$g;M,x=t,M' \vdash \langle s,i\rangle\, x \;\Rightarrow \langle s,i\rangle\, t \quad x \notin Dom(M')$$

$$g;M \vdash \langle s,i\rangle\, x \;\Rightarrow \langle s,i\rangle\, wrong \quad x \notin Dom(M)$$

$$\frac{\begin{array}{c} g;M \vdash \langle s,i\rangle\, p_1 \;\Rightarrow \langle s',i'\rangle\, t \quad t \neq wrong \\ g;M \vdash \langle s',i'\rangle\, p_2 \;\Rightarrow \langle s'',i''\rangle\, t' \end{array}}{g;M \vdash \langle s,i\rangle\, p_1\ p_2 \;\Rightarrow \langle s'',i''\rangle\, t'} \qquad \frac{g;M \vdash \langle s,i\rangle\, p_1 \;\Rightarrow \langle s',i'\rangle\, wrong}{g;M \vdash \langle s,i\rangle\, p_1\ p_2 \;\Rightarrow \langle s'',i''\rangle\, wrong}$$

$$\frac{\begin{array}{c} g;M \vdash \langle s,i\rangle\, p_1 \;\Rightarrow \langle s',i'\rangle\, t \quad t \neq wrong \\ g;M,x=t \vdash \langle s',i'\rangle\, p_2 \;\Rightarrow \langle s'',i''\rangle\, t' \end{array}}{g;M \vdash \langle s,i\rangle\, x=p_1\ p_2 \;\Rightarrow \langle s'',i''\rangle\, t'} \qquad \frac{g;M \vdash \langle s,i\rangle\, p_1 \;\Rightarrow \langle s',i'\rangle\, wrong}{g;M \vdash \langle s,i\rangle\, x=p_1\ p_2 \;\Rightarrow \langle s',i'\rangle\, wrong}$$

$$\frac{g;M \vdash \langle s,i\rangle\, p_1 \;\Rightarrow \langle s',i'\rangle\, t}{g;M \vdash \langle s,i\rangle\, p_1 \mid p_2 \;\Rightarrow \langle s',i'\rangle\, t} \qquad \frac{g;M \vdash \langle s,i\rangle\, p_2 \;\Rightarrow \langle s',i'\rangle\, t}{g;M \vdash \langle s,i\rangle\, p_1 \mid p_2 \;\Rightarrow \langle s',i'\rangle\, t}$$

$$\frac{g;M \vdash \langle s_{j-1},i_{j-1}\rangle\, p_j \;\Rightarrow \langle s_j,i_j\rangle\, t_j \quad 1 \leq j \leq k}{g;M \vdash \langle s_0,i_0\rangle\, c_{(B_1,\ldots,B_k)B}(p_1,\ldots,p_k) \;\Rightarrow \langle s_k,i_k\rangle\, c_{(B_1,\ldots,B_k)B}(t_1,\ldots,t_k)}$$

$$\frac{\begin{array}{c} g;M \vdash \langle s_{j-1},i_{j-1}\rangle\, p_j \;\Rightarrow \langle s_j,i_j\rangle\, t_j \quad 1 \leq j \leq k \\ (x:(x_1{:}B_1,\ldots,x_k{:}B_k)B{=}{=}p \in g \\ g;\oslash\ x_1=t_1\ \ldots\ x_k=t_k \vdash \langle s_k\rangle\, p \;\Rightarrow \langle s',i'\rangle\, t \end{array}}{g;M \vdash \langle s_0,i_0\rangle\, x(p_1,\ldots,p_k) \;\Rightarrow \langle s',i'\rangle\, t}$$

$$\frac{\begin{array}{c} g;M \vdash \langle s_{j-1},i_{j-1}\rangle\, p_j \;\Rightarrow \langle s_j,i_j\rangle\, t_j \quad 1 \leq j \leq k \\ (x:(x_1{:}B_1,\ldots,x_k{:}B_k)B{=}{=}p) \notin g \\ g;\oslash\ x_1=t_1\ \ldots\ x_k=t_k \vdash \langle s_k\rangle\, p \;\Rightarrow \langle s',i'\rangle\, t \end{array}}{g;M \vdash \langle s_0,i_0\rangle\, x(p_1,\ldots,p_k) \;\Rightarrow \langle s',i'\rangle\, wrong}$$

Figure 3: Parse rules for terms

An input stream is a sequence of identifiers, some of which may have been declared to be keywords (e.g., `"if"`) in g. We use the notation $K(g)$ to denotes the set of keywords defined in productions of g. The parsing rules for terminals use $K(g)$ to distinguish between keywords and identifiers appearing in the input stream.

The sort of a term can be determined without reference to an environment:

$$\mathit{unit} : \mathrm{Unit} \quad x_B : B \quad x_B^i : B \quad \frac{b_1 : B_1 \ldots b_k : B_k}{c_{(B_1,\ldots,B_k)B}(b_1,\ldots,b_k) : B}$$

A dynamic environment M is said to match a static environment L (written as $M \models L$) if its term bindings have names and sorts compatible with the names and sorts in L.

$$\emptyset \models \emptyset \quad \frac{M \models L \quad b : B}{M, x = b \models L, x : B}$$

The following theorem relates the dynamic parse rules in figure 3 with the static type rules presented in section 4.1.

Theorem 1 *(parsing respects typing) For all g, E, L, p, M, s, s', i and i' such that*

1. $\emptyset \vdash g :: E$
2. $\emptyset \vdash g$ *ok*
3. $E; L \vdash p : B$
4. $M \models L$
5. $g; M \vdash \langle s, i\rangle\, p \Rightarrow \langle s', i'\rangle\, t$

$t : B$ holds.

The proof of this theorem can be found in the full paper. In particular, if a non-parameterized ($L = M = \emptyset$) parser with result sort B for a root production p_0 defined in a type-correct grammar g consumes the full input stream s (returning the empty input stream $*$), the parse result t is guaranteed to be of sort B:

Corollary 1 *If*

- $\emptyset \vdash g :: E$
- $\emptyset \vdash g$ *ok,*
- $E; \emptyset \vdash p_0 : B$, *and*
- $g; \emptyset \vdash \langle s, 1\rangle\, p_0 \Rightarrow \langle *, i'\rangle\, t$

then $t : B$ and $t \neq \mathit{wrong}$.

5 An Extensible Parser Package

Extensible grammars as described in this paper were developed in the context of the Tycoon database programming environment [Mat93]. However, as sketched in figure 1, the extensible grammar package was implemented in a way that factors out all target-language dependencies (the base sorts B^i, the abstract syntax tree constructors $c_{(B_1,\ldots,B_k)B}$, and the renaming operation on abstract syntax trees) from the package implementation.

A token stream s is represented as an object with a local state and methods to inspect the current input token and to advance to the next input token.

A parser for terms of a sort B is represented as a function that takes a scanner object and returns a typed abstract syntax tree, modifying the state of the scanner object and a variable counter to generate fresh variable identifiers.

A grammar g_i is represented as an object of an abstract data type encapsulating information about the target language TL and the object language OL_i accepted by g_i. The implementor of a compiler for a language with an extensible grammar links the parser package into the compiler. A grammar for the target language at hand is generated via calls to the parser interface. Finally, a parser for this grammar is generated which in turn is used to parse actual program input.

The following steps have to be taken to generate the grammar g_0 and a parser for the initial object language OL_0. Each of these steps is implemented by a function call to the parser package that passes the grammar as an explicit argument.

1. Creation of an initial (empty) grammar g_0. Arguments to this operation provide information to the parser package about the tokens returned by the scanner and functions to create fresh identifiers. An initial grammar already contains the names of the builtin sorts Label, Var, and Binder.

2. Addition of named sorts to g_0. These sorts correspond to abstract-syntax-tree types in the target-language compiler. For each newly defined sort, an AST copy routine, an AST renaming routine, and a distinguished error value have to be supplied. The error value is generated by the parser package in case of parse errors.

3. Addition of named constructors to g_0. Constructors correspond to functions in the target-language compiler that take $k \geq 0$ typed abstract syntax trees and return an aggregated syntax tree. For each constructor, the list of its argument sorts and its result sort have to be specified.

4. Addition of a concrete syntax for grammar definitions to g_0. Target-language implementors can adopt either the concrete syntax used in this paper (**grammar** ...**end**) or they can define their own tailored syntax for the definition of productions p that match the abstract syntax given in section 4.1.

5. Generation of a parser for g_0. Parser generation involves the calculation of director sets to support efficient LL(1) parsing. Furthermore, variable and non-terminal references are resolved into direct table indices.

6. Parsing of a grammar extension g using the parser generated in the previous step. The grammar extension g defines the mapping from OL_0 terms

to TL terms.

7. Extension of g_0 by g.
8. Generation of a parser for the extended g_0.

A parser for OL_i derived from a grammar g_i returns either a term of the target language proper, or an abstract syntax tree for an incremental syntax extension g_Δ. In the latter case the parser package is invoked to check the type correctness of g_Δ in the scope of the environment E_i established by the current grammar g_i. If this check succeeds, the extended grammar is obtained by normalizing the grammar sequence $g_i, g_\Delta \rightsquigarrow g_{i+1}$. Finally, a new parser is generated for g_{i+1}; this parser can then be used to parse further input in the extended languageOL_{i+1}.

If the parsing result is a term t of the target language, the parser package also returns a list of variable renamings. These renamings have to be performed by the target-language compiler in t to establish bindings to global variable identifiers (see section 3.3).

6 Comparison with Related Work

Syntactic extensibility has been studied previously in the context of programming languages and theorem provers.

Linguistic reflection [SMM91, SSS$^+$92, SSF92, Kir92] in persistent programming languages has been used to add high-level (query) notations to strongly-typed programming languages. These extensions are achieved by executing user-defined code at compile time to transform syntax trees returned from the parser prior to further processing by the type checker and code generator. Our approach differs from this work since we are able to guarantee the termination of compilation, even when our transformation operations are defined recursively. Furthermore, we are not aware of work in the context of linguistic reflection to handle correctly the problematic binding situations sketched in section 3.3.

Some non-persistent language implementations, like CAML and SML, integrate YACC or a similar parser generator that allows them to introduce new syntax [MR92]. If the new syntax is to be mixed with the old one, the new syntax must be quoted in some way. Instead, we can freely intermix new and old syntax without special quotations.

Hygienic macros [KFFD92, Koh86] have goals similar to those of our extensible grammars; these macros also work on the abstract syntax and avoid binding anomalies. However, these macros account only for explicit (parameterized) macro calls and not for more liberal keyword-based syntax extensions. Hygienic macros employ a multi-pass time-stamping algorithm to prevent variable capture; this algorithm is different from our one-pass renaming algorithm. Furthermore, we do not handle quotation and antiquotation in the style of Lisp.

Griffin [Gri88] has enumerated desirable properties of notational definitions and has studied their formalization. Unlike Griffin who translates notations to combinator form, we are able to handle variables bound to non-local binders in patterns. Moreover, while Griffin discusses abstract translations, we provide a specific grammar definition technique and an efficient parsing algorithm. Parsing is efficient because it is LL(1) and because it avoids the creation of intermediate parse trees, producing abstract syntax trees that do not require normalization.

Bove and Arbilla [BA92] discuss how to use explicit substitutions to implement syntax extensions. Theirs is an elegant idea that may be exploited in systems where the target compiler supports explicit substitutions. As in the previous case, their work does not describe a parsing algorithm, but presents an interesting theory.

7 Concluding Remarks

Extensible grammars avoid many of the problems associated with traditional macro-expansion or program-rewrite tools by sort constraints at grammar-definition time and by a careful handling of identifiers in binding constructs. Furthermore, since our work extends the well-understood parser technology by a small set of concepts, extensible parsers can be integrated with little overhead in today's compilation environments.

Traditional database programming languages have a bias towards a specific data model by providing built-in syntactic support tailored to the structures and operations of that data model. In a programming environment equipped with extensible grammars, such syntactic forms can be eliminated from the core language definition and can be introduced in application libraries shared by larger user communities.

References

[ASU87] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1987.

[BA92] A. Bove and L. Arbilla. A Confluent Calculus of Macro Expansion and Evaluation. In *ACM Conference on Lisp and Functional Programming*, pages 278–287, 1992.

[BTBN91] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.

[Car93] L. Cardelli. An Implementation of $F_{<:}$. Report 97, Digital Equipment Corporation, Systems Research Center, 1993.

[Gri88] T. Griffin. Notational definition – A formal account. In *Proceedings Symposium on Logic in Computer Science*, pages 372–383, 1988.

[KFFD92] E. Kohlbecker, D.P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Conference on Lisp and Functional Programming*, 1992.

[Kir92] G.N.C. Kirby. Persistent Programming with Strongly Typed Linguistic Reflection. FIDE Technical Report Series FIDE/92/40, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, 1992.

[Koh86] E.E. Kohlbecker. *Syntactic extensions in the programming language LISP*. PhD thesis, Indiana University, 1986.

[KR77] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1977.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German.)

[MR92] M. Mauny and D. Rauglaudre. Parsers in ML. In *ACM Conference on Lisp and Functional Programming*, 1992.

[MS91] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.

[Naq89] S.A. Naqvi. Stratification as a Design Principle in Logical Query Languages. In *Proceedings of the Second International Workshop on Database Programming Languages*, Salishan, Oregon, 1989.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon*, pages 46–57, 1989.

[PT93] B. Pierce and D. Turner. Object-Oriented Programming without Recursive Types. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 299–312, 1993.

[SFL83] J.M. Smith, S. Fox, and T. Landers. ADAPLEX: Rationale and Reference Manual (2nd ed.). Technical report, Computer Corporation of America, Cambridge, Mass., 1983.

[SMM91] D. Stemple, R. Morrison, and Atkinson M. Type-safe Linguistic Reflection. In *Database Programming Languages: Bulk Types and Persistent Data*, pages 357–362, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.

[SQL87] ISO. *Standard ISO 9075, Information processing systems - Database language SQL*, 1987.

[SS91] D. Stemple and T. Sheard. A Recursive Base for Database Programming Primitives. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, volume 504 of *Lecture Notes in Computer Science*, 1991.

[SSF92] D. Stemple, T. Sheard, and L. Fegaras. Linguistic Reflection: A Bridge from Programming to Database Languages. In *Proc. HICSS, Hawaii*, pages 46–55, 1992.

[SSS88] D. Stemple, A. Socorro, and T. Sheard. Formalizing Objects for Databases using ADABTPL. In *Advances in Object-Oriented Database Systems*, pages 110–172, 1988.

[SSS$^+$92] D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fegaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, and S. Alagic. Type-Safe Linguistic Reflection: A Generator Technology. Research Report CS/92/6, Univ. of St. Andrews, Dept. of Comp. Science, 1992.

[Tri91] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages*, Nafplion, Greece, 1991. Morgan Kaufmann Publishers.