February 21, 1994

SRC Research Report

121

Extensible Syntax with Lexical Scoping

Luca Cardelli, Florian Matthes, and Martín Abadi

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Extensible Syntax with Lexical Scoping

Luca Cardelli, Florian Matthes, and Martín Abadi

February 21, 1994

A preliminary version of this paper appeared in the Proceedings of the Fifth Workshop on Database Programming Languages, 1993, under the title "Extensible Grammars for Language Specialization."

Florian Matthes is at the University of Hamburg. This work was supported by the European Commission, ESPRIT, EC/US-FIDE Collaborative Activity, 006:9829. Part of the work took place at Digital's Systems Research Center.

Authors' Abstract

A frequent dilemma in programming language design is the choice between a language with a rich set of notations and a small, simple core language. We address this dilemma by proposing extensible grammars, a syntax-definition formalism for incremental language extensions and restrictions.

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 constructs from database programming languages, such as SQL query expressions. We then give a formal description of the underlying rules for parsing, transformation, and substitution. Finally, we sketch how these rules are exploited in an implementation of a generic, extensible parser package.

Contents

1 Introduction

A frequent dilemma in programming language design is the choice between a user-friendly language with a rich set of notations and a small, conceptually simple core language. We address this dilemma by introducing extensible grammars, a syntax-definition formalism for incremental, problem-specific language extensions and restrictions.

The translation of programs written in rich object languages into a small core language is defined via syntax-directed patterns. The translation resembles macro expansion, with some essential differences. Traditional macro-expansion and program-rewriting tools attempt to manipulate programs as mere strings or trees. This is the source of many of their well-known defects. In contrast, our extensible grammars recognize and respect the scoping structure of programs.

The features of our approach are as follows:

- Lexical scoping is strictly preserved. Therefore, we can introduce new binding constructs like quantifiers, iterators, and type declarations, while avoiding problems with unwanted name clashes ("variable captures").
- Parsing remains independent of type checking and evaluation. It always terminates.
- We can determine, statically, what is the legal syntax in any region of the text of a program.
- We can freely introduce new notation and mix it with existing notation without special quotations, antiquotations, or explicit macro calls.
- New notation can be defined in terms of old notation, incrementally.
- Our syntax-definition package is language-independent.

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.

We motivate and illustrate the use of extensible grammars with examples from various domains, but we emphasize the application of extensible grammars for database programming. Current database systems typically rely on macro preprocessors in order to embed query notations in host languages like C or Cobol. Our extensible grammars may serve as a safe alternative to macros in this context.

Both syntax extensions and syntax restrictions occur commonly in practice, and extensible grammars are designed to support them both.

Syntax extensions provide syntactic sugar for problem-specific abstraction. Syntax extensions have long been used in Lisp systems; recent work has focused on avoiding variable captures (see section 6). Notational definitions make sense not only in programming but also in mathematics, in particular in logical frameworks [Gri88].

Syntax extensions have a variety of applications in database programming. For example, embedded query notations like the relational calculus, the relational algebra, iteration statements, or set comprehensions can be introduced as abstractions defined from 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. One rationale behind restrictions is to facilitate meta-level reasoning and optimizations tailored to a particular application domain. In addition, syntax restrictions can serve to enforce the use of subsets of languages. For instance, a syntax restriction may forbid imperative programming in student projects.

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 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, and in particular 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 next section gives an overview of the issues that must be addressed by a formalism for language extensions and restrictions. In section 3 we introduce extensible grammars by examples. An initial grammar for the lambda calculus is extended incrementally with new syntactic forms such as let-bindings, conditionals, and query notations. In section 4 we define the static type rules for grammar definitions and the semantics of parsers generated from extensible grammars. We also present a soundness result for the type system with respect to the evaluation semantics. In section 5 we describe the implementation of an extensible parser module for the Tycoon database environment [Mat93]. Finally, section 6 is a comparison with other approaches to syntax extension.

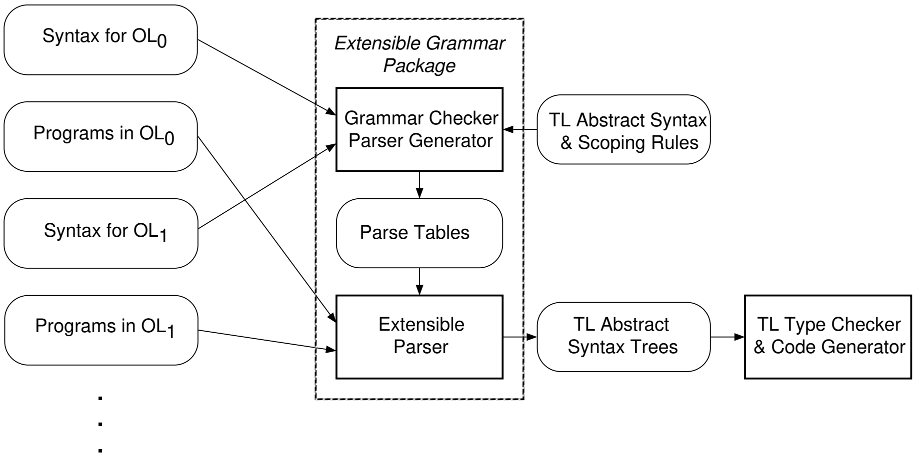

Figure 1: The syntax-definition scenario

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 also allows 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
```

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:

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

Subsection 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    ⇒    begin return plus(x,1) end
```

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 subsections 3.2 and 3.3. The definition of an SQL-like grammar in our formalism is given in subsection 3.4.

Extensible grammars require extensible parsers. That is, a parser has to be dynamically extensible 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 done at grammar-definition time and includes standard grammar analysis [ASU87] to avoid the first two problems. To address the third problem, we develop a typing discipline on productions (see subsection 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 causes a name clash between two declarations of the variable `z` in the same scope.

In order to solve the scoping problems caused by rewriting inside binding structures, a formalization of the scoping rules of the target language is required. 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 identifiers in rewrite rules could occur. Note that this solution is not an option for a simple token-based preprocessor. Subsection 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 subsection 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.

3.1 Initial Grammar Definitions

This subsection 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:

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

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 sorts, specific to the target language:

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

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

`Binder`	identifiers appearing in binding positions
`Var`	identifiers appearing in 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 subsection.

This grammar consists of four mutually recursive productions that define left-associativity of applications and precedence of applications over abstractions. 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. For example, the production `termIter` takes a parameter `a` of sort `Term` and returns a term of sort `Term`.

The body of 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,

```
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

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 subsection 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 subsection 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 the 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` $\Rightarrow$ (**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 subsection 3.2, abstract syntax trees produced by actions are specified with explicit constructor applications. In this subsection 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, when the token stream (**fun**`(y) b)(a)` is parsed as a `term`, the pattern `term<<(`**fun**`(y) b)(a)>>` yields the nested constructor application `mkTermApp(mkTermFun(y b) a)`. Note that the concrete syntax and the binding structure inside the `<<>>` brackets is defined by the grammar that is valid before the enclosing **grammar end** block.

The keyword **pIde** followed by a sort identifier is used in the initial grammar definition (see subsection 3.1) to indicate those positions in the input syntax where pattern variables may appear. For example, `f` is a pattern variable of sort `Fields` in the pattern `<<{f}>>`. 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.

To avoid variable captures and name clashes, many pattern-based syntax extensions require the introduction of fresh identifiers, that is, identifiers distinct from other identifiers appearing in `Binding` and `Var` positions. 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 renaming function is applied to the parsed term. The renaming function depends on the target language. In this case, 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 subsection 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 of the 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. Starting with the empty list `nil`, the loop 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 subsection 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 subsection 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 subsection 4.1. This result guarantees that parsers derived from well-typed grammars return well-formed parse trees. In subsection 4.3 we generalize the result 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
|	B^1 | ... | B^n	sorts specific to the target language
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 $x = \mathbf{ide}(\mathrm{Var})\ \mathtt{mkTermVar}(x)$.

p ::=		$\mathbf{unit}$	unit production
	|	$"x"$	keyword token production
	|	$\mathbf{ide}(B)$	variable token production (of sort B)
	|	$\mathbf{local}$	fresh object-language variable
	|	$\mathbf{global}(x)$	global object-language variable
	|	x	term variable
	|	$p_1\ p_2$	sequential composition
	|	$x = p_1\ p_2$	pattern variable binding
	|	$p_1 \mid p_2$	choice
	|	$x(p_1, \ldots, p_k)$	non-terminal application ($k \geq 0$)
	|	$c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k)$	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 constructors specific to the target language (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
	|	$g\ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B\ a\ p$	non-terminal definition
a ::=		==	grammar addition
	|	:==	grammar update
	|	|==	grammar extension

Each grammar (a block of possibly recursive definitions) is defined in the scope of its preceding grammar definitions:

gseq ::=			empty grammar sequence
	|	*gseq g*	grammar composition

A global environment E assigns signatures to non-terminals:

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

A local environment L assigns signatures to term variables:

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

Environments are ordered so that they model block-structured scoping. Environment concatenation is written as E, E'. The domain of an environment, denoted by $Dom(E)$, 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 four kinds of judgments, defined in the remainder of this subsection:

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

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

$$E; L \vdash \mathbf{unit} : \mathrm{Unit}$$

$$E; L \vdash "x" : \mathrm{Unit}$$

$$E; L \vdash \mathbf{ide}(B) : B$$

$$E; L \vdash \mathbf{local} : \mathrm{Binder}$$

$$E; L \vdash \mathbf{global}(x) : \mathrm{Var}$$

$$\frac{x \notin Dom(L')}{E; L, x : B, L' \vdash x : B}$$

$$\frac{E; L \vdash p_1 : B \qquad E; L \vdash p_2 : B'}{E; L \vdash p_1 \; p_2 : B'}$$

$$\frac{E; L \vdash p_1 : B \qquad E; L, x : B \vdash p_2 : B'}{E; L \vdash x = p_1 \; p_2 : B'}$$

$$\frac{E; L \vdash p_1 : B \qquad E; L \vdash p_2 : B}{E; L \vdash p_1 \mid p_2 : B}$$

$$\frac{E; L \vdash p_i : B_i \quad 1 \leq i \leq k}{E; L \vdash c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) : B}$$

$$\frac{E; L \vdash p_i : B_i \quad 1 \leq i \leq k \qquad x \notin Dom(E')}{E, x : (B_1, \ldots, B_k)B, E'; L \vdash x(p_1, \ldots, p_k) : B}$$

The type checking of a grammar g is performed in two passes in order to handle recursive non-terminal definitions correctly. 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:

$$E \vdash [] :: \oslash$$

$$\frac{E \vdash g :: E' \quad x \notin Dom(E')}{E \vdash g\ x : (x_1{:}B_1, \ldots, x_k{:}B_k)B == p\ :: \ E', x : (B_1, \ldots, B_k)B}$$

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

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:

$$E \vdash []\ \mathbf{ok}$$

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

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

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

It is possible to derive a simple consistency-checking algorithm from these inference rules as follows: Starting with the proof goal $\vdash gseq \Rightarrow E$, 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 subsection 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 subsection we describe the parsing of input token streams, while pattern parsing is described in the subsection 4.3.

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:

$gseq \rightsquigarrow g$ grammar sequence *gseq* 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 its local scope. It is easy to see that normalization preserves typing, that is, if $gseq \rightsquigarrow g$ 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:

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

This formula 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^i_B 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^i_B	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**
		$\emptyset$	empty environment
	$\mid$	$M, x = b$	term binding

An input stream is a sequence of identifiers, some of which may have been declared to be keywords in g (e.g., `"if"`, `"("`). We use the notation $K(g)$ to denote 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Parse rules for terms

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

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

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.

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

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

Theorem 1 *(Parsing respects typing) If g, E, L, p, M, s, s', i, and i' are such that*

- $\oslash \vdash g :: E$
- $\oslash \vdash g$ **ok**
- $E; L \vdash p : B$
- $M \models L$
- $g; M \vdash \langle s, i\rangle\, p \Rightarrow \langle s', i'\rangle\, t$

then $t : B$.

The proof of this theorem can be found in the appendix. In particular, if a non-parameterized ($L = M = \oslash$) 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*

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

then $t : B$ and $t \neq$ **wrong**.

It should be noted that the parse rules in figure 3 are non-deterministic due to the rules given for the choice operator $p_1 \mid p_2$. In the actual implementation, each choice operator in a grammar g is replaced (at grammar-definition time) by a choice construct of the form $T_1 : p_1 \mid T_2 : p_2$. The set $T_i \subseteq K(g) \cup \{\mathbf{ide}\}$ of possible start tokens for phrases accepted by p_i is called the director set for p_i. The refined parse rules perform a deterministic choice based on the current input token:

$$\text{Token}(x, g) = \begin{cases} "x" & \text{if } "x" \in K \\ \mathbf{ide} & \text{otherwise} \end{cases}$$

$$\frac{g; M \vdash \langle x :: s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, t \quad \text{Token}(x, g) \in T_1}{g; M \vdash \langle x :: s, i\rangle\, T_1 : p_1 \mid T_2 : p_2 \Rightarrow \langle s', i'\rangle\, t}$$

$$\frac{g; M \vdash \langle x :: s, i\rangle\, p_2 \Rightarrow \langle s', i'\rangle\, t \quad \text{Token}(x, g) \in T_2}{g; M \vdash \langle x :: s, i\rangle\, T_1 : p_1 \mid T_2 : p_2 \Rightarrow \langle s', i'\rangle\, t}$$

The computation of the director sets is accomplished by standard algorithms developed for non-incremental LL(1) parsers in time linear to the size of the grammar [WG85]. A grammar is rejected as ambiguous if it contains a production $T_1 : p_1 \mid T_2 : p_2$ where $T_1 \cap T_2 \neq \{\}$.

4.3 Pattern-based Production Generation

In the previous subsection we did not consider the parsing of productions defined by means of patterns enclosed in `<<>>`. In this subsection we describe a translation of such productions into simpler ones, so that they are covered by the parse rules and the theorem given in the previous subsection.

A pattern x`<<`s`>>` is a pair of a token stream s and a non-terminal x that defines which production is to be used to parse s. The result of parsing s is itself a production p that neither contains terminal productions that depend on input tokens nor choice operators $(p_1 \mid p_2)$. If p is later executed within an environment that defines bindings for the pattern variables occurring in s, then p performs the necessary steps to instantiate correctly (macro-expand) the pattern. These steps include defining term bindings, performing term constructor applications, and introducing fresh variables, where necessary.

We describe the effect of parsing a pattern `p<<s>>` with the notation

$$g; L; R \vdash \langle s\rangle\, p \Rightarrow \langle s'\rangle\, p'$$

This formula states that production p when executed in environment $g; L; R$ starting with a token stream s returns a token stream s' (the unread tokens of s) and a constructed production p'.

The environment L contains the names and sorts of the pattern variables bound in the scope enclosing the pattern. For example, L contains $\oslash, x :$ Binder$, a :$ Term$, b :$ Term for the pattern `term<<...>>` in the following grammar:

$$g; L; R \vdash \langle s \rangle\, \mathbf{unit} \;\Rightarrow \langle s \rangle\, \mathbf{unit}$$

$$g; L; R \vdash \langle x :: s \rangle\, \text{“}x\text{''} \;\Rightarrow \langle s \rangle\, \mathbf{unit}$$

$$g; L; R \vdash \langle x :: s \rangle\, \mathbf{ide}(\mathit{Var}) \;\Rightarrow \langle s \rangle\, \mathbf{global}(x) \quad x \notin K(g), x \notin \mathit{Dom}(L)$$

$$g; L, x : B, L'; R \vdash \langle x :: s \rangle\, \mathbf{ide}(B) \;\Rightarrow \langle s \rangle\, x \quad x \notin K(g), x \notin \mathit{Dom}(L')$$

$$g; L; R \vdash \langle s \rangle\, \mathbf{local} \;\Rightarrow \langle s \rangle\, \mathbf{local}$$

$$g; L; R \vdash \langle s \rangle\, \mathbf{global}(x) \;\Rightarrow \langle s \rangle\, \mathbf{global}(x)$$

$$g; L; R, x \leftarrow x', R' \vdash \langle s \rangle\, x \;\Rightarrow \langle s \rangle\, x' \quad x \notin \mathit{Dom}(R')$$

$$\frac{\begin{array}{c} g; L; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p_1' \\ g; L; R \vdash \langle s' \rangle\, p_2 \;\Rightarrow \langle s'' \rangle\, p_2' \end{array}}{g; L; R \vdash \langle s \rangle\, p_1\ p_2 \;\Rightarrow \langle s'' \rangle\, p_1'\ p_2'}$$

$$\frac{\begin{array}{c} g; L; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p_1' \\ g; L; R\ x \leftarrow x' \vdash \langle s' \rangle\, p_2 \;\Rightarrow \langle s'' \rangle\, p_2' \quad x' \notin \mathit{Dom}(L) \cup \mathit{Ran}(R) \end{array}}{g; L; R \vdash \langle s \rangle\, x = p_1\ p_2 \;\Rightarrow \langle s'' \rangle\, x' = p_1'\ p_2'}$$

$$\frac{g; L; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p_1'}{g; L; R \vdash \langle s \rangle\, p_1 \mid p_2 \;\Rightarrow \langle s' \rangle\, p_1'} \qquad \frac{g; L; R \vdash \langle s \rangle\, p_2 \;\Rightarrow \langle s' \rangle\, p_2'}{g; L; R \vdash \langle s \rangle\, p_1 \mid p_2 \;\Rightarrow \langle s' \rangle\, p_2'}$$

$$\frac{g; L; R \vdash \langle s_{j-1} \rangle\, p_j \;\Rightarrow \langle s_j, \rangle\, p_j' \quad 1 \leq j \leq k}{g; L; R \vdash \langle s_0 \rangle\, c_{(B_1, \ldots, B_k)B}(p_1, \ldots, p_k) \;\Rightarrow \langle s_k \rangle\, c_{(B_1, \ldots, B_k)B}(p_1', \ldots, p_k')}$$

$$\frac{\begin{array}{c} g; L; R \vdash \langle s_{j-1} \rangle\, p_j \;\Rightarrow \langle s_j, \rangle\, p_j' \quad 1 \leq j \leq k \\ R' = R, x_1 \leftarrow x_1', \ldots, x_k \leftarrow x_k' \quad x_i' \notin \mathit{Dom}(E) \cup \mathit{Ran}(R) \quad x_i' \neq x_j' \text{ for } i \neq j \\ g; L; R' \vdash \langle s_k \rangle\, p \Rightarrow \langle s' \rangle\, p' \quad (x : (x_1{:}B_1, \ldots, x_k{:}B_k)B{=}{=}p) \in g \end{array}}{g; L; R \vdash \langle s_0 \rangle\, x(p_1, \ldots, p_k) \;\Rightarrow \langle s' \rangle\, x_1' = p_1' \ \ldots\ x_k' = p_k'\ p'}$$

Figure 4: Parse rules for patterns in incremental grammar definitions

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

The notation $g; L; R \vdash \langle s \rangle\, p \;\Rightarrow \langle s' \rangle\, p'$ uses a set R of renamings to avoid name conflicts between the pattern variables in L and the term variables introduced during pattern parsing:

$$
\begin{array}{rcll}
R & ::= & \oslash & \text{empty renaming} \\
 & | & R, x \leftarrow x' & \text{rename } x \text{ by } x'
\end{array}
$$

When $R = \oslash, x_1 \leftarrow x'_1, \ldots, x_n \leftarrow x'_n$, we write $Ran(R)$ for the set $\{x'_1, \ldots, x'_n\}$.

The complete set of rules for parsing patterns is given in figure 4. As an example, here is the production p' that results from parsing the pattern `term<<(fun(x) b)(a)>>` in the environment $L = \oslash, x : \text{Binder}, a : \text{Term}, b : \text{Term}$ using the grammar defined in figure 2:

```
a₁=(x₁=x
    a₂=(a₃=(a₄=b
            a₄)
        b₂=(a₃)
        b₂)
    mkTermFun(x₁ a₂))
b₁=(b₃=(a₅=(a₆=a
            a₆)
        b₄=(a₅)
        b₄)
    a₄=mkTermApp(a₁ b₃)
    a₄)
b₁
```

In this example, we use subscripted identifiers for fresh term variable identifiers introduced during the translation process. Furthermore, we use brackets and indentation to indicate the scope of these variable identifiers. By removing redundant intermediate bindings, the generated production can be simplified to `mkTermApp(mkTermFun(x b) a)`, as expected.

The following theorem states that the successful parsing of a pattern `p<<s>>` using a production p of a type-correct grammar g yields a well-typed production.

Theorem 2 *If g, E, L_1, L_2, L_3, p, B, and R are such that*

- $\oslash \vdash g :: E$
- $\oslash \vdash g$ **ok**
- $E; L_3 \vdash p : B$ *where* $L_3 = \oslash, x_1 : B_1, \ldots, x_n : B_n$

- $R = \oslash, x_1 \leftarrow x'_1, \ldots, x_n \leftarrow x'_n$ *with* $x'_i \neq x'_j$ *for* $i \neq j$ *and* $Ran(R) \cap Dom(L_1) = \oslash$

- $g; L_1; R \vdash \langle s \rangle\, p \;\Rightarrow \langle s' \rangle\, p'$

then $E; L_1, L_2 \vdash p' : B$.

The proof can be found in the appendix. By specializing this theorem to a non-parameterized production p_0 with result sort B, we obtain that a pattern p_0<<s>> in an arbitrary local environment L can be translated into an equivalent production p' that has the result sort B also:

Corollary 2 *If*

- $\oslash \vdash g :: E$

- $\oslash \vdash g$ **ok**

- $E; \oslash \vdash p_0 : B$

- $g; L; \oslash \vdash \langle s \rangle\, p_0 \;\Rightarrow \langle * \rangle\, p'$

then $E; L, \oslash \vdash p' : B$.

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; the function modifies the state of the scanner object and a variable counter used for generating 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, and it 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 about the tokens returned by the scanner, and functions for creating fresh identifiers. An initial grammar already contains the names of the built-in 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 either adopt the concrete syntax used in this paper (**grammar** ...**end**), or define their own tailored syntax for the definition of productions p that match the abstract syntax given in subsection 4.1.

5. Generation of a parser for g_0. Parser generation involves calculating 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 language OL_{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 subsection 3.3).

6 Comparison with Related Work

Extensibility has been studied previously in the context of programming languages and theorem provers [Dow90]. In the early work on language extensibility [Gal74, Sta75], both syntax and semantics could be modified arbitrarily, sometimes with disastrous effects [Chr90]. Traditional macro facilities allow only syntax extensions. We have already discussed some of the defects of macros. Several recent works propose improvements on macros.

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; this code transforms 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 subsection 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; it is also possible to remove existing keywords by redefining non-terminals with the `:==` operator.

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.

Traditionally, the most sophisticated macro-definition facilities have been developed for Lisp-like languages; the regular syntactic structure of Lisp simplifies program manipulation. Recent work has extended AST macro manipulation to syntactically complex languages. For example, Weise and Crew use a full C language extended with patterns as a preprocessor for the C language [WC93]; their macros have syntactic types (our sorts) that guarantee the generation of well-formed AST's. We have achieved considerable flexibility in the manipulation of complex languages, but without resorting to a computationally complete macro language. This way, we can guarantee termination of the parsing phase.

7 Conclusion

Extensible grammars avoid many of the problems associated with traditional tools for macro expansion and program rewriting, by enforcing sort constraints at grammar-definition time and by respecting lexical scoping. Furthermore, since extensible parsers introduce only a small set of new concepts, they can be integrated with little overhead in current 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 shared application libraries.

Acknowledgments

Bill Kalsow and Cynthia Hibbard made useful comments on this paper.

Appendix

This appendix contains the proofs of our theorems.

Proof of Theorem 1

The proof is carried out by induction on the parsing derivations with the rules in figure 3. We treat the rules one by one:

- $g; M \vdash \langle s, i\rangle\, \mathbf{unit} \;\Rightarrow \langle s, i\rangle\, \mathbf{unit}$

  Suppose $E; L \vdash \mathbf{unit} : B$. Then $B = \text{Unit}$ according to the type rules for productions. Moreover, by definition $\mathbf{unit} : \text{Unit}$.

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

  Suppose $E; L \vdash "x" : B$. Then $B = \text{Unit}$ according to the type rules. Again, $\mathbf{unit} : \text{Unit}$.

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

  Suppose $E; L \vdash \mathbf{ide}(B) : B'$. According to the type rules, B' can be only B and matches the type of the concrete term $x_B : B$.

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

  Suppose $E; L \vdash \mathbf{local} : B$. According to the type rules, B has to be Binder. Moreover, the term x^i_{Binder} has type Binder.

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

  Suppose $E; L \vdash \mathbf{global}(x) : B$. Sort B has to be Var. Moreover, x_{Var} : Var.

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

  Suppose $E; L \vdash x : B$. According to the type rules, L has to be of the form $L', x : B, L''$ such that $x \notin Dom(L'')$. Since $M \models L$, $b : B$.

- $g; M \vdash \langle s, i\rangle\, x \;\Rightarrow \langle s, i\rangle\, \mathbf{wrong}$ where $x \notin Dom(M)$

  Suppose $E; L \vdash x : B$ to obtain a contradiction. According to the type rules, $x \in Dom(L)$. However, since $M \models L$, this implies $x \in Dom(M)$, contradicting the side condition of the parsing rule.

- $g; M \vdash \langle s, i\rangle\, p_1\; p_2 \;\Rightarrow \langle s'', i''\rangle\, t'$ where $t' \neq \mathbf{wrong}$

  According to the parse rules there has to be a $t \neq \mathbf{wrong}$ such that $g; M \vdash \langle s, i\rangle\, p_1 \;\Rightarrow \langle s', i'\rangle\, t$ and $g; M \vdash \langle s', i'\rangle\, p_2 \;\Rightarrow \langle s'', i''\rangle\, t'$. Moreover, suppose that $E; L \vdash p_1\; p_2 : B'$. According to the type rules $E; L \vdash p_2 : B'$. Applying the induction hypothesis we obtain $t' : B'$.

- $g; M \vdash \langle s, i\rangle\, p_1\ p_2 \Rightarrow \langle s'', i''\rangle\, \mathbf{wrong}$ and $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, \mathbf{wrong}$

  Suppose $E; L \vdash p_1\ p_2 : B'$ to obtain a contradiction. By the type rules $E; L \vdash p_1 : B$. However, applying the induction hypothesis, this contradicts the assumptions since there is no B such that $\mathbf{wrong} : B$.

- $g; M \vdash \langle s, i\rangle\, p_1\ p_2 \Rightarrow \langle s'', i''\rangle\, \mathbf{wrong}$ and $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, t \quad t \neq \mathbf{wrong}$

  Suppose $E; L \vdash p_1\ p_2 : B'$ to obtain a contradiction. By the type rules $E; L \vdash p_2 : B'$. According to the parse rules $g; M \vdash \langle s', i'\rangle\, p_2 \Rightarrow \langle s'', i''\rangle\, \mathbf{wrong}$. Applying the induction hypothesis leads to the false statement $\mathbf{wrong} : B'$.

- $g; M \vdash \langle s, i\rangle\, x = p_1\ p_2 \Rightarrow \langle s'', i''\rangle\, t'$ where $t' \neq \mathbf{wrong}$

  Suppose $E; L \vdash x = p_1\ p_2 : B'$. According to the type rules, it must be that, for some B, $E; L \vdash p_1 : B$ and $E; L, x : B \vdash p_2 : B'$. According to the parse rules $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, t$ and $g; M, x = t \vdash \langle s', i'\rangle\, p_2 \Rightarrow \langle s'', i''\rangle\, t'$. By induction hypothesis $t : B$. Hence $M, x = t \models L, x : B$ and by applying the induction hypothesis again one establishes that $t' : B'$.

- $g; M \vdash \langle s, i\rangle\, x = p_1\ p_2 \Rightarrow \langle s'', i''\rangle\, \mathbf{wrong}$ and
  $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, \mathbf{wrong}$

  Suppose $E; L \vdash x = p_1\ p_2 : B'$ to obtain a contradiction. According to the type rules, it must be that, for some B, $E; L \vdash p_1 : B$. Applying the induction hypothesis, this leads to a contradiction since there is no B such that $\mathbf{wrong} : B$.

- $g; M \vdash \langle s, i\rangle\, x = p_1\ p_2 \Rightarrow \langle s'', i''\rangle\, \mathbf{wrong}$ and
  $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, t \quad t \neq \mathbf{wrong}$

  Suppose $E; L \vdash x = p_1\ p_2 : B'$ to obtain a contradiction. According to the type rules, it must be that, for some B, $E; L, x : B \vdash p_2 : B'$. Applying the induction hypothesis, this leads to a contradiction since there is no B' such that $\mathbf{wrong} : B'$.

- $g; M \vdash \langle s, i\rangle\, p_1 \mid p_2 \Rightarrow \langle s', i'\rangle\, t$ and $g; M \vdash \langle s, i\rangle\, p_1 \Rightarrow \langle s', i'\rangle\, t$

  Suppose $E; L \vdash p_1 \mid p_2 : B$. According to the type rules this implies $E; L \vdash p_1 : B$. Applying the induction hypothesis to the derivation for p_1 establishes $t : B$.

- $g; M \vdash \langle s, i\rangle\, p_1 \mid p_2 \Rightarrow \langle s', i'\rangle\, t$ and $g; M \vdash \langle s, i\rangle\, p_2 \Rightarrow \langle s', i'\rangle\, t$

  Suppose $E; L \vdash p_1 \mid p_2 : B$. According to the type rules this implies $E; L \vdash p_2 : B$. Applying the induction hypothesis to the derivation for p_2 establishes $t : B$.

- $g; M \vdash \langle s_0, i_0 \rangle\, c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) \Rightarrow \langle s_k, i_k \rangle\, t$ where $t \neq$ **wrong**

  Suppose $E; L \vdash c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) : B'$. The type rules imply $B' = B \neq$ **wrong** and $E; L \vdash p_j : B_j$ for $1 \leq j \leq k$. Moreover, the parse rules guarantee that t is of the form $c_{(B_1,\ldots,B_k)B}(t_1, \ldots, t_k)$ with $g; M \vdash \langle s_{j-1}, i_{j-1} \rangle\, p_j \Rightarrow \langle s_j, i_j \rangle\, t_j$ for $1 \leq j \leq k$. Applying the induction hypothesis to the derivations for p_1 through p_k establishes $t_1 : B_1 \ldots t_k : B_k$. Using the definition for the types of base terms, we obtain $t = c_{(B_1,\ldots,B_k)B}(t_1, \ldots, t_k) : B$ and this case is settled.

- $g; M \vdash \langle s_0, i_0 \rangle\, x(p_1, \ldots, p_k) \Rightarrow \langle s', i' \rangle\, t$ where $t \neq$ **wrong**

  Suppose $E; L \vdash x(p_1, \ldots, p_k) : B$. The type rules assert that there exist (1) B_i such that $E; L \vdash p_i : B_i$ (for $1 \leq i \leq k$) and (2) a non-terminal $x : S \in E$ such that $S = (x_1{:}B_1, \ldots, x_k{:}B_k)B$. Applying the induction hypothesis to (1) and $g; M \vdash \langle s_{j-1}, i_{j-1} \rangle\, p_j \Rightarrow \langle s_j, i_j \rangle\, t_j$, we have $t_j : B_j$.

  Suppose $\oslash \vdash g :: E$. This implies that x as defined in (1) is unique in E. Furthermore, suppose that $\oslash \vdash g$ **ok**. This implies that (3) $E; \oslash, x_1 : B_1, \ldots, x_k : B_k \vdash p : B$. Note that $M' \equiv \oslash, x_1 = t_1, \ldots, x_k = t_k \models \oslash, x_1 : B_1, \ldots, x_k : B_k$. Applying the induction hypothesis to (3) and $g; M' \vdash \langle s_k \rangle\, p \Rightarrow \langle s', i' \rangle\, t$, we finally have $t : B$.

- $g; M \vdash \langle s_0, i_0 \rangle\, x(p_1, \ldots, p_k) \Rightarrow \langle s', i' \rangle\,$**wrong** and $(x : (x_1{:}B_1, \ldots, x_k{:}B_k)B{=}{=}p) \notin g$

  Suppose $E; L \vdash x(p_1, \ldots, p_k) : B$ to obtain a contradiction. The type rules assert that there exists a non-terminal $x : S \in E$ such that $S = (x_1{:}B_1, \ldots, x_k{:}B_k)B$. Furthermore, suppose that $\oslash \vdash g :: E$. This implies together with $x : S \in E$ that there is a non-terminal definition $x : S{=}{=}p \in g$ contradicting our initial assumption about the derivation. $\Box$

Proof of Theorem 2

The proof is performed by induction on the parsing derivations for patterns with the rules in figure 4. We treat each rule in turn:

- $g; L_1; R \vdash \langle s \rangle\,$**unit** $\Rightarrow \langle s \rangle\,$**unit**

  Suppose $E; L_3 \vdash$ **unit** $: B$. Then $B =$ Unit and $E; L_1, L_2 \vdash$ **unit** $: B$.

- $g; L_1; R \vdash \langle x :: s \rangle\, "x" \Rightarrow \langle s \rangle\,$**unit**

  Suppose $E; L_3 \vdash "x" : B$. Then $B =$ Unit and $E; L_1, L_2 \vdash$ **unit** $: B$.

- $g; L_1; R \vdash \langle x :: s \rangle\,$**ide**(Var) $\Rightarrow \langle s \rangle\,$**global**$(x)$ and $x \notin K(g), x \notin Dom(L_1)$

  Suppose $E; L_3 \vdash$ **ide**(Var) $: B$. Then $B =$ Var and $E; L_1, L_2 \vdash$ **global**(x) : Var.

- $g; L, x : B, L'; R \vdash \langle x :: s \rangle\, \mathbf{ide}(B) \;\Rightarrow \langle s \rangle\, x$ and $\quad x \notin K(g), x \notin Dom(L')$

  Suppose $R = \oslash, x_1 \leftarrow x'_1, \ldots, x_n \leftarrow x'_n$, $Ran(R) \cap Dom(L, x : B, L') = \{\}$, $L_3 = \oslash, x_1 : B_1, \ldots, x_n : B_n$, and $L_2 = \oslash, x'_1 : B_1, \ldots, x'_n : B_n$. Furthermore, suppose $E; L_3 \vdash \mathbf{ide}(B) : B$. Since $x \notin Ran(R) = Dom(L_2)$ it follows that $E; L, x : B, L', L_2 \vdash x : B$.

- $g; L_1; R \vdash \langle s \rangle\, \mathbf{local} \;\Rightarrow \langle s \rangle\, \mathbf{local}$

  Suppose $E; L_3 \vdash \mathbf{local} : B$. Then $B =$ Binder and $E; L, L_2 \vdash \mathbf{local} :$ Binder.

- $g; L_1; R \vdash \langle s \rangle\, \mathbf{global}(x) \;\Rightarrow \langle s \rangle\, \mathbf{global}(x)$

  Suppose $E; L_3 \vdash \mathbf{global} : B$. Then $B =$ Var and $E; L_1, L_2 \vdash \mathbf{global} :$ Var.

- $g; L_1; R, x \leftarrow x', R' \vdash \langle s \rangle\, x \;\Rightarrow \langle s \rangle\, x' \quad x \notin Dom(R')$

  Suppose $L_3 = L'_3, x : B, L''_3$ and $E; L_3 \vdash x : B$ and $L_2 = L'_2, x' : B, L''_2$. Since $x'_i \neq x'_j$ in L_2 it follows that $x' \notin Dom(R'')$. Hence $E; L_1, L'_2, x' : B, L''_2 \vdash x' : B$.

- $g; L_1; R \vdash \langle s \rangle\, p_1 \; p_2 \;\Rightarrow \langle s'' \rangle\, p'_1 \; p'_2$

  Suppose $E; L_3 \vdash p_1 \; p_2 : B'$, that is, $E; L_3 \vdash p_1 : B$ and $E; L_3 \vdash p_2 : B'$. We know that $g; L_1; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p'_1$ and $g; L_1; R \vdash \langle s' \rangle\, p_2 \;\Rightarrow \langle s'' \rangle\, p'_2$. Applying the induction hypothesis gives $E; L_1, L_2 \vdash p'_1 : B$ and $E; L_1, L_2 \vdash p'_2 : B'$. Hence via the type rules $E; L_1, L_2 \vdash p'_1 \; p'_2 : B'$.

- $g; L_1; R \vdash \langle s \rangle\, x = p_1 \; p_2 \;\Rightarrow \langle s'' \rangle\, x' = p'_1 \; p'_2$

  Suppose $E; L_3 \vdash x = p_1 \; p_2 : B'$, that is, (1) $E; L_3 \vdash p_1 : B$ and (2) $E; L_3, x : B \vdash p_2 : B'$. Using the induction hypothesis, (1) and $g; L_1; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p'_1$ establish (3) $E; L_1, L_2 \vdash p'_1 : B$. Since the environments $R' = R, x : B$ and $L'_3 = L_3, x : B$ and $L'_2 = L_2, x' : B$ satisfy $Ran(R) \cap L_1 = \{\}$ and $x' \notin Dom(L_1) \cup Ran(R)$, we can apply the induction hypothesis to $g; L_1; R, x \leftarrow x' \vdash \langle s' \rangle\, p_2 \;\Rightarrow \langle s'' \rangle\, p'_2$ and (2) giving $E; L_1, L_2, x' : B \vdash p'_2 : B'$. Using (3) the type rules establish $E; L_1, L_2 \vdash p_1 \; p'_2 : B'$.

- $g; L_1; R \vdash \langle s \rangle\, p_1 \mid p_2 \;\Rightarrow \langle s' \rangle\, p'_1$ and $g; L_1; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p'_1$

  Suppose $E; L_3 \vdash p_1 \mid p_2 : B$. Because of the type rules $E; L_3 \vdash p_1 : B$. Using the induction hypothesis for $g; L_1; R \vdash \langle s \rangle\, p_1 \;\Rightarrow \langle s' \rangle\, p'_1$ we obtain $E; L_1, L_2 \vdash p'_1 : B$.

- $g; L_1; R \vdash \langle s \rangle\, p_1 \mid p_2 \;\Rightarrow \langle s' \rangle\, p'_2$ and $g; L_1; R \vdash \langle s \rangle\, p_2 \;\Rightarrow \langle s' \rangle\, p'_2$

  Analogous to the previous case.

- $g; L_1; R \vdash \langle s_0 \rangle\, c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) \Rightarrow \langle s_k \rangle\, c_{(B_1,\ldots,B_k)B}(p'_1, \ldots, p'_k)$

  Suppose $E; L_3 \vdash c_{(B_1,\ldots,B_k)B}(p_1, \ldots, p_k) : B$. Because of the type rules $E; L_3 \vdash p_j : B_j$ for $1 \leq j \leq k$. Together with $g; L_1; R \vdash \langle s_{j-1} \rangle\, p_j \Rightarrow \langle s_j, \rangle\, p'_j$ we can apply the induction hypothesis and obtain $E; L_1, L_2 \vdash p_j : B_j$ and thereby $E; L_1, L_2 \vdash c_{(B_1,\ldots,B_k)B}(p'_1, \ldots, p'_k) : B$.

- $g; L_1; R \vdash \langle s_0 \rangle\, x(p_1, \ldots, p_k) \Rightarrow \langle s' \rangle\, x'_1 = p'_1 \ \ldots\ x'_k = p'_k\ p'$

  We know that

  1. $g; L_1; R \vdash \langle s_{j-1} \rangle\, p_j \Rightarrow \langle s_j, \rangle\, p'_j$ for $\quad 1 \leq j \leq k$
  2. $R' = R, x_1 \leftarrow x'_1, \ldots, x_k \leftarrow x'_k \quad x'_i \notin Dom(L_1) \cup Ran(R) \quad x'_i \neq x'_j$ for $i \neq j$
  3. $g; L_1; R' \vdash \langle s_k \rangle\, p \Rightarrow \langle s' \rangle\, p'$
  4. $(x : (x_1{:}B_1, \ldots, x_k{:}B_k)B{=}{=}p) \in g$

  Suppose $E; L_3 \vdash x(p_1, \ldots, p_k) : B$. From the type rules it follows that $E; L_3 \vdash p_j : B_j$ for $1 \leq j \leq k$. We can apply the induction hypothesis to (1) and obtain $E; L_1, L_2 \vdash p'_j : B_j$. This judgment still holds if we insert additional fresh identifiers $x'_i \notin Dom(L_1) \cup Ran(R)$ into the environment, that is, $E; L_1, L_2, x'_1 : B_1, \ldots, x'_{j-1} : B_{j-1} \vdash p'_j : B_j$. Assume for now that $E; L_1, L_2, x'_1 : B_1, \ldots, x'_k : B_k \vdash p' : B$. This allows us to apply the type rule for pattern variable bindings k times and we obtain $E; L_1, L_2 \vdash x'_1 = p'_1 \ \ldots\ x'_k = p'_k\ p' : B$.

  Now we prove that indeed $E; L_1, L_2, x'_1 : B_1, \ldots, x'_k : B_k \vdash p' : B$. Since $\vdash g \Rightarrow E$, we have $\oslash \vdash g :: E$ and $\oslash \vdash g$ **ok**. Together with (4) this establishes $E; \oslash, x_1 : B_1, \ldots, x_k : B_k \vdash p : B$, and hence $E; L_3, x_1 : B_1, \ldots, x_k : B_k \vdash p : B$. Applying the induction hypothesis to (3) we get $E; L_1, L_2, x'_1 : B_1, \ldots, x'_k : B_k \vdash p' : B$. $\square$

References

[ASU87] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 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 *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, 1991.

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

[Chr90] H. Christiansen. A Survey of Adaptable Grammars. *ACM SIGPLAN Notices*, 25(11):25–44, 1990.

[Dow90] G. Dowek. Naming and Scoping in a Mathematical Vernacular. Rapport de Recherche 1283, INRIA, Rocquencourt, 1990.

[Gal74] B. Galler. Extensible Languages. *Information Processing*, pages 313–316, 1974.

[Gri88] T. Griffin. Notational Definition—A Formal Account. In *Proceedings of the Third Annual 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, 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 *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, 1991.

[Naq89] S.A. Naqvi. Stratification as a Design Principle in Logical Query Languages. In *Proceedings of the Second International Workshop on Database Programming Languages*, 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*, 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, 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. Morgan Kaufmann Publishers, 1991.

[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*. Springer-Verlag, 1991.

[SSF92] D. Stemple, T. Sheard, and L. Fegaras. Linguistic Reflection: A Bridge from Programming to Database Languages. In *Proceedings 25th Annual Hawaii International Conference on System Sciences*, 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, University of St. Andrews, Department of Computing Science, 1992.

[Sta75] T. A. Standish. Extensibility in Language Design. *ACM SIGPLAN Notices*, 10(7):18–21, 1975.

[Tri91] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *Database Programming Languages: Bulk Types and Persistent Data*. Morgan Kaufmann Publishers, 1991.

[WC93] D. Weise and R. Crew. Programmable Syntax Macros. *ACM SIGPLAN Notices*, 28(6):156–165, 1993.

[WG85] W.M. Waite and G. Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer-Verlag, 1985.