# Meta-Programming with Concrete Object Syntax

Eelco Visser

www.stratego-language.org

Technical Report UU-CS-2002-028
Institute of Information and Computing Sciences
Utrecht University

June 2002

This technical report is a preprint of:
E. Visser. Meta-Programming with Concrete Object Syntax. To appear in D. Batory and Ch. Consel (editors) *Generative Programming and Component Engineering (GPCE'02)*. Lecture Notes in Computer Science. Springer-Verlag, Pittsburgh, Pensylvania, October 2002. (© Springer-Verlag)

Copyright © 2002 Eelco Visser

Address:
Institute of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
email: visser@acm.org
`http://www.cs.uu.nl/~visser/`

# Meta-Programming with Concrete Object Syntax

Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. http://www.cs.uu.nl/∼visser, visser@acm.org

**Abstract.** Meta programs manipulate structured representations, i.e., abstract syntax trees, of programs. The conceptual distance between the concrete syntax meta-programmers use to reason about programs and the notation for abstract syntax manipulation provided by general purpose (meta-) programming languages is too great for many applications. In this paper it is shown how the syntax definition formalism SDF can be employed to fit *any* meta-programming language with concrete syntax notation for composing and analyzing object programs. As a case study, the addition of concrete syntax to the program transformation language Stratego is presented. The approach is then generalized to arbitrary meta-languages.

## 1 Introduction

Meta-programs analyze, generate, and transform object programs. In this process object programs are structured data. It is common practice to use abstract syntax trees rather than the textual representation of programs [10]. Abstract syntax trees are represented using the data structuring facilities of the meta-language: records (structs) in imperative languages (C), objects in object-oriented languages (C++, Java), algebraic data types in functional languages (ML, Haskell), and terms in term rewriting systems (Stratego).

Such representations allow the full capabilities of the meta-language to be applied in the implementation of meta-programs. In particular, when working with high-level languages that support symbolic manipulation by means of pattern matching (e.g., ML, Haskell) it is easy to compose and decompose abstract syntax trees. For meta-programs such as compilers, programming with abstract syntax is adequate; only small fragments, i.e., a few constructors per pattern, are manipulated at a time. Often, object programs are reduced to a core language that only contains the essential constructs. The abstract syntax can then be used as an intermediate language, such that multiple languages can be expressed in it, and meta-programs can be reused for several source languages.

However, there are many applications of meta-programming in which the use of abstract syntax is not satisfactory since the conceptual distance between the concrete programs that we understand and the data structure access operations used for composition and decomposition of abstract syntax trees is too large.

This is evident in the case of record manipulation in C, where the construction and deconstruction of patterns of more than a couple of constructors becomes unreadable. But even in languages that support pattern matching on algebraic data types, the construction of large code fragments in a program generator can become painful. For example, even the following tiny program pattern is easier to read in the concrete variant on the left than the abstract variant on the right.

```
let ds
 in let var x ta := (es1)
     in es2 end
end
```

```
Let(ds,
    [Let([VarDec(x,ta,Seq(es1))],
         es2)])
```

While abstract syntax is manageable for fragments of this size (and sometimes even more concise!), it becomes unpleasant to use when larger fragments need to be specified.

Besides the problems of understandability and complexity, there are other reasons why the use of abstract syntax may be undesirable. Desugaring to a core language is not always possible. For example, in the renovation of legacy code the goal is to repair the bugs in a program, but leave it intact otherwise. This entails that a much larger abstract syntax needs to be dealt with. Another occasion that calls for the use of concrete syntax is the definition of transformation or generation rules by users (programmers) rather than by compiler writers (meta-programmers). For example, [18] describes the extension of Haskell with pragmas for domain-specific optimization in the form of rewrite rules on program expressions. Other application areas that require concrete syntax are application generation and structured document (XML) processing.

Hence, it is desirable to have a meta-programming language that lets us write object-program fragments in the concrete syntax of the object language. In general, we would like to write a meta-program in meta-language $M$ that manipulates a program in object language $O$, where $M$ and $O$ could be the same, but need not be in general. This requires the extension of the syntax of $M$ with the syntax of $O$ such that $O$ expressions are interpreted as data construction and analysis patterns. This problem is traditionally approached by extending $M$ with a quotation operator that lets the meta-programmer indicate object language fragments [1,17,23]. Antiquotation allows the use of meta-programming language constructs in these object language fragments to splice meta-computed object code into a fragment. If $M$ equals $O$ then the syntax extension is easy by just adding quote and antiquote operators to $M$. For example, MetaML [19], provides <...> for distinguishing a piece of object code and ~... to splice computed code into another piece of code. Several meta-languages, including ASF+SDF [16] and TXL [12], are designed around the idea of meta-programming with concrete object syntax, where the object language is user-definable.

Building a meta-language that supports a different object language is a difficult task with traditional parser technology. Building a meta-language that allows the user to define the object language to use and determine the syntax for quotation and antiquotation operators is even harder. Why is this a difficult

task? Traditional parser generators support only context-free grammars that are restricted to the LL or LR properties. Since these classes of grammars are not closed under composition, it is hard to extend one language with another. For example, combining two YACC grammars will usually lead to many conflicts, requiring editing the two grammars, probably extensively. The problem is compounded by the fact that lexical syntax is dealt with using a scanner, which operates separately from the parser. An embedded object language will almost certainly overlap at the lexical level with the host language (e.g., syntax of identifiers, literals, keywords). Thus, combining the two regular grammars will also require extensive editing.

The usual solution is to require a fixed syntax for quotation delimiters and parse the content of quotations in a separate pass. This requires quite some infrastructure and makes reporting syntax errors to the programmer difficult. The technology used to extend a meta-language is usually not portable to other languages.

In this paper we show how the syntax definition formalism SDF [14,20] can be employed to fit *any* existing (meta-) programming language with concrete syntax notation for object programs. *The approach does not require that either the meta-language or the object-language were designed with this application in mind.* Rather, the syntax definitions of the meta-language and object-language are combined as they are and wedded by providing appropriate injections from object language sorts into meta-language sorts. From the combined syntax definition a parser is generated that parses the entire meta-program including object code fragments, and thus reports syntactic errors immediately. An explosion algorithm that can be independent of the object language then maps embedded object code abstract syntax trees to appropriate meta-language representations. The approach is based on existing technology that is freely available and can be applied immediately.

We illustrate the approach by extending the strategic rewriting language Stratego [21,22] with concrete syntax. In Section 2 we motivate the need for meta-programming with concrete syntax by means of an example and contrast it to the use of abstract syntax. In Section 3 we show how meta-programming with concrete object syntax is implemented in and for Stratego. In Section 4 we outline a framework for extending a programming language to provide meta-programming with concrete syntax.

## 2 Abstract Syntax vs Concrete Syntax

In this section we motivate the need for concrete syntax in meta-programming by contrasting the use of concrete syntax with the traditional use of abstract syntax. As an example we consider a simple meta-program for instrumenting Tiger programs with tracing statements. Tiger [2] is an imperative language with nested function definitions and statements that can be used in expressions. We conclude the section with a discussion of the challenges posed to the meta-programming system by the use of concrete syntax.

### 2.1 Syntax Definition

A meta-programming system requires a syntax definition of the object language and a parser and pretty-printer that transform to and from the abstract syntax of the language used for internal representation. Figure 1 gives a condensed (i.e., incomplete) definition of the concrete and abstract syntax of Tiger. The concrete syntax is defined in the syntax definition formalism SDF [14,20]. An SDF production `sym1 ... symn -> sym` declares that an expression of sort `sym` can be constructed by the concatenation of expressions of sorts `sym1` to `symn`. SDF supports regular expression operators such as `{Exp ";"}*`, which denotes a list of `Exp`ressions *separated* by `;` semicolons. Furthermore, SDF integrates the definition of lexical and context-free syntax in one formalism. The formalism is modular so that (1) the syntax definition of a language can be divided into smaller (reusable) modules and (2) syntax definitions for separate languages can easily be combined. Since SDF definitions are declarative, rather than operational implementations of parsers, it is possible to generate other artifacts from syntax definitions such as pretty-printers [9] and signatures.

The abstract syntax is declared as a Stratego signature, which declares a term constructor for each language construct. The signature abstracts from syntactic details such as keywords and delimiters. Such a signature can be derived automatically from the syntax definition by using the `cons`tructor attributes as declarations of the constructor name for a language construct.

### 2.2 Example: Instrumenting Programs

Stratego [21,22] is a language for program transformation based on the paradigm of rewriting strategies. It supports the definition of basic transformations by means of rewrite rules. The application of such rules is controlled by programmable strategies.

Figure 2 shows the specification in Stratego of a transformation on Tiger programs that instruments each function `f` in a program such that it prints `f entry` on entry of the function and `f exit` at the exit. Functions are instrumented differently than procedures, since the body of a function is an expression statement and the return value is the value of the expression. It is not possible to just glue a print statement or function call at the end of the body. Therefore, a let expression is introduced, which introduces a temporary variable to which the body of the function is assigned. The rule `IntroducePrinters` generates code for the functions `enterfun` and `exitfun`, calls to which are added to functions and procedures. The transformation strategy `instrument` uses the generic traversal strategy `topdown` to apply the `TraceProcedure` and `TraceFunction` rules to all function definitions in a program, after which the printer functions are added, thus making sure that these functions are not instrumented themselves.

The top part of the figure shows the specification in concrete syntax while the bottom part shows the same specification using abstract syntax. For brevity, the `IntroducePrinters` rule is only shown in concrete syntax. Note that the full lexical syntax for identifiers and literals of the object language is used.

```
module Tiger-Condensed
exports
  sorts Exp
  context-free syntax
    Id                                  -> Var {cons("Var")}
    StrConst                            -> Exp {cons("String")}
    Var                                 -> Exp
    "(" {Exp ";"}* ")"                  -> Exp {cons("Seq")}
    Var "(" {Exp ","}* ")"              -> Exp {cons("Call")}
    Exp "+" Exp                         -> Exp {left,cons("Plus")}
    Exp "-" Exp                         -> Exp {left,cons("Minus")}
    Var ":=" Exp                        -> Exp {cons("Assign")}
    "if" Exp "then" Exp "else" Exp      -> Exp {cons("If")}
    "let" Dec* "in" {Exp ";"}* "end"    -> Exp {cons("Let")}
    "var" Id TypeAn ":=" Exp            -> Dec {cons("VarDec")}
    FunDec+                             -> Dec {cons("FunctionDec")}
    "function" Id "(" {FArg ","}* ")"
                      TypeAn "=" Exp -> FunDec {cons("FunDec")}
    Id TypeAn                           -> FArg {cons("FArg")}
                                        -> TypeAn {cons("NoTp")}
    ":" TypeId                          -> TypeAn {cons("Tp")}
```

```
module Tiger-Condensed
signature
  constructors
    Var         : Id -> Var
    String      : StrConst -> Exp
    Seq         : List(Exp) -> Exp
    Call        : Var * List(Exp) -> Exp
    Plus        : Exp * Exp -> Exp
    Minus       : Exp * Exp -> Exp
    Assign      : Var * Exp -> Exp
    If          : Exp * Exp * Exp -> Exp
    Let         : List(Dec) * List(Exp) -> Exp
    VarDec      : Id * TypeAn * Exp -> Dec
    FunctionDec : List(FunDec) -> Dec
    FunDec      : Id * List(FArg) * TypeAn * Exp -> FunDec
    FArg        : Id * TypeAn -> FArg
    NoTp        : TypeAn
    Tp          : TypeId -> TypeAn
```

**Fig. 1.** Concrete syntax definition in SDF (top) and corresponding abstract syntax signature in Stratego (bottom) of Tiger programs (condensed).

### 2.3 Concrete vs Abstract

The example gives rise to several observations. The concrete syntax version can be read without knowledge of the abstract syntax. On the other hand, the abstract syntax version makes the tree structure of the expressions explicit. The abstract syntax version is much more verbose and is harder to read and write. Especially the definition of large code fragments such as in rule `IntroducePrinters` is unattractive in abstract syntax.

The abstract syntax version *implements* the concrete syntax version. The concrete syntax version has all properties of the abstract syntax version: pattern matching, term structure, can be traversed, and so on. In short, the concrete syntax is just sugar for the abstract syntax.

**Extension of the Meta-Language** We do not want to use Stratego only for meta-programming Tiger. Rather we would like to be able to handle arbitrary object languages. Thus, the object language or object languages that are used in a module should be a parameter to the compiler. The specification of instrumentation is based on the real syntax of Tiger, not on some combinators or infix expressions. This entails that the syntax of Stratego should be extended with the syntax of Tiger.

**Meta-Variables** The patterns in the transformation rules are not just fragments of Tiger programs. Rather some elements of these fragments are considered as meta-variables. For example in the term `[[ function f(xs) = e ]]` the identifiers `f`, `xs`, and `e` are not intended to be Tiger variables, but rather meta-variables, i.e., variables at the level of the Stratego specification.

**Antiquotation** Instead of indicating meta-variables implicitly we could opt for an antiquotation mechanism that lets us splice in meta-level expressions into a concrete syntax fragment. For example, using `~` and `~*` as antiquotation operators, a variant of rule `TraceProcedure` becomes:

```
TraceProcedure :
  [[ function ~f(~* xs) = ~e ]] ->
  [[ function ~f(~* xs) =
       (print(~String(<conc-strings>(f," entry\\n")));
        ~e;
        print(~String(<conc-strings>(f," exit\\n")))) ]]
```

With such antiquotation operators it becomes possible to directly embed meta-level computations that produce a piece of code within a syntax fragment.

## 3 Implementation

In the previous section we have seen how the extension of Stratego with concrete syntax for terms improves the readability of meta-programs. In this section we describe the techniques used to achieve this extension.

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    [[ function f(xs) = e ]] ->
    [[ function f(xs) = (enterfun(s); e; exitfun(s)) ]]
    where !f => s
  TraceFunction :
    [[ function f(xs) : tid = e ]] ->
    [[ function f(xs) : tid =
         (enterfun(s);
          let var x : tid := nil in x := e; exitfun(s); x end) ]]
    where new => x ; !f => s
  IntroducePrinters :
    e -> [[ let var ind := 0
                function enterfun(name : string) = (
                  ind := +(ind, 1);
                  for i := 2 to ind do print(" ");
                  print(name); print(" entry\\n"))
                function exitfun(name : string) = (
                  for i := 2 to ind do print(" ");
                  ind := -(ind, 1);
                  print(name); print(" exit\\n"))
             in e end ]]
```

```
module Tiger-TraceAll
imports Tiger-Typed lib Tiger-Simplify
strategies
  instrument = topdown(try(TraceProcedure + TraceFunction));
               IntroducePrinters; simplify
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(tid), e) ->
    FunDec(f, xs, Tp(tid),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(tid),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
  IntroducePrinters :
    e -> /* omitted for brevity */
```

**Fig. 2.** Instrumenting functions for tracing using concrete syntax and using abstract syntax.

### 3.1 Extending the Meta-Language

To embed the syntax of an object language in the meta-language, the syntax definitions of the two languages should be combined and the object language sorts should be injected into the appropriate meta-language sorts. In the Stratego setting this is achieved as follows. The syntax of a Stratego module `M` is declared in the `M.syn` file, which declares the name of an SDF module. The SDF module combines the syntax of Stratego and the syntax of the object language(s) by importing the appropriate SDF modules. The syntax definition of Stratego is provided by the compiler. The syntax definitions of the object language(s) are provided by the user. For example, Figure 3 shows a fragment of the syntax of Stratego and Figure 4 presents SDF module `StrategoTiger`, which defines the extension of Stratego with Tiger as object language. The module illustrates several remarkable aspects of the embedding of object languages in meta-languages using SDF.

A combined syntax definition is created by just importing appropriate syntax definitions. This is possible since SDF is a modular syntax definition formalism. This is a rather unique feature of SDF and essential to this kind of language extension. Since only the full class of context-free grammars, and not any of its subclasses such as LL or LR, are closed under composition, modularity of syntax definitions requires support from a generalized parsing technique. SDF2 employs scannerless generalized-LR parsing [20,8].

The syntax definitions for two languages may partially overlap, e.g., define the same sorts. SDF2 supports renaming of sorts to avoid name clashes and ambiguities resulting from them. In Figure 4 several sorts from the Stratego syntax definition (`Id`, `Var`, and `StrChar`) are renamed since the Tiger definition also defines these names.

The embedding of object language expressions in the meta-language is implemented by adding appropriate injections to the combined syntax definition. For example, the production

```
"[[" Exp "]]" -> Term {cons("ToTerm"),prefer}
```

declares that a Tiger expression (`Exp`) between `[[` and `]]` can be used everywhere where a Stratego `Term` can be used. Furthermore, abstract syntax expressions (including meta-level computations) can be spliced into concrete syntax

```
module Stratego
exports
  context-free syntax
    Int                             -> Term {cons("Int")}
    String                          -> Term {cons("Str")}
    Var                             -> Term {cons("Var")}
    Id "(" {Term ","}* ")"          -> Term {cons("Op")}
    Term "->" Term                  -> Rule {cons("RuleNoCond")}
    Term "->" Term "where" Strategy -> Rule {cons("Rule")}
```

**Fig. 3.** Fragment of the syntax of Stratego

```
module StrategoTiger
imports
  Tiger Tiger-Sugar Tiger-Variables Tiger-Congruences
imports
  Stratego [ Id => StrategoId
             Var => StrategoVar
             StrChar => StrategoStrChar ]
exports
  context-free syntax
    "[[" Dec       "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" FunDec    "]]"  -> Term        {cons("ToTerm"),prefer}
    "[[" Exp       "]]"  -> Term        {cons("ToTerm"),prefer}
    "~"  Term            -> Exp         {cons("FromTerm"),prefer}
    "~*" Term            -> {Exp ","}+  {cons("FromTerm")}
    "~*" Term            -> {Exp ";"}+  {cons("FromTerm")}
    "~"  Term            -> Id          {cons("FromTerm")}
    "~*" Term            -> {FArg ","}+ {cons("FromTerm")}
```

**Fig. 4.** Combination of syntax definitions of Stratego and Tiger

expressions using the ~ splice operators. To distinguish a term that should be interpreted as a list from a term that should be interpreted as a list *element*, the convention is to use a `~*` operator for splicing a list.

The declaration of these injections can be automated by generating an appropriate production for each sort as a transformation on the SDF definition of the object language. It is, however, useful that the embedding can be programmed by the meta-programmer to have full control over the selection of the sorts to be injected, and the syntax used for the injections.

### 3.2 Meta-Variables

Using the injection of meta-language `Term`s into object language `Exp`ressions it is possible to distinguish meta-variables from object language identifiers. Thus, in the term `[[ var ~x := ~e]]`, the expressions `~x` and `~e` indicate meta-level terms, and hence `x` and `e` are meta-level variables. However, it is attractive to write object patterns with as few squiggles as possible. This can be achieved using SDF variable declarations. Figure 5 declares syntax schemata for meta-variables. According to this declaration `x`, `y`, and `g10` are meta-variables for identifiers and `e`, `e1`, and `e1023` are meta-variables of sort `Exp`. The prefer attribute ensures that these identifiers are preferred over normal Tiger identifiers [8].

### 3.3 Meta-Explode

Parsing a module according to the combined syntax and mapping the parse tree to abstract syntax results in an abstract syntax tree that contains a mixture of meta- and object language abstract syntax. Since the meta-language compiler only deals with meta-language abstract syntax, the embedded object language

```
module Tiger-Variables
exports
  variables
    [s][0-9]*      -> StrConst    {prefer}
    [xyzfgh][0-9]* -> Id          {prefer}
    [e][0-9]*      -> Exp         {prefer}
    "xs"[0-9]*     -> {FArg ","}+ {prefer}
    "ds"[0-9]*     -> Dec+        {prefer}
    "ta"[0-9]*     -> TypeAn      {prefer}
```

**Fig. 5.** Some variable schema declarations for Tiger sorts.

abstract syntax needs to be expressed in terms of meta abstract syntax. For example, parsing the following Stratego rule

```
[[ x := let ds in ~* es end ]] -> [[ let ds in x := (~* es) end ]]
```

with embedded Tiger expressions, results in the abstract syntax tree

```
Rule(ToTerm(Assign(Var(meta-var("x")),
                   Let(meta-var("ds"),FromTerm(Var("es"))))),
     ToTerm(Let(meta-var("ds"),
                [Assign(Var(meta-var("x")),
                        Seq(FromTerm(Var("es"))))])))
```

containing Tiger abstract syntax constructors (e.g., `Let`, `Var`, `Assign`) and meta-variables (`meta-var`). The transition from meta-language to object language is marked by the `ToTerm` constructor, while the transition from meta-language to object language is marked by the constructor `FromTerm`.

Such mixed abstract syntax trees can be normalized by 'exploding' all embedded abstract syntax to meta-language abstract syntax. Thus, the above tree should be exploded to the following pure Stratego abstract syntax:

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

Observe that in this explosion all embedded constructors have been translated to the form `Op(C,[t1,...,tn])`. For example, the Tiger 'variable' constructor `Var(_)` becomes `Op("Var",[_])`, while the Stratego meta-variable `Var("es")` remains untouched, and `meta-var`s become Stratego `Var`s. Also note how the list in the second argument of the second `Let` is exploded to a `Cons`/`Nil` list. The resulting term corresponds to the Stratego abstract syntax for the rule

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

written with abstract syntax notations for terms.

```
module meta-explode
imports lib Stratego
strategies
  meta-explode =
    alltd(?ToTerm(<trm-explode>) + ?ToStrategy(<str-explode>))

  trm-explode =
    TrmMetaVar <+ TrmStr <+ TrmFromTerm <+ TrmFromStr <+ TrmAnno
    <+ TrmConc <+ TrmNil <+ TrmCons <+ TrmOp

  TrmOp       : op#(ts) -> Op(op, <map(trm-explode)> ts)

  TrmMetaVar  : meta-var(x) -> Var(x)
  TrmStr      = is-string; !Str(<id>)
  TrmFromTerm = ?FromTerm(<meta-explode>)
  TrmFromStr  = ?FromStrategy(<meta-explode>)
  TrmAnno     = Anno(trm-explode, meta-explode)
  TrmNil      : [] -> Op("Nil", [])
  TrmCons     : [x | xs] -> Op("Cons",[<trm-explode>x, <trm-explode>xs])
  TrmConc     : Conc(ts1,ts2) ->
                <foldr(!<trm-explode> ts2,
                       !Op("Cons", [<Fst>, <Snd>]), trm-explode)> ts1
```

**Fig. 6.** Generic definition of meta-explode

The explosion of embedded abstract syntax does not depend on the object language, but can be expressed generically, provided that embeddings are indicated with the `FromTerm` constructor. The *complete* implementation of the `meta-explode` transformation on `Term` abstract syntax trees is presented in Figure 6. The strategy `meta-explode` uses the generic strategy `alltd` to perform a generic traversal over the abstract syntax tree of a Stratego module. Anywhere in this tree where it finds a `ToTerm(_)`, its argument is exploded using `trm-explode`. This latter strategy is composed from a number of rules that recognize special cases. The general case is handled by `TrmOp`, which decomposes a term into its constructor `op` and arguments `ts`, and constructs an abstract syntax term `Op(op,ts')`, where the `ts'` are the exploded arguments. The transformation `str-explode` is similar to `trm-explode`, but transforms embedded abstract syntax into strategy expressions.

## 4 Generalization

In the previous section we described the embedding of concrete syntax for object languages in Stratego. This approach can be generalized to other meta-languages. In this section we outline the ingredients needed to make your favorite language into a meta-language.

Given a (general-purpose) language $M$ to be used as meta-language and a language $O$, which may be a data format, a programming language, or a

domain-specific language, as long as it has a formal syntax, we can extend $M$ to a meta-language for manipulating $O$ programs. Figure 7 depicts the architecture of this extension and the components that are employed. The large box denotes the extension of the $M$ compiler `m-compile` with concrete syntax for $O$. From a meta-programmer's point of view this is a black box that implements the compiler (dashed arrow) `mo-compile`, which consumes source meta-programs and produces executable meta-programs. In the rest of this section we briefly discuss the components involved.

*ATerm Library*

- The communication between the various components is achieved by exchanging ATerms [6], a generic format for exchange of structured data.

*SDF tools*

- `pack-sdf`: collection of all imported SDF modules;
- `sdf2table`: parser generator for SDF;
- `sglr`: scannerless generalized-LR parser reads a parse table (`M-O.tbl`) and parses a source file according to it;
- `implode-asfix`: translation from parse trees to abstract syntax trees;
- Optionally one can use pretty-printer and signature generators.

*$M$ as meta-language*

- A syntax definition `M.sdf` of $M$
- A model for object program representation in $M$ (e.g., AST represented as term)
- An API for constructing and analyzing $O$ programs in $M$ (e.g., pattern instantiation and matching)
- `m-explode`: An explosion algorithm for transforming $O$ abstract syntax expressions into $M$ expressions. If the object language program representation is generic, i.e., does not depend on a specific $O$, this can be implemented generically, as was done for Stratego using `meta-explode`. This is a transformation on $M$ programs.

*$O$ as object language*

- A syntax definition `O.sdf` of $O$
- A combined syntax definition `M-O.sdf`, possibly resolving name clashes
- Meta-variable declarations for $O$
- Injection of $O$ expressions into $M$ expressions
  - Selection of $O$ syntactic categories to manipulate (e.g., `Exp` and `Dec`)
  - Selection of $M$ syntactic categories in which $O$ expressions should be injected (e.g., `Term`)
  - Quotation syntax (e.g., `[[...]]`)
  - Anti quotation syntax (e.g., `~...`)

  It is possible to automate this by generating syntax for variables, quotations, and antiquotations automatically from the syntax definition of $O$, provided that there is a standard convention for quotation and antiquotation.

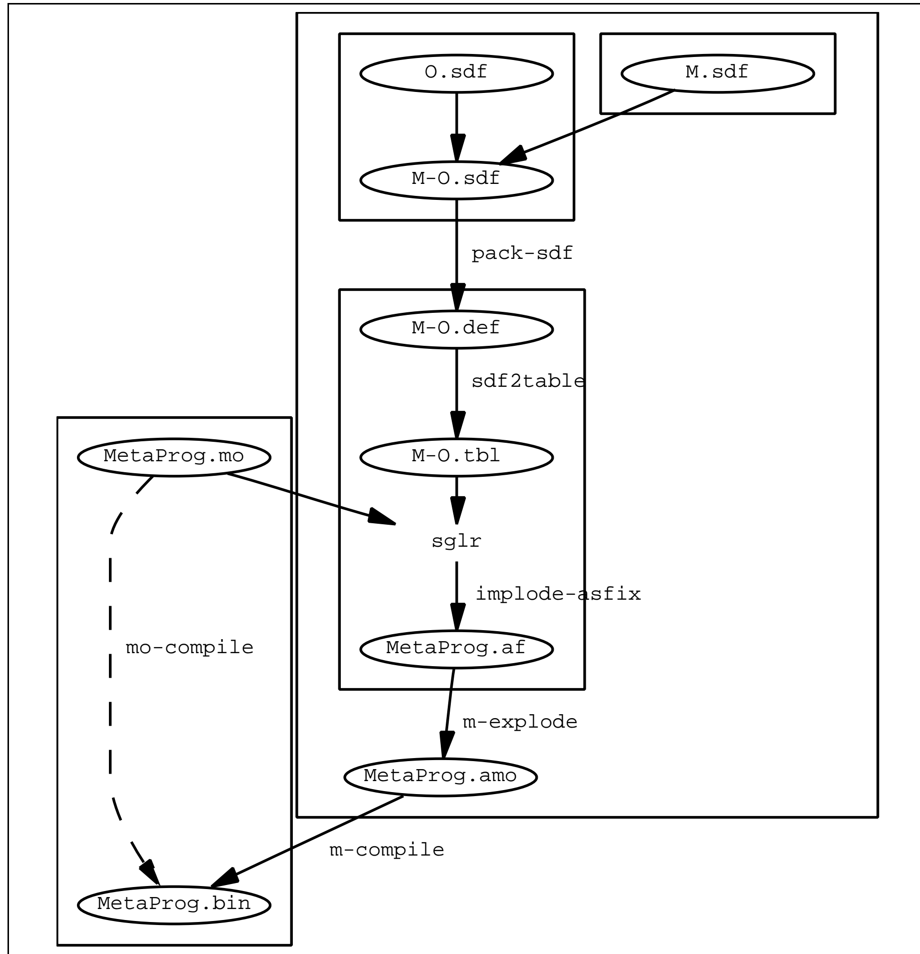


**Fig. 7.** Architecture for meta-programming with concrete object syntax

*M compiler*

– After `m-exploding` meta-programs they can be compiled by the usual compiler `m-compile` for $M$. If the compiler does not have an option to consume abstract syntax trees, but only text, it is necessary to pretty-print the program first.

*O meta-programs*

– Finally, we can write a meta-program `MetaProg.mo` using concrete syntax and compile it to an executable `MetaProg.bin` that manipulates $O$ programs.

# 5 Discussion

## 5.1 Syntax Definition and Parsing

SDF [14] was originally designed for use as a general syntax definition formalism. However, through its implementation it was closely tied to the algebraic specification formalism ASF+SDF [4,13], which is supported by the ASF+SDF Meta-Environment [16,7]. Redesign and reimplementation of SDF [20,8] has made SDF available for use outside the Meta-Environment. SDF is also distributed as part of the XT bundle of program transformation tools [15].

Syntax definition in SDF is limited to *context-free grammars*. This is a limitation for languages with context-sensitive syntax such as C (type identifiers) and Haskell (offside rule). However, in the setting of meta-programming with concrete object syntax, in which small fragments are used and not all context is always available, any parsing technique will have a hard time. This type of problem made Cameron and Ito [10] suggest that language designers should consider meta-programming in designing the syntax of a programming language.

## 5.2 Desugaring Patterns

Some meta-programs first desugar a program before transforming it further. This reduces the number of constructs and shapes a program can have. For example, the Tiger binary operators are desugared to prefix form:

```
DefTimes : [[ e1 * e2 ]]  -> [[ *(e1, e2) ]]
DefPlus  : [[ e1 + e2 ]]  -> [[ +(e1, e2) ]]
```

or in abstract syntax

```
DefPlus : Plus(e1, e2) -> BinOp(PLUS, e1, e2)
```

This makes it easy to write generic transformations for binary operators. However, all subsequent transformations on binary operators should then be done on these prefix forms, instead of on the usual infix form. This is the reason why in Figure 2, the expression `-(ind,1)` is used instead of `(ind - 1)`. However, users/meta-programmers think in terms of the infix operators and would like to write rules such as

```
Simplify : [[ e + 0 ]] -> [[ e ]]
```

However, this rule will not match since the term to which it is applied has been desugared. Thus, it might be desirable to desugar embedded abstract syntax with the same rules with which programs are desugared. This phenomenon occurs in many forms ranging from removing parentheses and generalizing binary operators as above, to decorating abstract syntax trees with information resulting from static analysis such as type checking.

### 5.3 User-definable Syntax

Programming languages with user-definable syntax have been a long standing goal of programming language research. To a certain extent programming languages do provide domain-specific or user-definable syntax. The use of infix syntax for arithmetic and logical operators is such a standard component of programming language syntax, that it is not considered a syntactic extension. However, they are clearly domain-specific operations, that could just as well be expressed using function call syntax. Indeed a number of languages (Prolog, Haskell, ...) allow the user to introduce new infix operators and define them just like a normal predicate or function. Other languages, especially in the domain of algebraic specification and theorem proving, have support for user-defined mixfix operators (e.g., OBJ, ELAN, Maude). This approach is taken to its extreme in the algebraic specification formalism ASF+SDF [4,13] in which all expression constructors are defined by the user, including the *lexical syntax*. An ASF+SDF specification consists of modules defining syntax and conditional equations over terms induced by this syntax. The equations are interpreted as term rewrite rules. The influence of ASF+SDF on the work described in this paper is profound—Stratego grew out of experience with ASF+SDF.

The architecture of JTS [3] is much like the one described in this paper, but the goal is the extension of languages with domain-specific constructs. The JTS tools are based on less powerful (i.e., lex/yacc) parsing technology.

Several experiments have been done with dynamically (parse-time) extensible syntax [23,11,5]. In these tools the program itself contains declarations of syntax extensions. This complicates the parsing process considerably. We have chosen to define the syntax in a separate file. This entails that the syntax for an entire module is fixed and cannot be extended half way. This is reasonable for meta-programming since the syntactic domain of meta-programs is usually a fixed object language or set of object languages. Changing the object language on a per module basis is fine grained enough.

### 5.4 Syntax Macros

The problem of concrete object syntax is different from extending a language with new constructs, for example, extending C with syntax for exception handling. This application known as syntax macros, syntax extensions, or extensible syntax [23,11,5] can be expressed using the same technology as discussed in this paper. Indeed, Stratego itself is an example of a language with syntactic extensions that are removed using transformations. For example, the following rules define several constructs in terms of the more primitive match (`?t`) and build (`!t`) constructs [22].

```
Desugar :
  [[ s => t ]] -> [[ s; ?t ]]
Desugar :
  [[ <s> t :S]] -> [[ !t; s ]]
Desugar:
  [[ f(as) : t1 -> t2 where s ]] -> [[ f(as) = ?t1; where(s); !t2 ]]
```

## 6 Conclusions

*Contribution* In this paper we have shown how concrete syntax notation can be fitted with minimal effort onto *any* meta-language, which need not be specifically designed for it. The use of concrete syntax makes meta-programs more readable than abstract syntax specifications. Due to the scannerless generalized parsing technology no squiggles are needed for antiquotation of meta-variables leading to very readable code fragments.

The technical contributions of this paper are the implementation of concrete syntax in Stratego and a general architecture for adding concrete object syntax to any (meta-)language. The application of SDF in Stratego is more evidence for the power of the SDF/SGLR technology. The composition of the SDF components with the Stratego compiler is a good example of component reuse.

*Future Work* The ability to fit concrete syntax onto a meta-programming language opens up a range of applications and research issues for exploration: transformation of various object languages such as Java and XML in Stratego; addition of concrete syntax to other meta-languages, which might involve mapping to a more distant syntax tree API than term matching and construction; object-language specific desugaring; and finally language extensions instead of meta-programming extensions.

*Availability* The components used in this project, including the ATerm library, SDF, and Stratego, are freely available and ready to be applied in other meta-programming projects. The SDF components are available from `http://www.cwi.nl/projects/MetaEnv/pgen/`. Bundles of the SDF components with other components such as Stratego are available from the Online Package Base at `http://www.program-transformation.org/package-base`

*Acknowledgments* Joost Visser provided comments on a previous version of this paper.

## References

1. A. Aasa. *User Defined Syntax*. PhD thesis, Dept. of Computer Sciences, Chalmers University of Technology and University of Göteborg, Göteborg, Sweden, 1992.
2. A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: A tool suite for building genvoca generators. In *5th International Conference in Software Reuse, (ICSR'98)*, Victoria, Canada, June 1998.
4. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
5. C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM'02*, 2002.
6. M. G. J. van den Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

7. M. G. J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-Environment: a component-based language laboratory. In R. Wilhelm, editor, *Compiler Construction (CC'01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–368, Genova, Italy, April 2001. Springer-Verlag.
8. M. G. J. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer-Verlag.
9. M. G. J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.
10. R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Trans. on Programming Languages and Systems*, 6(1):20–54, 1984.
11. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. SRC Research Report 121, Digital Systems Research Center, Palo Alto, California, February 1994.
12. J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Proc. IEEE 1988 Int. Conf. on Computer Languages*, pages 280–285, 1988.
13. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
14. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
15. M. de Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In M. G. J. van den Brand and D. Perigot, editors, *Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2001.
16. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
17. M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations in ML. In *Proc. 1994 Workshop on ML and its applications*, pages 70–78. Research report 2265, INRIA, 1994.
18. S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In R. Hinze, editor, *2001 Haskell Workshop*, Firenze, Italy, September 2001. ACM SIGPLAN.
19. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
20. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
21. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
22. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proc. of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.
23. D. Weise and R. F. Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI'93)*, Albuquerque, New Mexico, June 1993.