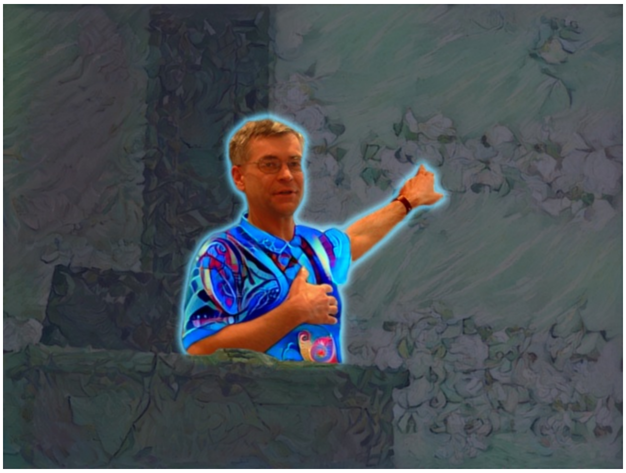
Chapter 12
A Suite of Metaprogramming Techniques



OLEG KISELYOV.[1]

Abstract Metaprogramming may be done with just a few programming techniques: an object-program representation (to capture the syntactical structure of object programs), pattern matching or accessors (to take apart object programs or to select suitable parts thereof), pattern building or constructors (to construct or compose object programs), and a computational model for tree walking (e.g., visitors in OO programming or possibly just recursion). In this chapter, we describe some metaprogramming techniques on the basis of which many metaprograms can be written in a more disciplined style. That is, we describe term rewriting, attribute grammars, multi-stage programming, partial evaluation, and abstract interpretation.

[1] Mastery of semantics-based techniques, type-system acrobatics, over-the-head functional programming – these labels pretty reliably map to Oleg Kiselyov without too much risk of hash-code collision. The photo shows him while he was talking about "typed final (tagless-final) style" [7, 34] (`http://okmij.org/ftp/tagless-final/`) – an advanced topic of metaprogramming not included in this book. One may wonder what a textbook would look like if Oleg was ever to write down a good part of his operational knowledge.

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_12

12.1 Term Rewriting

Term rewriting can be viewed as a computational paradigm for describing transformations as a collection of *rewrite rules* which match on object-program patterns and build new patterns from the matched parts in some way. Some implicit or explicit normalization strategy takes care of applying the rules, in some sense, exhaustively. A collection of rewrite rules together with a strategy for their application may be referred to as a *rewrite system*.

In theoretical computer science, rewrite systems are formal entities in themselves (just like grammars) and have been studied very well [16, 39, 17]. In practice, we are interested in metaprogramming systems with support for rewriting such as ASF+SDF [82], TXL [9, 10], Stratego [6], and Rascal [37, 36], and possibly in declarative programming languages that support some form of rewriting. We will exercise term rewriting in Haskell.

12.1.1 Rewrite Rules

As a running example, we will deal with optimization of expression forms; this example was introduced in a pragmatic metaprogramming manner in Section 5.4.1. Our objective here is to show that term rewriting provides a rigorous technique for describing such optimizations. Term rewriting boils down to the declaration and application of rewrite rules such as the following one:

$$(X * Y) + (X * Z) \leadsto X * (Y + Z)$$

This rule captures distributivity for multiplication and addition, as present in many languages. In the rest of this section, we exercise EL (*E*xpression *L*anguage), which is the language of expression forms that are common to the fabricated imperative and functional programming languages BIPL and BFPL in this book.

As a precursor to setting up a rewrite system, let us collect together several algebraic laws that we assume to hold for expressions. We use uppercase letters X, Y, Z, ... as metavariables for arbitrary expressions so that we can talk about patterns of expressions:

$X + 0$	$= X$	-- Unit of addition
$X * 1$	$= X$	-- Unit of multiplication
$X * 0$	$= 0$	-- Zero of multiplication
$X + Y$	$= Y + X$	-- Commutativity of addition
$X * Y$	$= Y * X$	-- Commutativity of multiplication
$(X + Y) + Z$	$= X + (Y + Z)$	-- Associativity of addition
$(X * Y) * Z$	$= X * (Y * Z)$	-- Associativity of multiplication
$(X * Y) + (X * Z)$	$= X * (Y + Z)$	-- Distributivity

We should think of applying these laws – from either left to right or from right to left – to appropriate subterms of a given term such as an EL expression or "bigger" program phrases such as a statement (in BIPL) or a function definition (in BFPL). In sample expressions, we use lowercase letters a, b, c, ... as program variables.

Here is a rewriting step applying the second equation:

$$a + \underline{b * 1} + c = a + \underline{b} + c$$

We have applied the equation from left to right. We have underlined the subexpressions which are instances of the left- and right-hand sides of the equation. We may also use the term "redex" to refer to subterms to which a rewrite rule (or an equation) is applied or applicable.

Exercise 12.1 (Additional rules for expressions) [Basic level]
Identify some additional algebraic laws for EL – specifically also some rules that involve operators that are not exercised by the laws stated above.

When equations are readily directed so that the direction of application is specified, then we speak of rules rather than equations. That is, we use an arrow "$\rightsquigarrow$" to separate left- and right-hand side, and rules are thus to be applied from left to right. For instance, the first law may reasonably be directed from left to right, as this direction would be useful in applying the rule for the purpose of *simplification*. In fact, the first three equations can be understood as simplification rules, when directed from left to right; in fact, we perform a transition from equations to rules:

$$\begin{array}{lll} X + 0 & \rightsquigarrow X & \text{-- Unit of addition} \\ X * 1 & \rightsquigarrow X & \text{-- Unit of multiplication} \\ X * 0 & \rightsquigarrow 0 & \text{-- Zero of multiplication} \end{array}$$

That is, a rewrite rule consists of a left- and a right-hand side; these are both patterns of object programs. The assumed semantics of applying a rewrite rule is that the left-hand side is matched with a given term, with the metavariables bound to subterms if matching succeeds; the result is constructed from the bound metavariables according to the right-hand side.

Exercise 12.2 (Semantics of term rewriting) [Intermediate level]
Specify the semantics of applying rewrite rules.

For now, let us use abstract syntax for expressions, as this makes it easy to implement rewrite systems in programming languages. In abstract syntax, the earlier simplification rules look as follows:

$$\begin{array}{lll} \mathit{binary}(\mathit{add}, X, \mathit{intconst}(0)) & \rightsquigarrow X & \text{-- Unit of addition} \\ \mathit{binary}(\mathit{mul}, X, \mathit{intconst}(1)) & \rightsquigarrow X & \text{-- Unit of multiplication} \\ \mathit{binary}(\mathit{mul}, X, \mathit{intconst}(0)) & \rightsquigarrow \mathit{intconst}(0) & \text{-- Zero of multiplication} \end{array}$$

The abstract syntax is defined as follows.

Illustration 12.1 (Abstract syntax of expressions)

ESL resource languages/EL/as.esl

```
// Expressions
symbol intconst : integer → expr ;
symbol boolconst : boolean → expr ;
symbol var : string → expr ;
symbol unary : uop × expr → expr ;
symbol binary : bop × expr × expr → expr ;

// Unary operators
symbol negate : → uop ;
symbol not : → uop ;

// Binary operators
symbol add : → bop ;
symbol sub : → bop ;
symbol mul : → bop ;
symbol lt : → bop ;
symbol le : → bop ;
symbol eq : → bop ;
symbol geq : → bop ;
symbol gt : → bop ;
symbol and : → bop ;
symbol or : → bop ;
```

12.1.2 Encoding Rewrite Rules

We may encode rewrite rules easily in Haskell, or in any other functional programming language for that matter. That is, rewrite rules become function equations. Functions are used for grouping rewrite rules. We need to be careful to define these functions in such a manner that function application will not throw an exception when the underlying rules are not applicable to a given term. Instead, failure should be communicated gracefully and, thus, we use the Maybe monad. In Section 5.4.1, we already encoded simplification rules in this manner, as we recall here:

```
simplify :: Expr → Maybe Expr
simplify (Binary Add x (IntConst 0)) = Just x
simplify (Binary Mul x (IntConst 1)) = Just x
simplify (Binary Mul x (IntConst 0)) = Just (IntConst 0)
simplify _ = Nothing
```

We may apply the Haskell-based rewrite rules as follows.

Interactive Haskell session:

```
▶ simplify (Binary Add (Var "a") (IntConst 0))
Just (Var "a")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (IntConst 42)
Nothing
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (Binary Add (Var "a") (Binary Add (Var "b") (IntConst 0)))
Nothing
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ simplify (Binary Add (IntConst 0) (Var "a"))
Nothing
```

The first application succeeds because the simplification rule for the unit of addition is applicable. The second application fails because no simplification rule applies to the expression at hand. Failure of application is modeled by returning *Nothing*. (In an alternative model, the input term could be returned as is if no rule is applicable.) The third application also fails despite the presence of a subexpression to which the simplification rule for the unit of addition would be applicable, but note that we apply simplify directly. We do not in any way descend into the argument to find redexes. Ultimately, we need "normalization", as we will discuss in a second. The fourth application also fails because the simplification rule for the unit of addition only checks for the unit on the right. We may need to combine simplification with the rules for commutativity somehow.

In Haskell, we may also write more versatile rewrite rules taking advantage of functional programming expressiveness. In the following examples, we use guards, extra parameters, and function composition in the "rewrite rules".

Illustration 12.2 (Additional rules illustrating the use of Haskell in rewriting)

Haskell module Language.EL.MoreRules

```
-- Cancel double negation on Ints
doubleNegate (Unary Negate (Unary Negate e)) = Just e
doubleNegate (Unary Negate (IntConst i)) | i <= 0 = Just (IntConst (−i))
doubleNegate _ = Nothing

-- Swap variable names
swap x y (Var z) | z == x = Just (Var y)
swap x y (Var z) | z == y = Just (Var x)
swap _ _ _ = Nothing

-- Compose simplification with optional commute
simplify' x = simplify x `mplus` commute x >>=simplify
```

That is, the doubleNegate function removes two patterns of double negation; the first pattern models double application of the negation operator, and the second pattern models application of the negation operator to a negative number. The swap function is parameterized by two variable names, x and y, and it replaces each occurrence of x by y and vice versa. The simplify' function builds a choice from the plain simplify function such that in the case of failure of simplify, the commutativity rules are applied prior to trying simplify again. Here we assume that we also have "directed" laws for commutativity; the actual direction does not matter in this case, obviously.

Illustration 12.3 (Commutativity for expressions)

Haskell module Language.EL.Rules.Commute

```
commute :: Expr → Maybe Expr
commute (Binary Add x y) = Just $ Binary Add y x
commute (Binary Mul x y) = Just $ Binary Mul y x
commute _ = Nothing
```

We may apply the commutativity-aware definition as follows:

Interactive Haskell session:

```
► simplify' (Binary Add (IntConst 0) (Var "a"))
Just (Var "a")
```

That is, this application succeeds and returns a simplified term, whereas the application of the original simplify function failed.

12.1.3 Normalization

Rewrite rules only model "steps" of rewriting. We need a normalization strategy atop so that rewrite rules are applied systematically (i.e., repeatedly and exhaustively in some sense). If we had a suitable function normalize, then we might be able to apply the simplify function in the following manner:

Interactive Haskell session:

```
► normalize simplify (Binary Add (Var "a") (Binary Add (Var "b") (IntConst 0)))
Binary Add (Var "a") (Var "b")
```

Thus,"$a + \underline{(b+0)}$" is simplified to "$a + \underline{b}$" when expressed in concrete syntax; we underline again the redex for clarity. However, there seem to be many possible behaviors for normalization, for example: (i) to apply rewrite rules in top-down or bottom-up manner; (ii) to aim at a single or an exhaustive application of the given rules; (iii) to succeed or fail in the case of no applicable rewrite rules; or (iv) to apply rewrite rules only to terms of suitable types or descend into terms to find subterms of suitable types.

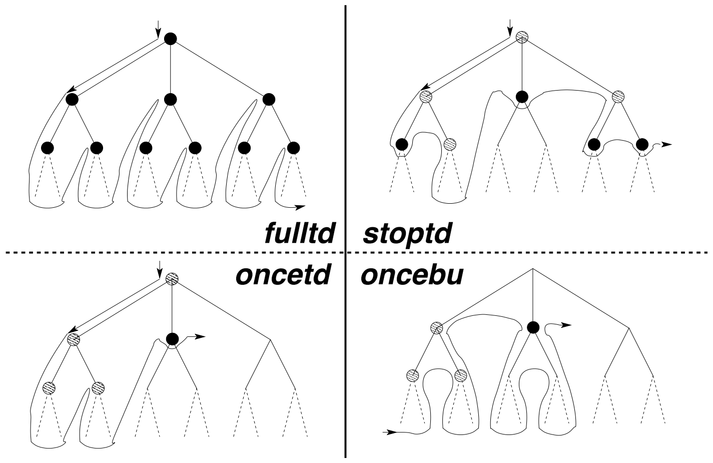


Fig. 12.1 Illustration of different traversal schemes. (Source: [50].) The illustration conveys which nodes are encountered during the traversal and whether the given strategy fails (see the gray nodes) or succeeds (see the black nodes).

Some rewriting approaches tend to favor one "built-in" normalization strategy so that rewrite rules are applied, in some sense, exhaustively [85]. One popular strategy is "innermost" which essentially attempts rules repeatedly in a bottom-up manner until no rule applications are feasible anymore. Other rewriting approaches permit programmers to define normalization strategies. This is the case for the style of (so-called) *strategic programming*, as discussed below. Without such flexibility, programmers end up controlling normalization by more complex rewrite rules.

12.1.4 Strategic Programming

Strategic programming is a discipline which enables the programmer to define and use strategies [86, 87, 50, 51, 52, 85, 6, 49] for applying (collections of) rewrite rules. A suite of reusable normalization strategies is provided to the programmer to choose from, and problem-specific strategies can be defined when necessary.

The notion of strategies is language-independent; it has been realized in several programming languages, for example, in Haskell [51, 52], Java [87], and Prolog [44], and it is available in different metaprogramming systems in one form or another, without necessarily being referred to as strategies; the notion was pioneered in Stratego/XT [86, 6].

Figure 12.1 illustrates a number of "strategic" traversal schemes. All of these schemes are applied to an argument strategy which may be a collection of rewrite rules or a more complex strategy. Let us explain these schemes informally and hint at applications:

fulltd The argument strategy is applied to all nodes in a top-down, depth-first manner; application needs to succeed for all nodes, otherwise the entire traversal fails. This scheme is used when a transformation should be applied "everywhere". The function *doubleNegate* in Illustration 12.2 could be applied in this manner; the scheme is suitable for finding and eliminating arbitrarily nested occurrences of double negation.

stoptd This scheme also models top-down, depth-first traversal, but traversal does not visit subtrees for nodes at which application succeeded. This scheme is used when either the existence of redexes below successful nodes can be ruled out to exist or rewrites may create redexes that must not be considered in the interests of termination, for example, when one is inlining recursive abstractions. The function swap in Illustration 12.2 could be applied in this manner. A *fulltd* traversal is not necessary, as redexes for renaming cannot occur inside variables identifier (i.e., strings or lists of characters).

oncebu The argument strategy is applied to all nodes in a bottom-up manner; traversal stops upon the first successful application. Focusing on one redex at a time is a testament to the overall assumption that a single traversal may be insufficient to find and eliminate all redexes, as rewrites may enable new rewrites. Thus, in general, a repeated application of given rewrite rules may be needed for the sake of completeness. We mentioned *innermost* before as a common normalization strategy; it can be defined by means of repeating *oncebu* until no more redexes are found in this manner.

oncetd This is just like **oncebu**, but traversal commences in a top-down manner.

Exercise 12.3 (Nonterminating traversal) [Intermediate level]
Describe a simple, concrete scenario for which a traversal based on **fulltd** *may fail to terminate.*

As an exercise in metaprogramming expressiveness, we would like to give precise definitions of these schemes. In fact, we would like to define the schemes as abstractions in Haskell, thereby revealing the expressiveness that may be needed when a strategic programmer wants to define yet other traversals or schemes for them. We need two special primitives for what we refer to as layer-by-layer traversal; see Fig. 12.2 for an illustration. Let us describe the traversal modeled by these primitives when applied to an argument strategy:

all The argument strategy is applied to all immediate subterms of a given term; in fact, all applications have to succeed, otherwise the *all* strategy fails. Thus, a (successful) *all* strategy essentially rewrites the immediate subterms (the "children") of a term.

one The argument strategy is applied to all immediate subterms (from left to right) until one application succeeds. If all applications fail, then the *one* strategy fails too. Thus, a (successful) *one* strategy essentially rewrites one immediate subterm (a "child") of a term.

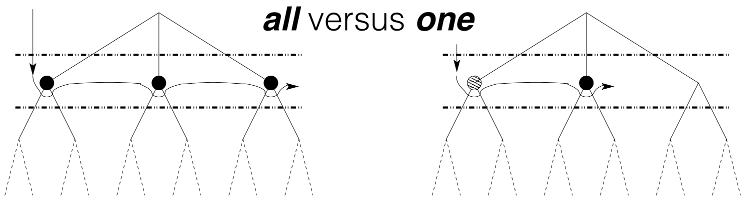


Fig. 12.2 Layer-by-layer traversal. (Source: [50].)

The following code sketch illustrates how *all* could be defined for expressions:

```
all s (IntConst i) = IntConst <$> s i
all s (BoolConst b) = BoolConst <$> s b
all s (Var v) = Var <$> s v
all s (Unary o e1) = Unary <$> s o <*> s e1
all s (Binary o e1 e2) = Binary <$> s o <*> s e1 <*> s e2
```

That is, the argument s (which is essentially a polymorphic function) is applied to all immediate subterms by combining the applications in the applicative functor style. (We could also use a monadic bind instead.) There is one case for every constructor.

In reality, *all* and *one* are generically defined or definable for all (at least most) Haskell types. For instance, in Haskell's "scrap your boilerplate" (SYB) approach to generic functional programming [45, 46, 47], suitable type-class instances are automatically derived. That is, the code shown would essentially be derived by a tool (such as a compiler).

We are ready to define the earlier traversal schemes as a Haskell library of function combinators. We also provide a few more basic combinators.

Illustration 12.4 (A small strategic programming library)

Haskell module Data.Generics.Strategies

```
-- Strategic traversal schemes
fulltd s = s `sequ` all (fulltd s)
fullbu s = all (fullbu s) `sequ` s
stoptd s = s `choice` all (stoptd s)
oncetd s = s `choice` one (oncetd s)
oncebu s = one (oncebu s) `choice` s
innermost s = repeat (oncebu s)

-- Basic strategy combinators
s1 `sequ` s2 = λ x → s1 x >>=s2 -- monadic function composition
s1 `choice` s2 = λ x → s1 x `mplus` s2 x -- monadic choice
all s = ... -- magically apply s to all immediate subterms
one s = ... -- magically find first immediate subterm for which s succeeds

-- Helper strategy combinators
```

```
try s = s `choice` return -- recover from failure
vary s v = s `choice` (v `sequ` s) -- preprocess term, if necessary
repeat s = try (s `sequ` repeat s) -- repeat strategy until failure

-- Strategy builders
orFail f = const mzero `extM` f -- fail for all other types
orSucceed f = return `extM` f' -- id for all other types
 where f' x = f x `mplus` return x -- id in case of failure
```

Thus, the traversal schemes are essentially defined as recursive functions in terms of sequential composition (sequ), left-biased choice, and the traversal primitives all and one. There are also function definitions for "strategy builders" which are needed to turn type-specific rewrite rules into generic functions. This transition is essential for one to be able to process terms of arbitrary types with subterms of different types – not all terms are of types of interest.

Let us illustrate the library in action:

Interactive Haskell session:

```
-- The expression "a + b * 0" with simplification potential
▶ let e1 = Binary Add (Var "a") (Binary Mul (Var "b") (IntConst 0))
-- The expression "((a * b) * c) * d" associated to the left
▶ let e2 = Binary Mul (Binary Mul (Binary Mul (Var "a") (Var "b")) (Var "c")) (Var "d")
-- The expression "0 + a" requiring commutativity for simplification
▶ let e3 = Binary Add (IntConst 0) (Var "a")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Incomplete simplification with fulltd
▶ fulltd (orSucceed simplify) e1
Binary Add (Var "a") (IntConst 0)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Complete simplification with fullbu
▶ fullbu (orSucceed simplify) e1
Var "a"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Incomplete association to the right with fullbu
▶ fullbu (orSucceed associate) e2
Binary Mul (Var "a") (Binary Mul (Binary Mul (Var "b") (Var "c")) (Var "d"))
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Complete association to the right with innermost
▶ innermost (orFail associate) e2
Binary Mul (Var "a") (Binary Mul (Var "b") (Binary Mul (Var "c") (Var "d")))
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Apply simplification module commutativity
▶ vary (orFail simplify) (orFail commute) e3
Var "a"
```

Exercise 12.4 (Applicability of innermost) [Basic level]
Consider again the swap *function of Illustration 12.2. Why would a traversal based on* innermost *not produce the correct result with all occurrences of the two variables consistently swapped?*

The present section is summarized by means of a recipe.

Recipe 12.1 (Design of a strategic program).

Test cases *Set up test cases for the strategic program, just like for any transformational program (Recipe 5.2). A positive test case consists of an input term and the expected output term. A negative test case consists of an input term and the expectation that the strategy fails.*
Rules *Implement the basic units of functionality, i.e., (rewrite) rules which match and build patterns of interest and possibly perform other computations along with matching and building.*
Groups *Group rules into logical units, for example, groups for simplification, normalization, desugaring, and other things. The groups may be specific to the problem at hand. For instance, there may be several groups of optimization rules, subject to separate phases.*
Strategy *Reuse (i.e., select) or define (i.e., compose) strategy combinators so that they can be applied to the appropriate groups of rules. The combinators may be concerned with traversal or other forms of "control" (e.g., order, alternatives, fixed-point computation).*
Testing *Test the composed strategy in terms of the test cases.*

12.1.5 Rewriting-Related concerns

12.1.5.1 Other Traversal Idioms

We mention in passing that we have limited ourselves here to *type-preserving* strategies. (We refer to "type" here in terms of the syntactic category of object programs being manipulated.) If we wanted to use rewriting or strategic programming to extract data using so-called *type-unifying* strategies or to perform any other kind of non-type-preserving operations, then we would need additional machinery, but we will not discuss this here. Traversal schemes may also need to maintain additional arguments in the sense of environments and states, so that information is passed down and updated along with traversal. There are also alternative models for combining traversal and rewriting. For instance, traversals may also be set up as walks subject to performing actions that descend into the children, proceed along the siblings, and return to the root [4].

12.1.5.2 Concrete Object Syntax

Rewriting on top of "large" syntaxes may, arguably, benefit from the use of concrete object syntax, as discussed earlier (Section 7.5), because a programmer may recognize object language patterns more easily. Several metaprogramming systems do indeed support concrete object syntax for this reason.

12.1.5.3 Graph Rewriting and Model Transformation

There is the related discipline of graph grammars and transformation [66, 23] – thus, rewriting may instead operate on graphs rather than terms (or trees). In model transformation [14, 56], one may operate on models ("graphs") that are instances of a metamodel with part-of, reference, and inheritance relationships. There exist dedicated model-transformation languages, for example, ATL [31]. These approaches also aim to eliminate boilerplate code for controlling the overall transformation process, including traversal. For instance, ATL provides a refining mode [80] so that transformation rules can be limited to the model elements that need to be replaced.

12.1.5.4 Origin Tracking

In term rewriting (or model transformation), traceability may be desirable in the sense that the "origin" of any given (sub-) term (or model element) can be traced back to some original term (or model element). This idea is captured in a fundamental manner by the notion of origin tracking [18, 88, 65]. For instance, if a semantic analysis was applied to an abstract or intermediate representation in a language implementation, then origin tracking helps in systematically relating back the results of the analysis (e.g., errors or warnings) to the original program. Origin tracking relies on deep support in a metaprogramming or model-transformation system.

12.1.5.5 Layout Preservation

When transforming object programs (by means of rewriting or otherwise), it may be desirable to retain the original layout (white space, line breaks, and even comments) in the programs to the extent possible. For instance, when one is performing a re-engineering transformation on legacy code, the code should be retained as much as possible so that programmers will still recognize their code. Such layout preservation [28, 41, 29] calls for a suitable object-program representation (CST or AST) which incorporates layout. Less obviously, a term-rewriting approach may need to manipulate object-program patterns in a special way so as to retain layout where possible, subject also to possibly incorporating an incremental formatter that applies to fragments without inherited layout or with invalidated layout.

12.2 Attribute Grammars

Attribute grammars (AGs) [40, 3, 60, 25] can be viewed as a computational paradigm for describing translations or analyses by means of adding attributes to nodes in a CST or AST. An AG combines a context-free grammar with computational rules. Each computational rule relates attributes of nonterminal symbols within the scope of a specific context-free rule. The order of computation (attribute evaluation) is not explicitly described, but it can be inferred from the attribute dependencies expressed by the computational rules. In Section 7.3.2, we discussed a limited form of an AG, i.e., grammars enhanced by semantic actions for AST construction to serve as input for a parser generator.

AGs are supported explicitly by some metaprogramming systems with dedicated AG languages, for example, Eli [32], JastAdd [25], Silver [83], or (Aspect) Lisa [60] and these systems support several AG extensions (see, e.g., [33, 48, 42]). The AG style of metaprogramming and computation can also be leveraged if a sufficiently powerful (declarative) metalanguage is used. In particular, AGs can be "encoded" in functional programming [75, 71], as we will show below. Furthermore, a limited form of an AG is also supported by mainstream parser generators such as ANTLR, which we discussed earlier. In this section, we introduce the notion of AGs as a means of approaching analysis and translation problems.

12.2.1 The Basic Attribute Grammar Formalism

We begin with a trivial problem to explain the basics of AGs. In Fig. 12.3, we show an attributed CST for a binary number in the sense of the ("fabricated") Binary Number Language (BNL). Decimal values for the relevant subtrees are shown next to the nodes. That is, the nodes for the individual bits carry attributes for values that take into account the position of each bit. The nodes for bit sequences carry attributes for values that arise as sums of values for subtrees. For instance, the decimal value for the bit sequence 101 is $5 = 4 + 0 + 1$. This trivial (illustrative) example is due to Knuth [40].

We may need additional (auxiliary) attributes to compute the actual decimal values. In one possible model, we may assign a position $(\dots, 2, 1, 0, -1, -2, \dots)$ to each bit and maintain the length of a bit sequence so as to be able to actually compute the value for any bit. Table 12.1 shows all attributes that we want to compute.

As the table clarifies, attributes are assigned to nonterminals. An attribute is classified as either inherited or synthesized. We use the classifiers *inherited* ("I") and

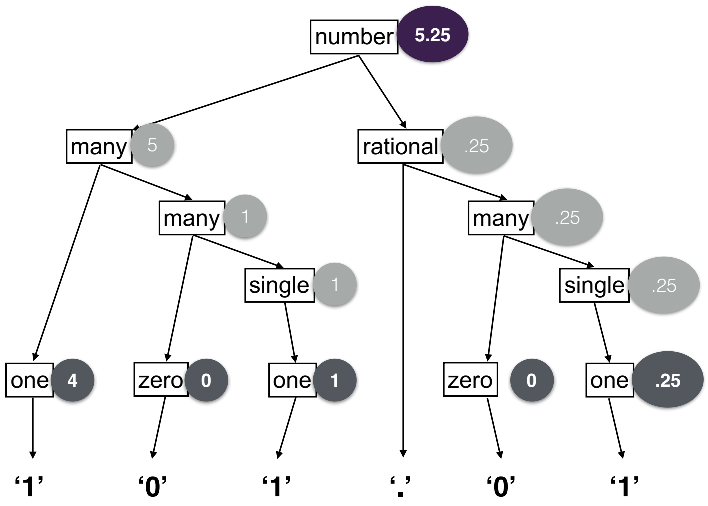


Fig. 12.3 An attributed syntax tree for the binary number 101.01. The attributes attached to the nodes model the decimal value of the subtree at hand.

Table 12.1 Attributes for binary to decimal number conversion

Nonterminal	**Attribute**	**I/S**	**Type**
number	*Val*	S	float
bits	*Val*	S	float
bit	*Val*	S	float
rest	*Val*	S	float
bits	*Pos*	I	integer
bit	*Pos*	I	integer
bits	*Len*	S	natural

synthesized ("S") to express that the attribute is to be passed down or up, respectively, in the tree. This classification has to do with attribute dependencies, as we will see in a second.

An AG associates a collection of computational rules with each context-free rule p. Each computational rule is of the following form:

$$x_0.a_0 = f(x_1.a_1, \ldots, x_m.a_m)$$

where $x_0, \ldots, x_m$ are nonterminals of the context-free rule, $a_0, \ldots, a_m$ are attributes of the nonterminals, and f is any sort of "operation" on the attributes. Conceptually, the computational rules state relationships on attributes. Computationally, these rules, when collected together for all attributes in a CST, can be evaluated to compute all attribute values in some order, subject to respecting the attribute dependencies.

There should be exactly one computational rule for each synthesized attribute of a context-free rule's left-hand side and for each inherited attribute of each nonterminal of a context-free rule's right-hand side. Intuitively, this means that synthesized attributes are indeed computed upwards in the syntax tree, whereas inherited attributes are passed down. Additional constraints are needed to make the AG well defined and, in particular, to avoid cycles [2], but we omit these details here.

We are ready to show all computational rules for number conversion.

Illustration 12.5 (An attribute grammar for number conversion)
Consider the first context-free rule and the associated computational rules:

```
[number] number : bits rest ;
```

$$bits.Pos = bits.Len - 1$$
$$number.Val = bits.Val + rest.Val$$

That is, the inherited attribute Pos of the right-hand symbol bits is equated with the difference between the synthesized attribute Len of the right-hand symbol bits and 1, thereby defining the position of the leading bit in the sequence. The synthesized attribute Val of the left-hand side is equated with the sum of the Val attributes of the right-hand side, thereby combining the value of the integer and the fractional parts of the binary number.

These are the remaining context-free rules and the associated computational rules:

```
[single] bits : bit ;
```

$$bit.Pos = bits.Pos$$
$$bits.Val = bit.Val$$
$$bits.Len = 1$$

In the following context-free rule, we assign subscripts 0 *and* 1 *to the different occurrences of bits so that we can refer to the different attributes in the computational rules accordingly:*

```
[many] bits0 : bit bits1 ;
```

$$bit.Pos = bits0.Pos$$
$$bits1.Pos = bits0.Pos - 1$$
$$bits0.Val = bit.Val + bits1.Val$$
$$bits0.Len = bits1.Len + 1$$

```
[zero] bit : '0' ;
```

$$bit.Val = 0$$

```
[one] bit : '1' ;
```

$$bit.Val = 2^{bit.Pos}$$

```
[integer] rest : ;
```

$rest.Val = 0$

```
[rational] rest : '.' bits ;
```

$rest.Val = bits.Val$
$bits.Pos = -1$

Figure 12.4 shows the CST for 101.01 with the attributes of the relevant nonterminals. We use superscripts on the attribute names to make them unique across the tree. (The ids model path-based selection of the node. For instance, the id 1.2 states that we select the first subtree of the root and then the second subtree in turn.) In the figure, we also show the attribute dependencies in the tree, as defined by the computational rules; see the dotted arrows. The target of an arrow corresponds to the left-hand side of a computational rule.

It is worth noticing how the attribute dependencies point downwards and upwards in the tree. Consider, for example, Len^1, which is computed upwards in the tree and is used in initializing Pos^1, which is then used in computing other positions downwards the tree.

12.2.2 Attribute Evaluation

Given a CST and an AG, the process of computing all attributes for the CST is referred to as attribute evaluation. There are various methods of attribute evaluation [2]; one overall option is to perform static code generation for a tree walk so that the computations can be performed for any given CST without any run-time analysis. We will not discuss the corresponding technicalities here. Conceptually, we may view attribute evaluation as a simple mathematical problem in the sense of solving a system of equations.

Consider again Fig. 12.4 which encodes all the context-free rules involved and assigns unique names to all the attributes involved. For each context-free rule applied, we instantiate its computational rules for the unique attribute names taken from the CST. For instance, the root of the CST shown, with its two children, corresponds to an application of the rule [number]. Accordingly, we instantiate the computational rules as follows:

$$\begin{aligned} Pos^1 &= Len^1 - 1 \\ Val &= Val^1 + Val^2 \end{aligned}$$

The left child (id 1) with its two children (ids 1.1 and 1.2) corresponds to an application of the rule [many]. Accordingly, we instantiate the computational rules as follows:

$$\begin{aligned} Pos^{1.1} &= Pos^1 \\ Pos^{1.2} &= Pos^1 - 1 \\ Val^1 &= Val^{1.1} + Val^{1.2} \\ Len^1 &= Len^{1.2} + 1 \end{aligned}$$

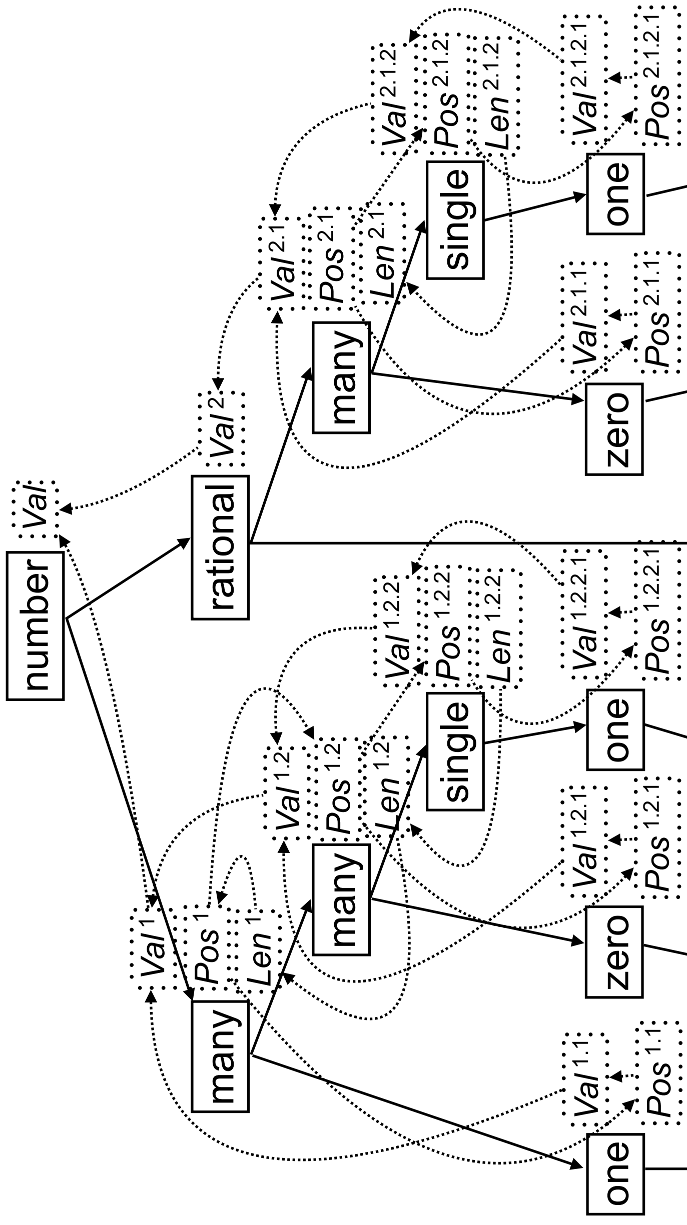


Fig. 12.4 Attributes to be evaluated for a CST of a binary number.

Once we have collected all these equations for all the applications of context-free rules together in a CST, we can simply start replacing references to attributes by values. The process starts with replacement for attributes with computational rules by constant expressions on the right-hand side. The process ends when replacements have assigned values to all attributes. The process is illustrated below.

Illustration 12.6 (Attribute evaluation)
Let us consider a much simplified example: the binary number 1 *for which we face the following equations; we also show the final value for each attribute:*

```
// [number]
  Pos^1      = Len^1 − 1      = 0
  Val        = Val^1 + Val^2  = 1
// [single]
  Pos^1.1    = Pos^1          = 0
  Val^1      = Val^1.1        = 1
  Len^1      = 1              = 1
// [one]
  Val^1.1    = 2^Pos^1.1      = 1
// [integer]
  Val^2      = 0              = 0
```

$$\begin{array}{lll}
\textit{// }\texttt{[number]} & & \\
\quad Pos^{1} & = Len^{1} - 1 & = 0 \\
\quad Val & = Val^{1} + Val^{2} & = 1 \\
\textit{// }\texttt{[single]} & & \\
\quad Pos^{1.1} & = Pos^{1} & = 0 \\
\quad Val^{1} & = Val^{1.1} & = 1 \\
\quad Len^{1} & = 1 & = 1 \\
\textit{// }\texttt{[one]} & & \\
\quad Val^{1.1} & = 2^{Pos^{1.1}} & = 1 \\
\textit{// }\texttt{[integer]} & & \\
\quad Val^{2} & = 0 & = 0
\end{array}$$

The solution of the equation system commences as follows:

- *Replace references to Val^{2} and Len^{1} by their values.*
- *Compute Pos^{1}.*
- *Replace the reference to Pos^{1} by its value.*
- *Replace the reference to $Pos^{1.1}$ by its value.*
- *Compute $Val^{1.1}$.*
- *Replace the reference to $Val^{1.1}$ by its value.*
- *Replace the reference to Val^{1} by its value.*
- *Compute Val.*

There exist various AG classes which impose constraints on attribute dependencies so that attribute evaluation can be performed more easily or efficiently. We briefly mention two such classes here:

S-attribution There are synthesized attributes only. Thus, all dependencies point upwards in a CST. In this case, attribute evaluation can be accomplished as a simple walk over CSTs. This scheme facilitates, for example, simple forms of CST/AST construction.

L-attribution There are also inherited attributes, but there are no right-to-left dependencies in a CST. This means that we can pass inherited attributes from the left-hand side to inherited attributes on the right-hand side and we can pass synthesized attributes of any nonterminal on the right-hand side to inherited attributes on the right-hand side if they are further to the right. In this case, attribute evaluation can be also be accomplished by a simple walk, which can be carried out during parsing if the CST is built from left to right.

Let us consider an S-attributed variation on the running example.

Illustration 12.7 (An S-attributed variation on number conversion)
Compared to Illustration 12.5, we do not use not any attributes for positions in the following variation. These are the computational rules:

```
[number] number : bits rest ;
```

$$number.Val = bits.Val + rest.Val$$

```
[single] bits : bit ;
```

$$bits.Val = bit.Val$$
$$bits.Len = 1$$

```
[many] bits0 : bit bits1 ;
```

$$bits0.Val = bit.Val * 2^{bits1.Len} + bits1.Val$$
$$bits0.Len = bits1.Len + 1$$

```
[zero] bit : '0' ;
```

$$bit.Val = 0$$

```
[one] bit : '1' ;
```

$$bit.Val = 1$$

```
[integer] rest : ;
```

$$rest.Val = 0$$

```
[rational] rest : '.' bits ;
```

$$rest.Val = bits.Val \,/\, 2^{bit.Len}$$

It should be clear by now that the computational rules in an AG are necessarily tied to the underlying CST structure. That is, given two context-free grammars that generate the same language (i.e., set of strings), the two grammars may require different computational rules to achieve the same ultimate result. We use the term “result” here in the sense of a dedicated synthesized attribute of the start symbol such as the decimal value of a binary number in the running example. The dependence between context-free and computational rules is illustrated below.

Illustration 12.8 (A left-recursive variation on Illustration 12.7)
The rule [many] *for bit sequences was defined in right-recursive style in Illustration 12.7. If we use left-recursive style instead, then the associated computational rules are adapted as follows:*

```
[many] bits0 : bits1 bit ;
```

$bits0.Val = 2 * bits1.Val + bit.Val$
$bits0.Len = bits1.Len + 1$

12.2.3 Attribute Grammars as Functional Programs

Attribute grammars provide a declarative computational paradigm that is actually very similar to (some form of) functional programming. That is, we may encode AGs as disciplined functional programs [75, 71]. One encoding scheme may be summarized as follows:

- Without loss of generality, we operate on the abstract as opposed to the concrete syntax. That is, we interpret computational rules on top of algebraic constructors as opposed to context-free rules.
- We associate each syntactic category (sort) with a function with one equation per alternative (constructor) to model the associated computational rules. That is, a function's patterns match on the syntactic structure. The inherited attributes of the category become function arguments, whereas the synthesized attributes become function results. Overall, we switch from the use of attribute names to the use of positions in argument lists and result tuples.
- Types of attribute values and operations on these types – as they are used in the computational rules – are also modeled in the functional program.

This encoding is illustrated for the AG for binary-to-decimal number conversion; see Illustrations 12.9 and 12.10 below.

Illustration 12.9 (Representation of binary numbers)

Haskell module Language.BNL.Syntax

```
data Number = Number Bits Rest
data Bits = Single Bit | Many Bit Bits
data Bit = Zero | One
data Rest = Integer | Rational Bits
```

Illustration 12.10 (Binary-to-decimal number conversion)

Haskell module Language.BNL.Conversion

```
number :: Number → Float
number (Number bs r) = val0
  where
    (len1, val1) = bits bs pos1
    pos1 = len1 − 1
    val2 = rest r
    val0 = val1 + val2

bits :: Bits → Int → (Int, Float)
bits (Single b) pos = (1, bit b pos)
bits (Many b bs) pos0 = (len0, val0)
  where
    val1 = bit b pos0
    (len1, val2) = bits bs pos1
    pos1 = pos0 − 1
    len0 = len1 + 1
    val0 = val1 + val2

bit :: Bit → Int → Float
bit Zero _pos = 0
bit One pos = 2^^pos

rest :: Rest → Float
rest Integer = 0
rest (Rational bs) = val
  where
    (_len, val) = bits bs pos
    pos = −1
```

Because of the generality of attribute grammars, the result of encoding may be such that function arguments depend on results. This is indeed the case for the example at hand; consider the function corresponding to the rule [number], which we repeat for clarity:

```
number (Number bs r) = val0
  where
    (len1, val1) = bits bs pos1
    pos1 = len1 − 1
    val2 = rest r
    val0 = val1 + val2
```

The result of applying the function bits includes the length len1 of the bit sequence, which is then used in setting up pos1, i.e., the position of the leading bit in the sequence, to be passed as an argument to the same function application. This functional program is sound only for lazy (as opposed to eager) language semantics. Thus, AGs can be said to be declarative because no particular order of computation is expressed directly; instead, an order must be determined which respects attribute dependencies. Lazy evaluation happens to determine a suitable order.

Exercise 12.5 (An AG for translation) [Intermediate level]
The running example (binary-to-decimal number conversion) can be seen as a trivial form of translation. Let us consider a more significant form of translation: imperative statements are to be mapped to bytecode (Section 5.2), just as in a real compiler. Devise an AG for this purpose.

12.2.4 Attribute Grammars with Conditions

AGs are routinely used to impose "context conditions" on syntactical structure, as discussed as a general concern earlier (Section 3.3). That is, we may use AGs effectively to represent the typing and name-binding rules of a software language. If we want to model conditions, then, in principle, we can simply use computational rules on Boolean-typed attributes. Alternatively, we may assume a more convenient AG notation with explicit support for conditions in addition to regular computational rules. Attribute evaluation is supposed to "fail" if any condition does not hold.

We now discuss conditions for a somewhat more complex example of an AG specification. Specifically, we consider an imperative language with a nested block structure: EIPL (*E*xtended *I*mperative *P*rogramming *L*anguage), which is an extension of BIPL. Each block (scope) may declare variables and (parameterless) procedures. The use of variables and procedures entails some nontrivial conditions to be understood relative to an *environment* maintaining scopes. Consider the following sample.

Illustration 12.11 (An imperative program with block structure)

EIPL resource languages/EIPL/sample.eipl

```
1  begin
2    var x = 0;
3    proc p { x = x + x; }
4    proc q { call p; }
5    begin
6      var x = 5;
7      proc p { x = x + 1; }
8      {
9        call q;
10       write x;
11     }
12   end
13 end
```

In particular, the sample program declares a variable x and a procedure p in two different scopes. Thus, it is important to understand what the different references to x and p actually resolve to. We assume lexical (static) scope here. The call to q (line 9) in the inner block makes q call p (line 4) in the outer block, whose reference to x (line 3) resolves to the x in the outer block (line 2). (If we assume dynamic scope instead, then x (line 3) resolves to the x in the inner block (line 6), as the call chain departed from there.)

We can model such conditions in an AG. In the following example, we mark conditions with the keyword "require".

Illustration 12.12 (An attribute grammar for checking block structure)
Within the conditions and computational rules, we use the following function and condition symbols on attributes of an assumed type Env *for environments:*

empty *This is the representation of the empty environment, i.e., an empty collection of scopes from which to start the semantic analysis.*
enterScope *This function modifies an environment to enter a new (nested) scope.*
noClash *This condition checks that a name is not yet bound in the current scope of the given environment.*
addVar *This function adds a variable with a name and a type to the current scope of the given environment.*
addProc *This function adds a procedure with a name to the current scope of the given environment.*
isVar *This condition checks that a name can be resolved to a variable in the current scope or an enclosing scope of the given environment.*
getType *The type of the variable is returned. The type is only defined if isVar holds.*
isProc *This condition checks that a name can be resolved to a procedure in the current scope or an enclosing scope of the given environment.*

```
program : scope ;
```

scope.EnvIn = empty

```
scope : 'begin' decls stmt 'end' ;
```

decls.EnvIn = enterScope(scope.EnvIn)
stmt.EnvIn = decls.EnvOut

```
decls0 : decl decls1 ;
```

decl.EnvIn = decls0.EnvIn
decls1.EnvIn = decl.EnvOut
decls0.EnvOut = decls1.EnvOut

```
decls : ;
```

decls0.EnvOut = decls0.EnvIn

```
[var] decl : 'var' name '=' expr ';' ;
```

require *noClash(decl.EnvIn, name.id)*
decl.EnvOut = addVar(decl.EnvIn, name.Id, expr.Type)
expr.EnvIn = decl.EnvIn

```
[proc] decl : 'proc' name stmt ;
```

require *noClash(decl.EnvIn, name.id)*
decl.EnvOut = addProc(decl.EnvIn, name.Id)
stmt.EnvIn = decl.EnvIn

```
[skip] stmt : ';' ;
```

```
[assign] stmt : name '=' expr ';' ;

    require isVar(stmt.EnvIn, name.Id)
    require getType(stmt.EnvIn, name.Id) = expr.Type
    expr.EnvIn = stmt.EnvIn

[call] stmt : 'call' name ';' ;

    require isProc(stmt.EnvIn, name.Id)

[scope] stmt : scope ;

    scope.EnvIn = stmt.EnvIn

// Remaining statement forms omitted for brevity
...

[intconst] expr : integer ;

    expr.Type = intType

[var] expr : name ;

    require isVar(expr.EnvIn, name.Id)
    expr.Type = getType(expr.EnvIn, name.Id)

// Remaining expression forms omitted for brevity
...
```

Exercise 12.6 (Recursive procedures) [Basic level]
Does the given AG permit (model) recursive procedures? Discuss how to change the AG so that recursive procedures are expressed or not expressed.

Exercise 12.7 (Functional encoding for block structure) [Basic level]
Exercise the functional program encoding (Section 12.2.3) for the given AG. (See the repository for additional positive and negative test cases.) You may use Boolean-typed attributes for the conditions or, instead, operate in the Maybe monad.

Exercise 12.8 (An interpreter for EIPL) [Intermediate level]
Implement an interpreter that includes block structure.

12.2.5 Semantic Actions with Attributes

In Section 7.3.2, we discussed semantic actions as a means of injecting statements of the target language for parser generation into a grammar. Specifically, we used semantic actions for AST construction during parsing. A parser description with semantic actions can be considered a limited form of an AG because computational actions are associated with context-free grammar productions.

In fact, parser generators may also support proper synthesized and inherited attributes. In particular, S-attribution is supported by the "typical" parser generator. This is demonstrated for ANTLR below. That is, we transcribe the S-attributed grammar variation on binary-to-decimal number conversion (Illustration 12.7) quite directly to ANTLR notation.

Illustration 12.13 (Binary-to-decimal number conversion)

ANTLR resource languages/BNL/ANTLR/BnlBnfConversion.g4

```
grammar BnlBnfConversion;
@header {package org.softlang.bnl;}

number returns [float val]
     : bits rest WS? EOF { $val = $bits.val + $rest.val; }
     ;
bits returns [float val, int len]
     : bit { $val = $bit.val; $len = 1; }
     | bits1=bits bit { $val = 2*$bits1.val + $bit.val; $len = $bits1.len + 1; }
     ;
bit returns [int val]
     : '0' { $val = 0; }
     | '1' { $val = 1; }
     ;
rest returns [float val]
     : { $val = 0; }
     | '.' bits { $val = $bits.val / (float)Math.pow(2, $bits.len); }
     ;
WS : [ \t\n\r]+ ;
```

While ANTLR does not support AGs in their full generality, ANTLR's support goes beyond S-attribution. That is, L-attribution (i.e., a limited form of inherited attributes on top of S-attribution) is also supported. We demonstrate L-attribution with a parser for FSML below. We use a synthesized attribute for the constructed AST. We use inherited attributes to pass appropriate "context" for AST construction.

Illustration 12.14 (A parser for finite state machines)

ANTLR resource languages/FSML/Java/FsmlToObjects2.g4

```
grammar FsmlToObjects2;
@header {package org.softlang.fsml;}

fsm returns [Fsm result] :
  { $result = new Fsm(); }
  state[$result]+
  EOF
  ;
state[Fsm result] :
  { boolean initial = false; }
  ('initial' { initial = true; })?
  'state' stateid
  { String source = $stateid.text; }
  { $result.getStates().add(new State(source, initial)); }
  '{' transition[$result, source]* '}'
  ;
transition[Fsm result, String source] :
  event
  { String action = null; }
  ('/' action { action = $action.text; })?
  { String target = source; }
  ('−>' stateid { target = $stateid.text; })?
  { $result.getTransitions().add(new Transition(source, $event.text, action, target)); }
  ';'
  ;
... // Lexical syntax as before
```

That is, we pass the FSM to the nonterminals *state* and *transition* as context so that states and transitions can be added to the appropriate collections in the scope of the corresponding productions. We also pass the state id of a state declaration to each of its transitions so that it can be used as the target state id when an explicit target is omitted.

The parser is invoked like this:

```
 Fsm fsm = parser.fsm().result;
```

Thus, the FSM is retrieved as the result of invoking the method *fsm*, i.e., we access the synthesized attribute result of the nonterminal *fsm*.

Let us return to the more significant AG for checking imperative programs with block structure (Section 12.2.4) and implement the AG with ANTLR. The following ANTLR code takes advantage of the imperative nature of the host language such that the environment is implemented as a global attribute rather than passing the object reference for the environment with attributes and copy rules.

Illustration 12.15 (Checking block structure)

ANTLR resource languages/EIPL/ANTLR/EiplChecker.g4

```
grammar EiplChecker;
@header {package org.softlang.eipl;}
@members {
public boolean ok = true;
public Env env = new Env();
}

program : scope EOF ;
scope : { env.enterScope(); } 'begin' decl* stmt 'end' { env.exitScope(); } ;
```

```
10 decl :
11         'var' NAME '=' expr ';'
12         { ok &= env.noClash($NAME.text); env.addVar($NAME.text, $expr.type); }
13     |
14         'proc' NAME stmt
15         { ok &= env.noClash($NAME.text); env.addProc($NAME.text); }
16     ;
17 stmt :
18         ';'
19     |
20         NAME '=' expr ';'
21         { ok &= env.isVar($NAME.text) && env.getType($NAME.text) == $expr.type; }
22     |
23         'call' NAME ';'
24         { ok &= env.isProc($NAME.text); }
25     |
26         scope
27     |
28         // Remaining statement forms omitted for brevity
29         ...
30     ;
31 expr returns [Env.Type type] :
32         INTEGER
33         { $type = Env.Type.IntType; }
34     |
35         NAME
36         { ok &= env.isVar($NAME.text); $type = env.getType($NAME.text); }
37     |
38         expr1=expr '+' expr2=expr
39         {
40           ok &= $expr1.type == Env.Type.IntType
41               && $expr2.type == Env.Type.IntType;
42           $type = Env.Type.IntType;
43         }
44     |
45         // Remaining expression forms omitted for brevity
46         ...
```

In line 9, we use a new symbol, exitScope, to exit the scope by means of a side effect. In the original formulation of the AG, there is no counterpart because the basic formalism is free of side effects. In various semantic actions, for example, in lines 12 and 15, we adapt a global attribute ok, as declared in line 4, to communicate condition failures, if any.

The use of global attributes may be acceptable if the technology-defined order of executing semantic actions and thus the order of side effects on the global attributes are easily understood to be correct. We mention in passing here that there exist AG extensions that aim at avoiding laborious computational "copy rules" in a more declarative manner [24, 33, 5].

We need a concrete realization of environments, as accessed within the conditions and computational rules. That is, we assume a corresponding type *Env*, as implemented below. We assume here again that ANTLR is used together with Java.

Illustration 12.16 (The environment for checking block structure)

Java source code org/softlang/eipl/Env.java

```
import java.util.Stack;
import java.util.HashMap;

public class Env {
  public enum Type { NoType, IntType, BoolType }
  private abstract class Entry { String id; }
  private class VarEntry extends Entry { Type ty; }
  private class ProcEntry extends Entry { }
  private Stack<HashMap<String, Entry>> stack = new Stack<>();
  public void enterScope() { stack.push(new HashMap<>()); }
  public void exitScope() { stack.pop(); }
  public boolean noClash(String id) { return !stack.peek().containsKey(id.intern()); }
  public void addVar(String id, Type ty) {
    VarEntry entry = new VarEntry();
    entry.ty = ty;
    stack.peek().put(id.intern(), entry);
  }
  public void addProc(String id) { stack.peek().put(id.intern(), new ProcEntry()); }
  public boolean isVar(String id) { return chase(id.intern()) instanceof VarEntry; }
  public boolean isProc(String id) { return chase(id.intern()) instanceof ProcEntry; }
  public Type getType(String id) {
    Entry entry = (VarEntry) chase(id.intern());
    return entry instanceof VarEntry ? ((VarEntry) entry).ty : Type.NoType;
  }
  private Entry chase(String id) {
    Entry entry = null;
    for (HashMap<String, Entry> map : stack)
      if (map.containsKey(id)) {
        entry = map.get(id);
        break;
      }
    return entry;
  }
}
```

In particular, lexical scopes are maintained as a stack of hash-maps – one hash-map per scope. Variables and procedures are searched for in the environment by a stack traversal, i.e., starting at the top of the stack, which is the current scope.

The present section is summarized by means of a recipe.

Recipe 12.2 (Design of an attribute grammar).

Syntax *Define the underlying syntax, typically, by means of a context-free grammar. (Tree grammars may be used instead of context-free grammars, for example, in the form of algebraic data types in a functional program* *encoding.) In this manner, we can already separate valid inputs from invalid inputs, subject to implementing the grammar as a syntax checker (Recipe 2.5).*

Test cases *Set up test case for the AG. In particular, a positive test case consists of a valid input and the expected output which is to be modeled eventually as a dedicated attribute of the start symbol.*

Attributes *Associate the grammar symbols with attributes (their names and types), thereby expressing what sort of information should be available at what sort of node in the parse tree. Mark the attributes as either synthesized or inherited, in accordance with attribute dependencies.*

Computations and conditions *Define computational rules for each context-free grammar rule, all synthesized attributes of the left-hand side nonterminal and all inherited attributes of the right-hand side nonterminals. If necessary, impose additional conditions on the attributes. (At this point, one may need to commit to a particular AG system or to an encoding of the AG, for example, as a functional program.)*

Testing *Test attribute evaluation in terms of the test cases. This may also entail parsing.*

12.3 Multi-Stage Programming

Multi-stage programming is a style of metaprogramming that is particularly useful for program generation, i.e., for writing programs that generate programs or parts thereof at compile time or runtime. A “stage” is essentially a point in time at which programs or parts thereof are compiled, generated, or evaluated, i.e., compile time versus runtime versus different stages at runtime [78, 76, 77]. We focus here on compile-time metaprogramming as a form of multi-stage programming, which means that parts of the program are executed at compile time to compute and compile additional parts of the program (again at compile time).

The motivation for multi-stage programming is often performance in the presence of using more or less advanced abstraction mechanisms. In a simple case, program generation may help to inline function applications for arguments that are known at compile time, as we will demonstrate below. In a more advanced case, program generation may help with providing domain-specific concepts in a performant and possibly type-safe manner.

Multi-stage programming is clearly a form of metaprogramming, as a multi-stage program is essentially a program generator with the metalanguage and the object language being the same language. (In reality, the language of generated code may be restricted compared with the full metalanguage.) Multi-stage programming language extensions do not just strive for syntactically correct generated code, but name-binding and typing rules may also be guaranteed (more or less) statically.

12.3.1 Inlining as an Optimization Scenario

In the sequel, we briefly demonstrate multi-stage programming in terms of Haskell's Template Haskell extension [69] for compile-time metaprogramming. We discuss a very simple scenario – essentially, programmatic inlining of recursive function applications. Template Haskell has seen many uses, especially also in the context of optimizations for DSLs [67].

Consider the recursive function definition as follows.

Illustration 12.17 (A power function)

Haskell module Power

```
power :: Int → Int → Int
power n x =
  if n==0
    then 1
    else x * power (n−1) x
```

Now assume that within some part of our program, we need the exponent 3 time and again. We may even define a dedicated application of the power function and use it as illustrated below:

Interactive Haskell session:

```
▶ let power3 = power 3
▶ power3 3
27
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ power3 4
64
```

Alas, the overhead of applying the recursively defined power function is incurred, even for applications of power3 unless we imagine a Haskell compiler that somehow decides to inline recursive function applications in some way. Multi-stage programming allows us to express explicitly that nonrecursive code is to be generated on the basis of the known exponent.

12.3.2 Quasi-Quotation and Splicing

In Template Haskell, we may define a variation on the power function which is recursive at compile time, and generates nonrecursive code for a fixed first argument, as shown below.

Illustration 12.18 (A staged power function)

Haskell module UntypedPower

```
power :: Int → Q Exp → Q Exp
power n x =
  if n==0
    then [| 1 |]
    else [| $x ∗ $(power (n−1) x) |]

mk_power :: Int → Q Exp
mk_power n = [| λ x → $(power n [| x |]) |]
```

Notice that the structure of the code is very similar to the original power function. There is an additional function, mk_power, that we will explain in a second. The following elements of multi-stage programming are at play:

- We use quasi-quote brackets [| ··· |] (or Oxford brackets) to quote Haskell code, thereby expressing that the corresponding expression evaluates to code. For instance, for the base case of power we simply return the code 1.
- Within the brackets, we use splicing $(···) to insert code that is computed when the quasi-quoted code is constructed. For instance, for the recursive case of the power function, we insert the code returned by the recursive application of the code-generating function.
- We use the quotation monad Q in places where we compute code. This monad takes care of fresh-name generation, reification (program inspection), and error reporting.
- The type Exp stands for "Haskell expressions". By using the quasi-quote brackets, we enter a scope in which Haskell expressions are constructed (in fact, "computed") as results.

The additional function mk_power serves the purpose of applying power to an actual exponent. A lambda abstraction is constructed to receive the missing argument x; the body of the function splices in the code generated for a given exponent n. Here is a demonstration where we apply the generated function and inspect the generated code for clarity:

Interactive Haskell session:

```
► let power3 = $(mk_power 3)
► power3 3
27
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
► power3 4
64
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
► runQ (mk_power 3) >>=putStrLn . pprint
λ x_0 → x_0 ∗ (x_0 ∗ (x_0 ∗ 1))
```

At the last prompt, we use the "run" function of Template Haskell's quotation monad (i.e., runQ) to actually perform code generation in the Haskell session, and we pretty print the code, as shown. The generated code clearly conveys that the recursive definition of the power function was unfolded three times, ending in the base case.

Exercise 12.9 (Fine-tuning the code generator) [Intermediate level]
The generated code clearly involves an unnecessary multiplication with "1" at the right end of the multiplication. It is reasonable to expect that the compiler may take care of this by implementing a unit law for multiplication. However, whether or not a particular compiler optimization is available and applicable is generally a complicated matter. So we might prefer to make the code generator avoid the unnecessary multiplication in the first place. Adjust the generator accordingly so that it returns this code instead:

```
λ x_0 → x_0 * (x_0 * x_0)
```

12.3.3 More Typeful Staging

Arguably, the staged code discussed above lacks some important type information that might be valuable for a user of the code generator and helpful in type checking. Most notably, the revised power function takes an argument of type Exp and its result is of type Exp too. Thus, we neither document nor enforce the condition that the power function operates on Ints. Template Haskell also provides a more typeful model such that we can track the expected type of Haskell expressions, as demonstrated below.

Illustration 12.19 (A more typefully staged power function)

Haskell module TypedPower

```
1 power :: Int → Q (TExp Int) → Q (TExp Int)
2 power n x =
3   if n==0
4     then [|| 1 ||]
5     else [|| $$x * $$(power (n−1) x) ||]
6
7 mk_power :: Int → Q (TExp (Int → Int))
8 mk_power n = [|| λ x → $$(power n [|| x ||]) ||]
```

That is, we use the type constructor TExp (lines 1 and 7) instead of Exp, thereby capturing the expected type of expression. The power function is more clearly typed now in that it takes an Int and code that evaluates to an Int; the function returns code that evaluates to an Int (i.e., the actual expression for computing the power). We use "typed" quasi-quote brackets [|| ⋯ ||] (e.g., in line 4) and "typed" splices $$(⋯) (e.g., in line 5). Other than that, the program generator remains unchanged, and it can be also used in the same manner as before. Incorrect uses would be caught by the type system at the time of checking the quasi-quotes in the staged abstractions, i.e., even before applying the staged abstractions.

In staging, as much as in using macro systems, one needs to be careful about unintended name capture so that names from the generated code do not interfere in an unintended manner with other names in scope, thereby giving rise to a notion of *hygiene* [1].

Staging does not need to involve systematic quasi-quotation and splicing, as demonstrated by the "Scala-Virtualized" approach [64]. In this approach, overloading is used in a systematic manner so that the transition between regular code and quoted code ("representations") is expressed by type annotations. This idea, which relies on some Scala language mechanisms, was fully developed in the body of work on *LMS* (Lightweight Modular Staging) with applications of staging for the benefit in performance across different domains [63, 74, 30, 38], for example, database queries or parsing.

This concludes our brief discussion of multi-stage programming. There exist different language designs that support multi-stage programming. In a simple case, macro systems may be used for program generation. In the case of the language C++, its template system caters for a form of multi-stage programming, i.e., template metaprogramming [84, 70, 59]. We have exercised Template Haskell [69], thereby taking advantage of dedicated language support multi-stage programming, including means of quotation and splicing. MetaML [79], MetaOCaml [54, 35], and Helvetia [62] also provide such support. There exists scholarly work comparing or surveying approaches in a broader context [15, 61, 19, 72].

Exercise 12.10 (A recipe for multi-stage programming) [Intermediate level]
Describe a recipe for the design of a multi-stage program (Recipe 12.3). Aim at adopting the style for recipes used elsewhere in this book. In particular, you may consult Recipe 12.4 for inspiration, as it is concerned with the problem of partial evaluation, which is closely related to multi-stage programming. In the view of the breadth of the field of multi-stage programming, as indicated by the discussion of related work above, you are advised to focus your recipe on the kind of optimization that we demonstrated above.

Recipe 12.3 (Design of a multi-stage program). *See Exercise 12.10.*

12.4 Partial Evaluation

Partial evaluation[2] is a style of metaprogramming where a program is systematically "refined" on the basis of partially known program input so that a partially evaluated (specialized) program is computed; the primary objective is optimization [27, 22]. Partial evaluation is based essentially on one simple idea: evaluate (execute) a program to the extent possible for incomplete input. The result of partial evaluation is a specialized program, also referred to as a *residual program*.

Partial evaluation has applications in, for example, modeling [81], model-driven development [68], domain-specific language engineering [26], generic programming [53], and optimization of system software [55]. The technique of partial evaluation is particularly useful and well understood when the program is an interpreter and the partially known program input is the program to be interpreted. In this case, one gets essentially a compiler.

We introduce partial evaluation (or program specialization) by means of writing a simple partial evaluator for a simple, pure, first-order, functional programming language. In particular, we will show that the partial evaluator can be derived as a variation on a compositionally defined interpreter. The style of partial evaluator developed here is called an *online* partial evaluator because it makes decisions about specialization as it goes, based on whatever variables are in the environment at a given point during evaluation [27]. (An *offline* partial evaluator performs a static analysis of the program to decide which variables will be considered known versus unknown.)

12.4.1 The Notion of a Residual Program

We pick up essentially the same example that we studied in the context of multi-stage programming, i.e., the application of the power function with a known exponent. However, this time around, we separate the metalanguage and the object language: Haskell versus a "fabricated" functional language (BFPL). That is, we use an object program as follows.

[2] Acknowledgment and copyright notice: This section is derived from a tutorial paper, jointly written with William R. Cook [8], who has kindly agreed to the material being reused in this book. The tutorial was published by EPTCS, subject to the rule that copyright is retained by the authors.

Illustration 12.20 (A power function and an application thereof)

BFPL resource languages/BFPL/samples/power.bfpl

```
power :: Int −> Int −> Int
power n x =
  if (==) n 0
    then 1
    else (∗) x (power ((−) n 1) x)

main = print $ power 3 2 −− Prints 8
```

Now let us suppose that only the value of the exponent is given, while the base remains a variable. A partial evaluator should return the following code:

```
  (∗) x ((∗) x ((∗) x 1))
```

In infix notation:

```
  x ∗ x ∗ x ∗ 1
```

That is, partial evaluation should specialize the program such that the recursive function is essentially inlined as many times as needed for the different exponents encountered recursively. In contrast to multi-stage programming, we do not want to "instrument" the power function in any way (by quasi-quotation and such); instead, inlining should be triggered by setting the corresponding function argument to unknown.

We use Haskell as the meta-language for implementing the partial evaluator. We take advantage of the fact that we have already implemented an interpreter for the object language (BFPL) in Haskell in Section 5.1.3. The power function is easily represented as a Haskell term as shown below.

Illustration 12.21 (The power function in abstract syntax)

Haskell module Language.BFPL.Samples.Power

```
power :: Function
power = (
  "power",
  (([IntType, IntType], IntType),
   (["n", "x"],
     If (Binary Eq (Arg "n") (IntConst 0))
        (IntConst 1)
        (Binary Mul
            (Arg "x")
            (Apply "power" [Binary Sub (Arg "n") (IntConst 1), Arg "x"])))))
```

This is the signature of a regular interpreter function (Illustration 5.8):

```
eval :: Program → Value
```

A “total evaluator” can handle applications of the power function only if the arguments denote values; evaluation fails if missing arguments are dereferenced:

Interactive Haskell session:

```
▶ eval ([power], (Apply "power" [IntConst 3, IntConst 2]))
Left 8
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ eval ([power], (Apply "power" [IntConst 3, Arg "x"]))
*** Exception: ...
```

A partial evaluator is similar to an interpreter, but it returns residual code instead of values. For now, we assume a simple scheme of partial evaluation such that a residual expression is returned:

```
peval :: Program → Expr
```

A partial evaluator agrees with a total evaluator, i.e., a regular interpreter, when values for all arguments are provided. However, when an argument is a variable without binding in the environment, some operations cannot be applied, and they need to be transported into the residual code:

Interactive Haskell session:

```
▶ peval ([power], (Apply "power" [IntConst 3, IntConst 2]))
IntConst 8
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ peval ([power], (Apply "power" [IntConst 3, Arg "x"]))
Binary Mul (Arg "x") (Binary Mul (Arg "x") (Binary Mul (Arg "x") (IntConst 1)))
```

12.4.2 Interpretation with Inlining

Let us implement the envisaged partial evaluator by enhancing a regular interpreter with inlining. In principle, the approach presented here works for any interpreter following the scheme of, more or less closely, big-step operational or denotational semantics.

Illustration 12.22 (An interpreter with inlining of function applications)

Haskell resource languages/BFPL/Haskell/Language/BFPL/Inliner.hs

```
type Env = Map String Expr

peval :: Program → Expr
peval (fs, e) = f e empty
```

```
5    where
6      f :: Expr → Env → Expr
7      f e@(IntConst _) _ = e
8      f e@(BoolConst _) _ = e
9      f e@(Arg x) env =
10       case Data.Map.lookup x env of
11         (Just e') → e'
12         Nothing → e
13     f (If e0 e1 e2) env =
14       let
15         r0 = f e0 env
16         r1 = f e1 env
17         r2 = f e2 env
18       in
19        case toValue r0 of
20          (Just (Right bv)) → if bv then r1 else r2
21          Nothing → If r0 r1 r2
22     f (Unary o e) env =
23       let r = f e env
24       in case toValue r of
25            (Just v) → fromValue (uop o v)
26            _ → Unary o r
27     f (Binary o e1 e2) env = ...
28     f (Apply fn es) env = f body env'
29       where
30         Just (_, (ns, body)) = Prelude.lookup fn fs
31         rs = map (flip f env) es
32         env' = fromList (zip ns rs)
33
34 -- Attempt extraction of value from expression
35 toValue :: Expr → Maybe Value
36 toValue (IntConst iv) = Just (Left iv)
37 toValue (BoolConst bv) = Just (Right bv)
38 toValue _ = Nothing
39
40 -- Represent value as expression
41 fromValue :: Value → Expr
42 fromValue (Left iv) = IntConst iv
43 fromValue (Right bv) = BoolConst bv
```

The inlining partial evaluator deviates from the regular interpreter as follows:

- The partial evaluator maps expressions to residual expressions, whereas the regular interpreter maps expressions to values. Values are trivially embedded into expressions through the constant forms of expressions, subject to the conversions fromValue and toValue (lines 34–43).
- The partial evaluator uses an environment (line 1) which maps argument names to expressions, whereas the regular interpreter's environment maps argument names to values. This is necessary when function arguments cannot be evaluated completely and, thus, residual code needs to be passed to the applied function.

- The cases of the partial evaluator for the different expression forms (lines 7–32) are systematically derived from the cases of the regular interpreter (Illustration 5.8) by performing regular evaluation when subexpressions are values and returning residual code otherwise. The cases are explained one by one as follows:

  IntConst/BoolConst A constant is partially evaluated to itself, just like in the regular interpreter.

  Arg An argument is partially evaluated to a value according to the variable's binding in the environment, just like in the regular interpreter, if there is a binding. Otherwise, the variable is partially evaluated to itself; the regular interpreter fails in this case.

  If An if-statement can be eliminated such that one of the two branches is chosen for recursive (partial) evaluation, just like in the regular interpreter, if the condition is (partially) evaluated to a Boolean value. Partial evaluation fails for an integer value, just like regular interpretation. If the condition is not partially evaluated to a value, an if-statement is reconstructed from the partially evaluated branches.

  Unary/Binary The corresponding operation is applied to the (partially) evaluated arguments, just like in the regular interpreter, if these are all values. Otherwise, a unary/binary expression is reconstructed from the partially evaluated arguments.

  Apply Partial evaluation involves argument (partial) evaluation, environment construction, and (partial) evaluation of the body in the new environment, just like in the regular interpreter – except that expressions for the partially evaluated arguments are passed in the environment in the case of the partial evaluator, as opposed to values in the case of the regular interpreter.

The treatment of if-statements and function applications is naive. In particular, partial evaluation of a function application may diverge, as illustrated by the following example:

Interactive Haskell session:

```
−− Result shown in concrete BFPL/Haskell syntax for clarity
► peval ([power], (Apply "power" [Arg "n", IntConst 2]))
if ((==) n 0)
  then 1
  else (*) 2 (if ((==) ((−) n 1) 0)
                then 1
                else (*) 2 (if ((==) ((−) ((−) n 1) 1) 0 ...))
```

The position with '...' proxies for infinite inlining. That is, in this example, the function power is applied to a specific base, 2, but the exponent remains a variable, n. Inlining diverges because the recursive case of power is expanded indefinitely.

Nevertheless, inlining is useful in a relatively well-defined situation. Before we generalize from inlining to full-fledged program specialization, let us discuss some variations on inlining by means of exercises.

Exercise 12.11 (Inlining with pairs) [Intermediate level]
Extend the functional language to support pairs. The following expression forms should be supported:

```
data Expr = ...
  | Pair Expr Expr -- Construction of a pair
  | Fst Expr -- 1st projection
  | Snd Expr -- 2nd projection
```

Another form of type is needed as well:

```
data Type = ... | PairType Type Type
```

For instance, a swap function for pairs of ints is defined as follows:

```
-- Haskell counterpart for comparison
-- swap :: (Int, Int) -> (Int, Int)
-- swap x = (snd x, fst x)
swap :: Function
swap =
  ( "swap",
    ( ([PairType IntType IntType], PairType IntType IntType),
      ("x", Pair (Snd (Arg "x")) (Fst (Arg "x")))
    )
  )
```

Extend the regular interpreter to support pairs. To this end, you also need to extend the type of values. Assume the following variant, which favors a dedicated algebraic data type over the use of Either:

```
data Value = IntValue Int | BoolValue Bool | PairValue Value Value
```

The extended interpreter must support this application:

Interactive Haskell session:

```
► evaluate ([swap], (Apply "swap" [Pair (IntConst 2) (IntConst 3)]))
PairValue (IntValue 3) (IntValue 2)
```

Extend the inliner to cover pairs so that it supports this application:

Interactive Haskell session:

```
► peval ([swap], (Apply "swap" [Pair (Arg "x") (Arg "y")]))
Pair (Arg "y") (Arg "x")
```

Exercise 12.12 (Loop unrolling in imperative programs) [Advanced level]
Let us consider partial evaluation in the context of an imperative language. This exercise is concerned with an optimization which is somewhat similar to function inlining. The optimization is to unroll loops in an imperative language. Consider the following imperative program for exponentiation, with base x*, exponent* n*, and result* r*:*

BIPL resource languages/BIPL/samples/exp-loop.bipl

```
{
  r = 1;
  while (n >= 1) {
      r = r * x;
      n = n − 1;
  }
}
```

Now suppose the exponent is known: $n = 3$*. In the absence of a known base* x*, a partial evaluator may still unroll the loop three times, since the loop condition depends only on* n*. This unrolling may result in code like this:*

BIPL resource languages/BIPL/samples/exp-unrolled.bipl

```
{
  r = 1;
  r = r * x;
  n = n − 1;
  r = r * x;
  n = n − 1;
  r = r * x;
  n = n − 1;
}
```

A data-flow analysis may determine that the result r *does not depend on* n *and, thus, all assignments to* n *may be removed. Such slicing may result in code like this:*

BIPL resource languages/BIPL/samples/exp-sliced.bipl

```
{
  r = 1;
  r = r * x;
  r = r * x;
  r = r * x;
}
```

Implement a partial evaluator for the unrolling part of this optimization.

12.4.3 Interpreter with Memoization

The proper treatment of recursive functions requires us to synthesize *residual programs* instead of just residual expressions. Also, we need to memoize specialization in a certain way, as we will discuss in a second. We need a partial evaluator of the following type:

```
peval :: Program → Program
```

The idea here is that the incoming function definitions and the main expression are specialized such that the resulting main expression refers to specialized function definitions. A given function definition may be specialized several times depending on the statically known argument values encountered. For instance, exponentiation with the exponent 3 would be specialized as follows; the result is shown in Haskell's concrete syntax for the sake of readability:

```
power'a x = x * power'b x
power'b x = x * power'c x
power'c x = x * power'd x
power'd x = 1
```

The names of the specialized functions are fabricated from the original name by some qualification scheme to account for disambiguation. Thus, specialized function definitions have been inferred for all the inductively encountered values 3, 2, 1, and 0 for the exponent. Subject to an inlining optimization, we obtain the familiar expression for x to the power 3. The inlining needed here is trivial, in that we would only inline nonrecursive functions. The "heavy lifting" is due to specialization.

Here is a demonstration of the implemented specializer; it returns the same specialized program, in abstract syntax.

Interactive Haskell session:

```
► peval ([power], (Apply "power" [IntConst 3, Arg "x"]))
( [
    ("power'a", (([IntType], IntType), (["x"],
      Binary Mul (Arg "x") (Apply "power'b" [Arg "x"])))),
    ("power'b", (([IntType], IntType), (["x"],
      Binary Mul (Arg "x") (Apply "power'c" [Arg "x"])))),
    ("power'c", (([IntType], IntType), (["x"],
      Binary Mul (Arg "x") (Apply "power'd" [Arg "x"])))),
    ("power'd", (([IntType], IntType), (["x"],
      IntConst 1)))
  ],
  Apply "power'a" [Arg "x"]
)
```

Exercise 12.13 (Inlining nonrecursive functions) [Intermediate level]
Implement an analysis (in Haskell) to determine for a given functional (BFPL) program the set of names of nonrecursive functions. For instance, all of the above functions power'a*, . . . ,* power'd *should be found to be nonrecursive. Hint: This analysis can be described like this:*

- *Start from the empty set of nonrecursive functions.*
- *Repeat the following step as long as new nonrecursive functions are still found:*
  - *Include a function in the set if it only applies functions that are already known to be nonrecursive. (Thus, initially, a function is included if it does not apply any function –* power'd *in our example.)*

Complement the analysis for nonrecursive functions to obtain the simple inlining optimization discussed above.

Let us illustrate how program specialization should handle the diverging example that we faced earlier. Program specialization should carefully track argument lists for which specialization is under way or has been completed. This solves the termination problem:

Interactive Haskell session:

```
► peval ([power], (Apply "power" [Arg "n", IntConst 2]))
( [
    ("power'a", (([IntType], IntType), (["n"],
      If (Binary Eq (Arg "n")
        (IntConst 0))
        (IntConst 1) (Binary Mul
                                    (IntConst 2)
                                    (Apply "power'a" [Binary Sub (Arg "n") (IntConst 1)])))))
  ],
  Apply "power'a" [Arg "n"]
)
```

Thus, the original definition of power has been specialized such that the argument position for the statically known base is eliminated. Note that the specialized function is recursive.

The program specializer is derived from the inliner and thus from the regular interpreter by making adaptations as described below. Overall, inlining is tamed so that termination is guaranteed. During inlining (in fact, specialization), specialized functions are aggregated in a data structure:

```
peval :: Program → Program
peval (fs, e) = swap (runState (f e empty) [])
  where
    f :: Expr → Env → State [Function] Expr
    ...
```

The state monad is applied to the result type to aggregate specialized functions along the way. The environment is of the same type as in the regular interpreter:

```
type Env = Map String Value
```

That is, the environment binds variables to values as opposed to expressions, as in the case of the naive inliner. Thus, the environment only serves to represent statically known arguments. Statically unknown arguments are preserved within the definitions of the specialized functions.

The cases for all constructs but function application can be taken from the inliner – except that we need to convert to monadic style, which is a simple, systematic program transformation in itself [43, 20], routinely performed by functional programmers. Thus, recursive calls to the specializer are not used directly in reconstructing terms, but their results are sequenced in the state monad. For instance:

```
f (Binary o e1 e2) env = do
  r1 ← f e1 env
  r2 ← f e2 env
  case (toValue r1, toValue r2) of
    (Just v1, Just v2) → return (fromValue (bop o v1 v2))
    _ → return (Binary o r1 r2)
```

It remains to define the case for partial evaluation of function applications; this case is significantly more complex than in the regular interpreter or the inliner. The case is presented below.

Illustration 12.23 (Specializing function applications)

Haskell resource languages/BFPL/Haskell/Language/BFPL/Specializer.hs

```
f (Apply fn es) env = do
  -- Look up function
  let Just ((ts, t), (ns, body)) = Prelude.lookup fn fs
  -- Partially evaluate arguments
  rs ← mapM (flip f env) es
  -- Determine static and dynamic arguments
  let trs = zip ts rs
  let ntrs = zip ns trs
  let sas = [ (n, fromJust (toValue r)) | (n, (_, r)) ← ntrs, isJust (toValue r) ]
  let das = [ (n, (t, r)) | (n, (t, r)) ← ntrs, isNothing (toValue r) ]
  -- Specialize body
  let body' = f body (fromList sas)
  -- Inlining as a special case
  if null das then body'
  -- Specialization
    else do
      -- Fabricate function name
      let fn' = fn ++ show sas
      -- Memoize new residual function, if necessary
      fs' ← get
      when (isNothing (Prelude.lookup fn' fs')) (do
22        -- Create placeholder for memoization
23        put (fs' ++ [(fn', undefined)])
24        -- Partially evaluate function body
25        body'' ← body'
26        -- Define residual
27        let r = ((map (fst . snd) das, t), (map fst das, body''))
28        -- Replace placeholder by actual definition
29        modify (update (const r) fn'))
30      -- Apply the specialized function
31      return (Apply fn' (map (snd . snd) das))
```

Here the following steps are performed:

1. The applied function is looked up (lines 2–3) and the arguments are evaluated (lines 4–5), just like in the regular interpreter. As a reminder, the list of function declarations is an association list mapping function names to lists of argument types ts, the result type t, argument names ns, and the body.
2. The partially evaluated arguments are partitioned into static arguments sas and dynamic arguments das (lines 6–10). Static arguments are values; dynamic arguments exercise other expression forms.
3. The body of the specialized function is obtained by partially evaluating the original body in the variable environment of the static variables (lines 11–12). In fact, we use a let-binding; the actual specialization needs to be demanded in a monadic computation (lines 14 and 25).
4. If there are no dynamic arguments, we switch to the behavior of the interpreter by (partially) evaluating the body of the applied function (lines 13–14).
5. The "identity" (name) of the specialized function is derived by concatenating the name of the applied function and the string representation of the actual values of the static arguments (lines 17–18); see Exercise 12.14 for a discussion of naming.
6. We need to remember (memoize) function specializations so that a function is not specialized again for the same static arguments, thereby guarding against infinite inlining (lines 19–29).
7. In order to deal with recursion, it is important that the specialized function has already been added to the state before its body is obtained so that it is considered known during specialization. To this end, an undefined function, except for the name, is initially registered as a placeholder (lines 22–23), to be updated later (lines 28–29).
8. The argument list of the specialized function (the "residual") includes only variables for the dynamic positions (lines 26–27). The specialized function is ultimately applied to the dynamic arguments; the expression for that application serves as the result of partial evaluation (lines 30–31).

Exercise 12.14 (Readable function names) [Intermediate level]
In the illustration of the specializer, we used readable names for the specialized functions, power'a, …, power'd, *in the specialized program. The actual implementation applies a rather crude approach to memoization:*

```
let fn' = fn ++ show sas
```

That is, it uses the string representation of the list of static arguments as part of the fabricated function name. For instance, power'a *would be rendered as* "power[(\"n\", Left 3)]"*. Revise the specializer so that readable (short) function names, as assumed in the illustration, are indeed fabricated.*

As a hint at a more interesting partial evaluation scenario, consider the following problem related to the language of finite state machines (FSML), as discussed in detail in Chapter 2. We would like to apply partial evaluation in the context of model-driven development such that partial evaluation of a model interpreter for a statically known model (an FSM) provides us with a code generator. That is, a program specializer, when applied to the model interpreter with a static FSM and a dynamic event sequence, creates essentially a compiled version of the model [21]. The tutorial notes [8] describe the development of a sufficiently powerful specializer for an FSML-like interpreter.

The present section is summarized by means of a recipe.

Recipe 12.4 (Design of a partial evaluator).

Interpreter *Pick a regular interpreter (Recipe 5.1) from which to start.*

Test cases *Set up test cases for the partial evaluator. A positive test case consists of a program (to be partially or totally evaluated), the input of the program, the partitioning thereof in terms of what parts are known versus unknown to the partial evaluator, the output of the program, and the partially evaluated program.*

Code domains *Extend the domains used by the regular interpreter, specifically those for results, to be able to represent code.*

Code generation *Complement the regular interpreter by extra cases that cover unknown input ("dynamic variables"). That is, when subexpressions cannot be evaluated to apply the regular interpreter's operations (e.g., if-then-else or addition), the corresponding expression is reconstructed from the recursively specialized subexpressions. Memoization is needed to avoid infinite code generation.*

Testing *Validate each test case as follows: the regular interpreter computes the expected output from the given program and the input; the partial evaluator computes the partially evaluated program from the given program and the part of the input known to the partial evaluator; and the regular interpreter computes the expected output from the partially evaluated program and the remaining input.*

12.5 Abstract Interpretation

Abstract interpretation is a semantics-based technique for *program analysis*. The expectation is that any such analysis will soundly predict the runtime behavior at some level of abstraction. We will describe two analyses by abstract interpretation: a form of type checking (Chapter 9), and "sign detection" for program variables to be used in program optimization. We will use denotational semantics or denotational style interpreters (Chapter 11) as a starting point for abstract interpreters, because the underlying compositional scheme of mapping syntax to semantics makes it easy to replace systematically the semantic domains of a standard semantics and the corresponding combinators by versions that serve the purpose of a specific program analysis.

12.5.1 Sign Detection as an Optimization Scenario

We would like to optimize imperative programs on the basis of knowing just the signs but not the precise values of some program variables. Here we assume that signs may be determined by a static analysis for "sign detection". The following program is amenable to such an optimization.

Illustration 12.24 (A program with an optimization opportunity)

BIPL resource languages/BIPL/samples/abs.bipl

```
{
  ...
  y = x * x + 42;
  if (y < 0)
    y = −y;
  ...
}
```

The basic laws of arithmetic suggest that the variable y in this program must be positive by the time it is tested by the condition of the if-statement. Thus, the condition must evaluate to false, which implies that the then-branch will never be executed. On the basis of such a static analysis, i.e., without knowing the exact input x, the program could be optimized. In this example, we consider the signs *Pos*, *Neg*, and *Zero* of variables in the program as properties of interest for abstract interpretation. Compare this with the standard semantics, where we care about actual numbers stored in variables. We can calculate on signs pretty much like on numbers, as illustrated by the following function tables for the arithmetic and comparison operators on signs:

$*$	Neg	Zero	Pos	?
Neg	Pos	Zero	Neg	?
Zero	Zero	Zero	Zero	Zero
Pos	Neg	Zero	Pos	?
?	?	Zero	?	?

$+$	Neg	Zero	Pos	?
Neg	Neg	Neg	?	?
Zero	Neg	Zero	Pos	?
Pos	?	Pos	Pos	?
?	?	?	?	?

$<$	Neg	Zero	Pos	?
Neg	?	True	True	?
Zero	False	False	True	?
Pos	False	False	?	?
?	?	?	?	?

In these tables, we use "?" to denote that the sign or truth value of an operand or result has not been assigned. For the program in Illustration 12.24, we can assign sign *Pos* to `y` because, for all possible signs of `x`, the result of `x*x` has sign *Pos* or *Zero* and thus the addition of `42` implies sign *Pos* for `x*x+42`. Hence, the condition must evaluate to `False` and the code in the then-branch is dead.

Abstract interpretation has found many application areas; see, for example, [13] for an application to refactoring and [12] for an application to grammar analysis and parsing. This section relies completely on representing abstract interpretation in Haskell, as opposed to using any semiformal notation for semantics or analysis. The development will be cursory and pragmatic overall. A thorough development can be found elsewhere [58, 57]. Also, Cousot & Cousot's line of seminal work on the subject may be consulted; see [11] for their first paper on the subject.

12.5.2 Semantic Algebras

An abstract interpreter can be seen as a variation on a regular interpreter where semantic domains and combinators are defined differently. In order to be able to explore such a variation in an effective manner, we revise a denotational interpreter so that it is parameterized in the semantic domains and combinators. This is done here for an imperative programming language (BIPL) and its direct-style denotational semantics.

That is, we aim at factoring out the algebra (an abstract data type) of meanings; we use the term "semantic algebra". We begin by identifying the corresponding signature as shown below.

Illustration 12.25 (Signature of semantic algebras)

Haskell module Language.BIPL.Algebra.Signature

```
-- Aliases to shorten function signatures
type Trafo sto = sto → sto -- Store transformation
type Obs sto val = sto → val -- Store observation
-- The signature of algebras for interpretation
data Alg sto val = Alg {
  skip' :: Trafo sto,
  assign' :: String → Obs sto val → Trafo sto,
  seq' :: Trafo sto → Trafo sto → Trafo sto,
  if' :: Obs sto val → Trafo sto → Trafo sto → Trafo sto,
  while' :: Obs sto val → Trafo sto → Trafo sto,
  intconst' :: Int → Obs sto val,
  var' :: String → Obs sto val,
  unary' :: UOp → Obs sto val → Obs sto val,
  binary' :: BOp → Obs sto val → Obs sto val → Obs sto val
}
```

That is, the signature is defined as a record type Alg. The record type carries one member for each language construct. There are type parameters sto and val for stores and values. These type parameters enable different type definitions for concrete and abstract interpreters and, in fact, for different abstract interpreters implementing different program analyses.

Given an actual algebra of the signature, an interpreter (an analysis) can be defined by simply recursing into program phrases and combining the intermediate meanings according to the operations of the algebra, as shown below.

Illustration 12.26 (The compositional scheme)

Haskell module Language.BIPL.Algebra.Scheme

```
interpret :: Alg sto val → Stmt → sto → sto
interpret a = execute
  where
    −− Compositional interpretation of statements
    execute Skip = skip' a
    execute (Assign x e) = assign' a x (evaluate e)
    execute (Seq s1 s2) = seq' a (execute s1) (execute s2)
    execute (If e s1 s2) = if' a (evaluate e) (execute s1) (execute s2)
    execute (While e s) = while' a (evaluate e) (execute s)
    −− Compositional interpretation of expressions
    evaluate (IntConst i) = intconst' a i
    evaluate (Var n) = var' a n
    evaluate (Unary o e) = unary' a o (evaluate e)
    evaluate (Binary o e1 e2) = binary' a o (evaluate e1) (evaluate e2)
```

The interpreter is equivalent to the earlier direct-style denotational interpreter (Illustration 11.2), except that the semantic combinators are not functions in scope, but instead are looked up as record components from the argument algebra a. Thus, interpretation is completely parametric at this stage.

12.5.3 Concrete Domains

The "standard semantics" can now be simply represented as a specific algebra – a record; we also speak of "concrete domains". The record components, as shown below, directly correspond to the top-level functions modeling semantic combinators, as defined in the underlying denotational interpreter (Illustration 11.3).

Illustration 12.27 (An algebra for interpretation)

Haskell module Language.BIPL.Algebra.StandardInterpreter

```
1  type Value = Either Int Bool
2  type Store = Map String Value
3  algebra :: Alg Store Value
4  algebra = a where a = Alg {
5    skip' = id,
6    assign' = λ n f m → insert n (f m) m,
7    seq' = flip (.),
8    if' = λ f g h m → let (Right b) = f m in if b then g m else h m,
9    while' = λ f g → fix (λ x → if' a f (seq' a g x) (skip' a)),
10   intconst' = λ i → const (Left i),
11   var' = λ n m → m!n,
12   unary' = λ o f m →
13     case (o, f m) of
14       (Negate, Left i) → Left (negate i)
15       (Not, Right b) → Right (not b),
16   binary' = λ o f g m → ...
17 }
```

Thus, the algebra commits to the sum of Int and Bool for values (line 1), and to maps from strings to values for stores (line 2), and it designates the usual operations for combining meanings. For instance, if-statements are eventually handled by a dispatch on a condition's two possible values, True and False (line 8).

12.5.4 Abstract Domains

An abstract interpretation devises abstract domains to analyze programs statically, as opposed to a description of the precise semantics in terms of its so-called concrete domains. For instance, an abstract interpretation for type checking would use abstract domains as follows:

```
data Type = IntType | BoolType −− Instead of values
type VarTypes = Map String Type −− Instead of stores
```

That is, abstract interpretation should compute variable-to-type maps as opposed to proper stores, i.e., variable-to-value maps. The idea is then that the semantic combinators on abstract domains are defined similarly to those for the concrete domains. In algebraic terms, we use (chain-) complete partial orders (CCPO or CPO). An abstract interpretation for sign detection would use abstract domains as follows:

```
data Sign = Zero | Pos | Neg | BottomSign | TopSign
data CpoBool = ProperBool Bool | BottomBool | TopBool
type Property = Either Sign CpoBool
type VarProperties = Map String Property
```

The key type is Sign, with constructors Zero, Pos, Neg for different signs of numbers. The type abstracts from the Int type used in the standard interpreter. The type Sign features additional constructors BottomSign and TopSign as least and greatest elements, which are needed for technical reasons. BottomSign ($\bot$) proxies for the analysis not having identified the sign yet. TopSign ($\top$) proxies for the analysis having failed to identify the sign. The type CpoBool adds least and greatest elements to Haskell's Bool. The type Property is a sum over Sign and CpoBool, and it thus abstracts from Value as a sum over Int and Bool in the standard interpreter. The type VarProperties abstracts from Store in the concrete interpreter, i.e., it maps variables to abstract values ("properties") rather than concrete values.

Let us take a closer look at the abstract domain for signs. We provide an implementation as follows.

Illustration 12.28 (Signs of numbers)

Haskell module Data.CPO.Sign

```haskell
data Sign = Zero | Pos | Neg | BottomSign | TopSign

instance Num Sign
 where
  fromInteger n
    | n > 0 = Pos
    | n < 0 = Neg
    | otherwise = Zero

  TopSign + _ = TopSign
  _ + TopSign = TopSign
  BottomSign + _ = BottomSign
  _ + BottomSign = BottomSign
  Zero + Zero = Zero
  Zero + Pos = Pos
  ...

instance CPO Sign where
  pord x y | x == y = True
  pord BottomSign _ = True
  pord _ TopSign = True
  pord _ _ = False
  lub x y | x == y = x
  lub BottomSign x = x
  lub x BottomSign = x
  lub _ _ = TopSign

instance Bottom Sign where
  bottom = BottomSign
```

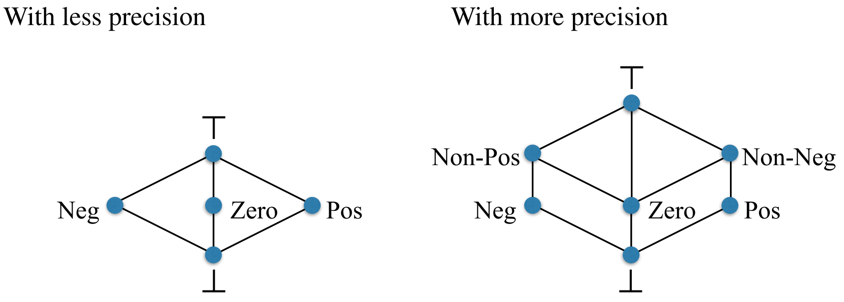


Fig. 12.5 Two options for an abstract domain of signs.

The excerpts given here illustrate the following aspects of signs:

- Signs are “abstract” numbers; Haskell’s library type class Num is instantiated for type Sign (lines 3–16), paralleling the standard instance for type Int. The type-class member fromInteger (lines 5–8) is the explicit manifestation of abstraction: integers are mapped to signs. Further, we hint at the addition operation on signs (lines 10–15). Several other operations on signs have been omitted for brevity.
- Signs form a partial order, and there are least and greatest elements; see the instances of dedicated type classes CPO and Bottom (lines 18–29). In Fig. 12.5, we show two options for a partial order on signs with different degrees of precision; the algebraic data type shown earlier corresponds to the less precise option on the left. We use Hasse diagrams for illustration. The idea is that the least element $\perp$ is the initial element for any sort of approximative, fixed point-based analysis (see below for details), whereas the greatest element $\top$ is the indicator of failure of analysis. We show two options for the abstract domain of signs in the figure because we want to indicate that one can make a trade-off in abstract interpretation or program analysis more generally, in terms of precision of results versus time and space complexity required.

For an abstract interpretation to be sound with regard to a given standard semantics and to enable effective computation of uniquely defined fixed points, various properties have to be satisfied by the abstract domains [57, 73], which we mention here only in passing. In general, abstract domains need to define chain-complete partial orders (ccpos). Further, there also needs to be a mapping from concrete to abstract domains (see fromInteger above) such that partial orders are preserved and homomorphisms are defined, i.e., mapping concrete values to abstract values and then combining abstract values with the abstract semantic combinators equals combining concrete values and mapping the result.

Exercise 12.15 (More discriminative signs) [Basic level]
Implement the more precise option shown in Fig. 12.5.

12.5.5 Examples of Abstract Interpreters

Let us work out the abstract interpreters for type checking and sign detection.

12.5.5.1 A Type-Checking Interpreter

The interpreter needs to compute types instead of values for expressions, and it computes variable-type pairs (in fact, a map) for statements. Clearly, type checking can fail, which happens when the given program or some phrase of it does not type-check. As discussed earlier (Chapter 9), we do not want the type checker to "throw" in the Haskell sense. Instead, we expect to observe failure of type checking on the basis of wrapping the result types of the type checker in the Maybe monad.

Here is how we expect to use the type checker:

Interactive Haskell session:

```
▶ interpret TypeChecker.algebra euclideanDiv (fromList [("x", IntType), ("y", IntType)])
Just (fromList [("q", IntType), ("r", IntType), ("x", IntType), ("y", IntType)])
```

That is, we type-check the sample program for Euclidean division We supply enough type context for the arguments x and y. Type checking infers that the program variables q and r are of type IntType.

The simple signature introduced above (Illustration 12.25) does not give us control to add Maybe to the result types in semantic domains. We need a more general, monadic signature as follows:

Illustration 12.29 (Monadic semantic algebras)

Haskell module Language.BIPL.MonadicAlgebra.Signature

```
-- Aliases to shorten function signatures
type Trafo m sto = sto → m sto -- Store transformation
type Obs m sto val = sto → m val -- Store observation
-- The signature of algebras for interpretation
data Alg m sto val = Alg {
  skip' :: Trafo m sto,
  assign' :: String → Obs m sto val → Trafo m sto,
  seq' :: Trafo m sto → Trafo m sto → Trafo m sto,
  ...
}
```

That is, the type synonyms Trafo and Obs at the top (lines 1–2) for transformers and observers wrap the result in a type constructor m. The compositional scheme for monadic algebras is the same as for non-monadic ones (Illustration 12.26). We define an algebra for type checking as follows.

Illustration 12.30 (An algebra for type checking)

Haskell module Language.BIPL.MonadicAlgebra.TypeChecker

```
 1  data Type = IntType | BoolType
 2  type VarTypes = Map String Type
 3  algebra :: Alg Maybe VarTypes Type
 4  algebra = Alg {
 5    skip' = Just,
 6    assign' = λ x f m → f m >>=λ t →
 7      case lookup x m of
 8        (Just t') → guard (t==t') >> Just m
 9        Nothing → Just (insert x t m),
10    seq' = flip (<=<),
11    if' = λ f g h m → do
12      t ← f m
13      guard (t==BoolType)
14      m1 ← g m
15      m2 ← h m
16      guard (m1==m2)
17      Just m1,
18    while' = λ f g m → do
19      t ← f m
20      guard (t==BoolType)
21      m' ← g m
22      guard (m==m')
23      Just m,
24    intconst' = const (const (Just IntType)),
25    var' = λ x m → lookup x m,
26    unary' = λ o f m → f m >>=λ t →
27      case (o, t) of
28        (Negate, IntType) → Just IntType
29        (Not, BoolType) → Just BoolType
30        _ → Nothing,
31    binary' = λ o f g m → ...
32  }
```

We discuss the first few record components as follows:

skip′ The transformer returns the given variable-type map as is (line 5).

assign′ The right-hand side expression is type-checked and its type, if any, is bound to t (line 6). The variable-type map m is consulted to see whether or not the left-hand side variable x has an associated type (line 7). If there is a type, it must be equal to t (line 8); otherwise, the map is adapted (line 9).

seq′ Flipped monadic function composition composes type checking for the two statements (line 10).

if′ The condition is type-checked and its type, if any, is bound to t (line 12); the guard checks that t equals BoolType (line 13). The then- and else-branches are type-checked for the same input map m and the result maps, if any, are bound to m1 and m2 (lines 14–15). The two maps are checked to be equal; this is one sound option for if-statements (Section 9.7.1) (line 16), and either of them (in fact, m1) is finally returned (line 17).

Exercise 12.16 (Fixed-point semantics of while-loops) [Basic level]
Why does the semantic combinator while' *for type checking not involve a fixed-point computation, whereas it does in the case of standard semantics?*

Exercise 12.17 (Algebras with monadic binding) [Intermediate level]
Would it make sense to factor out monadic binding, as exercised extensively in the algebra above, into the compositional scheme?

12.5.5.2 A Sign-Detection Interpreter

Let us set up a test case for sign detection first. We pick a program that involves a while-loop so that we are bound to also discuss the intricacies of fixed-point semantics. In fact, we offer two variations on a program computing the factorial of an argument x in the hope that sign detection works equally well for these variations; see below.

Illustration 12.31 (A program for the factorial (V1))

BIPL resource languages/BIPL/samples/factorialV1.bipl

```
// Assume x to be positive
y = 1;
i = 1;
while (i <= x) {
  y = y * i;
  i = i + 1;
}
```

Illustration 12.32 (A program for the factorial (V2))

BIPL resource languages/BIPL/samples/factorialV2.bipl

```
// Assume x to be positive
y = 1;
while (x >= 2) {
  y = y * x;
  x = x − 1;
}
```

We want sign detection to infer that the variable y is positive after execution of the program. As we will see, the minor idiomatic differences between the two variants will cause a challenge for the program analysis. That is, an initial, more straightforward version of the analysis will fail to predict the sign of y in the second variant. A refined, more complex version will succeed, though.

Here is how we expect the program analysis for sign detection to work:

Interactive Haskell session:

```
▶ interpret analysis facv1 (fromList [("x", Left Pos)])
fromList [("i", Left Pos), ("x", Left Pos), ("y", Left Pos)]
```

That is, applying the analysis to the first variant of the factorial code (facv1) and setting up x with sign Pos as a precondition, we find that y has sign Pos after program execution; the signs for the other program variables also make sense.

We define an algebra for sign detection as follows.

Illustration 12.33 (An algebra for sign detection)

Haskell module Language.BIPL.Analysis.BasicAnalysis

```
1  type Property = Either Sign CpoBool
2  type VarProperties = Map String Property
3  algebra :: Alg VarProperties Property
4  algebra = a where a = Alg {
5    skip' = id,
6    assign' = λ n f m → insert n (f m) m,
7    seq' = flip (.),
8    if' = λ f g h m →
9      let Right b = f m in
10       case b of
11         (ProperBool True) → g m
12         (ProperBool False) → h m
13         BottomBool → bottom
14         TopBool → g m `lub` h m,
15   while' = λ f g → fix' (λ x → if' a f (x . g) id) (const bottom),
16   intconst' = λ i → const (Left (fromInteger (toInteger i))),
17   var' = λ n m → m!n,
18   unary' = λ o f m →
19     case (o, f m) of
20       (Negate, Left s) → Left (negate s)
21       (Not, Right b) → Right (cpoNot b),
22   binary' = λ o f g m → ...
23 }
```

We discuss the semantic combinators one by one as follows:

skip′, assign′, seq′ These cases (lines 5–7) are handled in the same manner as in the standard semantics. That is, it does not matter if we operate on concrete or abstract stores when it comes to the empty statement, assignment, and sequential composition.

if′ There is a case discrimination with respect to the truth value computed for the condition of the if-statement (lines 8–14); the first two cases correspond to those also present in the standard semantics (True and False, lines 11–12). The third case (line 13) applies when the truth value is not (yet) defined, in which case the resulting variable-property map is also undefined, i.e., bottom. The fourth case (line 14) applies when the truth value is "over-defined", i.e., the analysis cannot derive any precise value, in which case the least upper bound of the variable-property maps for the then- and else-branches are computed. This is discussed in more detail below.

while′ A fixed point is computed in a manner similar to the standard semantics (line 15), except that a different fixed-point combinator fix' is assumed here, which aims at finding a fixed point computationally and effectively by starting from an initial approximation bottom. This is discussed in more detail below.

intconst′ The constant is mapped to a sign by plain abstraction (line 16), i.e., the Int is first mapped to an Integer so that the fromInteger member of the type class Num, as discussed above (Illustration 12.28), can be applied.

var′ This case (line 17) is handled in the same manner as in the standard semantics.

unary′ and binary′ Just as in the standard semantics, operations are applied to the arguments, except that we operate on the abstract domains here: Sign and CpoBool.

One crucial aspect is the treatment of if-statements in the case where the truth value for the condition cannot be precisely determined. In this case, the analysis approximates the resulting property-type map by simply assuming that either of the two branches may be executed and, thus, the least upper bound (LUB) of the two branches is taken. Here we rely on LUBs on maps to be defined in a pointwise manner as follows.

Illustration 12.34 (Partial order on maps with pointwise LUB)

Haskell module Data.CPO.Map

```
instance (Ord k, CPO v) => CPO (Map k v) where
  pord x y = and (map (f y) (toList x))
    where f y (k,v) = pord v (y!k)
  lub x y = foldr f y (toList x)
    where f (k,v) m = Data.Map.insert k (lub v (y!k)) m
instance (Ord k, CPO v) => Bottom (Map k v) where
  bottom = empty
```

If the two branches of an if-statement disagree in the sign of a variable (Pos versus Neg versus Zero), then the combined value is Top. This causes a precision challenge, to be discussed in a second.

Another crucial aspect is the treatment of while-loops in terms of the required fixed-point computation. We a dedicated fixed-point combinator as follows.

Illustration 12.35 (Fixed-point computation with iterands)

Haskell module Language.BIPL.Analysis.Fix

```
fix' :: Eq a => ((a → a) → a → a) → (a → a) → a → a
fix' h i x = limit (iterate h i)
  where limit (b1:b2:bs) = if b1 x == b2 x then b1 x else limit (b2:bs)
```

This combinator is polymorphic, but let us explain it in terms of the abstract domains at hand. The type variable a proxies for abstract stores (i.e., type VarProperties). Thus, the combinator, in this instance, returns an abstract store transformer. It is parameterized in the "transformer transformer" h, from which we can take a fixed point, an initial transformer i to start off the iteration, and an abstract store x. With iterate, the combinator builds an infinite list by applying h to i 0, 1, 2, ... number of times. With the local helper limit, the combinator finds the position in the list such that the applications of two consecutive elements b1 and b2 to x are the same; b1 is thus the fixed point, i.e., the store transformer for the while-loop.

We are now ready to apply the program analysis:

Interactive Haskell session:

```
▶ interpret BasicAnalysis.algebra facv1 (fromList [("x", Left Pos)])
fromList [("i", Left Pos), ("x", Left Pos), ("y", Left Pos)]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ interpret BasicAnalysis.analysis facv2 (fromList [("x", Left Pos)])
fromList [("x", Left TopSign), ("y", Left TopSign)]
```

Our analysis finds the expected sign for the first variant (Illustration 12.31); it fails for the second variant (Illustration 12.32), as it reports TopSign for y. Generally, program analysis (whether based on abstract interpretation or not) may be challenged by such precision issues. In this particular case, the origin of the problem lies in the handling of if-statements. The abstract store transformer for an if-statement will assign TopSign all too easily to variables whenever the condition evaluates to TopBool which may happen easily. Consider the second variant again; variable x is decremented in the loop body and, by the rules for signs, the sign of x is going to be TopSign. Thus, the truth value of the loop's condition is going to be TopBool.

Let us hint at a possible improvement. Given the abstract input store m for the if-statement, we do not simply compute the abstract output stores for the two branches from the given map m and combine them, but instead determine a smaller abstract store to serve as the input for each branch. For the then-branch, we determine the largest store such that the condition evaluates to True, and likewise for the else-branch. We revise the algebra accordingly as follows.

Illustration 12.36 (A refined algebra for sign detection)

Haskell module Language.BIPL.Analysis.RefinedAnalysis

```
1  algebra :: Alg VarProperties Property
2  algebra = a where a = Alg {
3    ...
4    if’ = λ f g h m →
5      let Right b = f m in
6        case b of
7          (ProperBool True) → g m
8          (ProperBool False) → h m
9          BottomBool → bottoms m
10         TopBool → g (feasible True f m) `lub` h (feasible False f m)
11           where feasible b f m = lublist (bottoms m) [ m' |
12                     m' ← maps (keys m),
13                     m' `pord` m,
14                     Right (ProperBool b) `pord` f m' ],
15   ...
16 }
```

The refinement concerns the TopBool case for if’ (lines 10–14). That is, g and h are not directly applied to m, as before. Instead, a variable-property map is computed for each branch by the function feasible. This function generates all maps m' such that they assign properties to the same variables as in m (line 12), they are less defined than or equally defined as m (line 13), and they permit the evaluation of the condition f for the given Boolean value b (line 14).

We apply the refined program analysis as follows:

Interactive Haskell session:

```
▶ interpret RefinedAnalysis.algebra facv1 (fromList [("x", Left Pos)])
fromList [("i", Left Pos), ("x", Left Pos), ("y", Left Pos)]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ interpret RefinedAnalysis.algebra facv2 (insert "x" (Left Pos) empty)
fromList [("x", Left TopSign), ("y", Left Pos)]
```

Thus, we infer sign Pos for y for both variants of the program now. Nevertheless, there is room for improvement, as illustrated by the following exercises.

Exercise 12.18 (More precise domains) [Intermediate level]

As the analysis stands, when applied to the second variant of the factorial program, it maps x *to* Top. *Refine the analysis so that it can make a more precise prediction. This exercise may require an adaptation of the abstract domains for signs.*

Exercise 12.19 (Precise squaring) [Intermediate level]
The present section started with an example in which the sign of an input variable x *was not fixed (Illustration 12.24) and, thus,* BottomSign *should be assigned to* x*. The analysis, as it stands, would be too imprecise to serve for this example. For instance, consider squaring* x*. Per the rules we would assign* BottomSign*BottomSign = BottomSign *to* x*x*, even though we "know" that the result of squaring a number* x *has sign* Zero *or* Pos *no matter what the sign of* x*. Refine the analysis accordingly.*

The present section is summarized by means of a recipe.

Recipe 12.5 (Design of an abstract interpreter).

Interpreter *Pick a compositional interpreter (Recipe 11.1) from which to start.*
Abstract domains *Define the abstract domains as abstraction from the concrete domains in the underlying interpreter so that the properties of interest (e.g., signs) are modeled. Abstract domains include bottom ($\perp$ = "undefined") and top ($\top$ = "overdefined") as abstract values, subject to a partial order for undefinedness.*
Test cases *Set up test cases for the abstract interpreter. A test case consists of an input term and the expected result of the analysis, thereby exercising the abstract domains. A positive test case does not exercise top for the result. Explore the precision the analysis by also incorporating negative test cases, as variations on positive test cases, for which the analysis returns top.*
Parameterization *Factor the compositional interpreter to become parametric in the semantic domains and semantic combinators, thereby enabling the use of the abstract domains for the analysis at hand.*
Testing *Test the abstract interpreter in terms of the test cases.*

Summary and Outline

We have presented term rewriting and attribute grammars as computational paradigms for developing certain kinds of metaprograms in a more disciplined way. Term rewriting fits well with rule-based transformational problems, for example, refactoring and optimization. Attribute grammars fit well with tree-annotation problems and, specifically, with translations and analyses. We have also presented (compile-time) multi-stage programming and partial evaluation as powerful program optimization techniques and abstract interpretation as a semantics-based method for analysis. Multi-stage programming serves the purpose of program generation by allowing one to derive optimized code based on appropriately "quasi-quoted code and splices". Partial evaluation serves the purpose of program specialization by allowing one to derive optimized code based on an appropriately refined interpreter, which, in the case we considered, combines function inlining and memoization for arguments. Abstract interpretation may serve, for example, the purpose of optimization.

This ends the technical development presented in this book. We will now wrap up the book in the Postface.

References

1. Adams, M.: Towards the essence of hygiene. In: Proc. POPL, pp. 457–469. ACM (2015)
2. Alblas, H.: Attribute evaluation methods. In: Proc. SAGA, *LNCS*, vol. 545, pp. 48–113. Springer (1991)
3. Alblas, H., Melichar, B. (eds.): Attribute Grammars, Applications and Systems, International Summer School SAGA, 1991, Proceedings, *LNCS*, vol. 545. Springer (1991)
4. Bagge, A.H., Lämmel, R.: Walk your tree any way you want. In: Proc. ICMT, *LNCS*, vol. 7909, pp. 33–49. Springer (2013)
5. Boyland, J.T.: Remote attribute grammars. J. ACM **52**(4), 627–687 (2005)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1-2), 52–70 (2008)
7. Carette, J., Kiselyov, O., Shan, C.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. J. Funct. Program. **19**(5), 509–543 (2009)
8. Cook, W.R., Lämmel, R.: Tutorial on online partial evaluation. In: Proc. DSL, *EPTCS*, vol. 66, pp. 168–180 (2011)
9. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3), 190–210 (2006)
10. Cordy, J.R.: Excerpts from the TXL cookbook. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 27–91. Springer (2011)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL, pp. 238–252. ACM (1977)
12. Cousot, P., Cousot, R.: Grammar semantics, analysis and parsing by abstract interpretation. Theor. Comput. Sci. **412**(44), 6135–6192 (2011)
13. Cousot, P., Cousot, R., Logozzo, F., Barnett, M.: An abstract interpretation framework for refactoring with application to extract methods with contracts. In: Proc. OOPSLA, pp. 213–232. ACM (2012)
14. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–646 (2006)
15. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers, *LNCS*, vol. 3016, pp. 51–72. Springer (2004)
16. Dershowitz, N.: A taste of rewrite systems. In: Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991-1992, McMaster University, Hamilton, Ontario, Canada, *LNCS*, vol. 693, pp. 199–228. Springer (1993)
17. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Handbook of Theoretical Computer Science B: Formal Methods and Semantics, pp. 243–320. North-Holland (1990)
18. van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symb. Comput. **15**(5/6), 523–545 (1993)

19. Erdweg, S.: Extensible languages for flexible and principled domain abstraction. Ph.D. thesis, Philipps-Universität Marburg (2013)
20. Erwig, M., Ren, D.: Monadification of functional programs. Sci. Comput. Program. **52**, 101–129 (2004)
21. Futamura, Y.: Partial evaluation of computation process — an approach to a compiler-compiler. Higher Order Symbol. Comput. **12**, 381–391 (1999)
22. Hatcliff, J.: Foundations of partial evaluation and program specialization (1999). Available at `http://people.cis.ksu.edu/~hatcliff/FPEPS/`
23. Heckel, R.: Graph transformation in a nutshell. ENTCS **148**(1), 187–198 (2006)
24. Hedin, G.: An overview of door attribute grammars. In: Proc. CC, *LNCS*, vol. 786, pp. 31–51. Springer (1994)
25. Hedin, G.: An introductory tutorial on JastAdd attribute grammars. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 166–200. Springer (2011)
26. Hudak, P.: Modular domain specific languages and tools. In: Proc. ICSR, pp. 134–142. IEEE (1998)
27. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice-Hall, Inc. (1993)
28. de Jonge, M.: Pretty-printing for software reengineering. In: Proc. ICSM 2002, pp. 550–559. IEEE (2002)
29. de Jonge, M., Visser, E.: An algorithm for layout preservation in refactoring transformations. In: Proc. SLE 2011, *LNCS*, vol. 6940, pp. 40–59. Springer (2012)
30. Jonnalagedda, M., Coppey, T., Stucki, S., Rompf, T., Odersky, M.: Staged parser combinators for efficient data processing. In: Proc. OOPSLA, pp. 637–653. ACM (2014)
31. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1-2), 31–39 (2008)
32. Kastens, U., Pfahler, P., Jung, M.T.: The Eli system. In: Proc. CC, *LNCS*, vol. 1383, pp. 294–297. Springer (1998)
33. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Inf. **31**(7), 601–627 (1994)
34. Kiselyov, O.: Typed tagless final interpreters. In: Generic and Indexed Programming – International Spring School, SSGIP 2010, Revised Lectures, *LNCS*, vol. 7470, pp. 130–174. Springer (2012)
35. Kiselyov, O.: The design and implementation of BER MetaOCaml – System description. In: Proc. FLOPS, *LNCS*, vol. 8475, pp. 86–102. Springer (2014)
36. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proc. SCAM, pp. 168–177. IEEE (2009)
37. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 222–289. Springer (2011)
38. Klonatos, Y., Koch, C., Rompf, T., Chafi, H.: Building efficient query engines in a high-level language. PVLDB **7**(10), 853–864 (2014)
39. Klop, J.W.: Term rewriting systems. In: Handbook of Logic in Computer Science, pp. 1–117. Oxford University Press (1992)
40. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2), 127–145 (1968)
41. Kort, J., Lämmel, R.: Parse-tree annotations meet re-engineering concerns. In: Proc. SCAM, pp. 161–168. IEEE (2003)
42. Lämmel, R.: Declarative aspect-oriented programming. In: Proc. PEPM, pp. 131–146. University of Aarhus (1999)
43. Lämmel, R.: Reuse by program transformation. In: Selected papers SFP 1999, *Trends in Functional Programming*, vol. 1, pp. 144–153. Intellect (2000)
44. Lämmel, R.: Scrap your boilerplate – Prologically! In: Proc. PPDP, pp. 7–12. ACM (2009)
45. Lämmel, R., Jones, S.L.P.: Scrap your boilerplate: a practical design pattern for generic programming. In: Proc. TLDI, pp. 26–37. ACM (2003)
46. Lämmel, R., Jones, S.L.P.: Scrap more boilerplate: reflection, zips, and generalised casts. In: Proc. ICFP, pp. 244–255. ACM (2004)

47. Lämmel, R., Jones, S.L.P.: Scrap your boilerplate with class: extensible generic functions. In: Proc. ICFP, pp. 204–215. ACM (2005)
48. Lämmel, R., Riedewald, G.: Reconstruction of paradigm shifts. In: Proc. WAGA, pp. 37–56 (1999). INRIA Technical Report
49. Lämmel, R., Thompson, S.J., Kaiser, M.: Programming errors in traversal programs over structured data. Sci. Comput. Program. **78**(10), 1770–1808 (2013)
50. Lämmel, R., Visser, E., Visser, J.: The essence of strategic programming (2002). 18 p.; Unpublished draft; Available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.198.8985&rep=rep1&type=pdf
51. Lämmel, R., Visser, J.: Typed combinators for generic traversal. In: Proc. PADL, *LNCS*, vol. 2257, pp. 137–154. Springer (2002)
52. Lämmel, R., Visser, J.: A Strafunski application letter. In: Proc. PADL, *LNCS*, vol. 2562, pp. 357–375. Springer (2003)
53. Landauer, C., Bellman, K.L.: Generic programming, partial evaluation, and a new programming paradigm. In: Proc. HICSS-32. IEEE (1999)
54. Lengauer, C., Taha, W. (eds.): Special issue on the first MetaOCaml Workshop 2004. Sci. Comput. Program. (2006)
55. McNamee, D., Walpole, J., Pu, C., Cowan, C., Krasic, C., Goel, A., Wagle, P., Consel, C., Muller, G., Marlet, R.: Specialization tools and techniques for systematic optimization of system software. ACM Trans. Comput. Syst. **19**(2), 217–251 (2001)
56. Mens, T.: Model Transformation: A Survey of the State of the Art, pp. 1–19. John Wiley & Sons, Inc. (2013)
57. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, corrected 2nd printing edn. Springer (2004)
58. Nielson, H.R., Nielson, F.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science. Springer (2007)
59. Porkoláb, Z., Sinkovics, Á., Siroki, I.: DSL in C++ template metaprogram. In: CEFP 2013, Revised Selected Papers, *LNCS*, vol. 8606, pp. 76–114. Springer (2015)
60. Rebernak, D., Mernik, M., Henriques, P.R., Pereira, M.J.V.: AspectLISA: An aspect-oriented compiler construction system based on attribute grammars. ENTCS **164**(2), 37–53 (2006)
61. Renggli, L.: Dynamic language embedding with homogeneous tool support. Ph.D. thesis, Universität Bern (2010)
62. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: Proc. ECOOP, *LNCS*, vol. 6183, pp. 380–404. Springer (2010)
63. Rompf, T.: The essence of multi-stage evaluation in LMS. In: A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, *LNCS*, vol. 9600, pp. 318–335. Springer (2016)
64. Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-Virtualized: linguistic reuse for deep embeddings. Higher Order Symbol. Comput. **25**(1), 165–207 (2012)
65. van Rozen, R., van der Storm, T.: Origin tracking + + text differencing = = textual model differencing. In: Proc. ICMT, *LNCS*, vol. 9152, pp. 18–33. Springer (2015)
66. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publishing Company (1997). Volume 1: Foundations
67. Seefried, S., Chakravarty, M.M.T., Keller, G.: Optimising embedded DSLs using Template Haskell. In: Proc. GPCE, *LNCS*, vol. 3286, pp. 186–205. Springer (2004)
68. Shali, A., Cook, W.R.: Hybrid partial evaluation. In: Proc. OOPSLA, pp. 375–390. ACM (2011)
69. Sheard, T., Peyton Jones, S.L.: Template meta-programming for Haskell. SIGPLAN Not. **37**(12), 60–75 (2002)
70. Siek, J.G., Taha, W.: A semantic analysis of C++ templates. In: Proc. ECOOP, *LNCS*, vol. 4067, pp. 304–327. Springer (2006)
71. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure embedding of attribute grammars. Sci. Comput. Program. **78**(10), 1752–1769 (2013)
72. Smaragdakis, Y.: Structured program generation techniques. In: GTTSE 2015, Revised Papers, *LNCS*, vol. 10223, pp. 154–178. Springer (2017)

73. Stoltenberg-Hansen, V., Lindström, I., Griffor, E.R.: Mathematical Theory of Domains. Cambridge University Press (1994)
74. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embedded Comput. Syst. **13**(4), 1–25 (2014)
75. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and implementing combinator languages. In: Advanced Functional Programming, Third International School, Braga, Portugal, September 12-19, 1998, Revised Lectures, *LNCS*, vol. 1608, pp. 150–206. Springer (1999)
76. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers, *LNCS*, vol. 3016, pp. 30–50. Springer (2004)
77. Taha, W.: A gentle introduction to multi-stage programming, part II. In: GTTSE 2007, Revised Papers, *LNCS*, vol. 5235, pp. 260–290. Springer (2008)
78. Taha, W., Sheard, T.: Multi-stage programming. In: Proc. ICFP, p. 321. ACM (1997)
79. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2), 211–242 (2000)
80. Tisi, M., Mart/´ınez, S., Jouault, F., Cabot, J.: Refining models with rule-based model transformations. Tech. rep., Inria (2011). Research Report RR-7582. pp.18
81. Ulke, B., Steimann, F., Lämmel, R.: Partial evaluation of OCL expressions. In: Proc. MODELS, pp. 63–73. IEEE (2017)
82. van den Brand, M., Sellink, M.P.A., Verhoef, C.: Generation of components for software renovation factories from context-free grammars. Sci. Comput. Program. **36**(2-3), 209–266 (2000)
83. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Sci. Comput. Program. **75**(1-2), 39–54 (2010)
84. Veldhuizen, T.: Template metaprograms. C++ Rep. **7**(4), 36—43 (1995)
85. Visser, E.: A survey of strategies in rule-based program transformation systems. J. Symb. Comput. **40**(1), 831–873 (2005)
86. Visser, E., Benaissa, Z., Tolmach, A.: Building program optimizers with rewriting strategies. In: Proc. ICFP, pp. 13–26. ACM Press (1998)
87. Visser, J.: Visitor combination and traversal control. In: Proc. OOPSLA, pp. 270–282. ACM (2001)
88. Williams, K., Wyk, E.V.: Origin tracking in attribute grammars. In: Proc. SLE, *LNCS*, vol. 8706, pp. 282–301. Springer (2014)