# 2

# Programmable Software Transformation Systems

This chapter gives an overview of the state-of-the-art in architectures and designs for programmable software transformation systems. This is highly warranted because unlike for business systems, compilers and web applications, no books exist which propose best practises for design and implementation of software transformation systems. In fact, even the research literature is to a large extent lacking in such information.

Architectural features and design considerations for these systems are explored using a formal notation called feature models, and further illustrated with examples taken from a careful selection of a dozen concrete research systems. The feature models [Bat05] are used to compare and contrast the design of both architectures and transformation languages. They give a sense of the complexity and breadth of the design space for software transformation systems. Special focus is placed on the program models found in transformation systems, and how these interrelate with the transformation languages.

## 2.1   Software Transformation Systems

A *software transformation system* is an application that takes a *source program* written in a *source language* and transforms this into an *target program* in a *target language*, according to instructions of a *transformation program*, written in a *transformation language*. The source language can be any formal language. What some refer to as (code) generators are included in the definition. In cases where distinguishing between the source and target language is not necessary, the term *subject language* will be used. It is meant to subsume both. The transformation is implemented by a *transformation programmer* and is always designed to preserve certain semantics. The exact semantics to be preserved are specific to the transformation, however. The goal of a transformation $T$ is to reduce some cost $C_m(p)$ of some metric $m$ on a program

$p$: we want $C_m(T(p)) < C_m(p)$, i.e. the transformed program should be "better", according to some metric [PP96, Pai96, CC02].

A traditional application area for software transformation is *transformation-oriented programming* [Par86, Fea87]. In this approach to software development, an executable implementation in the target language is derived mostly automatically from a formal, non-executable specification in the source language. Each transformation step is proved correct, either by only applying transformations guaranteed to preserve the desired semantics, or by manually filling in proof obligations the transformation system cannot automatically resolve. Here, the metric is executability – eventually an executable program is obtained, and the property being preserved is the correctness of the behaviour of the program, with respect to the source specification.

Another important application is *source-to-source* transformations, where the target and source language is the same. Typical applications in this area include program optimisation, where execution speed is the metric; re-engineering, where certain notions of maintainability are used as metrics; and refactoring [Opd92], where (often very loose) metrics for design quality are used. Software transformation techniques and systems have also been used to create compilers, source code documentation systems and program analysers. The survey by Partsch and Steinbrüggen [PS83] contains additional examples of applications for transformation systems.

A note about compilers is warranted. While the general definition above also treats compiler as software transformation systems, the subject of this survey – *programmable* transformation systems – differs from compilers in one crucial aspect: the transformation programmer can extend and adapt the software manipulation facility by supplying new transformations. A programmable software transformation system may be seen as a programming environment built specifically to manipulate programs, i.e. to implement transformation programs. It is therefore more natural to compare programmable transformation systems to compiler construction kits, so-called compiler compilers, rather than directly to compilers. Conceivably, transformation systems could be built directly on top of compilers, however. This is the subject of Chapter 10.

This chapter will show that software transformation systems are available in many variants, ranging from extensions to general purpose programming languages, to fully self-contained and stand-alone transformation environments.

### 2.1.1   Anatomy of a Transformation System

A common way to think about transformations is to divide them into *stages*. All stages taken together is considered a *pipeline*. The syntax of the input and output languages are specified by *source-* and *target grammars*, respectively. For source-to-source transformation systems, as illustrated in, Figure 2.1, the source and target language is the same.
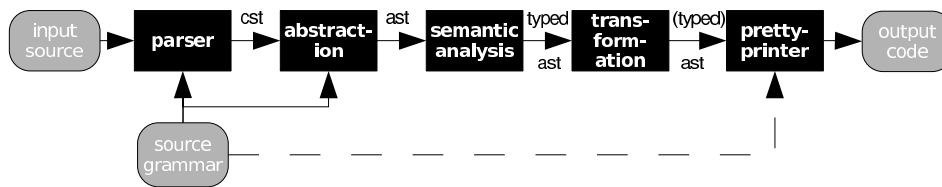
Figure 2.1: Conceptual pipeline for a source-to-source transformation system.

The process indicated in Figure 2.1 starts with the system parsing the source of the input language. The format of the input language is described by a *source language grammar*. The parsing stage constructs a parse tree, or *concrete syntax tree* (CST), from the input text. Layout and unnecessary lexical elements such as parentheses and keywords are removed from this tree in the abstraction stage, and an *abstract syntax tree* (AST) is derived. Semantic analysis is performed and the AST is annotated with type information. In practise, the AST may be constructed while parsing, and in some implementations, type checking is also done concurrently with parsing. The transformation rewrites the AST. After modifications are complete, the tree will be serialised back to source code, using a code formatter, or *pretty printer*.

This model is highly conceptual. Many source-to-source transformation systems, such as TXL [Cor04], transform the CST directly. ASTs are never derived. Some systems do not support type contexts and the AST in these systems will not contain type information. Others construct a higher level program model, or an abstract syntax graph, which is then subjected to graph rewriting techniques.

A complete transformation, from program code to program code, is called a *run*. Each of the boxes in Figure 2.1 represents a well-delineated transformation, and is called a *stage*. Each stage may internally be split into *phases*. Each phase consists of a sequence of rule application *steps*. A step, or rule application, is the smallest unit of transformation. They represent the atoms from which transformations are built.

Other architectural models for transformation systems also exist. A common example is the *incrementally updating* system. In these systems, the output of one run is the input to the next. A human operator is usually involved in adjusting the transformation parameters between each run.

## 2.1.2   Features of Software Transformation Systems

A software transformation system may be decomposed into three closely related parts: a *program representation* holding the program the system manipulates, a *transformation language* for expressing these manipulations, and an *environment* which is used to interact with the developer. Figure 2.2 shows a feature model fragment which visualises this decomposition. Details of each of these features will be described in the
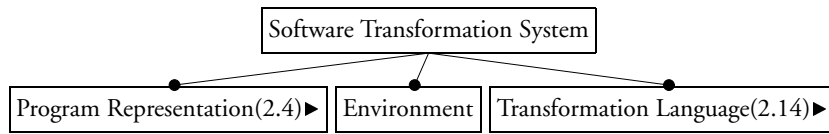
Figure 2.2: Top-level features of software transformation systems.

following sections. The numbers in parentheses refer to figure numbers for additional diagrams which elaborate on a particular feature. Not all features will be discussed in full detail. This dissertation is largely concerned with the interplay between abstract models for programs and transformation languages used to manipulate these. A full discussion of the user interface, i.e. *environments*, of transformation systems is therefore out of scope. Before continuing, the feature model notation is explained.

## 2.2    Feature Models

| Symbol | Explanation |
|---|---|
| $F$ | Solitary feature with cardinality [1..1], i.e., *mandatory* feature |
| $F$ | Solitary feature with cardinality [0..1], i.e., *optional* feature |
| $[n..m]$ $F$ | Solitary feature with cardinality $[n..m]$, $n \geq 0 \wedge m \geq n \wedge m > 1$, i.e., *mandatory clonable* feature |
| $F$ | Grouped feature with cardinality [0..1] |
| $F \blacktriangleright$ | Feature model reference $F$ |
| △ | Feature group with cardinality [1..1], i.e. *xor*-group |
| ▲ | Feature group with cardinality $[1..k]$, where $k$ is the group size, i.e. *or*-group |

Figure 2.3: Symbols used in cardinality-based feature modeling

Feature models [Bat05] provide a graphical notation for describing variation points found in the design of software systems. The notation is well suited for visualising the relationship between features using the precise and general kernel language described in Figure 2.3. Organising the feature space into hierarchical contexts helps guide discussions. The application of feature models spans from the purely conceptual, at the domain concept level, to implementation detail, at the design and architectural level. This chapter mainly uses feature models for describing architectural variation points.

By saying that feature models describe the essential variability of software transformation systems, it is meant that they describe the *characteristic concepts* and features for these systems, and that the models show the relationships and interactions between these. The characteristic concepts and features are described using a *design vocabulary*, which is introduced in the boxes of the feature diagrams. It is important to point out that this chapter is guided by the notion of *"characteristicness"*: a discussion of features which are also commonplace outside software transformation is avoided; features which pertain to software systems in general will not be discussed.

Alternative formalisms for describing design knowledge are ontologies [Gru93]. Feature models were chosen here because they are better suited to visualise the variability and configuration aspects of software designs. For a discussion of the relation between feature models and ontologies, refer to [CKK06].

## 2.3  Program Representation

Software transformation systems operate on formal documents which have a precise syntax definition and sometimes a detailed, formal semantics. These documents may be programs or specifications, or simply structured specification documents with little semantics. Both specifications and program source are commonly referred to as *program code* or *subject code* in the rest of this chapter. Though programs are formal documents, models representing programs are referred to as *program object models* (sometimes just program models) throughout this dissertation, to distinguish them from general document object models as found in the field of document processing. This dissertation takes the stance that subject code usually has an a priori defined semantics which operations on the program object model must preserve.

Due to its formal nature, program code has a clear structure, but this structure does not necessarily match how the transformation system represents program code internally. The choice of internal data structure used to represent programs affects the ease with which various operations can be performed. For example, if the program is represented as a control-flow graph, control flow analysis becomes easy, but structural or syntactic changes, such as refactoring is all the more difficult. The choice of representation significantly affects the possible applications of a transformation systems. Specific design and implementation choices for the representation further influence both performance characteristics and the difficulty of expressing different kinds of analyses and transformations. This argument also works in reverse: the intended transformations of a system will to a large extent dictate the choice of internal representation.

As an example, consider software transformation systems intended for source-based re-engineering. These usually employ a parse tree representation that accurately captures source code details. This may include layout and comments. On the other

hand, systems intended for software modelling mostly use graph-like representations that are far removed from the concrete syntax of the source language.
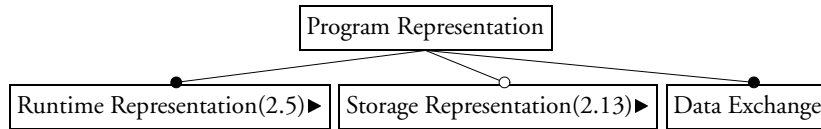


Figure 2.4: Feature decomposition for program representation.

Figure 2.4 shows a decomposition of the feature space for *program representations*.

- *Runtime representation* – refers to the data structure and features of how the program code is represented at runtime. Of all the features related to program representation, the choice of runtime representation has the largest impact on the expressiveness and performance of a transformation system, see p. 20.

- *Storage representation* – refers to the facilities for storing program code on disk at intermediate transformation stages. Choices pertaining to intermediate storage on disk affects the interoperability and modularity of a system, see p. 30.

- *Data exchange* – refers to facilities for loading source code into the system and produce target code as output. This might be features for parsing and pretty-printing, used with source-to-source transformations. These features fall mostly outside the scope of this dissertation.

The following sections discusses each feature in turn.
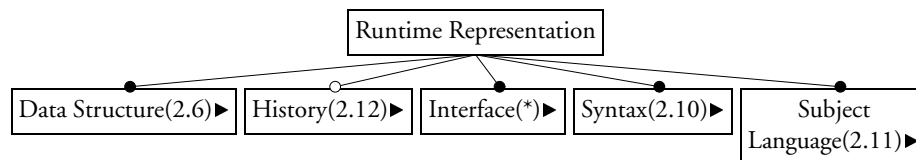
## 2.3.1   Runtime Representation



Figure 2.5: Feature decomposition for runtime representation.

Subject programs are contained in a runtime representation when the software transformation system executes. This may for example be an abstract syntax tree, a graph model, or a database. Collectively, these are called *program object models*, and may be described by the following features.

- *Data structure* – refers to the choice of (principal) abstract data type used for the program object model. This is arguably the most important aspect of the runtime representation. Common choices are trees and graphs, with various invariants on the well-formedness of the subject program, see the next section.

- *History* – the representation may optionally support the notion of transformation history by keeping a modification history of the program code, see p. 29.

- *Interface* – refers to the programming interface available for the runtime representation. In many systems, the interface is available as language constructs in the transformation language. That is, the transformation language is specifically designed with primitive constructs for manipulating the program object model. For this reason, the interface feature is discussed together with the other language features, in Section 2.4.

- *Syntax* – refers to the types of syntaxes available – abstract or concrete – for writing and reading program code when implementing transformation programs, see p. 27.

- *Subject language* – The language in which the program code must be expressed, i.e. the supported source and target languages, see p. 29.

**Data Structure**

The feature model for the data structure in Figure 2.6 describes which data types are used to represent the program code at runtime, and which support exists for maintaining well-formedness of the program code structure.
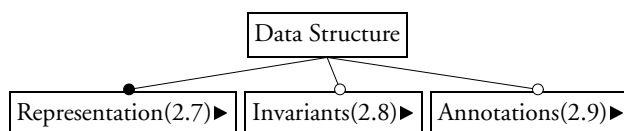


Figure 2.6: Feature decomposition for data structure.

- *Representation* – details the choice of abstract data type for the program code, ranging from strings and lists to relational databases, see the next section.

- *Invariants* – describes how structural and semantical invariants of the program code can be placed and enforced on the representation, see p. 24

- *Annotations* – refers to the ability of the representation to handle meta-information not part of the program code structure, see p. 26.

**Representation**

Figure 2.7 describes the features of the data type used to represent the program code. Under each principal choice (list, tree, graph, etc), the most common variants are shown.
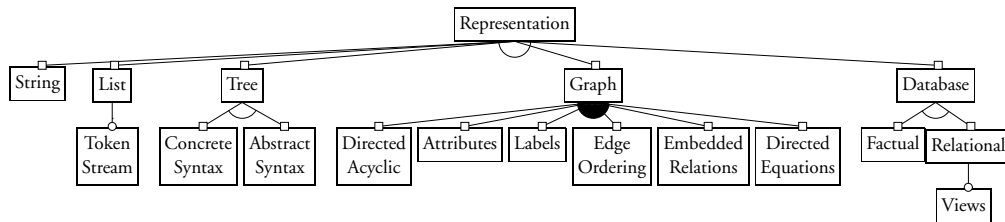


Figure 2.7: Variants of data structure representations.

- *String* – The simplest choice of data structure for representing program code is a text (or even binary) string. In this case, the transformation system amounts to a string rewrite engine, as in the theory of formal languages and automata. The C/C++ preprocessor is one example of such a "transformation system". The trio sed, grep and awk [DR97] of Unix tools is another, based on regular expressions. Representing programs as strings fails to capture the grammatical structure inherent in the program code. This quickly leads to subtle bugs for any non-trivial transformation. String rewriting engines can hardly be called software transformation systems.

- *List* – A slightly more structured representation than the string is the token stream output by a lexer, i.e. a *list* of tokens. Each token is marked with a type, such as keyword, identifier, string literal or parenthesis, e.g:

```
["if":keyword, "(":left_paren, ..., ")":right_paren]
```

The ANTLR parser toolkit [PQ95] supports rewriting on token lists. Both the string and token list representations of program code are useful for limited, layout preserving rewriting. As long as no context or grammatical information is needed, the matching can be done reliably at the lexical level.

- *Trees* – Most practical transformations need at least grammatical structure and most often also context knowledge such as variable binding or type information. Extracting the grammatical structure from program code text can be automated using syntax analysis, i.e. parsing. Syntax analysis produces trees. Representing program code as trees dates back to the earliest compilers, and multiple variants are possible. In the case of *concrete syntax trees*, the tree contains a faithful representation of the source code, possibly excluding

non-essential whitespace. In the case of *abstract syntax trees*, all non-essential nodes, such as whitespace, parentheses, statement and expression separators, have been removed. These can be automatically regenerated. Normally, comments and documentation are also left out. Trees are often given a textual syntax, in the form of terms, e.g.:

```
If(False,Int(1),Int(2))
```

The maximal sharing [vdBdJKO00] technique is a variation of the tree representation which improves execution time of matching and memory efficiency is. The tree is represented as a directed, acyclic graph (DAG), where equal subtrees occurring multiple times in the tree are stored only once. This technique improves the efficiency of term matching significantly. It has been used in several transformation systems, including ASF+SDF [vdBvDH⁺01], Stratego, ELAN [BKK⁺04], Tom [MRV03] and derivates of these. It is important to note that the maximal sharing technique hides the sharing, making the DAG behave as a tree. This is required for the term rewriting theory. Rewriting on DAGs, sometimes referred to as term graph rewriting, has different termination and confluence properties [Plu99, BEG⁺87]. An example of transformation system based around term graphs is HOPS [Kah99, KD01], an interactive program transformation and editing environment that ensures syntactic and semantic correctness. HOPS may also be described as a syntax directed editor.

- *Graph* – Plain trees are not sufficient for explicitly capturing some important types of context information, such as typing and variable binding. Section 2.4 discusses how tree-based transformation systems deal with this problem. The program code may be expressed as a *graph*. This allows additional links (edges) to be added from, say, a variable use to its definition, or from an identifier to its type, thus capturing context information. *Labelled* edges are handy for distinguishing between kinds of relationships between two nodes, for example, between a `use-def` and a `type-of` relationship. *Attributes* are named properties of nodes that contain values. In most graph-based systems, a node may have a set of named attributes. These can be matched on during rewriting. Some systems, such as the modelling system MetaEdit [SLTM91], also allow attributes on edges. Attribute grammar systems are capable of declaring dependencies between attributes across nodes using *directed equations*. Nodes may be related using *embedded relations*, as in PROGRES [Sch04]. These features are closely tied to transformation language features and are discussed in Section 2.4. None of the transformation systems known to the author employs hypergraphs directly, i.e. graphs where edges connect more than two nodes. The GAMMA multiset rewriting system [BM91, BM93], seems to come closest in terms of hypergraph semantics. Other works have been derived from this,

such as [CFG96]. Neither of the systems is used for program transformation.

- *Database* – Linking together nodes in a graph or subterms of a term can be done using a *relational* database. Transformation systems based on this approach are comparatively sparse. APTS is the only relational database system that allows arbitrary transformation. In [SNDH04], the authors describe an interactive system focused on the refactoring of Clean programs. The system described in [CNR90] extracts facts from C code into a database, but only allows subsequent analysis, not transformation. In all cases, the program code is expressed in tables with relations between them. Transformations and analyses are expressed as relational queries, in styles similar to normal relational databases. A feature unique to the database approach is the ability to declaratively construct custom *views* of the program code and do manipulation on these. In all other approaches, similar functionality must be provided by the developer, and is highly non-trivial. A related approach is the *factual* databases used in logic programming languages such as Prolog. This is employed by JTransformer [Win03, KK04]. The structure of the program code is stored as facts in a database. Questions (queries) may be asked. These are automatically resolved against the database by the Prolog inference engine. A discussion of the finer points of different database approaches is beyond the scope of this chapter, save to point out that while the Prolog model is based on the theory of predicate calculus, the relational database approach has its roots in relational algebra.

Perhaps the principal tradeoff in the selection of a suitable representation is between expressiveness and efficiency. Low-level and simple abstract data types such as lists and trees are very efficient to transform, but it often becomes difficult to embed analysis results in flexible ways. That is why annotations (discussed later) are only found as additions to the more "primitive" representations. The elaborate representations, such as general graphs and relational databases, are mainly used for high-level concepts. Models are first extracted from the source code. Queries and computations are performed on these models. The results are later used to guide transformations on the low-level representations.

Relational databases are often used for various types of code querying and analysis, as in the case of CodeQuest [HVdMdV05]. The program object models used for this are removed from the primary grammar structure, because encoding recursive data structures into relational databases (tables) is generally inefficient.

### Invariants

The syntax and semantics of the program code, no matter how it is represented, give rise to a large amount of invariants, see Figure 2.8. These must kept throughout the

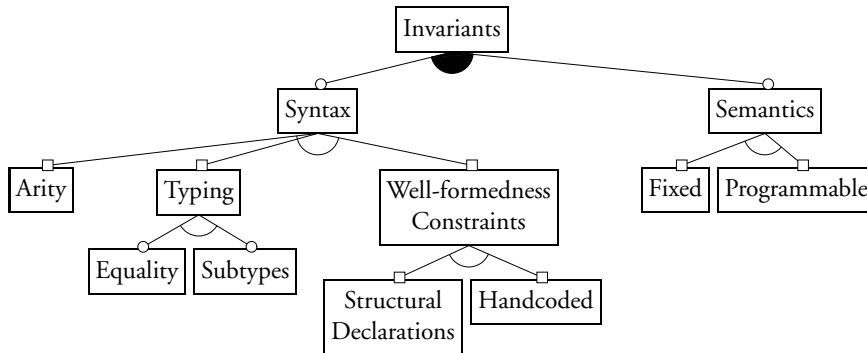transformation run, but may be lifted temporarily during transformation steps, or even phases.



Figure 2.8: Feature decomposition for data structure invariants.

**Syntactical Invariants**

- *Arity* – A weak variant of typing, found in the term-based system Stratego. For each type, only the numeric arity, i.e. the number of arguments is enforced. An if-then-else node may look like *If : Expr * Stmt * Stmt*, thus declaring terms on the form *If(e,st$_0$,st$_1$)* where *e* must be an *Expr* and $st_0$, $st_1$ must be *Stmt*. Stratego only requires that three subterms be attached to *If*. It does not verify their types.

- *Typing* – The most common way for ensuring grammatical well-formedness on the subject code is to use the type system of a strongly typed transformation language (not to be confused with the type system of the subject language). The variants include basic *equality*-based systems (only terms of type *T* may be used where *T* is expected) and systems which support the notion of *subtyping* (any subtype of *T* may be used where *T* is expected). In either case, the syntactical correctness of the program code with respect to a grammar can be ensured. It is worth noting that although token lists discussed above are by definition typed, they rarely offer any grammatical correctness guarantees. TXL [Cor04] operates on concrete syntax trees. The language ensures that when a subtree is replaced on a given node, the new subtree must be of a compatible type, i.e. the new subtree must parse to the same production as the old. The types are defined by the grammar for the language.

- *Well-formedness constraints* – A more powerful approach is to provide a declarative language for expressing *structural constraints*. It is then possible to either verify that a given transformation will never violate these constraints, or to insert constraint checking between transformation phases, at declared *safe spots*.

AGG [Tae04] is a graph transformation system which provides structural con-straints on its graphs. The constraints are specified as part of the graph gram-mar.

One could consider the structural constraint feature an extension or variant of user-defined types, but there are some essential differences. Ensuring that each rewrit-ing step respects a given grammar is computationally feasible, because grammatical constraints map relatively easily to most strongly typed languages. Checking struc-tural constraints after a rewrite step may not terminate in the general case (for exam-ple, if the constraint is given in a Turing-complete formalism). Even when the con-straint language offers termination guarantees, the computational complexity may be prohibitive.

**Semantic Invariants**    In addition to syntactical well-formedness, facilities may ex-ist for defining parts of the semantics of the program code.

- *Fixed* – The system comes with a fixed implementation that preserves (possibly a subset of) the semantics of the subject language. In the case where the lan-guage of the program code is fixed, a complete enforcement of the semantical invariants is possible thanks to a priori hand coded logic in the system. JTrans-former provides a library of conditional transformations for Java, many of which are guaranteed to preserve Java semantics. The FermaT [War02, War89] transformation library guarantees semantics preservation for FermaT's fixed subject language, WSL.

- *Programmable* – The system supports the programmer in implementing lan-guage semantics constraints, for example by providing suitable generic libraries or language constructs for capturing language semantics. Varying degrees of support for this is present in most transformation systems. Notable features are discussed in Section 2.4.

Few transformation systems enforce type-correctness of the subject code or sim-ilar forms of semantical correctness on their transformations. It may be exceedingly difficult to check for these during the runtime of the transformation, and it is also an open problem how to efficiently encode semantics for the subject language into the type system of the transformation language. For this reason, it is not uncommon to "outsource" questions of correctness to theorem provers.

### Annotations

The structure used to represent the program code should be precise and minimal. This reduces the complexity of the transformation programs: fewer node types means

fewer patterns for the rules. A minimal structure sometimes conflicts with extensibility, however. It is often necessary to store intermediate computation results and relate these to elements in the program code. A common way to handle this at the program representation level is to store such information as annotations, see Figure 2.9.
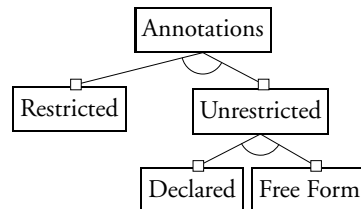


Figure 2.9: Feature decomposition of annotations on the program representation.

Annotations are (temporary) pieces of meta information that may result from analyses such as type inferencing, variable scoping or source code metric calculations. Annotations are also used to retain comments and layout information, without declaring these as part of the primary program code structure.

- *Restricted* – Only a limited, pre-defined number of annotations may be placed on the program code in the runtime structure.

- *Unrestricted* – Annotations can be freely defined by the programmer. In the case of *free-form* annotations, arbitrary meta information is allowed. This is the case for ASF and Stratego. In the case of *declared* annotations, syntactical (and optionally, semantical) restrictions are placed on the annotations. These must be declared in advance.

Annotations are different from (tree or graph) attributes in several ways. Since annotations are pieces of meta-information, they can be discarded at any time without changing syntactic or semantic validity. Moreover, even declared annotations are not part of the program code grammar, so one cannot expect that all transformations will respect and update them.

Most transformation system support annotations in one way or another. It is generally the case that strongly typed transformation languages necessitate declared, as opposed to free-form, annotations.

**Syntax**

Developers reading and writing fragments of subject language program code do so in the program code *syntax*. This syntax may be quite different from that of the subject language, and is is often influenced by the choice of program representation.
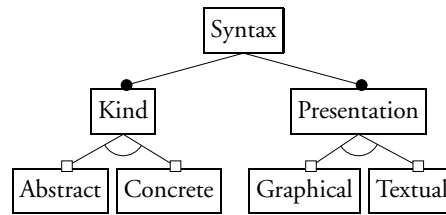
Figure 2.10: Features for supporting subject syntax in transformation programs.

- *Kind* – In source-to-source transformation systems such as TXL [CCH05] and
  ASF+SDF [vdBvDH⁺01], subject program code fragments are often written in
  the syntax of source language called the *concrete syntax*. The concrete syntax for
  the program code is embedded in the transformation program. When concrete
  syntax support is not present, program code must be written using the data
  types of the transformation language, that is, in an *abstract syntax*. This is the
  case for Tom and ANTLR, where tree nodes and trees are built like any Java
  data structure, using object instantiation.

  Stratego supports both concrete and abstract syntax, demonstrated in the func-
  tionally identical rules shown next, where the concrete syntax fragment is en-
  closed in "semantic" brackets:

  ```
  EvalIf: |[ if(true) ~e0 else ~e1 ]| → |[ ~e0 ]|
  EvalIf: If(BooleanLiteral("true"),e0,e1) → e0
  ```

- *Presentation* – For the systems mentioned above, the syntaxes were all *textual*.
  Another variation is to represent the program code using a *graphical* notation,
  irrespective of whether the source language is visual or not. This is done in
  AGG and PROGRESS which both offer abstract graphical syntax and presen-
  tation.

The primary tradeoff between concrete and abstract syntax is readability versus
preciseness. Concrete syntax patterns are mostly easier to read and write for program-
mers. However, extra care must be taken to ensure that the pattern (and the meta
variables) match exactly the types of AST nodes intended. Consider the following
concrete syntax pattern:

```
|[ boolean equals(Object ~n) { ~stm }]|
```

It does not match the following declaration, because of the visibility modifier `public`.

```
public boolean equals(Object o) { return false; }
```

The pattern, as written, specifies that only declarations without any visibility mod-
ifiers should be matched. Writing exact pattern matches in abstract syntax is often
easier, though significantly more verbose. On the other hand, code generation usually

benefits significantly from concrete syntax templates, since such templates are generally easier to write and maintain compared to equivalent templates in an abstract syntax.

**Subject Language**

The possible choices for subject language clearly defines the applicability of a given transformation system for a concrete problem.
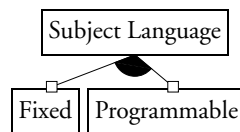


Figure 2.11: Feature decomposition for subject language.

- *Fixed* – The subject language is fixed to a particular language.

- *Programmable* – The subject language can be freely defined by the programmer.

Both JTransformer and FermaT are fixed to one subject language. This fixedness gives the systems an advantage in providing a convenient and robust transformation library. However, FermaT's subject language is WSL, a wide-spectrum language designed to capture a large set of source languages. It contains a small kernel of constructs to specify (non-deterministic) choice and iteration. Various assembler dialects have been transformed into it [War99]. Both C and COBOL code is in turn produced from WSL. The basic transformations in the FermaT library guarantee both syntactic and semantic correctness. JTransformer also comes with a library of basic transformations for its subject language, Java. Many of these preserve the Java semantics and syntax.

The choice of language may be programmable, as is the case for TXL [Cor04], ASF+SDF [vdBvDH+01], Stratego/XT [BKVV06] Tom [MGR05], DMS [BPM04], and Elegant [Aug93]. In these systems, the developer must supply all syntactic and semantics-preserving logic, using whatever support the transformation system provides for this. For realistic languages, this is a considerable undertaking. In many cases, separate projects exist which specialise general systems for a particular language. These aim to achieve the best of both worlds: a sound library of basic transformations with full flexibility, e.g. CodeBoost [BKHV03] specialises Stratego for C++.

**History**

Support for history as part of the runtime representation provides a low-level way of keeping track of changes to the program code. It complements execution traces, discussed in Section 2.4.
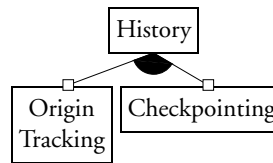
Figure 2.12: Features for history support.

- *Origin tracking* – A feature that retains tracking information with the program code elements throughout a complete transformation. It is used to determine how a code element in the final product relates to the code elements in the source input, i.e. where a given code element in the result program came from in the source program. Earlier prototypes of ASF had this feature [vDKT93].

- *Checkpointing* – Runtime representation support for transaction-like operations. With checkpointing, a snapshot can be taken of the program code so that this state can be restored if a transformation sequence fails. Stratego offers full checkpointing support due to the (local) backtracking feature of the language, as does Tom when rewriting functional terms (Tom also supports non-functional terms and graphs, where the backtracking is not available).

### 2.3.2   Storage Representation

Many transformation systems provide special support for storing intermediate forms of the program code, see Figure 2.13. The code may be stored in a special, efficient storage format, or as source code in the source or target language. The choice between a special storage format versus language source code affects how auxiliary information can be added.

Special storage of the internal data allows bundling of analysis results and constraints with the data. This may in turn be used to minimise costly analyses, such as parsing and type checking, by caching results on disk between executions of the system. Having a standardised internal storage facility opens up for interchange of analysis results between components of the transformation system: fragments of code can now easily be serialised and sent between separate processes, or over a network.

Aside from the size benefit offered by good storage formats, extra information such as accumulated transformation history can be added in the form of origin tracking or execution traces. This is not possible (or at least rather difficult) when the interchange format is fixed to the source code of the source (or target) language.

The storage representation feature from Figure 2.13 is decomposed into the following features:

- *Extensibility* – Either the storage representation is *fixed* for the transformation system, or it is *programmable*. This allows the programmer to extend it. Strat-
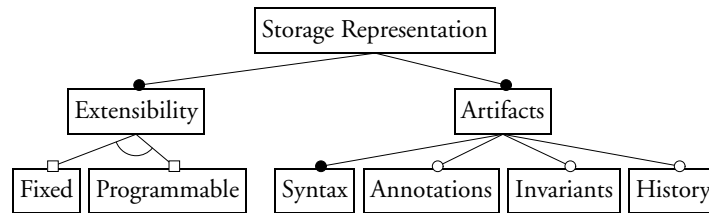
Figure 2.13: Feature decomposition of storage representation.

ego/XT and ASF use a fixed format called ATerm. In the case of Stratego/XT, additional formats may be added by the user. AGG has a fixed XML-based format for its graphs.

- *Artifacts* – The choice of *syntax* is the most influential design choice of the storage representation. When concrete syntax (of either the source or target language) is used for storage, auxiliary information is considerably more difficult to encode. By using an extensible abstract syntax, a transformation system provides the transformation programmers with more freedom.

  In the case of an abstract syntax, custom *invariants* concerning the data may accompany the program code. User-extensible invariants allows the transformation programmer to express additional invariants that must be respected by other programs and components processing this program code.

  Depending on the choice of syntax, the storage format may support storing *annotations*. There is usually a correspondence between how the runtime representation language handles annotations and how these are stored: the runtime typing and structural constrains must usually be respected.

  *History* – The stored files may contain checkpointing information which may allow backtracking across saved sessions. Such information allows mid-transaction saves and rewind. Additionally, origin traces may be included in the saved file This aids in origin tracking between sessions and between tools.

Storing of concrete syntax captures layout, even for visual languages, where the graphical objects in saved visual programs retain their user-edited placements. The GXL [HWS00] language encodes this information in special graph attributes in the stored files. Storing additional, custom transformation invariants along with the program is required if other transformation components are to know about these additional constraints. A caveat is that the formats used to store such constrains, and their meaning, must be known to all components. The AGG system preserves these constraints by coding both the program model and the constraints into the same unit.

## 2.4    Transformation Language

The transformation language is the centrepiece of any programmable transformation system. It is the main vehicle for expressing transformations, and should therefore easily express the kinds of transformations desired by its user. As with any programming language, the degree of expressiveness provided by the language is a double-edged sword: Having many language features generally increases expressiveness, so does avoiding usage restrictions on individual features. There is a tension between expressiveness and how easy proofs of transformations can be done. Usage restrictions on individual language features, and careful consideration of feature combinations are required if good provability is desired. However, not all program transformation approaches are concerned with provability. This has allowed a rich set of transformation language features to evolve.
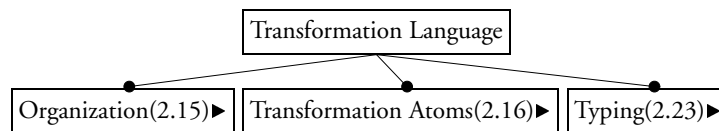


Figure 2.14: Feature decomposition of transformation languages.

Given the rich literature and existing surveys on the details of particular feature sets, such as [Fea87, PS83, Vis05a, vWV03], this section focuses on the broad lines and the relationship between transformation languages and program models. The features being considered are shown in Figure 2.14.

- *Organisation* – refers to the organisation of the rule and data declarations, see the next section.

- *Transformation Atoms* – refers to the properties of the units of transformations, i.e. the functions, rewrite rules, queries and strategies, see p. 36.

- *Typing* – describes characteristic features of how typing is realised in transformation languages, see p. 47.

### 2.4.1    Organisation

Features for organising the language declarations are shown in Figure 2.15. This organisation is necessary for managing the complexity of the transformation program itself. As transformation programs grow in size, they are subjected to the same issues of scale which are already seen in constructing other types of software applications.

- *Grouping* – The feature model suggests the hierarchical organisation of transformation expressions or statements into applications of *operations*. Operations
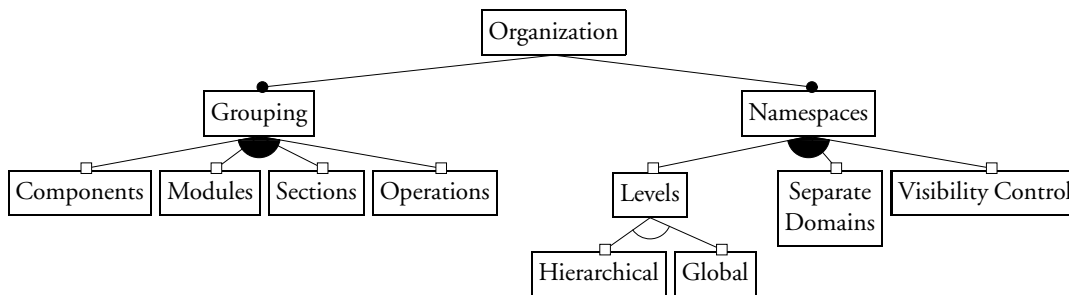
Figure 2.15: Language features for organization of declarations.

correspond to transformation atoms. For textual transformation languages, operations may be placed into *sections* inside their declaring file. Sections are in turn grouped into named *modules*. A module may be a file, or multiple files may make out one module. Modules may again be composed into *components*.

* *Namespaces* - The organisation of namespaces is related to grouping. Scoping may be done in *levels* which follow a *hierarchy*, usually the one provided by the grouping. Another alternative is no levelling at all. This gives one global name space for all declarations. In either case, the different types of declarations may be organised into *separate domains*, allowing both a rule and a type with the same name to exist at the same time without causing confusion. A final consideration is encapsulating names into their respective scope by restricting *visibility*.

The basic organisation features are combined in a plethora of ways. The class-based systems JastAdd and Tom group methods (operations) into named sections (classes) which become one level in the namespace. The methods reside in a different namespace domain than the variables and the types. It is possible to have both a type and a method with the same name without confusion.

For PROGRES, hierarchical visibility is only available for global graph constraints, on a per-section basis. For Elegant, a component is either a scanner, transformer or code generator, i.e. a phase of a compiler. For APTS, all definitions are maintained as entries in databases. You can load and store databases, composing them by merging two databases, thus emulating the concept of components. For Stratego, each file is a module, divided into sections. Rules and strategies are the only two types of operations, and both live in the same, global namespace. There is no visibility control, so rules and strategies with the same name may interfere.

One drawback of the numerous realisations of these basic features is that learning transformation languages might be daunting. The nuances and novel combinations serve to increase the learning curve for new developers. Another drawback is that there are few, if any, standardised ways of organising transformation programs. The

closest to a de facto standard, at least at the macro level, is perhaps those which mimic compiler pipelines. This is (mostly) the case for JastAdd, Elegant and Stratego.

A number of noteworthy characteristic organisation features are discussed next.

*Rule/Dependency Separation* – Attribute grammar systems combine the dependencies between nodes and the directed equations used to compute derived attributes into one construct. In PROGRES, the rule for computation of the derived attribute is kept separate from the dependence declaration.

*Inheritance* – In Tom and JastAdd, both of which are embedded domain-specific languages for Java, inheritance is used to encode the grammar structure of the subject language in the type system of the host language. Consider the following the grammar fragment:

$$expr ::= literal \,|\, binexpr \,|\, \ldots\,;$$
$$literal ::= string\_literal \,|\, integer\_literal \,|\, \ldots\,;$$

This translates into the following (Java) type declarations for JastAdd (the class `ASTNode` is always the root of such type hierarchies in JastAdd):

```
abstract class Expr extends ASTNode { ... }
abstract class Literal extends Expr { ... }
class StringLiteral extends Literal { ... }
```

The situation is similar with Tom, but the programmer may choose the root class freely.

In PROGRES, the graph grammar declaration uses subtyping to declare the types (and attributes) of nodes in the graph, e.g.

```
node class Root; intrinsic a : = 1; end
node class Child1 is a Root; redef b : = 2; end
node class Child2 is a Root; redef c : = 3; end
```

*Transcripts* – Transcripts are an organisational unit only found in APTS. A transcript implements either a rewrite or an inference rule for a relation. A transcript for a relation contains one or more inference rules which are used to analyse the CST and maintain a database of program properties. The rules inside the transcript are applied non-deterministically until no relations in the program database can be changed, i.e. until a fixpoint has been reached. Consider the following example, included to provide some flavour of the APTS language. The example defines the notion of free variables in a SETL-like subject language [Pag93].

```
1 transcript freevar();
2   rel freevar:  [node, tree];
3       free:  [tree];
4   prompt free:  [1, ' is a free variable '];
```

```
 5    external bvar:  [node, tree];
 6            op:  [node, node];
 7    key free:  [1];
 8  begin
 9    freevar(root(), .x) -> free(.x);
10    match(% expr, .x % ) | isavar(% expr, .x% )
11      -> freevar(% expr, .x% , % expr, .x% );
12    match(% lexpr, .x % ) | isavar(% lexpr, .x% )
13      -> freevar(% lexpr, .x% , % lexpr, .x% );
14    op(.x, .y) and freevar(.y, .z) and not bvar(.x, .z) -> freevar(.x, .z);
15  end;
```

This transcript, named `freevar()`, defines the relations `freevar` and `free`. It depends on the external relations `bvar` and `op` (defined in other transcripts). The `prompt` definition specifies how tuples of the relations in this transcript are displayed. The inference rules of this transcript are specified between `begin` and `end`. The rules on line 10-13 specify that any variable that is an expression on a left or right hand side of an assignment, is a free variable. `match` and `isavar` are builtins of APTS. `match` supports non-linear pattern matching (discussed later); here, `.x` is a pattern variable. The main inference rule for free variables is given on line 14, and states that a free variable `.z` of term `.y` is a free variable of term `.x` iff `.x` contains `.y` as an immediate subterm (the `op(.x, .y)` part) and `.z` is not a bound variable of `.x` (the `not bvar(.x, .y)` part). These rules are applied non-deterministically until none are applicable any more. This completes the update of the program database.

Rewrite rule transcripts are similar to relation transcripts. They consist of one or more rewrite rules, which rewrite the CST, as opposed to the program database.

Transcripts have some properties of modules. There is a simple kind of namespacing and visibility for transcripts: rules inside a transcript are not by default visible outside the transcript. Rules from other transcripts can only be invoked indirectly, by invoking their transcripts. Transcripts also enforce a special evaluation semantics. All external relations must have been evaluated before a transcript can be evaluated. Cyclic dependencies between relations are only allowed within transcripts. Multiple transcripts may be defined in the same file.

### Parametrisation

The different organisation units, such as types, rules, functions and strategies may for practically all transformation languages always be parametrised with values. In the transformation languages Elegant, Tom and JastAdd types may be parametrised with types. Stratego and Elegant offer higher-order operations.

*Higher-Order Operations* – A catch-all feature for higher-order rules, strategies, functions and queries. The known benefits from higher-order functions also apply to

rules, strategies and queries: they aid in parametrisation and subsequent composition of code, thereby allowing a very flexible, precise and familiar notation for expressing operations. The following Stratego strategy definition defines a top-down (pre-order) term (tree) traversal, where the strategy *s* is applied at every subterm (tree node) before its children are visited:

```
topdown(s) = s; all(topdown(s))
```

*Module Parametrisation* – Parametrisation of modules, as offered by the ML-family of languages, is seen in very few of the domain-specific languages provided by any of the transformation languages. JastAdd and Tom (both based on Java, which offers parametrised classes) are the only known exceptions. Also, no transformation system currently offers parametrised components. The absence of parametrisation at higher levels, and the absence of higher levels of organisation, may be taken as a sign that issues common to programming in the large have not been addressed for transformation systems yet.

## 2.4.2    Transformation Atoms

Transformation atoms are the fundamental building blocks of transformations, see Figure 2.16. For rule-based languages, they are the rewrite rules. For functional languages, they are the functions. For relational languages, they are the queries. In style with modern science, transformation atoms are not indivisible: functions are made from expressions, rewrite rules from patterns and conditions, and relational queries from path expressions and statements.
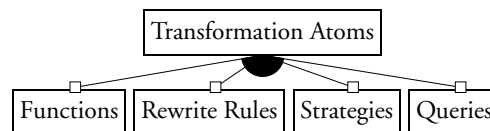


Figure 2.16: Feature decomposition of transformation atoms.

The following discussion will focus on characteristic properties of the rewrite rules, as this is arguably one of the most characteristic features of transformation languages. Functions and queries found in in transformation languages are familiar from general purpose and relational query languages.

*Relations* – Multiple variants of the relation feature exist. In APTS, the program database stores relations extracted from the CST using inference rules. In PROGRES, relations between nodes and node types are declared using graph queries and path expressions. In JTransformer, the Program Element Fact (PEF) database contains relations extracted from the Java AST, e.g.:

```
1  importT(10000, 30001, 20003).
2  importT(10001, 30001, 'java.util').
```

The fact on line 1 represent a Java import statement. Each fact has a unique id. The fact on line 1 has id 10000. It states that the class corresponding to fact id 20003 is imported by the compilation unit of id 30001. On line 2, another fact states that the same compilation unit also imports `java.util.*`, i.e. all classes of the package `java.util`.

In all systems, queries can be done on the relations; the relations often encode "refined facts" that are extracted and analysed from the CST and AST, i.e. information that is only implicit in the AST representation, such as the binding from a name to the actual definition for that name.

*Relation Functions* – Elegant provides a kind of function with a special semantics, called a relation. In contrast to functions, relations can have an arbitrary number of input and output arguments. The arguments are updated by the body in any order. The effect of a relation is to synchronise all the output domains with the input domains, [JAM99].

```
relations
MakeFunctions (NIL : List(Func), out {}, {}) { }
MakeFunctions (funcs : List(Func) out signs, decls) {
  MakeOneFunc (funcs.head out s : VOID, d : VOID)
  MakeFunctions(funcs.tail out ss : VOID, ds : VOID)
local
  signs : VOID = { s "\n" ss }
  decls : VOID = { d "\n" ds }
}
```

The relation `MakeFunctions` is used to traverse a list (`funcs`) of functions, and for each element, call the relation `MakeOneFunc` to compute its signature and its complete declaration. This results in two separate lists which are both returned from `MakeFunctions`, one for the signatures, in `signs`, and one for the declarations, in `decls`. The abstraction `MakeFunctions` therefore returns two values, whereas a function would only return one. It is possible for the return values to be declared as lazy values. In this case, they will only be computed if they are used by the caller.

*Congruences* – Congruences are a language construct for defining data structure specific traversals. They are described in Chapter 3 in the context of Stratego.

*Queries* – Queries are expressions for navigating, analysing or modifying the program code. In the case of graph queries, the queries are usually only used for analysis. Modification is done using graph rewrite rules. Queries on relational databases also allow database updates, which amounts to program code modification. The following PROGRES code illustrates a query, [Sch04].

```
1  query AllConsistentConfigurations(out CNameSet : string [0: n]) =
2    use LocalNameSet, ResultNameSet : string [0: n] do
3        ResultNameSet : = nil
```

```
 4      & GetAllConfigurations(out LocalNameSet)
 5      & for all LocalCName : = elem(LocalNameSet) do
 6        choose
 7          when ConfigurationWithMain(out LocalCName)
 8          and not ConfigurationWithUselessVariant(LocalCName)
 9          and for all LocalMName : = elem(LocalCName.-has->.=needs=>) do
10              ModuleInConfiguration(LocalCName, LocalMName)
11              end
12          then ResultNameSet : = ResultNameSet or LocalCName
13        end
14      end
15      & CNameSet : = ResultNameSet
16    end
17  end
```

This query computes all consistent configurations of a software package. It uses another query, `GetAllConfigurations`, to obtain its starting point. This is looped over. For each configuration, a few sanity checks are performed. The inner loop on lines 9-11 checks all variants that are targets of `has` edges, and sees if all necessary modules of these variants are part of the configuration currently selected by line 6 from the set iterated in line 5.

*Closures* – Closures are a common feature in functional programming languages, such as Haskell, ML and Elegant. They combine well with data structure navigation features for writing tree transformations. Dynamic rules, discussed later, share many properties of closures, but come with some unusual semantics for scoping and visibility.

*Editing Operations* – The FermaT language does rewriting using editing operations such as cut, copy, paste and delete. There is a requirement placed on how these operations are used. This allows FermaT to guarantee that any editing on the program code will always result in a syntactically and semantically valid result, though not behaviourally equivalent. A few examples of editing operations:

`@Cut` *// delete the current item and store it in the cut buffer*

`@Paste_Over(I)` *// replaces the current item with I*

`@Rename(old, new)` *// renames a variable throughout the current tree*

`@Delete` *// deletes the current item*

`@Splice_Over(L)` *// replaces the current with with the list L of items*

*Path Expressions* – Path expressions are declarations that express paths through the program code structure. In a sense, it provides a small declarative sublanguage for navigation and matching. The feature is mostly found in program transformation systems with graph representations. These are also found in some tree rewriting languages, such as XSLT. The following PROGRES path expression defines a path

(i.e. a relation) named `needs` from one or more `ATOM` nodes to one or more `MODULE` nodes.

```
path needs : ATOM [0: n] -> MODULE [0: n] =
    ( instance of VARIANT & -v_uses-> )
 or ( <-has- & instance of MODULE & -m_uses-> )
end;
```

It states that there is a `needs` path from an `ATOM` *a* to a `MODULE` *m* if *a* is a `VARIANT` (a subclass of the `ATOM` node type), and there is a `v_uses` edge from *m* to *a*, or if there is a `has` edge from *m* to *a*, *a* is a `MODULE` and there is also an `m_uses` edge from *a* to *m*.

*Logic Predicates, Assertions and Retractions* Predicates are used express queries on the program element fact (PEF) base. A predicate consist of one or more patterns which will be attempted matched against the facts database using unification. Logic assertions are used to enter facts in the PEF base. The facts are terms, expressing relations. Retractions are used in JTransformer to remove facts from the PEF base.

```
1  fullQualifiedName(20003, ?Fqn)
2  importT(10000, 30001, 20003).
3  retract(importT(10000, 30001, 20003)).
```

The predicate on line 1 instantiates the variable `Fqn` with the fully qualified name of the declaration with unique id 20003, which may for example be a class. Line 2 is an assertion of the relation `importT` between its three constant values. Its meaning in JTransformer was discussed in earlier in this section. Line 3 removes the fact asserted by line two from the PEF database.

A general tradeoff common to many of these features is that of expressiveness versus efficiency. For example, allowing existential quantification and universal quantifiers in queries may quickly result in even small queries which become prohibitive to compute on moderately sized graphs.

**Rewrite Rules**

A rewrite rules is a function $r$ which takes a (fragment of a) program $f_0$ to another (fragment of a) program $f_1$, i.e.: $r : f_0 \rightarrow f_1$. $f_0$ is referred to as a *left-hand side* pattern and $f_1$ a *right-hand side* pattern. Determining which $f_0$ a rule is applicable to, and what kind of computational expressiveness is allowed in computing $f_1$, are fundamental considerations.

- *Declaration* – refers to properties of the rule declaration. A declaration of a rewrite rule may keep the *domains separate*, i.e. the left and right hand side may be visually separate in the example above. Alternatively, they may be mixed together, as in the case for congruences. A rewrite rule normally has one left-hand side and one right-hand side, i.e. two *domains*, . In Elegant,
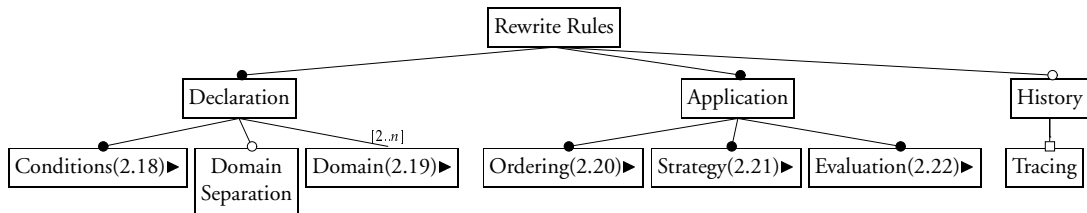
Figure 2.17: Feature decomposition of rewrite rules.

a third variation exists where an arbitrary number of domains can be combined into what is called a relation. Most transformation languages with rewrite rules support *conditions*, see p. 40.

- *Application*– Describes how the application of rules are *ordered*, see p. 43 , and also how the programmer can express *strategies*, see p. 44, for rule application on top of the *evaluation* mechanics, see p. 45, provided by the language.

- *History* – refers to features where the execution *trace* can be recorded.

The rendition of rewrite rules also varies considerably. The previous sections have illustrated examples from both APTS and Stratego. The following rewrite rule is from JastAdd:

```
rewrite Use {
  when(decl() instanceof TypeDecl)
  to TypeUse new TypeUse(getName());
}
```

*Transactions* – Transactions provide concurrency and consistency guarantees to a sequence of transformation operations. The concurrency guarantees allow multiple, simultaneous accessors to the program code. The consistency guarantees that the program code is consistent with respect to a set of invariants after the sequence of operations inside the transaction have been applied. The PROGRES language offers consistency. Concurrency is also supported by PROGRES at the runtime representation level but the language is not concurrent.

*Dynamic Rules* – Dynamic rules are described in Section 3.3.3 in Chapter 3.

**Conditions**   Variation of application conditions for rules, see Figure 2.18, exist in abundance. For purposes of discussion, the feature spaces is divided into four parts, described next.

- *Predicates* – predicates are declarative questions evaluated against the *structure* of the code, or against *relations* constructed from the code.
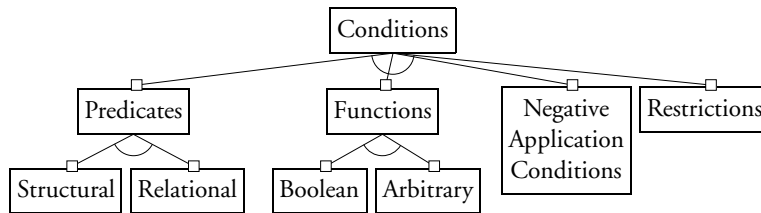
Figure 2.18: Feature decomposition of rule application conditions.

- *Functions* – functions are user-defined algorithms. They may return *arbitrary* results and often allow a more flexible way of encoding predicates into several computational steps.

- *Negative application conditions* – negative application conditions (NACs) are worth mentioning in relation to graph rewriting. Positive graph patterns only pose restrictions on which edges must be exist between nodes. Negative application conditions are used to express which edges *may not* exit. They may also be considered as a variant of structural predicates.

- *Restrictions* – restrictions are a special kind of a pattern found in PROGRES. Restrictions can be named and reused by rewrite rules. When evaluated, they can in turn call out to functions (called queries in PROGRES parlance), which can do arbitrary computations and graph traversals.

Functions and predicates are the primary variation points for conditions. These general concepts take many shapes, such as the negative application conditions and restrictions.

*Unification* – Unification is a generalisation of basic pattern matching. A query with multiple concurrent patterns can contain reoccurring variables which must be instantiated to the same value for each pattern, i.e. they must be unified. Unification is equivalent to instantiation in logic. In logic languages such as Prolog, unification is done against a set of terms, all stored in a facts database. Pattern matching with non-linear patterns can be considered a restricted form of unification; the matching is done against one term, using one pattern, but the recurring variable(s) in the pattern must be instantiated to the same value in all places.

*Node Folding* – Node folding provides a unification-like capability to graph pattern matching. It is found in graph rewriting systems where every pattern match is attempted across all nodes of the graph. In some systems, it is by default required that two different nodes, $n_1$ and $n_2$, in the left-hand side pattern match different nodes in the graph. Node folding allows specifying that $n_1$ and $n_2$ may match the same node.

*Reference Attributes* – Reference attributes allow placing cross-node links in an abstract syntax tree, i.e. links which do not go directly to a parent or a child, turning

it into a abstract syntax graph. In JastAdd, directed equations may be subsequently be expressed on top of the abstract syntax graph, whereas other attribute grammar systems such as Elegant only allows directed equations on the AST. The following JastAdd fragment declares the synthesised (lazily evaluated) attribute `booleanType()` on the `Program` node, which references the definition for the builtin type `boolean`.

```
syn lazy PrimitiveDecl Program.booleanType() =
  (PrimitiveDecl) localLookup("boolean");
```

*Overlays* – Overlays are described in Chapter 3.

**Domains**    Figure 2.19 describes the domains used for pattern matching in rewrite rules.
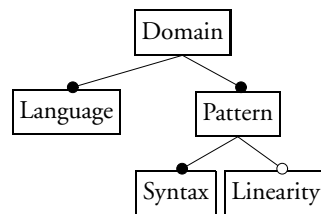


Figure 2.19: Feature decomposition of rule domains.

- *Language* – Specifies which subject language the pattern must be written in.

- *Pattern* – The pattern of the domain is expressed using either *abstract* or *concrete* syntax, with either a graphical or textual presentation, as discussed in Section 2.3.1. When the pattern variables instantiated non-linearly, the semantics is the same as for unification.

The choice of language may be fixed by the transformation system, or it may be user-definable. FermaT (fixed to WSL) and JTransformer (fixed to Java) are examples of fixed systems. Tom, Stratego and Elegant are examples of systems suporing user-definable subject languages.

*List Comprehension* – List comprehension is a language feature that improves syntax for list matching, list iteration and list transformations. The list comprehension syntax is very close to the mathematical syntax and semantics of list (or set) comprehension. This feature is also often found in functional programming languages.

*Pattern Matching* – Pattern matching offers structural matching on program code, either using abstract or concrete syntax. The patterns may contain pattern variables which will be bound during the matching process. Transformation programs using pattern matching on the program model often become tied to the structural details of that model. For example, rewrite rules in term rewriting systems often become closely

tied to the signature they were written against. This makes it difficult to switch or modify signatures, i.e. change or evolve the subject languages, while keeping the rewrite rules.

*Embedding Clauses* – Embedding clauses specify how to rearrange the edges in a graph during a rewrite step once a match has been found. The clauses declare how edges will be changed in the transition from the left-hand side to the right hand side in terms of copy, redirect and remove operations.

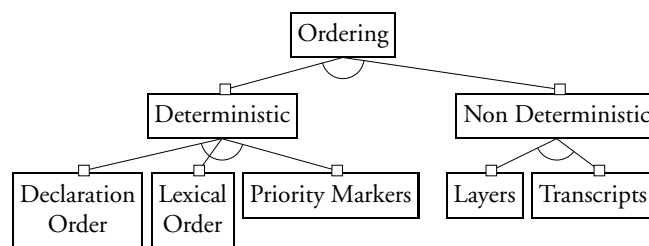**Ordering**   Features for ordering rules are shown in Figure 2.20.



Figure 2.20: Feature decomposition of rule ordering

- *Deterministic* – The selection of rules is completely deterministic.

- *Non-deterministic* – The selection of rules is non-deterministic.

Deterministic languages mostly use the *declaration order* of rules to determine the order, e.g. Elegant, JastAdd . Another alternative is to require explicit *priority markers* on the rules, as for example in XSLT.

*Directed Equations* – Directed equations declare how a given attribute of a node must be computed from attributes on other nodes in the graph or tree. They give both a declaration of the attribute dependencies and the expression for computing the derived attribute value. The following JastAdd fragment declares the attribute `isValue()` to be a synthesised attribute of type `boolean`, and that its value is constantly `true`.

```
syn boolean Exp.isValue();
eq Exp.isValue() = true;
```

The equation may be any expression (which results in a compatible) type. For example, the type of a `VarDecl` may be computed from the type of the declaration of the type of the current variable declaration node, or more succinctly:

```
eq VarDecl.type() = getType().decl().type();
```

*Traversal Strategies* – Traversal strategies are declarations for how to traverse trees, and how rewrite rules should be applied to the tree during traversal, see [Vis05a].

*Backtracking* – Backtracking provides the ability to unroll (a series of) changes made to the runtime representation during transformation, thus reverting to a previous state. As such, backtracking relates to transactions, discussed later. Efficiency of implementation rests on how much data needs to be duplicated for rollback to be possible, whether rollback is local or global, and also the runtime complexity of the rollback algorithms. The performance can be be improved by use of maximal sharing techniques [vdBdJKO00] and lazy evaluation.

*Rule Set Layering* – Rule set layering is a feature for imposing application ordering on a set of rules. The rule set is divided into layers. Each layer will be evaluated with a fixed evaluation strategy, such as fixpoint, until no more rules in that layer apply. At this point, the next layer will be evaluated in the same fashion. Effectively, this divides the application of a set of rules into phases. Layering retains the declarative approach to expressing rewriting systems. It combines well with critical pairs analysis to prove confluence: confluence must be proven on a per-layer basis.

*Tree Cursor* – The editing operations of FermaT always take place at the current position in the tree, maintained by a tree cursor. The cursor can be moved around with navigation commands such as up, down, left and right. For example, the function `@Parent` provides the parent of an item (node), and `I^n` will give the n-th child of an item `I`.

**Strategy**   The application strategy, Figure 2.21, determines how the rewrite rules will be applied to the runtime representation. Application strategies are very much related to ordering and scoping; they determine the location in the runtime representation an atom is applied, in which order, and how application failures should be handled.
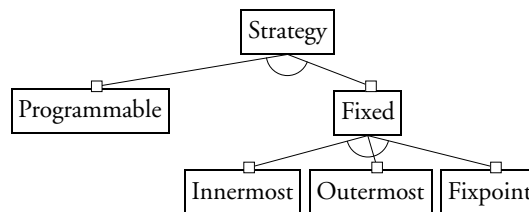


Figure 2.21: Feature decomposition of rule application strategies.

- *Programmable* – The application is programmable by the transformation programmer. Even when strategies are *programmable*, a library of ready made strategies may be available. This is the case with Stratego. Its library provides over a substantial collection of different application strategies.

- *Fixed* – The application strategy is pre-programmed into the transformation language and cannot be changed. Common alternatives are *innermost, outer-*

*most* and *fixpoint*, but the variation is immense. Refer to [Vis05a] for a broader catalogue of common evaluation strategies.

There is a tension between provability and flexibility. Having a fixed of a limited number of evaluation strategies makes analysis of the code possible, for example, critical pairs analysis. Allowing programmers to freely define custom strategies comes with Turing completeness. In general, this removes the ability for automatically proving or guaranteeing termination. It also removes automatic guarantees of confluence. A substantial survey of strategies in rule-based program transformation systems is given in [Vis05a].

*Relation Calls* – Embedded relations provide a limited relational-like functionality in graph rewriting systems. An embedded relation is placed on a node type to tie it to a set of other node types. Inferred links are encoded by path expressions which will be evaluated every time the link is accessed, allowing the members of the relation to change.
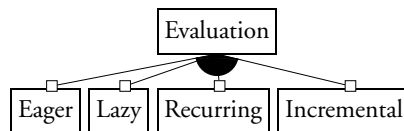


Figure 2.22: Feature decomposition of rule evaluation.

**Evaluation**

- *Eager* – expressions are computed in the order they are seen by the interpreter

- *Lazy* – expressions are not computed until their result is needed. Once evaluated, the result is memoized and used for all future evaluations of this expression.

- *Recurring* – Similar to lazy expressions; the expression is reevaluated every time the result is needed, taking updated values for all involved variables into account. Recurring evaluation is equivalent to lazy evaluation without memoization.

- *Incremental* – attaching a recurring evaluation to a variable gives incremental evaluation: Whenever such a variable is read, the evaluator is run, potentially recomputing all dependent variables which are also incremental.

Many transformation systems seem to be rather sensitive to how transformation algorithms are formulated. As with many high-level languages, developers may inadvertently write sound and clean transformation programs with prohibitive execution times. A number of optimisation features have been proposed for improving

the efficiency of the recommended ways of formulating transformation problems. It may therefore be no surprise that many of the characteristic features discussed in this chapter come from the PROGRES language. PROGRES was designed to make graph transformation practical. It offers a wide range of architectural and language features that aid in writing general graph transformations efficiently.

*Conditional Path Iteration* – Conditional path iterations are user-definable iterations over paths, similar to the mathematical notion of transitive closures on a set of predicates. Conditional paths are found in graph languages, such as PROGRES. Instead of returning all visited nodes, they return all possible termination points. This feature is also found in the tree rewriting language XSLT [Cla99]. An example of this feature was shown in the `needs()` example, under *path expressions* in Section 2.4.2.

*Memoization Markers* – Memoization makers allow programmers to declare that results of computations should be stored and reused whenever the same expression is reevaluated. The feature is found in graph systems with paths and attribute grammar systems, and is used to control recomputation of dependent values. When rules and functions are marked with a memoization marker, it implies that they are referentially transparent. The following JastAdd fragment declares the attribute `x()` of (node) class `A` to be a lazy, synthesised attribute, i.e. that its value should be memoized.

```
syn lazy A.x();
```

*Cycle Detection* – In attribute grammar systems, detecting cycles in the dependencies between attributes is necessary for correct evaluation. The job of cycle detection is to determine whether a given equation directly or indirectly depends on its own value.

*Cycle Breaking* – This feature is dependent on cycle detection. Once cycles are detected, various schemes are possible for breaking them. The simplest is to disallow the cycle altogether by refusing to compile grammar declarations with cycles. Another alternative is to ask the user to manually insert lazy evaluation where appropriate. In some systems, such as JastAdd, cycles are broken with a fixed, but automatic strategy. The following JastAdd attribute declaration specifies that the value for an attribute which turns out to be circular should be `true`.

```
syn lazy boolean ClassDecl.hasCycleOnSuperclassChain() circular [true];
```

*Derived Attributes* – Derived attributes are variables (attributes) inside nodes whose values depend on the value of other attributes. Updating the value of a dependent variable automatically recomputes the value of all its dependents. The dependencies are practically always expressed using directed equations. A typical attribute grammar system will define its attributes using equations, making all attributes derived attributes (except the ones which are defined by constant expressions). Both synthesised and inherited attributes are kinds of derived attributes.

*Finite Differencing* – Finite differencing is a transformation for replacing costly, repeating calculations with less expensive differential and semantically equivalent counterparts. The transformation is independent of the subject language, and mostly useful for algorithms with repeated calculations. A special case of finite differencing is the strength reduction optimisation found in most compilers. A detailed example is beyond the scope of this chapter, but refer to [PK82] for an explanation of finite differencing support in APTS.

### 2.4.3 Typing

The structure of the program code must be captured by the transformation language type system, see Figure 2.23. Transformation languages are primarily meant to work on a restricted domain of data. This opens up the opportunity for custom, or domain-specific, type systems. These may sometimes be simpler than ones found in general-purpose languages.
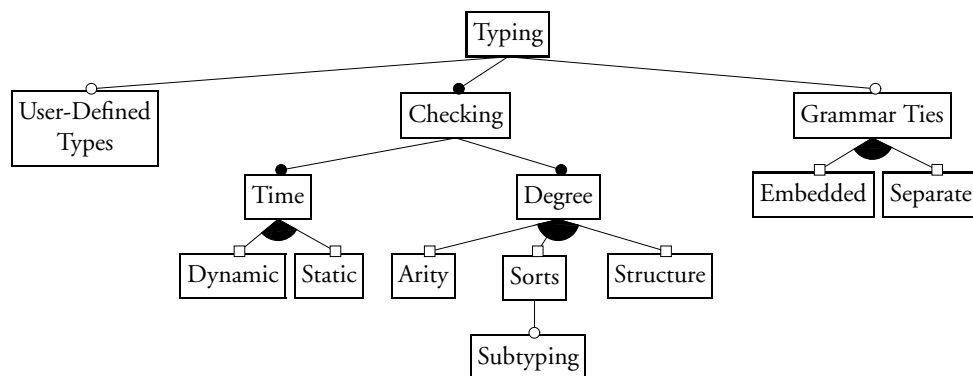


Figure 2.23: Feature decomposition of typing.

- *User-defined types* – The system allows the programmer to define new types.

- *Checking* – Refers to which features exist for checking type correctness.

    - *Time* – Determines when the type checking takes place. For solely *dynamic* type checking, all type checks are performed at runtime, and this may incurs a performance hit. For solely *static* type checking, the transformation program is guaranteed at compile time to maintain type consistency. Most languages fall in between.

    - *Degree* – Describes the nature of the type checking. The type checking may ensure structural validity of the program object model, type correctness or other semantic properties. FermaT, for example, ensures that

every transformation results in a semantically valid, executable subject program.

- *Grammar ties* – For many transformation systems, the data types for the subject code is derived directly from a grammar. In these cases, it is common for the type declarations to be *embedded* in the grammar. For other systems, the definition of the subject code structure is *separate* from the grammar, and grammar-independent.

A characteristic trait of the advanced type systems for transformation languages is that they offer flexible and powerful features for maintaining data structure consistency. The grammar-dependence of the types for the subject code is a characteristic feature of both tree- and graph-based systems. In Stratego, the term structure definition for subject-program terms is usually derived directly from the syntax declaration of a subject language. Some systems completely separate the subject language grammar from the type declaration of the internal program representation of subject programs. It is the programmer's responsibility to convert between the parser output and the type declaration for the subject code. This is the case for JastAdd, where any parser may be used, as long as it builds objects from the types declared in a separate, user-defined JastAdd AST declaration file.

*Transformation Invariants* – Transformation invariants are invariants on the program code which are guaranteed by the transformations. They are encoded as pre and post conditions on the transformation atoms or transactions. Such invariants are very useful for conducting proofs on the transformation program. Often, there is a clear correspondence between the transformation invariants and the data structure invariants discussed in Section 2.3. The PROGRES language can specify graph invariants in its graph grammar, such as the absence of cycles, which must be respected during graph rewriting:

```
constraint ACyclicAggregation = not (self in self.-contains-> +)
```

*Meta Attributes* – Meta attributes are attributes on node types, offered by the PROGRES language. They allow parametrisation of grammar declarations and are similar to (type) parameters on types. Meta attributes confer the ability to compose types at compile time, much like generic types. Consider the following container node class, defined in PROGRES.

```
1  node class CONTAINER;
2    meta ElementType : type in ELEMENT;
3    intrinsic contains:  ELEMENT [0: n];
4    constraint self.contains.type = self.type.ElementType;
5  end;
```

A `CONTAINER` node holds a list of elements of a given type `type`. It may be instantiated for a specific type, such as `Stmt` in the following way:

```
node type StmtContainer : CONTAINER;
  redef meta ElementType : = Stmt;
end;
node type Stmt : ELEMENT end;
```

This defined the container `StmtContainer` which may only contain elements of type `Stmt`. This constraint is ensured by line 4, above, and will be checked at runtime.

## 2.5   Discussion

This survey described and discussed numerous features characteristic to transformation system. Its main focus was on the program models and representations used in transformation systems, and how these relate to the transformation language used to manipulate the models.

The analysis undertaken behind this survey indicates that high-level program models support language-independence well. They often achieve this by replacing language-specific information present at the source code level – such as the difference between `for` and `while` loops in the C-language family – with more general concepts – e.g. bounded/unbounded loop. The abstracted model may often be easy to transform, but translating the result of a high-level transformation back to the underlying program is often difficult. As a consequence, if language-independence via abstract program models is required, many classes of transformations may have to be given up because required information is not present in the abstract model. Abstract models are therefore best suited for capturing problem-specific views on software.

There is a second observation related to the use of abstract program representations. The problem of general graph matching (determining an isomorphism between two graphs) is in NP. It is therefore common for program transformation systems based on graph to extract smaller, more abstract models from a code base. Additionally, general graph rewriting systems provide numerous optimisation features and language constructs for making graph rewriting computationally tractable. Some of these were discussed in Section 2.4.

A similar observation may be made for databases. The author has only found a handful of transformation systems based on relational databases. On the other hand, many analysis frameworks have been constructed by using databases to represent programs.

A remark on the use of meta information (annotations) might be in order. The introduction of meta information often makes transformations easier to write. By separating the logic for computing the meta information from the logic using this information, it is sometimes possible to formulate transformation algorithms in a

more language-independent way. Much of the logic for computing meta-information remains language-specific. By standardising on a given meta information format, large parts of the transformation algorithms may be reusable, however. The tradeoff is that the meta-information may have to be refreshed or recomputed throughout a transformation. Depending on the nature of the annotation, this may very expensive. Many systems, especially those based on attribute grammars, employ lazy evaluation to partly circumvent this problem.

## 2.6   Summary

This chapter presented a detailed survey of the state-of-the-art in software transformation systems and showed that this is a very feature-rich domain where many novel language features have been invented. The survey contained feature models describing central parts of the design space for transformation system. The models were supplemented with examples taken from about a dozen research systems.

The survey indicated that several features for abstracting over subject languages exist, especially for systems with very high-level program representations. A problem with these models is that transformations are difficult to translate back to concrete programs. There is therefore a rather clear case for additional abstraction facilities which provide good language abstraction facilities while simultaneously supporting easy rewriting of programs. In particular, the program model and language constructs for manipulating it are the central components that need good abstraction facilities if one is to attain transformation reuse and language-independence.