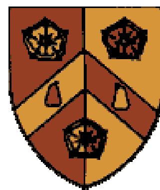
Data Refinement in Intentional Programming

Qualifying Dissertation

Iván Sanabria-Piretti



Wolfson College
University of Oxford
United Kingdom

July 27, 1999

Abstract

Intentional Programming is a new paradigm in software engineering that allows programming languages to be augmented in a highly flexible manner. Under this paradigm, new independent constructs, called *intentions*, are used to represent domain-specific abstractions and actions in the new language. Program realisation is achieved when abstract intentions are automatically converted into more primitive constructs down to an executable concrete program.

In this document, we argue that this conversion process, rather than being a one step indivisible process, should be a multi-step activity where optimising transformations and translations take place. With this aim, we review the contributions that higher order attribute grammars, mechanised data refinement and data abstraction can provide to this multi-step process. While the first one continually supplies updated context-sensitive information, the last two provide optimising opportunities to a rule-based transformation engine. In addition, we study their specification characteristics in order to identify features that an Intentional Programming meta-language should have and to ensure at least an equivalent expressive power. For this, we review the specification mechanisms offered by several existing systems that are based on these formalisms. We also develop a case study where two intentions and their transformations are specified in an aspect-oriented, sub-task decomposition style using higher order attribute grammars.

In this case study, we discuss the semantic effect that a very simple construct can produce when added in an existing programming language. In particular, we have found subtle relationships between the new constructs and existing ones that need to be clearly specified in order to avoid semantic ambiguities when combined. This suggests that including new intentions require a semantic analysis of their interactions with existing ones; which goes against our aspiration for intentions to be completely independent building blocks.

Our initial research has also yielded several open questions that need deeper understanding and study before an answer can be given. We plan to be able to answer them through the next stages of our research; for this, a research plan is proposed.

Acknowledgements

I would like to thank Microsoft Research for its generous financial support and Oxford University for its ORS award. I would also like to thank Mike Spivey for suggesting the key ideas about the case study and the detailed review of this final document; also Oege de Moor for introducing me to attribute grammars and his constructive comments on an earlier draft. The Comlab Programming Tools Group has contributed substantially through our meeting discussing on the Polya system, the case study, and the SG and Eli systems. Thanks also to Tony Hoare who read an earlier draft of the data refinement section and to Eric van Wyk for his encouraging comments on almost every part of this document. Aswin van der Berg kindly granted me access to his implementation of Polya and his comments. William Waite and Uwe Kastens guided me through the initial steps understanding the Eli system vision and using the actual system. Lex Augusteijn helped getting started with Elegant and enriched my understanding through his comments.

Contents

1 INTRODUCTION 1

2 INTENTIONAL PROGRAMMING 4

2.1 A META-LANGUAGE FOR IP 5

3 ATTRIBUTE GRAMMARS 7

3.1 DEFINITIONS 7
3.1.1 A functional implementation of attribute grammars 8
3.1.2 Attribute grammars as aspect-oriented programs 10
3.2 HIGHER ORDER ATTRIBUTE GRAMMARS 10
3.3 ATTRIBUTE COUPLED GRAMMARS 11
3.4 SHORTCOMINGS OF AGS 11

4 CASE STUDY 13

4.1 THE P LANGUAGE GRAMMAR 14
4.2 TRANSFORMATIONS 16
4.2.1 After Expression 16
4.2.2 While and Loop statements 23
4.3 A HAG FOR OUR LANGUAGE P 23
4.3.1 The Code and NewFncDcls attributes 24
4.3.2 The Block structure and its attributes 28
4.3.3 The Value Attribute 30
4.3.4 The Type Attribute 30
4.4 EXAMPLE 31
4.5 FINAL REMARKS 33

5 A REVIEW OF THE ELI SYSTEM 35

5.1 COMPILER SPECIFICATION 35
5.2 SYMBOLS AND RULES 36
5.3 DUAL USE OF ATTRIBUTES 36
5.4 REMOTE DEPENDENCE 37
5.5 SYMBOL COMPUTATIONS 39
5.6 INHERITANCE 40
5.7 MODULES 41
5.8 CUMULATIVE ATTRIBUTION 41
5.9 FINAL REMARKS 41

6 REVIEW OF OTHER SYSTEMS 43

6.1 AML 43
6.2 ELEGANT 44

7 DATA REFINEMENT AND PROGRAM TRANSFORMATIONS 46

7.1 THE POLYA SYSTEM 47
7.1.1 The Transforms 48
7.1.2 Program Transformation 50
7.2 DISTIL 51
7.3 POLYA AND DISTIL 53

8 DISCUSSION 54

9 FUTURE WORK 59

10 REFERENCES **54**

11 APPENDIX **66**

11.1 APPENDIX 1: SEMANTIC DEFINITIONS FOR *P* USING SG 66
11.2 APPENDIX 2: SCOPE ANALYSIS IN PASCAL USING ELI 74

1 Introduction

Intentional Programming (IP) is a new paradigm in software engineering that allows programming languages to be implemented in a highly extensible manner [3, 56, 58]. An IP language-independent framework and programming environment has been under development at Microsoft Research since 1991 [3].

A sibling research project "A Meta-language for IP" started in October 1998 at the Oxford University Computing Laboratory. As its name suggests, its specific goal is to create a meta-language for IP that allows the specification of intentions and their transformations, abstracting the programmer from the order in which they may be applied over program representation structures. In IP, transformations can be used for multiple purposes, for instance program optimisation, program translation and even editing activities within an IP environment [2, 47, 58]. The IP meta-language will hopefully contribute towards the IP goal of having complete, industrial-strength languages implemented entirely as collections of transformations, and will offer IP programmers a notation for effectively expressing their intentions.

It would be ideal if we could convert an abstract program specification based on the IP programmers' intentions into one of its optimised concrete representations in a single step process. This is not easy to achieve, if possible at all. The transformation engine has to arrange and control the application of a large number of small specific transformations and translations without apparent order. But application order of transformations does matter [3]. Optimising transformations can happen only if instances of the appropriate program structures are present, and under certain circumstances they may occur in the program representation as the result of the application of other transformations. Also, the quality of the optimised code can depend on the order in which the optimisations are made. This suggests that the transformation engine needs to apply every transformation or translation at every moment until none can be applied any more, turning it into a lengthy process. One could argue that more complex transformations and translations can reduce the problem size, but the more complex they are the higher their specialisation toward very particular implementation issues.

It is for this reason that we visualise the conversion process as a repetitive combination of transformation and translation steps. At each step, intermediate program structures can be automatically annotated and analysed, offering information about the current optimising opportunities that probably would not be present at previous or later steps of the process. The resulting program structure can then be translated to a more concrete representation; *i.e.*, to a different abstraction level, where the process would be applied again. In this view, the conversion process can be divided into concern stages that could be considered at distinct steps.

The chart in Figure 1 illustrates our general view of the process of converting from an abstract program representation to a concrete program representation. While the angled dotted line represents the ideal single step process, the stepped descent lines represent the same activity as a series of transformation and translation steps. The dashed segments in the latter indicate the occurrence of one or many steps in the process.

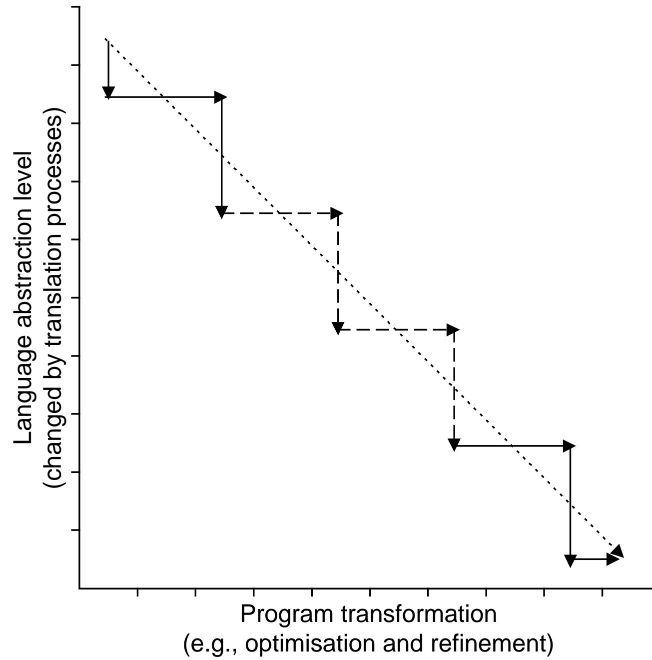


Figure 1. Our view of the conversion process.

On the one hand, program *transformation* in its purest form is a matter of term rewriting. It takes an existing construct within the current program representation and maps it into another construct or constructs of the same programming language, which is also known as *source-to-source* transformation. This mapping is frequently used for program optimisation [14], keeping a balance between abstraction (readability) and efficiency of the program and data structure. Beside, it may consider mappings between different constructs that could provide later optimising opportunities. On the other hand, program *translation* maps expressions between two different programming languages for the purpose of converting a program specification writing in one language into another. Homomorphic relations may be used here as well as in program transformation, but with the difference that they are established between two different languages. This is extensively used in compiler construction [1], where an abstract program specification is typically translated into a concrete specification through a multi-step conversion process. The concrete final specification may well be executable under a given computer architecture. Program translation may be specified using attribute grammars [43, 44], and more recently higher-order attribute grammars [63, 64].

It is with this chart in mind that we will refer to program transformations as *horizontal transformations* and to translations as *vertical transformations* when necessary. We will also refer to these two kinds of transformations simply as *transformations* in a broader sense, making reference to program transformations, program translations or both. The occurrence of a single kind of transformation is also possible.

To this extent, we want the new meta-language to be able to express horizontal and vertical transformations in an expressive, flexible, regular way. For this we study other specification formalisms already in existence that we believe may contribute to enrich our meta-language. They are mechanised data refinement and attribute grammars.

For the purpose of horizontal transformation, O. de Moor and G. Sittampalam are currently working on program transformations using rewrite rules and on higher order

matching [14, 15]. On their view, a more efficient specification can be obtained by successively applying these rules. As an extension to this work, we study mechanised data refinement with two goals in mind: to increase transformation options and to allow transformation layers. Mechanised data refinement can enable further construction of efficient programs by transforming data representations and their operations [50, 53], therefore offering subsequent optimisation paths. The equivalence preservation, which data refinement establishes between various data structures and their algebraic operations, guarantees that rewrite rules can safely replace existing data and control structures with others while preserving the program meaning. Likewise, these data changes may render, according to the user's desires, different levels of abstraction or layers when describing computations performed by the program under transformation. Layers represent de-coupling stages between data representations and algorithms, that allow reasoning about them individually.

We study attribute grammars with two purposes in mind. Firstly, attribute grammars allow the description of vertical transformations; they have many of the characteristics we seek. They allow declarative specification of translation rules; the execution order of these translation rules can be determined statically; and contextual information may affect the final result of the local or global translations among others. Secondly, we consider attribute grammars to be a good mechanism for computing useful context-sensitive information and for updating tree annotations during horizontal transformations, which may lead to improvements in the application of rewrite rules. We believe attribute grammars are also able to compute basic program analysis information during transformation.

Therefore, in this document we review attribute grammars and mechanised data refinement from the IP perspective as a first step towards the definition of our meta-language. We aim at the identification of the minimal set of features the meta-language should provide as suitable notation for describing program translation and data transformation rules.

In order to achieve this task, we review several systems that are based on the attribute grammars and elaborate a case study based on the Synthesizer Generator (SG) [55] and the Eli system [22]. The Eli system suggests a number of useful meta-language constructs for modular compilers; some of which directly find their way into the area of aspect-oriented compilers [13] as well. In addition, we study the way Polya [25] achieves mechanised data refinement and the strategy that the DiSTiL [60] undertakes data abstraction.

The structure of this document is as follows. We first give a brief review of Intentional Programming and what its meta-language should provide. Then we introduce attribute grammars to build the basis for the subsequent two sections: the case study and the Eli system review. It is through the case study that we review the SG specification facilities. The case study also shows our approximation to the way an IP intention should be implemented and deployed using attribute grammars. The Eli system suggests considerable improvements from the viewpoint of basic attribute grammar specifications, among them higher independence and modularity. In addition, a short review of the AML and Elegant systems is presented. Then, mechanised data refinement is introduced separately, where we also study the Polya system in detail and study the DiSTiL's data abstraction machinery. Finally, we present our conclusions and present the future plan of this research.

2 Intentional Programming

Intentional Programming (IP) allows programmers to exploit domain specific abstractions and optimisations to enable higher software reuse and automation of software development [56, 58]. In particular, the programmer can specify new abstractions that are specific to his problem domain while being able to associate with each of them any optimisation that may apply with such new abstractions.

While traditional programming languages may offer features for expressing domain specific abstractions, IP goes a step further allowing the programmer to express his knowledge of domain specific optimisations. The use of optimisations is vital in software reuse. Without them, the only tool to achieve reuse is parametrisation, and this inevitably introduces a performance overhead. However, to realise the potential of such optimisations they have to be described in a compositional fashion, so that they can be reused independently of the context in which they occur, and ensure they will "co-operate" to produce a correct program.

Intentional Programming can be thought as a framework for implementing domain-specific programming languages in a highly extensible manner with the help of a language-independent programming environment [56, 57, 59]. Language independence is achieved in IP by representing all source code (in whatever language) as an *abstract syntax tree* (AST). Nodes of an AST are called *intentions* and correspond to productions of the language. Examples of traditional intentions include `if`-statements, `for`-loops, type declarations, assignment statements, etc. Thus libraries of intentions can be created for representing programs in various programming languages. Plug-ins are used to share these libraries or simply particular language features. Many intentions are themselves language-independent; *i.e.*, their semantic meaning (but not their syntax) is shared in many languages. The if-statement, for example, with a general form of an `if` operator and a 3-tuple argument `<boolean-expression, then-statement, else-statement>`, is a standard intention in virtually all programming languages.

In an IP programming environment, the syntax and external representation of an intention is user-controlled, allowing them to be used in different languages. This variability is captured by *parsing* and *unparsing* methods that are associated with intentions. Parsing is understood as the process of converting plain text into the AST representation. It allows the reengineering of legacy code; *i.e.*, the user can recover legacy code written in a *non-intentional* way and *intentionalise* it by expressing it in terms of intentions. Editing activities happen at the AST level, where the user is provided with convenient editing tools. Unparsing is understood as the process of displaying an AST to the user for direct manipulation, where it is more than a pretty-printing as the process is fully graphical. This can allow domain-specific languages to express solutions in the problem-domain language (like a combinatorial formula) rather than in a given solution-domain language [56, 60].

The extensibility of IP lies in its ability to define new intentions and to define *enzymes*. New intentions express domain-specific programming constructs, which in fact correspond to extending the language's set of available constructs, therefore extending its grammar. New constructs have exactly the same status as those that originally make up the programming language. A programmer would typically take an existing set of intentions (such as C) as his starting point, and extend it while exploring a specific problem domain. Next he compiles the transformation code associated with the new intentions, which can then be used to build application programs. The process of compiling by transformation is named *reduction*.

New intentions require the definition of *xmethods*, which define the behaviour of the intention. Typically, the behaviour of a new intention will include:

- methods describing its transformation, which tell how the new abstractions can be expressed in more primitive, existing intentions,
- knowledge about optimisations the programmer wants to encode; this involves pattern recognition within the program tree structure of optimisation opportunities,
- ways the new intention should be laid out in program representations for different programming languages,
- methods for describing how it is rendered on the screen, and
- other aspects of its behaviour such as its role in type checking (if any) and debugging.

One of the current challenges in IP is for intentions to co-operate with other intentions in order to produce a correct program, even considering that they represent highly independent processes. Co-ordination problems are likely to occur when the behaviour of an intention produces changes that affects the environment shared with other intentions; therefore invalidating relevant contextual information for their ongoing execution.

Enzymes are transformations on ASTs that also contribute to the extensibility of IP. In the current reduction engine, called *R5* and still under development at Microsoft, enzymes can be understood as functions that extend the current AST with other ASTs. Enzymes are *additive*; *i.e.*, they do not replace or substitute subtrees but only add subtrees to the existing AST so that rollback can be possible. Enzymes are used to transform new intentions into existing ones or straight into the engine's output language, called *R-code*. In R5, their implementation is based on questions, and node and link creation facilities. Each enzyme consists of one or more question handlers that can be asked within the local scope of the AST node it is associated with. A question handler can obtain information from other regions of the AST by asking questions and waiting for an answer [46]. We have studied this interface through an enzyme specification example [8].

2.1 A meta-language for IP

There is no fundamental limitation to the extension approach to a programming language mentioned above. But there are two essential elements required, a language extensibility mechanism and a powerful meta-programming system that ensures independence between its components.

For a minimal IP environment to support the extensibility and independence mechanisms, it has to offer the user the following features:

- to view programs as abstract syntax trees where transformations can take place,
- to offer a completely unconstrained language for expressing transformations, so that any computation for applying domain specific optimisations can be conveniently and efficiently expressed,
- to have facilities for making intentions and their transformation as compositional as possible. This means that new intentions can be added and reused with only minimal knowledge of existing intentions and their semantics, in particular the transformation order of intentions cannot be fully specified by the programmer.

While the current IP environment satisfies the first two points in its own way, the last two points still require further research effort. This is precisely the observation that motivates our project. The current IP abstract machine is designed for executing meta-programs; it is similar to those evaluating attribute grammars [4]. It would also be a good idea that the meta-language would be self-applicable, where the problem domain of program transformation itself could profit by the programmer's expertise.

The use of an appropriate meta-language will contribute to attaining the IP goal of having complete, industrial-strength languages implemented entirely as collections of enzymes. For this, the meta-language has to offer notation facilities that allow the programmer to abstract from implementation issues in IP and hopefully its evaluation model. Specifically, it has to offer:

- A notation for describing transformations and their actions: the context information required by a transformation should be as unconstrained as possible, so that transformations can co-operate as independent components.
- A notation for specifying the application order of transformations: it could be that this control information had to be given for each transformation separately, ensuring its independence. When a transformation is added to the system, it will co-ordinate its own interaction with existing transformations.

These notation features and their semantic model are still open questions that are being addressed by the research group at Oxford. In an ideal scenario, an intention would operate on a particular aspect of a given program with just the required contextual information and anonymous interactions with other intentions if any. In addition, the application order of intentions would not be established beforehand but by the context in which they occur. Regarding data abstractions, it should also be able to allow programmers to concentrate on the task of devising powerful data structure abstractions without worrying about the infrastructure support.

3 Attribute Grammars

3.1 Definitions

Attribute grammars (AG) are a formalism for the specification of context dependent computations and dependence on tree structures. AGs are described by an underlying context-free grammar (CFG) and are based on a formal calculus introduced by Knuth [43, 44]. More recent overviews and applications can be found in [16, 33, 34].

The following definition of attribute grammars is taken, almost literally, from [66].

Definition 1 An attribute grammar is a quadruple AG = (G, A, R, B), where

$G = (T, N, P, Z)$ is a reduced context-free grammar,

$A = \bigcup_{X \in T \cup N} A(X)$ is a finite set of attributes,

$R = \bigcup_{p \in P} R(p)$ is a finite set of attribute computations, and

$B = \bigcup_{p \in P} B(p)$ is a finite set of plain computations and conditions.

$A(X) \cap A(Y) \neq \emptyset$ implies $X = Y$. For each occurrence of X in the derivation of the sentence of $L(G)$, at most one attribute computation is applicable for the computation of each attribute $a \in A(X)$. For each application of a production p in the derivation of a sentence of $L(G)$, all computations of $R(p)$ and $B(p)$ are carried out. □

Elements of R have the form

$$X.a := f(..., Y.b, ...)$$

In this attribute rule, f is the name of a function, X and Y are non-terminals and $X.a$ and $Y.b$ denote attributes. We assume that the functions used in these attribute rules are strict. Attribute rules are associated with its productions. Associations to symbols of the CFG describe results or effects of computations. Definitions and uses of attributes express dependencies between computations.

Definition 2 For each $p : X_0 \rightarrow X_1 ... X_n \in P$ the set of *defining occurrences* of attributes is $AF(p) = \{X_i.a \mid X_i.a := f(...) \in R(p)\}$. An attribute *X.a* is called *synthesised* if there exists a production $p : X \rightarrow \chi$ and *X.a* is in *AF(p)*; it is *inherited* if there exists a production $q : Y \rightarrow \mu X \nu$ and $X.a \in AF(q)$. □

AS(X) is the set of synthesised attributes of X. AI(X) is the set of inherited attributes of X.

The context-free grammar defines the structure of the tree being decorated. Each production describes a *context* consisting of a node and its children (if any). Attribute values decorate the nodes, attribute computations specify how those values are related, and plain computations extract information for other processes.

The order in which computations are written down is irrelevant. Attribute computations describe relationships between attributes, *not* an algorithm for computing the values of

those attributes. That is, an AG specification does not contain any explicit sequencing of the computations apart from those functional dependencies. But, if certain formal restrictions hold, an evaluator can be derived systematically from an AG specification; the designer avoids the need to think about tree traversal strategies and ways to store values used temporarily during the decoration of the tree.

Attribute grammars have shown themselves to be a useful formalism for specifying the syntax and the static semantics of programming languages, as well as for implementing syntax-directed editors, compilers, translator writing systems and compiler generators, and, more generally, any application that has a strong syntactic basis. The large body of literature on theoretical aspects, applications and systems based on attribute grammars shows how active this research area currently is. A bibliographic sample is available in [51].

3.1.1 A functional implementation of attribute grammars

Attribute grammars are used to specify the semantics of programming languages. They specify the computation of attribute values attached to nodes in a structure tree. An attribute grammar specification can be transformed into a compiler. A compiler based on attribute grammars usually consists of two parts: the first parses the input and builds a structure tree; the second part, the attribute evaluator, decorates the structure tree; *i.e.*, it evaluates attributes that are attached to the nodes of the tree. Traditionally, evaluation walks the structure tree; during each visit to a node a subset of the attributes attached to the node is evaluated.

An alternative way to structure a compiler based on attribute grammars is to let the first part of the compiler construct the dependency graph of the structure tree of the input program. The second part of the compiler will reduce the constructed graph. Nodes in the graph correspond with attribute occurrences. A node that corresponds to an attribute *a* is labelled with the semantic function defining the value of *a*. If attribute *a* directly depends on attribute *b* there will be an arc from the node corresponding with *a* to the node corresponding with *b*.

An attribute evaluation scheme that explicitly constructs the dependency graph and then reduces this graph is called a 2-phase evaluation scheme. The first phase builds the graph. The second reduces the graph.

In this approach attribute values are viewed as terms. A term is either a basic value or a function applied to a list of terms. The basic values in the terms are the basic values in the attribute grammar, like integer and characters. The function symbols in the terms are the names of the semantic functions in the attribute grammar. An attribute evaluator must compute the synthesised attributes of the root of the structure tree. The dependency graph is a representation of these attributes.

As presented in [45], the 2-phase attribute evaluation scheme can be implemented in a functional language with lazy evaluation and local definitions. A mapping can be defined so that it maps an attribute grammar into a functional program. This program takes as input a structure tree corresponding to the underlying context free grammar of the attribute grammar. Trees can be represented as lists. Every node consists of a marker and another lists representing the subtrees of the node. The marker in a node determines the applied production rule.

Pattern matching can be used to distinguish between different productions with the same left-hand side non-terminal. These program patterns are denotations of finite lists; the first element is the marker, which is a constant; the other elements are identifiers. The use of patterns in the function definitions is not essential. The different productions of a non-terminal can also be distinguished in the body of the functions by using conditional expressions.

Let us assume that an attribute grammar AG = (G, A, R, B) is given, and $B = \emptyset$. Assume, without loss of generality, that for all X in N

$$\mathrm{AI}(X) = \{X.inh_0, ..., X.inh_{k_X - 1}\}$$

and

$$\mathrm{AS}(X) = \{X.s_0, ..., X.s_{l_X - 1}\}$$

So X has k_X inherited and l_X synthesised attributes.

A non-terminal N_0 is translated into a function *eval_N₀*. The first argument of *eval_N₀* is a labelled tree. Production $p : N_0 \rightarrow N_1 ... N_n$ is translated into a definition for *eval_N₀*:

$$eval_N_0(p, L_1, \dots, L_n)\ inh_0^0 \dots inh_{k_{N_0}}^0 = (s_0^0, \dots, s_{l_{N_0}-1}^0)$$

where BODY(p)

BODY(p) is the translation of R(p), the attribution rules for p. For every attribution rule, defining a synthesised attribute of N_0,

$$N_0.s_j := f(...)$$

in R(p), BODY(p) contains a definition

$$s_j^0 := f(...).$$

For every attribution rule, defining an inherited attribute of $N_j \quad (1 \le j \le n)$,

$$N_j.inh_i := f(...)$$

in R(p), BODY(p) contains a definition

$$inh_i^j := f(...).$$

Occurrences of $N_j.s_i$ and $N_0.inh_l$ in $f(\dots)$ are replaced by s_i^j and inh_l^0 respectively. For every N_j, $(1 \le j \le n)$, BODY(p) contains a definition

$$(s_0^j, \dots, s_{l_{N_j}-1}^j) = eval_N_j\ L_j\ inh_0^j \dots inh_{k_{N_j}-1}^j$$

The case $B \ne \emptyset$ is an easy extension of the $B = \emptyset$. The result of an *eval* function is extended with a boolean value. This boolean value indicates whether all conditions in the tree passed to this function yielded true.

Therefore, in this mapping the non-terminals of the grammar correspond to functions, the productions to different parameter patterns and associated bodies, the inherited attributes to parameters and the synthesised attributes to elements of the result value returned by these functions, which may be functions as well. Such a mapping depends essentially on the fact that the functional language is evaluated lazily. This makes it possible that an

input parameter given to a function may depend on the result value of that same function. In functional implementations of AGs the seeming circularity is transformed away by splitting the function into a number of functions corresponding to the repeated visits to nodes. In this way some functional programs may be converted to a form which no longer depends on lazy evaluation. For this, all parameters in the attribute grammar formalism must correspond to strict parameters because of the absence of circularity.

Most functional languages that are lazily evaluated, however, allow circularities. In that sense they may be considered to be more powerful than non-circular attribute grammars [30]. More detailed discussion on the functional implementation of attribute grammars can be found in [6, 13, 29, 52, 63].

3.1.2 Attribute grammars as aspect-oriented programs

In their pure form, the only way attribute grammars are decomposed is by productions. Productions force the inclusion of all semantic attributes simultaneously, without the possibility of breaking them into different concerns. This is, it is not possible to separate out a single semantic aspect (such as the *environment*) across all productions, and then add that as a separate entity to the code already written. Many specialised attribute grammar systems offer decomposition by aspect, but at a *syntactic* level, not at a *semantic* one.

Aspect-oriented programming [41] offers a solution for problems of software modularity, but still at a syntactic level only. It introduces a new software unit, called *aspect*, that appears to provide a better handle on managing cross-cutting concerns. Like objects, aspects are intended to be used in both design and implementation. During design the concept of aspect facilitates thinking about cross-cutting concerns as well-defined entities. During implementation, aspect-oriented programming languages make it possible to program directly in terms of design aspects, just as object-oriented languages have made it possible to program directly in terms of design objects.

In [13], aspect-oriented programming and functional programming are combined to provide aspects with semantic content for compiler construction. Here, aspects are composed as functions; therefore independent semantic units are available as first-class building blocks that can be parameterised, manipulated and compiled independently.

From the point of view of attribute grammars, an aspect is understood as a set of definitions and computations of one or more related attributes [13]. Examples of aspects in an AG-based compiler are the *environment*, *type* and the *code* attributes; they may exit across all production rules of the abstract syntax specification. This semantic view of an attribute grammar is helpful when additional 'semantic layers' are to be added without affecting exiting aspects.

3.2 Higher order attribute grammars

Higher order attribute grammars (HAGs) [64] are an extension of normal attribute grammars in the sense that the distinction between the domain of parse-trees and the domain of attributes has disappeared: parse trees may be computed in attributes and grafted to the parse tree at various places. The term higher order is used because of the analogy with higher order functions; a function can be the result or parameter of another function.

In HAGs, abstract syntax trees (*i.e.*, recursive data types) are promoted to "first class citizens", where they can be the result of a semantic function, and they can be passed as attributes. Moreover, in HAGs trees can be grafted into the current tree, and then be attributed themselves, probably resulting in further trees being computed and inserted into the original tree.

Trees used as a value and trees defined by attribution are known as non-terminal attributes (NTAs). Hence, in the definition above computations in $R(p)$ and $B(p)$ can produce NTAs as a result of their evaluation.

3.3 Attribute Coupled Grammars

Attribute coupled grammars [21] were introduced in an attempt to model the multi-pass compilation process. In a multi-pass compiler compilation takes place in a fixed number of steps, intermediate trees are computed as a synthesised attribute of trees computed earlier. These attributes are then used in further attribute evaluation, by grafting them onto the tree on which the attribute evaluator is working. A pictorial description of this process is shown below.

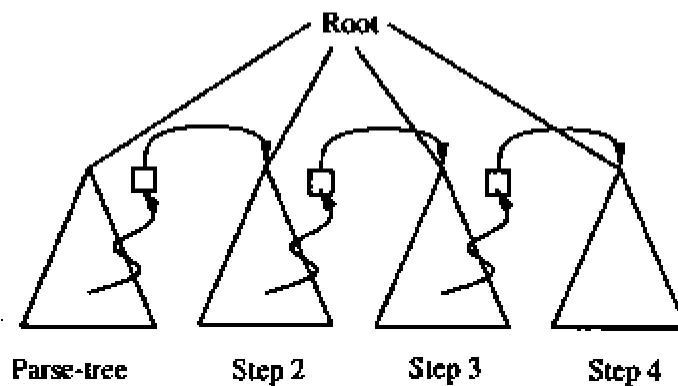


Figure 2. The tree of a 4-pass compiler after evaluation.

This model can be considered as a limited application of HAGs, in the sense that they allow a computed synthesised attribute of a grammar to be a tree that will be attributed again. This boils down to a HAG with the restriction that an NTA may be only created at the outermost level.

The Cornell Synthesizer Generator [55] and the Eli system [36, 67] offer full support for ACGs through computed subtrees. A large example of the application of this multi-pass mechanism can be found in [63] and [64].

3.4 Shortcomings of AGs

Attribute grammars have not come into general use and we may ask why this is, there are several possible reasons. As a compiler specification method, AGs have proven remarkably difficult to decompose into logical fragments [33], and consequently they have not yet been accepted as a viable specification technique.

Another reason for their lack of popularity is the style in which they are normally written – a style that hinders both modular decomposition and reuse of specifications. Moreover, it is often disputed whether AGs are an adequate method for the solution of practical problems. The main arguments presented against the use of AGs in practical specifications are [12, 33, 39]:

- Some AG specifications are as large as or even larger than a manually developed implementation for the same task (especially if the manual implementation uses a modern programming language such as ML).
- The concept of locality in AGs requires much redundant information reducing the comprehensibility of the specification.
- Large AGs lack a structure that improves comprehensibility and maintainability.
- The strict functional and declarative character of AGs discourages the use of certain well-known efficient implementations, hence yielding less efficient solutions.

Attribute grammars, despite their many attractions, are not even in widespread use in the academic community. This may be due to the restrictions imposed by attribute definition languages, which result in a lack of compositionality.

The first three arguments are true if the basic AG concepts are considered, and specifications are written in a notion for exactly that base. Having these shortcomings present, multiple AG systems have been proposed where each of these issues is directly improved. Specifically, in our case study we will be referring to the way SG and Eli solve these problems.

4 Case Study

In this section we present a HAG that transforms an input program in a way that depends on contextual information. This example focuses on the potential side effects that specific expressions can produce on the overall program structure when particular program constructs (*e.g.*, statements and expressions) are to be transformed.

In particular, this example supports our belief that local effects can produce global transformations via attribute inheritance, synthesis and computation — where attributes establish dependencies between subtrees. For instance, global transformations can vary from a simple substitution of the current subtree by an expanded one to the creation of new function declarations and their corresponding function calls in expressions within statements. In the latter case, we will see that more attributions are required in order to propagate the desired effect along the appropriate branches of the working tree.

In this case study, we start from a small grammar with a minimal set of attributes (kernel) that is then extended to support propagation of side-effects via new attributes and computations. This will allow us to recognise the way the kernel should be tuned in order to allow new programming elements in the grammar.

For the sake of comparison, this case study has been implemented using two attribute grammar systems, namely the Synthesizer Generator (SG) [55] and the Eli system [22, 38, 39]. Both systems are known for their particular characteristics in domain-specific applications. The SG is of interest to us because it merges the concepts of abstract-syntax definition, syntax-directed translation and user-defined attribute types, and implements a notation for factoring specifications into separate modules. The SG is strongly based on the construction of language-based editors. The Eli system goes a step further from the basic notation found in typical attribute grammar specifications, improving the abstraction level with concepts of inheritance, ADTs, libraries and modularity that we consider very appealing for our purposes.

As is typically the case when comparing development systems, an exhaustive one-to-one comparison of all system features is not possible, but we will instead present features we consider relevant for our purposes. In particular, we will focus our attention on the notational conveniences these systems provide in order to express subtree dependency, remote access specifications and modularity. By notational convenience we mean the existence of programming constructs that relieve the user of specifying tedious mechanical attribute manipulations and computations. These manipulations are typically used in AGs to allow upward and downward reference to attributes found in other parts of the tree structure. Other constructs can improve modularity and reuse. For a more comprehensive overview of systems based on attribute grammars see [16].

The analysis of the AG evaluation methods used by SG and Eli are beyond the scope of this study. An extensive survey of attribute evaluation methods can be found in [4] and [20]. The quality of the output application produced by these systems — size, performance — is not considered for analysis in this study either.

In the following sections we introduce the toy programming language used, we elaborate on the two transformations of interest in this case study, and we explain a HAG implementation that achieves these transformations.

4.1 The P Language Grammar

For the purpose of our Case study let us consider a small imperative block-structured language, which we will call ***P***. ***P*** will be the grammar of our input and output programs, and corresponds to an extension of the language proposed in [55]. We will concentrate our attention on the existing computations at the abstract representation level of ***P***. Besides, we will assume that the appropriate scanning, parsing and unparsing mechanisms are in place; therefore they will not be discussed in this document.

Figure 3 shows the abstract grammar of ***P***, which is composed by BNF production rules as in SG. The start symbol is `program`.

```
program       :     Prog progName progBody .
progBody      :     ProgBody declList funcDeclList block .
declList      :     DeclListNil
              |     DeclListPair decl declList .
decl          :     Declaration variable type .
funcDeclList  :     FuncListNil
              |     FuncListPair funcDecl funcDeclList .
funcDecl      :     FuncDecl funcName type block .
block         :     Block stmtList .
stmtList      :     StmtListNil
              |     StmtListPair stmt stmtList .
stmt          :     EmptyStmt
              |     block
              |     Assign variable exp
              |     IfThenElse exp stmt stmt
              |     While exp stmt
              |     Loop block
              |     Return exp
              |     Break
              |     Continue
              |     Label labelName
              |     Goto labelName .
exp           :     EmptyExp
              |     IntConst integer
              |     True
              |     False
              |     Id variable
              |     Equal exp exp
              |     NotEqual exp exp
              |     Add exp exp
              |     Minus exp exp
              |     After stmtList exp
              |     FuncCall funcName .
type          :     EmptyType  |  Integer  |  Boolean .
variable      :     identifier .
progName      :     identifier .
funcName      :     identifier .
labelName     :     identifier .
identifier    :     string .
```

Figure 3. *P* abstract syntax grammar.

From this grammar it can be seen that any program in ***P*** has a name (`progName`) and a program body (`progBody`). The program body represents the description of its algorithm; it consists of definitions and a `block` of statements. Moreover, the `progBody` production defines the semantic context where any program declaration and the actual statements —

for which these declarations are visible — exist. As usual, an environment attribute will be used for representing such scope visibility.

A list of variable declarations (`declList`) and a list of function declarations (`funcDeclList`) form the program declaration section. Both lists will be useful for semantic analysis and type checking. In general, we represent lists as a set of two productions: an empty list production and a list concatenation production. The use of Kleene closure definitely would have made list definitions generic, clearer, and more concise (*e.g.*, `declList : decl*` or `funcDeclList : funcDecl*`). But in SG and Eli non-terminal production rules require an *operator-name*, which actually builds the production structure from the current parameters. Then we could declare list definitions as `declList : DeclList(decl*)`, where **`DeclList`** is the operator-name. This last abstraction is possible only in Eli as its language pre-processor replaces it by two automatically generated production rules. Operator-names in SG require a pre-defined number of parameters, therefore the *star* notation is not provided and the corresponding two production rules have to be provided explicitly. For example, in the case of `declList` its production rules are specified by two operator-names **`DeclListPair`** and **`DeclListNil`**, the first take two parameters, `decl` and `declList`, while the second function has no parameters. This makes list concatenation cumbersome, because it will be specified differently depending on the kind of elements being concatenated. SG overcomes this by allowing an annotation that a production rule represents a list, then it allows the use of a "generic concatenation operator" (`::`).

A feature in ***P*** is the occurrence of production rules that represent "emptyness" or the absence of information; namely the terminal symbols **`DeclListNil`**, **`FuncListNil`**, **`StmtListNil`**, **`EmptyStmt`**, **`EmptyExp`** and **`EmptyType`**. They are typically found as the first production rule of the sequence of rules associated with a non-terminal symbol. Their existence has to do directly with the way the SG displays a new "empty" program for editing — it corresponds to an empty instance of the corresponding non-terminal. When an empty program is displayed, the user then edits this initial empty program by replacing these symbols with other symbols. Empty symbols are also used for semantic consistency during editing, and some are used later on in semantic and type analysis.

A `block` consists of a statement list (`stmtList`) and it primarily holds contextual information for the different statements that it may contain. The true importance of this construct only appears later when we discussed HAGs. The `block` production serves several purposes depending on the context in which it is found in the grammar. Specifically, a `block` can be either part of the `progBody` of a program or part of every function body or part of any `loop` statement. In the first two cases, a `block`'s information is used to restrict the occurrence of `return` statements to function bodies — a `return` statement within the main program body is not allowed in ***P***. In the third case, a `block`'s information is used to restrict the occurrence of `break` and `continue` statements to `loop` statements. Furthermore, it offers labelling information that is useful when transforming loop structures into a combination of conditionals and `goto` statements. A less interesting use of `block` is found when considering the block production as a statement (`stmt`). Here, a `block` simply represents a wrapper for a `stmtList` to be manipulated as a single statement.

Notice the difference between the body of a `while` statement (a `stmt`) and the body of a `loop` statement (a `block`). We have established it on purpose to ensure that `loop` statements always have context information associated with its body.

A function in ***P*** does not contain a declaration part or parameters; *i.e.*, functions do not declare a new local scope. It only consists of a `name`, a `block` that describes its algorithm and an explicit `type`. Functions can return values of any of the available types: **`Integer`**, **`Boolean`** or **`EmptyType`** values. The third type, **`EmptyType`**, is used to represent the absence of known type so far as mentioned before, which becomes useful during program editing and type analysis.

As presented in this abstract grammar, a `progName` is an `identifier`, which in turn is a *`string`* terminal symbol. This is also the case for function names (`funcName`), labels (`labelName`) and variable names (`variable`). This string classification in ***P*** allows domain separations. Entities may be defined both by the grammar and by the values representing the terminal symbols: the grammar selects a particular kind of phrase, while the instances of this phrase are differentiated by the values of their terminal symbols.

Finally, a variety of statements (`stmt`) and expressions (`exp`) are available in ***P***. In the case of statements, they correspond to classical control structures found on block-structure programming languages, and have been selected accordingly to the transformations we are interested to develop in the case study. For instance, a `while` statement can be transformed into a `loop` statement with conditional, `break` and `continue` statements. Moreover, a `loop` statement can be transformed into a conditional statement using `goto` statements. The only unfamiliar statement that is worthwhile mentioning is the **`EmptyStmt`**. It is used in the grammar to emphasise that a dummy statement is allowed.

Regarding expressions, the `after` expression (`After`) is undoubtedly the most interesting. Structurally, it consists of a list of statements (`stmtList`) and an expression (`exp`), which we will later refer as the *side-effect* and the *value* of the after expression respectively. Semantically, the `after` expression reads as follows: expression `exp` can be evaluated once after the list of statements `stmtList` has been executed. Notice that this expression may be transformed into more basic imperative statements and expressions in several ways. We will discuss the transformation options in the next section.

The evaluation order of expressions in ***P*** is `after` expressions first, then function calls, addition and subtraction and finally comparisons. When more than one element with the same order is executed on the same expression, a left to right traversal will establish the order.

4.2 Transformations

Recall our interest in HAGs that transform input programs in a way that depends on contextual information. Our discussion of transformations will be based mainly on the semantics of the `after` expression, although we will also consider the productions associated with the `while` and `loop` statements. The discussion of these examples will help us understand the way intentions can be specified using HAGs and could possibly be expressed in our future IP meta-language.

4.2.1 After Expression

Let us consider the `after` expression:

```
exp          :    …
             |    After stmtList exp
             |    … .
```

Recall its informal semantic definition again: expression `exp` (its value) can be executed once after the list of statements `stmtList` (its side-effect) has been evaluated. In the following example variable *y* will be assigned the value of *x + 3* immediately after the side-effect statement *x := x + 1* has completed. A C/C++ programmer would understand it as:

```
y := after (x := x + 1) (x + 3);     ⇔    y := (x := x + 1, x + 3);
                                          (i.e., y := (x++, x + 3);
```

Here, the comma (,) represents an expression list to be evaluated left-to-right, and the value of the left expression, in this case the assignment statement `x := x + 1`, is discarded. All side-effects from the evaluation of the left operand are completed before beginning evaluation of the right operand [40]. But such a construct is not allowed in ***P***, because expressions do not allow statements as part of them — except for the `after` expression itself. For this specific example an equivalent transformation in ***P*** is:

```
y := after (x := x + 1) (x + 3);        ⇔       x := x + 1;
                                                y := x + 3;
```

The type of an `after` expression is the type of its `exp` value, in this example **`Integer`**. If no side-effect is specified within the `after` expression then no transformation is required.

The combination of the `after` expression with other constructs may lead us to conclude that just one transformation may not be enough. For example, consider its combination with a `while` statement and the following transformation:

```
while (after (n := n + 1) (n < 10)) do     ⇔     n := n + 1;
   begin                                        while n < 10 do
      …                                            begin
   end;                                               …;
                                                      n := n + 1;
                                                   end;
```

In this example, the ellipsis represents any statement within the body of the `while` statement. In this case, the side-effect statement is involved in initialising and maintaining the `while` statement invariant. Here, there are two places where the side-effect is required: before the `while` statement itself and as the last statement of the `while` body.

`After` expressions in nested expressions

As expressions are defined recursively, an `after` expression can also have other `after` expressions as part of its side-effect or value. In such cases, we propose that a left-to-right traversal of the expression will determine the side-effects' evaluation order. The following example presents such a case:

```
… after (y := z)                    ⇔  y := z;                    ⇔  y := z;
   (after (a := b)                      … after (a := b)               a := b;
      (after (y := a)                      (after (y := a)             y := a;
               y)) …;                               y)                 … y
                                        …;                             …;
```

In this example, the ellipses represent the current statement where the expression appears. Here, we have chosen to follow an outer-most transformation style. Through the outside-in process, a nested sequence of `after` expressions can be represented as a sequence of side-effect statements followed by the last value expression. We have chosen an outside-in transformation order. This is, they may be represented as trees so that during tree transformation side-effects move up the tree, leaving the value at the bottom. A traditional evaluation order will determine which side-effect will move first up the tree. Therefore, if we consider the tree representation of any arbitrary legal expression and a variable is modified multiple times by side-effects of several `after` expressions, then the actual value associated with this variable will be determined by the closest side-effect to the value expression in the complete tree representation.

Notice in this example that using an inner-most transformation style would produce the sequence of side-effect statements to be in reverse order. This shows that different transformation orders may produce semantically different code.

Let us consider another expression:

```
…(after (x := x + 1) x) + (after (x := x * 2) x) …
```

Here the occurrence of more than one `after` expression shows the importance of the evaluation order. If we apply a left-to-right evaluation order we obtain:

```
x := x + 1;                              ⇔   x := x + 1;
… x + after (x := x * 2) x) …                x := x * 2;
                                             … x + x …
```

Notice the difference of this result when compared against the output of the same expression but in right-to-left evaluation order:

```
x := x * 2;                              ⇔   x := x * 2;
… (after (x := x + 1) x) + x …               x := x + 1;
                                             … x + x …
```

`After` expression in compound expressions

Now, consider the use of an `after` expression in a compound expression as in the following assignment statement:

```
y := x + after (x := x + 1) (x + 3) + x ;
```

We identify several possible interpretations for this expression based on the side-effect scope of the `after` expression. That is, they depend on the way we consider side-effects should affect the entire expression it exists within. Obviously other interpretations are possible by changing the semantics of the distinct constructs involved, in this case the assignment statement and the `+` operator. Once an interpretation is selected, an appropriate transformation can be given.

First Interpretation: Let's consider again the C/C++ interpretation again, using a left-to-right evaluation order. According to the semantics given above, it would be understood as:

```
y := x + (x := x + 1, x + 3) + x ;
```

Recall that comma represents an expression list. While the first occurrence of variable `x` on this expression will not be affected by the side-effect of the `after` expression, the value of its last occurrence, on the right, will certainly be. This would be represented in ***P*** as:

```
temp := x + 1 ;
y := x + (temp + 3) + temp ;
x := temp ;
```

Here, the context of the expressions found *before* the `after` expression value, (`x + 3`), is preserved using a temporal variables, which may appear multiple times. In general, this interpretation requires analysis of the side-effect statement list for the introduction of temporary variables. Moreover, extra care is required to ensure that these new variables do not alter the behaviour of the assignment statement containing the compound expression. For instance consider the following case under this same interpretation: `x := x +` **`after`** `(x := x + 1) (x + 3)`, here the `temp` variable does not need to be assigned to variable `x` as done above.

As suggested, interpretations may consider the contribution that any other existent construct provides; for instance, the associative property of the `+` operator produces slightly different parsing trees for this expression, as shown in Figure 4. Nonetheless, the same result is obtained when evaluating from left to right. Completely different results are computed if the order in this expression is changed, for example `y :=` **`after`** `(x := x + 1) (x + 3) + x + x ;` or `y := + x + x +` **`after`** `(x := x + 1) (x + 3);`.

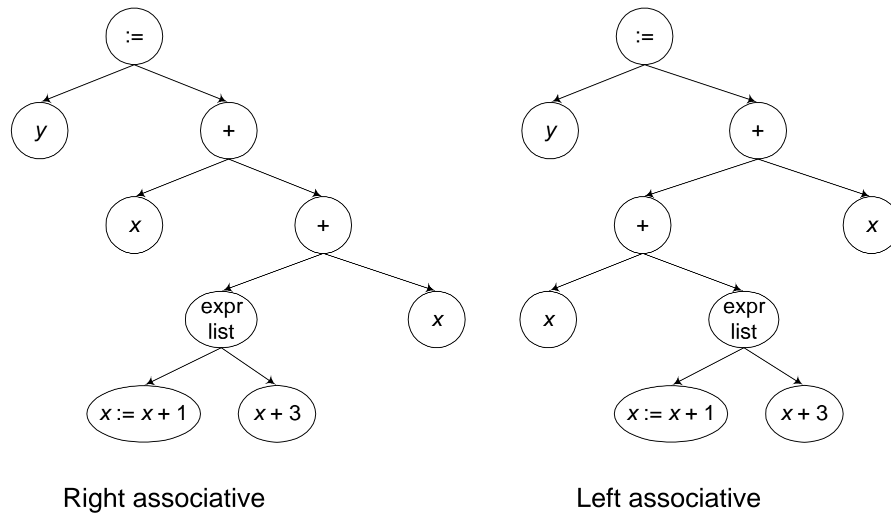


Figure 4. Parsing options of statement
`y := x + after (x := x + 1) (x + 3) + x ;`
considering the associative properties of the + operator.

We may choose to drive away from this interpretation and consider the following ones.

Second Interpretation: The side-effect affects the entire context in which the `after` expression appears as in our initial example. It can be understood as:

```
x := x + 1 ;
y := x + (x + 3) + x ;
```

Any other expression outside the `after` expression will have its context affected, as is the case with the `x` variables outside the parenthesised expression. The parenthesis may establish evaluation precedence, but it is not defined in ***P***.

Third interpretation: The side-effect of the `after` expression is visible only within the scope of its value expression, and in the context immediately after the composed expression. This can be understood as:

```
temp := x + 1 ;
y := x + (temp + 3) + x ;
x := temp ;
```

Here, both `x` variables outside the `after` expression are not affected during the computation of the value to be assigned to variable `y`, therefore preserving its original value. Again, as in our first interpretation, it requires a deeper analysis of the side-effect statement list because temporary variables need to be introduced.

`After` expressions as function calls

Another way of transforming an `after` expression is by converting it into a function. This approach implies the replacement of the actual `after` expression by a function call

expression (`funcCall`) and the construction of the new function to be called. The side-effect part of the `after` expression plus its value part compose the body of such function.

This transformation can be described in a general sense as follows:

```
… after side-effect     ⇔  Function afterFunc (formal_params): type;
        value …             begin
                                side-effect;
                                return (value);
                            end;
                            … FuncCall(afterFunc(actual_params)) …;
```

This is a more interesting transforming option from our point of view, because it reveals the fact that a minor semantic feature in an expression can produce changes to the whole program structure during transformation. In this case, every `after` expression will bring about a new function declaration that is to be included in the list of function declarations (`funcDeclList`) of the complete transformed program. We think an inconvenience of this transformation strategy is the potential explosion of functions and their intrinsic overhead.

Using the same combined `after` expression and `while` statement example presented before, the result of applying this transformation is rather different:

```
while (after (n := n + 1)       ⇔   Function
             (n < 10)) do             AfterFunc_01(var n: Integer):
    begin                                             boolean;
      …                             begin
    end;                                 n := n + 1;
                                         return (n < 10);
                                    end;

                                    while funcCall(AfterFunc_01(n)) do
                                       Begin
                                         …
                                       end;
```

In our basic example, we could also have chosen for function `AfterFunc_01` not to use parameters, because the use of global variables could produce the same behaviour. In fact parameters are introduced here for explanatory reasons, because functions in ***P*** have no parameters.

This alternative transformation shows the importance of considering other programming language features during the semantic interpretation of `after` expressions, namely the evaluation order of expressions (see page 16), and the passing of parameters by value or by reference in function calls, which actually affects function definitions.

Let's make use again the interpretations we presented before (see page 19) and consider the way they can be implemented using function definitions and function calls. The first interpretation can be obtained by allowing functions to produce side-effects. For this, variables modified in the side-effect part of the `after` expression have to be passed by reference to the function. The use of global variables would also produce the same result, but again we prefer to use parameters for explanatory purposes. The transformation of our compound expression is:

```
y := x + FuncCall(
```

```
y := g(f(x))    ⇔    y := after                    ⇔    temp := f(x);
                            (temp := f(x))               y := g(temp);
                            g(temp)
```

Here it allows both function calls to be treated separately. Notice that this is a rather different approach to `after` expressions.

4.2.2 While and Loop statements

In this case study we also consider a two-step transformation that can be applied to `while` statements. The first step transforms a `while` statement into a `loop` statement with `break` statements and optionally `continue` statements. Then a second step transforms this `loop` statement into a `condition` statement where `break` and `continue` statements are converted into `goto` statements. The following example shows this transformation.

```
While (n < 10) do    ⇔   loop                      ⇔   label_01:
   begin                    if (n < 10) then            if (n < 10) then
      a := a * n;              begin                       begin
      n := n + 1;                 a := a * n;                 a := a * n;
   end;                           n := n + 1;                 n := n + 1;
                               end;                        end;
                            else                        else
                               break;                      goto label_02;
                         end;                           goto label_01;
                                                        label_02:
```

In particular, on the first-step transformation the `while` body (`stmtList`) is converted into a `block` according to the `loop` grammar rule. Recall that a `block` has to be created in order to keep different contextual information that supports semantic analysis and later transformations. In this case, it contains information that restricts the use of `break` and `continue` statements, as well as labelling information for potential inclusion of `goto` statements. When the second transformation is applied, the `block` construct is no longer used in the resulting code.

Although this transformation is simple, it gives us insight into the way intermediate transformation steps lay down contextual information. Later transformations may use it or make it available for those program structures that may eventually need it. The transformation of `break` statements into `goto` statements is one such example. First the `break` statement verifies if its presence within the structure is semantically correct (*i.e.*, it exists within a `block` of a `loop` statement), and second it retrieves labelling information from its context in order to transform itself into a `goto` statement.

4.3 A HAG for our language P

Now that we have clearly specified the programming language used and the kind of transformations under study, we elaborate on the attributes our HAG has and their computations. This is a natural common development strategy found in the AG-based compiler literature [31, 33, 39]; this is, attributes are typically defined in terms of the application requirements. Consequently, in this section we will introduce the attributes that decorate our HAG based on ***P***.

The examples presented in this section are extracts of the actual HAG implementation of ***P*** using the SG, available in Appendix 11.1, page 66; we will also refer to the Eli system when differences need to be stated. The SG implementation is chosen because it is more readable than the Eli implementation and requires fewer abstract concepts.

4.3.1 The Code and NewFncDcls attributes

The most interesting attributes in our case study are the `Code` and `NewFncDcls` attributes. They are used directly in the construction of the transformed output program using functions (see section 4.2.1, page 16). They contain computed subtrees that result from the actual production rules they are attached to.

It is precisely the occurrence of subtrees as attribute values that classifies our AG as a HAG. In general, each HAG production rule is able to compute as many different tree representations as required, based on its actual components (their computed values or representations) and the current contextual information. This explains the knitted dependencies typically found within HAG tree structures.

Moreover, `Code` and `NewFncDcls` attributes are synthesised attributes; *i.e.*, subtrees computed at terminal and non-terminal symbols (or nodes in the tree representation) *flow up* the tree, contributing to the total tree under construction. In the case of Code, the complete resulting tree is found at the root production (attribute `program.code`); as for the `NewFncDcls` attribute its complete tree is found at the `progBody` production rule (attribute `progBody.newFncDcls`).

As we will see latter in this section, these attributes are associated with almost every production rule in the grammar ***P***. While `NewFncDlcs` is found only in production rules that allow expressions or contain non-terminals that allow them (from where new function declarations could be produced), `Code` is associated with every production rule without exception.

When such a proliferation of computations occur in a attribute grammar, it is easy to see that keeping attribute specifications as independent as possible becomes an important strategy; this can improve modularity. SG and Eli provide their own mechanisms in order to support this idea.

4.3.1.1 NewFncDcls

The synthesised attribute `NewFncDcls` has three specific goals: to create a function declaration for every existing `after` expression, to collect any new function declarations from the whole tree structure, and to include them in the function declaration part of the `program` body.

The first goal is achieved by computing the function declaration structure based on an `after` expression instance; recall that this is part of its transformation process. Such computation is done as follows:

```
exp:        …
      |     After
              { local funcName      newFuncName;
                local funcDecl      newFunc;

                newFuncName =  FuncName(gensym("AfterFunc_",&($$)));
                newFunc     =  FuncDecl(
```

```
                                    newFuncName,
                                    exp$2.type,
                                    Block(AppendStTail
                                            (stmtList.code,
                                             Return(exp$2.value))));

          exp$1.newFncDcls =
            AppendFncDcl(AppendFncDclLst(stmtList.newFncDcls,
                                         exp$2.newFncDcls),
                         newFunc); }
    |     …;
```

Here the dot notation is used to refer to an attribute of a locally visible production rule instance. When there are multiple instances of the same production rule present, their references are enumerated, for instance `exp$1.newFncDcls` and `exp$2.newFncDcls`. When making reference to attributes of the LHS of the current production rule, the following convention may be used: `$$.<attribute-name>`.

The actual construction of the new function (`newFunc`) is specified using a function `FuncDecl` (recall SG calls this function *operator-name*). This operator requires as arguments a new name (`newFuncName`), which is generated via a `gensym` function and made available as a local variable for further reference when generating its transformed code (refer to attribute `value`); a type, which is the actual type of the `after` expression; and a `function` body. This `function` body is a `Block` consisting of the side-effect code (`stmtList.code`) and a `return` statement as its last element. The value that is returned by this last statement is the `value` of the `after` expression.

This computation also shows the way in which the new declaration is included in the resulting list of new function declarations. Two user-defined functions are used: `AppendFncDclLst`, which returns the list that results from merging two lists of type `fncDclList`; and `AppendFncDcl`, which returns the list that results from appending a given function declaration to the end of a given function declaration list. It is a disadvantage not to have a generic list type. Notice that in the case of nested `after` expressions, it considers the presence of extra function declarations that may be produced within the side-effect statement list and the value expression.

The second goal of this attribute is the collection of these new function declarations. As indicated before in this section, in order to collect them we require these declarations to be passed up the tree, from the actual production instances where they are created (in this case, `after` expression production rules) to the `program` body.

The standard mechanism attribute grammars use for data passing is called *copy rules*. These are rules whose only purpose is to ensure local visibility of attributes that are computed somewhere in the tree. Copy rules can *broad-cast* up and down the tree data. These remote attributes are typically used when computing attributes that are semantically dependent on the context. In the simplest case, by associating copy rules with every possible production rule that may be found between two specific production rules in the grammar, attribute values can be passed up or down the tree.

In the case of our attribute `NewFncDcls`, copy rules are present in almost every production of our grammar (`progBody`, `funcDeclList`, `funcDecl`, `block`, `stmtList`, `stmt` and `exp`). Appendix 11.1, page 66, presents its complete specification. While copy rules are inevitable in SG, Eli provides abstraction mechanisms that relieve the user from specifying them (see section 5.4 below, page 37). A common copy rule is:

```
block :      Block        { block.newFncDcls = stmtList.newFncDcls; };
```

This is a straightforward copy rule found in the `block` production rule: the `newFncDcls` of a `block` (here it could have also been referred as `$$.newFncDcls`) is the `newFncDcls` value of its statement list. Other copy rules in ***P*** do a little more in some cases; they get lists of declarations and merge them into a single list to be passed up the tree. This is typically found in upward copying, where multiple occurrences of the same value are likely to be obtained.

```
exp    :      …
       |      Equal, NotEqual, Add, Minus
                       { $$.newFncDcls =
                           AppendFncDclLst(exp$2.newFncDcls,
                                           exp$3.newFncDcls); }
       |      …;
```

A code reuse facility is exercised here: expressions as `Equal`, `NotEqual`, `Add` and `Minus` are identical in their production structure and therefore could share the specification of (ultimately all) their attribute computations (this is a very useful facility found in SG, whereas in Eli it is available through inheritance mechanisms). For each of these expressions, the `newFncDcls` value corresponds to the union of the `newFncDcls` of their corresponding sub-expressions (here referred as `exp$2` and `exp$3`).

Once copy rules in a HAG are in place for a particular purpose, for instance our task of collecting function declarations, extending it in order to deal with new programming constructs requires minimum modification. A more powerful idea then is to be able to collect elements some type from anywhere in the tree without creating copy rules. This is precisely one of the abstraction facilities the Eli system offer (section 5.4 below, page 37).

Finally, the third goal of attribute `newFncDcls` is to ensure that collected function declarations are included in the function declaration part of the `program` body (`progBody`). From the `progBody` production rule we can see that new functions can result out of the main `block` and any function originally defined in the program (`funcDeclList`). This is done as follows:

```
progBody:    ProgBody  { local funcDeclList initialFuncDecls;
                         initialFuncDecls = funcDeclList;
                         $$.newFncDcls =
                            AppendFncDclLst(
                               AppendFncDclLst(block.newFncDcls,
                                               funcDeclList.newFncDcls),
                               initialFuncDecls); };
```

Here, three lists are merged using function `AppendFncDclLst`. Notice also the definition of variable `initialFuncDecls`. It contains the initial set of declared functions. Moreover, it is local to the actual `progBody` and hence visible through remote reference for type checking purposes (see section 4.3.4 below, page 30).

4.3.1.2 Code

The synthesised attribute `code` has two specific goals: to carry out transformations on statements, and to create a transformed representation of the current program tree structure.

Notice that a new program representation may correspond to another valid representation within the same grammar or an entirely different one. While our example is of the former case, because we are interested in source-to-source transformations within the same language, the latter case is typically found on attribute-based compilers where the goal is the translation; *i.e.*, the mapping of expressions between two different languages. Recall here our interest in using attribute grammars, but not exclusively, in both horizontal and vertical transformations.

As seen through out our discussion, there are certainly strong dependencies between attributes in AGs so that computations (and transformations in our case) can be carried out. The transformation of statements into code is another one of these cases. The following code computations illustrate this dependence:

```
stmt   :    …
       |    While        { $$.code = Loop(StToBlck(
                                         IfThenElse(exp.value,
                                                    stmt$2.code,
                                                    Break))); }

       |    Loop         { $$.code = StLstToSt(
                                      Label(block.initialLabel) ::
                                      BlckToSt(block.code) ::
                                      Goto(block.initialLabel) ::
                                      Label(block.finalLabel) ::
                                      StmtListNil); }

       |    Return       { $$.code = Return(exp.value); }

       |    Break        { $$.code = ({block.brkCntAllow} == "YES") ?
                                     Goto({block.finalLabel}) :
                                     Break; }


       |    Continue     { $$.code = ({block.brkCntAllow} == "YES") ?
                                         Goto({block.initialLabel}):
                                         Continue; }
       |    …
       ;
```

Recall our discussion on transformations presented in section 4.2.2, page 23; these computations are an example for their straightforward specification.

In our case, every production rule has the synthesised attribute `code`, whose type is identical to the type of the production it is attached to. Attributes within curly brackets denote upward remote reference; *e.g.*, `{block.initialLabel}` stands for a reference to the next enclosing attribute `initialLabel` of rule `block` that exists up the tree. Upward remote attribute reference is a very helpful mechanism for our purposes. It gives to possibility to refer to certain attributes of other productions that necessarily occur above any instance of the current production *p* in a term, without the explicit declaration of copy rules. By "above", we mean "between any instance of *p* and the root of the term." This is

the only remote mechanism found in SG, while Eli offers other useful mechanisms as well (see section 5.4). In functional programming this issue is trivial [12].

`StToBlck` and `StLstToSt` are user-defined functions; they carry out the conversion of a statement into a block and a statement list into a single statement respectively. List computations are also syntactically simplified when using "`::`" for concatenation.

The second goal of attribute `code` is to compute the complete transformed program representation. This is the reason why the synthesised attribute `code` is associated with every production rule of the grammar, including terminal and non-terminal symbols. In every case, `code` will refer to the actual computed subtree for that production it is attached to; hence the complete tree will result from the root production (`program.code`).

```
program      :  Prog      { $$.code = Prog(progName.code,
                                          progBody.code);};
progBody     :  ProgBody { $$.code =
                      ProgBody(
                         declList.code,
                         AppendFncDclLst(
                            funcDeclList.code,
                            AppendFncDclLst(block.newFncDcls,
                                            funcDeclList.newFncDcls)),
                         block.code); }

funcDeclList:  FuncListNil  { $$.code = FuncListNil; }
            |  FuncListPair { $$.code = (funcDecl.code ::
                                          funcDeclList$2.code); };

funcDecl     :  FuncDecl     { $$.code = FuncDecl(funcName.code,
                                                  type.code,
                                                  block.code); };
```

The `code` attribute computation is reminiscent of a recursive descent visit to the tree, but remember that these specifications do not establish evaluation order.

Recall the structure of the production rules `program`, `funcDeclList` and `funcDecl` from the abstract syntax grammar of ***P*** (Figure 3, page 14). In the case of `program`, its actual `code` attribute results from the construction of a new program node via the function `Prog`**,** where its actual parameters are the synthesised code subtrees of its components: `progName.code` and `progBody.code`. Notice that `funcDeclList` and `funcDecl` follow the same building pattern, where in the former production rule the recursive definition is used. So, the **`funcListPair`** function constructs a `funcDeclList` using the `funcDecl.code` attribute and the `funcDeclList.code` attribute which represents the rest of the list (the postfix "`$2`" in `funcDeclList$2.code` indicates that it is referring to the second `funcDeclList` occurrence in that specific production rule).

4.3.2 The Block structure and its attributes

As already mentioned when presenting our working grammar, the primary goal of the `block` production rule is to hold contextual information for different purposes. In our HAG, this is achieved by its four attributes, namely `initialLabel`, `finalLabel`,

`returnAllow` and `brkCntAllow`. Moreover, they will be relevant depending on the context `block` used within the grammar.

Notice that for all these cases the immediate outer context is the one establishing the information the inner context is constrained by. This is always the case as we have explained where referring to copy rules.

The first two attributes, `initialLabel` and `finalLabel`, correspond to the unique labelling information needed when transforming `while` structures into a combination of `condition` and `goto` statements. In addition, they are also used when transforming `break` and `continue` statements while computing the `code` attribute of statements. The following specification shows that `finalLabel` and `initialLabel` contain relevant information within the context of a `loop` statement. This statement defines the label's context.

```
progBody :  ProgBody     { block.initialLabel = LabelNameNull;} ;
funcDecl :  FuncDecl     { block.initialLabel = LabelNameNull;} ;
stmt     :  Loop         { block.initialLabel =
                             LabelName(gensym("LpInitLbl_",&($$)));};

progBody :  ProgBody     { block.finalLabel = LabelNameNull; } ;
funcDecl :  FuncDecl     { block.finalLabel = LabelNameNull; } ;
stmt     :  Loop         { block.finalLabel =
                             LabelName(gensym("LpFinalLbl_",&($$)));};
```

Blocks associated directly with the program body and function declarations have no relevant information in these attributes, but a `loop` statement does. When a label name is required, then one on generated.

The last two attributes, `returnAllow` and `brkCntAllow`, are used to restrict the occurrence of `return`, `break` and `continue` statements.

```
progBody :  ProgBody     { block.returnAllow = "NO"; } ;
funcDecl :  FuncDecl     { block.returnAllow = "YES"; } ;
stmt     :  Loop         { block.returnAllow = {block.returnAllow}; };

progBody :  ProgBody     { block.brkCntAllow = "NO"; } ;
funcDecl :  FuncDecl     { block.brkCntAllow = "NO"; } ;
stmt   :    Loop         { block.brkCntAllow = "YES"; } ;
```

Notice that, in the case of a `loop` statement, the value to assign to its block's `returnAllow` attribute is obtained from the current context. For this, remote reference is used.

In both cases, `block` information is used to restrict the occurrence of specific statements within their scope. For instance a `return` statement within the main program body has no semantic meaning in ***P***, presenting a semantic error; while the same situation will happen when `continue` or `break` statement exists out to a `loop` body.

4.3.3 The Value Attribute

The attribute `value` serves the same purpose of attribute `code` except for the fact that it only deals with expressions. This separation is intentionally made for the didactic purposes of our case study, so the `value` of `after` expressions is clearly distinguished from its side-effect.

```
exp    :      …
       |      True      { exp.value    = True; }
       |      False     { exp.value    = False; }
       |      Id        { exp.value    = Id(variable); }
       |      Equal     { exp$1.value = Equal(exp$2.value,exp$3.value);}
       |      …
       |      After     { exp$1.value = FuncCall(newFuncName); }
       |      FuncCall  { exp$1.value = FuncCall(funcName); } ;
```

When computing the `value` of an `after` expression a function call (`FuncCall`) is issued. This function corresponds to the one defined when transforming the current `after` expression. For the rest of the expressions, the `value` attribute corresponds to copy of the same expression.

4.3.4 The Type Attribute

The attribute `type` establishes a basic type checking mechanism in ***P***. Essentially, it enforces every expression, variable and function name to have a type.

As the following specification shows, some expressions have a fixed type associated, while others require that it be computed. Furthermore, two *global* lists specified in the `progBody` production rule are used as type lookup tables: `env` for variable types and `initialFuncDecls` function types. Two user-defined functions are used for table lookup activities: `LookupFuncType` and `LookupDeclType`.

On the one hand, `env` contains all variable declarations in a program. It is defined at the `progBody` production rule, and its is typically used via remote reference. On the other hand, `initialFuncDecls` contains all function declarations originally present in a program.

```
exp    :      EmptyExp     { exp.type = EmptyType; }
       |      IntConst     { exp.type = IntType; }
       |      True, False  { exp.type = BoolType; }
       |      Id           { exp.type = variable.type; }
       |      Equal,
              NotEqual     { exp$1.type = BoolType; }
       |      Add, Minus   { exp$1.type = IntType; }
       |      After        { exp$1.type = exp$2.type; }
       |      FuncCall     { exp$1.type =
                               funcName.type == EmptyType ?
                                  LookupFuncType(funcName,
                                          {ProgBody.initialFuncDecls}) :
                                  funcName.type; } ;
variable:     VariableNull{ variable.type = EmptyType; }
       |      Variable     { variable.type =
                             LookupDeclType(variable,{ProgBody.env});};
```

```
funcName:    FuncNameNull{ funcName.type = EmptyType; }
       |     FuncName    { funcName.type =
                             LookupFuncType(
                               funcName,
                               {ProgBody.initialFuncDecls}); }
```

4.4 Example

Let's see our transformations at work. For this, we make use of a simple algorithm that sums the first fifty square integers:

```
program sumSquares;
var
  a : integer;
  b : integer;
  n : integer;
  temp : integer;
  misc_counter : integer;

function SquareOfA(): integer;
begin
  misc_counter := (0 - 1);
  temp := 0;
  while (After (misc_counter := (misc_counter + 1))
               (misc_counter <> a)) do
    temp := (temp + a);
  Return temp
end;

begin
  a := (After b := 0 => (After n := 50 => 0));
  while (After a := (a + 1) => (b <> 0)) do
    b := (b + SquareOfA)
end.
```

On the first transformation step, the `after` expression and the `while` statement are transformed. The SG allows two view individual transformation steps: the abstract syntax tree, which is used as the starting point for the attribution, is computed as a synthesised attribute of the parse tree. Hence, by transforming the attributed tree obtained from this first transformation step the resulting tree second transformation step is computed.

First Transformation Step

```
Program sumSquares;
Var
  a : integer;
  b : integer;
  n : integer;
  temp : integer;
  misc_counter : integer;

function SquareOfA(): integer;
begin
  misc_counter := (0 - 1);
  temp := 0;
  loop
    if AfterFunc_39AA0 then
      temp := (temp + a)
    else
      Break;
  Return temp
end;

function AfterFunc_3AO60(): integer;
begin
  n := 50;
  Return 0
end;

function AfterFunc_39BU0(): integer;
begin
  b := 0;
  Return AfterFunc_3AO60
end;

function AfterFunc_3AOI0(): boolean;
begin
  a := (a + 1);
  Return (b <> 0)
end;

function AfterFunc_39AA0(): boolean;
begin
  misc_counter := (misc_counter + 1);
  Return (misc_counter <> a)
end;

begin
  a := AfterFunc_39BU0;
  loop
    if AfterFunc_3AOI0 then
      b := (b + SquareOfA)
    else
      Break
end.
```

Second Transformation Step

```
program sumSquares;
var
  a : integer;
  b : integer;
  n : integer;
  temp : integer;
  misc_counter : integer;

function SquareOfA(): integer;
begin
  misc_counter := (0 - 1);
  temp := 0;
  begin
    LpInitLbl_3CA7G :
    if AfterFunc_39AA0 then
      temp := (temp + a)
    else
      Goto LpFinalLbl_3CA7G;
    Goto LpInitLbl_3CA7G;
    LpFinalLbl_3CA7G :
  end;
  Return temp
end;

function AfterFunc_3AO60(): integer;
begin
  n := 50;
  Return 0
end;

function AfterFunc_39BU0(): integer;
begin
  b := 0;
  Return AfterFunc_3AO60
end;

function AfterFunc_3AOI0(): boolean;
begin
  a := (a + 1);
  Return (b <> 0)
end;

function AfterFunc_39AA0(): boolean;
begin
  misc_counter := (misc_counter + 1);
  Return (misc_counter <> a)
end;

begin
  a := AfterFunc_39BU0;
  LpInitLbl_3CMDG :
  if AfterFunc_3AOI0 then
    b := (b + SquareOfA)
  else
    Goto LpFinalLbl_3CMDG;
  Goto LpInitLbl_3CMDG;
  LpFinalLbl_3CMDG :
end.
```

4.5 Final Remarks

Modularity and decomposition model: We have shown the way SG supports modular specifications. This is also allowed in Eli, as it will be discussed in the next section. Moreover, the decomposition model exercised in SG and Eli is very appealing to the strategies IP and aspect-oriented programming (AOP) are aimed to [13, 41]. This is, a solution rather than staying well localised within a production rule tends to "cross-cut" the grammar structure and hence their computations.

This modular style is coherent with the aspect-oriented specification of compilers as proposed by [12, 13]; *i.e.*, it is feasible to specify and treat modularly the different programming language aspects of a compiler specification. But there is an important difference. De Moor's approach has a higher-abstraction level because aspects are described as functions that code attribute computations. Each function contributes partially to the definition of production rules. Then, a production rule is completely defined by the functional composition of the set of functions that describe it. This contrasts with the SG and Eli approach, where the complete specification of a production rule corresponds directly to the syntactic grouping of its attribute computations; here no activity at the semantic level occur whatsoever. These attribute computations may be found through out the AG specification.

Specification size: Copy rules make standard attribute grammar specifications to grow with a lot of simple production rules, reducing its readability and maintainability. We could have also produced a smaller specification if alternative mechanisms were used for attribute passing, for instance dictionaries. As known, dictionaries are typically used to speed up referencing in compilers, but it demands external elements from attribute grammars. We have avoided making use of alternative mechanisms, because of our interest in recognising the way a small attribute grammar (kernel) is modified when including new extensions.

A minimal set of user-defined functions was required for this case study (see end of section 11.1, page 71). They basically abstract manipulation and creation of program constructs.

This drawback is solidly addressed in Eli. Eli remote reference constructs eliminate the need to specify copy rules, reducing the actual specification size and more importantly introducing a level of attribute independence from the grammar structure.

Production rule sub-tasks and intentions: The implementation of a new programming construct may involve carrying out several sub-tasks. This is clearly shown when transforming `after` expressions into function definitions and function calls in this case study. For instance, the following general sub-tasks are required for this transformation:

1. Program Body: to collect new function declarations from any block.
2. Program Body: to add new function declarations to the existing declaration part if already there, otherwise create one.
3. Expression: to create a function whose structure is based on the expression's side effect.
4. Expression: to rewrite itself as a function call whose name is locally available.

These sub-tasks depend on the transformation strategy selected and its semantic interpretation; *i.e.*, a completely different set of sub-task could be necessary if a distinct interpretation were chosen. In our case, each of these sub-tasks is implemented in an aspect-oriented style, but we still can not confirm this is true for every case; what we can confirm is that the more independent sub-tasks are, the more applicable this style is.

Interestingly, we do not find this sub-task decomposition estranger to IP. Indeed, this example presents an approach we may think of when specifying an intention in IP and its decomposition. Each of these sub-tasks can represent simpler intentions, which we might call *sub-intentions*, and that produce the expected global effect when combined properly.

It is our impression these sub-tasks seem to have more clearly defined their interactions and contributions than the intention, and perhaps intentions, they contribute to. But this still is an open question for us. Intentions' nature undoubtedly tend to be this way, and, even more, they need to be able to interact correctly with other intentions that presumably do not exist yet.

Our sub-task decomposition has instinctively been presented in the exact evaluation order we need them to occur, although attribute grammar evaluators can deduce such interdependence from the existing attribute computations and establish an evaluation order.

Impact to changes when introducing a new construct: Just including a new programming construct can produce a reasonable impact in the grammar computations as shown in our discussion. The inclusion of `after` expression, characterised by its side-effect, has introduced new attributes and attribute computations in most production rules.

But more importantly, the new construct has led us into the analysis of possible semantic interpretations the new construct might have when combined with other existing constructs. Their relation can be very subtle and perhaps indirect. Furthermore, their relation may be evident only when a third construct is present, making this analysis remarkably complex.

We certainly want to move away from the idea that introducing a new construct, in the form of an intention, requires thinking about every conceivable interaction with any other existing element in the programming language. What needs to be done then when new constructs are included? So far, we see new constructs can not be completely independent entities.

This fact may represent a problem for the IP enterprise, as we have shown that a harmless construct, as the `after` expression is, requires carrying out careful analysis on its interaction and transformation strategy when composed with other existing constructs in the language.

5 A review of the Eli system

The Eli system [22, 32, 38, 39] is the result of a joint project held between the Compiler and Programming Laguange Group at University of Paderborn, Germany, and the Compiler Tools Group at University of Colorado, USA. It is a pioneering effort that shows that a combination of notational concepts can be use to create reusable attribution modules. A set of solutions for standard language implementation programs can be provided as a library of such modules. That library would simplify both definition and implementation of programming languages because an attribution module is a formal description of a relation from which a program fragment establishing that relation can be generated. Eli's emphasis is on reusability of specification modules rather than on modular decomposition of the generated implementation. Moreover, the Eli philosophy is to extend grammar specifications by providing inheritance of computational concepts as understood in object oriented programming.

Eli goal is clear and straightforward: to reduce the cost of producing compilers of language processors (standard or special-purpose programming languages). The system provides solutions common to all compilers, allowing the user to focus his attention on the requirements and design decisions that are unique to the language in hand.

In this section we present the particular features that distinguish Eli from other AG systems.

5.1 Compiler specification

Eli accepts descriptions of those requirements and design decisions, and combines them with its understanding of compiler construction problems to produce the corresponding compiler. Requirements and design decisions can be thought of as specifications of instances of subproblems or the language translation problem.

There are basic ways in which the user might specify an instance of a subproblem: by analogy ("this problem is the same as problem X"), by description ("this is a problem of class Y, and is characterised as follows …") and by solution ("here is a program to solve this problem"). The user must have sufficient understanding of the subproblem to recognise the most appropriate specification to use.

To support specification by analogy, Eli provides a library of solutions to common compiler subproblems. For example, consider a Pascal-alike language where a name is defined in a single procedure. If the same name is defined in a nested procedure, the outer definition is "hidden" within the inner procedure. This behaviour is common to any programming languages, and a solution for the problem of associating the definition of a name with each use of that name can be solved by analogy by instantiating a module from Eli's library:

```
$/Name/AlgScope.gnrc :inst
```

Similarly, other common problems like error reporting, string concatenation, symbol occurrence counting, optional identifiers generating, and others. Most modules solve a rather small task.

To support specification by description, Eli provides a set of notations for characterising common compiler subproblems. The input language grammar may be specified in such a notation; it is used to specify the phrase structure and thus characterise the syntax analysis

subproblem of that language. Another example is the notation used for the specification of tree computations and remote dependencies.

To support specification by solution, Eli accepts arbitrary C code that solves a unique compiler subproblem, and incorporates it into the generated translator. The user must take full control over the C interface and implementation (header and implementation files).

Specification by analogy and description are two different forms of reuse: a specification by analogy reuses a particular solution, while a specification by description reuses a problem-solving method.

5.2 Symbols and Rules

Recall the formal description of an attribute grammar presented in section 3.2, page 10. In Eli, **`SYMBOL`** constructs are used to specify the types of attributes in $A(X)$. Each **`RULE`** construct corresponds to one production of the reduced context-free grammar. The elements of $R(p)$ and $B(p)$ are given between the keywords `COMPUTE` and `END`. Again, the order in which the computations are written down is totally irrelevant. Attribute computations describe relationships among attributes, *not* an algorithm for computing the values of those values.

5.3 Dual use of attributes

Based on the generalisation that results of computations in $B(p)$ are not restricted only boolean values, on this calculus each attribute effectively represents a postcondition reached after completion of the attribute computation of R defining that attribute. Attributes used in a computation of R or B effectively represent preconditions for beginning that computation. It the dependence relationships satisfy certain constraints, it is possible to mechanically derive a complete computation algorithm from them [66].

With this approach, there are two views of attributes:

1. The approach found in most treatments of attribute grammars: use of attributes solely for the propagation of values. They demand that functions appearing in computations on R and B have no side effects, because such side effects cannot be reflected in the attribute values.
2. By considering attributes to represent pre and postconditions, however, side effects in component functions are easily accounted for.

The use of pre and postcondition terminology has nothing to do with its usual notion as presented in [23]

The following example, taken from [35], specifies how to print expressions in postfix notation, e. g. `1 2 3 * +` for the given expression `1 + 2 * 3`. It demonstrates how computations that yield an effect rather than a value are specified to depend on each other.

```
RULE: Root ::= Expr COMPUTE
      Expr.print = "yes";
      printf ("\n") DEPENDS_ON Expr.printed;
END;
```

```
RULE: Expr ::= Number COMPUTE
      Expr.printed = printf ("%d ", Number) DEPENDS_ON Expr.print;
END;
RULE: Opr ::= '+' COMPUTE
      Opr.printed = printf ("+ ") DEPENDS_ON Opr.print;
END;
RULE: Opr ::= '*' COMPUTE
      Opr.printed = printf ("* ") DEPENDS_ON Opr.print;
END;
RULE: Expr ::= Expr Opr Expr COMPUTE
      Expr[2].print = Expr[1].print;
      Expr[3].print = Expr[2].printed;
      Opr.print = Expr[3].printed;
      Expr[1].printed = Opr.printed;
END;
```

5.4 Remote dependence

The principle of locality commonly found in attribute grammars is extended: all of the preconditions and the postcondition for a computation must appear within the context of a single node and its children. But sometimes one of the preconditions for a computation may be established by another computation that is far away in the tree. Eli authors visualise this extension as a calculus. The calculus requires such a precondition to be established locally by computations provided in the intermediate contexts. This is the source of huge number of "copy rules" in attribute grammars formulated using only the notation of the underlying calculus.)

In most cases, remote dependence follows one of the tree simple patterns:

1. A computation at the root of a subtree containing the local context establishes the precondition.
2. The precondition is the union of the postconditions for some set of computations at nodes that are descendants of the subtree rooted in the local context.
3. The dependence involves an invariant for some iterative computation visiting nodes in (depth-first) left-to-right order.

Each of the tree patterns are directly formulated by an attribute grammar specification construct, which later can be transformed into the notation of the underlying calculus by generating all the intermediate computations mechanically.

Direct formulation of remote dependence reduces the size of an attribute grammar, but a more important characteristic is that it abstracts them from the intermediate tree structure. It is *invariant* with respect to modifications of that structure.

The SG only offers a remote dependence mechanism similar to first computing facility presented bellow. Please refer to page 20 for a discussion on SG's.

<u>Precondition at the root of a subtree</u>

As an example of a computation whose precondition is a *computation at the root of a subtree containing the local context*, consider the lexical nesting depth of a `block`. The precondition for computation of the nesting depth is that a value be available for the

nesting depth of the immediately enclosing `block`. If we assume that a `block` is one form of a `statement`, then this relationship might be expressed by the following attribute grammar rules:

```
RULE Program: Root ::= Block COMPUTE
      Block.Depth = 0;
END;

RULE Inner: Statement ::= Block COMPUTE
      Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

If the single attribute name in this example is replaced by a parenthesised list, the precondition it the first element of that list encountered when walking up the tree from the current node.

<u>Precondition is the union of the postconditions</u>

As an example of a precondition that is the *union of the postconditions for some set of computations at descendant nodes* is illustrated by the problem of determining the cost of each statement in a program. Suppose that each operator has an associated cost, and that the cost of a statement is the sum of the costs of its component operators.

```
RULE Calculation: Statement ::= Expression COMPUTE
      Statement.Cost = CONSTITUENTS Operator.Cost
            WITH (int, ADD, IDENTICAL, ZERO);
END;
```

In this example, `CONSTITUENTS` must yield a value: the sum of the operator costs. The `WITH` clause specifies the type of the "union" value (int) and names three user-defined functions used in its computation. The first (`ADD`) combines two "union" values to yield a "union" value, the second (`IDENTICAL`) creates a "union" value from an `Operator.Cost` value, and the third (`ZERO`) creates a "union" value from nothing.

<u>Left-to-right iteration</u>

An *invariant for a left-to-right iteration* is illustrated by the computation of enumerated constant values in Pascal. Pascal enumeration constants are defined by a list of identifiers, and the value represented by each identifier in the list is just the number of identifiers preceding it. Thus the first identifier represents 0 because there are no identifiers to its left, the second represents 1, and so on.

```
CHAIN Count: int;
RULE Enumeration: Type ::= '(' EnumConstList ')' COMPUTE
      CHAINSTART EnumConsList.Count = 0;
END;
RULE ConsListElt: EnumConst ::= Identifier COMPUTE
      EnumConst = ADD(EnumConst.Count, 1);
END;
```

Here, variable `Count` is a special kind of attribute called a "chained attribute" and describes an invariant. The invariant is established initially by a `CHAINSTART` directive at the root of some subtree. Computations updating the invariant may be associated with any node in that subtree.

The following example illustrates this new use of attributes. Here attribute Objects asserts that *all* visible names have their properties defined.

```
CHAIN Objects: VOID;

SYMBOL StandardBlock COMPUTE
  CHAINSTART HEAD.Objects=StandardIdProperties()
      DEPENDS_ON THIS.Env;
END;
```

The meta-variable `THIS` is used to make reference to the actual node where the computation is performed.

5.5 Symbol computations

A symbol can have computations associated with it rather than to each of the different contexts in which the symbol appears. This is possible if the attribute of a symbol has the following properties:

1. It only depends on attributes of a single symbol or on remote attributes.
2. It should be applied at (almost) every instance of that symbol in the tree. (In this second property, overridden rule may vary which computations visible on a symbol)

An example of the kind of factorisation possible through Symbols is the `Depth` computation:

```
RULE Program: Root ::= Block COMPUTE
      Block.Depth = 0;
END;

RULE Inner: Statement ::= Block COMPUTE
      Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;

RULE Proc: Body ::= Block COMPUTE
      Block.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

It would then pay to associate the nesting depth increment with the symbol `Block` instead of each context in which a block appears. In this way, the computation presented in rules `Inner` and `Proc` are factorised in a symbol computation:

```
SYMBOL Block COMPUTE
      INH.Depth = ADD(INCLUDING Block.Depth, 1);
END;
```

Here the keyword `INH` indicates that `Depth` is an inherited attribute and that its computation has to be associated with each context having `Block` on the right-hand side. In the case of the computation of a synthesised attribute, for instance `Block.s, it` would be denoted `SYNT.s` and would be associated with each context having `Block` on the left-hand side. A plain computation is associated with each context having its symbol on the left-hand side. There can be several plain computations and attribute computations of either class in a single symbol attribution.

A more complex example follows, where a `TypeNameUse` symbol verifies it is correctly defined:

```
SYMBOL TypeNameUse: Type: DefTableKey;

SYMBOL TypeNameUse COMPUTE
  SYNT.Type=
    IF(EQ(GetKind(THIS.Key,Undefined),Typex),THIS.Key,NoKey)
      DEPENDS_ON THIS.VisibleTypeProperties;
  IF(NE(GetKind(THIS.Key,Typex),Typex),
    message(ERROR,"Must be a type name",0,COORDREF))
      DEPENDS_ON THIS.VisibleTypeProperties;
END;
```

Another example is the general type checking on expressions:

```
SYMBOL Expression COMPUTE
  IF(AND(
    AND(NE(THIS.Type,THIS.ExpectedType),NE(THIS.Type,NoKey)),
    NE(NoKey,THIS.ExpectedType)),
    message(
      ERROR,
      "Type yielded is not compatible with the context",
      0,
      COORDREF));
END;
```

5.6 Inheritance

Several syntactic constructs of a language often share some set of semantic properties. Inheritance allows specifying computations independently from the symbols used in a particular language definition. It abstracts the semantic concept.

Consider the block nesting depth example again. To simplify the description of the semantic property, we might introduce an additional symbol, Contour, representing it:

```
SYMBOL Contour COMPUTE
      INH.Depth = ADD(INCLUDING Contour.Depth, 1);
END;
```

Note that this computation is completely independent of the symbols used in a particular language definition to denote constructs associated with the distinct activation records.

An abstract computation can be inherited by some set of symbols representing nodes in the tree:

```
SYMBOL Block INHERITS Contour END;
SYMBOL Procedure INHERITS Contour END;
```

Each symbol may also inherit computations from several other symbols, possibly through several levels of inheritance.

5.7 Modules

An attribution module embodies a set of related computations defined on a tree. In order to be useful, these computations must be mapped onto appropriate contexts in a specific tree that describes the abstract syntax of a program in the desired programming language. The module is reusable if this mapping is possible for a variety of programming languages.

Any technique for constructing reusable attribution modules must therefore be able to abstract the essential structural relationships among the computations, separating them from structural details that are relevant for those computations. Moreover, it must be able to associate the computations with symbols or productions that represent appropriate programming language constructs.

5.8 Cumulative attribution

Cumulative attribution is a simple and effective notational technique for decomposition of large attribute grammar specifications: the user is allowed to write an arbitrary number of rules with the same production. Since each context in a tree is uniquely defined by a single production in a reduced context-free grammar, the total set of computations for that context is the union of the sets of computations described by each rule with that production.

This makes it possible to decompose a specification into components, each covering one aspect of the total problem. This is very appealing to the aspect oriented-programming approach [41].

Moreover, physical decomposition of an attribute grammar can be easily achieved by storing each component in a distinct file.

5.9 Final Remarks

AG-based new paradigm? Eli authors claim Eli to be an example of a new paradigm. They based this assertion on the features we already discussed: the use of attributes as pre- and postconditions, patterns of remote dependence, symbol computations, inheritance and cumulative attributions [39].

We believe this is overstated, but we certainly agree that Eli brings a very valuable and pioneering contribution the future use of AG as a specification mechanism.

Reuse: From our point of view, the Eli's main characteristic is the capability to define *attribution modules* that can be reused in a variety of applications. Eli's modularity typically abstracts small aspects of a task, which are possible by combining the ideas of abstract remote attribute access and inheritance. This feature makes a strong difference when comparing it against other AG systems [19].

Specification Size: Eli remote reference constructs eliminate the need of specifying copy rules. This reduces the actual specification size and more importantly introducing a level of attribute independence from the grammar structure if compared with SG for instance.

The size contraction results out the specification of anonymous relationships declared through the Eli meta-language constructs `INCLUDING`, `CONSTITUENTS` and `CHAINSTART` among others.

Symbols and Inheritance: `SYMBOL` computations are also an interesting feature in Eli, because they can compute transformed versions of its associated symbol. This is similar to the IP view that transformations may be regarded as processes that are automatically associated with programming language constructs, and their use can annotate a syntax tree with its appropriate transformations.

Appendix 11.2, page 74, shows how scope analysis is established in a Pascal compiler using Eli [65]. In the general sense, it consists of the instantiation of generic modules and the creation of the appropriate set of symbols, which inherits behaviours from these modules. We have taken this Pascal implementation and bring it a step further by adding the `after` expression, `while` statement and `loop` statement transformations to it.

Incremental evaluation: Eli has been conceived as a set of tools for compiler construction. Therefore, it does not offer an incremental attribute update mechanism as found in SG. While editing in SG, users can modify the current tree structure and attributes are incrementally updated accordingly.

Nevertheless, we have also experienced difficulties using Eli. When dealing with tree manipulation and construction, it becomes extremely laborious because of the use of C code and the occurrence of obscure error messages. We could use macros to avoid the former, but it would not improve on the latter. Also, we believe that the collection of tools and little specific-oriented languages provided by the system reduces substantially its regularity and readability, allowing space for confusion.

6 Review of other systems

6.1 AML

AML (*Attribution in ML*) is an attribute grammar toolkit for building compilers and user interfaces [18]. It is a spiritual heir to the SG [55], particularly concerned about efficient incremental evaluation techniques, and the Pegasus project at AT&T Bell Laboratories [54], which emphasises on providing a high-level foundation for interactive systems. AML is built on the evaluation technology of the SG, while using a higher-level foundation for the implementation.

AML has an interesting design approach: the specification language is an extension in the implementation language (SML in this case) for writing attribute rules. In addition, SML is also used for specifying auxiliary types and functions. Starting from an AML specification, the system generates a collection of SML modules that implement the specification and support code. These are then combined with additional grammar-independent modules to construct a complete system.

AML design guidelines are gathered into six points [18]:

- Easy interaction between ML and AML. It should be straightforward to feed attributed trees into AML code, and likewise to attribute the results of AML computations. This will provide a simple connection between the attribute evaluator and any other component of the system.
- Minimal extension of ML. Unnecessary new keywords should be avoided without sacrificing the expressive power. The size of the extension will depend on the level of abstraction of the specification language constructs.
- Declarative specifications through higher level constructs. They will improve the specification quality without loss of generality. The declarative character, and the formal properties of AGs should be investigated. The expressive power can reduce the size of AGs drastically by elimination of redundancy. These constructs can be systematically transformed or expanded into the basic AG concepts.
- AML should remain *independent from the evaluation method* selected for its implementation. This would support experimentation using the different classes of attribute grammars and potential extensions.
- Attribute grammars can be large, so it is important to support modular specifications and to supply a mechanism to reduce redundancy. This will require further analysis about scope rules.
- Strongly typed language. AML will take advantage of ML's polymorphic type checking system as a necessary condition for programming safety. Type and function definitions will be based on the ML formalisms, including user-defined ones.

Besides, AML supports local attributes and syntactic references to them, but does not offer remote reference mechanisms. An AML specification consists of a collection of related declarations; in many ways, it is similar to an SML structure definition. It has the form

```
Grammar name = struct declarations end
```

Where `name` is the name of the grammar being specified. The rest of an AML specification is based on six kinds of declarations:

- `Termtype` declarations defining terminal symbols.
- `Prodtype` declarations defining non-terminals and their productions.
- `Root` declarations defining root non-terminals.
- `Attribute` declarations defining attributes.
- `Attribution` declarations defining semantic rules.
- SML top-level declarations that are used to define auxiliary types and functions.

The implementation of the AML compiler is done in a modular fashion, allowing easy experimentation with different classes of grammars and different evaluation techniques.

In our experience of declaring medium size attribute grammars using exclusively these specification elements in AML, the user will rapidly recognise the need of higher abstraction mechanisms and reuse that could simplify the activity. Nevertheless, we consider AML to describe what the base line of a kernel for an AG specification language should be. It would be interesting to add new features similar to Eli's to AML.

6.2 Elegant

Elegant is a compiler generator system developed at Philips Research Laboratories Eindhoven [5, 6, 7, 28] from 1987 to 1996. It has been used within Philips for the construction of several dozens of compilers, including the system itself. Elegant stands for **E**xploiting **L**azy **E**valuation for the **G**rammar **A**ttributes of **N**on-**T**erminals.

As its name suggests, the system offers the ability to specify a compiler by means of attribute grammars. It allows attribute grammars to be *pseudo circular*; *i.e.*, attribute functions exploit their non-strictness by allowing their computation from seemingly circular definitions as in [29]. The result is a powerful, yet very efficient, programming language which allows programming and compiler construction at a high level of abstraction.

In his thesis [6], Augusteijn claims that, even though attribute grammars have a long story, very few compiler generators have been constructed that exploit their full power. Most

Elegant shows that lazy evaluation allows compiler writers to ignore a separation into passes, and to focus on the logical structure of their compiler instead. This is also a feature of attribute grammars, which can be viewed as a particular style of lazy functional program. But we believe a separation into passes may be a logical structure that makes the compiler easier to understand. Compilers written in attribute grammar style are typically structured by production - it is hard to structure them by semantic aspect, such as "environment" and "lexical level", and it is certainly not possible to view these aspects as separate units of compilation.

During the years Elegant has developed into a powerful compiler generator system and a complete programming language which smoothly combines features from functional languages (polymorphism, higher orderness, laziness, comprehension) with imperative features (state, destructive update).

But there is an objection against attribute grammars as functional programs: the resulting programs are highly convoluted, and even less modular than standard attribute grammars [12]. Elegant certainly suffers from this problem. Essentially, all attribute definitions have to be grouped by production. It is thus not possible to group all definitions for a single attribute in one place, and then specify how each rule contributes to the behaviour of a production. One cannot use the same set of attribute rules, and make them contribute to different productions.

7 Data Refinement and Program Transformations

The idea of making software development an engineering discipline is one of the strongest motivations found in every new software engineering methodology. The development process of these methodologies shows the way ad hoc techniques are replaced by proposed uniform and more coherent approaches that can ensure software quality and correctness.

Following a typical top-down development, where programs are created to satisfy the user's intended requirements and their specifications, a remarkable research effort is taking place in formalising, controlling and verifying the program construction process. Thus, most frameworks for the development of correct software are based on some notion of *refinement* [27, 48, 68]; *i.e.*, a development step with some formal justification of its correctness. There are various ways of formal justification: correctness may be established by verification of explicit proof obligations that arise from the semantics of the development step; in fact, the derived unit may be conceived independently and the relation of refinement added as a separate step. This approach is sometimes referred to as the *a-posteriori* verification or the "invent-and-verify" approach. Another way of formal justification is to have a notion of pre-conceived formal development step, for which correctness-preservation is intrinsic. The correctness of a schematic development step is proved *a-priori*, but in general, it is required that the applicability of the instantiated development step is verified as a precondition and justification for its application. In both approaches the developer has to have expertise in choosing the right development step and creativity to come up with solutions.

The idea of applicability preconditions of correct refinements is the following, where we focus on the data refinement, but without forgetting the importance of the algorithmic refinement. Consider two blocks

B_1: **begin var** $v_1 := e_1$; S_1 **end**

and

B_2: **begin var** $v_2 := e_2$; S_2 **end**

Block B_2 is a refinement of B_1 if replacement of B_1 in a correct program by B_2 leaves a program correct. Thus B_2 is a refinement of B_1 if

$wp(B_2,R) \Rightarrow wp(B_1,R)$ (for all predicates of R).

The variables in list v_1 are often called *abstract variables* while those in v_2 are *concrete variables*. This terminology comes from the uses made of such refinements. For example, v_1 could be a variable of type *set*(*integer*) and v_2 could be a variable of some concrete data type that implements sets of integers; the data refinement is being used to replace an abstract variable by its implementation.

Proving that B_2 is a refinement of B_1 is usually split into two tasks. First, requirements are placed on the initialising expressions e_1 and e_2. Secondly, requirements are placed on S_1 and S_2. If B_1 and B_2 satisfy the requirements, one says that B_2 *is a data refinement of* B_1, or that B_2 *data-refines* B_1 [27, 48].

A good representation element for these refining tasks is the *transformation*. So, the concrete target is *constructed* by successive transformations of its abstract representation

and the correctness proof obligations are verified at each transformation rule; this represents the development step of a program development by transformation methodology. The usefulness of the transformational approach boils down to the successful formalisation of software development knowledge in the form of correct transformation rules, application techniques and the adequate system support.

In the following sections, we report on two systems, Polya [26] and DiSTiL [60], each providing a different approach towards data refinement and data abstraction.

7.1 The Polya System

Polya presents a very pure approach to the use of transformations [17, 25, 26, 62]. Its main goal is to allow automatic data refinement through the use of co-ordinate transformations, which are described by *transforms* (the definition is given in the following section, page 48).

Polya's motivation is to use theoretical results effectively on complex data structures in the software industry. By empowering the transformation from algorithmic developments to implementation, state-of-the-art algorithms and data structures can easily turn into useful tools faster and requiring less debugging actions. Current program abstractions, like procedures, modules, and classes, are not sufficient to provide for a smooth implementation transition because they force the programmer to use constructs that are not at a suitable level of abstraction. Thus, Polya stresses that programmers should be able to write in the standard language of the application domain; *i.e.*, the best abstraction is the application domain language itself.

The following example illustrates the problem. Consider the use of an ordered set *S* that may be implemented as a heap, and the operation $S := S - \{max(S)\}$ which deletes the maximal element from *S*. An abstract interface for sets will provide an ad hoc set operation like *delete_max*, thus forcing programmers to write *S.delete_max*() instead of what is more natural and direct, $S := S - \{max(S)\}$. Furthermore, if for some reason a different implementation is needed for *S*, for instance one in which *delete_max* is not a primitive operation and therefore is not implemented directly, this statement must be rewritten in another form.

In order to remove abstraction problems, once the programmer writes the algorithmic abstraction under Polya, an automatic mechanism would carry out one or more transformation steps on this specification to produce a concrete program. In general, this concrete program is equivalent to the abstract representation, due to the fact that equivalence preservation is kept through all the transformation process.

Polya [62] offers the following abstraction mechanisms:

1. The programmer may define the notation of the domain, totally independent of any implementation, and write programs in that notation. This notation represents domain-specific abstract data types, which closely fit the manner of thought in the domain and allow easier reasoning and proof of correctness.
2. Transformation rules are used as refinement mechanisms between abstract and concrete representations of types and their operations. Therefore, the interface between abstract algorithms and concrete programs are defined by transformations, which represent patterns of usage and are aggregated in transforms. Each transform specifies the representation of expressions of a given type. For example, one

transformation rule could recognise the pattern $S := S - \{max(S)\}$ and transform it into code to remove the root from the heap.

3. The programmer may use *transform directives* to direct how each variable of a program should be implemented via transforms. Hence, different variables of the same type may be implemented differently according to directives and still preserve equivalence. For example, the variable *S* in a program could have a transform associated that represents sets as lists through the use a directive **implement** *S* **using** *Set_List*.
4. A calculus of transform directives allows a flexible composition of implementations; *i.e.*, transforms can be specialised and composed to realise implementations for complex data structures. For example, the implementation of a set of integers as a static array of size *k* of infinite-precision integers can be directed as

   implement *S* **using** (Set_HeapList(*max*); Set_StaticArray(*k*))[Int_InfPrec]

   Here, transform Set_HeapList(*max*) implements a set as a list that implements the heap, using function *max* to order the elements; transform Set_StaticArray(*k*) then implements a set as an static array of size *k*. Int_InfPrec implements the integer elements of *S* as infinite precision integers.
5. The program transformation approach is used to implement domain specific data structures as target data structures. These target data structures are provided by a given language, such as C++, ML or Java.

7.1.1 The Transforms

The transformational element used in Polya is the *transform*. A transform describes how to change syntactically a variable or expression of one type (the *source type*) within a program into a variable or expression of another type (the *representation type*). For example, a transform could describe the replacement of a variable *S* of source type *set*(*int*) by a variable of representation type *list*(*int*), where the set is being implemented as a list. Or, *S* could be replaced by a variable of type

record *A* : **array**(*int*),*size* : *int* **end**

where the set is being implemented in an array. Notice also that, depending on the transforms given and the directives, a set may be transformed into a list and this list could then be implemented as a fixed-size array. Thus, the refinement process towards a concrete representation or implementation may well take several transformation steps. However, there is no intrinsic need for the representation type to be "more concrete", and it could even remain at the same level of abstraction.

A transform can be a partial implementation of an abstract data type [24], a data refinement of a piece of code [48, 49], or a general transformation like the transformation of a dummy variable in a loop for efficiency purposes [42]. This can be understood from the transform's basic form:

transform *id* : *type* **into** *type*
coupling invariant : *coupling_invariant*
    transform_rule_list
end

A basic transform consists of an identifier *id* (the name of the transform), the source and target types of the transform, the coupling invariant and a list of transform rules.

The list of transform rules contains rules of the from:

$[LHS_pattern]_{tr_exp} = RHS_pattern$

Here, *tr_exp* is a transform expression denoting the representation defined by the rule and *LHS_pattern* is any expression or statement that pattern-matches *tr_exp*. Any *LHS_pattern* is a *RHS_pattern* where pattern variables resulting from *LHS_pattern* can occurred.

For each transform, a coupling invariant is a predicate that defines the relation between source expressions, x, and their representations, $[x]_T$, where x corresponds to a matched variable obtained from a *LHS_pattern* and $[x]_T$ denotes the result of the transformation of x under transform T. Even though the coupling invariant has no bearing on the transformation process, the author of a transform uses it to prove the correctness of each transform rule. For example, for a variable *v*:*complex* that is represented by

u: **record** *re, im*:*real* **end**

the coupling invariant is $v = u.re + i \cdot u.im$. Note that a representation of an expression is defined only when transform T transforms a single variable to another single variable.

An example of a basic transform follows:

transform BN : boolean **into** natural
coupling invariant : $x \equiv [x]_{\mathrm{BN}} > 0$

$$
\begin{aligned}
[true]_{\mathrm{BN}} &= 1 \\
[false]_{\mathrm{BN}} &= 0 \\
[x \vee y]_{\mathrm{BN}} &= [x]_{\mathrm{BN}} + [y]_{\mathrm{BN}} \\
[x \wedge y]_{\mathrm{BN}} &= [x]_{\mathrm{BN}} * [y]_{\mathrm{BN}} \\
[\neg x]_{\mathrm{BN}} &= 1 - sgn([x]_{\mathrm{BN}}) \\
[x]_{\mathrm{ID}} &= [x]_{\mathrm{BN}} > 0
\end{aligned}
$$

end

Transform BN allows representing boolean values as natural numbers. For example, consider the transformation of the boolean expression $(a \vee b) \wedge (c \vee d)$:

$$
\begin{aligned}
&(a \vee b) \wedge (c \vee d) && \equiv \\
&[(a \vee b) \wedge (c \vee d)]_{\mathrm{ID}} && \equiv \\
&[(a \vee b) \wedge (c \vee d)]_{\mathrm{BN}} > 0 && \equiv \\
&[(a \vee b)]_{\mathrm{BN}} * [(c \vee d)]_{\mathrm{BN}} > 0 && \equiv \\
&([a]_{\mathrm{BN}} + [b]_{\mathrm{BN}}) * ([c]_{\mathrm{BN}} + [d]_{\mathrm{BN}}) > 0 && \equiv
\end{aligned}
$$

Now, let us assume boolean variables a, b, c and d are *true*:

$$
\begin{aligned}
&([true]_{\mathrm{BN}} + [true]_{\mathrm{BN}}) * ([true]_{\mathrm{BN}} + [true]_{\mathrm{BN}}) > 0 && \equiv \\
&(1 + 1) * (1 + 1) > 0 && \equiv \\
&2 * 2 > 0 && \equiv \\
&4 > 0 &&
\end{aligned}
$$

Thus, it can be seen that the rule $[true]_{\mathrm{BN}} = 1$ states that 1 is BN-presentation of *true*. The rule $[x \vee y]_{\mathrm{BN}} = [x]_{\mathrm{BN}} + [y]_{\mathrm{BN}}$ is a BN-rule containing LHS_pattern $x \vee y$ that, in the case of $(a \vee b)$, maps x to a and y to b, (x a, y b), and RHS_pattern $[x]_{\mathrm{BN}} + [y]_{\mathrm{BN}}$ with the

representations of $[a]_{BN}$ and $[b]_{BN}$. The coupling invariant $x \equiv [x]_{BN} > 0$ then states that the BN-representation of *x* is positive iff *x* has the value *true*. Every rule of BN maintains this invariant.

The last rule of transform **BN** shows that the identity transform ID does also conform to the invariant. Its contribution is that converts any boolean variable from its original form to the new representation reserving the same behaviour.

A transform can be composed with other transforms to allow the representation of complex abstract data types [62]. Any transform can be understood as the definition of a function from $T_1 \rightarrow T_2$, it takes arguments represented by T_1 and produces results represented by T_2. The concatenation of transforms produces a composition (T_1**;** T_2). Thus, a transform expression (*tr_exp*) denotes one of the following: a basic transform identifier *id*, the identity transform identifier (ID), a function transform, or a composition of two transform expressions.

$$tr_exp ::= id \mid \mathrm{ID} \mid tr_exp \rightarrow tr_exp \mid tr_exp\mathbf{;}\ tr_exp$$

These transforms satisfy the following rules, where T, T_1, T_2, T_3 and T_4 are transform expressions:

$$T\mathbf{;}\ \mathrm{ID} = T$$

$$\mathrm{ID}\mathbf{;}\ T = T$$

$$(T_1\mathbf{;}\ T_2)\mathbf{;}\ T_3 = T_1\mathbf{;}\ (T_2\mathbf{;}\ T_3)$$

$$(T_1 \rightarrow T_2)\mathbf{;}\ (T_3 \rightarrow T_4) = (T_1\mathbf{;}\ T_3) \rightarrow (T_2\mathbf{;}\ T_4)$$

The first two laws state that the identity transform is an identity of **;**. The other two laws state that **;** is associative and that **;** is distributive over $\rightarrow$.

7.1.2 Program Transformation

Once a set of transforms, transform directives and the above rules are given, the general transformation process from a program *P* to an equivalent program *P'* is achieved by replacing each identifier of *P* by a representation identifier given by a transform expression. For instance, the transform directive

implement *S* **using** *Set_List*

replaces identifier *S* by a new identifier denoting its *Set_List*-representation. A set of transform directives forms a mapping called a transform environment. A transform environment binds all free identifiers of a given expression to a transform expression.

The substitution of identifiers follows an applicative order; *i.e.*, transformations and transform directives are applied by performing a bottom-up traversal of the abstract tree. At each leaf of the tree, transform directives are used to construct the representation of the node. As each node is visited, the representations and transformation of that node are constructed. If no transform is mapped to an identifier, then no transformation is applied. On the other hand, it may be possible to construct more than one transformation for the program. In such cases, heuristic methods, controlled by the user, can be employed to construct an appropriate transformation; they could be based on the cost of operations involved in a replacement or the relative order of a pattern with respect to other patterns [17].

7.2 DiSTiL

DiSTiL is a software generator that implements a declarative domain-specific language (DSL) for container data structures [60]. Its motivation is that data structures should not have ad hoc interfaces. Instead they should provide a stable, homogeneous, well-designed interface that insulates applications from changes to data structure implementations. DiSTiL implementation is the first available *transformation library* for IP.

DiSTiL's authors consider it to be a representative of a new approach to domain-specific language implementation, because of its emphasis on the declarative specification of domain-specific constructs through component composition. DiSTiL follows the *GenVoca* methodology [11], where the integral part is to identify the fundamental building blocks of software construction for a target domain. These components actually define program transformations that convert domain-specific language constructs into their host language implementation. The claimed advantage of this approach is *scalability*.

As a transformation library, DiSTiL deals extensively with manipulation of code fragments. Programming languages usually determine the meaning of identifiers using their position in a program. Generated programs, however, are usually composed from small fragments, and it is typically the case that we are unaware of the final position and scope of a fragment in the generated code. To deal with this problem, a *generation scoping* mechanism is provided: a basic meta-programming system for IP in which DiSTiL components are expressed [61]. The system consists of *code template operators*, similar to the `backquote` and `comma` operators in the LISP language, that are instantiated accordingly to the context where they are used. Generation scoping is a general-purpose facility oriented towards large-scale code generation and was not designed to support only DiSTiL. This facility is meant to be an answer to the ambiguity problems typically found in macro-expansion methods and the environment in which generated variable references are resolved.

DiSTiL language extends the C programming language with declarative statements and operations on data structures. These statements isolate the actual data structure implementation from the application itself, thereby allowing radically different implementations of data structures to be evaluated without requiring modifications to the application's source code [10].

All data structures in DiSTiL are modelled using *containers* and *cursors* (iterators). These two facets explicitly de-couple the notion of element storage from that of element access. The cursor-container pair provides the only interface the user has to a data-structure. When viewing a data structure as a collection of elements, the most important operation that can be defined is that of a *selection*. A selection gives the user a way to define a subset of a collection according to a certain *selection criterion* and a retrieval order if necessary. This is achieved by assigning *selection predicates* to cursors. These predicates describe arbitrary relations as understood in relational model front-ends and databases, but in addition DiSTiL couples these abstractions with component technologies to generate vast families of efficient implementations. The following example, taken from [60] shows the basic use of cursors and containers for a phonebook:

```
// C struct declaration
Typedef struct {
      char [8] phone;
      char [31] name;
} phonebook_record ;
```

```
// abbreviated container declaration
Container (phonebook_record) cont1;

// cursor declaration
Cursor (cont1, phone == "283512") curs1;

// cursor declaration
Cursor (cont1, name > "Am" && name < "An") curs2;
```

They offer a regular interface independent of the final implementation. Containers can be opened and closed, and they can return their current number of elements or tell if they are full. Cursor operations allow actions such as: `insert`, `update`, `delete`, `goto_first`, `goto_next`, `goto_prev`, `goto_nth`, `is_legal`, `foreach`, `getrec` and `ref`. The `foreach` construct is used to iterate over elements in a cursor. The element at the current cursor position can be examined, updated or deleted using standard cursor operations. For example:

```
// for each selection entry
foreach(curs1) {
      //print name
      printf("$s", ref(curs1, name));

      //change phone number
      update(curs1, phone, "283513");
}
```

The actual data structure of the phonebook can be implemented in different ways. For example, it could be implemented as an ordered linked list, a binary tree, a hash-table, or any other structure or combination of structures. Nevertheless, the above program fragment would remain the same across all different implementations.

DiSTiL allows the user to define the features of the data structures to use and declare how their implementations are to be generated when necessary. For this, *type equations* are used. In the following example shows how the phonebook can be implemented as a hash table (`Hash`) in conjunction with a red-black tree (`Tree`) with elements that are allocated when needed (`Malloc`) from main memory (`Transient`).

```
// type equation specification
typeq (phonebook_record,
       Hash(Tree(Malloc(Transient)))) type1;

// container declaration
Container (type1, (Hash(phone), Tree(name))) cont1;
```

The container declaration specifies that the hash table is organised by phone number (for fast lookups by phone) while the red-black tree has the name field as its key (for fast retrieval of alphabetically ordered names). The type specification is independent of its container and cursor use, altering the type equation will not affect them and will only require compilation.

If no type equation is specified, DiSTiL can *statically* determine an efficient way by analysing the predicate, estimating the cost of the retrieval using each available index and selecting the data structure data offer the lowest cost. If the composition describing our

data structure changes, DiSTiL will re-evaluate the cursor predicate and choose an appropriate way to implement its operations in the new layout without requiring any programmer intervention.

When a DiSTiL program is "compiled", the declarative data structure specifications are replaced by their C implementation, which is specified by a composition of DiSTiL components. Notice that cursor actions are specified declaratively instead of operationally. The system transforms specifications into efficient code using its knowledge of the characteristics of the given data structure.

Cursors and containers can be composed arbitrarily. Thus we can have a data structure storing cursors, or containers, or containers of containers, etc. A composition and the operations performed on it can be validated to ensure that they are meaningful. This is the role of the *design rule checker* of DiSTiL. The checker works based on explicitly encode boolean attributes associated with DiSTiL components, which correspond to higher-level knowledge about their properties. This information may be used to express domain-specific properties like "this component does not leave the cursor in a valid position after deletion" or "this component keeps track of the data structure size". Compositions are checked in two ways: the system ensures that all their components can co-exist and that they support all operations performed on the particular composition. This is possible because attributes associated with components are at a much higher level than regular static information (types) in programming languages. As a consequence, the checking mechanism can provide more informative, comprehensive and accurate error messages [10].

7.3 Polya and DiSTiL

Even though DiSTiL and Polya have different approaches to the control and transformation of data structures in programs, they share particular common goals. In the general sense, both provide data abstractions and program transformation mechanisms for the user to concentrate his attention on domain-specific problems and their algorithms. This improves reasoning about domain-specific problems, as they are kept independent from any data implementation option.

Also, they allow the user to direct the data selection process during the generation of the target code or a transformation process between abstract levels of the specification. Certainly, each system represents an alternative mechanism for code reuse and maintenance.

There are several differences between them. One the one hand, Polya provides higher level abstraction mechanisms to represent programs; this simplifies code generation for multiple target languages. Furthermore, Polya is a transformation system where not only abstract-to-concrete transformations can be applied, but also abstract-to-abstract transformations. This means that transformations are not necessarily target-code oriented but that further manipulation of specifications can be made at different transformation stages when appropriate. On the other hand, data components in DiSTiL have domain-specific properties that ensure that components can co-exist and that they support all operations performed on the particular composition.

8 Discussion

Attribute Grammars and Intentional Programming

We have discussed how attribute grammars allow the description of horizontal and vertical program transformations, and surveyed several AG-based systems, exploring their specification features. In general, their declarative specification of transformations is characterised by the description of *what* instead of *how*, which is a very desirable property to have in our IP meta-language. This property is supported by the fact that application order of transformations does not need to be specified explicitly, as the evaluation engine determines it based on attribute computation dependence. This can be understood as a pre-processing activity. In addition to its features, the resemblance between attribute grammars and intentions in IP motives their study.

Some similarities and differences between attribute grammars and intentions in IP are presented below.

- Strong dependence on the AST structure. AST node computations in AG's and IP are based on this structure, its semantic dependencies and "virtual links" shared with other sections of the tree.

  Even so, we believe tree structure dependence needs to be reduced, or at least hidden from the programmer somehow, through the meta-language interface in order to allow reuse and modularity of tree elements and their computations. A good example and pioneering approach to solve this dependence problem is found in Eli's meta-language constructs. They de-couple attribute computations from the actual tree layout; allowing the abstract syntax to be extended with minimal modification, if any, of existing computations and specially their remote reference to non-local information.

  Particularly, the Eli's `SYMBOL` construct is of special interest. It allows computations to be implicitly associated with individual grammar symbols of the language rather than with productions to which they contribute. Therefore, `SYMBOL` computations can have its default behaviour "directly attached" to its corresponding symbol; such behaviour could involve providing different transformed versions of it. This is similar to the IP view that transformations may be regarded as processes that are automatically associated with programming language constructs, and their use can annotate a syntax tree with its appropriate transformations.

- Attribute definition rules and IP question handlers are similar in their goals. They carry useful context-sensitive information necessary for AST node computations. Both have local visibility; *i.e.*, its scope boundaries are delimited by the tree node context where they are defined. In the case of attribute grammars, direct subtrees and their attributes are also visible in the local context.

  But, while attributes actually contain such information, question handlers are the mechanism used to obtain it. A question handler will try to answer its question according to local information, if unable, it will delegate it to another "known" question handler only reachable through the links its current node has. Once a question handler forwards its question to a second question handler, the second one will try answering it according to its context, otherwise will forward it again, thus forming a "forwarding chain". Under this forwarding mechanism, question handlers

that have delegated a question on others will wait for their answer, but it may happen that an answer might never be given.

Moreover, under the current reduction engine, R5, a given answer can be "invalidated" later on, meaning that the context under which the answer was originally supplied has changed. Such invalidation forces to reverse the reduction process back to the point where that answer was first required and try again with another transformation order. In order to be able to do that, an existing rollback mechanism is used together with this *additive* reduction engine, where its key point is based on the fact that non-destructive operations are applied on the tree structure during transformation.

In attribute grammars, re-computation of attribute values is a typical activity in editing environments; *e.g.*, syntax-directed editors. Here, upon tree modification cause by the user, affected tree nodes re-compute their context-dependent values according to change-propagation strategies. See [37, 55] for more detail on incremental attribute update algorithms. In attribute grammars an IP rollback mechanism it not necessary, as evaluation order is predefined.

- Attribute values can also be understood as answers to general questions. As a special case, attributes can mimic binary questions and answers. For this, a boolean attribute value serves the purpose and allows control flow to be diverted upon its value. For example, review error message handling in our case study (page 70).

  But attributes can be used for other purposes. Eli shows attributes being used as the typical standard value passing mechanism and as *preconditions* to other computations. In the latter, attributes are used to help ensure side-effects have already occurred before other computations take place.

  We still need to confirm whether attributes and their evaluation strategies can carry out more complex tasks achieved by question handlers.

- A single attribute can be associated with the computation of one or more aspects of a program representation, for example the *type*, the *code* or the *environment*. Similarly, an intention makes use of xmethods to render its role and contribute in computing these different aspects.

  As our case study shows, intentions can be mapped into productions and their attribute computations. A single intention can be represented as a set of attribute computations that exist in different production rules of the grammar. They can be understood as aspects or areas of concern an AG-based compiler has to deal with.

Case study: lessons learned

The case study has shown the way context-sensitive information is propagated through a program representation structure using higher order attribute grammars. During transformation, it allows global changes to be effectively produced by local properties, as illustrated in the `after` expression transformation. Similarly, local changes can be produced by global properties, as the information contained in a `block` statement has shown.

In addition, we have exercised an aspect-oriented style in the specifications of this case study. This is, all necessary computations associated with a particular attribute are expressed together in a single place crossing the production rule boundaries. In our experience, the use of this style and a sub-task decomposition approach show that more

control and maintainability of the specification is obtained when compared to the pure attribute grammar specification style. This is relevant when specifying a significant number of sub-tasks over a large grammar structure.

We have rendered the `after` expression intention using attribute grammars. It transforms itself into simpler existing constructs. For this, the actual semantic sub-tasks that carry out the intention were identified and then each of them was expressed individually in the AG specification (see page 33). However, we have shown that there may be multiple transformations for a single interpretation; this is particularly true when studying the new construct's interpretation under different contexts, where undoubtedly some of them will be harder to specify than others. Therefore, it is sensible to analyse the dependence levels these sub-tasks implicitly have.

We believe the simple and "harmless" transformations discussed and implemented in our case study have yielded relevant points that need attention when extending programming languages in IP. The mapping of new constructs into exiting ones required deeper understanding than what we initially would have thought. In an early attempt to introduce a new construct, we assumed and analysed a simple transformation and a single interpretation. However, it seems to us that such analysis needs to go beyond the actual scope of the new construct itself and get into its interactions with existing constructs and other predefined "properties" of the language, *e.g.*, operators' precedence, scope, parameter passing mechanisms and evaluation order. Which is the inclusion strategy to follow when other new constructs are added? Would it be that these predefined properties are also intentional elements with which new intentions simply need to interact? How to make this interaction as anonymous as possible?

Our initial approach was for these constructs to be completely independent, so that they would be understood as "pure" intentions and would be used as basic building blocks for different languages. Under this scg-10(for-t25emete)-6 intew integuages intebe as simple as selecting constructs from available "intentional catalogues" and shuffle them together. But our case study provides evidence that there are subtle interrelationships between constructs that prevent complete independence and therefore such a simple b25emete activity. How acceptable is this dependence within the IP paradigm? How does one ensure the semantic eoal of a new construct is achieved when interactete with other existing constructs/intentions? Moreover, eiven a set of intentions, which is their resulting collective semantics? Perhaps some kind of intentions will be meaneteful in a inteontext if and only if other specific ones are already present. If this dependence were true, then it would suggest that a minimal set of other vital intentions had to be provided as the infrastructure or framework for the new intention to be used. Would this mean that intentions need to be provided as "intentional packages" instead of just individual elements? ion, we think that a public interface or "contract" will have
to be specified to ensure such co-existence.

Horizontal transformations with more contextual information

In our case study, we have seen the way in which AGs compute ontext-sensitive information durete source-to-source (horizontal) transformations; attribute values are passed up and down the tree structure in order to carry out the necessary computations. Not surprisetely this activity is very similar to what is done when usete attribute erammars on syntax-directed editors. In such editors, once a transformation has taken

place, an incremental attribute evaluation ensures changes are propagated and values are consistently updated through out the tree structure.

We believe context-sensitive information will lead to improvements in program re-writing using rule-based transformations with pattern matching. In our view, term rewriting benefits from this information, as the transformation engine will be able to "select" optimising transformation paths depending on its context (*e.g.*, type). Then on each rewriting step, an incremental attribute evaluator will re-compute and update the context, offering consistent information to the next rewriting step. Moreover, additional information can be obtained from a control flow or data flow analysis.

We plan to study in depth how to integrate coherently this mechanism with the rule rewriting system under development by de Moor and Sittampalam.

Mechanised data refinement, data abstraction and transformations

Regarding mechanised data refinement, Polya shows an interesting and elegant approach to the transformation of complex data structures. Such transformations are intended to simplify the specification and compilation of domain specific languages. Polya and IP share a very similar goal: the best abstraction of an application is its specification expressed in the right domain specific language. With this view, programmers should be allowed to write more natural and direct specifications instead of forcing them to use or to implement their own ad hoc interfaces in a standard programming language.

Transforms, as understood in Polya, allow an interesting declarative specification of a single datatype transformation step, and demonstrate how they can be composed into more sophisticated, multi-step transformations. We visualise transforms as a form of "groups of intentions" whose goal is precisely to provide a data refinement strategy for a particular datatype. Such group would essentially be the union of other intentions in charge of each of the given transformation rules — perhaps extra information would be required to co-ordinate their application. We also believe transforms may have a direct mapping into rewrite rules using pattern matching, where contextual information will let *coupling invariants* to be verified and enforced ensuring the equivalence preservation through the automatic process. Here again, attribute grammars are a good candidate for the job. We want to study closer both approaches and have more concrete view of these potential higher level intentions.

DiSTiL presents data abstraction in the form of a regular, declarative data-accessing interface that can simplify algorithmic obscurities in program specifications. Here, data abstractions are coupled with families of efficient implementations, which are selected depending on the usage and cost of that data representation in the current program and its interaction of other data abstractions. In DiSTiL, as well as in Polya, the user is allowed to introduce his judgement and directly select the implementation of specific data abstractions.

We believe horizontal program transformations can take advantage of the different data abstraction levels that data abstractions can introduce, because it would allow reasoning about data representations and algorithms, possibly individually, at distinct moments of the transformation process. Our next step in this direction will be focused in the intentional representation of data refinement and data abstractions, and the way it will offer control over the implementation exposure of datatypes and their associated operations. This are still a source of open questions for us, as we need to conciliate them with our multi-step transformation process.

On defining our meta-language

We review attribute grammars and mechanised data refinement from the IP perspective as a first step towards the definition of our meta-language. As a team, we aim at the identification of the minimal set of features the meta-language should provide as suitable notation for describing intentions and their program transformations.

With the same goal in mind, Sufrin and Backhouse [9] are currently conducting an experimental research approach by "intentionalising" WordPad. This will allow determining the kind of intentions that need to be described for such kind of applications and the meta-language features that are necessary for it. Preliminarily, they consider an intentional framework will be required for the specification of user interface intentions.

As a general conclusion, we have found interesting features in the reviewed systems that may benefit our meta-language and provide potential design guidelines that are worth consideration. In particular, the Eli's inheritance mechanism allows reusability and modularity to be introduced in attribute grammars. These three features substantially increase the specification abstraction level and we believe they reduce specification size in general. Its modularity is also used to produce abstract data types, thus providing a data and implementation-hidden mechanism. Eli also provides remote referencing facilities (*e.g.*, `CHAIN`, `INCLUDING` and `CONSTITUTES`) whose main contribution is to allow specifications to have a degree of independence from the actual grammar structure. Remote references are indeed a key element for reusability.

Regarding the specification style, we have written aspect-oriented specifications in our case study and recognised its contribution in helping to focus the programmer's attention on individual concerns one at the time.

We also believe it is reasonable to think that data abstraction and refinement as explicit representations are important in our meta-language because of the strong orientation towards imperative programs as target languages.

Finally, a very important feature that our meta-language must not omit is to allow users to extend their programming languages and the meta-language itself by defining new constructs and their behaviour. For this, it has to allow new grammar rules for new programming constructs to be included.

9 Future Work

This initial study proposes several areas that we consider require more understanding, and where a satisfactory solution could offer valuable contributions to the IP paradigm, its meta-language and its realisation.

Goal

The main goal of my research is to provide more understanding on the meta-specification, interaction and transformation of intentions for the purpose of data refinement in IP. It will be firmly based on convincing motivating examples.

Tentative tasks

To achieve the main goal, we are going to pursue two tentative tasks. We hope they will provide enough understanding to satisfy most of the cast doubts on our discussion section.

Firstly, we are going to offer a representation mechanism for the description of data refinement intentions in IP. We visualise the data refinement as modular "groups of intentions", where several closely related intentions may be specified together so that a common goal could be reached. Moreover, we consider they will help to identify different stages in our multi-step conversion, and allow a potential classification of the reasoning levels we can apply at each step.

These intentions will have to provide extra information, apart from purely refinement specifications, to ensure equivalence preservation is maintained. Our impression is that data about intentions' "co-existence" and "co-ordination" could be necessary as part of this extra information. Such data would provide the intention's algebraic properties such that composition and, hopefully, calculation of optimisations were applicable. We expect to be able to express this information in the most intentional way possible, ultimately as intentions.

If we succeed in the meta-specification of data refinement, we believe the same mechanism will be applicable to the specification of data abstractions in a very similar way. This is based on our understanding that transformations in data refinement and data abstraction mainly differ on the level of implementation exposure about a datatype they provide.

Secondly, we want to provide updated context-sensitive information during the conversion process and certainly when applying data refinement intentions on imperative programs. We believe that this information will lead to more code improvements and successful optimising opportunities for the rule-based transformation engine to operate during horizontal transformations. We hope to be able to validate it through a prototype and concrete examples. In addition, it will be the means to study the subtle interactions between intentions and the effects their individual transformations produce over the global context.

To achieve this, we consider higher order attribute grammars to be an appropriate mechanism for our purposes, not only for what we have already stated in this document,

but also because of their incremental evaluation strategies. Here, our meta-specification of intentions will have to allow the compilation of attribute computations in order to build the attribute grammar infrastructure. This will definitely provide substantial insights about the meta-language features that data refinement requires and its evaluation strategy. Then, we will integrate our prototyped mechanism with Ganesh Sittampalam's pattern matching and transformation system. In case further context-sensitive information is necessary, we plan to study the contribution control flow and data flow analysis could bring to the conversion process of imperative programs.

Activity Plan

The following activity plan is proposed to achieve our main goal during the following two years:

1. Microsoft Internship (2 months): during this period I will be in closer contact with the current IP implementation. There, I plan to study the way data refinement can be realise in this implementation.
2. Data refinement as intentions (3 months): during this time I will focus my research in discovering the most appropriate way to describe mechanised data refinement intentionally, including the minimum infrastructure for its realisation.
3. Study of Ganesh Sittampalam's rule-based transformation system and pattern-matching mechanism. (1 month)
4. Context-sensitive information available during the application of rule-based transformations and its integration with the rule-based transformation system. (1 month)
5. Reading of related topics: Among them aspect-oriented programming, exploration of code improving transformations, algebraic specifications in TICS, control flow and data flow analysis tools.
6. Prototype specification and construction (12 months): It will mainly consider the following specific points:
    - Specification of data refinement intentions.
    - Continuously updated context sensitive information.
   - Integration with rule-based transformation system and pattern-matching mechanism.
    - Identification of interesting rule-based transformations on imperative programming.
7. Conference paper (1 month)
8. Thesis writing (5 months)

In addition, we plan to participate on one or two scientific conferences closely related to our research interests.

10 References

[1] A. V. Aho, R. Sethi and J. D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley, Reading, Mass ; Wokingham, 1986.

[2] W. Aitken. Personal communication during his visit to the OUCL, February 1999.

[3] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter and C. Simonyi. Transformation in Intentional Programming. In J. Poulin, ed. *Procs. 5th International Conference on Software Re-Use*, IEEE Press, 1998. Available from URL: http://www.research.microsoft.com/ip/.

[4] H. Alblas. Attribute Evaluation Methods. In H. Alblas and B. Melichar, eds., *Attribute Grammars, Applications and Systems*, vol. 545 of Lecture Notes in Computer Science, pages 48-113, International Summer School SAGA, Prague, Czechoslovakia : proceedings, Springer-Verlag, 1991.

[5] L. Augusteijn. The Elegant Compiler Generator System. In P. Deransart and M. Jourdan, eds., *Attribute Grammars and Their Applications*, vol. 461 of Lecture Notes in Computer Science, pages 238-254, International Conference WAGA, Paris, France : proceedings, Springer-Verlag, 1990.

[6] L. Augusteijn. Functionl Programming, Program Transformations and Compiler Construction. PhD thesis, Eindhoven University of Technology, The Netherlands, 1993. See also: http://www.research.philips.com/generalinfo/special/elegant/elegant.html.

[7] L. Augusteijn, P. Jansen and H. Munk. *The Elegant Compiler Generator Tool Set, Release 7.0.* Philips Research Laboratories, The Netherlands, 1996.

[8] K. Backhouse, I. Sanabria-Piretti and G. Sittampalam. Using the R5 Reduction Engine, Oxford University Computing Laboratory, Oxford, Technical Report April 1999. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[9] K. Backhouse and B. Sufrin. Intentions in WordPad. 1999, pages 22. Unpublished paper. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[10] D. Batory and B. J. Geraci. Validating Component Compositions in Software System Generators. In Sitaraman. M, ed. *Fourth International Conference On Software Reuse, proceedings*, pages 72-81, Orlando, FL, 1996.

[11] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, vol. 1(4), pages 355-398, 1992.

[12] O. de Moor. First-class attribute grammars. February 1999, pages 27. Unpublished paper. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[13] O. de Moor, S. Peyton-Jones and E. Van Wyk. Aspect-Oriented Compilers. 1999, pages 13. Unpublished paper. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[14] O. de Moor and G. Sittampalam. Generic Program Transformation. *Procs. 3rd International Summer School on Advanced Functional Programming*, pages 34, Portugal, Springer Verlag, 1998. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[15] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. 1999, pages 27. Unpublished paper. Available from URL: http://www.comlab.ox.ac.uk/oucl/groups/progtools/publications.htm.

[16] P. Deransart, M. Jourdan and B. Lorho. *Attribute grammars : definitions, systems and bibliography*. Springer-Verlag, Berlin; London, 1988.

[17] S. Efremidis. On Program Transformations. PhD. thesis, Cornell University, 1994.

[18] S. Efremidis, K. Mughal, L. Søraas and J. Reppy. AML: Attribute Grammars in ML. *Nordic Journal of Computing*(January), 1997. Available from URL: ftp://ftp.ii.uib.no/pub/aml/jc-aml-paper.ps.Z.

[19] Eli-Developer-Group. Personal communication, February - April 1999. Accessible through Eli WWW site at: http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_homeE.html.

[20] J. Engelfriet. Attribute Grammars - Attribute Evaluation Methods. In B. Lorho, ed. *Methods and Tools For Compiler Construction an Advanced Course*, pages 103-138, Advanced course on methods and tools for compiler construction, Inst Natl Rech informatique & Automatique, Le Chesney, France, 1984.

[21] H. Ganzinger and R. Giegerich. Attribute Coupled Grammars. *Sigplan Notices*, pages 157-170, Assoc for Computing Machinery Special Interest Group in Programming Languages 84 Symp on Compiler Construction, Papers Presented, Montreal, Quebec, Canada, 17 - 22 June, 1984.

[22] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane and W. M. Waite. Eli: A Complete, Flexible, Compiler Construction System. *Communications of the ACM*, vol. 35(2), pages 121-131, 1992.

[23] D. Gries. *The science of programming*. Springer-Verlag, New York, 1981.

[24] D. Gries and J. Prins. A New Notion of Encapsulation. *Proceedings of the ACM Sigplan 85 Symposium On Language Issues in Programming Environments*, pages 131-139, Symp on Language Issues in Programming Environments, Seattle, WA, 25 - 28 June, 1985.

[25] D. Gries and D. Volpano. The Definition of Polya, Department of Computer Science, Cornell University, Technical Report 1992.

[26] D. Gries and D. Volpano. The transform - a new language construct. *Structured Programming*, vol. 11, pages 1 - 10, 1990.

[27] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, vol. 1, pages 271-281, 1972.

[28] P. Jansen, L. Augusteijn and H. Munk. *Introduction the Elegant*. Philips Research Laboratories, The Netherlands, 1996.

[29] T. Johnsson. Attribute grammars as a functional programming paradigm. In K. G., ed. *3rd Conf. on Functional Programming Languages and Computer Architecture*, vol. 274 of Lecture Notes in Computer Science, pages 154-173, Portland, Springer-Verlag, 1987. Available from URL: http://www.cs.chalmers.se/~johnsson/.

[30] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 81-93, Montreal, ACM Press, 1984.

[31] M. Jourdan, D. Parigot, C. Julie, O. Durin and C. LeBellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 25(6.), pages 209-222, 1990.

[32] A. Kastens. *Eli: Compiler Construction Made Easy*. Compiler and Programming Language Group, University of Paderborn, May 1999. Joint Project University of

Colorado at Boulder, University of Paderborn and James Cook University. On the World Wide Web at URL: http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_homeE.html.

[33] U. Kastens. Attribute Grammars as a Specification Method. In H. Alblas and B. Melichar, eds., *International Summer School on Attribute Grammars, Applications and Systems : SAGA*, vol. 545 of Lecture Notes in Computer Science, pages 16-47, Prague, Czechoslovakia, June 4-13 : proceedings, Springer-Verlag, 1991.

[34] U. Kastens. Attribute Grammars in a Compiler Construction Environment. In H. Alblas and B. Melichar, eds., *International Summer School on Attribute Grammars, Applications and Systems : SAGA*, vol. 545 of Lecture Notes in Computer Science, pages 380-400, Prague, Czechoslovakia, June 4-13 : proceedings, Springer-Verlag, 1991.

[35] U. Kastens. LIDO - Computations in Trees, Compiler and Programming Language Group, University of Paderborn, Paderdon, Germany, Reference Manual, Revision 4.10, 1997. Available from URL: http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_homeE.html.

[36] U. Kastens. LIDO - Reference Manual, Compiler and Programming Language Group, University of Paderborn, Paderdon, Germany, Reference Manual, Revision 4.21, 1997. Available from URL: http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli_homeE.html.

[37] U. Kastens. Ordered Attributed Grammars. *Acta Informatica*, vol. 13(3), pages 229-256, 1980.

[38] U. Kastens, P. Pfahler and M. Jung. The Eli system. In K. Koskimies, ed. *7th International Conference on Compiler Construction (CC 98) at the Joint European Conferences on Theory and Practice of Software (ETAPS 98)*, pages 294-297, Lisbon, Portugal, 28 Mar - 4 April, 1998.

[39] U. Kastens and W. M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, vol. 31, pages 601-627, 1994.

[40] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, 1988.

[41] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, eds., *11th European Conference on Object-Oriented Programming (ECOOP 97)*, vol. 1241 of Lecture Notes in Computer Science, pages 220-242, Jyvaskyla, Finland, Springer-Verlag, 1997. Available from URL: http://www.parc.xerox.com/spl/projects/aop/.

[42] D. E. Knuth. *The art of computer programming*. Addison-Wesley Pub. Co., Reading, Mass., 1973.

[43] D. E. Knuth. Semantics of Context-Free Grammars. *Mathematical Systems Theory*, vol. 2(2), pages 127-145, 1968.

[44] D. E. Knuth. Semantics of Context-Free Grammars (correction). *Mathematical Systems Theory*, vol. 5(1), pages 95-96, 1971.

[45] M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. *CSN'87: Computing Science in the Nertherlands*, SION, 1987. Available from URL: ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz.

[46] P. Kwiatkowski. Intentional Programming R5 Engine. Seminar presented at Oxford University, November 1998. Power Point presentation.

[47] Microsoft-IP-Development-Team. Electronic communication, January-June 1999.

[48] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, vol. 27, pages 481- 503, 1990.

[49] J. M. Morris. The laws of data refinement. *Acta Informatica*, vol. 26, pages 481 - 503, 1989.

[50] R. Paige. Future Directions in Program Transformation. *ACM Computing Surveys*, vol. 28(4es), 1996. Available from URL: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a170-paige/.

[51] D. Parigot. *Bibliography on Attribute Grammars*. Parigot, Didier, June 1999. On the World Wide Web at URL: http://www-rocq.inria.fr/oscar/www/fnc2/AG.html.

[52] D. Parigot, E. Duris, G. Roussel and M. Jourdan. Attribute grammars: a declarative functional language, INRIA, Rapport de Recherche, 2662, October 1995.

[53] A. Pettorossi and M. Proietti. Future directions in program transformation. *ACM Computing Surveys*, vol. 28(4es), 1996. Available from URL: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a171-pettorossi/.

[54] J. H. Reppy and E. R. Gansner. A Foundation For Programming Environments. *Sigplan Notices*, pages 218-227, 2nd Software Engineering Symp on Practical Software Development Environments of the Assoc for Computing Machinery Inc, Palo Alto, CA, 1987.

[55] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator : a System for Constructing Language-based Editors*. Springer-Verlag, New York, 1989.

[56] C. Simonyi. The Death of Computer Languages - The Birth of Intentional Programming, Microsoft Research, Presentation at IFIP WG 2.1 on Algorithmic Languages and Calculi, Technical Report, MSR-TR-95-52, September 1995. Available from URL: http://www.advtech.microsoft.com/pubs/msr-bib.htm.

[57] C. Simonyi. The Future is Intentional. *IEEE Computer*(May), 1999. Available from URL: http://www.research.microsoft.com/ip/.

[58] C. Simonyi. *Intentional Programming - Innovations in the Legacy Age* 1996. On the World Wide Web at URL: http://www.research.microsoft.com/research/ip.

[59] C. Simonyi. Invited talk given at Carnegie Mellon University, April 1999. Available from URL: http://www.research.microsoft.com/ip/.

[60] Y. Smaragdakis and D. Batory. DiSTiL: a transformation library for data structures. *Proceedings of the Conference On Domain-Specific Languages*, pages 257-269, Santa Barbara, CA, Usenix Association, 1997. Available from URL: http://www.cs.utexas.edu/users/smaragd/research.html.

[61] Y. Smaragdakis and D. Batory. Scoping Constructs for Program Generators, Department of Computer Sciences, University of Texas, Austin, Technical Report, TR-96-37, 1996. Available from URL: http://www.cs.utexas.edu/users/smaragd/research.html.

[62] A. van der Berg. Data Abstraction by Program Transformation in a Higher-Order Attribute-Grammar Framework. PhD thesis, Cornell University, 1997. Available from URL: http://www.cs.cornell.edu/home/aswin/.

[63] H. Vogt. Higher Order Attribute Grammars. PhD thesis, Utrech University, 1993. Available from URL: http://www.serc.nl/.

[64] H. Vogt, S. D. Swierstra and M. F. Kuiper. Higher Order Attribute Grammars. *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 131-145, Portland, Oregon, June, 1989.

[65] W. M. Waite. *A Complete Specification of a Simple Compiler*. Department of Electrical and Computer Engineering, University of Colorado at Boulder, March 1997. An

implementation of the Pascal- language. On the World Wide Web at URL: ftp://ftp.cs.colorado.edu/pub/cs/distribs/eli/Examples/.

[66] W. M. Waite and G. Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

[67] W. M. Waite and U. Kastens. Personal communication, February - April 1999.

[68] N. Wirth. Program development by step-wise refinement. *Communications of the ACM*, vol. 14, pages 221-227, 1971.

11 Appendix

11.1 Appendix 1: Semantic definitions for *P* using SG

The following is the complete definition of the HAG written in SG. The example's structure is based on [55], and it has been extended for the purposes of the case study presented in Section 4, page 13.

```
/*-----------------------------------------------------------------------------
 *                    SEMANTIC DEFINITIONS
 *             OF THE HAG FOR THE P PROGRAMMING LANGUAGE
 *--------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------
 * Semantic Root node ProgBody. A local variable env is defined.
 *--------------------------------------------------------------------------*/

progBody:     ProgBody { local declList env;
                         env = declList; }
       ;

/*-----------------------------------------------------------------------------
 * TYPE Attribute
 *--------------------------------------------------------------------------*/

exp, variable, funcName    { synthesized type type; };

exp    :      EmptyExp      { exp.type = EmptyType; }
       |      IntConst      { exp.type = IntType; }
       |      True, False   { exp.type = BoolType; }
       |      Id            { exp.type = variable.type; }
       |      Equal, NotEqual     { exp$1.type = BoolType; }
       |      Add, Minus    { exp$1.type = IntType; }
       |      After         { exp$1.type = exp$2.type; }
       |      FuncCall      { exp$1.type = funcName.type == EmptyType ?
                              LookupFuncType(funcName,
                                        {ProgBody.initialFuncDecls}) :
                              funcName.type; }
       ;
variable:     VariableNull  { variable.type = EmptyType; }
       |      Variable      { variable.type =
                               LookupDeclType(variable, {ProgBody.env}); }
       ;

funcName:     FuncNameNull  { funcName.type = EmptyType; }
       |      FuncName      { funcName.type =
                               LookupFuncType(
                                  funcName,
                                  {ProgBody.initialFuncDecls}); }
       ;

/*-----------------------------------------------------------------------------
 *  VALUE Attribute
 *  Every expression has a value.  In most cases, it contains a computed copy
 *  of its associated expression.  In the case of an After expression, a
 *  funcCall is created.
 *--------------------------------------------------------------------------*/

exp          { synthesized exp value; };

exp    :      EmptyExp    { exp.value    = EmptyExp; }
       |      IntConst    { exp.value    = IntConst(INTEGER); }
       |      True      { exp.value    = True; }
       |      False     { exp.value    = False; }
```

```
        |       Id        { exp.value   = Id(variable); }
        |       Equal     { exp$1.value = Equal(exp$2.value, exp$3.value); }
        |       NotEqual   { exp$1.value = NotEqual(exp$2.value, exp$3.value); }
        |       Add       { exp$1.value = Add(exp$2.value, exp$3.value); }
        |       Minus     { exp$1.value = Minus(exp$2.value, exp$3.value); }
        |       After     { local funcName newFuncName;
                              exp$1.value = FuncCall(newFuncName); }
        |       FuncCall   { exp$1.value = FuncCall(funcName); }
        ;

/*----------------------------------------------------------------------------
 * NEWFNCDCLS ATTR
 * In most cases, the equations for this attribute correspond to
 * 'group and pass up to the tree' rules.
 * The most interesting equation is found on the After expression,
 * where the newFncDcls is extended with a new funcDecl.
 *---------------------------------------------------------------------------*/

progBody, funcDeclList,
funcDecl, block, stmtList,
stmt, exp     { synthesized funcDeclList newFncDcls; };


/* this CODE attribute is defined here of compilation purposes */
stmtList      { synthesized stmtList code; };

progBody:     ProgBody  {local funcDeclList initialFuncDecls;

                        initialFuncDecls = funcDeclList;
                        $$.newFncDcls =
                          AppendFncDclLst(
                            AppendFncDclLst(block.newFncDcls,
                                            funcDeclList.newFncDcls),
                            initialFuncDecls); }
        ;

funcDeclList: FuncListNil   { $$.newFncDcls = FuncListNil; }
        |     FuncListPair  { $$.newFncDcls =
                                AppendFncDclLst(funcDecl.newFncDcls,
                                  funcDeclList$2.newFncDcls); }
        ;
funcDecl:     FuncDecl      { $$.newFncDcls = block.newFncDcls; }
        ;

block  :      Block         { $$.newFncDcls = stmtList.newFncDcls; }
        ;

stmtList:     StmtListNil   { $$.newFncDcls = FuncListNil; }
        |     StmtListPair  { $$.newFncDcls =
                                AppendFncDclLst(stmt.newFncDcls,
                                  stmtList$2.newFncDcls); }
        ;
stmt   :      EmptyStmt     { $$.newFncDcls = FuncListNil; }

        |     Assign        { $$.newFncDcls = exp.newFncDcls; }

        |     IfThenElse    { $$.newFncDcls =
                                AppendFncDclLst(exp.newFncDcls,
                                  AppendFncDclLst(stmt$2.newFncDcls,
                                                  stmt$3.newFncDcls)); }
        |     While         { $$.newFncDcls =
                                AppendFncDclLst(exp.newFncDcls,
                                                stmt$2.newFncDcls); }
        |     Loop          { $$.newFncDcls = block.newFncDcls ; }
        |     Compound      { $$.newFncDcls = stmtList.newFncDcls;}
        |     Return        { $$.newFncDcls   = exp.newFncDcls; }
        |       Break, Continue,
              Label, Goto   { $$.newFncDcls = FuncListNil; }
```

```
       ;
exp    :      EmptyExp, IntConst,
              True, False, Id { $$.newFncDcls = FuncListNil; }

       |      Equal,NotEqual,
              Add, Minus    { $$.newFncDcls =
                                  AppendFncDclLst(exp$2.newFncDcls,
                                                 exp$3.newFncDcls); }

/* After expression builds a new function declaration and appends it to
 * NEWFNCDCLS attribute. */

       |      After
                { /*local funcName  newFuncName;*/
                  local funcDecl    newFunc;

                  newFuncName = FuncName(gensym("AfterFunc_",&($$)));
                  newFunc =
                    FuncDecl(newFuncName,
                            exp$2.type,
                            Block(AppendStTail(stmtList.code,
                                              Return(exp$2.value))));

                  exp$1.newFncDcls =
                     AppendFncDcl(
                        AppendFncDclLst(stmtList.newFncDcls,
                                       exp$2.newFncDcls),
                        newFunc); }

       |      FuncCall      {exp$1.newFncDcls  = FuncListNil; }
       ;

/*--------------------------------------------------------------------------
 * RETURNALLOW Attribute
 * This attribute is used to allow Return stmts depending on the context.
 * The Return stmt is NOT allowed in the block part of a Program
 * production rule, while it IS allowed in the block part of functions.
 * This attribute is remotely referred by error reporting functions.
 *-------------------------------------------------------------------------*/

block         {inherited   STR returnAllow; };

progBody :    ProgBody      {block.returnAllow = "NO"; }
        ;
funcDecl :    FuncDecl      {block.returnAllow = "YES"; }
        ;

stmt    :     Loop          {block.returnAllow = {block.returnAllow}; }
        ;

/*--------------------------------------------------------------------------
 * BRKCNTALLOW Attribute
 * Break and Continue stmts are allowed only in the body of a loop
 * stmt.
 * This attribute is remotely referred by error reporting functions.
 *-------------------------------------------------------------------------*/

block         {inherited   STR brkCntAllow; };

progBody :    ProgBody      {block.brkCntAllow = "NO"; }
        ;
funcDecl :    FuncDecl      {block.brkCntAllow = "NO"; }
        ;
stmt    :     Loop          { block.brkCntAllow = "YES"; }
        ;

/*--------------------------------------------------------------------------
 * INITIALLABEL, FINALLABEL Attributes
```

```
 * Any Break, Continue or Goto stmt within a Loop stmt need to refer
 * these two labels in order to compute its transformed 'code'.  Each
 * of this statements uses upward remote reference to the closest block
 * instance.  See the specification for Loop.code.
 *-----------------------------------------------------------------------*/

block         { inherited   labelName initialLabel; };

progBody :    ProgBody      {block.initialLabel = LabelNameNull;}
        ;
funcDecl :    FuncDecl      {block.initialLabel = LabelNameNull;}
        ;
stmt     :    Loop { block.initialLabel = LabelName(gensym("LpInitLbl_",
                                                     &($$))); }
        ;


block         { inherited   labelName finalLabel; };

progBody :    ProgBody      { block.finalLabel = LabelNameNull; }
        ;
funcDecl :    FuncDecl      { block.finalLabel = LabelNameNull; }
        ;
stmt     :    Loop { block.finalLabel = LabelName(gensym("LpFinalLbl_",
                                                  &($$))); }
        ;

/*------------------------------------------------------------------------
 * CODE Attribute
 * In most cases, a copy of the original tree structure is created and
 * returned.  In other cases, a transformed tree is computed and returned.
 *-----------------------------------------------------------------------*/

program            { synthesized program code; };
progBody      { synthesized progBody code; };
funcDeclList  { synthesized funcDeclList code; };
funcDecl      { synthesized funcDecl code; };
block         { synthesized block code; };
declList      { synthesized declList code; };
decl          { synthesized decl code; };
type          { synthesized type code; };
variable      { synthesized variable code; };
labelName     { synthesized labelName code; };
progName      { synthesized progName code; };
funcName      { synthesized funcName code; };
/*stmtList    { synthesized stmtList code; }; */
stmt          { synthesized stmt code; };

program  :  Prog    { $$.code =
                       Prog(progName.code, progBody.code);}
        ;
progBody :  ProgBody      { $$.code =
                       ProgBody(declList.code,
                              AppendFncDclLst(funcDeclList.code,
                                AppendFncDclLst(block.newFncDcls,
                                             funcDeclList.newFncDcls)),
                              block.code); }
        ;
declList :    DeclListNil   { $$.code = DeclListNil;}
        |     DeclListPair  { $$.code = (decl.code :: declList$2.code); }
        ;
decl     :    Declaration   { $$.code = Declaration(variable.code,
                                              type.code); }
        ;
type      :   EmptyType     { $$.code = EmptyType;}
        |     IntType              { $$.code = IntType;}
        |     BoolType      { $$.code = BoolType;}
        ;
```

```
variable:     VariableNull  { $$.code = VariableNull; }
        |     Variable      { $$.code = Variable(IDENTIFIER); }
        ;
labelName:    LabelNameNull { $$.code = LabelNameNull; }
        |     LabelName     { $$.code = LabelName(IDENTIFIER); }
        ;
progName:     ProgNameNull  { $$.code = ProgNameNull; }
        |     ProgName      { $$.code = ProgName(IDENTIFIER); }
        ;
funcName:     FuncNameNull  { $$.code = FuncNameNull; }
        |     FuncName      { $$.code = FuncName(IDENTIFIER); }
        ;

funcDeclList: FuncListNil   { $$.code = FuncListNil; }
        |     FuncListPair  { $$.code =
                              (funcDecl.code :: funcDeclList$2.code); }
        ;
funcDecl :    FuncDecl      { $$.code = FuncDecl(funcName.code,
                                                 type.code,
                                                 block.code); }
        ;
block   :     Block         { $$.code = Block(stmtList.code); }
        ;
stmtList :    StmtListNil   { $$.code = StmtListNil; }
        |     StmtListPair  { $$.code = AppendStHead(stmt.code,
                                                   stmtList$2.code); }
        ;
stmt    :     EmptyStmt     { $$.code = EmptyStmt; }
        |     Assign        { $$.code = Assign(variable.code,
                                             exp.value); }
        |     IfThenElse
                 { $$.code =  IfThenElse(exp.value,
                                     stmt$2.code,
                                     stmt$3.code); }
        |     While
                 { $$.code =      Loop(StToBlck(
                              IfThenElse(exp.value,
                                       stmt$2.code,
                                       Break))); }

        |     Loop          { $$.code = StLstToSt
                                         (Label(block.initialLabel) ::
                                          BlckToSt(block.code) ::
                                          Goto(block.initialLabel) ::
                                          Label(block.finalLabel) ::
                                          StmtListNil); }

        |     Compound      { $$.code = StLstToSt(stmtList.code);}
        |     Return        { $$.code = Return(exp.value); }
        |     Break         { $$.code = ({block.brkCntAllow} == "YES") ?
                                          Goto({block.finalLabel}) :
                                         Break; }
        |     Continue      { $$.code =
                               ({block.brkCntAllow} == "YES") ?
                                  Goto({block.initialLabel}) :
                                  Continue; }
        |     Label         { $$.code = Label(labelName); }
        |     Goto          { $$.code = Goto(labelName);  }
        ;

/*----------------------------------------------------------------------
 * Rules defining error messages in attribute computations.
 *---------------------------------------------------------------------*/

decl   :      Declaration  {
                local STR error;
                error = (variable != VariableNull
                         && NumberOfDecls(variable, {ProgBody.env}) > 1)
```

```
                       ? " { MULTIPLY DECLARED }" : "";
             }
        ;
stmt    :      Assign  {
             local STR assignError;
             local STR error;
             assignError = IncompatibleTypes(variable.type, exp.type)
                              ? " { INCOMPATIBLE TYPES IN := }" : "";
             error = (variable == VariableNull ||
                               IsDeclDeclared(variable, {ProgBody.env}))
                              ? "" : " { NOT DECLARED }";
             }
        |      IfThenElse, While  {
                    local STR typeError;
                    typeError = IncompatibleTypes(exp.type, BoolType)
                              ? " { BOOLEAN EXPRESSION NEEDED }" : "";
             }
        |      Return  {
             local STR error;
             error = ({block.returnAllow} == "YES")?
                       "" : " { RETURN STMT NOT ALLOWED IN THIS CONTEXT } ";
             }
        |      Break, Continue  {
             local STR error;
             error = ({block.brkCntAllow} == "YES")?
                       "" :
                       " { BREAK/CONTINUE STMT MUST BE IN A LOOP STMT } ";
             }
        ;
exp     :      Id  {
             local STR error;
             error = (variable == VariableNull ||
                               IsDeclDeclared(variable, {ProgBody.env}))
                              ? "" : " { NOT DECLARED }";
             }
        |      Equal, NotEqual  {
             local STR error;
             error = IncompatibleTypes(exp$2.type, exp$3.type)
                              ? "{ INCOMPATIBLE TYPES } " : "";
             }
        |      Add, Minus  {
             local STR leftError;
             local STR rightError;
             leftError = IncompatibleTypes(exp$2.type, IntType)
                              ? " { INT EXPRESSION NEEDED }" : "";
             rightError = IncompatibleTypes(exp$3.type, IntType)
                              ? "{ INT EXPRESSION NEEDED } " : "";
             }
        |      After {
             local STR stmtError;
             stmtError = (stmtList != (EmptyStmt :: StmtListNil))
                       ? "" : " { STATEMENT NEEDED }";
             }
        |      FuncCall {
             local STR error;
             error = IsFuncDeclared(funcName, {progBody.newFncDcls})
                       ? "" : " { FUNCTION NOT DECLARED } ";
             }
        ;


/*----------------------------------------------------------------------------
 * Functions that abstract stmt and stmtList construction
 *--- -----------------------------------------------------------------------*/

stmtList StToStLst(stmt st) {
       with (st) (
          Compound(x): x,
```

```
          EmptyStmt  : StmtListNil,
          default    : (st :: StmtListNil)
              )
       };

stmt StLstToSt(stmtList stLst) {
       with (stLst) (
          StmtListNil                       : EmptyStmt,
          StmtListPair(EmptyStmt,aStLst) : StLstToSt(aStLst),
          StmtListPair(aSt,StmtListNil)  : aSt,
          StmtListPair(aSt, aStLst)       : Compound(
                                        AppendStHead(aSt, aStLst))
              )
       };

stmtList AppendStHead(stmt st, stmtList stLst) {
       with (stLst) (
          StmtListNil                        : StToStLst(st),
          StmtListPair(Compound(aStLst),
                      StmtListNil)       : AppendStHead(st, aStLst),
          StmtListPair(aSt, StmtListNil)  : (st :: aSt :: StmtListNil),
          StmtListPair(aSt, aStLst)          : (st :: aSt :: aStLst)
              )
       };

stmtList AppendStTail(stmtList stLst, stmt st) {
       with (stLst) (
          StmtListNil                        : StToStLst(st),
          StmtListPair(Compound(aStLst),
                      StmtListNil)       : AppendStTail(aStLst, st),
          StmtListPair(aSt, StmtListNil)  : (aSt :: st :: StmtListNil),
          StmtListPair(aSt, aStLst)          : (aSt ::
                                       AppendStTail(aStLst, st))
              )
       };


block StToBlck(stmt st) { Block(StToStLst(st)) };

stmt BlckToSt(block blck) {
       with (blck) (Block(stLst): StLstToSt(stLst)) };


stmtList AppendStLst(stmtList stLst1, stmtList stLst2) {
       with (stLst1) (
          StmtListNil                        : stLst2,
          StmtListPair(Compound(aStLst),StmtListNil)
                                       :  AppendStLst(aStLst,stLst2),
          StmtListPair(aSt, StmtListNil)  : (aSt :: stLst2),
          StmtListPair(aSt, aStLst)          :
                    (aSt :: AppendStLst(aStLst,stLst2))
              )
       };

funcDeclList AppendFncDcl(funcDeclList fdLst, funcDecl fd) {
       with (fdLst) (
          FuncListNil : (fd :: FuncListNil),
          FuncListPair(afd, FuncListNil): (afd :: fd :: FuncListNil),
          FuncListPair(afd, afdLst) : (afd :: AppendFncDcl(afdLst, fd))
              )
       };

funcDeclList AppendFncDclLst(funcDeclList fdLst1, funcDeclList fdLst2) {
       with (fdLst1) (
          FuncListNil                       : fdLst2,
          FuncListPair(afd, FuncListNil) : (afd :: fdLst2),
          FuncListPair(afd, afdLst)         :
                    (afd :: AppendFncDclLst(afdLst, fdLst2))
```

```
            )
        };

/*---------------------------------------------------------------------------
/* Function declarations that define the auxiliary functions
/* LookupType, LookupFuncType, IsDeclared, FuncIsDeclared,
/* NumberOfDecls, and IncompatibleTypes.
*--------------------------------------------------------------------------*/

/* Determine the first type bound to i in e, or EmptyType if there is none. */

type LookupDeclType(variable i, declList e) {
        with (e) (
                DeclListNil: EmptyType,
                DeclListPair(Declaration(id, t), dl):
                        (i == id) ? t : LookupDeclType(i, dl)
                )
        };


type LookupFuncType(funcName i, funcDeclList e) {
        with (e) (
                FuncListNil: EmptyType,
                FuncListPair(FuncDecl(id, t, b), dl):
                        (i == id) ? t : LookupFuncType(i, dl)
                )
        };

/*--------------------------------------------------------------------------*/
/* Return true iff there exists a type bound to i in e. */

BOOL IsDeclDeclared(variable i, declList e) {
        with (e) (
                DeclListNil: false,
                DeclListPair(Declaration(id, t), dl):
                        (i == id) ? true : IsDeclDeclared(i, dl)
                )
    };

BOOL IsFuncDeclared(funcName i, funcDeclList e) {
        with (e) (
                FuncListNil: false,
                FuncListPair(FuncDecl(id, t, b), dl):
                        (i == id) ? true : IsFuncDeclared(i, dl)
                )
    };

/*--------------------------------------------------------------------------*/
/* Determine the number of types bound to i in e. */
INT NumberOfDecls(variable i, declList e) {
        with (e) (
                DeclListNil: 0,
                DeclListPair(Declaration(id, *), dl):
                        ((i == id) ? 1 : 0) + NumberOfDecls(i, dl)
                )
        };

/* Determine the number of types bound to i in e. */
INT NumberOfFuncDecls(funcName i, funcDeclList e) {
        with (e) (
                FuncListNil: 0,
                FuncListPair(FuncDecl(id, t, b), dl):
                        ((i == id) ? 1 : 0) + NumberOfFuncDecls(i, dl)
                )
        };

/* Return true iff neither t1 nor t2 is EmptyType and t1 is not equal to t2. */
BOOL IncompatibleTypes(type t1, type t2) {
```

```
        (t1 != EmptyType) && (t2 != EmptyType) && (t1 != t2)
        };
```

11.2 Appendix 2: Scope analysis in Pascal using Eli

This appendix shows how scope analysis is established in a Pascal compiler using Eli. In the general sense, it consists of the instantiation of generic modules and the creation of the appropriate set of symbols, which inherits behaviours from these modules.

```
~O~<scope.con~>~{ StandardBlock: Program . ~}

~O~<scope.lido~>~{
ATTR Key: DefTableKey;
 SYMBOL Block INHERITS RangeScope END;

ATTR Sym: int;
SYMBOL NameOccurrence COMPUTE SYNT.Sym=TERM; END;

SYMBOL NameDef INHERITS IdDefScope, NameOccurrence END;
SYMBOL NameUse INHERITS IdUseEnv, NameOccurrence END;

SYMBOL Block INHERITS RangeUnique END;
SYMBOL NameDef INHERITS Unique END;

SYMBOL NameUse COMPUTE
  IF(EQ(THIS.Key,NoKey),
    message(ERROR,"Undefined identifier",0,COORDREF));
END;

SYMBOL StandardBlock: Env: Environment;

SYMBOL StandardBlock INHERITS RootScope COMPUTE
  SYNT.Env = StandardEnv(NewEnv());
END;

SYMBOL Program INHERITS Block END;
SYMBOL ProcedureBlock INHERITS Block END;
SYMBOL FunctionBlock INHERITS Block END;

SYMBOL ConstantNameDef INHERITS NameDef END;
SYMBOL TypeNameDef INHERITS NameDef END;
SYMBOL VariableNameDef INHERITS NameDef END;
SYMBOL ProcedureNameDef INHERITS NameDef END;
SYMBOL FunctionNameDef INHERITS NameDef END;
SYMBOL ParameterNameDef INHERITS NameDef END;

SYMBOL ConstantNameUse INHERITS NameUse END;
SYMBOL TypeNameUse INHERITS NameUse END;
SYMBOL VariableNameUse INHERITS NameUse END;
SYMBOL ProcedureNameUse INHERITS NameUse END;
SYMBOL FunctionNameUse INHERITS NameUse END;
SYMBOL ParameterNameUse INHERITS NameUse END;
~}

~O~<scope.specs~>~{
$/Name/AlgScope.gnrc :inst
$/Prop/Unique.gnrc :inst
~}
```